JAVA

23/02/2021
Tues

→ JAVA is an Object oriented, Platform independent Programming Language. The Basic purpose of a programming Language to develop software Applications.

→ Software Applications / software basically a set of programs which will help us to perform specific task. (It automate the manual work.) [Set of program = set of instruction]

PROGRAMS:-

→ Programs are Basically some set of instructions which is meant for Platform. (Standard, Enterprise, Micro, FX Edition)

→ Platform is the combination of OS & Processor. (S/w + H/w)

OBJECT ORIENTED:-

→ JAVA basically follows 4-core principles.

(i) Inheritance.
(ii) Polymorphism.
(iii) Encapsulation.
(iv) Abstraction.

→ In order to achieve this 4-core principles, we need Class & Objects.

(i) CLASS:-

→ Class is basically a Blueprint designed / planned using which we can create objects.

→ Class is also known as a logical entity.

(ii) OBJECTS:-

→ Anything which has Physical presence or physical appearance can be considered as an Objects.

→ Objects is also known as a real world physical entity. (e.g → Mobile, Laptop)

→ Object is madeup of 2 components.

(i) States.
(ii) Behaviour.

(a) STATES:-

→ States are nothing but a data or an information which describes about the object.

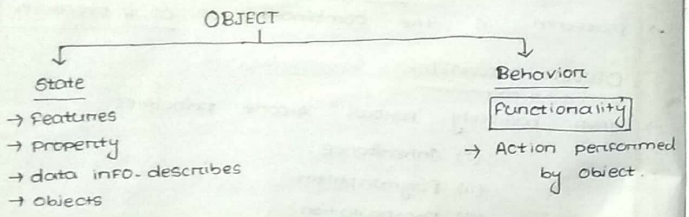
→ property, features, specifications are also known as state of an object.

e.g → Mobile (camera, Processor, RAM, ROM)

(b) Behaviour:-

→ It is nothing but functionality of an object. It is also known as action performed by an object.

→ e.g → Mobile (calling, Texting, games)



24/02/2021
Wed

Keywords:-

→ Keywords are basically predefined words or reserved words, which are already present in JAVA. Each & every keyword has its own meaning and also has its own purpose to serve. Ex:- import, for, if, class, new, else.

Syntax:-

```
class <class Name>
  {
    Body of class /scope.
  }
```

NOTE:- All the keywords has to be in lower case.

IDEANTIFIERS:-

→ It is basically the name which we are giving to a class, where in it will help us to ideantify a perticular class.

Rules for an Ideantifier:-

(i) We can't have any space while providing a class name or for an ideantifier.

e.g → class my db (x)

(ii) We can't use any special character w.r.t class name or an ideantifier except '$' & '_'.

e.g → class my_db
{ ✓
}

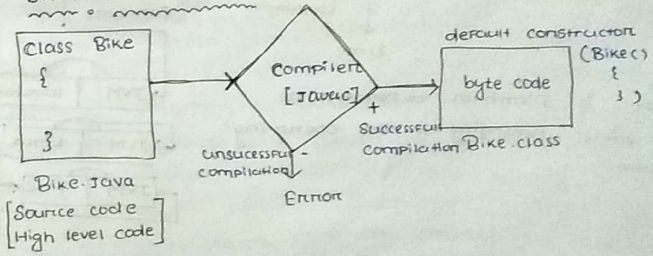
(iii) We can't start a class name or an ideantifier with number, but we can add number in between or at the end of an ideantifier.

e.g → class 2spider
{
x
}

(iv) We can't use keywords for a class name or for an ideantifier.

e.g → class class
{
x
}

JAVA - COMPILATION



→ Java compilation is a 2 step process

i) compiler will check syntax w.r.t the JAVA source code.

ii) If compiler finds any syntax mistake w.r.t the Java source code, then compiler throughs an error. (Information about the syntax mistake)

iii) If compiler finds no syntax mistake w.r.t the Java source code, then compiler generate byte code or .class file.

Byte Code:-

→ It is neither High level or Low level, but it is an intermediate code. (Theoratically Human readable + Machine readable.)

Q:- Who is going to generate Byte code?
Compiler.

Q:- When does compiler generate Bytecode?
At the time of successful compilation.

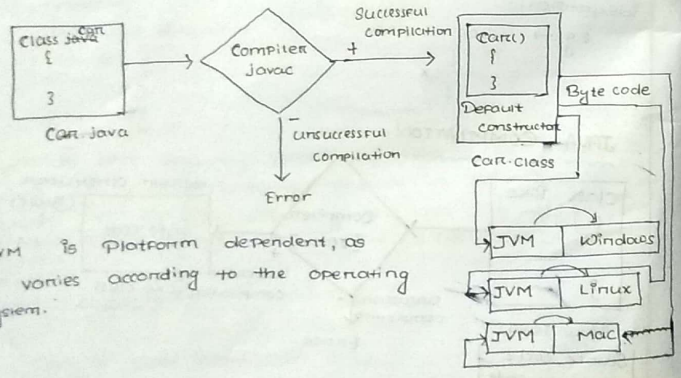
Q:- Who will generate default constructor & when?
Compiler will generate at the time of successful compilation.

→ Platform can't understand Byte code. The name of default constructor should be the class name.

→ JAVA is Platform independent. i.e. WORA (Write once Run Anywhere)

25/02/2021
Thu

*. WORA Architecture (Write once Run Anywhere)



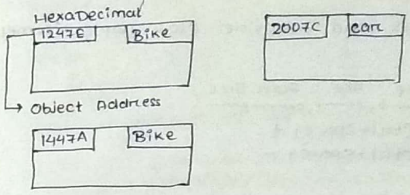
→ JVM is Platform dependent, as it varies according to the operating system.

OBJECT CREATION:-

(i)
```
Class Bike
{
Bike b1 = new Bike();
Bike b2 = new Bike();
}
```
b1, b2 → Object References

(ii)
```
Class Cart
{
Cart c1 = new Cart();
}
```

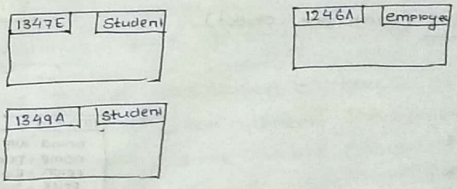
Heap Memory:-



(iii)
```
Class Student
{
Student S1 = new Student();
Student S2 = new Student();
}
```

(iv)
```
Class Employee
{
Employee e1 = new Employee();
}
```

Heap Diagram:-



WRITE A PROGRAM

26/02/2021
FRI

(i) Create a Folder. (Workspace)
(ii) Double click on eclipse Icon.
(iii) Select a directory as Workspace.
→ Click on Browse and Select the Folder. (Step-1)
→ Click on lunch
(iv) Close all Tabs.
(v) Click on Quick Access. (RHS TOP CORNER)
→ Type navigator and hit enter.
(vi) Click on new icon. (LHS TOP corner below file)
(vii) Double click on Java project.
(viii) Provide Project name. (Camel Case)
NOTE:- Execution Environment JRE → 1.8
(ix) Click on Finish.
(x) Expand Project Folder.

(xi) Click on source folder. (src)

(xii) Click src then press ctrl+N.

(xiii) Double click on class and provide class name. (camel)

(xiii) Click finish.

(xiv) Increase or Decrease the font size.

ctrl+shift + + ctrl+space+ +

ctrl+shift + − ctrl+space+ −

(xv) CREATE MAIN METHOD inside class:-

Type MAIN, use the shortcut ctrl+space, then hit Ent

(xvi) Creat object.

(xvii) sysout ctrl+Space.

TO RUN JAVA PROGRAM:-

→ Right click on the program. click on Run as Java application

27/02/2021, Sat

States:- Case-1 (with one object)

```
Class Bike
 {
   brand = ktm;
   name = Duke;
   color = Black;
   price = 200000;

   Public Static void main (String[] args)
       {
         Bike b1 = new Bike();
       }
 }
```

1032A	Bike
brand = ktm	
name = Duke	
color = Black	
price = 200000	

Case-2 (with two object)

```
Class Student
 {
   Public Static void main (String[] args)
       {
         Student S1 = new Student();
         Student S2 = new Student();
       }
   id = 1;
   Name = Bidya;
   age = 23;
 }
```

1835E	Student
id = 1	
Name = Bidya	
age = 23	

1835E	Student
id = 1	
Name = Bidya	
age = 23	

Q:- Create Employee class. Declare & Initiate 3 states id, name, salary. Create one employee object. WAP

```
Class employee
 { {
   Public Static void main (String[] args)
       {
         employee e1 = new employee();
         employee e2 = new employee();
       }
   id = 25;
   Name = Bidya;
   Salary = 100000;
 }
```

1034E	employe
id = 25	
Name = Bidya	
salary = 100000	

NOTE:-

(i) Initialization:- It is the process of assigning values to the states on a container.

(ii) Declaration:- Just declaring not assigning the value

* This type of initilization is known as direct Initialization. Whenever we go for direct Initialization then all the objects stores some values. (states)

Case-03 Initialization of States Using object References Init/Decl

```
Class Bike
 {
   brand;
   Name;
   color;
   price;
   Public static void(String[]arg)
       {
         Bike b1 = new Bike()
         b1.brand = ktm;
         b1. Name = Duke;
         b1. color = Black;
         b1. price = 200000;

         Bike b2 = new Bike()
         b2. brand = TVS;
         b2. Name = Scootypep;
         b2. color = Pink;
         b2. price = 70000;
 } }
```

1043A b1 ⟹

1043A	Bike
brand = ktm	
Name = Duke	
color = Black	
price = 200000	

1044A b2 ⟹

1044A	Bike
brand = TVS	
Name = Scootypep	
color = Pink	
price = 70000	

```
Class employee
{
    id;
    name;
    salary;
    public static void (String[]args)
    {
        employee e1 = new employee();
        e1.id = 1;
        e1.name = Bidya;
        e1.salary = 100000;
        employee e2 = new employee();
        e2.id = 2;
        e2.name = Bidya1;
        e2.salary = 20000;
    }
}
```

1042A e1 → 1042A | employee | id=1 | Name=Bidya | Salary=100000

1043A e2 → 1043A | employee | id=2 | Name=Bidya1 | Salary=20000

* Interpreter & Compiler ?
* Execution.
* Object References.

01/03/2021
Mon

Data Types :-

→ It is basically specifies about the data (types of Data).
→ Data types are classified into 2 types.
  (a) Primitive DT.
  (b) Non-Primitive DT.

(a) Primitive DT :- These are nothing but predefined datatypes, which are available in the form of keywords.

Ex:- byte, short (-32687 to +32686), int, long — Non-decimal numbers — -128 to 127, 30, 30, 100, -2^{+31} to $2^{31}-1$, -2^{+63} to $2^{+63}-1$
- float (6 to 7), double (12 to 15) — Decimal numbers — 2.5, 6.8732
- boolean — TRUE, FALSE
- char — 'a', 'b', 'c', 'd'

Data Type container = Data

Ex:-
```
Class Bike
{
    String brand = "KTM";
    String name = "Duke";
    int price = 200000;
    Public static void main (String[]args)
    {
        Bike b1 = new Bike();
        Bike b2 = new Bike();
    }
}
```

Complete Program :-
```
Public class car
{
    String brand;
    String name;
    int price;
    public static void main (String[]args)
    {
        car c1 = new car();
        c1.brand = "audi";
        c1.name = "R8";
        c1.price = "400000";
        car c2 = new car();
        c2.brand = "BMW";
        c2.name = "M3";
        c2.price = 800000;
        System.out.println(c1.brand);
        System.out.println(c1.name);
        System.out.println(c1.price);
    }
}
```

C1 → 1034E | Car | brand=audi | name=R8 | price=400000

C2 → 1035E | Car | brand=BMW | name=M3 | price=800000

O/P → audi, R8, 400000

Q. Create employee class.
Define 3 properties id, name, salary
Create minimum 3 objects, initialize all the objects with different values using object references

```
Public Class Employee
{
    int id;
    String name;
    int salary;
    Public static void main (String[]args)
        {
            Employee e1 = new Employee()
            e1.id = 125;
            e1.name = "Bidya";
            e1.salary = 500000;
            Employee e2 = new Employee()
            e2.id = 126;
            e2.name = "Devaki";
            e2.salary = 500000;
            Employee e3 = new Employee()
            e3.id = 127;
            e3.name = "Alok";
            e3.salary = 500000;
            System.out.println(e1.id);
            System.out.println(e1.name);
            system.out.println(e1.salary);
        }
}
```

```
o/p → 125
      Bidya
      500000
```

03/03/2021
Tues

Q- Two cartoon object. (Ben10, TOM).
16 5

```
Public class Cartoon
{
    String name;
    int age;
    public static void main(String[] args)
    {
        Cartoon c1 = new cartoon();
        c1.name = "Ben10";
        c1.age = 16;
        Cartoon c2 = new cartoon();
        c2.name = "TOM";
        c2.age = 5;
        system.out.println(c1.name+" "+c1.age);
        system.out.println(c2.name+" "+c2.age);
    }
}
```

```
o/p →
Ben10 16
TOM 5
```

Q-

12344A	Bike1	10099C	Bike2	12550Z	Bike1
brand = Yamaha		brand = suzuki		brand = ktm	
name = r15		name = gixxer		name = rc200	
color = blue		color = black		color = orange	

98776A	car1	77665I	car1
name = civic		name = civic	
color = white		color = white	
price = 2100000		price = 2100000	

*.
```
Public class Bike1
    {
        String brand;
        String name;
        String color;
        Public static void main (String[]args)
            {
                Bike1 b1 = new Bike1();
                b1.brand = "yamaha";
                b1.name = "r15";
                b1.colour = "blue";
                Bike1 b2 = new Bike1();
                b2.brand = "suzuki";
                b2.name = "gixxer";
                b2.color = "black";
                Bike1 b3 = new Bike1();
                b3.brand = "ktm";
                b3.name = "rc200";
                b3.color = "orange";
                system.out.println(b1.brand+" "+b1.name+" "
                                   +b1.color)
                system.out.println(b2.brand+" "+b2.name+" "
                                   +b2.color)
            }
    }
```

```
o/p → Yamaha r15 blue
      suzuki gixxer black
```

4.
```
Public class car1
    {
        String name = "civic";
        string color = "white";
        int price = 2100000;
    }
```

Default values:-

```
Public Class Car
{
    int price = "350000";
    String name = "nano";
    String colon: "blue";
    public static void main (string[]args)
    {
        Car c1 = new car()
        c1. price = 2100000;
        c1. name = "civic";
        c1. color = "white";
        Car c2 = new car()
        System. out. println (c1. name +" "+ c1. price+" "+ c1.color
        System. out. println (c2. name +" "+ c2. price+" "+ c2.color
    }
}
```

o/p → civic 2100000 white
nano 350000 blue

NOTE:- It will always take the updated value.

→ The object reference initialization can be in any sequence. (c2 under c3 not c3 under c2) 03/03/2021 Wed

Default values:- When there is no initialization.

byte, short, int, long → 0
float, double → 0.0
String - null
Boolean - false
char - '_' (space)

Ex:-
```
Class employee
{
    int id;
    String name;
    int salary;
    public static void main (String[]args)
    {
        employee e1 = new employee()
        System. out. println (e1. id+" "+e1. name+" "+e1. salary
    }
}
```

o null 0

employee
id = ∅
name: null
salary = ∅

*. When initialization done, then the default value will be updated.

CONSTRUCTORS:-

→ Constructor is basically one of the members of class or part of a class. It is also called as a specialized method.

→ Constructor is used for 2 purpose.
i) object creation.
ii) initialize the states.

Syntax:-
```
Class name()        →    Car1()
{                        {
    Instructions.
}                        }
```

Rules for constructor:-

→ constructor name and class name should be same.

→ Constructor does not have any return type or return statement.

*. Constructor is classified into 2 types.
i) Default constructor.
ii) User defined / custom / Parameterized constructor

(i) Default constructor:-

→ It is basically provided by the compiler upon successful compilation.

→ Default constructor is used to initialize state, o.r.t. default value.

(ii) User Defined / custom / Parameterized constructor:-

→ This type of constructor is basically provided by the user itself.

→ It is used to initialize state o.r.t user defined values or custom values.

Developer View

```
class car
{
  int price;
  String name;
  String color;
  public static void main(String[]args)
  {
    car c1 = new car()
    system.out.println(c1.price)
  }
}
```

Compiler View

```
class car
{
  int price;
  String name;
  String color;
  public static void main(String[]args)
```

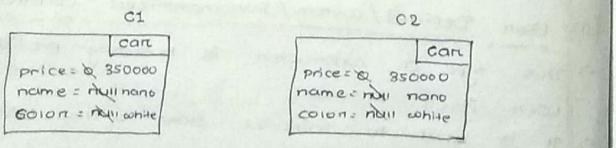
car()
{
}
Default Constructor

```
Public static void main(String[]args)
{
  car c1 = new car()
}
}
```

USER Defined Constructor:-

CASE-1

```
class car
{
  int price;
  String name;
  String color;
  car()
  {
    price = 350000;
    name = "nano";
    color = "white";
  }
  public static void main(String[]args) {
  car c1 = new car()
  system.out.println(c1.price+""+c1.name+" "+c1.color)
  car c2 = new car()
  system.out.println(c2.price+""+c2.name+" "+c2.color)
  }
}
```



O/P = 350000 nano white
350000 nano white

04/03/2021
Thu

CASE-2

```
class Student
{
  String name;
}
Student(String s)
{
  name = s;
}
public static void main (String[]args)
{
  Student s1 = new Student ("Bidya");
  Student s2 = new Student ("Bidya1");
  system.out.println (s1.name);
  system.out.println (s2.name);
}
}
```

Parameters:-

→ Parameter is basically a container where in it holds arguments. (Value passed).

Arguments:-

→ It is basically the value passed while creating object.

Rules to initialize states using Parameterized constructor:-

(i) The no. of available parameters and the no. of value passed should be same.

(ii) The data type of the available parameter and the data type of the value passed should be same.

(iii) The order of the avaliable Parameters and the order of value passed w.r.t data type should match.

CASE-03:-

```
class Student {
  Student(String S, int i, double P)
  {
    name = S
    id = i
    per = P
  }
```

```
Public static void main (String[]args)
{
    Student S1 = new Student ("Bidya", 125, 90.0)
    System.out.println (S1.name + " " + S1.id + " " + S1.per)
}
}
```

05/08/2021
FRI

CASE-04

```
Public class product
{
String name;
int price;
double qty;
product (string name, int price, double qty)
{
name = name;
price = price;
qty = qty;
}
Public static void main (string[]args)
{
  product p1 = new product("shampoo", 400, 8)
} system.out.println (p1.name + " " + p1.price + " " + p1.qty)
```

product
name = null shampoo
Price = 0 400
qty = 0.0 8.0

States Containers: name = name; price = price; qty = qty;
Parameter Containers: this.name = name; this.price = price; this.qty = qty;

o/p → null 0 0.0

o/p → shampoo 400 8.0

→ Whenever the States container name and the parameter container name are same, then we differentiate States container name using "this" keyword.

→ "this" keyword is basically a current invoking object reference.

06/08/2021
Sat

METHODS

06/08/2021
Sat

→ Methods are basically a member of the class.

→ Method is nothing but a block of codes, which get executed based on the method called or method invokation.

→ It is also known as behaviour of an object.

→ Syntax:- Returntype (or Void) <method Name>

```
{
  instruction /statements
}
```

→ Methods are basically used to execute some block of codes again and again based on the requirment.

*.
```
Public class Demo
{
Void display()
  {
  System.out.println ("bidya");
  System.out.println ("bidya1");
  }
  public static void main (String[]args)
    {
    Demo d1 = new Demo();
    System.out.println ("Panda");
    System.out.println ("Panda1");
    d1.display();
    }
}
```

o/p → Panda
Panda1
bidya
bidya1

*.
```
Public class method1
{
  int id = 125;
  String name = "bidya";
  void study()
    {
    System.out.println ("Study 3hours");
    }
```

```
Public static void main (String[]args)
    {
      method1 m1 = new method1();
      System.Out.Println(m1.id+" "+m1.name);
      m1.Study();
    }
}
```

O/p → 0 null
Study - 9hours

08/03/2021
Mon

METHOD CHAINING:-

→ One method calling another method is called method chaining.

*.
```
Public class method_100P
{
 int id;
 String name;
 void Study()
 {
  System.out.println("Read=4hours");
  Sleep(); or this.Sleep();
 }
 Void Sleep()
 {
  System.out.println("Sleep= 7hours");
 }
 public static void main (String[] args)
    {
      method_100P m1 = new method_100P();
      System.out.println(m1.id+" "+m1.String);
      m1.Study(); // ~~m1~~. this.Study(); (x)
    }
}
```

O/p → Read=4hr
Sleep=7hr

*.
```
Class Demo
{
 void display()
 {
   S.O.P(this);
 }
  main()
  {
    demo d1 = new demo()
  } d1.display();
```

O/p → Object address

①
```
Create Class JSpiders
define methods → Java(), Sql(), programming(), webtechnology(),
React(), framework(). Add one print statement in each method
Perform method chaining.
Java → Sql → Web technology → programming → React → Framework

Public class JSpiders
{
void Java()
  {
   S.O.P("JAVA"); Sql();
  }
void Sql() {
    S.O.P("SQL"); programming();
  }
void React() {
    S.O.P("React"); Framework();
  }
void programming() {
    S.O.P("programming");
    React();
  }
void framework() {
    S.O.P("framework");
  }
Public static void main (String[]args)
    {
      JSpider j1 = new JSpider()
      j1.Java();
    }
}
```

O/p → JAVA
SQL
programming
React
framework

NOTE:- (i) We can use constructor in the place of method to print the statement multiple time by using new keyword. But for every line we have to ~~one~~ create one object using new keyword.

Variables:-

Tues

→ Variables are nothing but data holder or container.

eg → String name = "bidya" ;

↳ container/variable.

→ Variables are two types:

(i) Global variables.

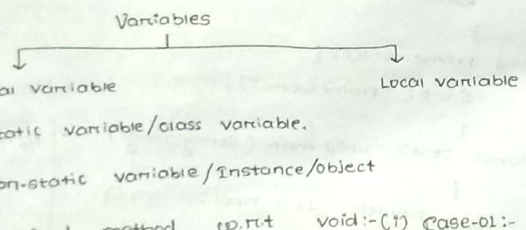
(ii) Local variables.

(i) Global variables:-

→ Global variables are those variable, which are written inside the class, but out side of the method, constructor or Blocks.

(ii) Local variables:-

→ These are those variables which we directly write inside method/constructor/Blocks.



Parameterized method w.r.t void:-(i) Case-01:-

→ Public class ArithematicOperation

```
{
  void add()
  {
    int a = 401;
    int b = 620;
    int res = a+b;
    System.out.println(res);
  }
  public static void main (String[] args)
  {
    ArithematicOperation a1 = new ArithematicOperation();
    a1.add(40);
  }
}
```

o/p → 1021

→ Public class ArithematicOperation; (ii) Case-02:-

```
{
  void add (int a1, int b1)
  {
    int a = a1;
    int b = b1;
    int res = a+b;
    System.out.println(res);
  }
  Public static void main (String[] args)
  {
    ArithematicOperation a1 = new ArithematicOperation()
    a1.add(40,60);
  }
}
```

o/p → 100

NOTE:-

→ Parameter variables and variable inside the method both should not have same names. (variable duplication)

→ No two local variables should have same name.

→ Local variables has to be initialize at the time of declaration. But not mandatory for Global variables.

→ Local variable can't be accessed using object reference or this keyword.

Scope of Local variables:-

→ The scope is limited to that perticular method, constructor/block.

→ Public class Demo

```
{
  int i = 100; // Global variables
  System.out.println(i)
  void display()
  {
    int x = 200;
    System.out.println(x);//(200)(Local)
    S.O.P(i); // 100 (Global)
    S.O.P(z); // Error - Local variable from different scope
  }
  Demo()
  {
    int z = 500;
    S.O.P(z); // 500 (Local)
```

```
}
Public static void main (String[] args)
{
    Demo d1 = new Demo();
    d1.display();
}
}
```

(b) Non-Static / Instance Variable:-

→ Variables return inside the class outside the main method, constructor, method, blocks is known as Global variables.

→ Global variables not prefixed to be at with any keywords is known as instance - Non-static variable.

Scope of Instance Variable:-

→ Non-static variable / Instance variables can be accessed anywhere inside the class. (same class).

→ Instance variables can also be accessed from a different class using object reference.

NOTE:-

Instance Variable

- same class = directly
  - this
  - object reference
- Different class = object reference.

→ Each time we create an object one copy of instance variable will be created inside the object.

→ n-object gives n-copies of instance variable.

→ Instance variable will be created only when the object is created.

→ memory allocation of instance variable will happen only when the object is created.

→ Instance variable will get destroyed only when the object is destroyed.

→ Instance variable will always be the part of an object.

10/03/2021
Wed

(i) Void <method name>
{
  Statements
}

(ii) returntype
  or
<method name>
Datatype
{
  Statements
  return returning value; return statement
}

Rules for methods:-

→ A void method can either Void or return type.

→ if → returntype

(i) returntype → min & max ⇒ 1

(ii) min & max ⇒ 1 return statements.

(iii) return statements ⇒ last executable line inside method

(iv) Returning value & returntype should match.

Example:-

(i)
```
String s()
{
  return "25";
}
```
(✓)

(ii)
```
int m3()
{
  return 43;
  S.O.P("hello")
  return 70;
}
```
(x)

(iii)
```
void m()
{
  return;
}
```
(✓)

(iv)
```
int m7()
{
  Return 77.0;
}
```
(x)

→
```
Public class Test{
    String display()
    {
      S.O.P ("Hello");
      return "Java";
    }
```

```
{
    Test t1 = new Test();
    String s = t1.display();
    S.O.P(t1);
    // S.O.P(t1.display());
```

o/p → Hello Java

```
Public Class square
{
  int square1(int n)
    {
      int res = n*n;
      return res;
    }
  int cube(int n)
    {
      int res = n*n*n;
      return res;
    }
  P.S.V.M
    {
      square s1 = new square();
      S.O.P(s1.square1(5));
      S.O.P(s1.cube(5));
    }
}
```

o/p → 25 125

11/03/2021 Thu

```
Public class employee
{
  int id;
  String name;
  int salary;

  void work()
  {
    S.O.P("Hardwork pays");
  }
  P.S.V.M {
    employee e1 = new employee()
    e1.id = 2;
    e1.name = "bidya";
    e1.salary = 500000;
    e1.work();
    S.O.P(e1.id+" "+e1.name+" "+e1.salary);
  }
}
```

o/p → 2 bidya 500000 Hardwork pays

```
Public class Test
{
  P.S.V.M
  {
    employee e2 = new employee()
    e2.id = 25;
    e2.name = "bidya1";
    e2.salary = 500000;
    S.O.P(e2.id+" "+e2.name+" "+e2.salary);
    e2.work();
  }
}
```

o/p → 25 bidya1 500000 Hardwork pays

Number of ways to initialize state:-

5 ways are there to initialize state.

(i) Direct initialization.
(ii) Using object reference.
(iii) Constructor.
(iv) Setter-method.
(v) Blocks.

(v) Blocks:- (Instance initialization Block - IIB) / Non-static Blocks.

Instance block are basically a member of a class or a part of a class.

Instance blocks are used to initialize instance variables or state.

Syntax:-

```
{
// Statements
}.
```

Case-01:-

e.g →

```
Public class employee
{
  int id; String name; int salary;
  {
    id = 125;
    name = "bidya";
    salary = 300000;
  }
  P.S.V.M {
    employee e1 = new employee();
    S.O.P(e1.id+" "+e1.name+" "+e1.salary);
  }
}
```

Case-02:-

* Each time when we create an object blocks will get executed.

Multiple Blocks:-

```
Public class employee
{
  {
    S.O.P ("Block-I");
  }
  {
    S.O.P ("Block-II");
  }
  {
    S.O.P ("Block-3");
  }
  P.S.V.M {
     employee e1 = new employee();
  }
}
```

O/p → Block-I
Block-II
Block-3

NOTE:-

→ Whenever we have multiple blocks, each time when we create an objects, all the blocks will get executed sequentially for once.

Blocks & constructor togeather in same program:-

→ Whenever we have multiple blocks and constructor togeather, each time we create an object first all the blocks get executed then constructor.

NOTE:-

→ Whenever compiler finds block and constructor togeather in a same program, then compiler modifies the existing program.

→ Inside any constructor always the first executable statement will be super,

```
Employee()
  { super();
    S.O.P ("constructor");
```

User view

```
Public class employee
{
  S.O.P ("Block-1");
}
 employee
 {
  super();
  S.O.P ("constructor");
 }
P.S.V.M {
   employee e1 = new employee();
  }
}
```

Compiler view

```
Public class employee
 {
  employee()
  {
   super();
   {
    S.O.P ("Block-1");
   }
   S.O.P ("constructor");
  }
  P.S.V.M {
     employee e1 = new employee
                        ();
  }
```

NOTE:-

→ When only Blocks are present, the blocks will be inside the default constructor after super().

→ In case of Blocks there is no parameterized initialization concept.

12/03/2021
FRI

METHOD OVERLOADING:-

→ Whenever we have morethan one method with same name, but different in signature, we call this as method overloading.

→ Signature ⇒ int add (int a, int b)
{ res=a+b; [Signature]
return res;
}

Rules for Signature:-

→ no. of parameters should be different.

→ Datatypes of the parameter should be different.

→ Sequence/order of the parameters should be different.

NOTE:-

To achieve method overloading out of 3 rules, minimum one Rule should be followed.

→ Public class Facebook

```
{
  void login (String email, String pwd)
  {
    S.O.P (" login via mail");
  }
  void login (long phNo, String pwd)
  {
    S.O.P (" log in via PhNo");
  }
  P.S.V.M [
    Facebook f1 = new Facebook();
    f1. login (" bidyadharapanda25@gmail.com", " bidya123");
      f1. login (9984222821 , " bidya123");
  }
}
```

→ Public class Airtel

```
{
  void payment()
  {
    S.O.P (" Payment via cash");
  }
  void payment (int cusID, String pwd)
  {
    S.O.P (" payment via Net-Banking")
  }
  void payment (int UPID, String pwd1, int pin)
  {
    S.O.P (" payment via UPI")
  }
  void Payment ( long cardno , int cvv, String expdate)
  {
    S.O.P (" payment via card")
  }
  P.S.V.M [
    Airtel a1 = new Airtel()
    a1. Payment() ;
    a1. payment ( 2345," pay@12") ;
    a1. payment ( 234 , " pay@12";@);
    a1. payment ( 4567 4562 4567, 321, 22/24);
  }
}
```

→ Class Demo

```
{
  void m()
  {
  }
  int m (int a)
  {
    return 50;
  }
  double m (string x)
  {
    return 50;
  }
```

```
  String m (int x, String s)
  {
    return "S";
  }
  int m (String s, int x)
  {
    return 4;
  }
```

Q:- Can we overload Main method ?

→ Yes, we can overload main() method in Java. But the JVM will always call the original main() method. It does not call the overload main() method.

→ public static void main (String[] args).

Default signature understood by JVM.

→ Public class main_method_overloading

```
{
  PSVM (int args)
  {
    S.O.P ( " Integer argument ");
  }
  PSVM ( char args)
  {
    S.O.P ( " Character argument");
  }
  PSVM (String[]args)
  {
    S.O.P (" original method") ;
  }
}
```

O/P → original method.

Variable Shadowing

13/03/2021 Sat

→ Whenever Global variable and local variable is having same name inside the local scope, then local variable dominates over the Global variable.

→ Public class Student

```
{
 String name = "Rahul";
 void display()
 {
  String name = "Rohit";
  S.O.P ("name");    // Rohit
  S.O.P ("this.name"); // Rahul
 }
 PSVM {
   Student s1 = new Student()
   s1.display()
 }
}
```

CONSTRUCTOR OVERLOADING

→ When we have multiple constructors in a class with different signature. This we call it as constructor overloading.

→ In order to initialize an object with multiple different ways we go for constructor overloading.

→ Public class Student

```
{
 int id;
 String name;
 String place;
 Student(int id)
   {
    this.id = id;
   }
 Student ( int i, String n, String p)
   {
    id = i;
    name = n;
    place = p;
   }
```

```
PSVM {
    Student s1 = new Student (125);
    S.O.P (s1.id + s1.name); // 125 NULL
    Student s2 = new Student( 1, "bidya", "odisha");
    S.O.P(s2.id+" "+s2.name+" "+s2.place); // 1 bidya odisha
  }
}
```

Q:- Create new Java project called Overloading. Creat class employee. Define 5 properties (id, name, age, salary, place).

Initialize employee e1 object → id, name
" " e2 " → id, name, salary
" " e3 " → id, salary
" " e4 " → id, name, age, salary, place

Print all the statements.

Ans:- IN LAPTOP.

Q:- Creat new JAVA project called ConstructorOverloading. Create class vegetables. Define 5 properties (name, color, price, qty.)

Initialize vegetable v1 object → name.
" " v2 object → name, price.
" " v3 object → name, price, color
" " v4 object → name, price, color, qty.

Ans:- IN LAPTOP.

Q:- Create a new project methodOverloading. Create class addition. Create 3 overload add method with different signature.

i) To add 2 numbers.
ii) To add 3 numbers.
iii) To add 4 numbers.

Call the 3 overload methods and return & print the result.

Ans- In LAPTOP.

INHERITANCE (IS-A Relationship)

15/03/2021
Mon

→ Inheritance is also known as parent child relation or is-A Relationship.

→ It is the process or mechanism of one class aquiring the properties and behaviour from another class.

*. Parent/Super/Base Class:-

→ The class which provides the properties and behaviour to another class is known as parent/super/Base class.

*. Child/sub/Derived Class:-

→ The class which aquires or takes the properties and behaviour from another class is known as child/sub/Derived class.

*. Advantage/uses of Inheritance:-

i) To avoid code duplication
ii) To achieve code reusuability.
iii) To achieve generalization.
iv) To achieve polymorphism (Indirectly).

→ Inheritance can be achieve through extends keywoord.

→ Inheritance is unidirectionally.

→ Public class vehicle
{
String name;
int price;
void startEngine()
{
S.O.P ("Start the Engine");
}
void stopEngine()
{
S.O.P ("Stop the Engine");
}
} Public class bike extends vehicle {
Public static void main (String[] args) {
bike bike
vehicle V1 = new vehicle ();
V1. name = "suzuki";
V1. price = "350000";
S.O.P (V1. name + " " + V1. price);
V1. startEngine();
V2. stopEngine();
}
}

→ Public class Vegetable {
int price;
String color;
double qty;
void washvegetable () {
S.O.P (" wash vegetable");
}
void cutvegetable () {
S.O.P (" cut vegetable");
}
}
Public class potatto extends vegetable
{
P.S.V.M {
potatto P1 = new pottatto();
P1. price = 40;
P1. color = "black";
P1. qty = 5.5;
S.O.P (P1. price + " " + P1. color + " " + P1. qty);
P1. washvegetable();
P1. cut vegetable();
}
}

Q1- Create a class Pen. Define 3 property brand, color, name. define a behaviour write(). Create a class marker_Pen. Insert the properties and behaviour of pen class. Using child class object access all the properties and behaviour. Print all the states/properties.

16/03/2021
Tues

→ Properties and behaviors defined with in Child class are considered as specific properties and specific behavior.

→ Using child class object we can access child class properties and behavior as well as parent class property and behavior.

→ Using Parent class object we can only access the Parent class property and behavior.

→ Public Class Bike extends Vehicle

```
{
  int cc;
  String color;   } specific properties
  void throttle() // specific behaviors
   {
    S.O.P ("Move forward");
   }
  void changegear() // specific behaviors.
   {
    S.O.P ("change gear to move faster");
   }
  P.S.V.M {
     Bike b1 = new Bike();
     b1.name = "duke";
     b1.price = 400000;
     b1.cc = 200;
     b1.color = "white";
     S.O.P (b1.name+" "+b1.price+" "+b1.cc+" "+b1.color);
     b1.StartEngine(); // common Behavior.
     b1.StopEngine(); // common Behavior.
     b1.throttle(); // specific Behavior.
     b1.changegear(); // specific Behavior.
   }
}
```

O/P → duke 400000 200 white
Start the engine
Stop the engine.
Move forward.
Change the gear to move faster.

→ ~~Public~~ Adding Restriction to the Properties and Behavior (Parent Class)

```
Public class Father
{
  int money = 100000;
  private String girlfriend = "katrina"; // Restricted property
  Void walking()
    {
      S.O.P("walking is good for health");
    }
  private void drinking()   // Restricted Behavior.
    {
      S.O.P ("drinking is injurious");
    }
  private void smoking() // Restricted Behavior.
    {
      S.O.P ("smoking kills");
    }
}
Public class son extends father
{
  String name;
  int age;
  Void spendmoney()
    {
      S.O.P("Invest in share market")
    }
  P.S.V.M {
     son s1 = new son();
     s1.name = "Salman"; //Specific property.
     s1.age = 55; //   ''
     S.O.P(s1.name+" "+s1.age); //  ''
     S.O.P(s1.money+" "+s1.girlfriend); // common properties
     s1.walking(); // common behavior
     s1.drinking(); // common behavior (Restricted)
     s1.smoking(); //   ''          (  ''  )
     s1.spendmoney(); // Specific behavior.
   }
}
```

Restricting the Entire Class ... Behavior from getting inherited)

→ If the class is prefixed with final keyword, meaning the perticulan class can't be inherited or that perticulan class can't have child class. e.g→ public final class Father{}

Final Keyword:-

→ If a variable is prefixed with final keyword meaning the value is fixed for that perticulan variable.

NOTE:- Final variable has to be initialized at the time of declaration.

e.g→
```
public class Demo
{
  final String name = "Rohit";
  public static void main (String[]args)
  {
    Demo d1 = new Demo();
    d1.name = Rahul // can't be Reassigned
    S.O.P (d1.name);
  }
}
```

NOTE:-

Private Proverty & private behavions → can't be inherited

final class → can't be inherited

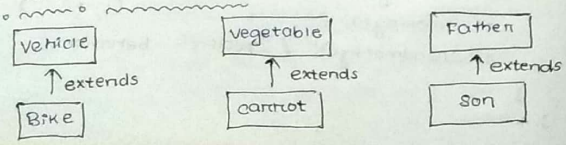
final properties & final behavions → can be inherited

17/03/2021
Wed

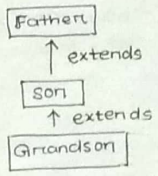
TYPES OF INHERITANCE

(i) Single level. } Class
(ii) Multi level. } Class
(iii) Hierarchical. } Class
(iv) Multiple level. } Inheritance Interface
(v) Hybrid level. } Inheritance Interface

(i) Single level Inheritance:-



(ii) Multilevel Inheritance:-

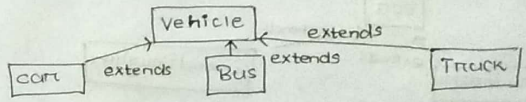


Son = Father + Son

Grandson = Father + Son + Grandson

→
```
Public class Grandson extends Son
{
  double weight;
  double height;
  String color;
  void play()
  {
    S.O.P("after playing study Java");
  }
  P.S.V.M {
    Grandson g1 = new Grandson()
  S.O.P (g1.money);
    g1.walking();
    g1.name = "amir khan";
    S.O.P (g1.name);
    g1.Spendmoney();
    g1.color = "Pink";
    S.O.P (g1.color);
    g1.play();
  }
}
```

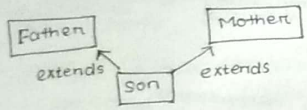
(iii) Hierarchical Level:-



→
```
Public class Car extends Vehicle
{
  String type;
```

```
void accelerate()
{
  S.O.P("Push the pedal to accelerate");
}
P.S.V.M() {
  Car c1 = new Car();
  c1.name = "nano";
  c1.type = "mini";
  S.O.P(c1.name + c1.type);
  c1.startEngine();
  c1.accelerate();
}
```

(IV) Multiple Level Inheritance:-

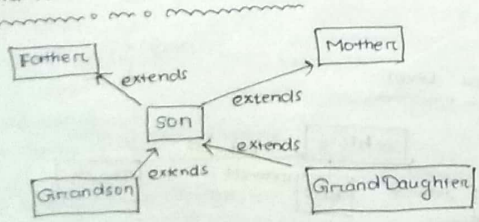



```
Class Father
{
  void care()
  {
    S.O.P("Father cares");
  }
}

Class Mother
{
  void care()
  {
    S.O.P("mother cares");
  }
}

Class Son
  extends
    Father, Mother
{
  P.S.V.M()
  {
    Son S1 = new Son();
    S1.care();
  }
}
```

(V) Hybrid-Level of Inheritance:-



Multiple + multi

Multiple + Hierarchal

Thu

→ Whenever the inherited method logic becomes outdated for child class, then we re-write the method declaration in child class and then we provide an updated Logic.

→ Writing the inherited method in child class and changing the logic is called method over-riding. OR

→

```
Public class Car
{
  void StartEngine() {
    S.O.P("Start Engine");
  }
  Void stopEngine()
  {
    S.O.P("Stop Engine");
  }
  Void drive(){
    S.O.P("drive at the speed of 100-120 kmph");
  }
}
Public class BMW extends Car
{ @Override
  void drive() {
    S.O.P("drive at the speed of 200-220Kmph");
  }
  P.S.V.M() {
    BMW b1 = new BMW();
    b1.startEngine();
    b1.stopEngine();
    b1.drive();
  }
}
```

O/p → Start Engine
Stop Engine
drive at the speed of 200-220Kmph

→ Void m() → Declaration
{
Statements → Implementation or definition
}

NOTE:-
→ With respect to method overriding the declaration of inherited method and the declaration of overriding method has to be same.
→ If not the method will not consider as specific method or specific behavior.

→ @Override:- Basically it is an annotation → we are providing the extra information to the compiler.

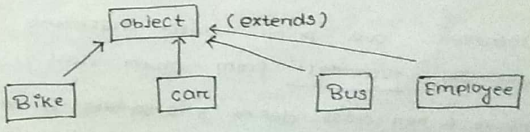
→ Public class mobile
{
Void call()
{
S.O.P ("call method");
}
Void camera()
{
SOP ("camera=0");
}
}
public class oneplus extends mobile
{ @Override
Void camera()
{
S.O.P ("camera=4");
}
P.S.V.M() {
oneplus o1 = new oneplus();
o1. camera();
}
}

O/p → camera=4

Q:- Create one Engineer class and write work method.
→ Class Engineer
{
work()
{
S.O.P("Engineer works");
}
Class Software_Engineer
{
work()
{
S.O.P ("Software works");
}
class civilEngineer
{
work()
{
S.O.P ("civil works")
}
P.S.V.M {
Software_Engineer s1 = new Software_Engineer();
s1. work();
}
}
O/p → Software works

19/03/2021

NOTE:-
→ In Java, whichever class we create for those classes, the parent is object class.
→ object class contains 11 methods inside it.
→ Even though the class we create is not using any extends keyword but the inheritance is happening internally.

→ 

→ The class which we create inherits 11 method from object class.
→ Right click → source → create object

tostring():-

```
Public class Student
{
  int id;
  String name;
  String branch;
  Public Student (int id, String name, String branch)
  {
    this.id= id;
    this.name = name;
    this.branch = branch;
  }
  @Override
  Public String tostring()
    {
      return " "+this.id+" "+this.name+" "+ this.branch;
    }
  PSVM ()
  {
    Student S1= new Student (1, "Ram", "EE");
    Student S2= new Student (2, "Bidya", "EEE");
  } S.O.P (S1);
}
```

↳ S1.tostring

O/p → 2 Bidya EEE

```
Public class ~~tostring~~ object
{
  Public String tostring
    {
      Logic for providing object address
    }
}
```

→ Whenever we print object reference internally it will call tostring() from object class.

Q:- Create a pen class. define 3 properties. Brand, name, color. Initialize the property using constructor. Override tostring() Create minimum 3 object and print the object addresses (States).

Non-primitive Data Types

22/03/21
Mon

→ In case of storing object information the 8-primitive data types can't be used, hence we go for user-defined data type or non-primitive data type.

→ Class serves 2 purpose

(i) Blue-print design / plan.

(ii) User-defined data type / Non-primitive datatype.

→
```
Public class Vendingmachine
{
  coffee button1()
    {
      coffee c1 = new coffee();
      return c1;
    }
  tea button2()
    {
      tea t1 = new tea();
      return t1;
    }
  P.S.V.M ()
    {
      Vendingmachine V1 = new Vendingmachine();
      S.O.P (V1.button1());
      S.O.P (V1.button2());
    }
}
```

```
Public class coffee
{
}

Public class tea
{
}
```

O/p → coffee@15db9742
tea@6d06d69c

→
```
public class conductor
{
  ticket issue()
    {
      ticket t1 = new ticket()
      return t1;
    }
  P.S.V.M ()
  {
    conductor c1 = new conductor();
    S.O.P (c1.issue());
  }
}
```

```
public class ticket
{
}
```

O/p → ticket@15db9742

```
Public class Atm
{
  Money dispense()
  {
    Money m1 = new Money();
    return m1;
  }
  P.S.V.M()
  {
    Atm a1 = new Atm();
    S.O.P(a1.dispense());
  }
}
```

O/p → Money@15db9742

```
Public class Money
{
}
```

```
Public class CounterStrike
{
  Knife button1()
  {
    Knife k1 = new Knife();
    return k1;
  }
  gun button2()
  {
    gun g1 = new gun();
    return g1;
  }
  bomb button3()
  {
    bomb b1 = new bomb;
    return b1;
  }
  P.S.V.M()
  {
    CounterStrike c1 = new CounterStrike();
    S.O.P(c1.button1());
    S.O.P(c1.button2());
    S.O.P(c1.button3());
  }
}
```

O/p → Knife@15db9742
gun@6d06d69c
bomb@7852e922

```
Public class Knife
{
}
Public class gun
{
}
Public class bomb
{
}
```

Package:-

23/03/2021 Tues

→ Packages in Java are nothing but folders, which contains Java resources in it.

→ Uses :- i) To achieve modularity.
ii) To avoid naming conflict.
iii) Resources are arranged in organized manner.
iv) Maintenance become easier.
v) we can achieve quick access to the resource.

Folder Creation:-

→ Right click on source, goto new click on folder & provide a folder name.

→ ctrl+N, type Folder. Double click on folder and provide a folder name. (Package1)

```
→ Package Package1
Public class student
{
  int id;
  String name;
  Void Study()
  {
    S.O.P("Study for 3 hours");
  }
}
```

```
Public class test
{
  P.S.V.M.()
  {
    Student S1 = new Student();
    S.O.P(S1.Study());
    S1.id = 1
    S1.name = "bidya";
    S.O.P(S1.id + S1.name);
  }
}
```

O/p → Study for 3 hours
1bidya

Access Modifiers:-

→ Access modifiers are basically used to provide access permission and restrictions. (States and behaviour).

(i) Public:- If the properties and behaviors are prefixed with public keyword meaning those properties and behaviour can be accessed,
i) Same class
ii) Different class same package.
iii) Different class different package.

→ Package package2
import package1.Student;
Public class test
{
public static void main()
{
Student s = new student()
s.id = 1;
s.name = "bidya";
S.O.P (s.id+" "+s.name);
s.study();
}
}

NOTE:-

→ When two classes are in different package and we want to import access the properties and behaviour from one class to another class, at 1st we need to import the package and then we need to get the access permission. (public).

(iii) default:- If the properties and behaviours are not prefixed with any keyword the access modifier is default.

→ If the properties and behavior, and access modifier is default means we can access those properties and behaviour with in same class and different class some package, but not different class different package.

(iii) Private:- If the properties and behaviours are prefixed with private keyword, means we can access within the same class only.

(iv) Protected:- If the properties and behavions are prefixed with protected, then we can access them only through inheritance. (childclass).

→ Package package1;
Public class King
{
Protected int money = 4000000000;
Protected void RuleKingdom()
{
S.O.P("Rule the Kingdom");
}
}

Package package2
import package1.King;
Public class Prince extends King
{
PSVM()
{
Prince P1 = new Prince();
S.O.P (P1.money);
~~S.O.P~~ P1.RuleKingdom();
}
}

25/03/2021
Wed

*. For importing Packages:-

import packagename.classname;

Constructor Chaining:- CASE-01

→ It is the process of one constructor calling another constructor.

→ The chaining can happen from Child class to Parent class or with in the same class.

→ Public class father
{
father()
{ //super()
S.O.P("Parent constructor");
}
}

Public class son extends father
{
son()
{ //super()
S.O.P("Child constructor");
}
P.S.V.M() {
son s1 = new son();
}
}

o/p → Parent constructor
Child constructor

→ super() → Call immediate parent class constructor. Inside each and every constructor super() calling statement is present internally or implecetly.

→ Public class Father

```
{
  Father(int i)
  {
    // super() - internally present.
    S.O.P("Parent constructor");
  }
}
```

Public class Son extends Father

```
{
  Son()
  {
    super(10); - explicitely
    S.O.P("child constructor");
  } P.S.V.M(){
      Son s1 = new Son();
}
```

O/p → Parent constructor
child constructor

NOTE:-

(i) When we have Parameterrmized constructor in super class i.e. excluding zero parameter constructor then compulsory child class the constructor chain should happen explicetely, calling the Parameterized parent constructor, from parent class.

(ii) Whenever we have overloaded constructor in parent class including zero parameter constructor, here compulsory child class explicit constructor chaining is not necessary.

CASE-03 (explecit constructor chaining)

→ Public class Father

```
{
  Father(int i)
  {
    // super (Internal)
    S.O.P ("Parent constructor-1");
  }
  Father()
  {
    //super
    S.O.P("Parent constructor");
  }
}
```

Public class Son extend Father

```
{
  Son()
  {
    super(10);
    S.O.P ("Child constructor");
  }
  Public.S.V.M()
  {
    Son s1 = new Son();
  }
}
```

O/p → Parent constructor-1
child constructor.

NOTE:-

→ Whenever we are performing explecit constructor chaining we should make sure that the parameter rules are followed.

CASE-04 (Overload constructor Chaining)

→ Public class Son extends Father

```
{
  Son()
  {
    S.O.P("child constructor-1");
  }
  Son(int i)
  { super(100);
    S.O.P("child constructor-2");
  }
  P.S.V.M() {
    Son s1 = new Son(10);
  }
```

O/p → Parent constructor-2
Child constructor-2

CASE-05

→ Public class Car

```
{
① Car()
  {
    this(10);
    S.O.P("1");
  }
② Car(int i)
  {
    this("A");
    S.O.P("C");
  }
⑤ Car(int i, String s) {
    this S.O.P ("3");
  }
③ Car(String s)
  {
    this ("A", 20);
    S.O.P("7");
  }
④ Car(String, int i)
  {
    this (10, "A")
    S.O.P("10");
  }
  PSVM() {
    Car c1 = new Car();
  }
}
```

O/p → 3
10
7
6
1

FRI

→ Class Vehicle

```
{
⑥ vehicle()
   {
    S.O.P("108");
   }
⑤ vehicle(int x, string y)
   {
    this();
    S.O.P("2005");
   }
④ vehicle(string s)
   {
    this(10,"s");
    S.O.P("1000");
   }
```

O/P→ 108
2005
1000
255
201
25

Class Cart extends Vehicle

```
{
① Cart()
  {
   this("s");
   S.O.P("25");
  }
  Cart(int i)
  {
   S.O.P("45");
  }
  Cart(int i, string s)
  {
   S.O.P("100");
  }
② Cart(string s)
  {
   this("s", 100);
   S.O.P("201");
  }
③ Cart(string s, int i)
  {
   super("s");
   S.O.P("255");
  }
  P.S.V.M()
  {
   Cart c1 = new Cart();
  }
}
```

→ Class Vegetable

```
{
② Vegetable()
④ {//super();
   S.O.P("Parent Constructor 1");
  }
  {
   S.O.P("Parent block-1");
  }
  Vegetable(int i)
  {//super();
   S.O.P("Parent constructor-2");
  }
  {
   S.O.P("Parent block-2");
  }
```

Class Cannot extends Vegetable

```
{
① Cannot()
  {//super();
   S.O.P("child-constructor 1");
  }
  {
   S.O.P("child-block 2");
  }
③ Cannot(int i)
  {//super();
   S.O.P("child constructor 2");
  }
  {
   S.O.P("child-block 1");
  }
}
Class maincodes
{
 P.S.V.M(String[] args)
 {
   Cannot c1 = new Cannot();
   Cannot c2 = new Cannot(55);
 }
}
```

O/P→ Parent block-1
Parent constructor 1
child-block 2
child-constructor 1
Parent block-2
Parent constructor-2
child

⇒ Parent block-1
Parent block-2
Parent constructor 1
child block 2
child block 1
child constructor-1
Parent constructor-2
child constructor-2

Parent block-1
Parent block-2
Parent constructor-1
child block 2
child block 1
child-constructor 1
Parent block-1
Parent block-2
Parent constructor-2

child block 2
child block 1
child constructor 2

Static Variables

08/03/2021
Mon

→ Global variable prefixed with static keyword is nothing but static variable. Here the no. of copies created is only one for each static variable.

→ Static data members are also known as class variables

NOTE:-

→ Static data members are related to class, and non-static data members are related to object.

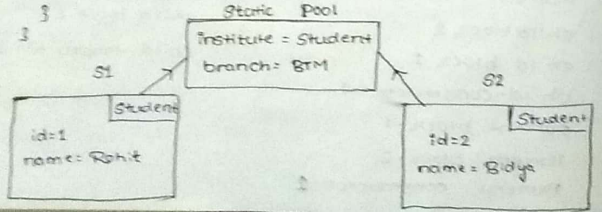
→ Static members can be directly accessed within the same class.

→ Static members can also be accessed in different class by using class name, without creating object.

```
→ Public class Demo
  {
    Static int a=100;
    int b=900;
    P.S.V.M() {
      S.O.P(a);
    }
  }
```

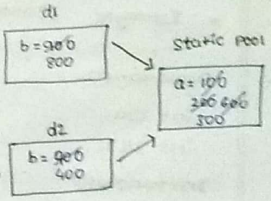
```
Public class Test {
  Public S.V.M() {
    S.O.P(Demo.a)
  }
}
```

```
→ Public class student {
    int id;
    String name;
    Static String institute = "Jspiders";
    Static String branch = "BTM";
    PSVM() {
      Student S1 = new Student(); S1.id=1; S1.name: "Rohit";
      Student S2 = new Student();
      S1.id=2;
      S2.name = "Bidya";
      S.O.P (S1.id+" "+S1.name+" "+S1.institute+" "+S1.branch);
      S.O.P (S2.id+" "+S2.name+" "+S2.institute+" "+S2.branch);
    }
}
```



Prove of Static Variable has one copy

```
→ Public class Demo {
    Static int a=100;
    int b=900;
    P.S.V.M() {
      a=200; (without using int/object reference)
      S.O.P ("Static"+" "+a);   // Static 200
      Demo d1=new Demo();
      d1.b=800;
      System.O.P("d1:"+" "+d1.b);   // d1 = 800
      Demo d2=new Demo();
      d2.b=400;
      S.O.P ("d2="+" "+d2.b); // d2=400
      d1.a=600;
      d2.a=800;
      S.O.P (d1.a+" "+d2.a); // 800 800
      S.O.P (d1.b+" "+d2.b); // 800 400
    }
  }
```



30/03/2021
TUES

Static Method :-

```
→ Public class Demo
  {
    Static void display() {
      S.O.P ("This is static method");
    }
    PS.V.M() {
      display();   // This is static method
    }
  }

Public class test
  { PSVM() {
    Demo.display(); // This is static method
  }}
```

Arrays

→ It is basically a container which holds data of same data type OR it is basically collection of elements or data of same datatype.

→ [n-data ⟶ 1 variable.]

→ Syntax:-

*. Datatype []arrayName = new Datatype [Size]
                                  ↳ Length

*. Datatype []arrayName = { $V_1, V_2, V_3 \ldots$ }

→ Declaration

```
int []a;
int b[];
```

Instanciation

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

Initialization

```
a[0] = 10;
a[1] = 20;
a[2] = 30;
a[3] = 40;
a[4] = 50;
```

a[5] = 100; ✗

OR

```
int a[] = {10,20,30,40,50};
```

a

10	20	30	40	50
a[0]	a[1]	a[2]	a[3]	a[4]

length = 5

last index of array = length-1
= 4

01/03/2021
Thu

LOOPS:-

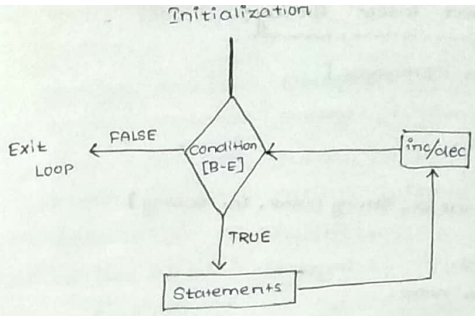
→ Whenever we want to execute a piece of code again and again continously we will go for loops.

For LOOP:-

→ Whenever the number of repeatation is known as we go for LOOP.

Syntax:-

```
for ( initialization ; Boolean expression ; inc/dec )
```



→
```
Public class Test
{
  PSVM (){
    int a[] = new int[6];
    a[0] = 10;
    a[1] = 20;
    a[2] = 30;
    a[3] = 40;
    a[4] = 50;
    for (int i=0; i<=4; i++)
    { S.O.P (a[i]);
    }
  }
}
```

O/P → 10
20
30
40
50 [Forward]

```
for( int i=4; i>=0; i--)
{
  S.O.P(a[i]);
}
```

O/P → 50
40
30
20
10 [Reverse]

→
```
Public class Test1
{
  P.S.V.M ()
  {
    String bike[] = {"DUKE", "RE", "Ninja", "kawazaki", 'R15"};
    S.O.P (bike.length);
    S.O.P ("Forward direction");
    for (int i=0; i<bike.length ; i++) {
        S.O.P (bike[i]);
    }
    for (int i= bike.length-1; i>=0; i--) {
        S.O.P(bike[i]);
  } } }
```

O/P → DUKE
RE
Ninja
kawazaki
R15

R15
kawazaki
Ninja
RE
DUKE

Storing Object inside [illegible]

```
Public class Employee {
  int id;
  String name;
  int salary;
public Employee(int id, String name, int salary){
    super()
    this.id = id;
    this.name = name;
    this.salary = salary;
  }
  @Override
  public String toString(){
     return "Employee [id=" + id + ", name=" + name + ",
                        " salary=" + salary + "]"

Public class Test2 {
    P.S.V.M() {
    Employee e1 = new Employee (1, "Rappen", 2000);
    Employee e2 = new Employee (2, "Banu", 4000);
    Employee e3 = new Employee (3, "Bidya", 5000);
    Employee emp[] = {e1, e2, e3}
    For (int i=0; i<emp.length; i++) {
        S.O.P (emp[i]);
        }
    }
}
```

02/04/2021
FRI

```
Public class student {
    int id;
    String name;
    public student (int id, String name) {
      this.id = id;
      this.name = name;
    }
    @Override
    public String toString(){
        return this.id + " " + this.name;
    }
}
```

```
Public class Test2 {
    P.S.V.M() {
      Employee e1 = new Employee (1, "Rappen", 2000);
      Employee e2 = new Employee (2, "Banu", 4000);
      Student s1 = new Student (1, "Rahul");
      Student s2 = new Student (2, "Raina");
      Object obj[] = {e1, e2, s1, s2};
      For (int i=0, i<=obj.length-1; i++){
          S.O.P (obj[i]);
      }
    }
}
```

NOTE:-

→ With respect to array's we use length property to check the size of array.

WRapper Class

→ For each and every primitive data type we have corresponding inbuild classes and these class will work as wrapper and hence the name wrapper class.

→
byte → Byte	float → Float
short → Short	double → Double
int → Integer	boolean → Boolean
long → Long	char → Character

→ These wrapper classes are present insid java.long package and they are final classes.

→ Wrapper class can be used for 2 reason.

(i) primitive to object representation. (Non-primitive)

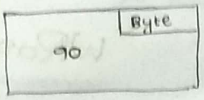
(ii) To convert String to actual primitive.

(i) Primitive to Object Representation :-

```
Public class wrapperdemo {
    Public static void main () {
        byte b = 75;
        int i = 1000;
        String s = "Wrapper-class"; (already non-primitive)
```

```
Boolean b1= TRUE ;
Short s = 5100;
Byte wb= new Byte (b);
Integer wi= new Integer (i);
Long wl = new Long (l);
Boolean wbl = new Boolean(b1);
Short ws = new Short(s);
Object obj[] = { wb , wi , wl , wbl , ws , s}
for (int i=0 ; i< obj.length ; i++)
   {
    S.O.P (obj [i]);
    }
  }
}
```

```
byte b=90;
Byte wb = new Byte(b);
```




```
int i= 200;
Integer wI = new Integer (i);
```

03th April 2021
SATURDAY

→ To convert String to Actual primitive :-

```
Public class ParseDemo {
    P.S.V.M () {
     int x = Integer. ParseInt ("10");
     int arr[] = {x};
     double d= Double. ParseDouble ("102");
     boolean b1 = Boolean. ParseBoolean ("true");
     boolean b2 = Boolean. ParseBoolean ("false");
     boolean b3 = Boolean. ParseBoolean ("anything");
     boolean b4 = Boolean. ParseBoolean ("100");
     int x1 = Integer. ParseInteger ("10.00");
     S.O.P (x);          o/p → number format Exception
     S.O.P (d);          10
     S.O.P (b1);         102.0
     S.O.P (b2);         true
     S.O.P (b3);         false
     S.O.P (b4);         false
} }                      false
```

→ ParseDouble :- It can take both decimal and non-decimal string value if anything else apart from decimal and non-decimal string value we get number format exception.

→ parseBoolean → For parse Boolean method if we pass "true" as a string value irrespective of case sensitive the method converts to primitive true. And also some for "false".

→ If we pass anything else apart from "true", "false" as string the method returns primitive false. (Default).

AutoBoxing :-

→ It is the process / mechanism of automatic conversion of primitive to its corresponding wrapper object

```
Byte wb= 100;
Integer wi= 1090;
Object obj[]= {wb,wi};
```

UnBoxing :-

→ It the process or mechanism of converting wrapper object to its corresponding primitive type.

```
int x = new Integer(50);
int arr[]= {x};
```

4.

```
class switch
{
 int price;
 String name;
Switch (String n, int p)
 {
  price =p;
  name =n;
 }

light (String n, int p)
 {
 // super (name,price)
 }

P.S.V.M() {
     light l1 = new light("A", 45);
```

```
class light extends switch
 light l1 = new light()
 M:price
 light ()
  {
  super ( "A", 5)
  }
 P.S.V.M() {
   light l1 = new light()
  }
```

TYPE-CASTING

Monday

→ It is the process or mechanism of converting a variable of one datatype into another datatype.

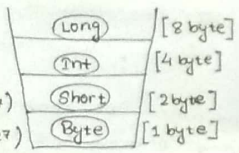
→ It is classified into 2 types.

(i) Primitive casting.

(ii) Non-primitive casting.

(i) Primitive casting	Non-primitive casting
→ Data widening.	→ UP-casting.
→ Data Narrowing.	→ Down-casting

(i) **Primitive casting**:- Process of converting one primitive data type to another primitive data type is called primitive casting.



→ Primitive casting is of 2 type again.

(a) Data Widening.

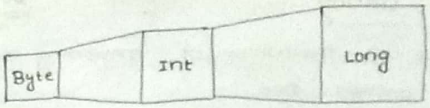
(b) Data Narrowing.

(a) **Data Widening** :- The process of converting a smaller primitive data type to a larger primitive data type is called as Data widening.

→ It is also known as implicit casting i.e. the conversion happens automatically.

→ Public class castingDemo {

```
P.S.V.M () {
  byte b = 100;
  int i = b;
  int I = i;
  S.O.P (b);                    %p→ 100
} S.O.P (i); S.O.P(I);              100
                                    100
```



Widening → L.H.S ≥ R.H.S

```
int i = 1000;
long l = i;
S.O.P(l);
```

L.H.S. ≥ R.H.S

(ii) **Data Narrowing** :- The process of converting a higher datatype into a lower datatype is known as Data Narrowing. Here the conversion is not happening automatically it should done explicitly.

NOTE:- While converting higher data type to lower data type there may be chances of data loss. (Reading the Range of specific datatype)

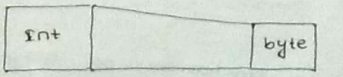
→
```
P.S.V.M() {
    int i = 130;
    byte b = i ; (No-auto conversion) ( L.H.S. ≤ R.H.S)
    byte b = (byte) i ;  ( L.H.S. ≤ (type of L.H.S) R.H.S )
    S.O.P (i);
    S.O.P (b);

    %p → 130
         -126
```

→
```
int i = 120;
byte b = i ;   // 120
```

→
```
int Y = 200;
double d = Y;  // 200.0
```

```
int  i = 128 ;
byte b = i ;   // -128
```
[Special case]



Narrowing → L.H.S. ≤ (type of L.H.S.) R.H.S

Non-Primitive Casting

06th April 2021 TUESDAY

→ It is the process or mechanism of converting one non primitive D.T to another one.

→
```
Public class counterstrike {
    Weapon pressbutton() {
        int score = 100
        if (score >0 && score <=100)
          {
            Knife K = new Knife();
            return K;
          }
        else if (score >100 && score <=300)
            {
              gun g = new gun();
              return g;
            }
        else
            {
              bomb b = new bomb();
              return b;
            }
      P.S.V.M() {
          counterstrike C1 = new counterstrike();
          Weapon W = C1.PressButton();
          S.O.P (W);
        }
    }
```

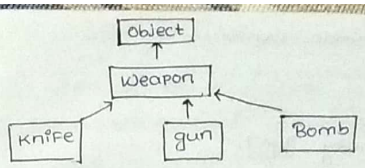
```
Weapon W = cs.pressbutton
Weapon W = new Knife()
```

↓ Object reference ↓ Object creation

[LHS ≥ R.H.S]

UP CASTING:-

→ It is the process or mechanism of assigning a subclass object to superclass reference.




```
Weapon W1 = new Knife()
Weapon W2 = new gun()
Weapon W3 = new bomb()
```
} [LHS ≥ RHS]

→ Food

Momos Pizza Panipuri

```
Food F1 = new Momos()
Food F2 = new pizza()
Food F3 = new Panipuri()
```

Characteristics of Upcasting:-

→ W.r.t. upcasting using super class reference we can only access the super class properties and super class behaviours as well as using super class reference we can access overriden method.

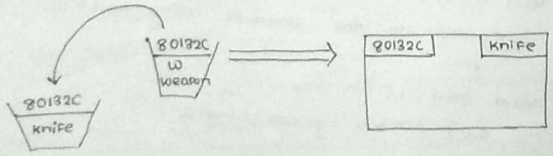
→
```
Public class Weapon {
    Void Kill() {
        S.O.P ("Kill the enemies using weapon");
      }
    Void changeweapon(){
        S.O.P ("Gun" + " " + "Bomb" + " " + "Knife");
      }
Public class Knife extends Weapon {
    Void Kill() {
        S.O.P ("Kill the enemies using knife");
      }
    Void stab() {
        S.O.P ("Kill by stabbing");
      }
```

```
public class gun extends weapon
 {
  void kill()
   {
    S.O.P ("kill by gun");
   }
  void shoot()
   {
    S.O.P (" by shooting");
   }

Public class bomb extends weapon()
 {
  void kill()
   {
    S.O.P (" kill by using Bomb");
   }
  void blast()
   {
    S.O.P (" blasting bomb");
   }

public class counterstrike {
   P.S.V.M() {
   counterstrike cs = new counterstrike();
   Weapon w = cs. pressbutton();
   knife k = (knife) w;                    w.stab() (x)
   k.stab();
  }
 }

Weapon w = cs. pressButton()
Weapon w = new knife();
knife k = w;     LHS > R.HS (x)
knife k = (knife) w; (Down casting)
```



Down Casting:-

→ It is the process or mechanism of assigning super class reference which containing the object address of child object to child class reference.

→ To perform down-casting, up-casting has to be done, if not we get class cast exception.

`Bomb b = (Bomb) w;` [NO upcasting of Bomb done]

→ When we perform upcasting using super class reference we can't access subclass property and behaviour. Hence we need sub-class reference, in order to do this we need downcasting.

07/04/2021
Wed

Dynamic Method Dispatch

→ Whenever we have multiple overriden methods, when the overridden method is called, the method execution and implemenation depends upon the object created. OR the behavior of the method / implementation of the method depends upon invoking object.

```
→ Public class Engineer {
     Void work() {
       S.O.P ("Engineer works");
     }
     P.S.V.M (String[] args) {
       Engineer e = new SoftwareEngineer();
       e.work();                          o/p →
     }                                    software engineer works
   }
Public class SoftwareEngineer extends Engineer {
   @Override
   void work() {
     S.O.P ("software engineer works");
   }
   }
```

Method Binding

→ It is the process or mechanism of attaching the method call with respective method implementation or method Body.

→
```
class Bike
  {
    Void throttle()
     {
      S.O.P ("Raise the throttle to move faster");
     }
    Void StartEngine()
     {
      S.O.P ("Start Engine");
     }
    Void StopEngine()
     {
      S.O.P ("Stop Engine");
     }
  P.S.V.M (—)
   {
    Bike b1 = new Bike();
    b1.StartEngine();
   }
```

(Binding)

→
```
Void m()
{
}
```

→ Declaration

Implementation
Definition
Body

POLYMORPHISM

→ Poly → Many
Morphe → Form

→ Anything in JAVA with multiple forms can be called as Polymorphism.

→ We have 2 types Polymorphism.
(a) Compile-Time Polymorphism. → Method overloading.
(b) Run-time Polymorphism. → Method overriding.

(a) **Compil-Time Polymorphism:-** Here the method binding / method resolution is decided by the compiler at the compile time. Hence it is called as compile time Polymorphism. It can achieved by method overloading.

→
```
public class Addition {
    Void add()
    {
     S.O.P (" This is add method");
    }
    Void add (int x, int Y)
     { res = x+Y;
      S.O.P (res);
     }
    Void add (int i, int j, int K)
     {
       res = i+j+K;
       S.O.P (res);
     }
  P.S.V.M (—) {
    addition a = new addition();
    a.add (10,20);
    a.add();
    a.add (10,20,30);
  }
```

O/P → 30
This is add method
60

→ In the above program we have 3 overridden method. Which method will execute at what time depends upon 2 criteria.

i) Method call.

ii) Based on the argument passed.

→ Based on this two criteria the compiler will take decision on the method binding.

→ And hence the method decision will takes place at compile time. Therefore the name compile time polymorphism.

(b) Run-Time Polymorphism :-

→ It can be achieved through method overriding.

→ Here

```
public class Engineer {
    void works() {
        S.O.P ("Engineer works");
    }
    P.S.V.M ( ) {
        Engineer e = new SoftwareEngineer();
        e.work();
    }
}
public class SOFTwareEngineer extends Engineer {
    @override
    Void work() {
        S.O.P ("software engineer works");
    }
```

→ In the above program we have overridden method i.e. method overriding the method binding or method resolution will be taken by the JVM at Run time & Hence the name Runtime polymorphism.

→ OR, in method overriding the decision Factor is object & the decision maker is JVM & the decision will taken at Run time. Hence this call as Run-time polymorphism.

07/04/2021
Thu

ENCAPSULATION :-

→ It is the process of or mechanism of binding up of private data members with the respective data handling methods (getter and setter methods).

Advantages:-

→ We can make the data read only and write only.

→ To achieve data security.

→ To avoid unauthorized access.

→ To achieve validation of data.

To achieve all the above we encourage indirect access and avoid direct access.

Rules for Encapsulation or Java Bean Specification:-

→ Class must be public and Non abstract.

→ States or data members should be private.

→ For each private data members there should be public getter and setter methods.

→ There should not be any user defined constructor.

Rules for getter and setter Method :-

→ The return type of setter methods should always be void and should have one parameter.

→ The return type of getter methods will always be the respective data type (same as respective state)

→ Public class Student {

```
    private String name;
    private int age;
    private double perc;
```

```
// setter Methods
public void setname (string name) {
    this.name = name;
}
public void setage (int age) {
  [validation]
  if (age > 0 && age <= 100)
  {
    this.age = age;
  }
  else
    S.O.P ("enter valid age");
}
public void setperc (double perc) {
  [validation]
  if (perc > 0 && perc <= 100)
    this.perc = perc;
  else
    S.O.P ("enter valid percentage");
}
// Getter Methods
public string getname() {
    return this.name;
}
public int getage() {
    return this.age;
}
public double getperc() {
    return this.perc;
}
public class website {
    p.s.v.m(__) {
    student s1 = new student();
    s1.setname ("Bidya");
    s1.setage (22);
    s1.setperc (94);
    S.O.P (s1.getname());
    S.O.P (s1.getage());
    S.O.P (s1.getperc());
```

09/04/2021
FRI

To make the data Read Only

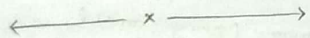
```
public class car {
    private int price = 30000;
    private string name = "Honda";
    public int getprice() {
        return price;
    }
    public string getname() {
        return name;
    }
}
```

To make the data write only

```
public class car {
    private int price;
    private string name;
    public void setprice (int price) {
        this.price = price;
    }
    public void setname (string name) {
        this.name = name;
    }
}
```

```
public class Employee {
    private int id;
    private string name;
    private string company;
    public int getId() {
        return id;
    }
    public void setID (int id) {
        this.id = id;
    }
    public string getName() {
        return name;
    }
    public void setName (string name) {
        this.name = name;
    }
```

```
public String getCompany() {
    return company;
}
public void setCompany(String company) {
    this.company = company;
}
public class Test {
    PSVM(--) {
    Employee e1 = new Employee();
    e1.setID(1);
    e1.setName("bidyo");
    e1.setCompany("space-x");
    S.O.P (e1.getID());
    S.O.P (e1.getName());
    S.O.P (e1.getCompany());
    }
}
```



JAR File

→ JAR means JAVA ARCHIEVE.

→ Here we are going to compress many file into one.

Steps to create JAR FILE:-

→ Right click on project → export → Type 'JAR' → Double click on JAR file (ICON) → Browse and select the destination place to save the JAR file (JAR file should be in .jar extension)

Steps to import JAR:-

→ We can import the JAR in 2 ways.

(i) Internally.

(ii) Externally.

(i) Internally → Right click on project → click on new → new Folder (name 'lib') → copy the JAR file and paste it on JAR Folder → Right click on project → properties → click on JAVA build path → libraries → Add Jar → Browse the respective Folder and select JAR.

(ii) Externally → Right click on project → Properties → click Java build path → libraries → add external JARS → Browse and select JAR file.

10/04/2021
Sat

1. Can we make a constructor as Final (NO)
   as Static (NO)
2. Can we inherit Static (NO) - using child class ref we can access the parent class static behavior.
3. Static won't provide polymorphism behavior.
4. Can we make constructor as Final?
   No, because we can't override constructor, we can't inherit constructor.
5. Why can't inherit a constructor?
   Because constructor name & class name should be same.
6. Can we override Static method?

NO, we can't override static method because static methods will not provide any polymorphism behavior.

→
```
Public class Car {
    static void drive() {
        S.O.P ("Car travels at 120kmph");
    }
}
PSVM (—) {
    Car C = new Car
Public class BMW {
    static void drive() {
        S.O.P ("BMW travels at 170kmph");
    PSVM (—) {
        Car C1 = new Bmw();
        C1.drive();
    }
}
```
O/P → Car travels at 120kmph

→ Here both the drive methods present in parent class & the child class are static. And in the above program the implementation of the methods are not depending upon the object, they depending upon the class reference type, so hence we are not getting any polymorphism behaviors. (static can't be overridden.)

NOTE:-
@Override anotation is not supported by static methods

→
```
Public class Bmw extend Car {
    Static void drive() {
        S.O.P ("BMW travels at 170kmph");
    }
    PSVM (—) {
    Car C = null;
    C.drive();
    S.O.P(C);
}
```
O/P → Car travels at 120kmph
null

12/04/2021

Exception :-

→ It is basically a run-time error or abnormal event which interupt or disturb the normal program flow execution.

→ e.g → Arithmetic Exception.
Number Format Exception.
Class-Cast Exception.

→
```
Public class Demo {
    PSVM (—) {
        S.O.P ("Statement-1");
        S.O.P ("Statement-2");
        S.O.P ("Statement-3");
        S.O.P (1%);
        S.O.P ("Statement-4");
        S.O.P ("Statement-5");
```
O/P → Statement-1
Statement-2
Statement-3
Error (Arithmetic Exception Error)

Exception Handling

→ It is the process or mechanism of supressing an exception such that we get back our normal program flow execution.

→ We can achieve this by using 'try' and 'catch' Block.

→ Syntax:-
```
try
{
  // Instruction.
      (which tends to give Exception)
}
catch (Exception Name RefType)
{ // Error Message.
}
```

(i) try:- It is basically a block which may contain Risky code / Risky instruction. (Instruction which tends to give Exception).

(ii) Catch:- It is basically contains the instruction which is responsible for providing error message.

→
```
Public class Demo{
    PSVM (—) {
        S.O.P ("Statement-1");
        S.O.P ("Statement-2");
        try {
            S.O.P (10%); // Arithmetic Exception.
            }
        catch (ArithmeticException e)
            {
            S.O.P ("This is Arithmetic Exception");
            }
        S.O.P ("Statement-3");
        S.O.P ("Statement-4");
        }
```

O/P →
```
Statement-1
Statement-2
This is Arithmetic Exception.
Statement-3
Statement-4
```

NOTE:-

→ try and catch Block should always be associated together.

→ We can't write any instruction between try and catch block.

→ If an instruction which tends to give an exception,

→ When the program control reaches this code the program terminate. If this instruction is written inside the try block and if now the program control reaches that line it searches for matching catch Block. If the matching catch block is available, the catch block will execute other print statement.

→ If the matching catch block is not available then the program terminates.

→
```
Public class Demo {
    P.S.V.M (—) {
        S.O.P ("Statement-1");
        S.O.P ("Statement-2");
        try {
            S.O.P (10%); // Arithmetic Exception.
            }
        catch (NumberFormatException e) // No matching catch Block / Format
            {
            S.O.P ("This is a NumberFormat Exception");
            }
        S.O.P ("Statement-3");
        }
```

O/P →
```
Statement-1
Statement-2
Error (Exception Error)
```

→ In the above program, when the exception raised and matching catch block is not available, then the rest of the code will not executed.

NOTE:-

→ Once the control comes out of try Block again the control will not go back to the same try Block.

→
```
try {
    int i = Integer.parseInt ("Hello");
    }
catch (ArithmeticException e) // No Matching catch Block
    {
    S.O.P (" Number format exception");
    }
```

→ Program Terminates.

```
try {
  S.O.P (1/0);
}
Catch (NumberFormatException e) // NO Matching catch Block.
{
  S.O.P ("Arithmetic Exception");
}
```

```
Public class Demo{
    P.S.V.M (—) {
      S.O.P ("Statement-1");
      try {
        S.O.P ("try-1");
        S.O.P (1/0);
        S.O.P ("try-2");
      }
      Catch (ArithmeticException e)
      {
        e.printStackTrace(); // To print the type & Line of Exception.
      }
      S.O.P ("Statment-2");
    }
```

O/P → Statement-1
try-1
Arithmetic Exception in line 6.
Statement-2

14/04/2021
Wed

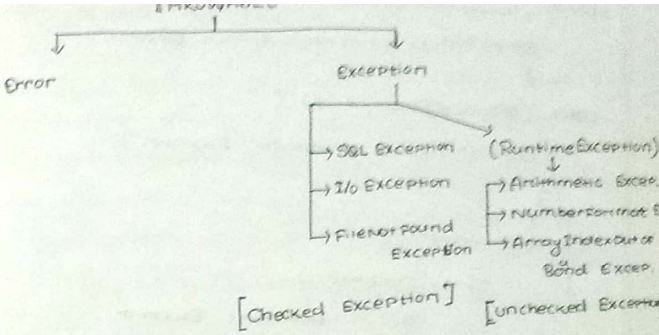
1. When to use Single catch Block ?

If the handling senario or the handler code is same for multiple exception, we go for single catch Block.

NOTE:-

→ W.r.t exception hierarchy the super class can handle all of its sub-class exception.

→ At a time only one Exception can arise.




```
Public class Demo{
    psvm (—) {
      S.O.P ("Statement-1");
      try {
        S.O.P (10/0);
        int i = Integer.parseInt("Hello");
      }
      Catch (Throwable e) {
        S.O.P ("There is an Exception");
      }
      S.O.P ("Statement-2");
    }
```

O/P → Statement-1
There is an Exception
Statement-2

2. When to write multiple Catch Block ?

If the handling senario or the handler code is specific for multiple exception we go for multiple catch Block.

```
Public class Demo{
    psvm (—) {
      S.O.P ("Statement-1");
      try {
        S.O.P ("10/0");
      }
```

```
Catch (NumberFormatException e) {
    S.O.P ("This is number format Exception");
    }
Catch (ArithmeticException e) {
    S.O.P ("This is Arithmetic Exception");
    }
S.O.P ("Statement-2");
}
```

O/P → Statement-1
This is Arithmetic Exception
Statement-2

15/04/2021
Thu

Sequence Of Multiple Catch Block

→
```
Public class Demo {
    PSVM (—) {
        S.O.P ("Statement-1");
        try {
            S.O.P (1%);
            int i = Integer.parseInt ("Hello");
            int a[] = {1,23,4}
            S.O.P (a[4])
        }
        Catch (Throwable e) {
            S.O.P ("Reached Throwable");
        }
        Catch (NumberFormatException e) {
            S.O.P ("This is NumberFormatException");
        }
        Catch (ArithmeticException e) {
            S.O.P ("This is ArithmeticException");
        }
        S.O.P ("Statement-2");
```

O/P →
Statement-1
Reached Throwable
Statement-2

→ In the above Program the Sequence of multiple catch block is from super class to sub-class.

→ In this case when the super class catch block can handle all the situation, the program control will never reach to the sub-class catch Block, then it become unreachable catch block, so hence the sequence of the catch block from sub-class to super class

→
```
Public class Demo {
    PSVM (—) {
        S.O.P ("Statement-1");
        try {
            S.O.P (10/0);
        }
        Catch (ArithmeticException e) {
            S.O.P ("This is Arithmetic Exception");
        }
        Catch (Throwable e) {
            S.O.P ("Reached Exception");
        }
        S.O.P ("Statement-2");
    }
```

O/P → Statement-1.
This is Arithmetic Exception.
Statement-2.

CHECKED AND UNCHECKED EXCEPTION

(i) Checked Exception :-

```
Public class checked-Unchecked {
    PSVM (—) {
        FileReader Fr = new FileReader ("Demo.txt");
    }
}
```

Complier will give ↓ (ERROR)

[We can handled by using 'try' & 'catch']

→ In the above program we have written a Risky code, which may tends to give an exception.

→ Here the complier ~~through~~ throws an error as un-handled exception type file not found exception.

→ It means here the compiler is checking whether the Risky code is handled or not.

→ The compiler forces the user to handle the exception mandatorily. Hence the type of exception which arises from the Risky code comes under checked exception catagory. (Because the compiler is checking whether the Risky code is handled or not).

Ex:- File not found Exception.
SQL Exception.
I/O Exception.

(ii) Unchecked Exception:-

```
Public class checked_unchecked {
    P.S.V.M(-) {
        S.O.P (1%0); // Error. (compiler won't give)
    }
}
```

→ In the above program we have return Risky code, and here the compiler is not checking whether the Risky code is handled or not

→ And the exception which arises from these type of Risky code we call it as unchecked Exception.

16/04/2021
FRI

Finally Block

→ It will execute even though the exception arises.

→ Basically it's use to close all the costly resources. Costly resources are those resources which consume system property.

Ex:- JDBC connection, Scanner etc.

→
```
Public class checkFinal {
    P.S.V.M(-) {
        S.O.P ("Statement-1");
        try {
            S.O.P (1%0);
        }
        catch (NumberFormatException e) {
            S.O.P ("catch-Block");
        }
        Finally {
            S.O.P ("Finally Block");
        }
        S.O.P ("Statement-2");
    }
}
```

O/P → Statement-1
Finally Block
Arithmetic Exception error

NOTE:-

→ In the above program the type of exception arises is Arithmetic Exception.

→ When the control comes out of try Block here the matching catch Block is not available, hence the program should get terminated. But since Finally Block is available, Finally Block will get executed before the program terminates.

→ try, catch and Finally should always be associated togeather.

→
```
psvm(-) {
    S.O.P ("Statement-1");
    try {
        S.O.P (1%0);
    }
    Finally {
        S.O.P ("Final Block");
    }
    S.O.P ("Statement-2");
}
```

O/P → Statement-1
Final Block
Error (Arithmetic Exception).

CUSTOM EXCEPTION

→ Whenever the inbuilt exception is not sufficient or not suitable for the senario, we can also define our own Exception and we can also catagonized to checked and unchecked exception.

→ To create a custom exception basically we need to create a class with a desired custom Exception name.

→ This class should extend either with throwable or exception or Run-time Exception class.

→ If the class is extending throwable or exception class, then the custom exception comes under checked exception catagory. (i)

→ If the class is extending Run-time exception then the custom exception comes under unchecked exception catagory. (ii)

→ Create Exception:-

```
Public Class BhanuException extends Exception {
    BhanuException () {
        Super();
    }
    BhanuException (String msg) {
        Super(msg);
    }
}
```

→ Call Exception:-

```
Public Class Test {
    P.S.V.M (-) {
        int i = 100;
        int j = 0;
        if (j==0) {
            try {
                throw new BhanuException ("This is custom
                                                Exception");
            }
            catch (BhanuException e) {
                e.printStackTrace();
            }
        }
        else {
            int res = i/j;
        } S.O.P (res);
```

O/p → Bhanu Exception
This is a custom Exception.

throw

→ This is basically a keyword which is used to raise an exception by throwing an exception object. Basically we use throw keyword inside the method implementation.

We can handle an exception in 2 ways

i) try and catch.
ii) using throws.

try and catch:- W.r.t try and catch we are handling an exception by catching the exception using respective catch block, so that even when exception arises the catch block will handle it and the respective code will execute.

throws:- We can handle an exception using throws key word also but here we are handling an exception by giving the indication to the user about the exception which may raise. And here if the exception arises there is no handler code (catch Block), and hence the program terminates and the rest of the code can't execute.

→ Throws keyword can give indication about multiple exception at one point of time.

→ import java.io.FileNotFoundException;

→ import java.io.FileReader;

```
public class Demo {
    P.S.V.M (-) Throws FileNotFoundException, Exception {
        S.O.P ("Main Start");
        FileReader Fr = new FileReader ("Demo.txt");
        S.O.P ("Main end");
    }
}
```

O/p → Main Start
File not found Exception Error.

ABSTRACTION

17.04.2021
Sat

```
Void display()
{
  // Statements
}
```

Declaration
Implementation
Defⁿ
Body

concrete method = Declaration + Implementation

```
Abstract void display();
```

Abstract method = Declaration + ~~Implementation~~

concrete method → Concrete class
concrete method → Abstract class

Abstract method → Abstract class
Abstract method → Interface

Concrete Method:-

→ It is the method which contains both the declaration and implementation.

→ It can be defined in two places.
(i) concrete class.
(ii) Abstract class.
(iii) (Interface.)

Abstract Method:- This method contains only the declaration part but no implementation.

→ Abstract method can define in 2 places.
(i) Abstract class
(ii) Interface.

(1) **Abstract class:-** It is the class with Abstract key word prefix.

→ **concrete class:-** class without abstract keyword is known as concrete class.

→ Abstract means incomplete.

```
Public abstract class Switch {
    abstract void switchON(); // Abstract method don't
    abstract void switchOFF();          specify Body.
}
Public class Television extends Switch {
    @override
    void switchON() {
        S.O.P (" TV gets turn ON");
    }
    @override
    void switchOFF() {
        S.O.P (" TV gets turned OFF");
    }
    Main (-) {
    Television t1 = new Television();
    t1.switchON();
    t1.switchOFF();
    }
Public class Projector extends Switch {
    @override
    void switchON() {
        S.O.P (" Projector gets turned ON");
    }
    void switchOFF() {
        S.O.P (" Projector gets turned OFF");
    }
    Main (__) {
    Projector P1 = new projector();
    P1.switchON();
    P1.switch OFF();
    }
}
```

→ Public abstract class ... {

Mon

abstract double getInterest();

}

Public class axisbank extends Bank {

@Override

double getInterest() {

return 9.2;

}

P.S.V.M (—) {

axisbank a1 = new axisbank();

S.O.P (a1.getInterest());

} O/p → 9.2.

}

Public class Sbibank extends Bank {

@Override

double getInterest() {

return 7.5;

}

P.S.V.M (—) {

Sbibank S1 = new Sbibank();

double t = S1.getInterest();

S.O.P (t);

} O/p → 7.5.

}

1. Can we make abstract class as Final?

NO, we can't make abstract class as final, because abstract class must be inherited.

2. Can we make abstract method as Final?

NO, we can't make abstract method as Final, because abstract method has to be overriden.

3. Can we made abstract method as static?

We can't override static methods so abstract method has to be overriden, hence we can't make the abstract method as static.

4. Can we create an object of abstract class?

We can't create an object of abstract class for the following 2 reason.

(i) Abstract means incomplete, so here the class itself is incomplete, hence we can't create object.

(ii) Abstract class may contain abstract method i.e. method with no implementation. And hence we can't create an object of abstract class.

5. Is it necessary to write minimum one abstract method inside abstract class?

NO, it is not necessary to have minimum one abstract method inside abstract class, we can also write only concrete methods inside abstract class.

6. Can we write constructor for abstract class?

Yes, we can create constructor for abstract class.

7. When we can't create object for abstract class, then what is the purpose of creating a constructor.

We use constructor to initialize the states of the abstract class

NOTE:-

→ We invoke/call the abstract class constructor through explicit constructor chaining.

→ Public abstract class switch {

int price;

String name;

Public switch (int price, String name) {

this.price = price; //65

this.name = name; // Anchor

}

}

Public class Anchor extends switch {

Anchor (int price, String name) {

super (price, name);

}

```
P.S.V.M (—) {
    Anchor a = new Anchor(45, "Anchor");
    S.O.P (a.price + " " + a.name);
}
```
o/p → 45 Anchor

```
abstract class Switch {
    abstract void m1();
    abstract void m2();
    "
    "
    "
    abstract void m10();
}
```

All 10 need to be

```
class Anchor extends Switch {
}
```

→ In the above abstract class we have 10 abstract method.

→ Here the Anchor class extending Switch class (abstract) whereever a child class is inheriting a parent class that contains abstract method in it. It is mandatory for child class to override all the 10 abstract methods. IF NOT, the child class becomes incomplete and we should make the child class as abstract.

```
abstract class Switch {
    abstract void m1();
    abstract void m2();
    "
    "
    "
    abstract void m10();
}
```

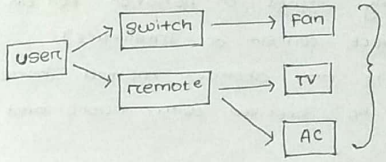
```
abstract class Anchor extends Switch {
    abstract void m1();
    abstract void m2();
}
```

INTERFACE

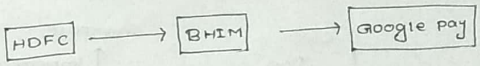
20/04/2021 Tues

→ Interface is an intermediate between service and consumer.

→ Basically we use interface in order to communicate between two entities.



[Consumer] [Interface] [Service]



→
```
Public interface ISwitch {
    void switchON();
    void switchOFF();
}
Public class Fan implements ISwitch {
    @Override
    Public void switchON() {
        S.O.P ("switchON Fan");
    }
    @Override
    Public void switchOFF() {
        S.O.P ("switchOFF Fan");
    }
    PSVM (—) {
        Fan F1 = new Fan();
        F1.switchON();
        F1.switchOFF();
    }
}
```
o/p → switchON Fan
switchOFF Fan

NOTE:-

→ Interface will not inherit any class. (Not even object class).

→ We can't create an object of interface because there is no constructor concept for interface.

→ We can't write any .states or behavior for an interface (There is no object concept of interface).

→ Whenever we write an abstract method inside interface by default it will be prefixed with public and abstract.

→ abstract class ↑extends abstract class

Class ↑extends Class

Abstract class ↑extends Class

interface ↑ ~~extends~~ ~~class~~ interface

Class ↑(x) interface

interface ↑ implements Class

1. Will the compiler generate .class file for interface?
Yes, .class file will generate for all type of class.

→
```
Public interface Bhim {
    void transferMoney (int money);
}
Public class HDFC implements Bhim {
    @Override
    Public void transferMoney (int money) {
        S.O.P ( Money + "  " + "Transferred sucessfully");
    }
public class phonepe {
    PSVM (—) {
        HDFC H = new HDFC();
        H. transferMoney (5000);
    }
}
```

O/P → 5000 transferred successfully