OOPs

→ object oriented programming system/structures
→ also called OOP

→ OOP is a programming ~~paradigm~~ paradigm/ methadology.

Paradigm: Paradigm can also termed as method to solve some problem or do some task,

Programming Paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach.

Paradigm
- ↳ object oriented paradigm
- ↳ procedural paradigm
- ↳ functional paradigm
- ↳ logical paradigm
- ↳ structural paradigm

→ 6 main pillars of OOPs
- → class
- → objects and methods
- → inheritance
- → polymorphism
- → Abstraction
- → Encapsulation

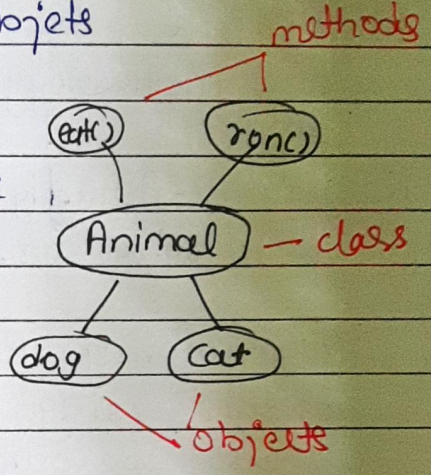
Purely object oriented programming language
→ small talk [first]

Class:

→ class is the collection of objects

→ class is not a real world entity. It is just a tempelate or blueprint.

→ class does not occupy memory, objects occupy memory.



Syntax:

→ if not access modifier then it default by default.

```
access_modifier ClassName
{
    - methods, - constructor, - fiels and variables
}
```

Methods:

→ A set of code which perform a particular task

Pros:
1) Code reusability
2) Code optimization

Syntax: access-modifier return-type methoName (parameters)
{

default access-modifier → default
}

Objects:

→ object is an instance of class
→ object is real world entity.
→ object occupy memory.

Object = dog
Identity = name of dog
State: Breed, Color

→ object consist of

1) Identity: name {always uniques}
2) state/attribute: represent field
3) behaviour: represent methods

behaviour: eat, run

HOW TO CREATE AN OBJECT?

→ new Keyword
→ new Instance() method
→ clone() method
→ deserilization()
→ factory methods

New Keyword Object Creation:

→ Declaration
→ Instantiation
→ Initialization

Declaration: It is declaring a variable name with an object type.

Ex: Animal buzo; (Animal is class name and buzo is reference-variable-name)
dog name

Working: Animal buzo;
└ null Memory
(null is assigned to memory)

Note:

1) All the objects (instances) share the attributes and the behaviour of the class.

2) Simply declaring a reference variable does not create an ~~class~~ object.

Instantiation:

→ This is when memory is allocated for an object

→ The (new) keyword is used to create the object.

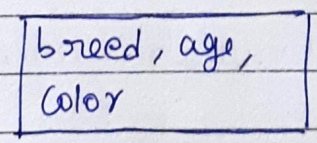
A reference to the object that was created is returned from the new keyword.

Initialization:

→ The new keyword is followed by a call to a constructor. This call initializes the new object.

Ex: buzo = new Animal();

Working: In initialization, the values are put into the memory that was allocated.



Memory (all the states and behaviours of an object are loaded in the memory)

Dot operator: • calling the methods through object

```
Animal buzo = new Animal()
buzo.color
buzo.run()
```

Scenario:

Suppose you have created a file without main method

```
class Java{
}
```

It will compiled successfully, with throw run time error. because not found main method.

The object of which class are you creating only the methods of this class can be accessed. until inheritance.

```
class Animal{
    void run(){
        sout("running");
    }
    psvm(string[] args){
        Animal buzo = new Animal();
        buzo.run()
        buzo.fly()  // compile time exception
        Birds bp = new Birds();
        bp.fly() //
    }
}
class Birds{
    void flying(){
        sout("flying");
    }
}
```

HOW TO INITIALISE OBJECT ?

1) By reference variable

```
class Animal {
    String color;
    int age;

    psvm (String args[]) {
        Animal buzo = new Animal();
        buzo.color = "black";
        buzo.age = 10;
        sout (buzo.color + "  " + buzo.age);
    }
}
    // black 10
```

2) By Using Method

```
class Animal {
    String color;
    int age;
    void colAge (String s, int a) {
        color = s;
        age = a;
    }
    void display () {
        sout (color + "  " + age);
    }
    psvm (String args[]) {
        Animal buzo = new Animal();
        buzo.colAge ("black", 10);
        buzo.display   // black 10
    }
}
```

Constructors

→ Constructor is a block (similar to method) having same name as the class name.

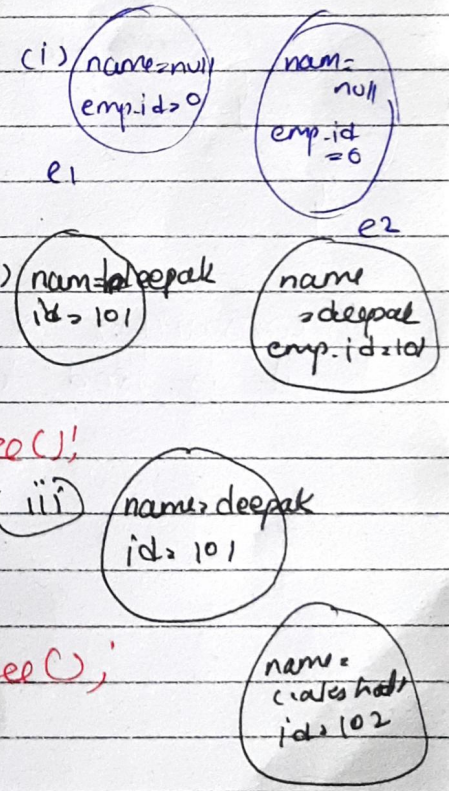
→ does not have any return type not even void.

→ The only modifier applicable for constructors are:
1) Public
2) protected
3) default
4) Private

→ It executes automatically when we create object.

IF NO CONSTRUCTOR

```
class Employee {
    String name //=deepak
    int emp-id  //= 101

    psvm (String [] args) {

        Employee e1= new Employee();
        e1.name= "deepak";
        emp-id = 101;

        Employee e2= new Employee();
        e2.name = "akshat";
        emp-id = 102;

    }
}
```



```
class Employee{

    String name;
    int emp_id;

    psvm(String[] args){}

    Employee(String name, int emp-id){
        this.name=name;
        this.emp-id= emp_id;
    }
    psvm(String[] args){

        Employee e1= new Employee("akshat", 101);
        or
        Employee e2= new Employee("tanisha", 102);
    }

}
```

name=akshat
emp-id=101

name="tanisha"
emp-id=102

constructor

// constructor is not used to create an object it is used to initialise an object.

1) Default constructor

-> no-arg constructor
-> created by compiler not JVM.

-> when you do not create a constructor then compiler itself create a constructor.
contains only super();

```
class Test {
  Test() {
    super();
  }
  psvm () {
    Test t = new Test()
  }
}
```

2) No-argument constructor

-> User defined

```
class Test {
    Test() {

    }
    psvm () {
        Test t=new Test()
    }
}
```

3) Parametrize constructor

```
class Test {
  Test(String name){

  }
  psvm () {
    Test t=new Test
    ("akshat");
  }
}
```

WHY CONSTRUCTOR NOT HAVE RETURN TYPE?

→ constructor main work is to initialise the object. there is no need to return type.

→ Suppose we haven't declare the constructor then compiler will create a constructor itself then compiler cannot judge what should be return type of constructor.

Inheritance

→ Inheriting the properties of parent class to child class.

→ Inheritance is the procedure by which one object acquires all the properties and behaviours of a parent object.

→ Inheritance takes place using extends keyword.

```
class Animal{
    void eat(){
        Sout("I am eating");
    }
}
class Dog extends Animal{
    psvm(){
        Dog d1 = new Dog();
        d1.eat();
    }
}
```

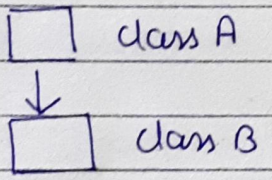
relation => IS-A
dog is a animal

→ Inheritance makes is-A relationship

→ code reusability
→ we can achieve polymorphism using inheritance
↳ method overriding

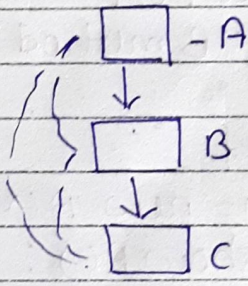
Types

1) Single Inheritance



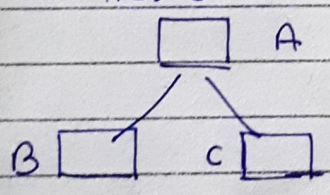
(V)

2) Multilevel Inheritance



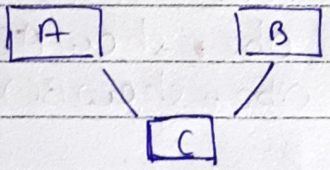
(V)

3) Hierarchical



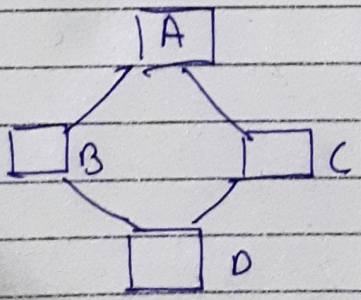
(V)

4) Multiple



(X)

5) combination of any two inheritances

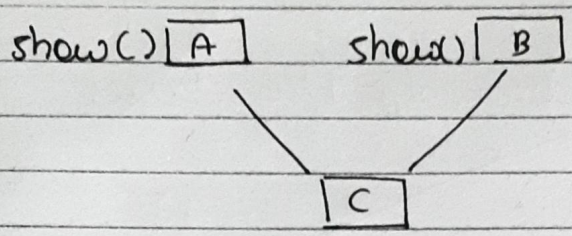


(X)

(i) Single Level

```
class A
{
  void showA(){
      sout("A method");
  }
}
class B extends A{
    void showB(){
       sout("B method");
    }
    psvm(){
       A ob1 = new A();
       ob1.showA();
       ob1.showB();   (X)

       B ob2 = new B();
       ob2.showA();
       ob2.showB();
    }
}
```

(4) Multiple




```
class C extends A,B {
    psvm() {
        C ob = new C()
        ob.show()        // ambiguity error
    }
}
```

multiple inheritance is not possible in java as you can see compiler will get confused whose class show() method is being calling here is ambiguity error.

Same as in hybrid.

* constructor never inherit in inheritance.
* private method also not get inherited.

↳ ∵ constructor is not a part of class, but the constructor of the superclass can be invoked from the subclass.

Cons:

Using inheritance the two classes are tightly coupled

There can be only one super class, not more than one because it will led to multiple inheritance.

→ Every class has a parent class except object class.

Types of Relationship b/w classes

(IS-A)
Inheritance

- → extends
- → blood relation
- → strong coupling

(HAS-A)
Association

- → ref variable, new
- → non-blood relation
- → not Hightly coupled

Association can be one to one, one to many, many-to-one, many-to-many.

Aggregation **composition**

Advantages:

1) Code reusability
2) cost cutting
3) Reduce redundancy

Inheritance

```
class Vehicle
{

}
class Car extends Vehicle
{

}
```

⇒ car is-A vehicle

Association (HAS-A):

```
class Student {
    string name;
    int roll-no;
}
```

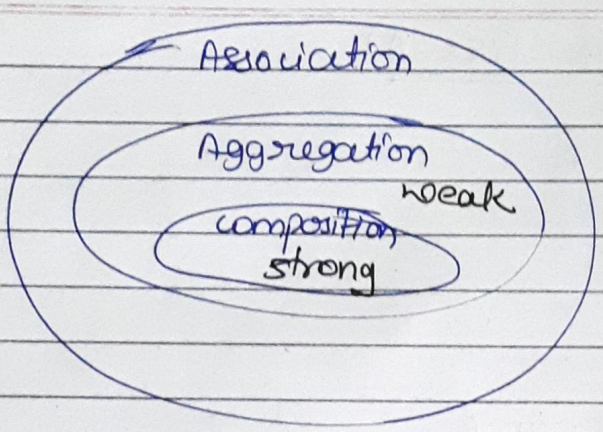
Student HAS-A name
Student Has-A name

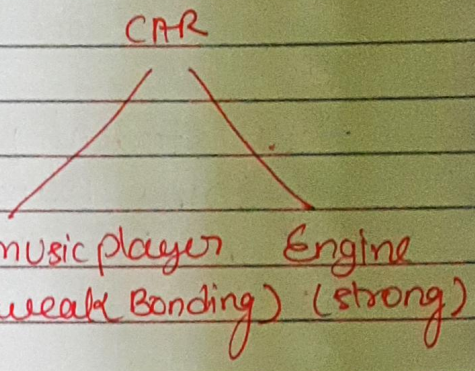
Association by new Keyword

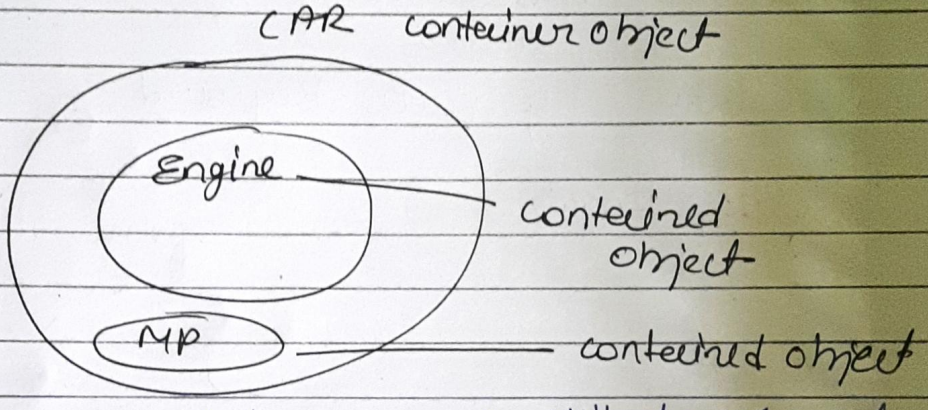
```
class Engine
{

}
class Car
{
    Engine e = new Engine();
}
```

↳ same properties as per requirements

→ Car Has-A engine
→ two classes are not tightly coupled
→ effect on one - class not affect another.







→ If no container object still chances of existence of contained object then weak bonding composition.

→ If no container object lead to ~~no other~~ no existence of contained object then strong bonding aggregation

Polymorphism

multi forms

Ex;

Water : solid, liquid, gas

sound : bark, roar, meow

Types:

1) Compile Time Polymorphism
   - static polymorphism
   - method overloading
   - handle by compiler

2) Run Time Polymorphism
   → Dynamic polymorphism
   → method overiding
   → handle by JVM

Method Overloading	Method Overiding
1) Same name	1) Same name
2) Same class	2) Different class
3) Different arguments: - No. of arg - Seq of arg - Type of arg	3) same arguments - No. of arg - seq of arg - Type of arg
	4) Inheritance

Method Overloading

```
class Test
{
  void show()
  {
    sout(1);
  }
  void show(int a)
  {
    sout(2);
  }
  psvm(string[] args)
  {
    Test t= new Test();
    t.show();      // 1
  }
}
```

1) Can we achieve Method Overloading by changing the return type of method ?

```
void show(){ }
string show(){ return ""; }
```

here we can not achieve method overloading by changing just return type.

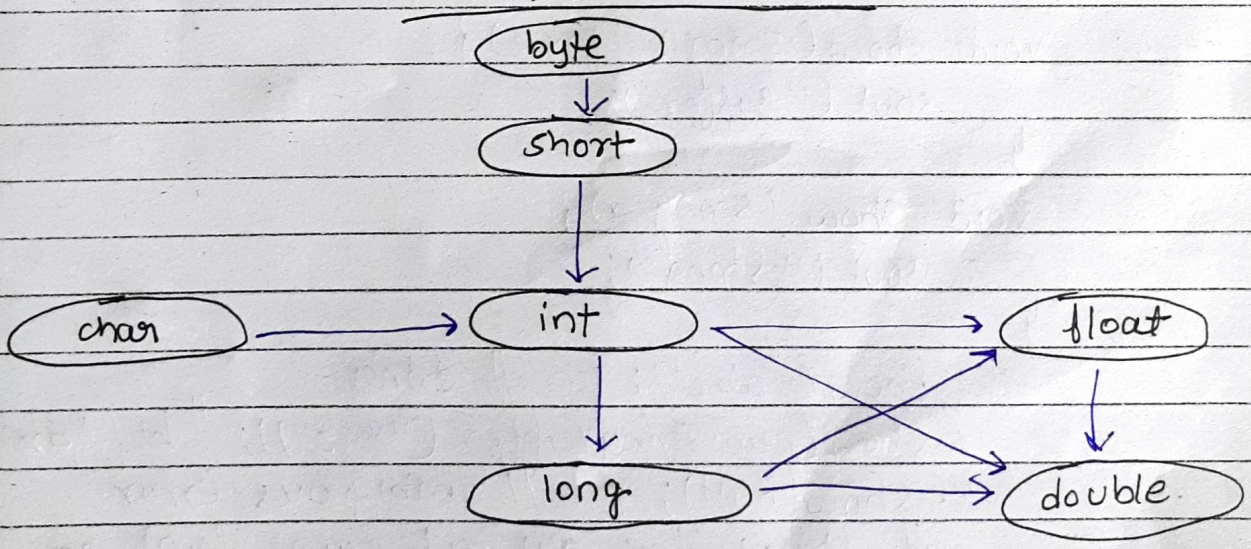
2) Can we overload java main() method?

Yes, we can overload main method

```
class Test{

    public static void main (int a){
            sout(a);
    }
    public static void main (String args[]){
            sout(1);
            Test t = new Test();
             t.main(20);          // 20
    }
}
```

∵ JVM always calls main() method which recieves string array as arguments only.

Automatic Promotion



Ex:

```
class Test {
    void show(int a){
        sout("int");
    }
    void show(String a){
        sout("String");
    }
    psvm(){
        Test t = new Test();
        t.show('a');
    }
}
```

automatic promotion to int.

※ While resolving Overloaded Methods, compiler will always give precendence for the child type argument.

case:

```
void show(StringBuffer a){
    sout("Buffer");
}
void show(String a){
    sout("String");
}

show("abc");                          // string
show(new StringBuffer("xyz"));        // buffer
show(null);                           // ambiguous error
```

String and StringBuffer are at same level so null cannot be refered.

Case:

Varargs: The varargs allows the method to accept zero or multiple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it lead to the maintenance problem. If we don't know how many arguments we will have to pass in the method, Varargs is the better approach.

```
class Test {
    void show (int a) {
        sout ("int");
    }
    void show (int...a) {
        sout ("varargs");
    }
    psvm (String args[]) {
        Test t = new Test();
        t.show (10);        // int
        t.show (10,20);     // varargs
        t.show ();          // varargs
    }
}
```

Method overriding

```
class Test{
    void show(){
        sout(1);
    }
class XYZ extends Test{
    void show(){
        sout(2);
    }
    psvm(String args[]){

        Test t = new Test();
        XYZ x = new XYZ();
        t.show(); //1
        t.show(); //2
    }
}
```

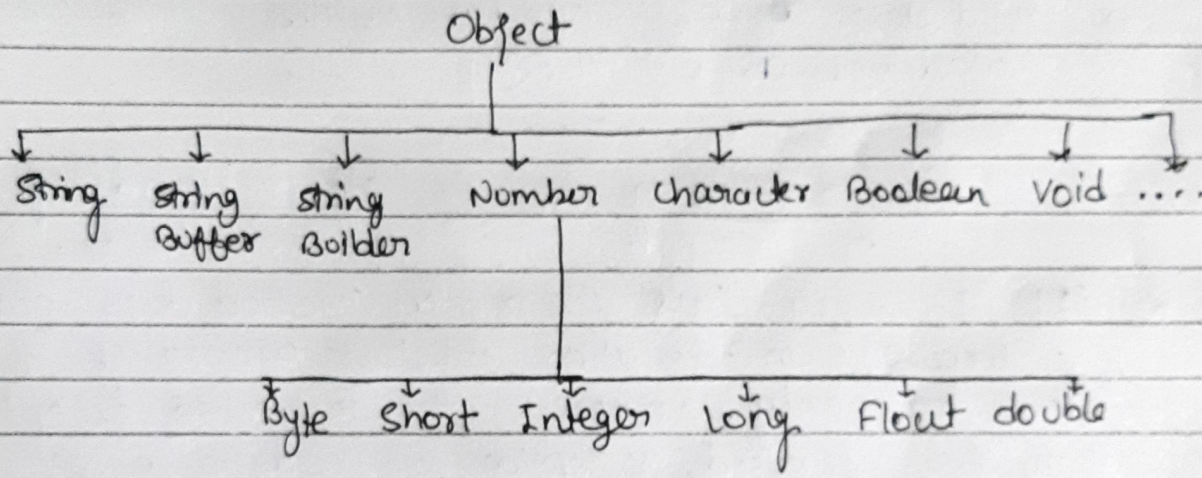
※ the object of which class you create its method get called.

// method overriding allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes. The implementation in the subclass overides (replaces) the implementation in the superclass by providing a method that has same name, same parameters or signature, and same return type as the method in the parent class.

→ Do overriding method must have same return type?

Before 1.4 version of Java, return type must same

From
~~After~~ 5.0 version we can use covariant return type




```
class Test {
    // Number
    parent class show() {
        sout(1);
    }
class Xyz extends Test {
    // Integer
    child class show()
    { sout(2);
    }
```

Access Modifiers in Overriding

→ By default, default

→ The access modifier for an overiding method can allow more, but not less, access than the overriden method.

→ child class के overridden method का access modifier बड़ा होना चाहिए।

Overiding and Exception Handling

Rule 1: If super-class overridden method does not throws an exception, subclass overiding method can only throws the unchecked exception, throwing checked exception will lead to compile time error.

Rule 2: If the super class overridden method does throws an exception, subclass overiding method can only throw same, subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error. Also there is no issue if subclass overridden method is not throwing any exception.

Overriding and Abstract Method

Abstract methods always override. otherwise compile time error.

```
abstract class Test{
    abstract void display();
}
  class Xyz extends Test {
     void display() {
     }
     psvm() {
      Xyz x = new Xyz();
         x.display();
     }
  }
```

Invoking overidden method from sub-class:

```
class Test {
   void show(){ sout(1);
}
 class X extends Test {
    void show()
    {  super.show()
       sout(2);
    }
    psvm(){
     X ob = new X();
     ob.show()        //1 2
    }
}
```

We can called parent class overridden method through child class object using super keyword

Which methods cannot override?

→ Final methods can not be overridden

→ Static method

METHOD HIDING?

When you defines a static method with same signature as a static method in base class, it is know as method hiding.

→ Private Methods

Private methods are bonded to compile time therefore can't be overridden.

Overriding and synchronized / strictfp method

The presence of synchonized / strictfp modifier with method have no effect on the rules of overriding

i-e. it is possible that a synchronized / strictfp method can override a non-synchronized / strictfp one and vice-versa.

strict floating point

Abstraction

Abstraction	Encapsulation
1) Abstraction is detail hiding (implementation hiding)	1) Encapsulation is data hiding (information hiding)
2) Data abstraction deals with exposing the interface to the user and hiding the details of implementation.	2) Encapsulation groups together data and methods that act upon the data.

→ Abstraction is hiding internal implementation and just highlighting the setup services that we are offering.

Abstraction
- Abstract class (0-100%)
- Interfaces (100%)

→ A method without body called abstract method.

→ Abstract method must in abstract class
Abstract class can contain no abstract methods.

Ex:

```
abstract class Vehicle {

    int no-of-types;
    abstract void star();

}
class Car extends Vehicle {

    void star() {
        sout ("car started");
    }

}

class Scooter extends Vehicle {

    void start () {
        soot ("Scooter started");
    }

}
```

* Abstract methods in an abstract class are meant to be overidden in derived concrete classes otherwise compile time will be throwon.

* Abstract class cannot be instantiated, means we can't create an object of abstract class. but can create reference.

Interfaces

→ Similar to abstract class but they have all methods as abstract.

→ ~~Java~~ Interface is a group of related methods with empty bodies.

→ Interface is a blueprint of the class which tells the class what ~~do~~ must to do.

→ It supports multiple inheritance

→ It can be used to achieve loose coupling

```
Interface InterfaceName {
    // abstract methods
    // fields
    // 8th version
    -> We can create default concrete methods
    -> static methods

    // 9th version
    -> private methods also can create
}
```

all methods are by default public abstract by compiler.

By default public static final

```
interface I1{
    public abstract void show();
    public static final int a=10;
    default void display(){
        // concrete methods
    }
    public static void run(){
        // concrete method
    }
}

class Test implements I1{
    public void show(){
        SOut(1);
    }
    psvm(){
        Test t= new Test();
        t.show();
    }
}

// We cannot create object of interface
```

by default

keyword

Multiple inheritance :

```
interface A {
    void show();
}

interface B {
    void show();
}

class Myclass implements A, B {

    public void show() {
        sout ("show method A, B");
    }
}
public class Main {
    psvm ( ) {

    Myclass ob = new Myclass ();

    Ob.show()    // Sinle show method A, B

    A objA = obj;    // reference of A
    B objB = obj;    //  "        "  B

    objA.show();    // Show method A, B
    objB.show();    //  "    "    "  "
    }
}
```

*** Important ***

WHY MULTIPLE INHERITANCE IN INTERFACES NOT IN CLASSES?

→ Java does not allow multiple inheritance with classes to avoid ambiguity and complexity especially the diamond problem.

1) Interface Don't Inherit State

→ Interfaces do not store data or fields like classes do

→ So, there is no risk of inheriting duplicate states or variables

2) Before JAVA 8, all methods inside an interface were explicitly (by default) public and abstract

```
interface A {
    void show();   // implicitly public and abstract
}
interface B {
    void show();  // same method name.
}
```

* Both A and B have the same method show()
* Neither provides any implementation
* A class that implements both the interfaces must implement show() method - no ambiguity

Encapsulation

→ Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.

Steps to achieve encapsulation:

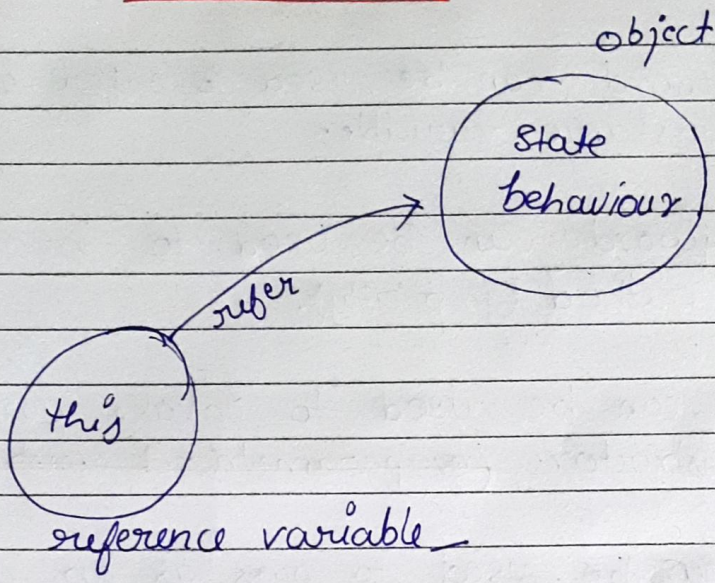
1) Declare the variables of a class as private
2) Provide public setter and getter methods to modify and view the variables values.

Data Hiding: When you declare a variable private it can be accessed within the class only ie. hide the data.

```
class Employee {

    private int emp_id;
    public void setEmp_id(int emp_id){
        emp_id = emp_id1;
    }

    public int getEmp_id(){
        return emp_id;
    }
}
```

Data hiding (encapsulation)

This Keyword




```
class Test{
    int i;          // instance variable

    void setValues(int i){      // local variable
        this.i=i;   // this.i refers to current class instance variable. i=i; // here instance variable and local variables are of same name, here in both will be local variable so output =0
    }
    void show(){


    }
    psvm (){
        Test t = new Test();
        t.setValues(10);
        t.show();   // 0 without this
    }
}
```

Uses of 'this' keyword:

1) this keyword can be used to refer current class instance variable.

2) this keyword can be used to invoke current class method (implicitly).

3) this() can be used to invoke current class constructor. ~~(ex parameterized constructor)~~

4) this can be used to pass as an argument in the method call.

5) this can be used to pass as an argument in the constructor call.

6) this can be used to return the current class instance from the method.

2)
```
class Demo {
    void display() {
        sout("hello");
    }
    void show() {
        this.display();
    }
    psvm() {
        Demo d = new Demo();
        d.show();
    }
}
```

here compiler automatically adds this keyword while invoking the method.

(3) class Demo {

```
    Demo() {          this(10);
        sout("No arg");
    }
    Demo(int a) {
        this();
        sout("arg");
    }
    psvm() }
```

```
Demo d = new Demo();  // No arg
Demo d = new Demo(10);  // No arg
                        // arg
Demo d = new Demo();  // arg
                      // no arg
```

(4)

```
class Demo{
    void m1(Demo d){
        sout("m1 method");
    }
    void m2(){
        m1(this);
    }
    psvm(){
        Demo d = new Demo();
        d.m2();    // m1 method
    }
}
```

Super

-> Super is also the reference variable

-> If any class inherit/extends another class then super will refer to the object of parent/patent second class while this keyword refer to the variable of same class.

```
class A extends B{

class A {
    int a=10;

}
class B extend A
{
   int a = 20;
   void show (int a)
   {  sout(this.a); // 10
    sout(a);        // 30
   }   sout( super.a); //10
}psvm () {

  B obj = new B();
  Obj.show(30);
```

Uses:

1) "super" keyword can be used to refer immediate parent class instance variable.

~~class A~~
~~{~~
~~int i=~~

```
class A {
    int i=10;
}
class B extends A {
    int i=20;
    void show(int i) {
        Sout(i);          // 30
        Sout(this.i);     // 20
        Sout(super.i);    // 10
    }
    psvm (String [] args) {
        B ob= new B();
        ob.show(30);
    }
}
```

2) Super keyword can be used to invoke immediate parent class method.

```
class A {

    void m1(){
        sout("class A");
    }
}
class B extends A {
    void m1() {
        sout("class B");
    }
    void show() {
        m1();       // class B
        super.m1(); // class A  {parent class}
    }
}
psvm (String[] args) {
        B ob = new B();
        ob.show();
    }
}
```

super => keyword
super() => constructor call

3) Super keyword can be used to invoke immediate parent class constructor.

```
class A {

    A() {
        sout("class A");
    }
class B extends A {
        B() {
            super();
            sout("class B");
        }
        psvm(String[] args){

            B ob = new B();
        }
}
```

here compiler automatically put super constructor or we can declare it manually.

output
=> class A
=> class B

If we declare the parent class and child class both constructor and create child class object the compiler automatically put super();

Final keyword

Variable	method	class
→ If we want a variable constant like no one can change it. Ex. Pi value, G force value which are constant	→ If we want a method to not override	→ If we want a class not to be inherited

Access Modifiers	Non-access Modifiers
public	static
private	final
protected	abstract
default (No modifier)	synchronized
	transient
	volatile
	strictfp

static keyword

-> static variable can be used be class level variable not local variable

-> static methods

-> static block

-> static inner class ⟶ (outer class X)
(nested class)

static cannot be used with local variable

```
class A {
    static int a = 10;  // not error
    void show() {
        static int b = 10;  // error
    }
    psvm() {
    }
}
```

-> static variables belong to class not object

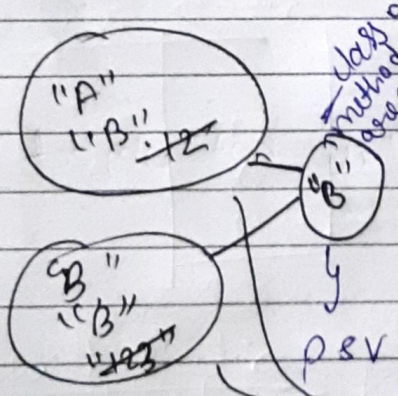
```
class A { int b = 20;
    static int a = 10;
}
class B {
    psvm() {
        sout(a);      // error
        sout(A.a);    // not error
        sout(A.b);    // error
    }
}
```

Why variables need to be static?

→ static variable are used for memory management.

- When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Thus company name i.e. SP will be shared among e1, e2 and all the objects.

```
class Emp{
    string name;
    static string company = "B";
    string emp_id;

    Emp( string name, string company,
         string emp_id){
        this.name = name;
        this.company = company;
        this.emp_id = emp_id;
    }

    psvm (String args []){
        Emp ob1 = new Emp("A", "B", "24");
        Emp ob2 = new Emp("8", "B", "23");
    }
}
```

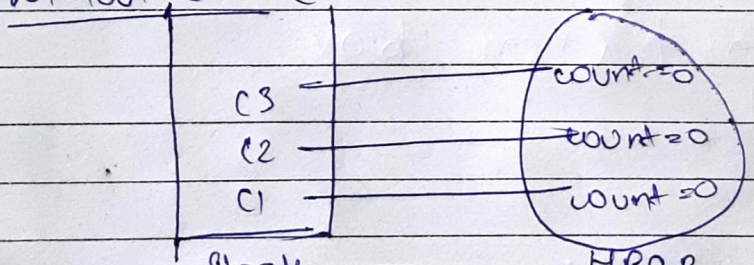


Static variable gets memory only once in the class area at the time of class loading

Counter

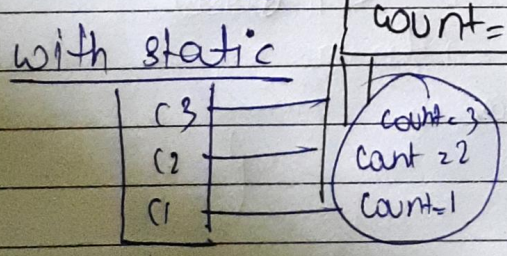
```
class CounterDemo
{  static int count=0;  // this is for class
// int count=0;        // this is for object not class
   CounterDemo() {
      count++;
      Sout(count);
   }
   psvm (String [] args) {
      CounterDemo c1 = new CounterDemo();
      CounterDemo c2 = new CounterDemo();
      CounterDemo c3 = new CounterDemo();
                                          // ...
                                          ///
                                          ///
}
```

without static



// every time constructor ^ initialise count=0

with static



class Area memory

count will become zero again when you load the class

Static Methods

→ Static methods belongs to class not object

→ static method can be called directly by class name.

ClassName.methodName();

```
class Test {

    void display() { }

    static void show { }

    psvm() {
        Test t = new Test();
        t.display(); // simple method
        show(); // can be called within class only
        Test.show(); // static method
    }
```

Rules of Static Methods

1) "static" methods belongs to the class, not to the object.
2) A "static" method can be accessed directly by class name and doesn't need any object.
3) A "static" method can access only static data. It cannot access non-static data (instance data).

4) A "static" method can ~~calls~~ call only other static methods and cannot call a non static method.

5) A "static" method cannot refer to "this" or "super" keyword in anyway.

Ex of Rules:

```
class staticDemo {
    int i = 10;  => static int i = 10;
    static void display() {
        sout(i);  //error static can access only static data
    }
    static void m1() { sout(1); }
    void m2() { m1(); }
        // error static can call static method
    ~~static~~ int a = 10;
    static void m3() {
        sout(this.a);  //error
    }
}
```

Static Block

Static block

```
class Test
{
    static { }
}
```

⇒ call static ?

no way to call static method, static block executes automatically when the class is loaded in the memory.

⇒ Before 1.6 version of java we can execute static block without main function but after 1.6 version there should be main method.

⇒ static block executes first.

⇒ multiple static blocks are posible

⇒ execution ⇒ top → Bottom

```
class {
    static {
        sout("static");
    }
    psvm() {
        sout("main");
    }
}
// static
// main
```