METHODS / FUNCTIONS

A method is a named block of code that is reusable in nature

The syntax for creating the methods consists of 2 parts
① Method Declaration ② Method Definition / Implementation

Syntax

<access-modifier> <modifiers> returntype methodName(<arguments>)

{

}

access-modifier: public, private, protected
modifiers: static, final (public, private, final)
<optional>

- A method should always be declared directly in the class body
- A method will only execute when it is called
- After creating a method it can be called n-number of times
- we can declare method multiple methods in a class based on our requirement

PASSING VALUES

It is the concept of sending value from calling method to called Method

- We can pass values with the help of arguments
  formal arguments : variable created in method declaration
  actual arguments : values provided in calling method
- We can declare multiple arguments and pass multiple values
- When declaring multiple arguments it should be separated using "," (comma) symbol
- Arguments are also sometimes called as parameters
- programmers responsibility is to provide arguments of correct type, length and sequence

Programs

```
class Program1
{
  static void test (int ar)
  {
    System.out.println("value:"+arg);
  }
  public static void main(String[] arg)
  {
    System.out.println("program start");
    test (10);
    test(142);
    test (420);
    System.out.println("program end");
  }
}
```

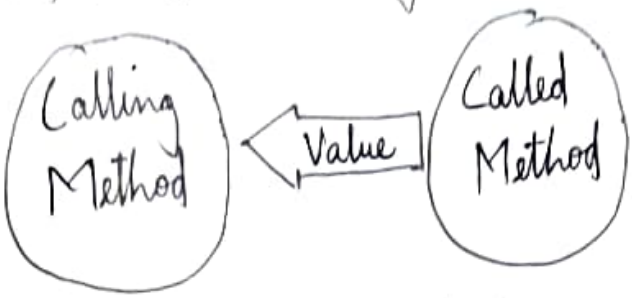
```
class Program2
{
  static void push(char c)
  { System.out.println("My Value:"+c);
  }
  static void play(double d)
  { System.out.println("Value:"+d);
  }
  public static void main(String[] args)
  {System.out.println("program start");
   play(3.45);
   push('J');
   System.out.println("program end");
  }
}
```

```
class Program 3
{
  static void multiply(int x, int y)
  { int mult = x*y;
    System.out.println("Result:"+mult);
  }
  static void addition(int a, int b)
  { int sum = a+b;
    System.out.print("Result:"+sum);
  }
  public static void main(String[] args)
  { addition(2,3);
    multiply(2,3);
    addition(2,6);
  }
}
```

main() → test()
static
i) Object Creation
ii) Same Modifier
→ non-static

Returning Value

- It is concept of sending a value from called method back to calling method



- To return a value the programmer has to follow 3 steps :

i) Declare returntype based on value to be returned
ii) Return a compatible value using return keyword
iii) Capture returned value in calling method

```
class Exa
{
 static int test()
 { return 10;
 }
 public static void main(String[] args)
 {
    int val = test();
    System.out.println("Value returned is :" + val);
   System.out.println("Returned value :" + test());
    S.O.P("Return value" + play() + "Return value" + disp());
 }
 static double disp()
   { return 4.5;
 static char play()
   { return 'J'; }
```

program for returning & passing value

```
class Ex4
{
  static int add(int a, int b)
  {
      int sum = a+b;
      return sum;
  }
  public static void main (String [] args)
    {
      int res = add(10,20);
      System.out.println("Result : "+res);
    }
}

class Ex5
{
  static void verify()
  static boolean verify (int a)
  { if (a%2==0) {
      return true;
    }
    else {
      return false;
    }
  }
  public static void main (String[] args)
    {
      System.out.println("Status : "+verify(1));
    }
}
```

Note

- void return type indicates that the method does not return any value to calling method

```
class Ex
{ static void test()        // Method without argument and without return value
  { }
  static void play(int a)   // Method with argument and without return value
  { }
  static double disp()      // Method without argument and with return value
  {
    return 4.5;
  }
  static boolean send(int a)  // Method with argument and with return value
  {
    return false;
  }
```

Recursion in Methods

or recursion

A. method is said to be in recursion in below 2 scenarios,

a) The method gives a call to calling method

b) The method gives a call to itself

If method recursion is not controlled it leads to stackOverFlowError.

```
class Activity
{ static void displayAscending(int a){
    System.out.println(a);
    if (a<5){
      a++;
      displayAscending(a);
    }
  }
  public static void main(String[] args)
  { displayAscending(1);
  }
```

// 1 2 3 4 5

```
static void display(int a){
  System.out.println(a);
  if (a>0){
    a--;
    display(a);
  }
}

display(5);
```

Interview Question

Q What is a Method

Defination:
Syntax
Basic level properties: 4-points

Q Explain Passing Value
Defination:
How: Arguments (Formal/Actual) multiple arguments

Q Explain Returning Value
Defination:
How: 3-steps

Q. Is it mandatory to declare arguments for a method? No

Q. Is it mandatory to provide "return type" for a method? Yes

Q What is the use of void return type?
A It is used to inform the method doesnot return any value

Method are of 2 types → Parameterised
→ non-parameterised

Identifiers

Any Name that is given by the programer is called as Identifier

Generally the programmer is responsible for providing the following names

* classnames
* methodNames
* variableNames

Rules

- Identifier name should alway begin with alphabet
- Numbers are allowed but not in the beginning
- "$" and "_" symbols are allowed but not recommended
- Keywords cannot be used as Identifies
- Blank Spaces are not allowed in Identifiers

Industry Standard Naming Convention

(i) ClassNames

- In case of classnames The first letter of each word should be in uppercase

Ex class Login class CredentialManager

class EmployeeProfileVerifier

ii) MethodNames

```
void test()
void printBill()
void updateEmployeeSalary()
```

iii) Variable Names

```
int age;
int employeeId;
long creditCardNumber;
```

- In case of Method Names and Variable Names the first word is completely in lower case with subsequent words first letter in uppercase

SCANNER

- It is an Inbuilt program which is present in java library
- We can make use of scanner to read user input
- To make use of scanner we have to follow 3 steps

step 1 import scanner from "java.util" pakage

step 2 create an Instance/Object of scanner

step 3 ~~reads~~ read values using the methods of scanner

PRIMITIVES	METHOD
byte	nextByte()
short	nextShort()
int	nextInt()
long	nextLong()
Float	nextFloat()
Double	nextDouble
char	~~nextC~~ NA
boolean	nextBoolean()

```
class Demo
{

}
Demo d = new Demo();

class Sample
{

}
Sample s = new Sample();
```

```
import java.util.Scanner
public class ProgramForScanner
{
    public static void main (String[] args)
    {
        Scanner scn = new Scanner(System.in)
        System.out.println("Enter Integer Number");
        int integerNumber = scn.nextInt();
        System.out.println("Integer Number is "+integerNumber);
    }
}
```

```
import java.util.Scanner
public class ProgramForTwoScanner
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter number1 is");
        double num1 = scan.nextDouble();
        System.out.println("Enter number2 is");
        double num2 = scan.nextDouble();
        double sum = num1+num2;
        System.out.println("& Sum is" + sum);
    }
}
```

FOR-LOOP

Syntax

```
for(initialization ; condition ; operation)
{
    // for loop body
}
```

1 → 5 a — iterating

```
for(int a=1 ; a<=5 ; a++)
{
    S.o.pln("Java");
}
```

a=1 ✓ Java
a=2 ✓ Java
a=3 ✓ Java
a=4 ✓ Java
a=5 ✓ Java
a=6 ✗

```
for(int x=5 ; x>0 ; x--)
{
    S.o.pln("Full");
}
```

x=5 ✓ Full
x=4 ✓ Full
x=3 ✓ Full
x=2 ✓ Full
x=1 ✓ Full
x=0 ✗

```
for(int b=1 ; b<=10 ; b=b+2)
{
    S.out("Biryani");
}
```

b=1 ✓ biryani
b=3 ✓ "
b=5 ✓ "
b=7 ✓ biryani
b=9 ✓ biryani b=11 ✗

```
for(int m = 10; m > 0; m = m-3)
{
    System.out.println("Jspider");
}
```

m = 10 ✓ Jspider
m = 7 ✓ Jspider
m = 4 ✓ Jspider
m = 1 ✓
m = -2 ✗ Jspider

```
int n = 5;
for(int i=0; i<n; i++)
{
    System.out.println("*");
}
```

i = 0 ✓ *
i = 1 ✓ *
i = 2 ✓ *
i = 3 ✓ *
i = 4 ✓ *
i = 5 ✓ *
i = 6 ✗

```
int n = 5;
for(int i=0; i<5; i++)
{
    System.out.print("*"+" ");
}
```

i=0	i=1	i=2	i=3	i=4	i=5
✓	✓	✓	✓	✓	✗
*	*	*	*	*	

1 0 1 0 1

```
int n=5;
for(int i=0; i<n; i++)
{
    if(i%2==0) {
        System.out.println("1 +" ");
    }
    else {
        System.out.print(0+" ");
    }
}
```

1 * 2 * 3

```
int n = 5; int value = 1;
for(int i=0; i<n; i++){
    if(i%2==0){
        System.out.print(value+" ");
        value++;
    }
    else {
        System.out.print("*"+" ");
    }
}
```

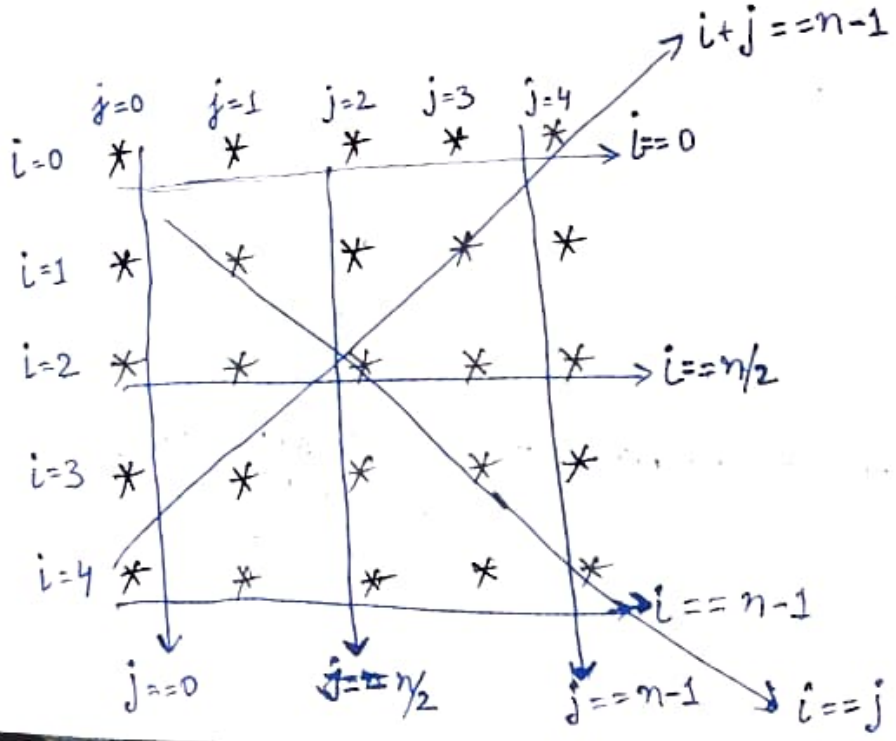
Q) 12321
A) a * b * c

Nested For Loop

```
int n = 5; //Size of Square
for (int i=0; i<n; i++) <-- // No of ROWS
{
    for(int j=0; j<n; j++) //No. of Columns
    {
        System.out.print("*"+" ");
    }
    System.out.println( ); //move to new lines
}
```

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

	j=0	j=1	j=2	j=3	j=4	j=5
i=0	*	*	*	*	*	
	j=0	j=1	j=2	j=3	j=4	j=5
i=1	*	*	*	*	*	
	j=0	j=1	j=2	j=3	j=4	j=5
i=2	*	*	*	*	*	
	j=0	j=1	j=2	j=3	j=4	j=5
i=3	*	*	*	*	*	
	j=0	j=1	j=2	j=3	j=4	
i=4	*	*	*	*	*	—



Scanner does not provide any inbuild methods for reading char type value

next() → read single word
nextline() → read multiple vale → string

When using Scanner we should always enter values of the correct datatypes otherwise it will cause "InputMismatchException"

ARRAYS

→ data structure

- An Array is a finite set of Homogeneous values.

finite : Arrays are fixed in size
set : Array contain group of values
Homogeneous : All value inside an array will be of same data

In java a progrommer can create an array in 2 ways

i) Using Dimensions
ii) Using Array Initialization

Using Dimension : In this approach an array is created by providing datatype and size of a array

Syntax: datatype variableName = new datatype[size]

```
int [ ]    arr1 = new int [10]
char [ ]   arr2 = new char [7]
double [ ] arr3 = new double [3]
```

Using Array Initializers

In this approach an array is created by providing datatype and actual value.

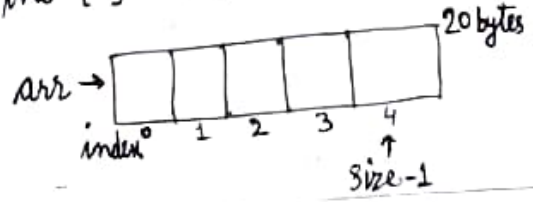
Syntax

```
datatype [ ] variableName = {value1, value2, ..... valuen};
```

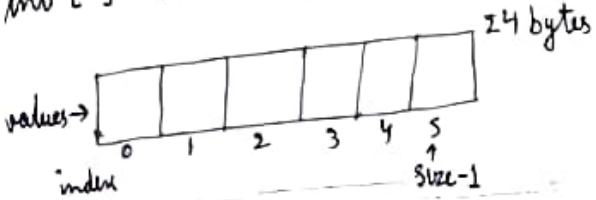
```
int [ ]    dd = { 10, 20, 27, 28};
double [ ] height = {10.2 , 5.2, 6.3, 2.2};
char [ ]   vowels = {'a', 'e', 'i', 'o', 'u' };
```

```
datatype [ ] variableName;
datatype variableName [ ];
datatype [ ]variableName;
```

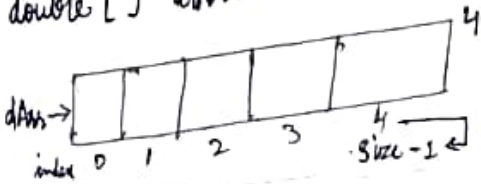
```
int [ ] arr = new int [5];
```



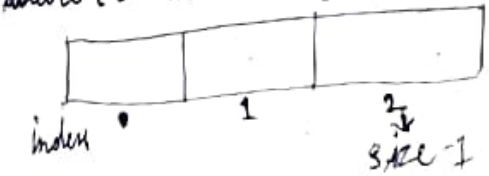

```
int [ ] values = { 13, 12, 10, 2, 1, 10};
```




```
double [ ] dArr = new double [5];
```

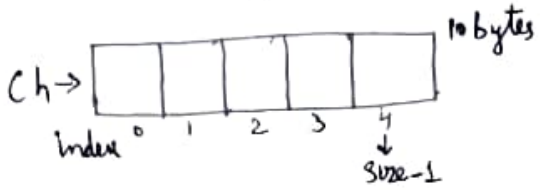



```
double [ ] dValues = {7.2, 4.3, 2.2};
```



```
char [] myChar = new char[10];
```

```
char [] ch = {'T','i','g','e','r'};
```



* Whenever we create an array the JVM allocates a ~~contigous~~ contiguous block of memory based on the datatype and size of the array.

The block is internally divided into segments of equal size

Each segment inside the array can be accessed using index values.

i) index value alway begin at "0"

ii) index value alway end at "arraylength-1"

→ length is a property in array not function

ARRAY LENGTH

length is a unique property with array that helps the programer to find the size or capacity of an array

```
int [] arr = new int [7];
Sopln (arr.length); //7

double[] cgpa = {7.9, 8.4, 9, 2};
Sopln (cgpa.length); //3

int [] evNo = {6, 4, 8, 10, 12};
Sopln (evNo.length); //5
```

```
char [] ch = new char[15]
Sopln (ch.length); //15
```

Note: Whenever we create an array with dimentions approch it contains default values

byte, short, int, long → 0
float, double → 0.0
char → \u0000
any other data type → null
boolean → false

ArrayIndexOutOfBoundsException

The JVM throws this exception when the programer trys to access an invalid index value in in an Array
A valid index will be >=0 && <array.length

```
public class Array1 {
    public static void main(String[] args) {
        int [] arr = new int [5];
        arr [0] = 15;
        arr [1] = 17;
        arr [2] = 19;
        arr [3] = 21;
        arr [4] = 23;
        for (int i=0; i< arr.length; i++){
            System.out.println (arr[i]);
        }
    }
}
```

int [] arr = {15,17,19,21,23};

```
import java.util.Scanner;
class Program
{
    public static
```

STRING

. String is a non primitive datatype that is used to represent a sequence of characters
. we can represent string type values as shown bellow

```
String city = "Banglore";
String cource = "Java
```

String provides several redimade method that can be used by the programmer to manipulate strings values

String method names:-

* charAt()
* indexOf()
* startsWith()
* equalsIgnoreCase()
* toLowerCase()
* trim()
* length()
* lastIndexOf()
* endWith()
* toCharArray()
* contains()
* equals()
* toUpperCase()
* substring()
* split()

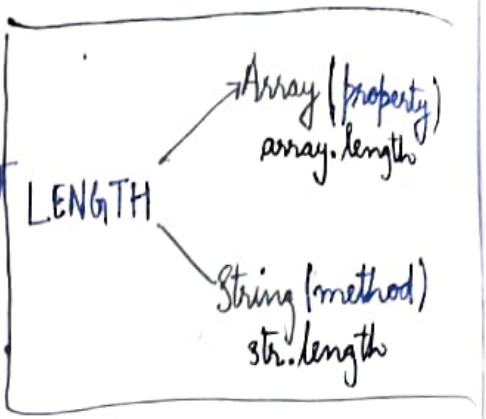
charAt()

```
String str = "Developer";
             0 1 2 3 4 5 6 7 8

Sopln(str.charAt(4));   // l
Sopln(Str.charAt(12));  // Exception → StringIndexOutOfBoundsException
```

- It returns the character at specified index from the current string value
- If given index is ~~invalid~~ invalid it will cause StringIndexOutOfBoundsException

length()

```
String s1 = "BTM";
Sopln (s1.length()); //3
String s2 = "Ja_Va23";
Sopln (s2.length()); //7
String s3 = "Soft Lan";
Sopln (s3.length()); //8
```



It returns the number of characters count in the current String value

toCharArray()

```
String str = "Tiger";
char[] ch = str.toCharArray();
for(int i=0; i<ch.length; i++)
  {
    Sopln(ch[i]);
  }
```

toCharArray() method helps the programmer to represent a String value in the form of a character array.

toCharArray() method does not convert string to character array it creates an equivalent character array

indexOf()

- It returns the index value of given character in current string
- It searches in left --> right direction
- If given character is not present in current string it returns -1
- we can search for multiple occurance

lastIndexOf()

- It returns the index value of given character in current string
- It searches in right --> left direction
- If given character is not present in current string it returns -1
- we can search for multiple occurance

```
String str = "Developes";
Sopln(str.indexOf('v')); //2
Sopln(str.indexOf('l')); //4
Sopln(str.indexOf('z')); //-1
int a = str.indexOf('e'); //1
int b = str.indexOf('e', a+1); //3
int c = str.indexOf('e', b+1); //7
Sopln(a);
Sopln(b);
Sopln(c);
```

```
Sopln(str.lastIndexOf('v')); //2
Sopln(str.lastIndexOf('l')); //4
Sopln(str.lastIndexOf('z')); //-1
int x = str.lastIndexOf('e');
int y = str.lastIndexOf('e', x-1);
int z = str.lastIndexOf('e', y-1);
Sopln(x); //7
Sopln(y); //3
Sopln(z); //1
```

contains()

This method is used to check if current String contains a particular sequence of values

startsWith()

This method is used to check if current String begins with a particular sequence of Values

endsWith()

This method is used to check if the current string ends with a particular sequence of characters

```
String str = "Engineering";
Sop ln (str.contains ("job")); //false
Sop ln (str.contains("ing")); //true
Sop ln(str.startsWith("engine")); //false
Sop ln(str.startsWith("Eng")); //true
Sop ln(str.endsWith("engine")); //false
Sop ln(str.endsWith("ing")); //true
```

```
String str = "j2ee";
Sopln (str.equals("J2ee"));          // false
Sopln (str.equals('j2ee'));          // true
Sopln ( str.equals ('adv java'));    // false
Sopln ( str.equalsIgnoreCase ("J2ee"));  // true
Sopln( str.equalsIgnoreCase ("Adj Java"));  // false
```

equals()

This method is used to check for equality b/w 2 String Values.

equalsIgnoreCase()

This method is used to check for equality b/w 2 string while ignoring the case differences

toUpperCase() , toLowerCase()

```
String str = "Java_SE22";
Sopln ( str.toUpperCase());   // JAVA_SE22
Sopln (str.toLowerCase());    //java_se22
Sopln ( str);  //Java_SE22
```

~~to~~ toUpperCase()

This method returns an equivalent String where all alphabets are in uppercase

toLowerCase

This method returns an equivalent string where all alphabets are in lowercase

Substring()

```
String str = "Developer";
             0 1 2 3 4 5 6 7 8
Sop ln ( str.substring(6)); // per
Sopln ( str.substring(3, 8)); // elope
Sopln ( str. substring(0, 7)); // Develop
```

This method is used to derive a sequence of characters from the current String

trim()

```
String str = "   Core   Java   ";
Sopln ( "->"+ str +"<-");
Sopln ("->"+ str.trim() + "<-");
```

This method is used to eliminate unnecessary space from beginning and ending of String

split()

```
String str = "Java is a Object Orientation Programing lang";
String [] arr = str.split (" ");
for ( int i = 0; i< arr.length; i++)
{ System.out.println (arr[i]);
}
```

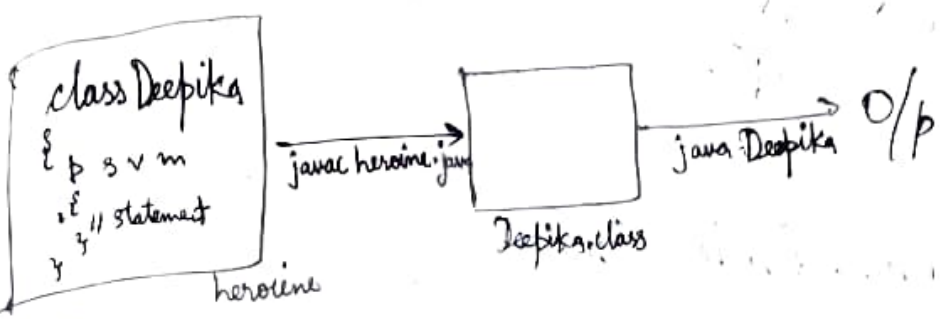
Java
is
a
Object
Orientation
programing
language

This method is used to cut open the current String into several smaller strings
it returns the smaller strings in the form of a String type array.

Trick-1

It is possibile for the file name and classname to be different but it is not recommended

In this, The dot class file generated by the compiler will have the same name as the class Name we should always compile using the file name we should always execute using the class name



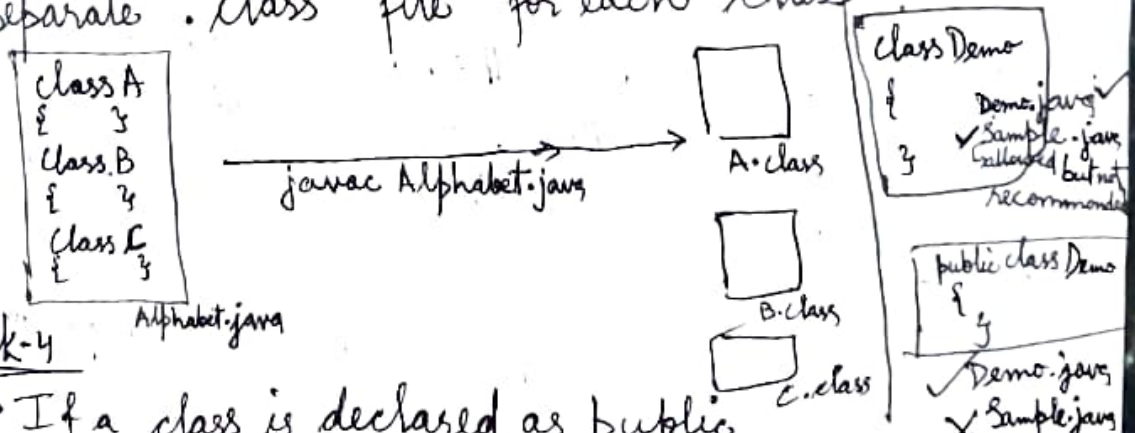
Trick-2

We can create and compile a java class without main method however we cannot execute a class without main method

Trick-3

In a single (".java") file we can create multiple classes

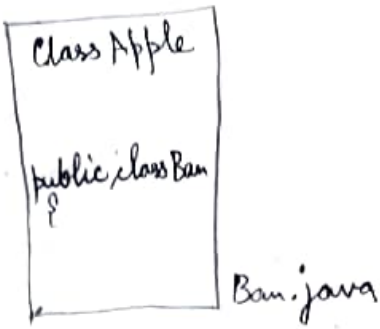
In case of such files the compile create a separate ".class" file for each class



Trick-4

→ If a class is declared as public
It is compalsury for file name to be same as classname

In a single ".java" file only one class can be declared as public



OOPs

static & Non static Members

Members : Any variable or method which is declared directly in the class body is called as a member

- Members are classified into two types
  - static members
  - non-static members
- The variables declared directly in a class are called as DataMember
- The methods are called as Member function

```
class Delta
{ static int a = 10;    // static        (member variable) data member
  int b = 20;           // non-static    data member
  static void play()    // static        member function
  {
  }
  void display()        // non-static    member function (method)
  {
  }
}
```

STATIC METHOD

Static members

Static members are declared with static Keyword

- Static member can be accessed using classname with dot operator

```
class Demo
{ static int a=10;
  static void test()
  { Sopln("-Execute test()---");
  }
public class Program1
{ public static
  { System.out.println(Demo.a);
    Demo.test();
  }
}
```

```
class Sample
{ static double val=4.5
  static char ch='J';
  static void play()
  { Sopln("Play");
  }
  static void run()
  { Sopln("run");
  }
}
```

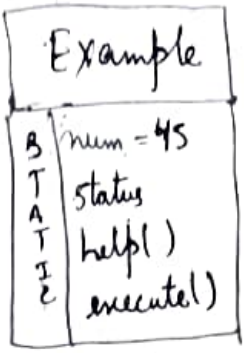
Sample	
STATIC	val=4.5
	ch=J
	play()
	run()

```
public class Program2
{ p s v m( )
  {
```

```
Sopln(sample.val);
Sopln(sample.ch);
Sample.play();
Sample.run();
}}
```

Program2.java

```
public class Example
{
    int num=45
    static int num = 45;
    static boolean b = true;
    static void help()
    {
        Sopln ("help");
    }
    static void execute()
    {
        Sopln (" execute");
    }
}
```



Example.java

public ~~cat class~~

Program3.java

```
public class Program3
{
    public static void main(.String[] args)
    {
        Sopln (Example.num);
        Sopln (Example.b);
        Example.help();
        Example.execute();
    }
}
```

Non Static

The non static members are declared without using non static keyword

System → new class_name();

The non static members in a class can be accessed by creating an object

```
class Demo
{
  int a = 10;
  void test():
  {
     System.out.println("Executing test...");
  }
}
public class Program1
{
    public
    {
        // Sopln(new Demo().a);
        // new Demo().test();
        Demo d = new Demo();
        Sopln(d.a);
        d.test();
    }
}
```

//creating an object

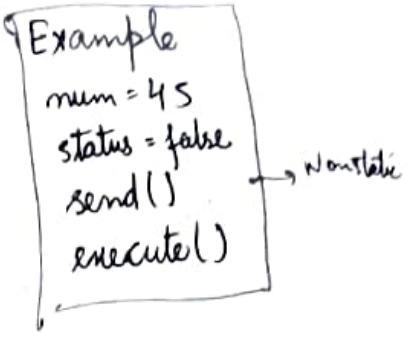
//d → reference variable

P1

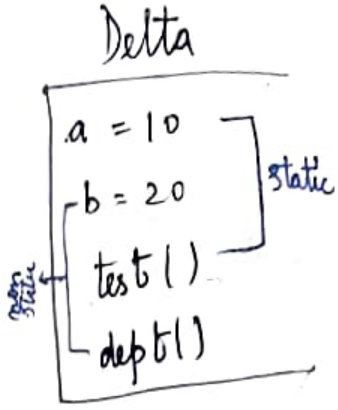
```
class Sample
{
    double val = 4.5
    chas ch = 'c';
    void play()
    {
        Sopln("Executo");
    }
    void sum()
    {
        Sopln("Executesum");
    }
}
class Program2
{
    p s v m(          )
    {
        Sample s = new Sample();
        Sopln(s.vab);
        Sopln(s.ch);
        s.play();
        s.sum();
    }
}
```

Sample	
Non static	val = 4.5
	ch = c
	play()
	sum()

```
class Example
{
    int num = 45;
    boolean status = false;
    void send()
    {
        Sopln("Send");
    }
    void run()
    {
        Sopln("run");
    }
}
class Program3
{
    p s v m(  )
    {
        Example z = new Example();
        Sopln(z.num); Sopln(z.status);
        z.send(); z.run();
    }
}
```



```
class Delta
{
    static int a = 10;
    int b = 20
    static void test()
    {
        Sopln("Execute test");
    }
    void disp()
    {
        Sopln("Execute disp");
    }
}
public class Program4
{
    p s v m( . )
    {
        Delta obj = new Delta();
        Sopln(Delta.a);
        Sopln(obj.a);
        Delta.test();
        obj.disp();
    }
}
```



```
class Tessaract
{   static double rad = 6.7;
    char sym = 'A';
    static void push()
    {
      Sopln ("push");
    }
    void pull()
    {
      Sopln ("pull");
    }
}
public class Program5
{
      p  s  v m (  )
      {
        Tessaract tes = new Tessaract();
        Sopln ( tes.sym);
        Sopln (Tessaract.rad);
        Tessaract.pull();
          tes.pull();
      }
}
```

Tessaract
rad = 6.7
sym = 'A'
push()
pull()
static

// static members can be accessed via reference varcable but it is not the correct approach.

```
class Fcontrol
{
    static int x = 15;
    static void connect()
    {
        System.out.println("Execute connect");
    }
}
public class Program6
{
    p s v m( )
    {
        Fcontrol fox = new Fcontrol();
        Sopln(fox.x);
        fox.connect();
    }
}
```

CLASS & OBJECT

CLASS : A class is a defination block that is used to define the states and behaviours of an object

OBJECT An object is a real world entity that has its own states and behaviours

value based properties
action based properties

```
package classObject
class Student {
    String name;        > Data member
    int id;
    void study() {
        Sopln("Student " + name + " is studying");
    }
    void sleep() {
        Sopln("Student " + id + " is sleeping");
    }
}
public class Program2 {
    p s v m ( )  → Student s1 = new Student();
    s1.name = "Ram";
    s1.id = 101;        > object states
    s1.study();         → Behaviours
    Student s2 = new Student
```

member function

Player
- playerId
- hiScore
- play()
- exit()

Bike
- model
- mileage
- riding()
- stopping()

Profile
- name
- age
- gender
- update()
- remove()
- create()

CLASS

A java class is a definition block used to define the states and behaviours of an object

A class contains members, members are of 2 types
i) Data Members ii) Member functions

Data Members are variables used to define the states of an object
Member functions are methods used to define the Behaviour of an object

OBJECT

An object is a real world entity that has its own states and behaviours

Syntax: `ClassName variable = new ClassName();`

- ClassName → datatype
- variable → reference variable
- new → Keyword
- ClassName() → Constructor call

* In Java, object is a copy of the class that contains all the non-static members
* The non-static members in a class gets loaded to memory during object creation.

object = Instance

object is created RHS in syntax

Reference Variable

- Any variable that is declared using a non primitive datatype is called as a reference Variable.
- Generally we declare reference variable using the following names as datatype
  - * classname
  - * interfacename
  - * enumname

classname referenceVariable = (object reference) → Address of object / (null reference) → NULL

A reference variable in java can hold either address of the object or NULL value

```
class Actor
{   void acting()
    {   Sopln("Actor is Acting");
    }
}
public class Mainclass1
{   public static void main( ):
    Actor a1 = new Actor();
    Actor a2 = null;
    Sopln(a1);
    Sopln(a2);
    a1.acting();
    a2.acting();  //--> Exception: NullPointerException
```

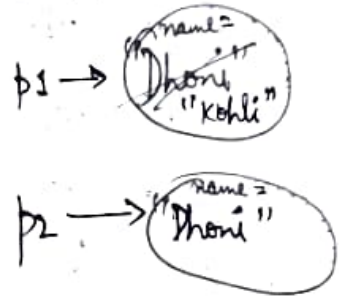
Null : null is a Keyword in java

- null meaning is nothing
- If a reference variable is assigned to null it is called null reference
- null is the default value for all reference variables.
- If we try to access any non-static members using a null reference it will cause "NullPointerException".

Deep Copies & Shallow Copies

Deep Copies : Multiple reference variable of the same class with each reference variable assigned to a distinct object is called as DEEP Copies

In Deep Copies changes made by one reference variable will not ~~affect~~ affect the others




```
class Player
{   String name="Dhoni";
}
public class Mainclass2
{       p s v m (  )
        {
            Player p1 = new Player();
            Player p2 = new Player();
            Sopln (p1.name);        // Dhoni
            Sopln (p2.name);        // Dhoni
        →   p1.name = "Kohli";
            Sopln (p1.name);
            Sopln (p2.name);
}   }
```

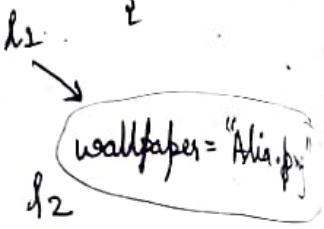
Shallow Copies → Shallow Cloning

Multiple reference variable of the same class assigned to a common object are called as shallow copies

In case of shallow copies changes made by one reference variable will affect the others.

```
class Laptop
{
    String wallpaper = "Depika.jpeg";
}
Laptop l1 = new Laptop();
Laptop l2 = l1;
Sopln(l1.wallpaper);    // Depika.jpeg
Sopln(l2.wallpaper);    // Depika.jpeg
l1.wallpaper = "Alia.png";
Sopln(l1.wallpaper);    // Alia.png
Sopln(l2.wallpaper);    // Alia.png
```

```
public class main
{
    p s v m()
    {
```



Q What is a Class?

Defination

Members, Classify

Data Members, Member Function.

Q What is an Object

Definition

Syntax

Point 1

Point 2

Q What is a reference variable

Definition

Created

Stored

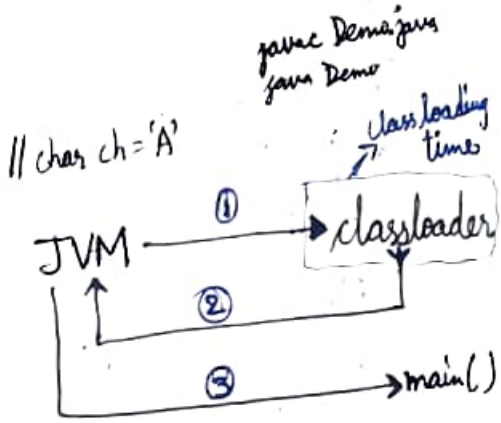
Q What is null? When do we get NullPointerException?

- Null is a keyword . Null meaning is nothing
- When a reference variable is assigned to null it is called null reference
- Null is default value for all reference variables
- If we try to access non-static members using null-reference it will cause NullPointerException

Q Difference b/w Deep Copies & Shallow Copies?

Memory Managment in Java

- In java entire memory managment is handled by the jvm at run-time
- The jvm makes use of the following memory areas:
  i) Class Area
  ii) Heap Area
  iii) Method Area
  iv) Stack Area



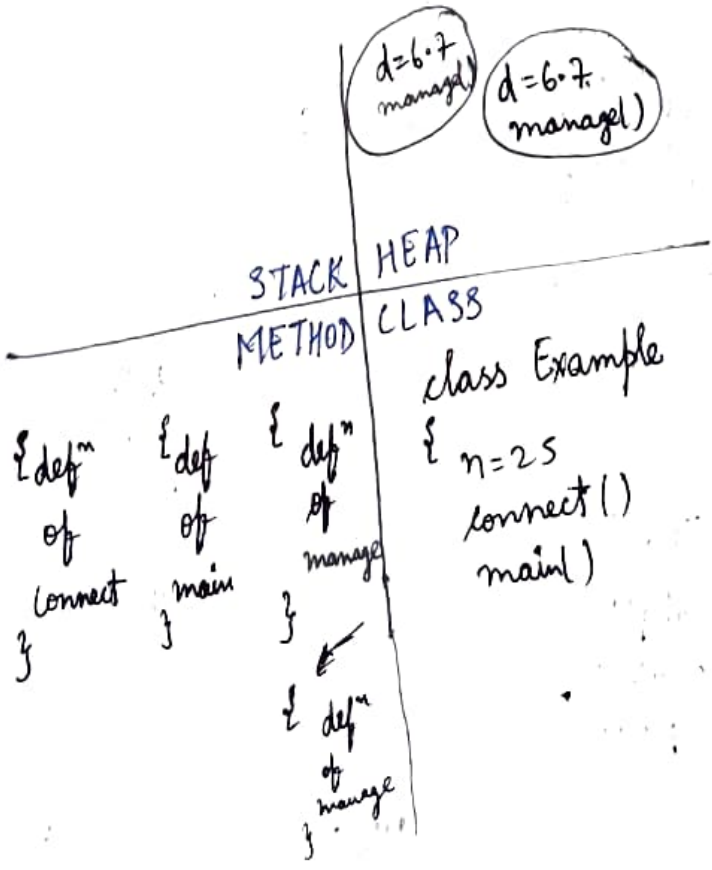

```
class Demo
{   static int a = 10;
    int b = 20;
    static void test()
    {
    }
    void disp()
    {
    }
    public static void main (String [] args)
    {
        Demo ref = new Demo();
    }
}
```

← object creation time

b=20
disp()

STACK	HEAP
METHOD	CLASS
{ def^n of test }	class Demo { A = ~~test~~ 10* test() main() }
{ def^n of main }	ch=A
{ def^n of disp }	

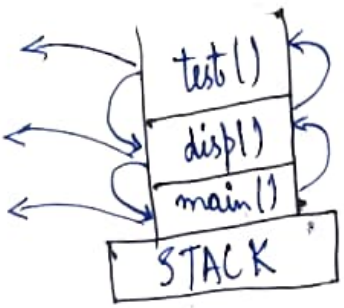
```
class Example
{  static int n=25;
   double d=6.7;
   void manage()
   {
   }
   static void connect()
   {
   }
    p  s  v  m(_)
    {
       Example ex = new Example();
       Example ex1 = new Example();
    }
}
```



classloader → load all static members
→ Execute all the static blocks

```
class Alpha
{
    static void test()
    {
        Sopln("test starts");
        Sopln("test ends");
    }
    static void disp()
    {
        Sopln("disp starts");
        test();
        Sopln("disp ends");
    }
    public static void main(String[] args)
    {
        Sopln("main starts");
        disp();
        Sopln("main ends");
    }
}
```

Entry Order	Exit Order
① main()	① test()
② disp()	⑪ disp()
⑩ test()	⑭ main()



HEAP

```
{ def^n of test }
{ def^n of disp }
{ def^n of main }

class Alpha
{ test()
  disp()
  main()
}
```

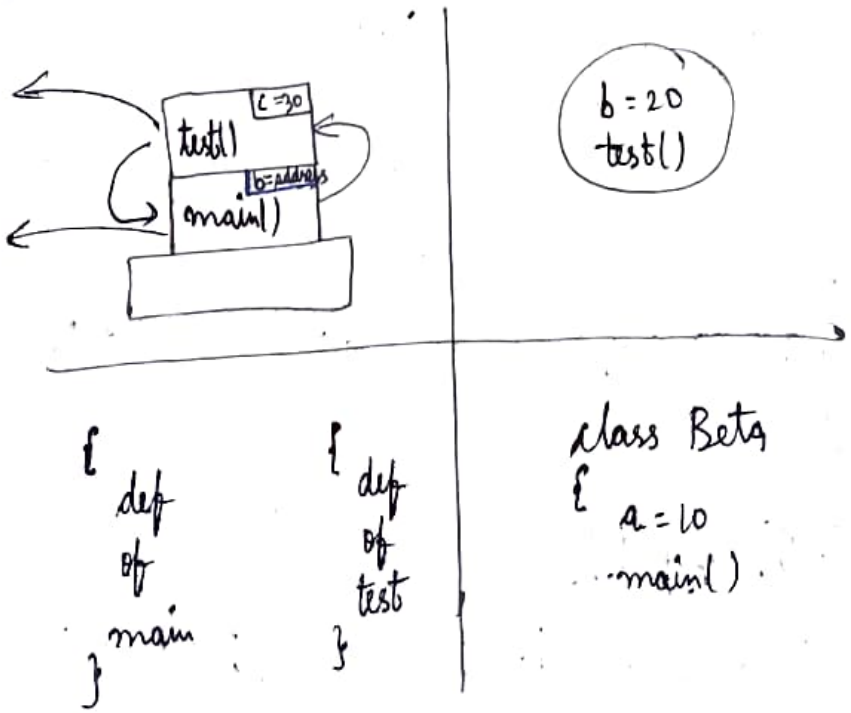
output

```
main starts
disp starts
test starts
test ends
disp end
main ends
```

LIFO → last in first out

FILO → First in last out

```
class Beta
{   static int a = 10;
    int b = 20;
    void test ()
    {
        int c = 30;
        System.out.println (" Executing test ");
    }
    public static void main(
    {
      Beta b = new Beta ();
      b.test ();
    }
}
```



CLASS AREA

- All static members are loaded to class area
- static members are loaded to class area by the class loader
- static member are loaded to memory by default during class loading time
- There exists only one copy of static members in memory.
- class area is also known as static pool

HEAP AREA

- The non-static members are loaded to heap area
- non-static members are loaded to heap area by JVM
- non-static members are loaded to memory during Instantiation (Object creation)
- non-static members multiple copies may be present in heap area
- All objects created in java will be alloted memory in heap area

METHOD AREA

- All method defenitions are stored in the method Area

STACK AREA

- Stack Area is also known as execution memory.
- All method executions is done in the stack area.
- Stack Area follows a LIFO or FILO order of Execution
- All local variables are alloted memory in stack area

STATIC MEMBERS	NON STATIC MEMBERS
• created with static keyword	created without static keyword
• accessed using classname	accessed by creating an Instance
• loaded to class area	loaded to heap area
• only 1 copies exists in memory	multiple copies may exist in memory
• loaded during class loading	loading during instantiation
• loaded by class loader	loaded by JVM
• class level members	instance members

Initialization Blocks

- These are special blocks in a java program that are used to initialize the data members present in a class
- Initialization blocks are classified into 2 type
  i) Static Initialization Block ii) Non-static Initialization block

Static Initialization Block

Static Initialization Block

- Static I. Blocks are created using static keyword
- Static I. Blocks are used to initialize static data members

In a single static block we can initialize multiple static data members.

We can create multiple static blocks inside a class

static blocks are executed by the class loader at the time of ~~class~~ class loading

static blocks will execute only once in the program lifetime

Non Static Initialization Block

Non-static Blocks are created without using static keyword

Non-static Blocks are used to initialize non-static data members

In a single non-static block we can initialize multiple non-static data members

we can create multiple non-static blocks inside a class.

Non static blocks are executed by the JVM at the time of object creation

Non-static blocks may execute more than once in a program lifetime

It is also known as Instance Initialization Block

It is possible to initialize a static member inside a non static member however it is not recommended as because initialization of non static member becomes subject to object creation.

Different stages in which a static data member can be initialized

```
class Foxtrot
{
    static int a = 10;   // Initialization during declaration
                            before class loading
    static
    {
        a=20;            // Initialization during class loading
    }
    public static void main(
        Foxtrot.a = 30; // Initialization after class loading
        Sop/m (Foxtrot.a);
    }
}
```

Different stages in which a non-static data member can be initialized

```
class Gamma
{
    int n=15;      // Initialization during declaration
                   before object creation
    {
        n=25;      // initiazation during class loading
    }
    p s v m (__)
    {
        Gamma ref = new Gamma();
        ref n = 45;    // initialization after
        Sopln(ref.n);      class loading
    }
}
```

Note We cannot store/initialize data members inside a static block

We can store/initialize static data members inside a non static block but it is not recommended

```
✗  int a;   → non-static
   static {
       a=10;
   }
```

```
✓  static int a;
   {
       a=10;   → static
   }
   → non-static
```

It is possible to do any type of coding inside initialization block. but it is recommended to only write the code related to initialization activities

Static Block	Non Static Block
→ created with static keyword	created without static keyword
→ executed by classloader	executed by JVM
→ executed during class loading	executed during object creation
→ executes only once	executes multiple times
→ Responsible to initialize static data members	Responsible of initalize non-static data members

What is classloader

Classloader is sub-system of the JVM present inside the JRE.

→ It is responsible for
- loading all the static members to memory
- executing all the static initikization block

The time interval in which class loader workes is known as class loading time

CONSTRUCTORS

Constructors are special members in a class that are used to initialize non-static data member.

- Every class in java will have a constructor.
- Constructor will always have the same name as the class name.
- Constructors get executed during object creation.
- In java constructor are classified into two categories:-

USER DEFINED CONSTRUCTOR
1. Default Constructor
2. Zero Argument Constructor
3. Parameterized Constructor

Default Constructor

```
class Demo
{
    int a;
    double b;
    String c;
}
public class Mainclass
{
    public static void main(String[] args)
    {
        Demo ref = new Demo();   // object is created by calling the default constructor.
        Sopln (ref.a);   // 0
        Sopln (ref.b);   // 0.0
        Sopln (ref.c);
    }
}
```

② class Sample
{ double val;
  void test()
  {
    So.pln("Executing test()...");
  }
}
public class Mainclass2
{
  p s v m( )
  {
    Sample s = new Sample(); //creating object using by calling default constructor
    So.pln(s.val); //0.0
    s.test();
  }
}

Default Constructors

- The default constructor is provided by JAVA (compiler)
- The compiler provides default constructor when programmer fails to create a constructor
- The default constructor provides default values for the non-static data members which are not initialized

Zero Argument Constructor

①
```
class Alpha
{
    int val;
    Alpha()
    {
        Sopln("Executing Cons");
        val=25;
    }
}
public class Mainclass3
{
    p s v m(Str[] args)
    {
        Alpha a1=new Alpha();
    // Alpha a2=new Alpha();
    // Alpha a3=a2;
        System.out.println(a1.val);
    }
}
```

Executing Cons
25
output

②
```
class Beta
{
    int a;
    double b;
    char c;
    Beta()
    {
        a=15;
        b=4.5;
        c='A';
    }
}
```

```
public class Mainclass4
{
    p s v m( )
    {
        Beta obj=new Beta();
        Sopln(obj.a);      // 15
        Sopln(obj.b);      // 4.5
        Sopln(obj.c);      // A
    }
}
```

Zero Argument Constructor

- It is created by the programmer
- It is possible to initialize non-static data members in the body of zero Argument constructor.
- As the name suggest it does not accept any arguments

Parameterized Constructor

①
```
class Example
{   int a;
    Example (int b)
    {   System.out.println(" Executing Constructor Ex ");
        a = b;
    }
}
public class Mainclass5
{   public static
    {   Example ex1 = new Example(15);
        System.out.println( ex1.a);
        System.out.println("=======");
        Example ex2 = new Example(25);
        System.out.println( ex2.a);
    }
}
```

②

```
class Delta
{
    int num;
    double val;
    char ch;
    Delta (int a, double d, char c)
    {
        Sopln ("Execute Delta Constructor");
        num = a;
        val = d;
        ch = c;
    }
}
public class Mainclass6
    {
        p  s  v  m ( )
        {
          Delta d = new Delta (10, 10.1, 'A');
          Sopln (d.num);
          Sopln (d.val);
          Sopln (d.ch);
        }
    }
```

Parameterized Constructor

It is created by the programmer

In parameterized Constructor we can initialize non-static data members with the help of arguments inside the constructor.

As the name suggest it accepts arguments

Note

When the programmer wants to create an object for a class but is not concerned about initializing the non-static data member then default constructor is best option

When the programmer wants to create objects for a class with all the objects having same initialization for the non static data member then zero argument constructor is best suited

When the programmer want to create objects and provide different initialization value for the non static data member in each object then parameterized constructor is best suited

Q Can we give return type for a constructor?

A No, if by mistake return type is given its behaves like a method

Q Can constructor have modifiers

A Yes, but we can provide access modifiers with respect to normal modifiers

Q Can we initialize static data members inside a constructor

A Yes but it is not recommended as it is not the correct approach

Every class in java have constructor including main

CONSTRUCTOR OVERLOADING

(=0) It is concept of developing multiple constructors inside a class

Rule

When creating multiple constructors in a class

- It is mandatory to vary the argument lists in each constructor
- we can vary the arguments in 3 ways
  - i) Type of arguments
  - ii) length of arguments
  - iii) sequence of arguments

①

```
class Delta
{
    Delta(int a)
    {

    }
    Delta(int b)   // double
    {

    }
    Delta(String c)
    {

    }
}
```

②

```
class Alpha
{
    Alpha(int a)
    {

    }
    Alpha(int a, int b)
    {

    }
    Alpha(int a, int b, int c)
    {

    }
}
```

③

```
class Sample
{
    Sample(int a, String s)
    {

    }
    Sample(String s, int a)
    {

    }
}
```

• Constructor overloading is used to create objects of the same class with different initialization capabalities

Ex → youtube

```
class Amazon
{
    Amazon()
    {
        System.out.println("Normal Acc: Shopping");
    }
    Amazon(double fees)
    {
        System.out.println("prime Acc : Shop, Videos");
    }
}
public class MainClass1{
    static void main(String[] args)
    {
        Amazon a1 = new Amazon();      //Normal Acc
        Amazon a2 = new Amazon(99.0);  // Prime Acc
    }
}
```

Eclipse

i) Eclipse is an IDE (Integrated Development Environment)
ii) It is a free software distributed by Eclipse foundation
iii) It allows programmer to write, compile, execute and debug java programs from a single window

Advantages of Eclipse

i) It provide shortcuts that can save programmer time & efforts
ii) It auto organizes all the imports
iii) It allows easy integration support from 3rd party libraries
iv) Eclipse provide plugins for accessing external application like browsers, web servers etc

Passing and Returning Objects

```
//Model
class Product
{
    int pid;
    double price;
    Product(int pid, double price)
    {
        this.pid = pid;
        this.price = price;
    }
}
//Controller
class ProductHelper{
    static Product createProduct(){
        Scanner scn = new Scanner(System.in);
        Sopln("Enter Pid");
        int id = scn.nextInt();
        Sopln("Enter price");
        double pr = scn.nextDouble();
        Product pro = new Product(id, pr);
        return pro;
    }
    static void displayProductInfo(Product p){
        Sopln("Pid" + p.pid);
        Sopln("Price" + p.price);
    }
}
//View
public class MainClass{
    p s v m(SI) args){
    Product p1 = ProductHelper.createProduct();   // returning objects
    ProductHelper.displayProductInfo(p1);   // passing objects

    Product p2 = ProductHelper.createProduct();
    ProductHelper.displayProductInfo(p2);
    }
}
```

Inheritance

Inheritance is the concept of a class aquiring the properties of an another class

Inheritance can be done using extends keyword

The concept of inheritance is also known as (IS-A relationship)

```
class A
{
```

- Super class
- Base
- Parent

```
}
```

```
class B extends A
{
```

- Sub class
- Derived
- child

```
}
```

```
class Demo{
    void test(){
        Sop.ln("Execute test");
    }
}
class Sample extends Demo{
    void disp(){
        Sop.ln("Execute disp");
    }
}
public class MainClass1{
    p s v m(__){
        Demo d = new Demo();
        d.test();
        Sample s = new Sample();
    } } s.test();   s.disp();
```

o/p

```
Execute test
Execute test
Execute disp
```

Ex

```
class Alpha{
    int a=10;
    void play(){
        Sopln("play");
    }
}
class Beta extends Alpha{
    int b=20;
    void send(){
        Sopln("send");
    }
}
public class MainClass2{
    p  s   v   m(____){
        Beta ref = new Beta();
        Sopln(ref.a);
        ref.play();
        Sopln(ref.b);
        ref.send();
    }
}
```

Rules of Inheritance

1) The sub class cannot access the following members of the super class
   a) private members
   b) static members
   c) constructor
   d) Initialization Blocks
2) The super class cannot access properties of the sub class
3) When a class is declared as final it cannot have a sub class

Advantages of Inheritance

i) Code Reusability
ii) Software Enhancement
iii) Code Modification
iv) Generalization Vs Specialization

```
class InstagramOld {
    void post() {
        Sopln("Upload Post");
    }
}
class InstagramNew extends InstagramOld {
    void message() {
        Sopln("Send Message");
    }
}
public class MainClass3{
    public static void main(String[] args) {
        InstagramOld i1 = new InstagramOld();
        i1.post();
        InstagramNew i2 = new InstagramNew();
        i2.post();
        i2.message();
    }
}
```

```
class Whatsapp1
{
    void message() {
        Sopln(" Send/Receive Message");
    }
}
class Whatsapp2 extends Whatsapp1 {
    void call() {
        Sopln (" Audio/Video calls");
    }
}
class Whatsapps3 extends Whatsapp2
{
    void status() {
        Sopln (" Show Tem photos");
    }
}
public class
    p   s       v       m(_){
        Whatsapp1 w1 = new Whatsapp1();
        w1.message();
        Whatsapp2 w2 = new Whatsapp2();
        w2.message();
        w2.call();
        Whatsapp3 w3 = new Whatsapp3();
        w3.message();
        w3.call();
        w3.status();
    }
}
```

TYPES OF INHERITANCE

i) Single level Inheritance.

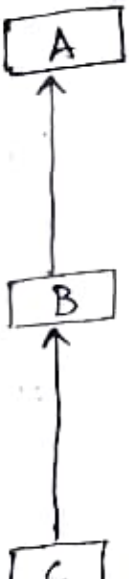
In this form of inheritance property of the super class will be inheritance by only one sub class


```
class A
{   void test()
    {
    }
}
class B extends A
{   void disp()
    {
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        A a1 = new A();
        a1.test();
        B b1 = new B();
        b1.test();
        b1.disp();
    }
}
```

ii) Multi Level Inheritance

In this form of Inheritance the properties of super class is derived by more than one sub-class in different level



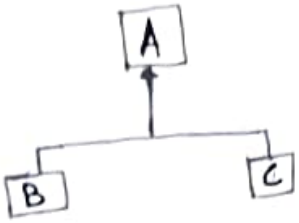
A → B → direct child
A → C → indirect child

```
class A {
    void test() {
    }
}
class B extends A
{   void disp()
    {
    }
}
class C extends B {
    void play()
    {
    }
}
```

```
A a1 = new A();
a1.test();
B b1 = new B();
b1.disp();
b1.test();
C c1 = new C();
c1.play();
c1.test();
c1.display();
}
}
```

iii) Hierarchical Inheritance

In this form of Inheritance the properties of the super class is derived by more than one sub class in the same level

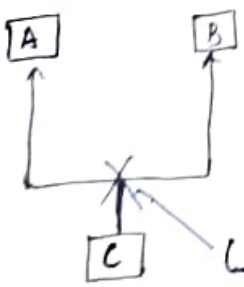



```
class A
{
    void test()
    {
    }
}
class B extends A
{
    void disp()
    {
    }
}
class C extends A
{
    void play()
    {
    }
}
```

```
public class main3 {
    p s v m(_){
        A a1 = new A();
        a1.test();

        B b1 = new B();
        b1.test();
        b1.disp()

        C c1 = new C();
        c1.test();
        c1.play();
    }
}
```

iv) Multiple Inheritance

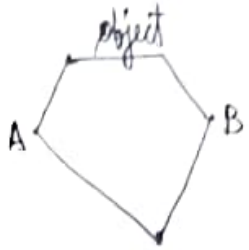
In this form of Inheritance a single sub class try's to directly inherit the properties of more than one super class



- This form of inheritance is illegal and not supported in java using classes
- Multiple Inheritance is not supported with classes as it lead to java's Diamond Ambiguity Problem

- In Multiple Inheritance the sub class does not know from which super class it must aquire the properties of object class
- Multiple Inheritance also creates problem with respect to constructor chaining

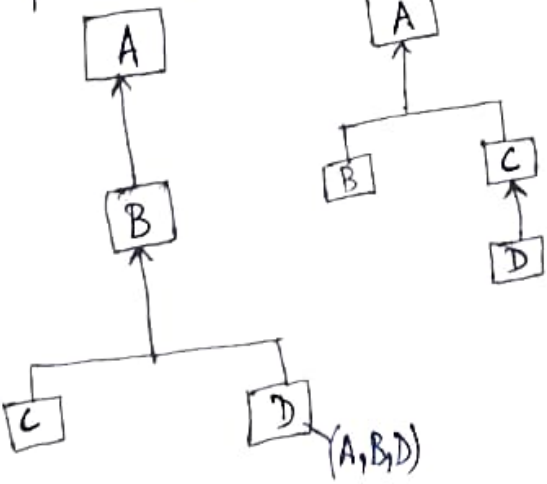
```
class Object
{

}

class A (object)
{

}

class B (object)
{

}

class C extends A,B  ~~~> CTE
{

}
```

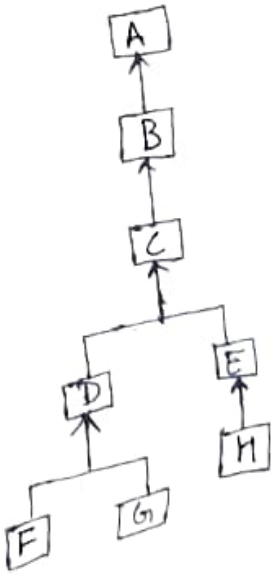


jacob's diamond
Ambiguity Problem

v) Hybrid Inheritance

Hybrid Inheritance is a combination of more than one form of inheritance





Q. What is Generalization and specialization?

A. Generalization is the process of developing common functionalities in super class and specialization is the process of developing specific functionalities in sub class.

CONSTRUCTOR CHAINING

It is the phenomenon of one constructor calling another constructor

In java constructor chaining can be achived in two ways

i) Using this() statement

ii) Using super() statement

this() statement :-

```
class Hotel{
    Hotel(){
        Sopln("KFC");
    }
    Hotel(int a){
        this();
        Sopln("Dominos");
    }
    Hotel(String a){
        this(19);
        Sopln("McDonalds");
    }
}
public class Main1{
    p  s  v  m(S'_M)
    {
        Hotel h1 = new Hotel("ABC");
    }
}
```

Ex (11)

```
class Amazon{
    Amazon(){
        Sopln("Code to initialize shopping");
    }
    Amazon(double d){
        this();
        Sopln("Code to initiatize prime features");
    }
}
public class Main4
{
    p  s  v  m(  )
    {
        Amazon a1 = new Amazon(99.0);
    }
}
```

- this() statement is used to call another constructor of same class
- we have to write this() statement in the first line of the constructor
- we can only write one this() statement in the constructor
- this() statement can only be written by the programmer

Super() Statement

```
class Demo{
    Demo(){
        Sopln("Demo Const");
    }
}
class Sample extends Demo{
    Sample(){
        super();
        Sopln("Sample Const");
    }
}
public class Main1{
    p s v m(){
        Sample s = new Sample();
    }
}
```

Ex ②

```
class Alpha{
    Alpha(){
        Sopln("Alfa Cons");
    }
    Alpha(int a){
        Sopln("Alfa(int) cons");
    }
}
class Beta extends Alpha{
    Beta(){
        // super(45);    //super();
        Sopln("beta cons");
    }
}
```

```
public class Main2{
    p s v m(){
        Beta b = new Beta();
    }
}
```

Ex3

```
class Delta
{
    Delta()
    {
      this(10);
      Sopln("Delta");
    }
    Delta(int a)
    {
      Sopln("Delta(int)");
    }
}
class Example extends Delta
{
    Example()
    {
      Sopln("Example");
    }
    Example(int a)
    {
      this();
      Sopln("Example(int)");
    }
}
```

→ main class

```
Example ex = new Example(15);
```

Ex4

```
class Demo
{

}
class Sample extends Demo
{
    Sample()
    {
      super(); ✓ x ↦ // it is calling the default constructor of the super class
      Sopln("Cons")
    }
}
```

Ex:-

```
class Foutrot
{
    Foutrot()
    {
        super();  → It is calling the constructor of object class
        System.out.println("Four Lons");
    }
}
```

Note:-

- When a super class is having only parametries constructor it is mandatory for the programmer to create a constructor in the sub class and manually write super statement

```
class A
{
    A(int a)
    {
    }
}
class B extends A
{
    B()
    {
        super(10);
    }
}
```

```
class A
{
    A()
    {
    }
    A(int x)
    {
    }
}
class B extends A
{
}
```

```
class A
{
    A(int x)
    {
    }
    A(double d)
    {
    }
}
class B extends A
{
    B()
    {
        super(45);
    }
}
```

```
class Tiger
{
    Tiger()
    {
        this();
    }
}
```

```
class Sample
{
    Sample(int a)
    {
        this(7.8);
    }
    Sample(double d)
    {
        this(10);
    }
}
```

- Super statement can be used to call another constructor of the super class
- Super() statement should always be written in the first line of the constructor
- We can only write one super statement inside a constructor
- If The super() statement can be written either by the programmer or by the compiler
- Multiple Inheritance problem
  Constructor chaining problem ~~acces~~ associated with multiple inheritance
- The super statement in the sub class constructor does not know which constructor should be called

```
class A
{

}

class B
{

}

class C extends A, B
{
    C()
    {
        super();
    }
}
```

TYPE CASTING

Type casting is the concept of assigning a value of one datatype to the variable of another datatype

Type casting is of two 2 types

Primitive TYPE CASTING

NON Primitive Type casting

Primitive Typecasting

It is the concept of assigning a value of one primitive datatype to the variable of another primitive datatype

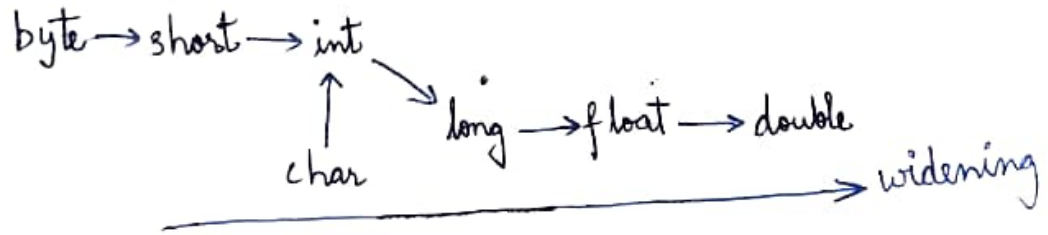
primitive Typecasting is classified into two types

i) Widening

ii) Narrowing

Widening :

- it is the concept of assigning a lower primitive datatype to a higer primitive datatype
- widening is an implicite process that is supported by the compilier



Q1

```
int a = 10;
float f = a;    // widening : int → float
Sopln(f);       // 10.0
long val = 1234567L;
double num = val;   // widening : long → double
Sopln(num);     // 1234567.0
char c = 'A';
int b = c;      // widening : char → int
Sopln(b);       // 65
```

Q2

```
public class Prog2
{
    static void run(double d)
    {
        Sopln("Value :"+d);
    }
    public static void main(String[] args)
    {
        run(4.5);
        run(10);    // Widening ---> int to double
        run('A');   // Widening ---> char to double
    }
}
```

2024-02

Narrowing

- It is the concept of assigning a higer primitive datatype to a lower primitive datatype
- Narrowing is an explicit process that is not supported by the compilier
- to do narrowing the programmer has to write a casting statement

```
public class Prog3
{   p   s   v   m(--)
    {
        double d=3.4;  → casting statement
        int a = (int) d;   // Narrowing : double → int
        Sop ln (a);
        int x=97;
        char c = (char) x;   // Narrowing : int → char
        Sop ln (c);
        double val = 5.678;
        float num = (float) val;   // Narrowing : double → float
        Sop ln (num);
    }
}
```

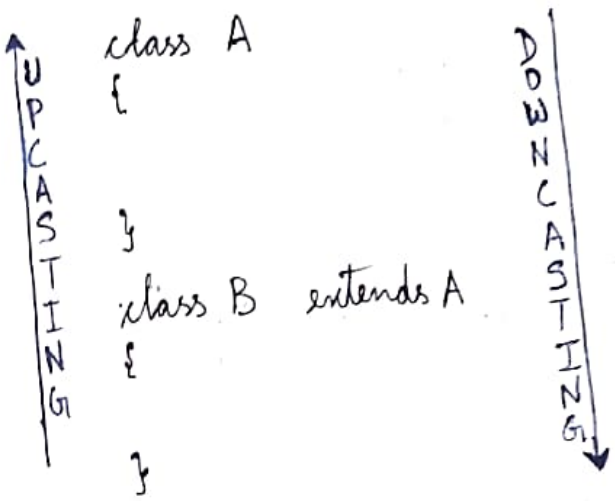
Note

Boolean

- Boolean does not participate in both widening and narrowing
- Keyword → true, false

Non-primitive Typecasting

- Non primitive typecasting is the concept of assigning the referance variable or object of one class/type to the referance variable of another class/type
- To perform non primitive typecasting ~~inheri~~ inheritance is a pre-requisite
- Non primitive typecasting is classified into 2 types
  i) Up casting ii) Down casting



Upcasting

It is the concept of assigning a referance variable or object of the sub class to the referance variable of super class

Upcasting is an implicit process that is supported by the compilier

```
public class Main1 {
    p  s  v  m( )
    {
        Sample obj = new Sample1();
        Demo ref = obj; //Upcasting:
        Sopln(obj);
        Sopln(ref);
    }
}
```

Sample ---> Demo

> print same address

```
class Demo
{
}
class Sample extends Demo
{
}
```

* After Upcasting we can not access the properties of Sub-Class

Ex

```
class Alpha{
        int a=10;
        void test(){
            Sopln("Execute test()");
        }
}
class Beta extends Alpha
{
        int b=20;
        void displ()
        {
            Sopln(" Execute displ()");
        }
}
public class Main2
{
        public static void main(String[] args)
        {
            Beta ref = new Beta();
            Sopln(ref.a);
            ref.test();
            Sopln(ref.b);
            ref.displ();

            Alpha obj = ref;    //Upcasting : Beta → Alpha
            Sopln(obj.a);
            obj.test();
        }
}
```

Alpha obj = new Beta();

```
class A {
    void test() {
        Sopln("test");
    }
}
class B extends A {
    void disp() {
        Sopln("disp");
    }
}
class C extends B {
    void play() {
        Sopln("play");
    }
}
public class Ex4 {
    p  s  v  m(  )
    {
        C c1 = new C();
        c1.test();
        c1.disp();
        c1.play();
        B b1 = c1;   // Upcasting : C → B
        b1.test();
        b1.disp();
        A a1 = b1;   // Upcasting : B → A
        a1.test();
        A a2 = new C();   // Upcasting : C → B → A
        a2.test()
    }
}
```

→ 1 object & 3 reference variables

```
class Fruit {
}
class Apple extends Fruit{
}
class Mango extends Fruit{
}
class Grape extend Fruit{
}
class FruitBasket
{
    static void addToBasket(Fruit obj)
    {
        if(obj!=null)
        {
            Sopln("Fruit Added");
        }
    }
}
```

```
public class Main1
{
    p s v m()
    {
        Apple a = new Apple();
        FruitBasket.addToBasket(a); //Upcasting

        Mango m = new Mango();
        FruitBasket.addToBasket(m);

        Grape g = new Grape();
        FruitBasket.addToBasket(g);
```

Method based conversion

Apple → Fruit

Upcasting is the concept of assigning the object or referance variable of the sub class to the referance variable of the super class

After Upcasting we cannot access the properties of the sub-class

Upcasting is an implicit process that is supported by the compiler

Downcasting

Downcasting is the concept of assigning a reference variable of the super class to the reference variable of the sub class

Downcasting is an explicit process that is not supported by the compiler

To perform downcasting programmer has to write casting statement

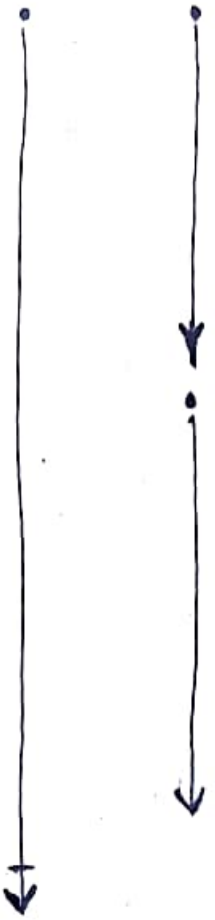
It is mandatory to perform upcasting before downcasting

If we perform downcasting without upcasting it will cause "ClassCastException"

After down-casting we can access the properties of both subclass and super class

```
class Demo
{
}
class Sample extends Demo
{
}
public class Main1
{
    p  s  v  m(—)
    {
        Demo ref = new Sample();
        Sample obj = (Sample) ref;   // Downcasting
        Sop(ref);
        Sop(obj);
```

```
class A
{
    void test()
    {
        Sopln("test");
    }
}
class B extend A
{
    void disp()
    {
        Sopln("disp");
    }
}
class C extends B
{
    void play()
    {
        Sopln("play");
    }
}
public class Mainclass
{
        p      s      v    m(  ){
        A a1 = new C();   // UPCasting C --B--> A
        a1.test();
        B b1 = (B)a1;     // Downlasting (A → B)
        b1.test();
        b1.disp();
        C c1 = (C)b1;     // Downlasting (B → C)
        c1.test();
        c1.disp();
        c1.play();
        C c2 = (C)a1;
        c2.test();
        c2.disp();
        c2.play();
```



test()
disp
play

```
A a1 = new B();    // Upcasting B → A
a1.test();
B b1 = (B) a1;     // Downcasting A → B
b1.test();
b1.disp();
C c1 = (C) b1;     // Downcasting B → C
```

test()
disp()

→ Exception → ClassCastException

instanceof

i) Using instanceof keyword we can determine which instance is present in our reference variable.

```
class SoftwareEngineer
{
    void meeting()
    {

    }
}
class SoftwareDeveloper extends SoftwareEngineer
{
    void coding()
    {

    }
}
class TestEngineer extends SoftwareDevelopes
{
    void testing()
    {

    }
}
class Manager
{
    static void assignTask(SoftwareEngineer s)
    {
        s.meeting();
        if(s instanceof SoftwareDeveloper)
        {
            SoftwareDeveloper dev = (SoftwareDeveloper) s;
            dev.coding();
        }
```

```
        else if (s instanceof TestEngineer)
        {
            TestEngineer qa = (TestEngineer) s;
            qa.testing();
        }

    }

    public class Main
    {
        public static void main(String [] args)
        {
            Manager.assignTask(new SoftwareDeveloper());
            Manager.assignTask(new TestEngineer());
        }
    }
```

Ex

```
class Vegetables{ ①
}
class Potato extends Vegetables{
}
class lemon extends Vegetables{ ②
}
class Zepto {
    static void buyVegetables(Vegetables v)
    {
        if(v instanceof Potato){
            Sop("Potato in kg");
③       }
        if(v instanceof Lemon){
            Sop("lemon in Pieces");
        }
    }
}
```

Main Class

```
④ Zepto.buyVegetables(new Potato());
  Zepto.buyVegetables(new Lemon());
```

Method Overloading

Method Overloading is the concept of developing multiple methods in the class with the same name and different argument list

- we can vary the arguments in three ways
  - i) type of Arguments
  - ii) length of Arguments
  - iii) sequence of Arguments

(i)

```
class Example
{
    void play(int a)
    {
    }
    void play(double b)
    {
    }
    void play(String s)
    {
    }
}
```

(ii)

```
class Demo
{
    void play(int a)
    {
    }
    void play(int a, int b)
    {
    }
    void play(int a, int b, int c)
    {
    }
}
```

(iii)

```
class Sample
{
    void book(int a, String s)
    {
    }
    void book(String, int)
    {
    }
}
```

- Method Overloading is used to perform the same task with the different arguments
- Method Overloading is used to achive compile time polymorphism
- * We can overload static, private and final methods

Ex1
```
class Flipkart
{
    void payment()
    {
        Sopln("COD");
    }
    void payment(long card)
    {
        Sopln("Card");
    }
    void payment(String upi)
    {
        Sopln("UPI");
    }
    void payment(String un, String pass)
    {
        Sopln("Netbanking");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Flipkart obj = new Flipkart();
        obj.payment();
        obj.payment(432112312312312312L);
        obj.payment("pavan@bdfc");
        obj.payment("pavan", "pavan123");
    }
}
```

Ex2
```
class Irctc
{
    void search(int trainNumber)
    {
        Sopln("Search with train No");
    }
    void search(String src, String dest)
    {
        Sopln("Search with Source & Dest");
    }
}
```

```
Irctc ref = new Irctc();
ref.search(15214);
ref.search("pnb", "gwd");
```

Ex3
```
class Facebook
{
    void login(String email, String pwd)
    {
        Sopln("login with email");
    }
    void login(long mobile, String pwd)
    {
        Sopln("login using mobileNo");
    }
}
```

```
Facebook fb = new Facebook();
fb.login("pas123@gmail.com", "pass123");
fb.login("9431363999", "pwd123");
```

Q1

```
class Mobile
{
    void unlock()
    {
    }
    void unlock(int pin)
    {
    }
    void unlock(face id)
    {
    }
    void unlock(pattern p)
    {
    }
    void unlock(FingerPrint fp)
    {
    }
}
```

```
class Whatsapp
{
    void send(String s)
    {
    }
    void send(Image a)
    {
    }
    void send(Video v)
    {
    }
    void send(Audio a)
    {
    }
```

Q2

```
class Demo
{
    static void disp()
    {
    }
    static void disp(int a)
    {
    }
}
```

```
class Sample
{
    private void run()
    {
    }
    private void run(int a)
    {
    }
}
```

```
class Example
{
    final void help()
    {
    }
    final void help(int a)
    {
    }
}
```

```
class Delta
{
    void test()
    {
    }
    int test()
    {
        return 10;
    }
}
```

CTE

```
Delta d = new Delta();
d.test();
```

METHOD OVERRIDING

```
class Demo
{  void test()
   {  Sop("Manual");
   }
}
class Sample extends Demo
{  @Override
   void test()
   {  Sop("Automation");
   }
}
public class Main1 {
   p s v m (S[] arg)
   {  Demo d = new Demo();
      d.test();     //Manual
      Sample s = new Sample();
      s.test();     //Automation
      Demo s1 = new Sample();  //Plasting
      s1.test();    // Automation
   }
}
```

```
class Father
{  void motorcycle()
   {  Sop("Splender");
   }
}
class Sanket extends Father
{  void motorcycle()
   {  Sop("Modified");
   }
}
```

B1

```
class WhatsApp1
{
    void deliveryReport()
    {
        Sopln("Sent");
    }
}
class WhatsApp2 extends WhatsApp1
{
    void deliveryReport()
    {
        Sopln("Sent-Delivered");
    }
}
class WhatsApp3 extends WhatsApp2
{
    @Override
    void deliveryReport()
    {
        Sopln("Sent-Delivered-seen");
    }
}
```

WhatsApp1 w1 = new WhatsApp1();
w1.deliveryReport()

* It is the process of sub class inheriting a method from the super class and changing the method defination while keeping the same method signature
* To perform method overriding Inheritance is mandatory
* We can use method overriding to upgrade existing software features
* We cannot override static, private and final methods
* Method overriding is used to achieve run-time polymorphism

- After overriding and upcasting we will always get latest implementation
- writing @Override before method overriding is optional but highly recommended

@Override → Annotation
It is use to inform java runtime. This is a modified version of a method which is present in super class

Has-A- Relationship

It is a form of association in which one class will act as a container for the object of an another class.

There are two forms of Has-A- Relationship
Composition Aggregation

- The main use of has a relationship is code reusibility

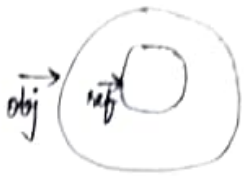
Composition is a strong form of associtation where the contend object can not exist without the container

Aggregation is a weak form of association in which the contend object can exist without the container

Ex

```
class Demo
{
    void test()
    {
    }
}
class Sample
{
    Demo ref = new Demo();
}
```

```
public class Main1
{
    p s v m(S[] a)
    {
        Sample obj = new Sample();
        obj.ref.test();
    }
}
```



Ex

```
class PhoneCamera
{
    void takePhoto
    {
    }
}
class SimCard
{
    void connectNetwork()
    {
    }
}
class Mobile
{
    PhoneCamera cam = new PhoneCamera(); // Composition --> Mobile has A-Came
    SimCard sim = new SimCard(); // Aggregation --> Mobile has A Simca
}
```

```
public class Main2
{
    p s v m(S[] args)
    {
        Mobile mob = new Mobile();
        mob.cam.takePhoto();
        mob.sim.connectNetwork();
    }
}
```

Ex

```
class Teacher{
    void teach(){
    }
}
class Department{
    Teacher tr = new Teacher();
}
class College
{
    Department dr = new Department();
}
```

```
public class Main3
{
    p s v m(String[] args)
    {
        College col = new College();
        col.dr.tr.teach();
    }
}
```

Ex

```
class Printer
{   void write()
    {

    }
    void write(int a)
    {

    }
}
class Computer
{   static Printer out = new Printer();   // object is neither static or non static

}
```

```
Computer.out.write();
Computer.out.write(10);
```

```
package java.io
public class PrintStream
{   void println()
    {

    }

}
```

```
package java.lang;
import java.io.PrintStream;
public class System
{   static PrintStream out = new PrintStream();

}
```

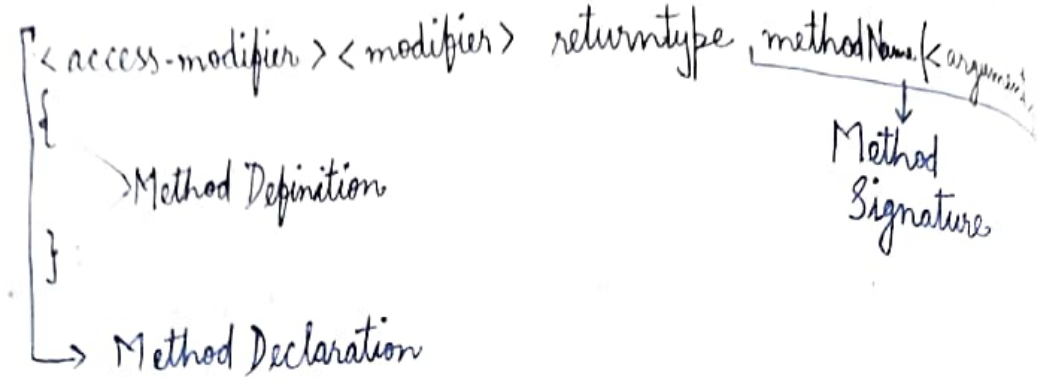
```
System.out.println();
```

- Inbuild class In Java Library
- static reference variable of PrintStream
- non-static method of PrintStream

Method Binding

It is the process of joining method declaration with method defination

The method binding activity is done by the JVM at run-time when the method gets called



POLYMORPHISM

polymorphism is the concept of one object showing different behaviours in its lifetime

In Java We have 2 type of polymorphism

i) Compile-Time Polymorphism

ii) Run time polymorphism

Compile-Time-Polymorphism

i) We can do Compile Time polymorphism using method overloading

- In compile time polymorphism the method binding decision are taken by compiler at compile time based on the arguments
- The decision taken by the compiler cannot be changed hence it is also known as static polymorphism or static binding
- The method binding decision are taken at the compile time before program starts execution hense it is known as early binding

```
class Whatsapp
{
    void send(String s)
    {
        Sop("Text Message");
    }
    void send(Audio a)
    {
        Sop("Audio Message");
    }
    void send(Image i)
    {
        Sop("Image Message");
    }
    void send(Video v)
    {
        Sop("Video Message");
    }
}
```

```
class Audio
{
}
class Image
{
}
class Video
{
}
```

```
public class Main {
    p  s  v  m(S[]a)
    {
        Whatsapp ref = new Whatsapp();
        ref.send("Hello");
        ref.send(new Audio());
        ref.send(new Image());
    } ref.send(new Video());
}
```

Run Time Polymorphism

We can do run time polymorphism using method overriding

In run time polymorphism the method binding decisions are taken by the JVM at run time based on the current instance

The decisions taken by the JVM can be changed hence it is also known as dynamic polymorphism or dynamic binding

The method binding decisions are taken by JVM at the run time during program execution hence it is known as late binding

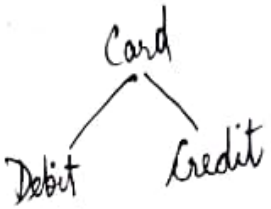
Compile Time Polymorphism	Run Time Polymorphism
i) Achieved using Method Overloading	Achieved using Method Overriding
ii) Method Binding Decision is taken by Compiler at @Compile time table	Method binding decision is taken by JVM at runtime
iii) Binding decision is based on arguments	Binding decision is based on current instance
iv) Binding decision cannot be changed hence it is static polymorphism	Binding decision can be changed hence it is dynamic polymorphism
v) Binding decision are made before program start executions hence it is called early binding	~~Achieved~~ Binding decision are made during program execution hence it is known as late binding

```
class Facebook
{  void display()
   {  Sopln("Display Response");
   }
}
class FacebookWeb extends Facebook
{  void display()
   {  Sopln("Display in Web");
   }
}
class FacebookMobile extends Facebook
{  @Override
   void display()
   {  Sopln("Display in Mobile");
   }
}
```

```
public class Main1
{  p  s  v  m(String[] args)
   {
      Facebook fb;
      fb = new FacebookWeb();
      fb.display();
      fb = new FacebookMobile();
      fb.display();
   }
}
```

Eg:

```
class Card
{  void swipe()
   {  Sopln("Please wait...");
   }
}
class DebitCard extends Card
{  @Override
   void swipe()
   {  Sopln("Account Bal. Remaining");
   }
}
class CreditCard extends Card
{  @Override
   void swipe()
   {  Sopln("Credit Due Inc");
   }
}
```




```
Card ref;
ref = new DebitCard();
ref.swipe();
ref = new CreditCard();
```

Encapsulation

Encapsulation is the concept of binding the relating data members and member function into a class body

Every class in java is an example for encapsulation because java syntax doesnot allow the programmer to declare datamembers and member functions outside a class

Note: The main goal of encapsulation is data security

For java class is an encapsulation of data members and member function, package is an encapsulation of java program

JAR file is an encapsulation of packages

Access - Modifier

Access modifies can be used by the programmes to control the visibility of members inside a class

In java the following access - Modifies are available

- *PRIVATE
- *<Default/Package>
- *PROTECTED
- *PUBLIC

Baby
Kid
Teen
Youth

```
package com.jspiders.encapsulation;
public class Foxtrot
{
    private static int a=10;
    static int b = 20;
    protected static int c=30;
    public static int d = 40;

    public static void main(S[] args)
    {
        Sopln(Foxtrot.a);
        Sopln(Foxtrot.b);
        Sopln(Foxtrot.c);
        Sopln(Foxtrot.d);
    }
}
```

```
package com.jspiders.encapsulation
public class Geronimo
{
    p   s   v   m(S[] a)
    {
        Sopln(Foxtrot.b);
        Sopln(Foxtrot.c);
        Sopln(Foxtrot.d);
    }
}
```

```
package com.jspiderss.intro;
import com.jspider.encapsulation.Foxtrot;
public class Flower extends
{
    p   s   v   m(
    {
        Sopln(Foxtrot.c);
        Sopln(Foxtrot.d);
    }
}
```

Private: Members with private access modifier can be accessed only inside the same class.

Default / Package: Members that are declared without an access modifier are said to have default or Package level access

Members with default or Package level access can be accessed in the same package

Protected: Members with protected access modifier can be access directly in the same package

Protected members can be accessed out side the package through inheritance

Public: Members with public access can be accessed directly from anywhere in the project

Ranking based on ~~V~~. Security and Visibility

Security	Visibility
private	public
<default/package>	protected
protected	<default/package>
public	private

```
package com.jspiders.encapsulation;
public class Karnataka{
    private static void test(){
        Sopln(" test()--- ");
    }
    static void disp(){
        Sopln(" disp()--- ");
    }
    protected static void play(){
        Sopln(" play--- ");
    }
    public static void send(){
        Sopln(" send--- ");
    }
    public static void main(String[]args){
        Karnataka.test()
        Karnataka.disp()
        Karnataka.play()
        Karnataka.send()
    }
}
```

```
package com.jspiders.encapsulation;
public class TamilNadu
{   public static void main(String[] args)
    {
        Karnataka.disp();
        Karnataka.play();
        Karnataka.~~play~~send();
    }
}
```

```
package com.jspiders.intro;
import com.jspiders.encapsulation.Karnataka;
public class Bihar extends Karnataka
{   public static void main(String[] args)
    {
        Karnataka.play();
        Karnataka.send();
    }
}
```

ACCESS-Modifiers on a class/interface/enum

A class/interface/enum can only be declared with access modifiers

i) public
ii) <default/package>

Overriding rules on Access-Modifiers

Whenever the subclass overrides a method of the super class it can retake the same access-Modifiers or declare an access-modifiers with higer visiability

super class:	<default/package>	protected	public	private
sub-class	<default/package> protected public	protected public	public	CTE we cannot inherit privates methods

Data Hiding

It is the concept of hiding sensitive informatio from the outside world by using private data membe

```
class Credentials
{
    String username; //not hidden
    private String password; //hidden
}
```

Tightly Encapsulated class

A java class in which all data members are declare with private access modifier is called as a tightly Encapsulated class

class Student → Tightly Encapsulated class
{ private long mobileNumber;
  private String emailId;
}

class Alpha → Tightly Encapsulated class
{ private int x;
  private int y;
}
class Beta extends Alpha
{ private int a;
  int b;
}
class Gamma extends Beta
{ private int q;
  private int p;
}

Java Bean Class / POJO (Plain Old Java Object) class

previous call as

Java Bean class can be used to achieve 100% / pure Encapsulation

Java Bean class objects are used to transfer data b/w programs / layers

Java Bean class objects are called DTO / Data Transfer objects)

Rules to create Java bean class

i) All data members must be private
ii) Access to private data members via public getter and setter methods
iii) No methods except getter and setter methods
iv) Java Bean class must have a default constructor
v) Must implement "java.io.Serializable"

Ex1

```
public class Person{
    private int age;
    public int getAge()
    {
        //Authorization Or Validation
        return age;
    }
    public void setAge(int age)
    {
        //Authorization & Validation
        this.age = age;
    }
}
```

```
public class Main1
{
    public static void main(String[] args)
    {
        Person p1 = new Person();
        p1.setAge(16);
        Sopln(p1.getAge());
    }
}
```

Ex2

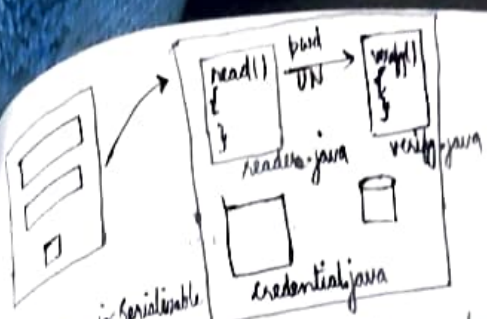
```
import java.io.Serializable → implements Serializable
public class Employee{
    private int eId;
    private double ctc;
    public int getEId(){
        return eId;
    }
    public void setEId(int eId){
        this.eId = eId;
    }
    public int getCtc(){
        return ctc;
    }
    public void setCtc(double ctc){
        this.ctc = ctc;
    }
}
```

```
public class Main2{
    public static void main(String[] args)
    {
        Employee e1 = new Employee();
        e1.setEId(101);
        e1.setCtc(4.5);
        Sopln("EID:" + e1.getEId());
        Sopln("CTC:" + e1.getCtc());
    }
}
```

Ex3

```
public class Account{
    private long accNum;
    private double accBal;
    public long getAccNum(){
        return accNum;
    }
    public void setAccNum(long accNum){
        this.accNum = accNum;
    }
    public double getAccBal(){
        return accBal;
    }
    public void setAccBal(double accBal){
        this.accBal = accBal;
    }
}
```

```
public class Main3{
    public static void main(String[] args)
    {
        Account acc = new Account();
        acc.setAccNum(6532147691 2L);
        acc.setAccBal(45000.0);
        Sopln("Account Number:" +
                acc.getAccNum());
        Sopln("Account Balance:" +
                acc.getAccBal());
    }
}
```



import java.io.Serializable

public class Credentials implements Serializable

```
{   private String username;
    private String password;
    public String getUsername()
    {   return username;
    }
    public void setUsername()
    {   this.username = username;
    }
    public String getPassword()
    {   return password;
    }
    public void setPassword()
    {   this.password = password;
    }
}
```

```
Credentials c = new Credentials();
c.setUsername("admin");
c.setPassword("admin123");
```

→ Reader.java

```
Sopln(c.getUsername());
Sopln(c.getPassword());
```

→ Verify.java

ABSTRACTION

- In General, Abstraction is concept of Hiding Implementation Details and exposing functionality to end user
- In Java, Abstraction is concept of hiding Method Defination and exposing method declaration
- In Java we can do abstraction using
  - i) Abstract method
  - ii) Abstract class
  - iii) Interface

→ Gun
→ Licience holder
→ Military

Abstract method + Abstract class → Partial Abstraction (→ concrete → default → object class → method inherit)

Abstract method + Interface → pure/100% Abstraction

Ex:
```
class Whatsapp {
    void send (String msg)
    {

    }
}
```
(Definition / expose)

Abstract Methods

i) Abstract methods can be declared using the keyword abstract
ii) we can develop abstract methods inside an abstract class or interface
iii) abstract methods can't have a method definition
iv) abstract methods cannot be static, final and private

6/
```
abstract class Demo
{   abstract void test();
    abstract void disp();
}
class Sample extends Demo        // concrete class
{   @Override
    void test(){
        Sout("Executing test");
    }
    @Override
    void disp(){
        Sopln("Executing disp");
    }
}
public class Main1
{   p  s  v  m (String[] args){
        Demo ref = new Sample();
        ref.test();
        ref.disp();
    }
}
```

→ is a partial abstract class

7/
```
abstract class Alpha{
    abstract void play();
    void send(){
        Sopln("Executing send");
    }
}
class Beta extends Alpha
{
    @Override
    void play(){
        Sopln("Executing send2");
    }
}
```

```
Alpha a = new Beta();
a.send();
a.play();
```

```
abstract class Delta{
    void run(){
        Sopln("Execute Run");
    }
    void push(){
        Sopln("Execute Push");
    }
}
class Gramma extends Delta{

}
```

```
Delta d = new Gramma();
d.run();
d.push();
```

Ex

```
abstract class Hotstar
{   void login(){
①   }
    abstract void view();
}
```

```
class HotstarFree extends Hotstar{
    @Override
②   void view(){
        Sopln("Ad, Trailer");
    }
}
class HotStarVip extends Hotstar{
    @Override
②   void view(){
        Sopln("Ad, Regional Content");
    }
}
class HotStarPremium extends Hotstar{
    @Override
②   void view(){
        Sopln("No Ad, All Access");
    }
}
```

```
public class Main4{
    public static void main(String arg)
    {
        Hotstar hs;
        hs = new HotstarFree();
③       hs.login();
        hs.view();
        hs = new HotstarPremium();
        hs.login();
        hs.view();
    }
}
```

Ex

```
abstract class Android
{   void call(){

    }
①   void message(){

    }
    abstract void ui();
}
```

```
class Samsung extends Android{
    @Override
    void ui(){
        Sopln("GalaxyOS");
    }
}
class Redmi extends Android{
    @Override
②   void ui(){
        Sopln("MIUI");
    }
}
class OnePlus extends Android{
    @Override
    void ui(){
        Sopln("OxygenOS");
    }
}
```

Main

```
Android a;
a = new Samsung();
③ a.ui();
a = new Redmi();
a.ui();
```

Abstract class:

- Abstract class is created using abstract keyword
- Inside an abstract class there is no restriction on the data member and member function
- We can't create an object for abstract class
- We can access the properties of an abstract class through inheritance and over-ridding
- Abstract class is considered as a partial abstract body

Interface

```
interface InterfaceName
{
    data members  ⟶ public static final
    member function ⟶ public abstract
}
```

class ↑extends class interface ↑implements class

interface ↑extends interface class ↑X (CTE) interface

Ex:

```
interface Demo{
    void test();
}
class Sample implements Demo{
    @Override
    public void test(){
        Sopln("Execute test---");
    }
}
public class Main1{
    p  s  v  main(String[] args)
    {
        Demo ref = new Sample();
        ref.test();
    }
}
```

```
interface delta{
    void play();
    void disp();
}
class Example implements delta{
    @Override
    public play(){
        Sopln(" play");
    }
    @Override
    public disp(){
        Sopln("disp");
    }
}
```

Ex

```
interface Alpha{
    void test();
}
interface Beta{
    void send();
}
class Grama implements Alpha, Beta{
    @Override
    public void send(){
        Sopln("Send");
    }
    @Override
    public void test(){
        Sopln("Test");
    }
}
```

```
public class Main3{
    public static void main(String[] args)
    {
        Grama ref = new Grama();
        ref.send();
        ref.test();
    }
}
```

Ex

```
interface Apollo{
    void run();
}
interface Bravo extends Apollo{
    void push();
}
class Discovery implements Bravo{
    @Override
    public void run(){
        Sopln("Run");
    }
    public void push(){
        Sopln("Push");
    }
}
```

main4

```
Discovery d = new Discovery();
d.run();
d.push();
```

Ex

```
interface Shape
{
    void draw();
}
class Circle implements Shape{
    public void draw(){
        Sopln("CIRCLE");
    }
}
class Triangle implements Shape{
    public void draw(){
        Sopln("TRIANGLE");
    }
}
class Square implements Shape{
    public void draw(){
        Sopln("SQUARE");
    }
}
class ShapeToolkit
{
    static void drawShape(Shape s)
    {
        if(s != null)
        {
            s.draw();
        }
    }
}
```

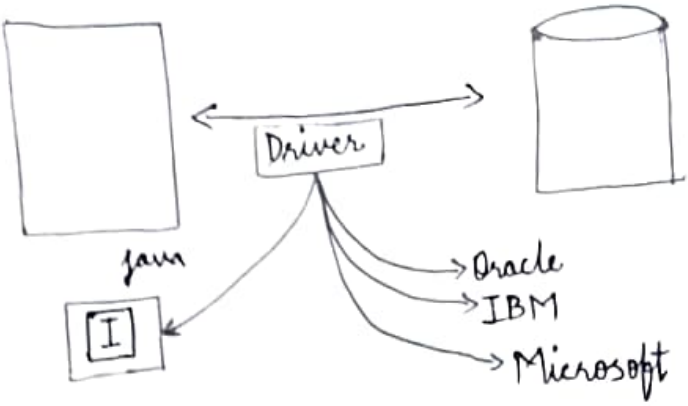
main5

```
Circle c = new Circle();
ShapeToolkit.drawShape(c);
Square s = new Square();
ShapeToolkit.drawShape(s);
```

```
interface Translator
{ void translate();
}
class KannadaTranslator implements Translator
{ @Override
  public void translate(){
      Sopln("translate to Kannada");
  }
}
class HindiTranslator implements Translator
{   @Override
    public void translate(){
        Sopln("translate to Hindi");
    }
}
class FrenchTranslator implements Translator
{
    @Override
    public void translate(){
        Sopln("translate to French");
    }
}
class Google_Translator
{   static void translateTo(Translator t)
    {
        if(t!=null)
        {
            t.translate();
        }
    }
}
```

main {

```
KannadaTranslator kt = new KannadaTranslator();
HindiTranslator ht = new HindiTranslator();
FrenchTranslator ft = new FrenchTranslator();
Google_Translator.translateTo(kt);
Google_Translator.translateTo(ht);
Google_Translator.translateTo(ft);
```




```
interface Driver
{
    void connect();
}

class OracleDriver implements Driver
{
    @Override
    public void connect(){
        Sopln("Connected to Oracle");
    }
}

class MicrosoftDriver implements Driver
{
    @Override
    public void connect(){
        Sopln("Connected to MicrosoftSQLServer");
    }
}

class DriverManager
{
    static void registerDriver(Driver ref)
    {
        if(ref!=null)
        {
            ref.connect();
        }
    }
}

class DB2 implements Driver
{
    @Override
    public void connect(){
        Sopln("Connected to DB2");
    }
}

public class MainByProgrammer
{
    public static void main(String[] args)
    {
        OracleDriver ref = new OracleDriver();
        DriverManager.registerDriver(ref);
        MicrosoftDriver ref1= new MicrosoftDriver();
        DriverManager.registerDriver(ref1);
    }
}
```

→ An interface is declared using the keyword interface
→ Inside an interface all data members are public, static and final
→ Inside an interface all member function are public & abstract
→ We can not create an Object for interface directly
→ We can access properties of an interface by creating an implementation class and overriding the abstract methods
→ We can create an implementation class for an interface by using implements keyword
→ A single class can provide implementation for multiple interfaces directly

ABSTRACT CLASS

- created with abstract class keyword
- no restriction on data members
- no restriction on member functions
- we can inherit with extents keyword
- we cannot do multiple inheritance
- partial abstract body
- will contain constructor

INTERFACE CLASS

created with interface keywords
all data members are public, static & final
all member functions are public & abstract
we can implement with implements keyword
we can do multiple inheritance
100% or pure Abstract Body
will not have a constructor

Note: When a concrete class provides implementation for two separate interface which is having method with the same declaration it is enough to override ones

```
interface A
{
  void test();
}
interface B
{
  void test();
}
```

```
class C implements A, B
{
    @Override
    public void test(){

    }
}
```

Note

It is possible for a class to simultaneously inherit from a class and interface

```
class A                          class C extends A implements B
{                                {
}
class B interface B              }
    {
    }
```

Note

When a class is inheriting from an another class and interface at the same time If the super class contain the concrete methods matching the abstract method in the interface It becomes optional to override the abstract method

```
class A                          class B
{   public void run()            {
    {                                public void
    }                            interface B
}                                {   void run();
                                 }

class C extends A implements B
{

}
```

FUNCTIONAL INTERFACE

Any interface in java that contains only one abstract method is called as a functional interface
A functional interface should be generally preceded with the annotation "@FunctionalInterface"
Functional interface was previously known as SAM type interface (Single Abstract Method)

```
@FunctionalInterface
interface Playable {
    void play();
}
```

```
interface Delta
{   void play();
}
```

MARKER INTERFACE

Marker Interfaces are special type of interface which has contains an empty body

The implementation for these interfaces are provided internally by the jvm

Ex: java.lang.Cloneable, java.io.Serializable, java.util.RandomAccess etc

Java8 Enhancements for Interfaces

In java8 to support functional programing several/2 major changes where made in the syntax of interface

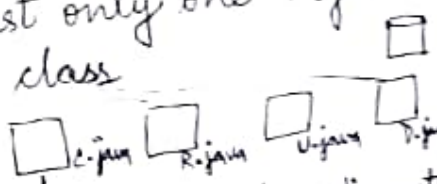
i) Interface support static concrete methods
ii) Interface support default concrete methods

```
interface Ex
{   static void play(){
    }
    default void test(){
    }
}
```

SINGLETON CLASS

A java class for which there exist only one object in the memory is called as singleton class

```
class Connection
{   private static Connection con = null;
    public static Connection getConnection()
    {   if (con == null){
            con = new Connection();
        }
        return con;
    }
    private Connection(){
    }
}
```



Connection c1 = Connection.getConnection();

Connection c2 = Connection.getConnection();

Connection c3 = Connection.getConnection();

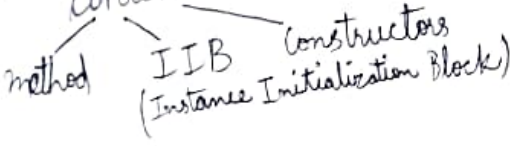
Connection c4 = Connection.getConnection();

Advantages of Singleton class

Saves memory
enforces consistancy in object behaviour

this Keyword

It is used to refer to the current instance of a class
this Keyword can be used to access the non static ~~data~~ members of the ~~cu~~ current class
We can use this keyword to differenciate between locale variable and data member of current class
this keyword can only be used inside a non static context



Ex

```
class Utility{
    void test(){
        Sopln(this);
    }
}
public class Mainclass1{
    p  s  v  m  (S  args)
    {
        Utility u1 = new Utility();
        Sopln(u1);
        u1.test();
        Sopln("-----");
        Utility u2 = new Utility();
        Sopln(u2);
        u2.test();
    }
}
```

```
class Geronimo {
    int num = 25;
    void test() {
        Sopln("Executing test()");
    }
    void disp() {
        Sopln("this.num");
        this.test();
    }
}
public class Mainclass2 {
    p
    {
        Geronimo g = new Geronimo();
        g.disp();
    }
}
```

Ex

```
class Student {
    String name;
    int id;
    Student(String name, int id) {
        this.name = name;
        this.id = id;
    }
}
class Hello {
    int a = 10;
    void test() {
        int a = 20;
        Sopln(a);
        Sopln(this.a);
    }
}
```

Main

```
Student s1 = new Student("Pa", 10);
Hello h = new Hello();
h.test();
```

super Keyword

super keyword is used to access the non-static members of the super class

super keyword can only be written inside a non-static function

```
class Demo{
   int val = 25;
}
class Sample extends Demo{
    int val = 50;
    void play(){                    int val = 75;
        Sop(super.val);   // 25
        Sop(this.val);    // 50
        Sop(val);         // 75
    }
}
```

MainClass

```
Sample s = new Sample();
s.play();
```

```
class SoftwareV1{
    void login(){
        Sop("Login with uname & pwd");
    }
}
class SoftwareV2 extends SoftwareV1{
    @Override
    void login(){
        super.login();
        Sop("Captcha Validation");
    }
}
```

Main class

```
SoftwareV1 v1 = new SoftwareV1();
v1.login();

SoftwareV2 v2 = new SoftwareV2();
v2.login();
```

o/p → login with uname & pass
login ... ...
Captcha Validation

JAR FILE

JAR stand for Java Archive

JAR file is used to transport Java Programs

The Jar file contains compressed ".class" files

All JAR files will have file extension ".jar"

Java Library

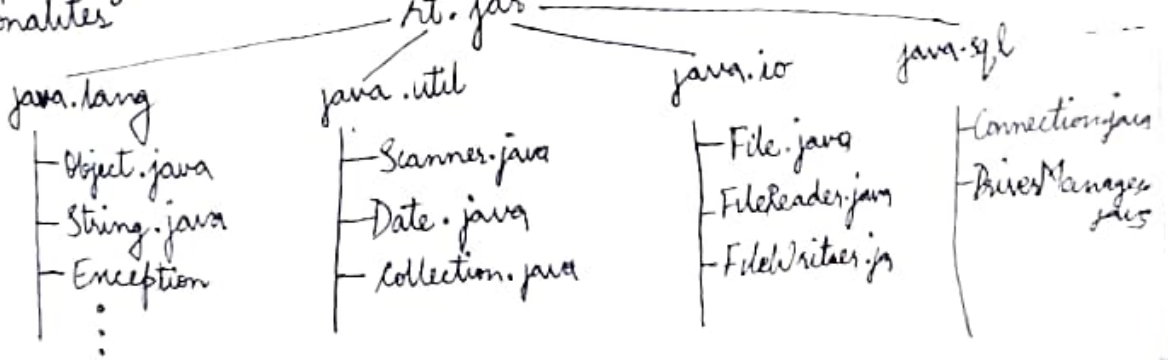
A library is a group of predefined programs that will make programmers life easy
Libraries are classified into 2 types
i) Internal libraries ii) External libraries

Internal library: It is provided by the language creaters ex: core Java libraries, ...
It is provided a part of the language installer (jdk)
To makes use of the programs present in the internal library there is no need of additional configuration

External library: The external libraries are provided by 3rd parties vendors
It is not a part of language installer and has to be configured or downloaded separately
to make use of the programs present in the external libraries additional configuration is required

Note
→ The external jar/library that we add to projects build path is called as dependencies

Note
→ **rt.jar:** It is a jar file that is present in java language that contains all the inbuild programs of the core java library
→ rt.jar file is organised in the form of packages
→ Each package contains a group of programs with similar functionalites



Package: A package is a location within a project or jar file which contains programs having similar functionalities

banking project
- JRE System library
- src
  - com.citybank.login
    - login.java
    - CreateAccount.java
    - ForgotPassword.java
  - com.citybank.transactions
    - UpiTransaction.java
    - AtmTransaction.java
- Referencelibraries
  - Spring.jar

"java.lang" Package

It is a package in the core java library that is present inside rt.jar file

It contains several programs that are related to language fundamentals

The programs persent in java.lang package can be directly accessed from anywhere in java programming

Lambda Exp

Objective of lambda Expr

i) To introduce functional programming features
ii) code optimization
iii) lambda Expression is an Anonymous function

Note
Anonymous function is a function
- which doesn't have name
- donot have return type
- donot have access specifies

Lambda Expn is introduced in Java 1.8

Normal function

```
public void m()
{   Sopln("      ");
}
public void sum(int n1,int n2)
{   Sopln(n1+n2);
}
```

lambda Expression

```
() -> { // statements }

(int n1,int n2) -> { Sopln(n1+n2); }
(n2,n2) -> Sopln(n1+n2);
```

It is not mandatory to specify the data type for parameters
If we not specify compiler will only add the datatype (consider)

```
public int square (int n)
{   return n*n;
}
```

```
(int n) -> { return (n*n); }
(n) -> (n*n);
n -> (n*n);
```

It is mandatory to write return statements inside the curly br {}

Functional Interface. Any Interface which has only 1 abstract method but can have any no. of static and default methods.

Ex Runnable. = run() Callable. = call();
Comparable. compare.To()

Note
Lambda Exp^n are used to give the implementation to the abstract method present in a functional interface.

Q. WAJP to find the product of 2 nos. using lambda exp^n

Q. WAJP to find the cube of a no. using lambda exp^n

Object CLASS

→ It is an Inbuild class that is present in java library
→ It is present inside java.lang package
→ It is the supermost class in java that is every class in java is either directly or indirectly a child of object class
→ Object class defines several common methods needed by all java programs

* toString()
* hashCode()
* equals()
* notify()
* notifyAll()
* finalize()
* clone()
* getClass()
* wait(), wait(long), wait(long, int)

toString()

public String toString() → MDeclaration

* The default implementation of toString() provides String representation of the object
* The String representation contains 3 elements
  a) FullyQualifiedClassname
  b) @
  c) Hexadecimal Pseudo Address
* If nesseccary the programer can override toString method to return string value of there own choice
* Whenever the programer prints reference variable directly the JVM internally calls toString method

```
package com.jsiders.example
class Student1
{
    String name = "Ram";
}
class Student2
{
    String name = "Ram";
    @Override
    public String toString()
    {
        return "S.Name" + name;
    }
}
public class Main1{
    p  s  v  m (  ){
        Student s1 = new Student1();
        Sopln(s1);  // Sopln(s1.toString());
        Student s2 = new Student2()
        Sopln(s2);
    }
}
```

toString() → with overriding
→ without overriding

hashCode()

public int hashCode() → method Declaration

- The default implementation of hashCode Method returns the unique integer id given by the JVM
- The unique integer id given by the JVM is decimal representation of the hexadecimal pseudo address
- If nessecery the programer can override hashCode() method to return an integer value suited to their requirements

```
package com.jspiders.example;
class Employee1{
  int E empId = 101;
}
class Employee2{
  int empId = 101;
  @Overriding
  public int hashCode(){
    return empId;
  }
}
public class Main2{
  p s v m(--){
    Employee1 e1 = new Employee2();
    Sop(e1.hashCode());
    Employee2 e2 = new Employee2();
    Sop(e2.hashCode());
  }
}
```

equals()

public boolean equals(Object obj)

The default implementation of equals method will work in the same ways as == double equals operator

If the programmer wants to do containt comparision b/w objects it is neccasary to override equals() and provide required implementation

Ex:

```
public class Main{
  p s v m(--){
    Circle c1 = new Circle();
    Circle c2 = new Circle();
    Sop(c1.equal(c2));  → false
    Sop(c1.radius == c2.radius); → true
    Sop(c1 == c2);  → false
  }
}
```

```
class Circle
{  int radius = 5;
}
```

Ex

```
package com.jsfiders.examples;
class Circle
{
    private int radius = 5;
    @Override
    public boolean equals(Object obj) {
        Circle c = (Circle)obj;
        if (this.radius == c.radius){
        }    return true;
        return false;
    }
}
```

Main

```
Circle c1 = new Circle();
Circle c2 = new Circle();
Sopln(c1.equals(c2));
```

Ex

```
package com.jsiders.examples;
class OneTimePassword{
  private int otp = 4545;
  @Override
  public boolean equals(Object obj){
      OneTimePassword otp = (OneTimePassword)obj;
      if(this.otp == otp.otp){
          return true;
      }
      return false;
  }}
```

Main

```
OneTimePassword otp1 = new OneTimePassword();
OneTimePassword otp2 = new OneTimePassword();
Sopln(otp1.equals(otp2));
```

Ex

```
class Account{
  private long accountNumber = 350017411008461;
  private long aaddharNumber = 123412341234L;
  @Override
  public boolean equals(Objects obj){
     Account acc = (Account) obj;
     if(this.accountNumber == accountNumber && this.aadharNumber == acc.aadharNumber
        return true;
     }
     else{
     } return false;
  }
```

Note
Whenever we override hashcode method of object class it is expected that we will also override the equals method

wait(), wait(long), wait(long, int)

public final void wait()

public final void wait(long arg)

public final void wait(long arg1, int arg2)

The wait method is used internally by the jvm to pause program execution till the occurrance of an event or for a certain time interval

The wait method is overloaded in three versions

All three versions of wait method are final and cannot be overridden

E.g. -> higher priority program call/Alarm 23

Scanner sc input -> ①

notify()

public final void notify()

The jvm internally uses notify method to inform a particular thread about the occurrance of events

notifyAll()

public final void notifyAll()

The jvm internally uses notifyAll method to inform all the threads about the occurance of an event

Garbage Collector

It is a thread based machanism that monitors all object present in heap / keeps track of objects in heap area
Garbage collector is responsible for destroying the unwanted objects
Befor the garbage collector destroys an object it will internally call the user finalize method
The programmer can explicitly call the garbage collector by writing the java statement "System.gc()"

finalize()

protected void finalize() throws Throwable
It is a protected method of object class that contains cleanup or closure code
The code present in finalize method will be executed by the garbage collector before destroying an object

Ex

```
class Person{
  @Override
  protected void finalize() throws Throwable
  {
     Sopln("Tata Bye Bye");
  }
}
public class Main1{
   p
      Person p=new Person();
      System.gc();
      Sopln("-----");
      p=null
      System.gc();
   }
}
```

Ex

```
class Connection{
  @Override
  protected void finalize() throws Throwable
  {
     Sopln("Close all resources");
  }
}
Main class
Connection con = new Connection();
System.gc();
con=null;
System.gc();
```

clone()

protected Object clone() throws CloneNotSupportedException

clone method of object class is used to perform object cloning that is creating an exact replica of the existing object

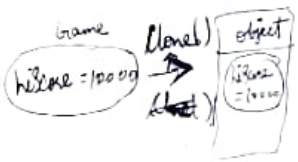
points to remember when using clone method

i) clone method is risky and may cause CloneNotSupportedException

ii) clone method can perform object cloning only if when the objects are clonable type objects

iii) whenever clone method performs object cloning the created copy will be internally upcasted to object class, hence it is mandatory to perform downcasting

```
public class Game implements Cloneable {          (2)
    int hiScore;
    public static void main(String[] args) throws CloneNotSupportedException   (1)
    {
        Game ref = new Game();
        ref.hiScore = 10000;
        Game copy = (Game) ref.clone();   (3)
        Sopln(copy.hiScore);
    }
}
```




```
public class Square implements Cloneable {
    int side;
    public static void main(String[] args) throws CloneNotSupportedException {
        Square orignal = new Square();
        orignal.side = 105;
        Square copy = (Square) orignal.clone();
        Sopln(copy.side);
    }
}
```

```
public class Data implements Cloneable{
double size;
public static void main(String[] args) throws CloneNotSupportedException {
    Data obj = 4.5;
    Data ref = (Data) obj.clone();
    Sopln(ref.size);
}
}
```

Cloneable type Object: Any object that is created from a class which implements Cloneable marker interface is known as cloneable type object

```
class Demo
{
}
Demo ref = new Demo();
// cloneable not
```

```
class Demo implements Cloneable
{
}
Demo ref = new Demo();
// cloneable
```

- Cloning in general is a restricted process and is discouraged

getClass()

public final Class getClass()

getClass method returns class Class representation of the current instance that is the fully qualified class name.

```
package com.jspiders.examples
import java.util.Scanner;
class Demo
{
}
class Sample extends Demo
{
}
public class Main1{
    p s v m(___){
        Object o1 = new Demo();
        Sop(o1.getClass());
        Object o2 = new Sample();
        Sop(o2.getClass());
        Object o3 = new Scanner(System.in);
        Sop(o3.getClass());
        Object o4 = new Object();
        Sop(o4.getClass());
    }
}
```

	Compile Time Error	Exception	Run Time Error
Identified by	Javac	JVM	JVM
Cause	Syntax Mistakes	Logic Mistakes	System Issues
Overcome	Code Rework	Exception handling	Not a programmer Issue
Examples	; expected Cannot Find Symbol Incompatible Types	ClassCastException NullPointerException CloneNotSupportedException	VM Out of Memory StackOverflowError LinkageError
	Error	Exception	

Exception

Exceptions are unexpected events that occurs at the runtime and causes the program to terminate abnormally

In Java Exception are classified into two types
i) Checked Exception ii) Unchecked Exception

Checked Exception

These exceptions are identified by the compiler at compile time. It is mandatory to handle checked exception

Ex: ArithmeticException, NullPointerException
↓ ClassCastException, etc...

Unchecked Exception

These exceptions are not identified by the compiler at a compile time. It is not mandatory to handle unchecked exceptions

Ex CloneNotSupportedException, IOException
SQLException, etc... Checked Exception

Checked Exception

- Identified by compile at the compile time
- Mandatory to handle

Ex IOException
InterruptedException
CloneNotSupportedException

Unchecked Exception

Not identified by the compiler at compile time
Not mandatory to handle

Ex ArithmeticException
NullPointerException
ClassCastException

Note: Both checked and unchecked exception, will always occurs at runtime

Note

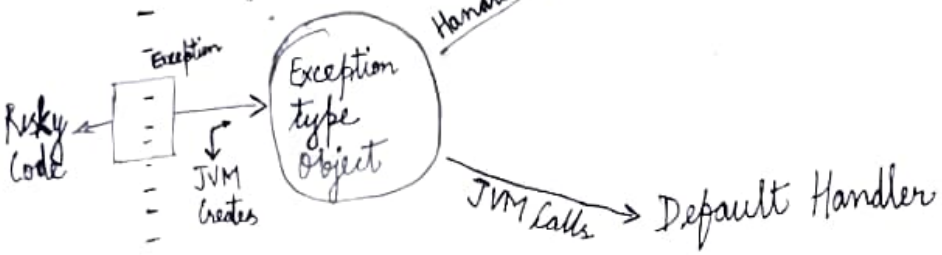
Whenever an exception occurs in the program, the JVM internally creates an exception type object, to represent the exception event

Once the exception object is created, the JVM excepts the programmer to handle the exception

If the programmer fails to handle the exception the JVM calls default handler

Ex

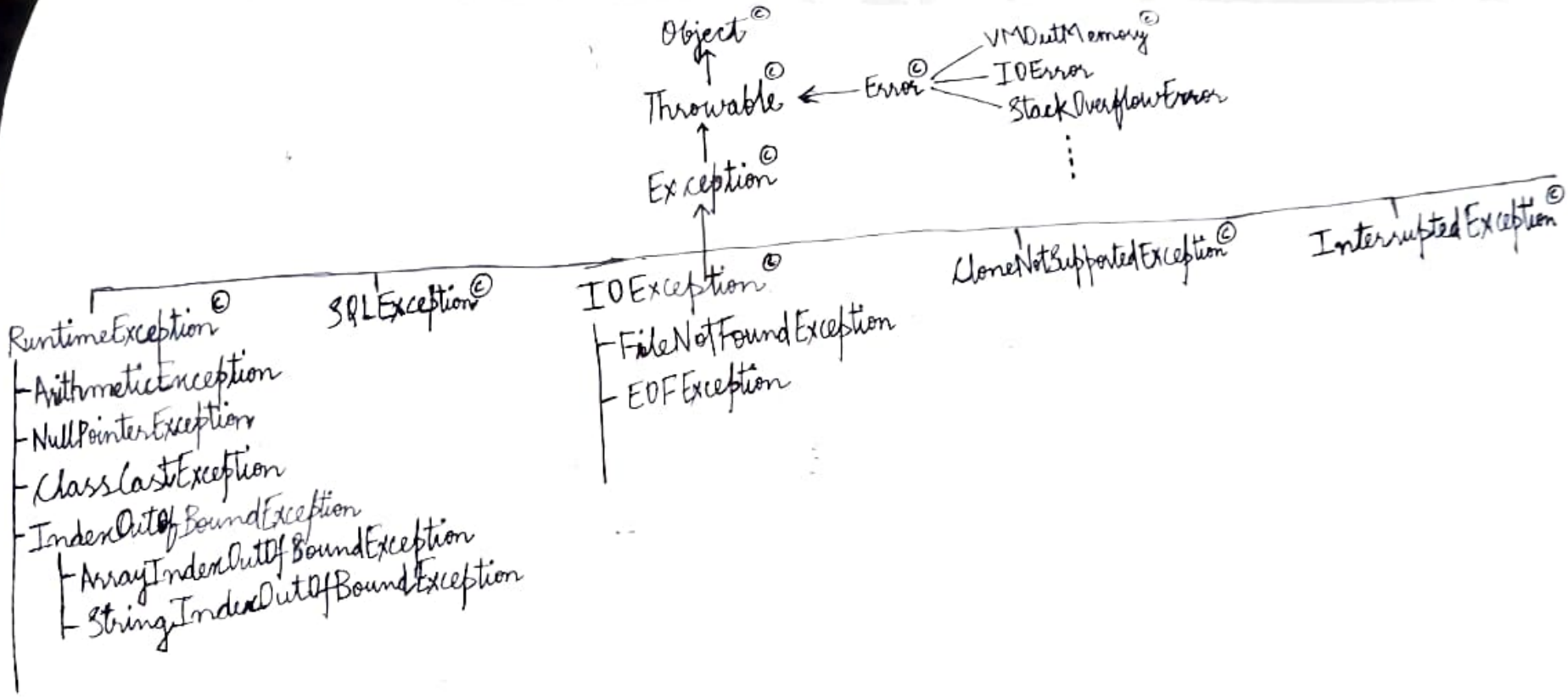
```
class Example{
    -
    -
    -
    -
    -
}
```



Ex

```
p s v m(_){
  Sopln("Program starts");
  try
  {
    Sopln(10/0);
  }
  Catch(ArithmeticException e)
  {
    Sopln("/ By zero");
  }
  Sopln("Program Ends");
}
```

```
public class Square{
  int side;
  p s v m(_){
    Sopln("Program Starts");
    Square original = new Square();
    original.side = 10;
    try
    {
      Square copy = (Square) original.clone();
      Sopln(copy.side);
    }
    Catch(CloneNotSupportedException e)
    {
      Sopln("Clone Operation failed");
    }
    Sopln("Program Ends");
  }
}
```



Exception Hierarchy: There is an inbuild class in the Java library for an each and every exception.

© → Concrete class
(Ac) → Abstract Class
(I) → Interface
(E) → enum

Ex1
```
public class Main1 {
    p s v m(—){
    String str = "Java";
    try{
        Sopln(str.charAt(5));
    }
    Catch (NullPointerException e){
        Sopln("Access with Null Reference");
    }
    Catch (StringIndexOutOfBoundException e){
        Sopln("Invalid Index Access");
    }
}
```

* try can have multiple catch block, but catch block can have only one try block

Ex2
```
Public class Main2{
    p s v m(—){
    try {
      Hello h = (Hello)Class.forName("com.ijspiders.exception.Hello").newInstance();
      h.greeting();
    }
    catch(ClassNotFoundException e){
        Sopln("No Such class Exists
    }
    catch(IllegalAccessException e){
        Sopln("Access Denied");
    }
    catch(InstantiationException e){
        Sopln("Instatiation failed");
    }
}
```

* Deprecated → It means, this is supported now, but in future it can be removed

```
class Hello {
    Void greeting(){
        Sopln("Hello Everyone");
    }
}
```

Ex3
```
public class Main3{
    p s v m(—){
    try {
        Sopln("try starts");
        Sopln(10/0);
        Sopln("try ends");
    }
    catch(ArithmeticException e){
        Sopln("/ By zero");
    }
    finally {
        Sopln("Biryani");
    }
}
```

finally block will run irrespective of exception means it will surely executed

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class Main4{
    p s v m(—){
        Scanner s=new Scanner(System.in);
        Sopln("Enter Value");
        try{
          int val= s.nextInt();
          Sopln("value:" +val);
        }
        catch(InputMismatchException e){
            Sopln("Invalid Input Entered");
        }
        finally{
            Sopln s.close();
            Sopln("Scanner closed");
        }
    }
}
```

Exception handling with try-catch-finally

Try-Block

The try block contains risky code (code that may cause exception)
Every try block should be followed by one-catch block.
Control shifts from try block to catch block as soon as the exception occur
A single try-block can be followed by multiple catch blocks

Catch-block

It contains recovery code (code to be executed on occurance of exception)
catch-block is written after try-block
catch block takes an object of exception type as argument
catch block executes when there is an exception event

finally block

It is an optional block used in exception handling
code inside finally block will executes irrespective of exception
The finally code generally contains cleanup/ closure code

Different possible combination of try catch & finally

```
try{
}
catch(  ){
}
```

```
try{
}
catch(  ){
}
catch(  ){
}
```

```
try{
}
catch(  ){
}
finally{
}
```

```
try{
}
catch(  ){
}
catch(  ){
}
finally{
}
```

```
try{
  try{
  }
  catch(  ){
  }
}
catch(  ){
}
finally{
}
```

```
try{
  try{
  }
  catch(  ){
  }
}
catch(  ){
  try{
  }
  catch(  ){
  }
}
```

```
try{
}
finally{
}
```

→ Allowed but not recommended

Default Handler

The default handler is an inbuilt mechanism, i.e (that is) called by the JVM, when programmer fails to handle an exception
The default handler, will perform the following tasks

i) It will stop program execution
ii) perform stack trace
iii) Display the details
  - → Name of the exception
  - → Message (If avaible)
  - → stack trace info

Eg

```
public class Main{
    p s v m(--){
        Sopln("main starts");
        dis();
        Sopln("main ends");
    }
```

```
static void test(){
    Sopln("test start");
    Sopln(10/0);
    Sopln("test end");
```

Creating Custom Exceptions

It is possible for the programmer to declare and define customs Exception in Java. to suit their requirements.

To create a custom exception the programmer has to follow 3 step

i) create a class that extends exception (checked) / Runtime exception Unchecked
ii) Develop Triger logic to cause the exception event
iii) Handle the exception

```
class CustomException extends Exception
{

}
class Trigger
{
    static void verify(int num) throws CustomException
    {
        if (num%5 ==0){
            throw new CustomException();
        }
    }
}
public class Main1{
    p s v m (__){
        try{
            Trigger.verify(24);
            Trigger.verify(25);
        }
        catch (CustomException e){
            e.printStackTrace();
        }
    }
}
```

InvalidMobileNumberException — Ex — CustomException → OverSpeedException
MobileNumberVerifier ← Trigger → HighwayPatrol
verifyMobileNumber ← verify → checkSpeed
(String mobile) (int num) (int speed)
if(mobile.length()!=10) if(speed>80)

String Class

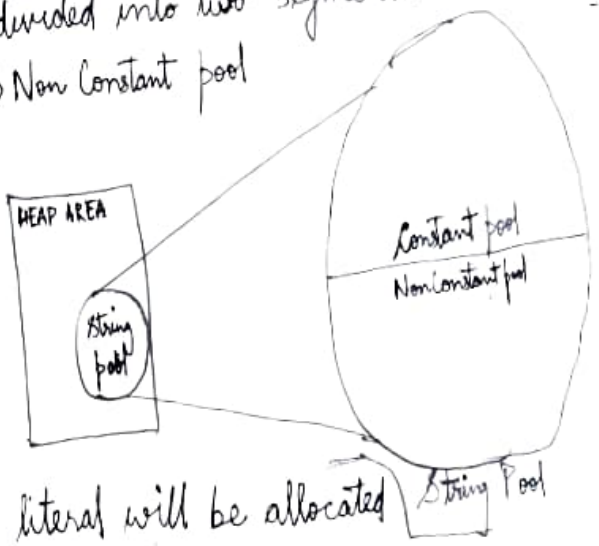
It is an inbuild class present in java.lang package
String class is used by programmer to represent a sequence of characters in java program
It is a final class and can not have a sub class
In java we can create String class objects in 2 ways
i) String literals ii) String with new Keywords

```
String s1 = "Java"        ⟶ by literal ⟶ 1st object created
String s2 = new String("Java"); ⟶ by new keywords ⟶ 2nd object created
Sopln(s1);
Sopln(s2);
```

Note: String class objects is created in both when we use literal and new Keyword approach

Memory Managment with respect to String Objects

All String objects are allocated memory in special part of the known as the string pool
The String pool is internally divided into two segments
i) constant pool ii) Non Constant pool



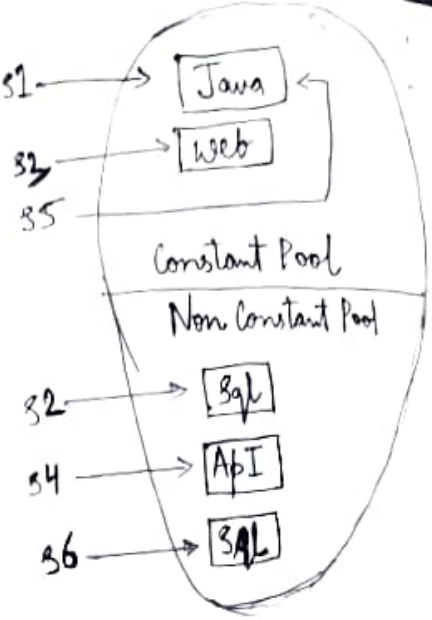
Constant pool

All String objects created as literal will be allocated memory in constant pool
before allocating memory in the constant pool the JVM checks for duplicate

Non Constant pool

All String objects created with new key word are allocated memory in the non constant pool
The JVM doesnot check for duplicates before allocating memory in the Non constant pool

```
String s1 = "Java";
String s2 = new String("Sql");
String s3 = "web";
String s4 = new String("API");
String s5 = "Java";
String s6 = new String("Sql")
```



Overridden methods of object class in String class

- String class overrides the following 3 methods of object class
  - i) toString()
  - ii) hashCode()
  - iii) equals()

The toString method is overridden to return the String value present in the current object

The hashCode method is overridden to return UNICODE value of the current String characters

The equals method is overriden to compare contents b/w two String objects

```
String s1 = "Jspider";
Sopln(s1.toString());      // Jspider
String s2 = "A";
Sopln(s2.hashCode());      // 65                    // "AA" → unicode value after hashing
String s3 = "Java";
String s4 = new String("Java");
Sopln(s3.equals(s4));      // true
```

String classes inheriting all the methods of 8 Object class but overriding only toString(), hashCode() and equals()

String class Objects are comparable type objects.
String class implements the functional interface Comparable.
String class provides implementation for the compareTo()
method present in comparable interface.
using the compareTo method the programmer can perform
a lexicographical comparision between String Objects.
The compareTo method returns an integer value based on which
decisions can be taken

Note
If the result is positive integer then current String should
come after the ~~given~~ given String
If the result is a negative integer the current String should
come before the given String
If the result is zero, current String and given String are same

java.lang.Comparable (I)
↓
int compareTo()

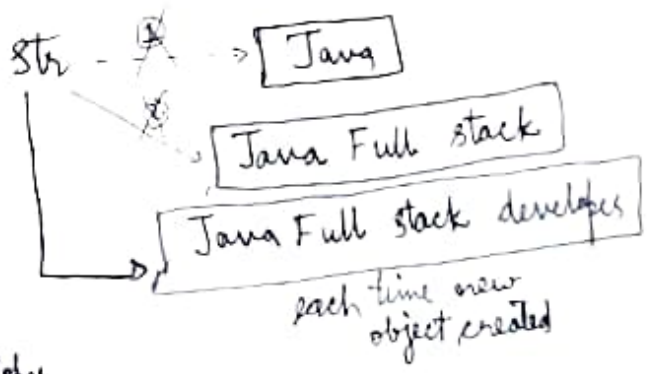
str.compareTo("----");
↑ current String ← Given String

```
String str = "Delhi";
Sopln(str.compareTo("Banglore"));  // +ve
Sopln(str.compareTo("Mumbai"));    // -ve
Sopln(str.compareTo("Delhi"));     // 0
```

Sopln("Kerna".compareTo("Karnataka"); // -ve

String class Objects are immutable
Once a String Object is created it cannot be modified or
changed If the programmer trys to make changes
or modification it will internally create a new
String Object

```
String str = "Java";
Sopln(str); → Java
str = str + "Full stack";
Sopln(str); Java Full stack
str = str + Developer;
Sopln(str); Java Full stack developer
```



to overcome this drawback of String class, Java provides string buffer and string Builder

StringBuffer & STRINGBUILDER

STRINGBUFFER & STRINGBUILDER are inbuild classes present in java.lang package

StringBuffer & StringBuilder can be used by the programmer as an alternate for String class.

Both StringBuffer and StringBuilder are final classes

We can create StringBuffer & StringBuilder objects by using the new Keyword

```
StringBuffer s1 = new StringBuffer ("Java");
Sopln(s1);    // Java
StringBuilder s2 = new StringBuilder("python");
Sopln(s2);    // python
```

StringBuffer & StringBuilder both override only the toString method of object class

The toString method is overridden to return the contents present inside the StringBuffer or StringBuilder object

```
StringBuffer buffer = new StringBuffer ("A");
Sopln(buffer.toString());   // A
Sopln(buffer.hashCode());   // 16509123
StringBuilder builder = new StringBuilder ("a");
Sopln(builder.toString());  // a
Sopln(builder.hashCode());  // 66123442
```

StringBuffer & StringBuilder class donot implement comparable interface. Hence, they are not comparable type objects.

StringBuffer & StringBuilder objects are mutable. that is it can be modified after creating

```
String str = "SQL";
str = str + "Database";   //IMMUTABLE: New Object gets created internally

StringBuffer s1 = new StringBuffer("Java");
s1.append(" Full Stack");   // MUTABLE : existing object gets modified
Sopln(s1);
StringBuilder s2 = new StringBuilder("Python");
Sopln(s2);   // Python
s2.append(" Data Science");   // MUTABLE : Existing object gets modified
Sopln(s2);   // Python Data Science
```

StringBuffer

- Introduced in v1.0
- legacy class
- ThreadSafe and Synchronized
- Secure
- Slow

StringBuilder

- Introduced in v1.5
- Non legacy class
- Not ThreadSafe and Not Synchronized
- less Secure
- Fast

Multiple Thread can access StringBuilder object at a same time

Parameter	String	StringBuffer	StringBuilder
version	v1.0	v1.0	v1.5
legacy class	Yes	Yes	No
literal creation	Yes	No	No
Overridden methods of Object class	toString(), hashCode(), equals()	toString()	toString()
Comparable Type	Yes	No	No
Immutable	Yes	No	No
ThreadSafe	Yes	Yes	No
Synchronized	No	Yes	No
Speed	Slow	Slow	Fast
Secure	Secure	Secure	less Secure

```
String str = "Tiger";
char[] ch = str.toCharArray();      // String ---> char[]
Sopln(ch);
StringBuffer buffer = new StringBuffer(str);   // String ---> StringBuffer
Sopln(buffer);
StringBuilder builder = new StringBuilder(str);   // String ---> StringBuilder
Sopln(builder);
```

```
char[] ch = {'J','a'};
String str = new String(ch);    // char ---> String
Sopln(str);
StringBuffer buffer = new StringBuffer(str);
Sopln(buffer);
```

```
StringBuffer buffer = new StringBuffer("Jdbc");
String str = buffer.toString();   // StringBuffer ---> String
Sopln(str)
char[] ch = str.toCharArray();
Sopln(ch);
StringBuilder builder = new StringBuilder(str);
Sopln(builder);

StringBuilder builder = new StringBuilder("Selenium");
String str = builder.toString();    // String
char[] ch = str.toCharArray()
StringBuffer buffer = new StringBuffer(str);
```

WRAPPER CLASSES

Wrapper classes are inbuilt classes in java.lang package.
Wrapper classes are use to support premetive datatype
All wrapper classes in java are final and immutable

PRIMITIVE DATATYPE	WRAPPER CLASS
byte	Byte
short	Short
int	Int
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Using Wrapper class we can perform following 3 operation
Boxing Unboxing Parsing

BOXING

It is the concept of representing a premitive value as a wrapper class objects

```
int a = 10;
Integer ref = new Integer(a);   BOXING: int --> Integer
Sopln(ref);
double d = 4.5;          // Boxing: double --> Double
Double obj = b;
char c = 'J';
Character ch = c;        // Boxing
Sopln(ch);
```

Unboxing

It is the process of representing a wrapper class object as a primitive value

```
Integer obj = 10;
int x = obj;     // Unboxing :- Double Integer --> int
Sopln(x);
Boolean bool = true;
boolean b = bool        // Unboxing   Boolean --> boolean
Sopln(b);
```

PARSING

Parsing is the process of extracting primitive values that are embeded inside a String

To support parsing the wrapper class provide predefined parser method

wrapper class	Parser method
Byte	→ parseByte()
Short	→ parseShort()
Integer	→ parseInt()
Long	→ parseLong()
Float	→ parseFloat()
Double	→ parseDouble
Character	→ NA
Boolean	→ parseBoolean

```
String s1 = "143";
int x = Integer.parseInt(s1);
Sop ln (x);
String s2 = "1.96";
Double d = Double.parseDouble(s2);
Sop ln (d);
String s3 = "false";
boolean b = Boolean.parseBoolean(s3);
```

If we use an invalid parser method it will cause NumberFormatException

Ex:
```
String str = "12.69";
int val = Integer.parseInt(str);   // Exception: NumberFormatException
Sop ln (val);
```

We can extract character value from String using charAt() method or toCharArray() hence there is no need of parser method for char

Ex:
```
String str = "admin";
char ch = str.charAt(4);
char[] ch = str.toCharArray();
```

Ex
```
Class Demo {
    static void test (int a){
        Sop ln ("Kohli");
    }
    static void test (Integer a){
        Sop ln ("Rohit");
    }
}
```

Main

```
Demo.test (10); // Kohli
```

because compare to wrapper class object parameter takes less space

Whenever there is a conflict b/w a primitive and ref type class the conflict is resolved in favour of primitive.

```
class Demo{
  static void test(Object obj){
      Sop("Hello");
  }
  static void test(String str){
      Sop("Rohit");
  }
}
```

```
Object obj = null;
String str = null;
Demo.test(null);   // Rohit
```

Whenever there is a conflict b/w parent and child the resolution is in favour of the child -

```
class Container{
    void add(Object obj){
        Sop("Inserted");
    }
}
```

```
Container c = new Container();
c.add(new Object());
c.add("Java");          //UPCASTING : String --> Object
c.add(new StringBuffer("SQL"));   //UPCASTING : StringBuffer --> Object
c.add(25);   // 1) Boxing : int --> Integer  2) Upcasting : Integer --> Object
```

THREADS

A Thread is a light weight process that is a program or ~~execution~~ application which is under execution and consumes less amount of system resources (CPU/RAM)

Threads concept is the one and only way in which the programmer can perform multitasking at a program level

The main goal of multithreading is to improve the speed of execution

Default Threads in Java

Whenever a Java program begins execution the JVM internally creates the following 3 threads

1. Main Thread
2. Thread Scheduler → DAEMON THREADS
3. Garbage Collector → DAEMON THREADS

DAEMON THREADS → low priority thread → background thread → not under control of programmer

The **Main** Thread is responsible for executing the code written by the programmer.
The **Thread Scheduler** is responsible for managing the life cycle of all threads in Java.
The **garbage** collector is responsible for memory management in heap area.
Thread Scheduler and Garbage collector are **Daemon Threads**.
Daemon Threads are low priority background threads which are not under the control of programmer.

How to create threads in java

As a java programmer we can create threads in 2 ways.

i) By extending thread class
ii) By implementing runable interface

by extending thread class

① Create a class that extends "java.lang.Thread"
② Override "run()" method inherited from Thread class
["run()" method should contain code to executed by the new thread]
③ Create an Object of Target class
④ Call "start()" method to begin Thread execution

```
class DemoThread extends Thread { ①
    @Override ②
    public void run() {
        for (int a=1; a<5; a++){
            Sop("demothread");
        }
    }
}
public class Main2 {  p s v m(-){
    Sop("p starts")
    DemoThread dt = new DemoThread(); ③
    dt.start(); ④
    for(int a=1; a<5; a++){
        Sop("main thread");
    }
    Sop("p ends");
}
```

by implementing Runnable Interface

(i) Create a class that implements "java.lang.Runnable"
(ii) Override "run()" method derived from Runnable
["run()" method should contain code to be executed by newThread]
(iii) Create an Object of Target Class
(iv) Create Object of Thread class by providing the target class Object
(v) Call "start()" method to begin Thread execution

```
class SampleThread implements Runnable{
    @Override
    public void run(){
        for(int a=1;a<5;a++){
            Sopln("demoThread");
        }
    }
}
public class M{
    p s v m(_){
        Sopln("p starts");
        SampleThread st = new SampleThread();
        Thread th = new Thread(st);
        th.start();
        for(int a=1; a<5 ; a++){
            Sopln("main Thread);
        }
        Sopln("p ends");
    }
}
```

Note: Between extending Thread class and implementing runnable Interface the recommended approach for creating a Thread is implementing Runnable interface as it provid more option for inheritances

Role of 'start()' method

An actual thread is created in java memory only when we call the "start()" method
'start()' method performs the following 3 tasks
i) Registers Thread with JVM
ii) Allocate Resource needed for execution
iii) Internally calls "run()" method

Note:

* What happens if we call run() method directly instead of calling start()
  i) No new thread will be created in Java memory
  ii) The code inside run method will be executed as a part of current thread

Thread.currentThread()

- currentThread() is a static method present in Thread class.
- We can use this method to get the currently executing thread as an object of thread class

Syntax: `Thread th = Thread.currentThread();`

Ex

```
public class MainClass1{
    p  s  v  m (_){
        Sopln("p start");
        for(int a=1; a<5; a++)
        {
            Thread th = Thread.currentThread();
            Sopln("ID:" + th.getID() + "Name:" + th.getName()
                                    + "Priority:" + th.getPriority());

            Sopln("p end");
        }
    }
}
```

Ex

```
class MyThread implements Runnable{
    @Override
    public void run(){
        for(int a=1; a<5; a++){
            Thread th = Thread.currentThread();
            Sopln(" ID: " + th.getID() + "Name:" + th.getName() +
                                    " Priority" + th.getPriority());
        }
    }
}
```

Main class

```
Thread t1 = new Thread(new MyThread());
t1.setName("IronMan");
t1.setPriority(10);
Thread t2 = new Thread(new MyThread());
t2.setName("Thor");
t2.setPriority(1);
t1.start();
t2.start();
```

Thread Properties

Every thread created in java will have the following 3 properties

ID

Name

Priority

ID :- ID is an integer value given by the jvm. It is used by the JVM to keep track of the thread

Name :- It is a string value which can be given either by the programmer or by the JVM

Priority :- It is an Integer value ranging b/w 1 to 10 with 1 being the lowest priority and 10 being a higest priority

Note
We can access the thread properties by using the getter methods provided in thread class

i) getId() ii) getName() iii) getPriority()

We can assign thread properties by calling the setter methods of thread class

i) setName(String) ii) setPriority(int)

Note
i) Id is a read only properities i.e, we can access the id value but we can not assign

ii) The priority of a thread is checked only when there is a resource crunch

Synchronization

It is the concept of making a method/resource thread safe

We can perform syncronization in java using the keyword synchronized

If a method or resource is declared with synchronized keyword it can only be accessed by 1 thread at a time

```
class Application
{
T1,T2,T3 → void read()
           {
             ...
           }
T1,T2,T3 → synchronized void write()
           {
             ...
           }
           void modify()
           {
             ...
             synchronized
             {
               ...
             }
             ...
           }
}
```

```
class BookMyShow
{
    void browser()
    {
    }
    Synchronized void payment()
    {
    }
    void selectSeat()
    {
        ...
        synchronized
        {
          ...
        }
        ...
    }
}
```

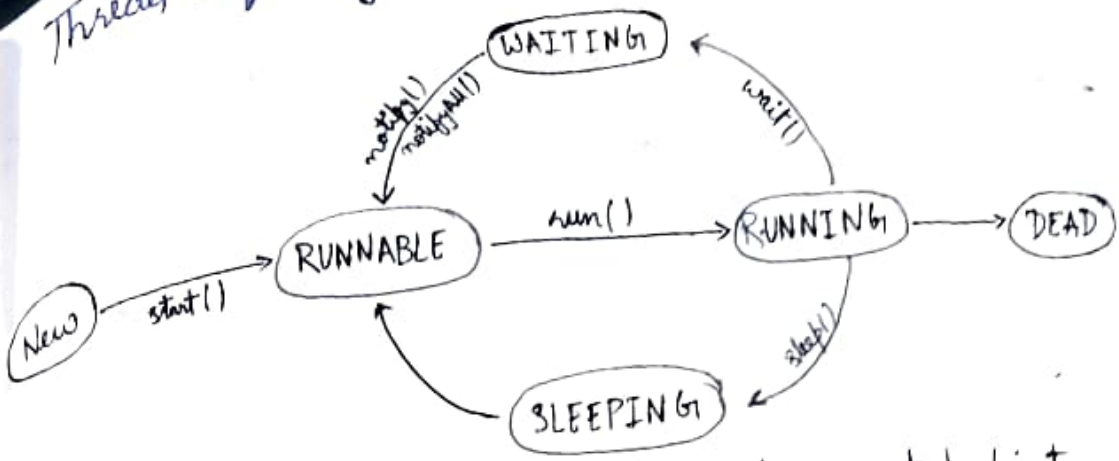
If all the methods in a class is created/declared with synchronized keyword then the class can be considered as a thread Safe class → Object lock

Thread.sleep()

sleep is a static method present in Thread class
The programmer can use sleep method to pause program execution for a certain time
Sleep() method is risky and may cause InterruptedException

```
public class Main{
    p  s  v  m(_){ throws InterruptedException{
        Sopln("Program Starts");
        Thread.sleep(5000);
        Sopln("End");
    }
}
```

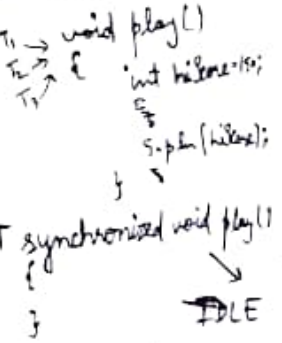
Thread life cycle



New In this state thread is a newly created object
Runnable In this state thread is ready to began execution
Running In this state thread is currently executing
Waiting In this state thread execution is paused by JVM
Sleeping In this state thread execution is paused by programmer
Dead In this state threat has completed their execution

Race Condition

Multiple threads competing for the same resource is called as Race Condition
Race Condition may lead to data inconsistency
We can prevent race condition using synchronization



Dead Lock

- In a dead lock scenario 2 or more threads are mutually waiting for each other to complete before starting execution
- In a dead lock scenario the resource become IDLE and its not utilised
- We can prevent dead lock scenario by using inter-process thread communication

FILE HANDLING

File: A file is a container that holds information and has a well known file extension

Folder/Directory: A folder/Directory is a container that contains files

File System: It is an arragments of files and folders in a computer

File Handling

It is the concept of manupulating the files and folders present in a computer

Java library provides several predefined programs which can be used by the programmer to perform file handling

All pre-defined java program related to file handling can be found inside "java.io" package

Note: The Java file handling concept is based on the UNIX (operating system) OS

```
// create a folder/directory
import java.io.File;
public class Day1{
    p s v  m(){
        File obj = new File("C:/FILEIO");
        boolean flag = obj.exists();
        if(flag==false)
        {
            obj.mkdir();
            Sopln("Folder Created");
        }
        else
        {
            Sopln("Folder Already Exists");
        }
    }
}
```

```
// create a file
import java.io.File
import java.io.IOException;
File ref = new File("C://FILEIO/Demo.txt");
boolean flag = ref.createNewFile();
if(flag == false)
{ try {
      ref.createNewFile();
      Sopln("File Created");
  }
  catch (IOException e){
      e.printStackTrace();
  }
}else {
   Sopln("File Already Exists");
}
```

```
// Delete a file/folder
import java.io.File
File ref = new File("C://FILEIO");
// File ref = new File("C://FILEIO/demo.txt");
boolean flag = ref.exists();
if(flag == true){
    ref.delete;
    Sopln("Deleted");
}
else {
    Sopln("Not found");
}
```

```
// length of file (character count)  // Path find
File ref = new File("C://FILE/DEMO.txt");
boolean flag = ref.exists();
if(flag == true){
    Sopln(ref.length());
    Sopln(ref.getAbsolutePath());
}
else{
    Sopln("File Does Not Exist");
}
```

```
// list file & folder -> list()
File ref = new File("C://FILEIO");
String[] arr = ref.list();
for(int i=0; i<arr.length; i++)
{   Sopln(arr[i]);
}
```

```
for(int i=0; i<arr.length; i++){
    File obj = new File(ref, arr[i]);
    boolean flag = obj.isFile();      // to list file only
    // boolean flag = obj.isDirectory(); // to list folder only
    if(flag == true){
        Sopln(arr[i]);
    }
}
```

"java.io.File"

- It is an inbuild class present in java.io package
- The File class provides some common methods needed by all java programs that perform basic file manupulation

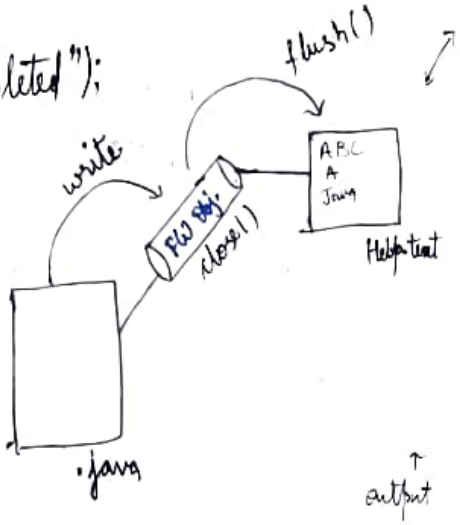
i) exists() → to check file/folder is present → return (true/false)
ii) mkdir() → use to make folder/Directory
iii) createNewFile() → use to create new file
iv) delete() → delete file/folder
v) length() → count character in file
vi) getAbsolutePath → use to find location of file/folder
vii) list() → use to list/show files/folders
viii) isDirectly() → use to list folder only
ix) isFile() → use to list file only

Constructor of File class

File ref = new File(String Path);
File(File obj, String Path);
File(String loc, String name);

Writing character data to a file

```
import java.io.IOException;   import java.io.FileWriter;
public class Prf
    p s v m(-){
        FileWriter ref = null;
        try {
            ref = new FileWriter("C:/FILES0/Demo.txt");
            ref.write("ABC");
            ref.write("\n");
            ref.write(65);
            ref.write("\n");
            char[] ch = {'J','a','v','a'};
            ref.write(ch);
            ref.flush();
            Sop.ln("write Completed");
        }
        catch(IOException e){
            e.printStackTrace();
        }
        finally {
            try {
                ref.close();
            }
            catch(IOException e){
                e.printStackTrace
        } }
    }
}
```



Steps
i) Create an Object & File Writer
ii) Write data to FileWriter Object using "write" method
iii) Transfer data to Physical File using "Flush" method
iv) Close FileWriter by calling "close()" method

"java.io.BufferedWriter"

i) Create an Object of BufferedWriter
ii) Write data using "write()" method
iii) Transfer data to physical using "flush()" method
iv) Close BufferWriter by calling "close()" method

```
import java.io.BufferedWriter; import java.io.FileWriter
BufferedWriter ref = null;
try{
    ref = new BufferedWriter(new FileWriter("C:/FILE/name.txt"));
    ref.write("XYZ");
    ref.write(97);
    char[] ch = {'M','a','v','i'};
    ref.write(ch);
    ref.flush();
}
catch(IO
```

"java.io.PrintWriter"

i) Create an Object of PrintWriter
ii) Write data use "print()" or "println()"
iii) Transfer to physical file by calling "flush()"
iv) Close PrintWriter by calling "close()" method

```
PrintWriter ref = null;
try{
    ref = new PrintWriter("C:/FILE/menu.txt");
    ref.println("Java");
    ref.println(65);
    ref.flush();
}
catch(IOException e){
    e.printStackTrace();
}
finally{
    ref.close();
}
```

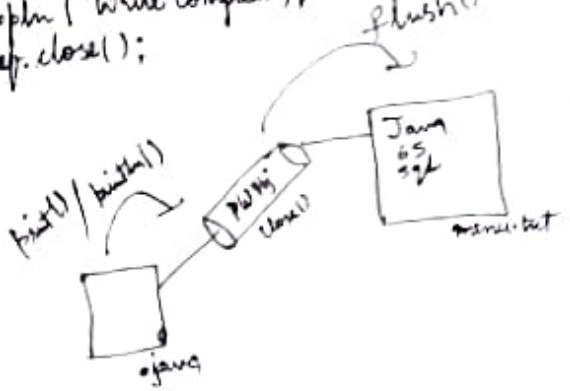
```
ref.newline();
ref.newline();
ref.println("sql");
Sopln("closed");
```

```
catch(IOException e){
    e.printStackTrace();
}
finally{
    try{
        ref.close();
    }
    catch(IOException e){
        e.printStackTrace();
    }
}
```

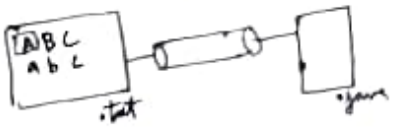
```
ref.write(ch);
Sopln("write complete");
```

→

```
PrintWriter ref = new PrintWriter("location");
ref.println("Java");
ref.println(65);
ref.flush();
Sopln("Write complete");
ref.close();
```



Reading character data from a file



drawback
↓
→ it is reading data one by one

"java.io.FileReader"

i) Create an object of FileReader
ii) Read Data from file using "read()" method
iii) Close FileReader by calling "close()" method

```
FileReader ref = null;
try{
    ref = new FileReader("C:/FILE/record.txt");
    int val = ref.read();
    while(val != -1){
        Sopln((char) val);
        val = ref.read();
    }
```

```
catch(IOException e){
    e.printStackTrace();
}
finally{
    try{
        ref.close();
    }catch(IOException e){
    }
}
```

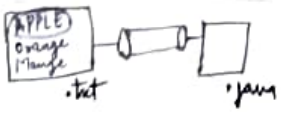
java.io.BufferedReader"

i) create an Object of BufferedReader
ii) Read Data from file using "readline()" method
iii) Close BufferedReader by calling "close()" method

```
BufferedReader ref = new BufferedReader(new FileReader("C:/F/f.txt"));
BufferedReader ref = null;
try {
    ref = new BufferedReader(new FileReader("C:/F/f.txt"));
    String val = ref.readline();
    while(val != null){
        So.pln(val);
        val = ref.readline();
    }
} catch(IOException e){
    e.printStackTrace();
} finally {
    try{
        ref.close();
    } catch(IOException e){
        e.printStackTrace();
    }
}
```

```
BufferedReader ref = new BufferedReader(
                     new FileReader("location"));
String val = ref.readline();
while(val != null){
    So.pln(val);
    val = ref.readline();
}
ref.close();
```

← Sudo Code.



* Serialization Vs Deserialization

Serialization is the concept of writing a java object into ~~text~~ text file

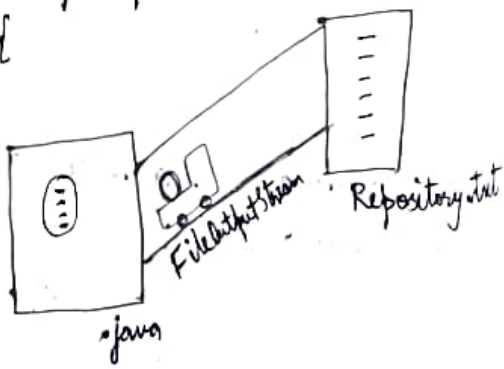
During Serialization the object is converted into a stream of ~~bytes~~ byte

The concept of serialization only works for objects that are serializable type

The concept of serialization is also known as MARSHALLING

import java.io.FileOutputStream; .IOException ; .ObjectOutputStream ; .Serializable;

```
class Employee implements Serializable{
    int eid;
}
public class Main1{
    p  s  v  m(-){
        Employee emp = new Employee();
        emp.eid = 101;
        FileOutputStream fos = null;
        ObjectOutputStream = null;
        try{
            fos = new FileOutputStream("C:/FILE/Repository.txt");
            oos = new ObjectOutputStream(fos);
            oos.writeObject(emp);
            So.pln("Serialization completed");
        }catch(IOException e){
            e.printStackTrace();
        }finally{
            try{oos.close();
                fos.close();
```



serialization is use to share objects to other programs or program.

Deserialization

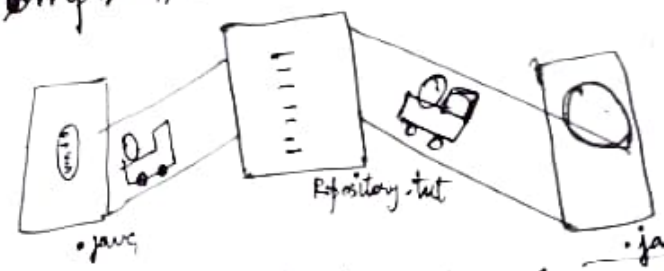
It is the concept of reading a java object from a text file

During deserialization the object is reconstructed from a stream of bytes

The concept of deserialization is also known as UNMARSHALLING

```
import java.io.FileInputStream;
FileInputStream fis = null;
ObjectInputStream ois = null;
try {
    fis = new FileInputStream("C:/FILE/Repository.txt");
    ois = new ObjectInputStream (fis);
    Employee emp = (Employee) ois.readObject();
    System.out.println(emp.eid);
}
```

```
; .IOException ; .ObjectInputStream;
catch(IOException){
    e.printStackTrace();
}
catch(ClassNotFoundException e)
{
    e.printStackTrace();
}
finally {
    try{
        ois.close();
        fis.close();
    }catch(IOException e){
    }
}
```



only work for data members

We can prevent serialization in java by using the keyword transient

File Permissions

Every file created in a Computer will have following three permission attributes, which are

i) Read ii) Write iii) Execute

We can Read the file Permissions for any file by using the below 3 methods of file class.

i) canRead() ii) canWrite() iii) canExecute()

```
File ref = new File("C:/FILEIO/PERMISSIONS/newFile.txt");
boolean flag = ref.exists();
if(flag==true){
        Sopln("Readable :"+ref.canRead());
        Sopln("Writable :"+ref.canWrite());
        Sopln("Executable:"+ref.canExecutable());
}
else{
        Sopln("File not exists");
}
```

We can assign permission to a file using the below methods of file class

i) setReadable(boolean)
ii) setWritable(boolean)
iii) ~~setReado~~ SetExecutable(boolean)

The file permissions attributes assigned by a java program is having lesser priority in comparision to the permission attributes assigned by the Operating System