# OOPM

# INDEX

NAME. : PRANEET BOMMA | Std. : | Div. : | Roll No. : 10

# INDEX

NAME. : Std. : Div. : Roll No. :

# JAVA

* Printing

Use System.out.println(" ");

→ This is because println method is stored in System.out directory.

System.out.print(" ");

→ This keeps the cursor on the same line.

→ The above one goes to next line.

* Primitive Datatypes.

| Keyword | Size |
|---|---|
| char | 2 bytes - 16 bits |
| byte | 1 byte - 8 bits |
| short | 2 bytes - 16 bits |
| int | 4 bytes - 32 bits |
| long | 8 bytes - 64 bits |
| float | 4 bytes - 32 bits |
| double | 8 bytes - 64 bits |

→ Double is ~~a~~ taken by default by Java. Hence even after ~~a~~ initializing it as float, typecast to float.

P.T.O.

Page No.
Date

→ for - float n = 3.2, m = 2.2

The output for addition will be
result = 5.400000 ---etc.

→ So by typecasting like this

float n = 3.2(f), m = 2.2f

The output for addition will be
result = 5.4

→ single characters should be initialized in single quote.

eg:
'A', 'B', 'C', 'a', 'b', 'c' etc.

→ String should be in double quotes.

eg: "Praneet", "Bomma".

* Typecasting

eg: char c = 'A';

Typecasting.

∴ System.out.println ((int) c);

Output - 65.

eg: System.out.println ((char) 67);

Output - C

⟹ Comment line.

→ // Single Line Comment

→ /* Multi
Line
Comment */

* Printing Value of Variables.

```
int i = 5, j = 2, k = i+j;
System.out.println(" Addition = 7");
```

→ To print using variable

```
System.out.println(" Addition of " + i +
                   " and " + j + " is " + k);
```

→ Here + is used as concatenation.

→ The second + after i and j are also used for concatenation.

→ It can be printed using printf like C Programming.

```
System.out.printf(" Addition of %d and %d
                  is %d ", i, j, k);
```

→ The difference between println and printf is - println return type is void whereas printf return type is PrintStream.

## * Type Casting

eg: byte b = 8;

b = b * 2.5; // ——— (1)

b *= 2.5; // ——— (2)

→ Here (1) gives out error because float value cannot be stored in byte. As float value holds larger space.

→ whereas (2) works fine. This is because in (2) the ans that is calculated is automatically calculated for datatype of (b) and not double.

→ b = (double) b * 2.5;
This can't be done because a larger datatype cannot be fit into smaller datatype.

⟹ To convert binary to ~~inte~~ decimal value.

eg: int i = 0B100_00_00_00_00;
S.O.P (i);
Output: 1024

→ To define its binary (zero B) or (zero b) should be written before binary number.
→ Underscore can be used. (New feature in Java)

## * Post & Pre Increment

eg:
```
int i = 5;
j = i++;

S.O.P(j);
```

→ Here j should give 6 — but output is 5.

→ This is because it works like this

```
int i, temp;

temp = i;
i++;
i = temp
```
} = i = i++;

→ Even if this shows the output should be 6. but the output is 5 because i again takes the previous value of i and not the incremented value.

## * Bitwise Operators.

→ & = AND Bitwise Operator

→ | = OR Bitwise Operator

This works on bits
i.e only 1 & 1 = 1
and only 0 | 0 = 0

→ Left shift Operator = <<

→ Right shift Operator = >>

Page No.
Date

* if-else

'same as C'

```
if (condition)
{ body }
else
{ body }
```

* Ternary Operator

Syntax

```
(condition)? true body : false body;
```

☆

```
P S V M (_ _)
{ Object obj1;

  obj1 = true? new Integer(10): new
                                Double(15.0);
}
```

Here the output is obj1 = 10.0

⟹ This is because in ternary operator it checks the type of true as well as false and gets converted into part that is larger.
In this case Double is larger datatype than Integer. Hence Integer gets converted from Integer value to Double value i.e 10 to 10.0.

* Printing Hello without semicolon (;)

```
if (System.out.printf("Hello")==null)
```

Output : Hellonull

→ This is because printf in java returns printstream. It is obviously null here, but Hello gets printed.

* Switch

Note: ☆ Switch in Java supports string.

Eg:

```
String s = "Hello";

switch (s)
{
    case "Hello":
        System.out.println("Hi");
        break;
}
```

Output : Hi

* Loops

while, do while, for loop "Same as C"!

## * Labelled break Statement

```
int i,j;
praneet:          // Here : signifies labelling.
for(i=1;i<=4;i++)   // This for loop is labelled as praneet.
{  for(j=1;j<=4;j++)
   {
      if(i==3)
         break praneet;   //labelled break
      Print(" * ");
   }
   println("");
}
```

Output: * * * *
* * * *

This is because when i==3 the whole label is broken or stopped i.e outer loop is broken, hence only two rows of * * * * is printed.

## * Print ASCII Values

```
int i;
for(i=0;i<=127;i++)
{
   System.out.printf("%d : %c \n",i,i);
}
```

This will print all the ASCII values line by line.

## * PATTERNS

```
*   *   *   *
*           *
*           *
*   *   *   *
```

Print this

```
for (i = 1; i <= 4 ; i++)
{
    for (j = 1; j <= 4 ; j++)
    {
        if ((i == 1) || (i == 4) || (j == 1) || (j == 4))
        {
            SOP (" * ");
        }
        else
        {   SOP ("  ");
        }
    }
        println ( );
}
```

(2)

```
1   2   3   4
2   3   4   1
3   4   1   2
4   1   2   3
```

Print this

```
int i, j, k, l;
for (i = 1 ; i <= 4 ; i++)
{
    k = i;
    l = 1;

    // P.T.O
```

```
for(j=1;j<=4;j++)
{
    if(k<=n)
    {
        soPrint(k+" ");
        k++;
    }
    else
    {
        soPrint(l+" ");
        l++;
    }
}
soPrintln();
}
```

(3) Print

```
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
```

```
for(i=1;i<=n;i++)
{   k=1, l=0;
  if(i%2==0)
   for(j=1;j<=i;j++)
   {
      sop(l);
      l++
   }
else
   for(m=1;m<=i;m++)
      sop(k);
```

```
for (i=1; i<=n; i++)
{
    for (j=1; j<=i; j++)
    {
        if ((i+j)%2 == 0)      // Even prints 1
        SOP("1");
        else                   // Odd prints 0
        SOP("0");
```

* INPUT

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

i.e `Scanner variable = new Scanner(System.in);`

⟹ Perfect Number

eg 6 - factors of 6 = 1, 2, 3

∴ 1 + 2 + 3 = 6

∴ 6 is a perfect number.

28 is a perfect number.

⟹ Most efficient way of swapping two numbers.

a = 5, b = 4

```
a = a^b   // 101 ^ 100 ⟶ 001
b = a^b   // 001 ^ 100 ⟶ 101 ⟶ 5
a = a^b   // 001 ^ 101 ⟶ 100 ⟶ 4
```

∴ a = 4, b = 5 Swapped.

This is EXOR (^)

Page No.
Date

⟹ Another efficient way to swap

a = 5, b = 6

∴ b = a + b - (a = b);
b = 5 + 6 - 6

∴ Here first a is assigned value of b i.e 6 and then b is assigned the result i.e 5.

* ARRAY.

Syntax

int a[] = new int[5];
↓ ↓ ↓
datatype variable[] = new datatype[size];

* Enhanced for loop

Syntax:

for (int i : array_name)
↓ ↓ ↓
datatype variable array_name.

eg: for (int i : a)
SOP(i);

Note: It is used only if you want to access all the elements of array.

## * Random :

Random is a class that belongs to java.util package.

Syntax

```
Random variable = new Random();
```

In variable.nextInt() method there is something called as bound. i.e it will take value ~~with when it~~ which is under bound.

Syntax

```
= variable.nextInt(int bound)
```

eg `variable.nextInt(50)`

it will take value only less than 50.

## * Enhanced for loop - 2D Array

```
int p[][] = { {1,2,3,4},
              {5,6,7,8},
              {7,8,9,1}
            };

for (int i[] : p)
{
    for (int j : i)
    {
        SOP(j);
    }
}
```

Page No.
Date

## * Jagged Array

```
p[][] = {
            {5,6,7,8},
            {1,2},
            {3,4,5}
        };
```

```
for(int i[] : p)
    for(int j : i)
        SOP(j)
```

} works. ✓

But

```
int k[][] = new int[3][ ]
```

← ?

→ This is where the problem arises.

∴ This can be solved by:

∴

```
k[0] = new int[4];
k[1] = new int[2];
k[2] = new int[3];
```

## * 3D (Multi-Dimensional Array)

```
int c[][][] = new int[2][2][2]
```

i.e 2 x 2 x 2 = 8 values.

→ 3D Array can be imagined as a 'Rubik's Cube'.

⟹ Can be used by two ways:
i.e Normal for loops or Enhanced for loops

① 
```
for (int i=0 ; i<2 ; i++)
    for (int j=0 ; j<2 ; j++)
        for (int k=0 ; k<2 ; k++)
            SOP (c[i][j][k]);
```

② 
```
for (int i[][] : c)
    for (int j[] : i)
        for (int k : j)
            SOP (c[i][j][k]);
```

Works same.

* Varargs - (Variable arguments)

```
1  class Varargs
2  {
3      public static void main (String args[])
4      {
5          Display obj = new Display();
6          ~~obj =~~ obj.show(
7      }
8  }
9  class Display
10 {   public void show (int ... a)
11     // 3 dots - Varargs
12     for(int i : a)
13         SOP (i);
14 }
15
```

(6)

⟹ Here in line 6, doesn't matter how many elements you pass, it will take the elements in an array a[] - line 10.
This is due to new feature in java called Varargs.

Note: 3 dots (...) are important - part of syntax.

Now if method overloading is done in class Display - line 9.
and only one element is passed in
obj.show(1); - line 6

i.e obj.show(1);

```
} }
class Display
{   public void show(int ... a)
    { for(int i : a)
      sop(i); }
    public void show(int a)
    {
      sop(a + "This is second method");
    }
}
```

Here the output will be:
1 This is second method

⭐ This is because when we pass an element it checks for method with exact parameter. Hence second method is called.
In absence of second method, the first method will work too.

* Different ways of writing Main method.

(1) public static void main (String ... args)
Here it is a Varargs

(2) static public void main (String ... args)

(3) public static void main (String[] args)

(4) public static void main (String a[])

⟹ These are all the changes you can make.

⟶ The changes you cannot make are

☆ You cannot change String to Integer or anything.

* Class & Objects & Methods

⟹ Syntax

⟶ Class

class Name

⟶ Method

↱ references
void Add ()
return_type ↳ parameters.

⟶ objects.

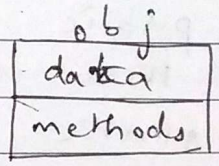
↱ Reference ↱ Constructor
Add variable = new Add ()

Page No.
Date

## * Objects

A obj = new A();

This creates




obj = new A(); This breaks the link to previous object. The previous object is dumped into garbage collection.

→ Another way to send object to garbage coll is
obj = null;
This sends it to garbage collection.

## * Constructor

public static void main(String args[])

Q] Why is the parameters array String?

→ Because String can take any ~~time~~ type of character.

Q] Why is main method static?

→ Main is the first method that is accessed in a program. To call a method, we need to first create an object of it.

☆ As the main method is the first method, object for it cannot be created. Hence main is always static. It can be called without using an object.

# static methods can be called without an object.

(1) Constructor is a member method.
(2) It has same name as class name.
(3) It is used to allocate memory to the object.
(4) It doesn't return anything.

A obj = new A(); this is the constructor.

(5) Even if you don't create a constructor, there is constructor created by default by JVM.
This is called default constructor.

(6) Because of default constructor, the default values of int, float initialized are 0 & 0.0.

⟹ Constructor Overloading

```
public A()
  { }
public A(int k)
  { }
```

We can have multiple constructors, provided they have different parameters.

```
class A()
{
    int i;
    public A()
    {
        SOP ("Hi");
    }
    public A (int k)
    {
        SOP ("Hello");
    }
}
```

→ if we create object
A obj = new A();

→ Output will be Hi

→ This is because we called without parameters i.e first constructor was called.

→ if we create object
A obj = new A(5);

→ Output will be "Hello"

→ This is because we called with a parameter i.e second constructor was called.

~~* Java~~

* Access Modifiers

⟹ Java has Access Modifiers & not specifiers.

Types of Modifiers.

1. Abstract
2. Final
3. Public
4. Private
5. Protected

→ * Classes cannot be private or protected.

→ Only inner classes can be private.

⟹ To access the class outside the package specify public

→ without public - class cannot be accessed outside the package.

i.e class A // Cannot be accessed outside package

public class A // Can be accessed outside package.

Page No.
Date

⟹ Even for variables i-e (int i) etc

To access them ~~of~~ outside the package they should be specified as public.

i-e public int i; //Accessible outside package.

⟹ Private : Specific Class

Default : Specific Package

Public : Any Class or Package

Protected : Subsiding Class

# When a *.java file is compiled it creates a *.class file. This *.class file is the bytecode.

# ~~Java~~ *.class can be used to obtain *.java file.

Note: Only the structure is obtained, entire code cannot be obtained.

This is done by

→ Command

javap file_name

## * Static Methods & Static Variables

We make a method static to call it without using an object.

```
{   p s vM ( )
    {  Abc.i = 5;
       Abc.show();
    }
}
```

This is used to access static variable

This is used to access static method.

```
class Abc
{
    static int i;
    public static void show()
    {
        sop (i);
    }
}
```

Note: Non-static ~~method~~ variable cannot be accessed through a static method.

i.e if it is int i (not static int i) then sop will give error.
Because show() method is static method.

Syntax to call ~~Meth~~ static Method or variable.

→ Variable - Class_Name . variable ;

→ Method - Class_Name . method_name ( ) ;

Page No.
Date

* To count number of objects made.

Every time you create an object, the instance variable is copied to it with its original value.

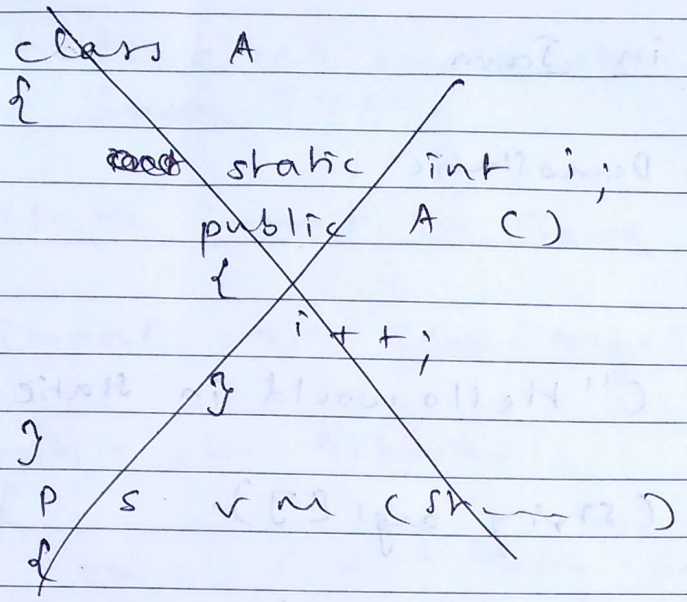
Eg:

```
class A
{ int i = 0;
  public A()
  { i++;
  }
}
PSVM( )
{ A obj1 = new A();
  A obj2 = new A();
}
```

→ in this obj1 takes int i as 0 as well as obj2 takes int i as 0.

i.e it always takes the initial value Hence the objects cannot be counted with this. Because everytime you make an object it will be i = 1.

⟹ To overcome this change int i to static int i;



```
class A
{
    static int i;
    public A ()
    {
        i++;
    }
}
P s v m (Sh---- )
{
```

```
class A
{
    static int i;
    public A ()
    {
        i++;
    }
    public void counter ()
    {
        SOP (i);
    }
}
P s v m ( )
{
    A obj1 = new A ();
    A obj2 = new A ();
    obj1.counter;
    obj2.counter;
}
```

} Constructor

**Output :** 2
2

This is because two objects were made and then i was accessed.

Page No.
Date

* To count number of objects made.

Every time you create an object, the instance variable is copied to it with its original value.

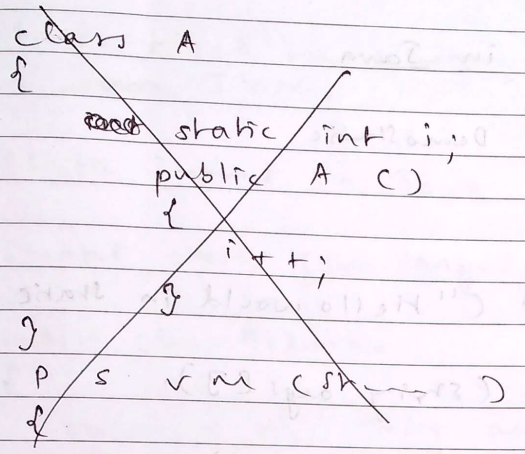
Eg:

```
class A
{ int i = 0;
    public A()
    { i++;
    }
}
PSVM( )
{ A obj1 = new A();
  A obj2 = new A();
}
```

→ in this obj1 takes int i as 0 as well as obj2 takes int i as 0.

i.e it always takes the initial value Hence the objects cannot be counted with this. Because everytime you make an object it will be i = 1.

⟹ To overcome this change int i to static int i;



```
class A
{
    static int i;
    public A ()
    {
        i++;
    }
}
p s v m (st---- )
{
```

```
class A
{
    static int i;
    public A ()
    {
        i++;
    }
    public void counter ()
    {
        SOP (i);
    }
}
p s v m ( )
{
    A obj1 = new A ();
    A obj2 = new A ();
    obj1.counter;
    obj2.counter;
}
```

} Constructor

**Output :** 2
2

This is because two objects were made and then i was accessed.

Page No.
Date

## * Static Block in Java

```
public class DemoStatic
{
   static
   {
     S O P ("Hello world in static 1");
   }
   P S VM (String args[]).
   {
     S O P ("In Main");
   }
   static
   {
     SOP (" Bye in static 2 ");
   }
}
```

Output:

```
Hello world in static 1
Bye in static 2
In Main.
```

⟹ static block runs before the main ~~fun~~ method.
So even if you write static block after main block, still static runs first.

⟶ In case of multiple static blocks - it runs in sequence.
i.e first static then second static.

→ Static block runs when class is loaded in ~~Java~~ JVM.

* Static import in Java

```
import static java.lang.System.out;

public class Telusko                    // out is static type
{                                          reference
    public static void main(String args[])
    {
        out.println("Hello");
    }
}
```

Output: Hello.

This is how static import is used.

* ENCAPSULATION

```
class A
{
    private int i, j;
    public void setI(int j)
    {
        i = j;
    }
    public int getI()
    {
        return i;
    }
}
```

P.T.O (Main)

Page No.
Date

```
class MainClass
{
    P s v M ()
    {
        A obj = new A();
        obj.setI(5);
        sop(getI());
    }
}
```

⟹ Hence here variables are Encapsulated by setting them private.

∴ They can only be accessed ~~by a~~ through method ~~and obj~~ by object.

If obj.i = 5; ⟶ This will give error.

This is because i is private.

## * INHERITANCE

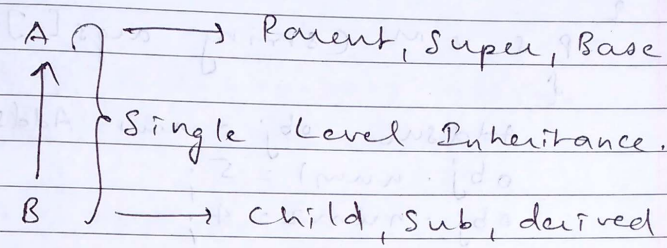
Syntax:

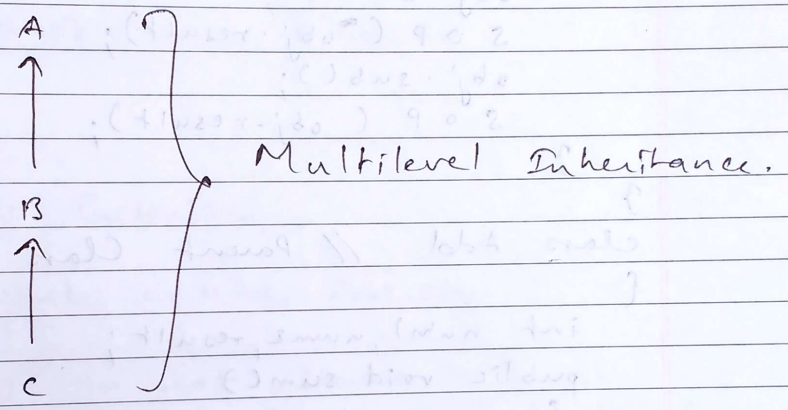
class subclass extends superclass

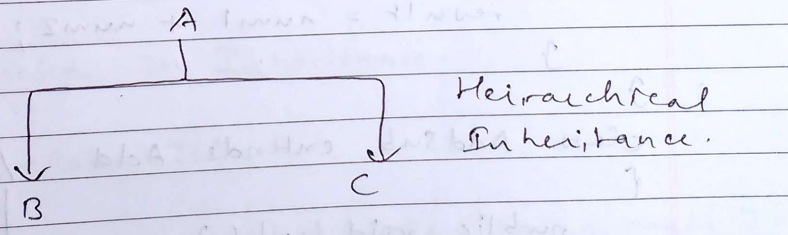
eg: class B extends A

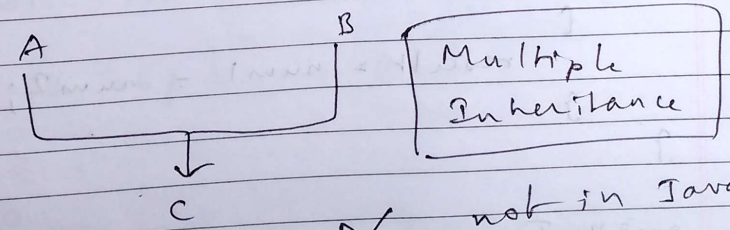
Keyword.

## ⟹ Types of Inheritance




✗ not in Java.

```
class Main
{
   P S V M (String args[])
    {
      AddSub obj = new AddSub();
      obj.num1 = 5;
      obj.num2 = 4;
      obj.sum();
      S O P ("obj.result);
      obj.sub();
      S O P (obj.result);
    }
}
class Add // Parent Class
{
   int num1, num2, result;
   public void sum()
    {
      result = num1 + num2;
    }
}
class AddSub extends Add // Single Level Inheritance
{                        //                 |
   public void sub()     //                 -> Child Class
    {
      result = num1 - num2;
    }
}
```

OUTPUT: 9

# Previous Program Continued.

```
class AddSubMul extends AddSub
    // Multi-level Inheritance.
{
    public void multi()
    {
        result = num1 * num2;
    }
}
```

# Multiple Inheritance

class Calc extends Add, Div

✗ This is not allowed in Java.

→ We can overcome this problem by Interface.

## ☆* Constructor in Inheritance.

```
1  public class Teluko
2  {
3      public static void main (String args[])
4      {
5          B obj = new B();
6      }
7  }
```

P.T.O

Page No.
Date

```
8  class A
9  {
10     public A()
11     {
12        System.out.println("In A const");
13     }
14     public A(int i)
15     {
16        System.out.println("In A const Int");
17     }
18 }
19 class B extends A
20 {
21     public B()
22     { // super(); Hidden code.
23        S O P("In B const");
24     }
25     public B(int i)
26     {
27        S O P("In B const Int");
28     }
29 }
```

According to line 5: when B constructor is called the output is

```
In A const
In B const
```

⟹ This is because of Inheritance

→ whenever you call subclass constructor it first goes to the superclass default constructor and then goes to the subclass constructor.

⟹ If on line 5 we write as

```
B obj = new B(5);
```

the output is

```
In A const
In B const Int
```

⟹ This is because only the default constructor of super class is called.

Hence Output is not

In A const Int ✗.

* Method Overloading OR Compile Time Polymorphism

→ Same Method Names but with different Parameters is called Method Overloading.

Eg:
```
public void Add()
public void Add(5, 3)
public void Add(3.3, 2.1)
```

+ Same Names but different Parameters.

Page No.
Date

## * Method Overriding OR Runtime Polymorphism

```
class A
{
    public void show()
    { SOP("In A");
    }
}
class B extends A
{
    public void show()
    {
        SOP("In B");
    }
}
class Main
{
    P S V m(String args[]).
    {
        B obj = new B();
        obj.show();
    }
}
```

Output : In B

⟹ This is because of Method Overriding.

Method of class B has overriden method of class A.

Note: Both methods have same name.

⟹ Method Overriding is
methods with same name and parameters but belong to different classes.

* Polymorphism

Poly = many
morph = behaviour.

⟹ Polymorphism is the combination of Method Overloading & Method Overriding.

⟹ Dynamic Method Dispatch is also called as Run time Polymorphism.

* Super keyword:

Syntax

super(parameters);

It is used to call super class.

* Use of Final Keyword

~~final int i;~~ final int i=5;
~~i=~~ i++ ⟶ This will give Error.

Because we cannot assign a value to a final variable.

```
final int i;
i = 6   ⟶ This will be ok.
X  i = 7   ⟶ This will give error.
```

Because once assigned we cannot change its value again.

⟹ Final Keyword with Method.

```
class A
{
    final public void show()
    {
        SOP("In A show");
    }
}
class B extends A
{
X ⟶ public void show()
    {
    }
}
```

This will give error because we cannot override a final method.

Here due to final keyword, void show is final method.

⟹ Final Keyword with Class.

~~class A~~

```
final class A
{

}
```

X → class B extends A

This will give error because, no class can extend a final class.

* Types of Variables

(1) Instance Variable.

A Variable that is defined in a class but outside method.

(2) Local Variable

A Variable that is defined inside a method.

(3) Class Variable.

A Variable that is a static variable.

## * This Keyword

```
1  public class ThisKey
2  {
3      p s v m(string args[])
4      {
5          A obj = new A(6);
6          obj.show();
7      }
8  }
9  class A
10 {
11     private int x;      // Instance Variable
12     public A(int x)     // Local Variable
13     {
14         this.x = x;
15     }
16     public void show()
17     {
18         S O P("x is " + x);
19     }
20 }
```

OUTPUT: x is 6

Here if in line 14 - this keyword is not used.
Output will be: x is 0

→ This is because 6 is assigned to local variable
In SOP - Instance Variable is printed
i.e Instance Variable remains with default value
i.e 0.

* Anonymous Object

primitive variable.

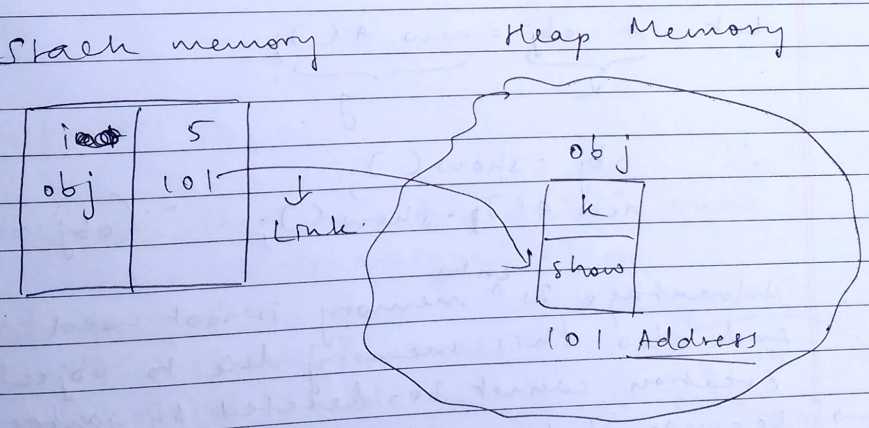
```
int i = 5;

A obj = new A();
```

Reference variable

```
obj.show();

class A
{
  int k;
  show()
  {
    sop(k);
  }
}
```




⟹ This shows that obj occupies memory in stack as well as heap memory.

## * Anonymous Object

primitive variable.

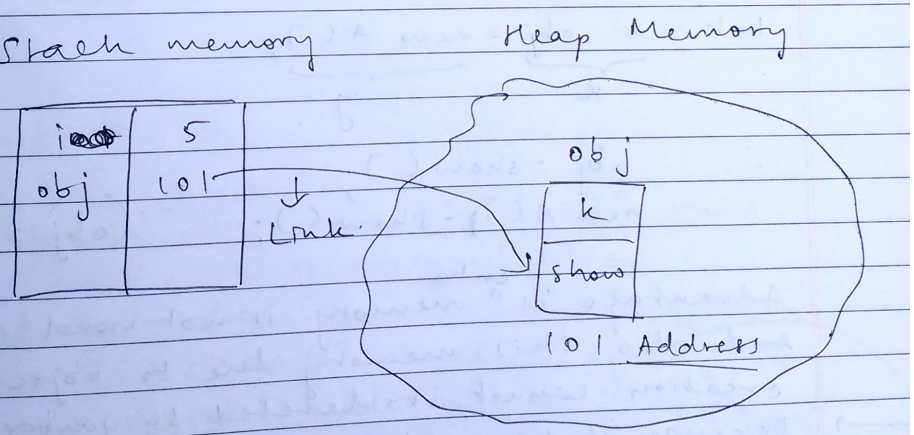
```
int i = 5;

A obj = new A();
```

Reference variable

```
obj.show();

class A
{
   int k;
   show()
   {
     SOP(k);
   }
}
```




⟹ This shows that obj occupies memory in stack as well as heap memory.

Page No.
Date

If we write

~~obj.show~~

```
A obj = new A();
  obj.show();
```

} as

```
new A().show();
```

This will take memory only in heap memory and not in stack memory. This is because no reference variable is assigned.

```
obj.show();
```

can be written as

```
new A().show();
```

because.

Proof!

Let `A obj = new A();`

(A obj = x; new A() = y;)

∴ `obj.show();`
`new A().show();` ∵ (obj : y)

∴ Advantage is ^extra memory is not used.
And also this memory due to object creation cannot be deleted by garbage collection
→ Because it has a link to stack memory and can be used any time in the program.

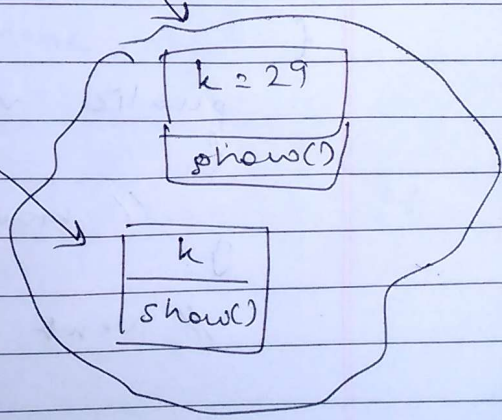
## # Disadvantage

This cannot be done when you want to use it multiple times.

eg:
```
class A
{
    int k;
    show()
    {
        sop(k);
    }
}
Main
{
    new A().k = 29;
    new A().show();
}
```

Output: 0

Because when new A().k = 29; runs
k is assigned 29.
And again when
new A().show(); runs
a new object is made.
∴ The value is default value
i.e 0.




∴ Hence use only if you want to use object only one.

Note: Don't implement when you want to use multiple times.

Page No.
Date

* Abstract

⟹ Abstract Methods are methods that can only be declared, not defined.

```
public abstract void abc();
```

→ Abstract Methods can only be written under Abstract Class.

→ Abstract Class is a class of which you ~~cannot~~ cannot create an object.

```
public abstract class Mahesh
{
    public void call()
    {
        // call code here;
    }
    public abstract void move();
    public abstract void dance();
    public abstract void microwave();
}
abstract class Ramesh extends Mahesh
{
    public void move()
    {
        // move code here;
    }
```

// Next Page continued

```
    public void dance()
    {
        // dance code here
    }
    public abstract void microwave();
}
class Suresh extends Ramesh
{
    public void microwave()
    {
        // microwave code here;
    }
}
```

This is called concrete class.

→ Because it implements the abstract method declared in subclass.

* How Abstraction is used.

```
public static void main(String args[])
{
    show(new Samsung());
}
public static void show(Phone obj)
{
    obj.showConfig();
}
```

// Continued on next page.

Page No.
Date

```
abstract class Phone
{
    public abstract void showconfig();
}
class IPhone extends Phone
{
    public void showConfig()
    {
        SOP("2 GB, IOS 9");
    }
}
class Samsung extends Phone
{
    public void showConfig()
    {
        SOP("Octa Core, Lollipop");
    }
}
```

⟹ With the help of abstraction class Phone, we can easily call configuration of both phones.

→ If showConfig of IPhone is called it will give output : 2 GB, IOS 9

→ If showConfig of Samsung is called with method overriding it will give output
Octa Core, Lollipop.

## Another Example next Page.

```
1  public clas AbDemo
2  {
3      public static void main(String a[])
4      {
5          Iphone obj = new Iphone();
6          Samsung obj1 = new Samsung();
7          show(obj);
8      }
9      public static void show(Phone obj)
10     {
11         obj.showConfig();
12     }
13 }
14 public abstract class Phone
15 {
16     public abstract void showConfig();
17 }
18 class Iphone extends Phone
19 {
20     public void showConfig()
21     {
22         S O P("2 Gb, IOs 9.3");
23     }
24 }
25 class ~~I~~Samsung extends Phone
26 {
27     public void showConfig()
28     {
29         S O P("2 Gb, Lollipop");
30     }
31 }
```

Page No.
Date

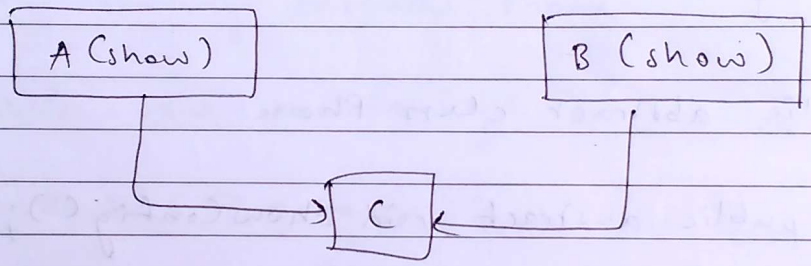
→ Here on line 7 as we pass obj the output is

2Gb, IOS 9.3

→ If we pass obj1 on line 7 the output will be

2Gb, Lollipop

* WHY NOT MULTIPLE INHERITANCE ?




```
C obj = new C();
obj.show();
```

→ Here C gets into a dilemma whether to choose show from A or B.

→ This is called Ambiguity.

→ Hence Multiple Inheritance is not supported in Java.

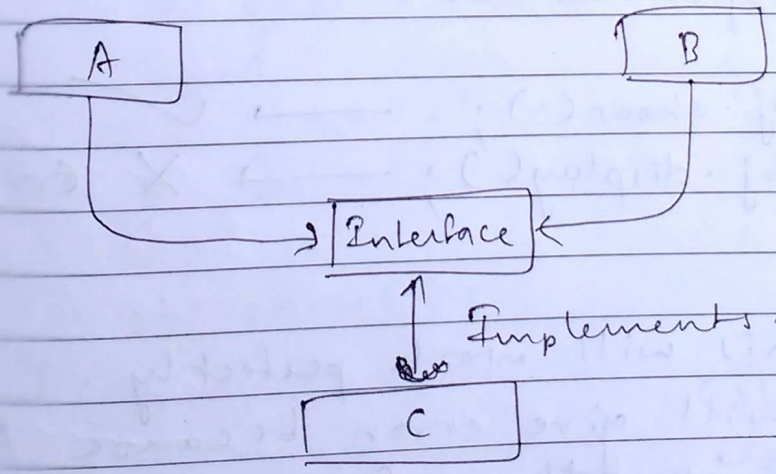
# INTERFACE

| ABSTRACT CLASS | INTERFACE |
|---|---|
| It is a class | It is interface |
| Can declare and define a method | Can only declare a method not define |
| Cannot create object of abstract class | Cannot create object of interface. |

When you go for class and class we use extends

When you go for class & interface we use implements.

Here, a class implements an interface.

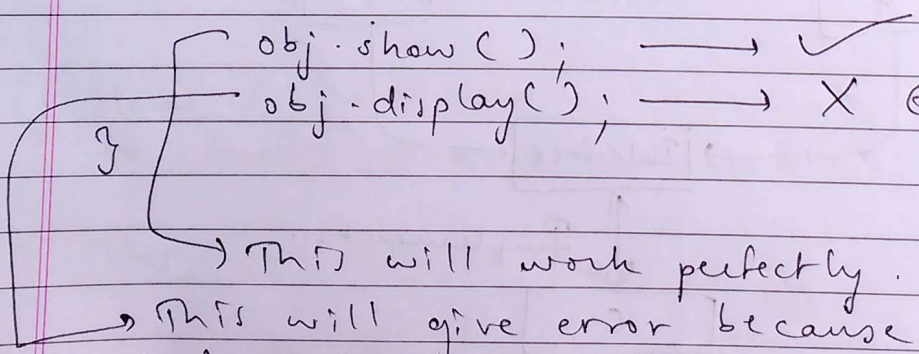



Syntax

Interface name (Just like class)

Page No.
Date

```
interface A
{
    public void show();
}
class B implements A
{
    public void show()
    {
        sop("Hello");
    }
    public void display()
    {
        sop("Hi");
    }
}
Main()
{
    A obj = new B()
```



⟹ Every method in interface is by default public abstract.
Hence we need not mention it.

# Using interface through object by implementing inner class.

```
interface Abc
{
    void show();
}
public class Telusko
{
    public static void main(String args[])
    {
        Abc obj = new Abc()
        {
            public void show()
            {
                SOP("in show");
            }
        };
        obj.show();
    }
}
```

(This is inner class.

Hence we use interface through object.

Page No.
Date

## * LAMBDA EXPRESSIONS in JAVA (Only 1.8 Version)

```
interface Abc
{
    void show();
}
public class Teluisko
{
    public static void main(String args[])
    {
        Abc obj = () -> SOP("in show");
        obj.show();
    }
}
```

Lambda Expression

## * INTERFACE IN JAVA 8 (1.8)

→ A method cannot be defined in an interface.
→ It can only be declared.

But in JAVA 8 (1.8) we are allowed to define a method in JAVA.

Syntax

```
default void show()
{
    body
}
```

* default is necessary. This is how we define method in interface in JAVA 8.

eg:

```
interface I
{
    void add();
    default void show()
    {
        SOP("I");
    }
}
interface J
{
    SOP("J");
}
class A
{
    public void show()
    {
        SOP("A");
    }
}
class C entends A implements I
{
    C obj = new C();
    obj.show();
}
```

→ this in Main

⟹ Without implementing interface
Output will be A

⟹ With implementing interface
Output will be A and not X I

Because class has preference over interface.
default will have low priority.

Page No.
Date

⟹ In Java 8

static methods are allowed in interface.

Syntax :

```
static void show()
{
    SOP("Hi");
}
```

Note: default not needed when static.

## * MARKER INTERFACE

→ Interface that are empty i.e without any methods are called marker interfaces.

→ Used for Permission.

~~interface~~

```
interface P
{
}
class Demo implements P
{
    void show()
    {
        SOP("Hello");
    }
}
// P.T.O Main.
```

```
ps vm (String args[])
{
    Demo obj = new Demo();
    if (obj instance of P)
        obj.show();
    else
        SOP("No Print");
}
```

→ instance of is a keyword.
when you create an object you also create an instance of the interface it is implementing.
Hence if condition is true.
Output : Hello.

## * STRING OPERATIONS IN JAVA

### (1) CONCATENATION

→ s1.concat(s2); s1 string concatenates with s2
→ s1 + s2; —"—

### (2) Length of string

s1.length();

### (3) UpperCase, LowerCase

s1.toUpperCase(); s1.toLowerCase();

Page No.
Date

## BUFFERED READER INPUT

Syntax:

```
InputStreamReader a = new InputStreamReader(System.in);
BufferedReader b = new BufferedReader(a);
c = b.readLine();
```

(a: variable; b: variable; c: variable.)

→ If you are taking input only once then you can simplify by not creating object of InputStream...

~~BufferedReader b = new BufferedReader~~

```
BufferedReader b = new BufferedReader(new
                   InputStreamReader(System.in));
```

→ This is done when you want to take only one input.

• BufferedReader & InputStreamReader belong to java.io

## * Scanner Input

→ ~~This belo~~ Scanner class belongs to java.util (package).

Syntax

variable
↓
Scanner sc = new Scanner(System.in);

```
int i = sc.nextInt();
String s = sc.nextLine();
```

## * Wrapper

⟹ Primitive Datatypes

int i; , float f;

⟹ Classes

Integer iobj = new Integer();

Integer is a wrapper class

Similarly there are classes for Double, Float, etc.

⟹ Wrapping

int i = 5;

Integer iobj = new Integer(i);

passing a variable to the object is called wrapping.

Date

⟹ **Unwrapping**

```
int j = iobj.intValue();
```

This gives j = 5

This is a method that gives value.

⟹ **Boxing**

```
int i = 5
Integer ii = new Integer(i);
```

This is called Boxing

⟹ **Auto Boxing**

```
Integer jj = i;
```

This is called Autoboxing (i.e) it gives value directly to the object

⟹ **UnBoxing**

```
int j = jj.intValue();
```

This is Unboxing

⟹ **AutoUnboxing**

```
int k = jj;
```

This is Auto Unboxing (for same reason)

Note: Processing of int is faster than Integer.

→ Integer or Wrapper Classes are used for frameworks.

→ Because frameworks run on Objects only.

* String Splitting from (CSV)

→ CSV = Comma Seperated Values.

→ String str = "Praneet, Chetan, Kashyap, Saurabh";

SOP (str);

This gives ~~value~~ output

Praneet, Chetan, Kashyap, Saurabh

⟹ String splitting function (method)

str.split(" ");

↳ Here it asks for the seperator from where it needs to split. (You can use any)

String str = "Praneet, Akshay, Smriti, Gourav";
String names[] = str.split(" ");
SOP (names[1]);

↳ index of array.

Output : Akshay

Note:: Array is necessary to store the splitted strings.

## * Exception Handling

Here main class is Throwable class

Subclasses of Throwable class are

- Exception class
- Error class
  - This can be handled
  - cannot be handled.

Two types of Exceptions

→ Checked & Unchecked Exceptions.

→ Checked Exceptions include

IOException
SQLException and other such exceptions

→ Unchecked Exceptions include

RuntimeException
this includes FileNotFoundException
ArrayIndexOutOfBoundException
and many more Exceptions.

Syntax:

```
try
{
    // Critical Statement
}
catch
{
    // Changes you want to make in case of error.
}
finally
{
    // default - This runs any how (Optional)
}
```

→

```
psvm (---)
{
    int i, j, k = 0;
    i = 5;
    j = 0;
    try
    {
        k = i / j;   // Unchecked Exception.
    }
    catch (Exception e)   → Can be any variable
    {
        sop("Cannot divide by zero");
    }
}
```

Page No.
Date

```
try
{
    k = i/j;

    for(int a=0; a<3; a++)
    {
        SOP(a);
    }
}
catch
{
    SOP("Cannot divide by zero");
}
```

→ Here for loop is not executed because after getting an Exception for k = i/j; It directly jumps to catch.

⟹ **Multiple Catches**

```
try
{
    // Critical Code (Statement)
}
catch (ArithmeticException e)
{
}
catch (ArrayIndexOutOfBoundsException e)
{
}
```

→ If you don't know the Exception
Use Common Exception Class

```
catch (Exception e)
{
}
```

☆ → If you use above catch (Arithmetic Except
then will execute be cause
it executes with preference no.

→ Hence always write Exception catch
at the end of all Exception (Specific) Catches.

⟹ Checked Exceptions

```
BufferedReader b = new BufferedReader(new
                   InputStreamReader(System.in));

    j = Integer.parseInt(b.readLine());
```

→ This may throw exception i.e it may give
a string value that cannot be converted to Int

Add here

```
try
{
  j = Integer.parseInt(b.readLine());
}
catch (IOException e)
{
  SOP("You were told to input integer");
}
```

## Finally Block

This is the default block it runs no matter what happens.

```
try
{

}
catch (Exception e)
{
}
finally
{
   SOP ("Bye");
}
```

## Try with Resources.

```
BufferedReader b = new BufferedReader (new
InputStreamReader(System.in));
String str = "";
str = br.readLine();

catch (Exception e)
}
finally
{

br.close();
}
```

→ Here b obj is the resource as it gives a input value.

→ It is important to close all resources in java for efficiency.

This can be done without catch & finally.

```
try (BufferedReader b = new ---(System.in));
{
    String str = "";
    str = b.readLine();
}
```

→ Here if there is an Exception it will exit and also close the resource.

→ It will close the resource even if there is not an exception.

→ This was introduced in Java 1.7.

## ⟹ Throws & Throw

→ In order to suppress the error and not handle it we can use throws.

```
BufferedReader b = new - - - - - -
    j = Integer.parseInt(b.readLine());
```

Here j will give Exception Error
i.e it will say that it may give an Exception.
"May"

Page No.
Date

→ When we know that it may throw an Exception we can suppress this error and not handle.
→ i.e If there is no error it will run smoothly and if there is error it will show error.

→ But there will be no compile time error.

This can be done by throws.

→ Throws is always written after methods.

```
PS VM (String args[]) throws Exception
{ int j;
  BufferedReader b = new - - - - - -
  j = Integer.parseInt(b.readLine());
}
```

The possibility of error at j will be suppressed.

After throws you can also write multiple Exceptions. using comma (,)

eg: throws IOException, ArithmeticException

⟹ Throw Exception

→ This is used to manually throw an Exception.

→

```
try
{ k = i + j;
  if (k < 10)
  {
    throw new ArithmeticException();
  }
}
catch (ArithmeticException e)
{
  SOP("Minimum output is 10");
}
```

* User Defined Exception

```
PSVM(---)
{int i = 5;
 try
 {
   if (i < 10)
   {
     throw new MyException("Error");
   }
 } catch
 catch (MyException e)
 {
   sop(e);
 }
}
```

// Continued next Page

Page No.
Date

```
class MyException extends Exception
{
    public MyException (String msg)
    {
        super(msg);
    }
}
```

Output :

ex. MyException : Error

## * Object Cloning

```
A obj = new A();
obj.i = 5
obj.j = 6

A obj1 = obj.clone();
```

This creates a new object i.e (obj 1) with same values of obj

⟶ This is called object cloning.

⟹ Other two types are
(1) Shallow Copy (2) Deep Copy.

(1) Shallow Copy.

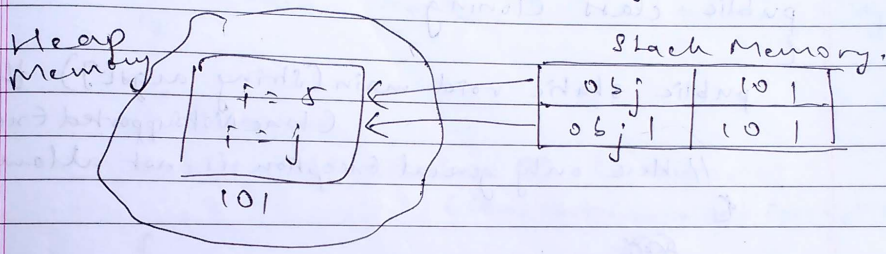
```
A obj = new A();
obj.i = 5;
obj.j = 6;

A obj1 = obj;
```

Here it copies the data but both have SAME REFERENCE

∴ If you change value in one object the values change in other object too.

Practically there is only one object but ~~two~~ two names pointing at it.




(2) Deep Copy

→ Here you copy values manually.

```
A obj = new A();
obj.i = 5;
obj.j = 6;

A obj1 = new A();
obj1.i = obj.i;
obj1.j = obj.j;
```

→ This should be done manually so it is not recommended.

Page No.
Date

## * Object Cloning

→ This is not done directly. It needs permission.
→ Object Cloning is not safe for the software Hence JVM doesn't allow cloning unless you provide permission.

```
public class Cloning
{
    public static void main(String args[]) throws
                                CloneNotSupportedException
    // Here only general Exception is not allowed.
    {
        Abc obj = new Abc();
        obj.i = 5;
        obj.j = 6;

        Abc obj1 = (Abc) obj.clone();
        /*
                ↑
            This is compulsory to specify
            whose object it is.
        */
        obj1.j = 8;

        sop(obj1);
        sop(obj);
    }
}
```

// P.T.O

Permission Through Marker Interface.

```
class Abc implements Cloneable
{
    int i;
    int j;

    @Override
    public String toString()
    {
        return "Abc{" + "i=" + i + ",j=" + j + "}";
    }

    @Override
    public Object clone() throws
                        CloneNotSupportedException
    {
        return super.clone();
    }
}
```

→ JMP

## * FILE HANDLING

→ To write content to a file use function.

writeUTF(" ");

→ This function belongs to ~~~~ class DataOutputStream

→ This ~~ is subclass of ~~~~ asks for object of FileOutputStream

Page No.
Date

→ To read content of a file use method

readUTF();

→ This method belongs to class.

DataInputStream

→ This asks for the object of the class FileInputStream.

```
publiz class FileDemo
{
    p s v m (.....) throws Exception
    {
        FileOutputStream fos = new FileOutput
                               Stream("demo.txt");
        // File Name
        DataInputStream dos = new DataInput

        DataOutputStream dos = new DataOutput
                               Stream(fos);
        // objectofFileOut... class

        dos.writeUTF("Demo Content written");
        // belongs (dos object) i-e
```

Writing to file

// ⟹ These throw exceptions like file not found
// Hence throws Exception is used.
// P.T.O

read from a file

read from file a

```
FileInputStream fis = new FileInputStream
                      ("demo.txt");
//              File Name  <-----
DataInputStream dis = new DataInputStream
                      (fis);
// object of FileInputStream <-----
String str = dis.readUTF();
// gets a string when it reads from a file.
// and stores it in str.
}
}
```

⟹ All of these belong to java.io;

→ Simpler way is creating object of the file. and using the object wherever it asks for the file name

→
```
{
File f = new File("demo.txt");
FileOutputStream fos = new
        FileOutputStream(f);
FileInputStream fis = new
        FileInputStream(f);
}
```

Page No.
Date

## * MULTI THREADING

```
1  public class ThreadingDemo
2  {
3     ps vm (_ ... )
4     {
5        A obj = new A ();
6        obj.start ();
7
8        B obj1 = new B ();
9        obj1.start ();
10    }
11 }
12 class A extends Thread
13 {
14    public void show ()
15    {
16       for(int i=1; i<=5; i++)
17       {
18          S O P (" Hi ");
19       }
20    }
21    public void run ()
22    {
23       show ();
24    }
25 }
```

```
26  class B extends Thread
27  {
28      public void show()
29      {
30          for(int j=1;j<=5;j++)
31          {
32              sop("Hello");
33          }
34      }
35  }   public void run()
36      {
37          show();
38      }
39  }
```

⟹ This gives a Random output of : Hi, Hello etc.

⟹ We can implement sleep function so that the thread waits for that period of time.

Eg: We can add Thread.start(milliseconds)
1000 milliseconds = 1 second.
between lines 18 & 19 & 32 & 33.

→ This may still bring randomness but will be visibly distinct.

→ This sleep function should be written with try catch because it may give exception as InterruptedException.

Page No.
Date

→ run method is compulsory in Threading.

→ start method basically searches for run method and starts the execution.

→ Main class is the inbuilt thread.

Program

```
public class ThreadDemo
{
    p s vm (------)
    {
        A obj1 = new A();
        obj1.start();

        B obj2 = new B();
        obj2.start();
    }
}
class A extends Thread
{
    public void show();
    {
        for(int i=0; i<=5; i++)
        {
            SOP("Hi");
            try
            {
                Thread.sleep(1000);
            }
```

// Sleep belongs to thread class

1000 until 1 second = 1 second

// Sl

```
            catch (InterruptedException en)
            {
                - - - -
            }
        }
    }
    public void run()
    {
        show();
    }
}
class B extends Thread
{
    public void show();
    {
        for(int j=1 ;j<=5 ;j++)
        {
            SOP ("Hello");
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException en)
            { - - - }
        }
    }
    public void run()
    {
        show();
    }
}
```

// Sleep belongs to Thread

→ = 1 second

Page No.
Date

Output :

Hi
Hello
Hi
Hello
Hello
Hi
Hello
Hi
Hello
Hi

This can be random too
Depends on scheduler of our OS

⟹ Runnable

→ Sometimes,
class C extends A, Thread

→ Here,
we want to inherit A but also use Multithreading.

→ we cant use multiple Inheritance in Java
Hence, we can overcome this
using Runnable Interface.

i.e class C extends A implements Runnable

⟹ Efficiency using Anonymous object, Anonymous Class, Inner Class, Lambda Expressions, etc

→ check online if necessary.

# Vectors

It is a dynamic array.

The default capacity of a vector is 10
If you add 11 elements the capacity updates to 20 then to 40 then 80 and so on. i.e increases by 100%.

We can also add by add method.

Vector v = new Vector();
→ Vector Class

v.add(number) or element

We can also remove element in vector.

v.remove(index) → Note (index) not element.

In vector when we remove it appends the next value to previous value.

Vector v = new Vector();

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 10 | 30 | 40 | 50 | 20 |

v.remove(1);

| 10 | 40 | 50 | 20 |
|---|---|---|---|

SOP(v[1]);

Output: 40

Page No.
Date

## * ArrayList

→ It is same as Vector.

Differences

| → | ArrayList | Vectors |
|---|---|---|
| ① | Fast | Slow |
| ② | Not ThreadSafe | ThreadSafe |
| ③ | Increases size by 50% | Increases size by 100% |
| ④ | Doesn't waste too much memory | Wastes too much memory. |

→ Always prefer ArrayList over Vectors.