}.

OOPS.

(security)
Data-Hiding

① Security [Data hiding.

GUI.

User.

balance

withdraw

.java

class Account{

private double balance;

public void withDraw
(double balance){

// perform authentication.

transfer the money.

}.

}.

username
password.

"abstraction"
abstract class
interface

database

MySQL

Oracle

PostgreSQL.

database Name

table

col1 col2

row1

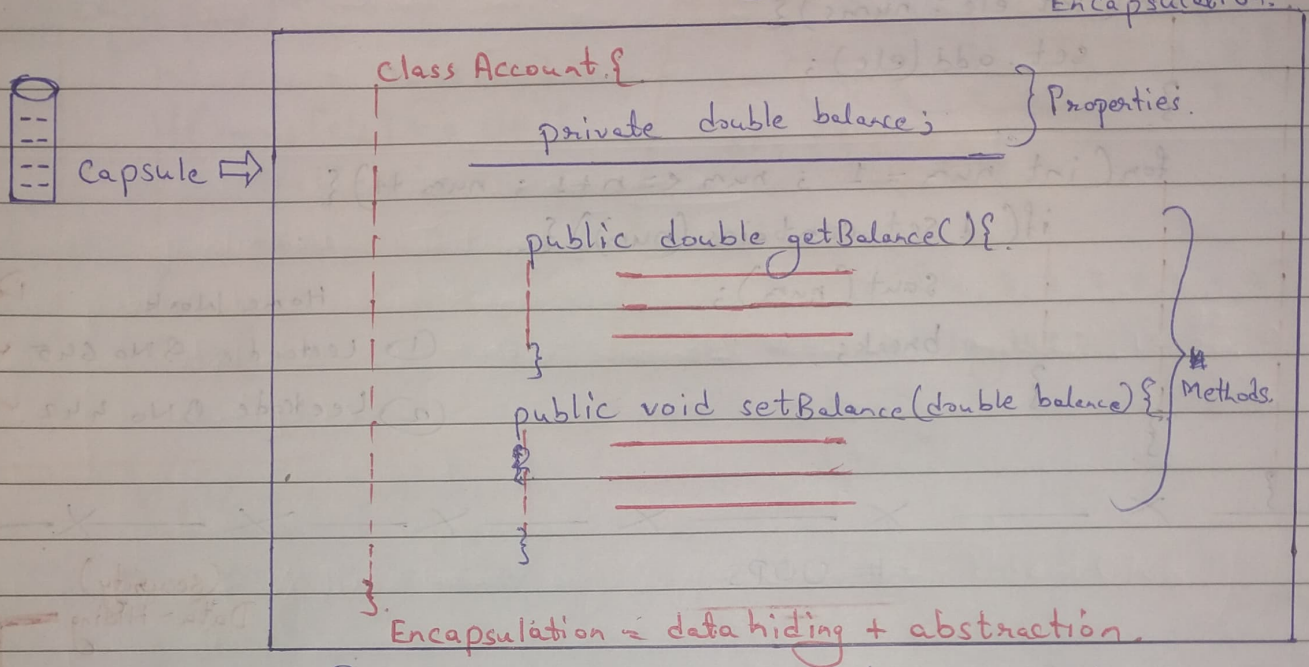
row2

row3.

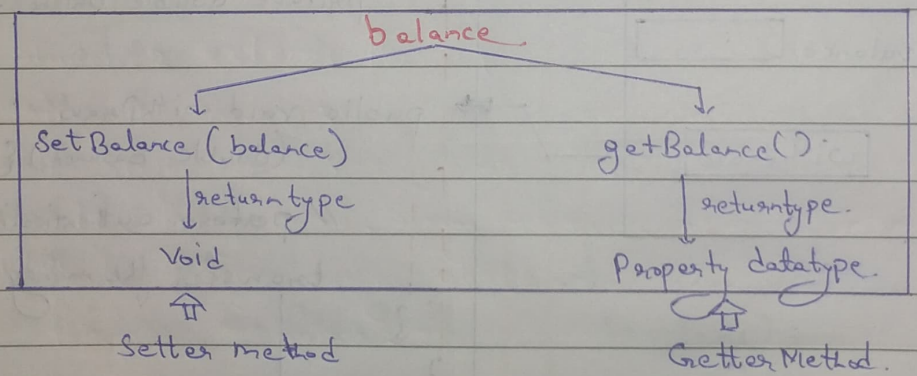
Our internal data should not go the outside world directly, that is outside person can't access our internal data directly is referred as "Data-hiding".

Hiding the internal implementation details, but exposing the set of services offered is technically called a "abstraction"

Encapsulation.



Binding data & corresponding methods into single unit is called "Encapsulation".



Pillars of Object Orientation

(a). Datahiding ⇒ Private

(b). Abstraction ⇒ abstract class + interface.

(c). Encapsulation ⇒ Datahiding + abstraction

(d). Polymorphism ⇒ inheritance

Compile Time Polymorphism
1. Method Overloading.
2. Method hiding.

Runtime Polymorphism
1. Method Overriding.

⇒ Code :-

```
Class Account {
    // Data Security.
    Private double balance;
    // Method : : public,
    public double getBalance(double balance){
        // Perform Authentication.
        boolean result = validate("sachin", "sachin123");
        //Withdrawing the Money.
        if(result == true) {
            this.balance = this.balance - balance;
            return balance;
        }
        else{
            // throw a meaningfull message to user.
            Sout("Invalid username/ password try again...");
            return 0.0;
        }
    }

    public void setBalance (double balance) {
        // Perform Authentication
        boolean result = validate("sachin", "sachin123");
        if(result == true){
            // Deposite The Money.
            this.balance = this.balance + balance;
            Sout("Credited to the account");
        }
        else{
            // Throw A Meaningful Message To the User.
            Sout("Invalid Username/Password Try Again");
        }
    }
    // Method : : private
    private boolean validate(String userName, String Password){
        // Logic for Authentication.
        return userName.equalsIgnoreCase("Sachin") && password.equals
                                                    ("sachin123");
    }
}

Public Class TestApp { } ——→ Next page
    psv main{
    }
```

```
Public Class TestApp{
    public static void main(String[] args){
        Account acc = new Account();
        acc.setBalance(1000.0);
        double balance = acc.getBalance(500);
        Sout("Withdrawing :: " + balance + "Amount");
    }
}
```

Inheritance

⇒ Code :-

```
public class TestApp1{
    public static void main(...){
        Student2 s1 = new Student2("sachin", "mumbai", 18, 92, "A");
        s1.disp();
    }
}

class Person{ // Base Class or Supper Class or Parent Class
    public String name;
    public String address; // Instance Variable
    public int age;
}

Class Student2 extends Person{ // Derived Class or Sub Class or Child Class
    public int marks;
    public String grade;
    Student2(String name, String address, int age, int marks, String grade){
// Constructor  Sout("Calling during object Creation....");
        this.name = name;
        this.address = address;
        this.age = age;
        this.marks = marks;
        this.grade = grade;
    }
    public void disp(){ // Normal Method
        Sout("Name is : " + name);
        Sout("Address is : " + address);
        Sout("Age is : " + age);
        Sout("Marks is : " + marks);
        Sout("Grade is : " + grade);
    }
}
```

Output :-

```
Calling during Object Creation....
Name is : sachin
Address is : mumbai
Age is : 18.
Marks is : 92
Grade is : A.
```

~~int~~ ⇒ Inheritance :-

① The process of acquiring the properties & behaviour of one class to another class is called "inheritance".

② In java, inheritance can be achieved in 2 forms
   (a). Is :- A (using extends keywords).
   (b). Has :- A (Declaring one ref variable inside another class).

③ The class which share properties / behaviours to another class is referred as " BaseClass / ParentClass / SuperClass ".

④ The class which uses properties / behaviours of another class is referred as " DerivedClass / ChildClass / subclass ".

Q. What is a Constructor ?

Ans :-→ Constructor is a method, which has got the same name as that of the classname.
→ While writing a constructor, we should not keep return type for the method.
→ Constructor gets called automatically at the time of object Creation.
→ Since constructor gets called automatically at the time of object creation, we use this constructor to initialize the instance variables of the Class.

Q. How many types of Constructors we can have in java Class?
⇒ (a) Zero Argument Constructor.
   (b) Parameterized Constructor.

Q. In Java how many types of variables are there?
⇒ (a) local Variable ⇒ variables declared inside the method are local.
   (b) Instance Variable ⇒ Variables which are declared inside the class but outside the method.
   (c) Static Variable ⇒ Variables which are declared inside the Class but outside the method with Static keyword.

(a) Zero Argument Constructor.

⇒ Code :-

```
Class Employee1{
    String name; // Instance Variable
    int age;
    Employee1(){ // zero argument Constructor
        Sout("Employee Constructor Called");
    }
    public void disp(){ // Instance Method
        Sout("Name is : " + name);
        Sout("Age is : " + age);
    }
}
public Class TestApp2{
    public static void main(...){
        //Usage of zero Argument Constructor
        Employee1 e1 = new Employee1();
        e1.disp();
    }
}
```

Output :-
Employee Constructor Called....
Name is : Null
Age is : 0.

Local Variable :-
⇒ Memory in stack area.
⇒ No default value.
⇒ Memory initialized at the time of method call.
⇒ Memory will be deactivated at the time of control leaving the method call.

Instance Variable :-
⇒ Memory in heap area.
⇒ Default value depending on datatypes.
⇒ Memory initialized at the time of object creation.
⇒ Memory will be deactivated at the time of object Destruction.

Static Variable :-
⇒ Memory in Method area ⇒ Default value depending on datatypes.
⇒ Memory initialized at the time of loading the .class file.
⇒ Memory will be deactivated at the time of unloading the .class file

#Local Variable

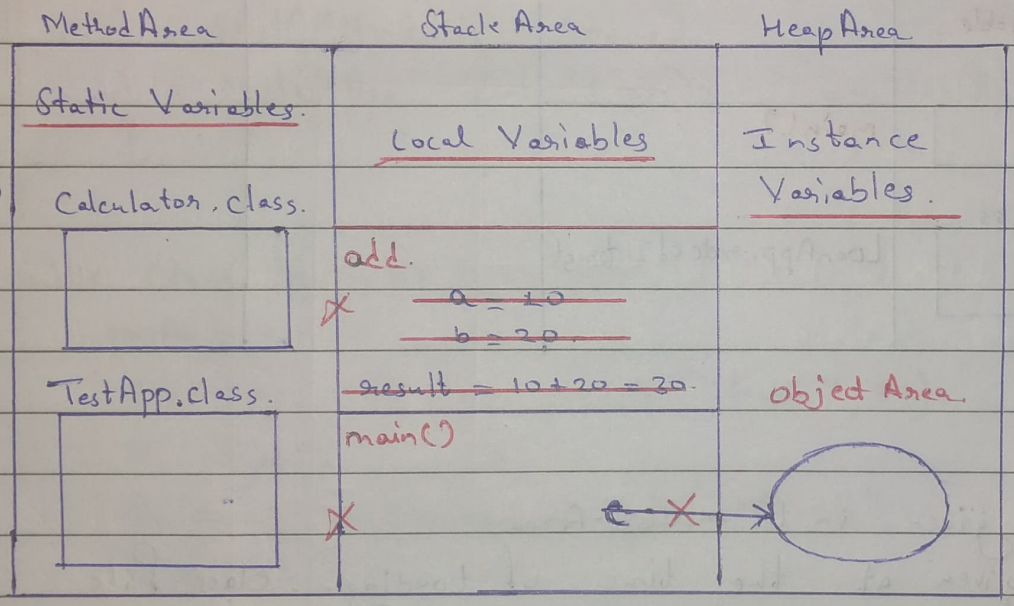
⇒ Code :-

```
Class Calculator{
    public void add(int a, int b){
        int result = a+b;
        Sout("The sum is : " + result);
    }
}

public class TestApp3{
    public static void main(String[] args){
        Calculator c = new Calculator();
        c.add(10,20);
    }
}
```

Output :-

The Sum is : 30.



⇒ Local Variable :-

→ Variables declared inside the method.

→ Memory will be given inside stack area.

→ Once the control enters inside the method memory will be ~~given~~ taken given.

→ Once the control leaves the method memory will be taken out.

→ No Default Value will be given to the local variable.

Static Variable.

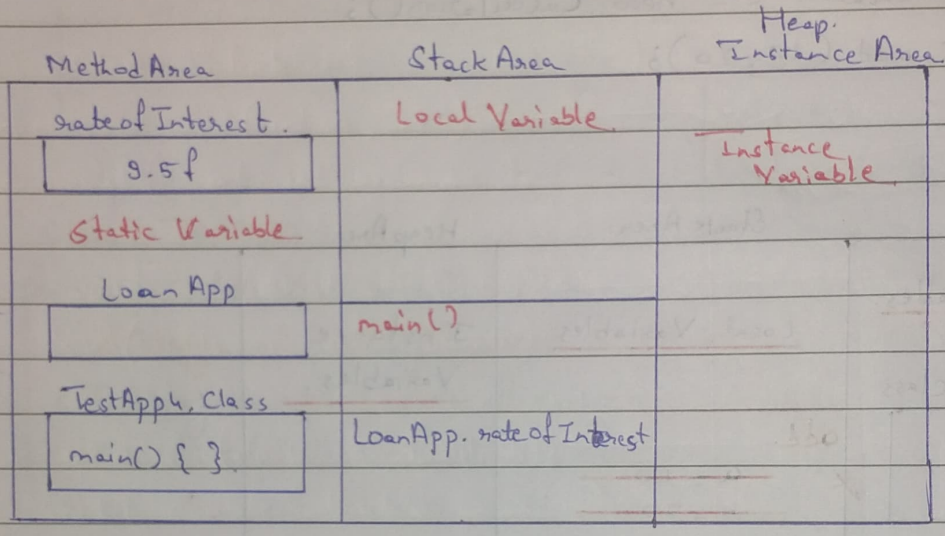
M T W T F S S
Page No.:
Date:
YOUVA

=> Code:-

```
Class LoanApp {
    // Static variable
    Static float rateofInterest = 9.5f;
}

public class TestApp4{
    public static void main(...){
        Sout(LoanApp.rateofInterest);
    }
}
```

Output:-
9.5.



=> Static Variable:-

① Memory will be given in the Method-Area.
② Memory will given at the time of loading .class file.
③ Default value will be given if user ~~can~~ can't specify any variable.
④ Memory will be taken out at the time of unloading ~~the~~ the .class file.
⑤ Static variables can be accessed in 2 ways.
   (a) Using Class Name.
   (b) using reference of the object.

Based on the type of value holded by variable we classify the variables into 2 types.

(a). primitive variables.

```
int a = 10; // primitive variable.
```

(b). reference variables

```
Student std = new student(); // reference variable.
```

Instance Variable

It is also ex. of

(b) Parameterized Constructor.

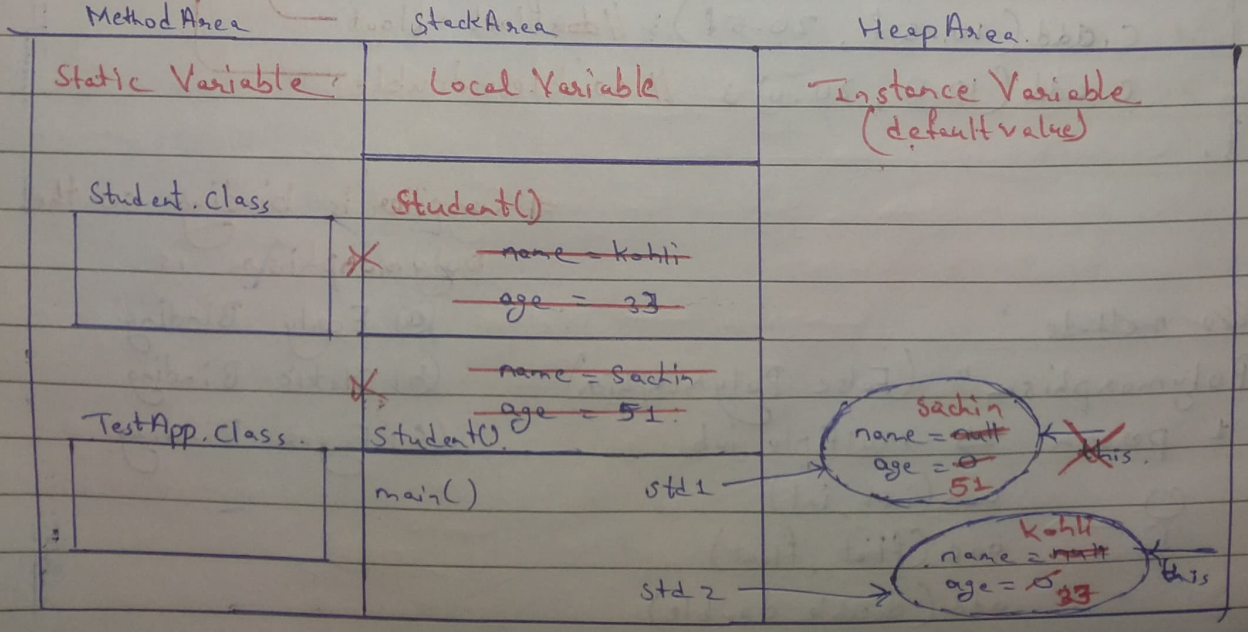
=> Code :-

```
Class Student3{
    // instance variable
    String name;
    int age;
    // constructor
    Student3(String name, int age){
        this.name = name;
        this.~~a~~age = age;
    }
    // Instance Method
    public void disp(){
        Sout("Name is : "+name);
        Sout("Age is : "+ age);
    }
}
public Class TestApp5 {
    ~~public~~psvm(){
    Student3 std1 = new student3("Sachin", 51);
    std1.disp();
    Student3 std2 = new student3("~~kohli~~Kohli", 33);
    std2.disp();
    }
}
```

Output :-

Name is : Sachin.
Age is : 51.

Name is : Kohli
Age is : 33.



Pillars of OOPs :-

(a) Data Hiding :- achived using private access modifier.
(b) Abstraction :- achived using abstract & interfaces.
(c) Encapsulation :- Data hiding + abstraction.
(d) Polymorphism :- achived in inheritance (Static, Dynamic).

Polymorphism → many forms.

- (a) Static polymorphism (eg:) → Method Overloading, Method Hiding.
- (b) Dynamic Polymorphism → Method Overriding.

Method Overloading.

=> Code :-

```
Class Calculator {
    public void add(int a, int b){
        Sout("int-int argument");
    }
    public void add(float a, float b){
        Sout("float-float argument");
    }
    public void add(double a, double b){
        Sout("double-double argument");
    }
}

public class Test {
    psvm(...){
        Calculator c = new Calculator();
        c.add(10, 20); // int-int
        c.add(10.5f, 20.5f); // float-float
        c.add(10.0, 20.0); // double-double
    }
}
```

public class Test {
psvm(...){
Calculator c = new Co

Compiler → int-int.
Compiler
Compiler

Compiler is binding so the polymorphism is
(a) Early Binding
(b) Static Binding
(c) Eager Binding.

"Polymorphism" (False Polymorphism)
1 person ⇒ Multiple job.

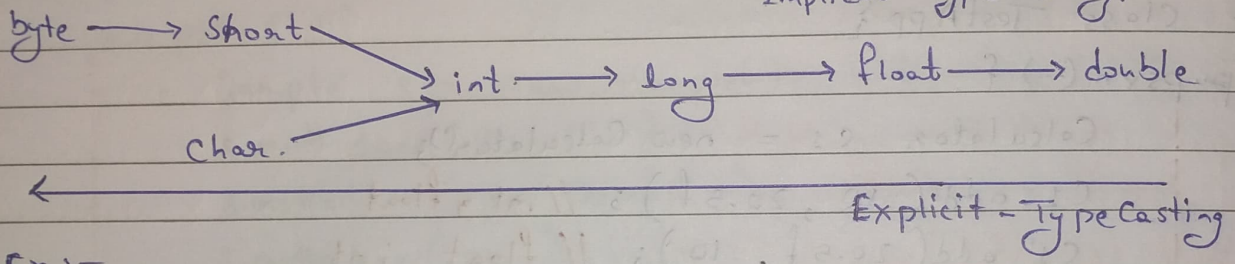
add(int, int)
add(float, float)
add(double, double)

Note :-
→ Two Methods is said to be overloaded, if both the methods have same name but different Argument types.
→ In Case of methodOverloading, Compiler will bind the call of the method to the body of the methods.
→ JVM should just execute the method body, so we say Method Overloading as ~~"Compile"~~ "Compile Time Binding / Early Binding".

M T W T F S S
Page No.:
Date:
YOUVA

Automatic type Promotion in Overloading.

~~Implicit~~ TypeCasting.



Ex :-

=> Code :-

```
Class Calculator {
    public void add(int a) {
        Sout("int argument");
    }
    public void add(float a) {
        Sout("float argument");
    }
}
public Class TestApp {
    ~~pub~~psvm(...) {
        Calculator c1 = new Calculator();
        c1.add('a'); // Char → char, int.
        c1.add(19L); // Long → Long, float, double.

        // Compile Time Error : No Suitable Method Found.
        c1.add(10.5); // double → double.
    }
}
```

Note :- Overloading.
Compiler ~~bui~~ binding the call based on arguments.
(a) If exact match is found bind the call.
(b) If exact match not found, perform implicit-Typecasting till it reaches to bind.
(c) Upon implicit-Typecasting still if the call can't be bind, it coould result in "CE".

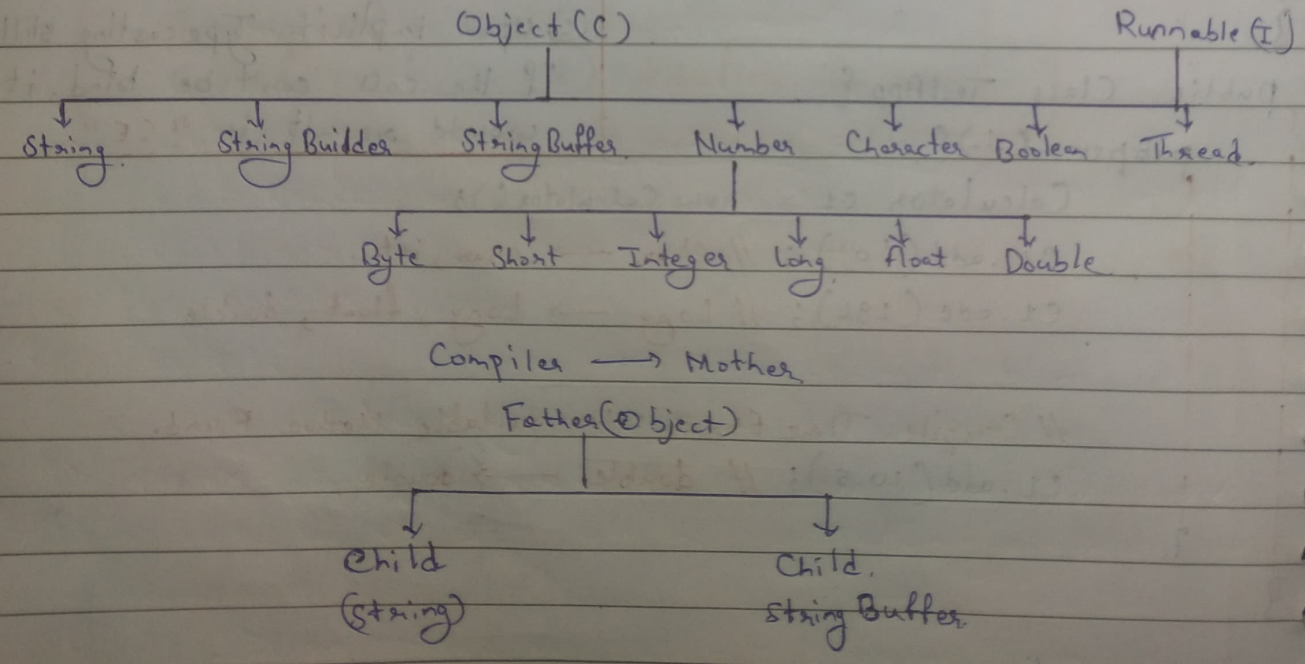
#Ambigous Method Call CompileTime Error.

=> Code :-

```
Class Calculator{ {
    public void add( int a , float b){
        sout(" int - float argument");
    }.
    public void add( int float a , int b){
        sout(" float - int argument ");
    }.
}.
pubic Class TestApp{
    pupsvm(...) {
        Calculator c1 = new Calculator();
        c1.add( 10 , 20.5f ); // int , float
        c1.add( 20.5f , 10 ); // float , int.
        // Ambigous Method Call Compile time Error.
        c1.add (10 , 20);
    }.
}.
```

int , int ⟶ exact match ✗

int , float
float , int
float , float.
} ⟶ Type Promotion.



Compiler ⟶ Mother

Father(Object)

Child (String)

Child. StringBuffer

Ex.①

=> Code :-

```
Class Sample {
    public void methodone(String s){
        Sout("String Version...");
    }
    public void methodone(Object o){
        Sout("Object Version...");
    }
}
public class Test {
    psvm(){
        Sample s = new Sample();
        s.methodone("sachin"); // String ⟶ String
        s.methodone(new Object()); // Object ⟶ object
        s.methodone(null); null ⟶ String (Reference), Object (Reference)
    }
}
```

Output :-

String version...
Object version...
String version...

Here String is Child class & object is (Father) parent class. Compiler is Mother so, She has always have soft corner for child.

Ex(2) => Code :-

```
Class Sample{
    public void methodone(String s){
        Sout("String version...");
    }
    public void methodone(StringBuffer o){
        Sout("StringBuffer...");
    }
    public void methodone(Object o){
        Sout("Object version...");
    }
}
public class Test {
    psvm(){
        Sample s = new Sample();
        s.methodone(new String("sachin"));
        s.methodone(new StringBuffer("sachin"));
        // Compile Time Error. Ambigous Call.
        s.methodone(null); // String (Reference), StringBuffer (Reference)
    }
}
```

Both String & StringBuffer are Child Class Compiler is Not able to Select one of them.

In case of methodoverloading, compiler will bind the method call based on the reference type but not on the runtime object.

=> Code :-

```
Class Animal{
}
Class Monkey extends Animal{
}
Class AnimalApp {
    public void m1(Monkey m){
        Sout("Monkey version...");
    }
    public void m1(Animal a){
        Sout("Animal version...");
    }
}
public class Test{
    psvm(){
        AnimalApp a = new AnimalApp();
        Monkey m = new Monkey();
        a.m1(m); // m(Monkey) ⟶ Monkey
        Animal animal = new Animal();
        a.m1(animal); // animal ⟶ Animal
        // Parent obj = new Child(); valid.
        Animal an = new Monkey();
        a.m1(an); // an(Animal) ⟶ Animal
    }
}
```

Runtime Object

Reference Object.

Output :-

Monkey Version...
Animal Version...
Animal Version...

Var-args in Java

=> This Mechanism is available in Java from JDK 1.5v.

=> In case of var-args all the arguments should be of same data type.

=> U can call var-args by passing arguments from 0...n.

Ex :-

=> Code :-

```
Class Calculator {
    // Method Overloading : Same argument type, But different argument count.
    public void add(int a, int b){
        sout(a+b);
    }
    public void add(int a, int b, int c){
        sout(a + b + c);
    }
    public void add(int a, int b, int c, int d){
        sout(a + b + c + d);
    }
    public void add(int a, int b, int c, int d, int e){
        sout(a + b + c + d + e);
    }
}
```

```
public Class Test {
    psvm() {
        Calculator c = new Calculator();
        c.add(10, 20);
        c.add(10, 20, 30);
        c.add(10, 20, 30, 40);
        c.add(10, 20, 30, 40, 50);
        sout();

        AdvancedCalculator ac = new AdvancedCalculator();
        ac.add();
        ac.add(10);
        ac.add(10, 20);
        ac.add(10, 20, 30);
        ac.add(10, 20, 30, 40);
        ac.add(10, 20, 30, 40, 50);
    }
}
```

Output :-

```
30
60
100
150

0
10
30
60
100
150
```

```
Class AdvancedCalculator {
    // Var-Args : 0 to n
    public void add(int... args){
        int sum = 0;
        for(int ele : args){
            sum += ele;
        }
        sout(sum);
    }
}
```

Var-Arg Vs Overloaded Method.

M T W T F S S | Page No.: | Date: | YOUVA

=> Code :-

```
Class Demo{
    // Exact Match : One-Argument
    public void methodone(int i){
        Sout("General Method");
    }
    // Var-Arg : 0 to n-
    public void methodone(int ... i){
        Sout("Var-Arg Method");
    }
}
public Class Test{
    public s v m(){
        Demo d1 = new Demo();
        d1.methodone(); // Var-Arg
        d1.methodone(10); // Exact Match
        d1.methodone(10, 20); // Var-Arg Method
    }
}
```

Output :-
Var-Arg Method
General Method
Var-Arg Method.

Overriding

ex①

=> Code :-

```
public class Test{
    public s v m(){
        // Parent Object
        Parent p1 = new Parent();
        p1.property();
        p1.marry();
        Sout();
        // Child Object
        Child c1 = new Child();
        c1.property();
        c1.marry();
        Sout();
        // Child Object.
        Parent p2 = new Child();
        p2.property();
        p2.marry();
    }
}
```

```
Class Parent{
    pv property(){
        Sout("Land + Cash + Gold");
    }
    pv marry(){
        Sout("Relative Girl");
    }
}
```

```
Class Child extends Parent{
    // Overriding
    p v marry(){
        // Changing the implementation
        Sout("Some Other Girl...");
    }
}
```

=> Output :-
- Land + Cash + Gold
- Relative Girl
- Land + Cash + Gold
- Some Other Girl.

- Land + Cash + Gold
- Some Other Girl.

⇒ Note :- # Overloading.

→ Two or more methods with same name, but different argument type is refered as "Overloading".

→ In Case of Overloading, Compiler will bind the methodcall based on the argument type we are passing, so, we say Overloading has "FalsePolymorphism / EagerBinding / StaticBinding / EarlyBinding"

⇒

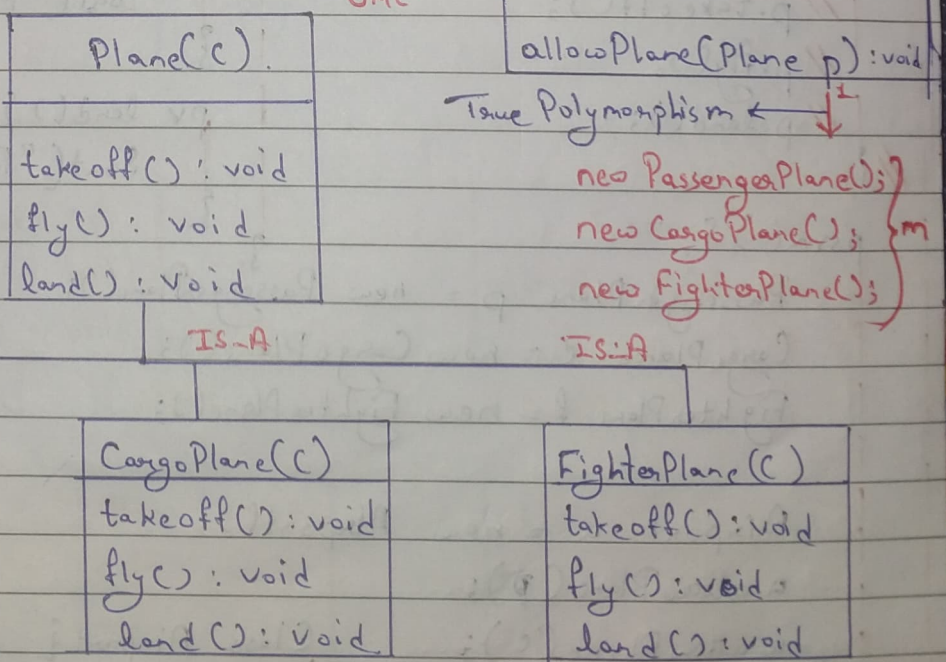
→ Note :- # Overriding.

→ During inheritance, the parent class method implementation would not match the needs of the child class so, Child Class will take the method name, but it will change the implementation as per the needs of the Child Class. This mechanism is called as "Overriding".

→ In Case of Overriding, JVM will bind the method calls based on the runtime object, but not on the reference type so, we say Overriding has "TruePolymorphism / LateBinding / RuntimeBinding".

U → Uniformed
M → Modeling
L → Language

"UML"



ex (2) :- => Code :-

```
Public Class Test {
  //psvm(){
    //Plane p = null;
    //p = new PassengerPlane();
    //p.takeoff();
    //p.fly();
    //p.land();

    //Sout();

    //p = new CargoPlane();
    //p.takeoff();
    //p.fly();
    //p.land();

    //Sout();

    //p = new FighterPlane();
    //p.takeoff();
    //p.fly();
    //p.land();
  //}

  psvm(){
    PassengerPlane p = new PassengerPlane();
    CargoPlane c = new CargoPlane();
    FighterPlane f = new FighterPlane();

    Airport a = new Airport();
    a.allowPlane(p);
    a.allowPlane(c);
    a.allowPlane(f);
  }
}

Class Plane{
  pv takeoff(){
    Sout("Plane Tookoff...");
  }
  pv fly(){
    Sout("Plane is Flying...");
  }
  pv land(){
    Sout("Plane is Landing...");
  }
}

Class PassengerPlane extends Plane{
  pv takeoff(){
    Sout("PassengerPlane Tookoff...");
  }
  pv fly(){
    Sout("PassengerPlane is Flying...");
  }
  pv land(){
    Sout("PassengerPlane is Landing...");
  }
}

Class Airport{
  pv allowPlane(Plane p){
    p.takeoff();
    p.fly();
    p.land();
    Sout();
  }
}
```

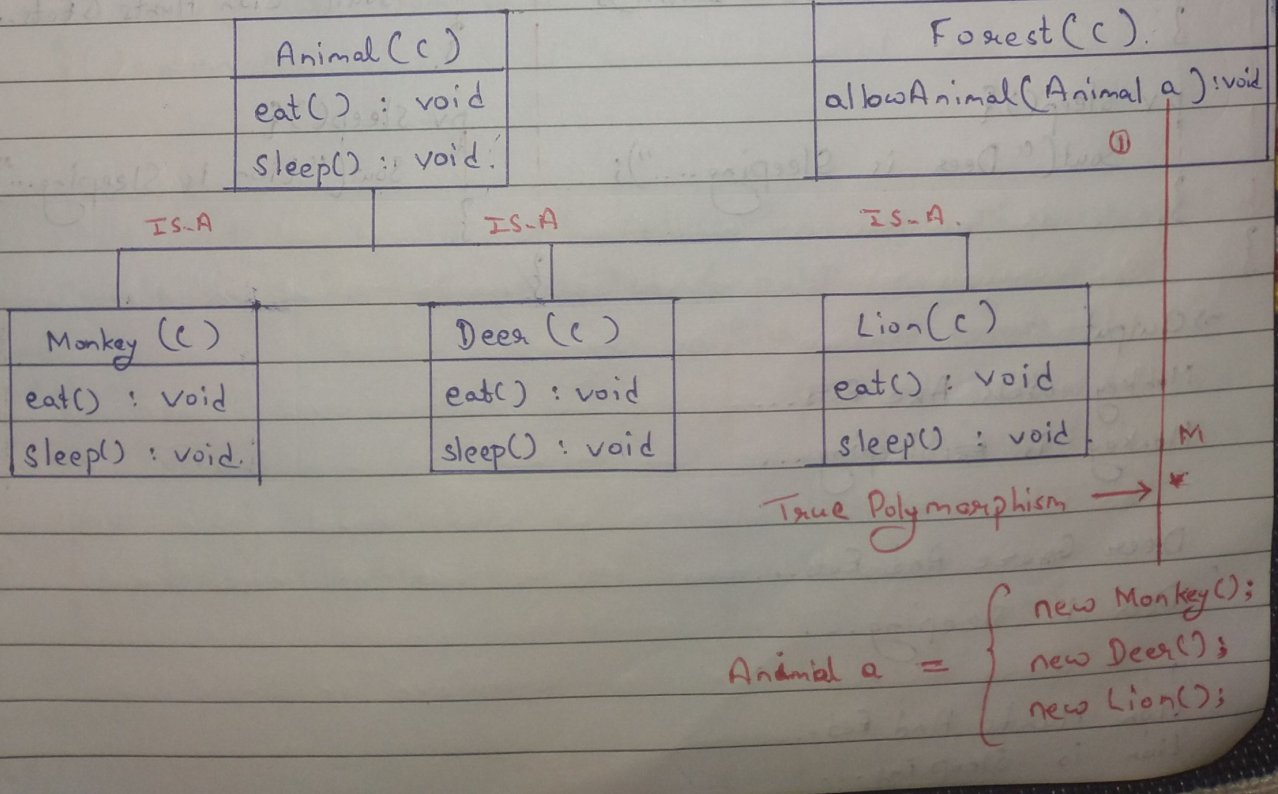
```
Class CargoPlane extends Plane{
    pv takeoff(){
        Sout("CargoPlane Took off...");
    }
    pv fly(){
        Sout("CargoPlane is Flying...");
    }
    pv land(){
        Sout("CargoPlane is Landing...");
    }
}
Class FighterPlane extends Plane{
    pv takeoff(){
        Sout("FighterPlane is Tookoff...");
    }
    pv fly(){
        Sout("FighterPlane is Flying....");
    }
    pv land(){
        Sout("FighterPlane is Landing...");
    }
}
```

Output:-

PassengerPlane tookOff....
PassengerPlane flying....
PlassengerPlane landing....

CargoPlane tookoff....
CargoPlane flying....
CargoPlane landing....

FighterPlane tookoff....
FighterPlane flying....
FighterPlane landing....



ex ③. => Code :-

```
Public Class Test {
    psv m() {
        Forest f = new Forest();
        f.allowAnimal(new Monkey());
        f.allowAnimal(new Deer());
        f.allowAnimal(new Lion());
    }
}
```

```
Class Forest {
    pv allowAnimal(Animal a){
        a.eat();
        a.sleep();
        Sout();
    }
}
```

```
Class Animal() {
    pv eat() {
        Sout("Animal is Eating...");
    }
    pv sleep() {
        Sout("Animal is Sleeping...");
    }
}
```

```
Class Monkey extends Animal {
    pv eat() {
        Sout("Monkey Steals & Eats...");
    }
    pv sleep() {
        Sout("Monkey is Sleeping...");
    }
}
```

```
Class Deer extends Animal {
    pv eat() {
        Sout("Deer Graze & Eats...");
    }
    pv sleep() {
        Sout("Deer is Sleeping...");
    }
}
```

```
Class Lion extends Animal {
    pv eat() {
        Sout("Lion Hunts & Eats...");
    }
    pv sleep() {
        Sout("Lion is Sleeping...");
    }
}
```

=> Output

Monkey Steals And Eats....
Monkey is Sleeping...

Deer Graze And Eats...
Deer is Sleeping...

Lion Hunts And Eats...
Lion is Sleeping...

#Rules Of Overriding.

M T W T F S S
Page No.:
Date:
YOUVA

Rule.
① In case of Overriding, we can't change the return type of the method, if we want to change then there should be ~~no~~ relationship b/w returntypes of the method.

⇒ Ex ①.

⇒ Code :-

```
Class Parent{
    public Object methodOne(){
        return null;
    }
}
```

```
Class Child extends Parent{
    public void methodOne(){
        Sout("Hello From Child...");
    }
}
```

```
Public Class Test {
    ps v m( ) {
        Parent p = new Child();
        p.methodOne( );
    }
}
```

Ex ① Output :-
CE : Void & Object are not related.

⇒ Ex ②

```
Class Child extends Parent{
    public String methodOne(){
        Sout("Hello From Child...");
        return null;
    }
}
```

Ex ② Output :-
Hello from Child...

Rule.
② While Overriding, we can't reduce the scope of access modifier.
private < default < protected < public

⇒ Ex ①

⇒ Code :-

```
Class Parent {
    public void methodOne(){
        Sout("Hello from Parent Class");
    }
}
```

⇒ Output :-
CE : Can't reduce the scop of access modifier.

```
Class Child extends Parent{
    protected void methodOne(){
        Sout("Hello from Child Class");
    }
}
```

```
Public Class Test {
    psvm(){
        Parent p = new Child();
        p.methodOne();
    }
}
```

Ex ②:-

```
Class Parent {
    ~~public~~ void methodOne(){        → Default access modifier.
        Sout("Hello From Parent Class...");
    }
}

Class Child extends Parent {
    //default / Protected / Public you can use this.
    protected void methodOne(){
        Sout("Hello From Child Class...");
    }
}

public Class Test {
    psvm(){
        Parent p = new Child();
        p.methodOne();
    }
}
```

⇒ Output → Hello From Child Class...

Rule ③ Private methods won't participate in inheritance, so Overriding them in Child Class is NOT Possible.

• Ex :-

⇒ Code :-

```
Class Parent{
    private void methodOne(){
        Sout("Hello From Parent Class");
    }
}

Class Child extends Parent{
    private void methodOne(){
        Sout("Hello From Child Class");
    }
}

public Class Test {
    public s v m ( ) {
        Parent p = new Child();
        p.methodOne();
    }
}
```

⇒ Output :-

CE: error: methodOne() has private access in Parent.

Rule ④ final is an access modifier applicable at.

ⓐ. variable ⇒ If applied at variable level, then the value can't be changed.

ⓑ. method ⇒ If applied at method level, then we can't override the method in child Class.

ⓒ. Class ⇒ If applied at Class level, then the Class coon't ~~parici~~ participate in inheritance.

final methods can't be overriden in Child Class.

ex ① :-

```
Class Parent {
    public final void methodOne(){
        Sout("ParentClass :: methodOne()");
    }
}

Class Child extends Parent{
    public void methodOne(){
        Sout("ChildClass :: methodOne()");
    }
}

Public Class Test {
    psvm(){
        Parent p = new Child();
        p.methodOne();
    }
}
```

Output :-

error: methodOne() in Child cannot override ~~methodOne() in Parent~~

Rule

⑤ abstract is an access modifer applicable at.

ⓐ method ⇒ If we are not giving the body for a method then mark the method as "abstract".

ⓑ Class ⇒ If we don't want the Object to be created for a class, then mark the class as "abstract".

ⓒ variable ⇒ This access modifier can't be applied on variables.

In case of overriding, compulsorily the child class should give implementation for all the abstract method ~~Parent~~ present in the parent class, if the implementation is not given then that child class should be marked as "abstract".

Ex ① :-

⇒ Code :-

```
abstract Class Parent{
    public ~~void~~ abstract void methodOne();
}

Class Child extends Parent{
    public void methodOne(){
        Sout("ChildClass :: methodOne()");
    }
}
```

```
public Class Test{
    psvm(){
        Parent p = new Child();
        p.methodOne();
    }
}
```

⇒ Output :-

ChildClass :: methodOne().

→ In Java abstraction can be achieved using
(a) abstract Class
(b) Interface.

M T W T F S S
Page No.:
Date:
YOUVA

→ In real world, for few cases object should not be created, to handle such scenarios in Java we need go for a keyword called "abstract".

→ abstract access modifier can be applied at.
(a) Class level :- Yes possible, if we make class as abstract then object can't be instantiated.
(b) method level :- yes possible, if we make method as abstract, then we can't give body for the methods.
(c) Variable level :- no we can't use abstract at variable level.

Rules of abstract access-modifier in Java.

(1) If a class contains atleast on abstract method, then mark the class as "abstract".

(2) abstract class can't be instantiated.

(3) for an abstract class, we can create a reference, but not the object.

(4) Inside abstract class, we can write concrete methods also.

(5) If a parent class is abstract, then compulsorily the child class should give implementation for all the abstract methods otherwise the child class also would become "abstract".

(6) Even if the class doesn't contain abstract methods/concrete method still we can mark the empty class as "abstract"

Ex:- //Exposing the set of services but hiding the internal implementation.

=> Code :-

```
abstract class Plane{
    public abstract void takeoff();
    public abstract void fly();
    public abstract void land();
}

Class PassengerPlane extends Plane{
    pv takeoff(){
        Sout("PassengerPlane took off...");
    }
    pv fly(){
        Sout("PassengerPlane flying...");
    }
    pv land(){
        Sout("PassengerPlane landing...");
    }
}

Class FighterPlane extends Plane{
    pv takeoff(){
        Sout("FighterPlane took off...");
    }
    pv fly(){
        Sout("FighterPlane flying...");
    }
    pv land(){
        Sout("FighterPlane landing...");
    }
}

Class CargoPlane extends Plane{
    pv takeoff(){
        Sout("CargoPlane tookoff...");
    }
    pv fly(){
        Sout("CargoPlane flying...");
    }
    pv land(){
        Sout("CargoPlane landing...");
    }
}

Class Airport{
    pv allowPlane(Plane ref){
        ref.takeoff();
        ref.fly();
        ref.land();
        Sout();
    }
}

public Class Test{
    psvm(..){
        Airport a = new Airport();
        a.allowPlane(new PassengerPlane());
        a.allowPlane(new FighterPlane());
        a.allowPlane(new CargoPlane());
    }
}
```

Output :-

```
PassengerPlane took off...
"  "  flying...
"  "  landing...
FighterPlane took off...
"  "  flying...
"  "  landing...
CargoPlane tookoff...
"  "  flying...
"  "  landing...
```

→ Q1. Can we create an Object for abstract Class?
Ans :- No.

→ Q2. When will the Constructor gets Called?
Ans :- During the ~~Constructor~~ Creation of an object.

→ Q3. Does abstract class has Constructor?
Ans :- Yes.

** Q4. Why we need Constructor in abstract Class, when we can't instantiate an object?
Ans :- To get the properties of parent Class to Child Class, we need Constructors in abstract Class also.

Ex :-

```
abstract Class Person {
    String name;
    int age;
    float height;
    Person(String name, int age, float height){
        this.name = name;
        this.age = age;
        this.height = height;
    }
}
Class Student extends Person {
    int marks;
    float avg;
    Student(String name, int age, float height, int marks, float avg){
        super(name, age, height);
        this.marks = marks;
        this.avg = avg;
    }
    public void display(){
        Sout("Name   is : " + name);
        Sout("Age    is : " + age);
        Sout("Height is : " + height);
        Sout("Marks  is : " + marks);
        Sout("Avg    is : " + avg);
    }
}
```

```
Public Class Test {
    psvm(){
        Student s = new Student("Sachin", 51
            , 5.3f, 100, 53.3f);
        s.display();
    }
}
```

Output :-

```
Name   is : Sachin
Age    is : 51
Height is : 5.3
Marks  is : 100
Avg    is : 53.3
```

* Super is a method call which is used to call Parameterised Constructor of parent from Child class.

Ex:-

=> Code :-

```
abstract class Bird{
    public abstract void fly();
    public abstract void eat();
}

abstract class Eagle extends Bird{
    pv fly(){
        Sout("Eagle fly @very high");
    }
    public abstract void eat();
}

Class SerpentEagle extends Eagle{
    public void eat(){
        Sout("serpent eagle eats snake..");
    }
}

Class GoldenEagle extends Eagle{
    public void eat(){
        Sout("Golden eagle catches prey over the Ocean...");
    }
}

Class Crow extends Bird{
    p v fly(){
        Sout("crow fly @medium height..");
    }
    pv eat(){
        Sout("Crow eat grain...");
    }
}

Class Sparrow extends Bird{
    pv fly(){
        Sout("Sparrow fly @shot height");
    }
    pv eat(){
        Sout("Sparrow eat grains...");
    }
}

Class Sky{
    pv allowBird(Bird ref){
        ref.fly();
        ref.eat();
        Sout();
    }
}

Public Class Test{
    ps v m(...){
        Sky sky = new Sky();
        sky.allowBird(new Sparrow());
        sky.allowBird(new SerpentEagle());
        sky.allowBird(new GoldenEagle());
        sky.allowBird(new Crow());
    }
}
```

Output:-

```
Sparrow fly @shot height
Sparrow eat grains...

Eagle fly @very high
Sepspent eagle eats snakes...

Eagle fly @very high
Golden eagle catches prey over
    the Ocean...
Crow fly @medium height
Crow eat grains...
```

Note:-

→ abstract Class can contain Concrete & abstract methods.
→ Concrete methods :- A method with implementation
→ abstract methods :- A method without implementation.

M T W T F S S
Page No.:
Date:
YOUVA

① abstract Class contains concrete methods & abstract method, so we say through abstracts Class 100% abstraction can't be achieved.

② To achieve 100% abstraction, we need to go for "interfaces".

Interface in Java

① In java interfaces has been introduced to provide "SRS"
SRS ⇒ Software Requirement specification.

② In java interfaces could act like a contract b/w the Client & the ~~server~~ service provider.
In java interfaces also represents the mechanism to put rules from the client end to the server provider.
eg:- (i) JDBC API (rules given by SUNMS) for Database vendors like MySQL, Oracle, Sybase....
(ii) " servlet API (rules given by SUNMS) for server vendors like Tomcat, Glassfish, JBoss, ...

③ In java interfaces always represents 100% abstraction.
→ Inside java interface we can write only abstract methods, we can't write concrete methods.

Note:- Inside interface the methods we write always indicates "public & abstract".

Ex ①.

⇒ Code :-

```
interface ICalculator {
    // public abstract
    void add(int a, int b);
    void sub(int a, int b);
    void mul(int a, int b);
    void div(int a, int b);
}
```

=> interfaces should be implemented by a class.

=> Which ever class is implementing an interface that class should compulsorily give the body of all the abstract methods present in Interface.

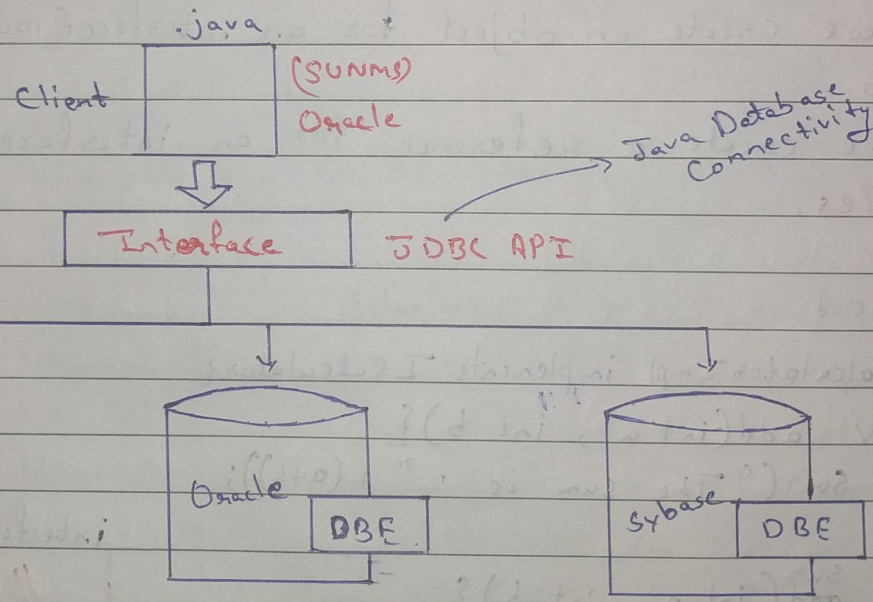
=> If the implementation class fails to give the body for at least one abstract methods then the class would become abstract Class.

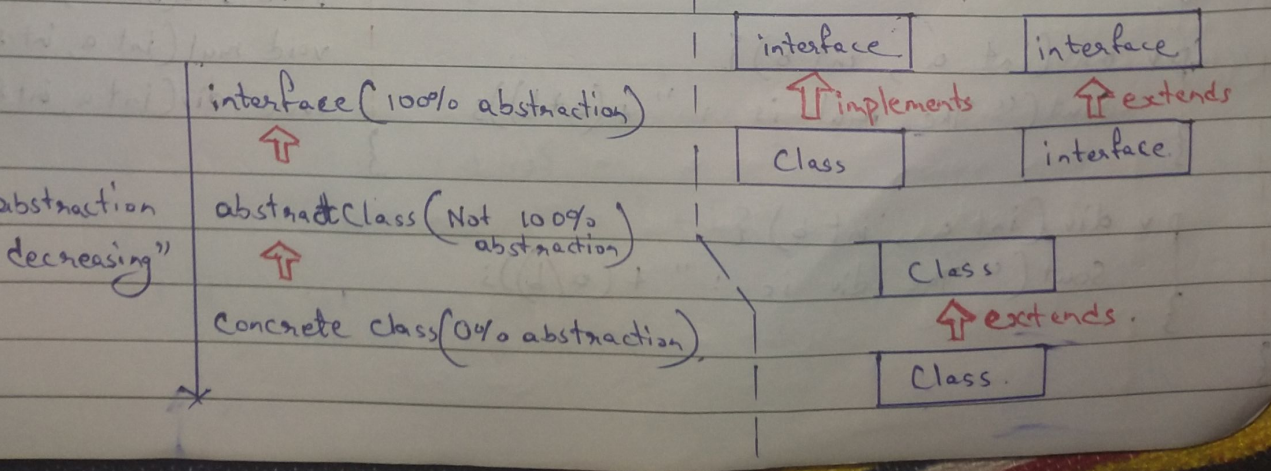
Ex(2).

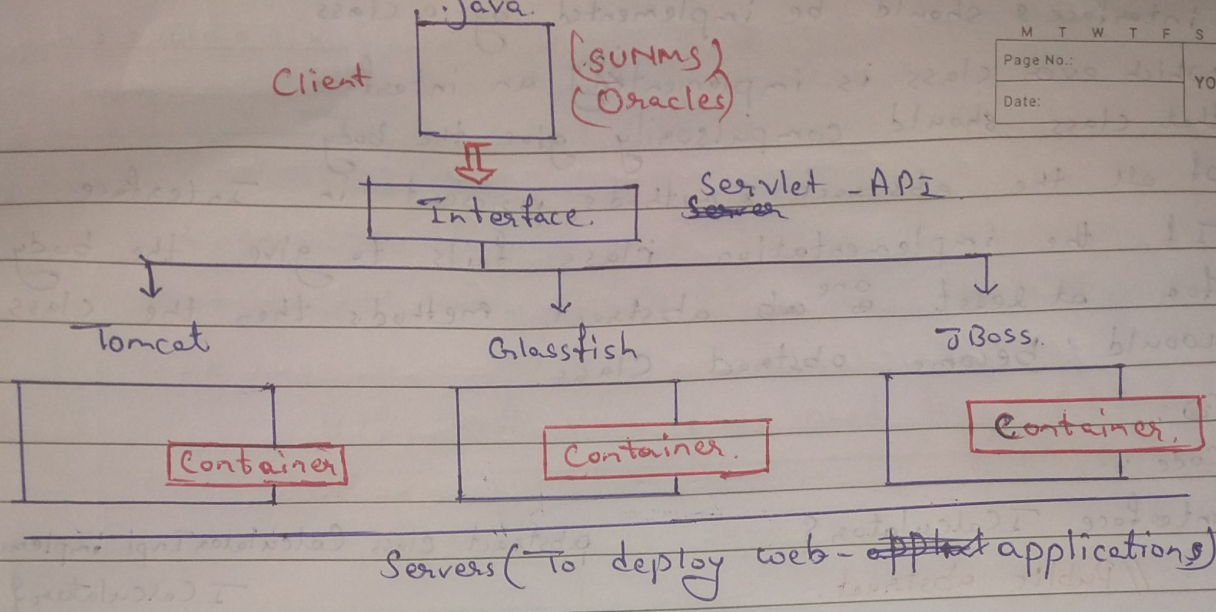
=> Code :-

```
interface ICalculator {
    // Public abstract.
    void add(int a, int b);
    void sub(int a, int b);
    void mul(int a, int b);
    void div(int a, int b);
}
```

```
abstract class CalculatorImpl implements
    ICalculator {
}
```







Q. Can we create an object for abstract class?

Ans:- No

Q. Can we create a reference for an abstract class?

Ans:- Yes.

Q. Can we create an object for an interface (100% abstraction)?

Ans:- No

Q. Can we create a reference for an interface?

Ans:- Yes.

Ex (3). :- Code :-

```
Class CalculatorImpl implements ICalculator{
    pv add(int a, int b){
        Sout("The sum is :" +(a+b));
    }
    pv ~~add~~ sub(int a, int b){
        Sout("The diff is :" +(a-b));
    }
    pv mul(int a, int b){
        Sout("The mul is :" +(a*b));
    }
    pv div(int a, int b){
        Sout("The div is :" +(a/b));
    }
}
```

```
interface ICalculator{
    // Public abstract
    void add(int a, int b);
    void sub(int a, int b);
    void mul(int a, int b);
    void div(int a, int b);
}
```

```
public Class Test{
    psvm{
        // loose Coupling : Polymorphism
        ICalculator calculator = new CalculatorImpl();
        Calculator.add(10, 20);
        calculator.sub(100, 20);
        calculator.mul(10, 20);
        calculator.div(100, 20);
    }
}
```

⇒ Output :-

The sum is : 30.
The diff is : 80.
The mul is : 200
The div is : 5.

extends V/s implements

(a). A class can extends only from one class at a time

Ex(1). Code :-

```
Class One{
    public void methodOne(){
    }
}
```

```
Class Two{
    public void methodOne(){
    }
}
```

(b) A class can implements any no. of interfaces at a time.

Ex(1). Code :-

```
interface ICalculator{
    void add(int a, int b);
    void sub(int a, int b);
}
```

```
interface IAdvanceCalculator{
    void mul(int a, int b);
    void div(int a, int b);
}
```

```
Class CalculatorImpl implements ICalculator, IAvanceCalculator{
    pv add(int a, int b){
        sout(" The sum is : " + (a+b));
    }
    pv sub(int a, int b){
        sout(" The diff is : " + (a-b));
    }
    pv mul(int a, int b){
        sout(" The mul is : " + (a*b));
    }
    pv div(int a, int b){
        sout(" The div is : " + (a/b));
    }
}
```

```
Public void Test {.
    psvm{.
        ICalculator c1 = new CalculatorImp();
        c1.add(10, 20);
        c1.sub(100, 20);
        IAdvanceCalculator c2 = new CalculatorImp();
        c2.mul(10, 20);
        c2.div(100, 20);
    }.
}.
```

Output → Same as previous. Ex ③.

M	T	W	T	F	S	S

Page No.: YOUVA

Date:

* A Class can extends a class & can implement any no. of interfaces simultaneously.

Ex ① :- Code :-

```
interface ICalculator {
    void add(int a, int b);
    void sub(int a, int b);
}.

interface IAdvanceCalculator {
    void mul(int a, int b);
    void div(int a, int b);
}.

Class CalculatorAdvaced {
    pv mul(int a, int b){
        Sout("The mul is : " + (a*b));
    }.
    pv div(int a, int b){
        Sout("The div is : " + (a/b));
    }.
}.

public Class Test {
    ps vm{
        CalculatorImpl c1 = new
                CalculatorImp();
        c1.add(10, 20);
        c1.sub(100, 20);
        c1.mul(10, 20);
        c1.div(100, 20);
    }.
}

Output :- → Same.

//inheritance
//Implementation
Class CalculatorImpl extends CalculatorAdvanced implements ICalculator {
    public void add(int a, int b){
        Sout("The sum is : " + (a+b));
    }
    public void sub(int a, int b){
        Sout("The diff is : " + (a-b));
    }
}.
```

An interface can extends any no. of Interface

Page No.: VOUVA
Date:

Ex①. :- Code :-

```
interface ICalculator {
    void add(int a, int b);
    void sub(int a, int b);
}
```

```
interface IAdvanceCalculator {
    void mul(int a, int b);
    void div(int a, int b);
}
```

```
interface IAdvanceCalculator extends ICalculator {
    void mul(int a, int b);
    void div(int a, int b);
}
```

```
Class CalculatorImpl implements IAdvanceCalculator {
    pv add(int a, int b){
        Sout("The sum is:" + (a+b));
    }
    pv sub(int a, int b){
        Sout("The sub is:" + (a-b));
    }
    pv mul(int a, int b){
        Sout("The mul is:" + (a*b));
    }
    pv div(int a, int b){
        Sout("The div is:" + (a/b));
    }
}
```

```
public Class Test {
    psvm(){
        IAdvanceCalculator c1 = new CalculatorImpl();
        c1.add(10, 20);
        c1.sub(100, 20);
        c1.mul(10, 20);
        c1.div(100, 20);
    }
}
```

=> Can we write a variable inside a interface?

Ans → Yes

Ex ① => Code :-

```
interface IRemote{
    // public static final
    int MIN_VOLUME = 0;
    int MAX_VOLUME = 100;
}
```

=> Can we write a constructor inside an interface?

Ans → No

=> Can we write static block inside an interface?

Ans → No

=> Can we write instance block inside an interface?

Ans → No.

Ex ① :- Code :-

```
interface IDemo{
    // public static final
    int x = 10;
}

public class Test implements IDemo{
    psvm(){
        int x = 100; Local Variable
        Sout(x);
        Sout(IDemo.x);
        Sout(Test.x);
    }
}
```

Output :-

100

10

10

Note :-

Inside an interface we can write

(a) Variable (b) methods (public abstract)

We can also write an interface without variable & methods, this type of interface is called "Marker Interface".

ex :- interface IDemo{

}

=> Benift of marker interface is the implementation class would get some extra advantage like.

M T W T F S S
Page No.:
Date:
YOUVA

(1) If a ~~code~~ Class implements Serializable then the object can be sent over the network.

(2) If a class implements Cloneable then object can be ~~sent over the network.~~ Cloned (duplicated).

=> Inbuilt Marker Interfaces :-

(1) public interface java.lang.Cloneable{
}

(2) public interface java.io.serializable{
}