CHAPTER 2 : Object Oriented Programming Structure (OOPS)

There are four principles of oops known as Abstraction, Encapsulation, Inheritance & Polymorphism.

Chaining of Methods : For most of the methods in String and StringBuffer the return types are the same String and StringBuffer objects only.

After Applying a method we are allowed to call another method on the result which forms a method change.

sb.m1().m2().m3().m4().m5()…..;

All these method calls will execute from left to right.

```
StringBuffer sb = new StringBuffer("raju");
sb.append("software").reverse().insert(2,"abc").delete(2,5).append("xyz");
```

Object based programming language(obpl) :-- A language which supports 3 principles of oops is known as obpl(ex– visual basic, as it does not support inheritance)

Object oriented programming language(oopl) :-- Languages which support 4 principles of oops are known as oopl. e.g..c++

Puerly OOPL :-- Languages which support 4 principles of oops as well as everything within the concept of a class are known as purely oopl. Eg. Java, c#.net etc.

Java is more object oriented than c++ because it does not support any thing out side of a class. Hence we cannot define global method or global variable in java.

In case of obpl we are concentrating on logic than on data hence an application can be developed by top-down approach.

In case of oopl we are concentrating on data than on logic hence an application can be developed by bottom up approach.

What is difference between Top-Down approach and Bottom-up approach ?

In case of Top-down approach we are concentrating the 'what to do' i.e. on requirement part of the s/w requirement specification and then we are thinking 'how to do it' i.e. logic to implements. Hence we are concentrating on logic than on data.

In case of Bottom-up approach we are reading the s/w requirement specification to pick up nouns and verbs, then we've to group related nouns and verbs to create class, because nouns will become properties and verbs will become behavior of a class. Here we are concentrating to create class i.e. user defined data type. Once the class is created then we should focus on 'what to do' i.e. on requirement part of the s/w requirement specification and then we are thinking 'how to do it' i.e. logic to implements by creating object of the class. Hence here we are concentrating on data than on logic.

Advantages: The advantage of oopl are :

1) Development will be easy.
2) Debugging will be faster.
3) Maintenance will be low cost effective.
4) Codes will become independent and can be reused.

Abstraction: The collection of similar properties and behaviors of related objects are known as abstraction, or we can say, Abstraction refers to the act of representing essential features without including the background details or explanations. It is implemented by class.

Encapsulation: Hiding internal information from external manipulation is known as encapsulation, or we can say, Encapsulation is a technique used for hiding the properties and behaviors of an object and allowing outside access only as appropriate. It prevents other objects from directly altering or accessing the properties or methods of the encapsulated object.. It is implemented by access specifier.

Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class

```
public class EncapTest{
   private String name;
   private long regdno;
   private int sem;

   public int getSem() {
     return sem;
   }
   public String getName() {
     return name;
   }
   public String getRegdno() {
     return regdno;
   }
   public void setSem( int newsem) {
     sem = newsem;
   }
   public void setName(String newName) {
     name = newName;
   }
   public void setRegdno( String newId) {
     regdno = newId;
   }
}
```

What is the difference between abstraction and encapsulation ?

- **Abstraction**: Abstraction focuses on the outside view of an object (i.e. the interface), Encapsulation (information hiding) prevents clients from seeing it's inside view, where the behavior of the abstraction is implemented.
- **Abstraction** solves the problem in the design side while Encapsulation is the Implementation.
- **Encapsulation** is the deliverables of Abstraction. Encapsulation talks about grouping up your abstraction to suit the developer needs.

Inheritance: Using the properties and behavior of an existing component in a new component is called inheritance. It provides the benefit of code reuse and polymorphism. It is implemented by inheriting a class from another class, inheriting a class from an interface.

Polymorphism: When a component provides multiple functionality then it is called polymorphism. It is achieved by overloading and overriding.

Class: -- It is a component which is used to represent and manipulate the data or information or value. But the data or the value or information are generally stored in the memory location of an object.

Syntax:--

```
[access specifier][access modifier]  class  classname {
        Data member
        Constructor
        Method
        Blocks( static & nonstatic)
```

Inner class/interface.
}; → semicolon at the end of the class is optional in java but compulsory in c++.

```
Ex :    class A{ };
        class B { int x , y; }
class C {
          void sum(int a, int b){
            System.out.println("sum:" + ( a + b ) );
}};

class D{
        int p;
        D(){  p = 12; }
}
```

The classes in C++ contains 5 members as data member, constructor, destructor, member method, nested class.

Let's define a complete or concrete class called Arith which contain all members

```
class Arith{
        int x, y=12;   → Data member/instance variables/ properties of a class.
        Arith(){x=5;   y=8;        } → Default constructor
        Arith(int p, int q){      x=p; y=q;          } → Parameterized constructor
        void add(){    System.out.println("sum = " + (x+y) );  }
        int diff(){       return x-y;  }                          } Member function/ method/
        int cube(int z){          return z*z*z;     }                  behavior of class
};
```

- Generally a class contains some members known as data member/properties/instance variables (instance variables), constructer and method/member function /behavior.
- The instance variable behaves like a global member for the other members (constructer and method) of the class because the instance variables will accept the value from constructors & can be manipulated within the method.
- Java does not support destructor and delete operator. Although we cannot use address of operator still we can create copy constructer.
- Java does not support nested constructor or nested method. Similarly we cannot define a method within a constructor and vice versa. A member of a class can be accessed by other member directly/indirectly.
- We cannot write any independent statements outside of a constructor or method in java/c++.
- We can initialize data members of the class directly in java, but it will be error in c++. In case of java we can define inner (nested) class as a member within a class, it is also possible in c/c++.
- The position of members of the class in java is not fixed.
- The access specifier of members and the class is implicitly default in java. It can be accessed by any other class that stored within the same folder, but in c++ by default the members are private and class is public.

Situation	public	Protected	Default	private
Accessible to class from same packages?	yes	Yes	Yes	no
Accessible to class from different packages?	yes	no, unless it is a subclass	No	no

Q. WAP to represent the states of India along with CM & population.
Object :--

- It's an instant of a class that shares the properties and behavior of class.
- Each object has a state, identity and behaviour. The state is represented by the value of instance variable of the object, where as the identity is represented by the name of the object.
- Each object will get separate copy of instance variables of the class. Since, the variables of the class belong to instances (objects) of the class, hence they are known as instance variable of the class.
- The member of a class are generally accessed by the object (instance variables, method) or for the object creation(constructer).

State:

- Instance variables value is called object state.
- An object state will be changed if instance variables value is changed.

Behavior:

- Behavior of an object is defined by instance methods.
- Behavior of an object is depends on the messages passed to it.
- So an object behavior depends on the instance methods.

Identity:

- Identity is the hashcode of an object, it is a 32 bit integer number created randomly and assigned to an object by default by JVM.
- Developer can also generate hashcode of an object based on the state of that object by overriding hashcode() method of java.lang.Object class.
- Then if state is changed , automatically hashcode will be changed.

```
                                    object  definfition
    Arith     a3      =    new    Arith( 15, 20);
object  declaration
```

new allocates the memory location required for the object depending upon the no. of data member of the class. Constructor provides the value for the instance variables of the object.

//using Arith class

```
class Calc {
        public static void main(String args[]){
        Arith  a1 = new Arith();
        Arith  a2 = new Arith(15,22);
        a1.add();
        a2.add();
        int z = a1.diff();
        System.out.printlln(z);
        System.out.printlln(a2.diff());
        //System.out.printlln(a1.add());
        System.out.println(a2.cube(6));
        new Arith(2,3).add();  //anonymous object-it is an object without any name.
        //Arith  a5(11,4);    //it's an object in c++, but it is an error in java.
        Arith  a3;  // a3 is an object in C++ but it is a reference in java, that will point/refer to the memory
                    location of an object of same Arith class or derived class.
        //a3.add();  //error in java, because a3 does not refer to any memory location, but no error in C++
        a3 = a1;   // now a3 point to memory location of a1.
        a1.x=20;
        a3.add();
        Arith  a4;
        a4 = new Arith(11,12);
        a4.add();
}};
```

There are **different ways to create an object** in java. They are:

- By new keyword (including copy constructor).

- By newInstance() method
- By clone() method
- By factory method etc.
- By deserialization
- By lookup()

CLASS	OBJECTS
1. The process of binding the data members and associated methods in a single unit is known as class	1. Class variable is known as object
2. Class will have logical existence.	2. Objects will have physical existence.
3. Whenever we define a class no memory space is allocated for the data members and the methods.	3. Whenever we create an object we get the memory space for data members and methods of a class.
4. In one java program the definition of one particular class will exists only once.	4. With respect to one class we can create multiple objects.
5. Whenever we execute the java program the class will be loaded first in the main memory.	5. After loading the class whose corresponding class object can be created.

Java Naming conventions

Java **naming convention** is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method etc. But, it is not forced to follow. So, it is known as convention not rule. All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. Readability of Java program is very important. It indicates that **less time** is spent to figure out what the code does.

Name	Convention
class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thread etc.
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionListener etc.
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), println() etc.
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

If name is combined with two words, second word will start with uppercase letter always e.g. actionPerformed(), firstName, ActionEvent, ActionListener etc.

CONSTRUCTER : It is a block of code which is used to initialize the data members of a class during object creation. **Constructor in java** is a special type of method that is used to initialize the object. Java constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.

- The name of the constructor must remain the same as that of the class name in proper case.
- It is always invoked/called implicitly with new operator when the object is created..
- A constructer has no access modifier but it can have access specifier.
- The access specifier of the constructor may or may not be private.
  - If the access specifier of the constructor is private then an object of corresponding class can be created in the context of the same class but not in the context of some other classes.
  - If the access specifier of the constructor is not private then an object of corresponding class can be created both in the same class context and in other class context.
- A constructer can be overloaded but it cannot be overridden.
- Constructor should not return any value not even void. Because basic aim is to place the value in the object. (if we write the return type for the constructor then that constructor will be treated as ordinary method in java, but it will be generate compilation error in C++).
- A constructer has an implicit return type called class that return an object.
- Constructor definitions should not be static. Because constructors will be called each and every time, whenever an object is creating.
- Constructors will not be inherited from one class to another class (Because every class constructor is created for initializing its own data members).

```
class A{
        int x,y;
        A() {  x = 4;   y = 8;  }
        A(int  p) {  x = p;   y = p+2  }
        A(int p,intq) {  x=p;    y=q;  }
        void A() {
          x = x+2;      y = y-2;  }
        void A(int p) {
           x = p+1;    y = y-1;  }
        void A(int p,  int q) {
           x  =  x+p;   y  =  q-y;   }
};
```

```
class B{
    public static void main(String args[]){
        A a1 = new A();
        A a2 = new A(7);
        A a3 = new A(10,12);
        a1.A(3,5);
        a2.A();
        a3.A(2);
        System.out.println(a1.x + " : " +a1.y);
        System.out.println(a2.x + " : " +a2.y);
        System.out.println(a3.x + " : " +a3.y);
}}
```

TYPES OF CONSTRUCTER :

- Generally constructers are of 2 types, known as default constructer & parameterize constructer. As well we can define copy constructer even if does not support address of operator.
- The default constructor generally does not accept any parameter and we can define only one default constructor in a class, but parameterized constructor accept some parameter, hence we can define many parameterized constructor in a class.
- The default constructor is generally used to provide the value to the data member according to the programmer (creator of the class), but parameterized constructor is used to initialized the data member by other programmer who will create the object. Therefore parameterized constructor is a mechanism or a way by which we can transmit data or information from one class to other.

Q. WAP to create a class D consisting of a constructer and 2 methods. Create one object of class D in class E so that both the method of class will execute but do not invoke a method of class D in class E.

COPY CONSTRUCTOR : It is a constructor which copies the value ofdata member of an object given in the constructor(a1) to the data member of another object created by the constructor(a2).

```
class A{
        int x,y;
        A( ){x=5; y=9;}
        A(A ob){    //copy constructer
                x = ob.x;  y=ob.y;    }
}
```

```
class B{
 public static void main(String args[] ){
        A a1 = new A();
        A a2 = new A(a1);
        System.out.println(a2.x+" : "+a2.y);
}}
```

Copying values without constructor : We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
class Stud{
        int id;
        String name;
        Stud(int i,String n){
        id = i;
        name = n;
        }
        Stud(){    }
        Void  display(){
           System.out.println(id+" "+name);
        }
```

```
public static void main(String args[]){
        Stud s1 = new Stud(111,"Karan");
        Stud s2 = new Stud();
        s2.id=s1.id;
        s2.name=s1.name;
        s1.display();
        s2.display();
}}
```

CONSTRUCTOR RULES IN JAVA :

1) If we do not provide any constructer in a class then **compiler automatically creates** a default constructor. The default constructor will initialize the instance variable with default only if the instance variable has no values.

```
class A{
        int x , y=12;
        boolean flag;
        public static void main(String args[]){
        A a1 = new A();
        System.out.println(a1.x + " : " + a1.y + " : " + a1.flag);
}}
```

2) If we do not provide any constructer in a class then we will never get the parameterized constructor hence if we write the following statement in above class, then it will generate compilation error i.e. A a2 = new A(12,14,false);

3) If we define parameterized constructor in a class then we can create object by using parameterised constructor, but we will never use default constructor, because java will not provide default constructor if we provide parameterized constructor.

4) Therefore in order to use both constructors we have to define both constructors.

Q. WAP to create a class F consisting of 4 constructors create one object of class F in class G, so that 4 constructer of class F will execute.

Q. WAP to create a class H that will perform arithmetic addition, subtraction, multiplication, division of 2 no. give form the command lines of class K.

Q. WAP to create a class called M which will consist an array as the data member, initialize the array through constructer, the class should contain a method called add which will add a new value at the end of the existing array by increasing the size of array, the removed method will remove the last element of the array, the show method will displaying the element of array. Use class M in class N.

Q. WAP to create a class Snake that will hold information of a snake like name, habitat, length & type(poison or not). Create a snake array in class Jungle & pass the snake array to the bite method of the jungle class to display the information of snake.

Q. WAP to create a class R. Explain whether we can create an object of class R within its member or not.

Q. WAP to create a class S & T. The class T should use the object of class S as a data member.

this keyword:

- It is a pointer in c++ but it is a keyword in java. It is also called a non static variable in java.
- It is used to refer the object for which the current execution is going on. "this" is used to refer the object during the current context.
- When we evaluate instance variables of the class within the constructer or member method then interpreter uses "this" implicitly to evaluate the instance variables as "this.instance variables".
- But when a local variable/parameter of a method/constructor has same name as that of the name of the instance variables, then within the local scope the local variable/parameter will get more priority or precedence and no value will given to the instance variables. As in the following parameterized constructor.

```
class A{
   int  x ,  y;
```

```
class B {
 public static void main(String args[])   {
```

```
    A() {
         x = 4 ;   y = 8 ;    }
    A( int  x,   int   y  ) {
          x  =  x;     y = y;     }
    void add(){
       System.out.println ("sum : "+(x+y) );
}};
```

```
    A  a1  =  new  A();
    A  a2  =  new  A(15,20);
    a1.add();
    a2.add(); //sum :
}}
```

In the above parameterized constructor the parameters(x,y) will collect the value during the object creation but the value will not be given to the instance variables(x, y) because the parameters will get more precedence than the instance variables. Hence we have to use this keyword explicitly to refer the data member within the parameterized constructor as given bellow.

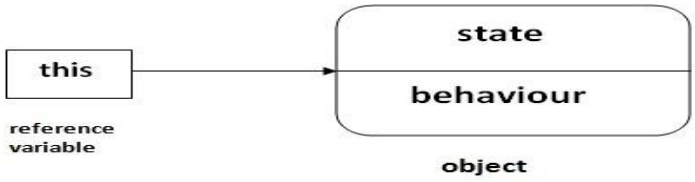
```
A(int x ,  int  y)  {
  this.x=x;
  this.y=y;
}
```

After this modification, if we invoke a2.add() then we will get 35

this keyword in java : In java, this is a reference variable that refers to the current object Here is given the 6 usage of java this keyword.

1. this keyword can be used to refer current class instance.
2. this() can be used to invoke current class constructor.
3. this keyword can be used to invoke current class method (implicitly)
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this keyword can also be used to return the current class instance.

Suggestion: If you are beginner to java, lookup only two usage of this keyword.



Proving this keyword : Let's prove that this keyword refers to the current class instance. In this program we are printing the reference variable and this, output of both variables are same.

```
class A{
void m(){
        System.out.println(this);
        //prints same reference ID
}
```

```
public static void main(String args[]){
        A  obj=new A();
        System.out.println(obj);
        //prints the reference ID
        obj.m();
}}
```

1. The this keyword can be used to refer current class instance variable: If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity. Understanding the problem without this keyword :

```
class Stud{
   int id;
   String name;

   Stud(int id, String name){
        id = id;
        name = name;
   }
```

```
   public static void main(String args[]){
        Stud  s1 = new Stud(111,"Karan");
        Stud  s2 = new Stud(321,"Aryan");
        s1.display();
        s2.display();
   }
};
```

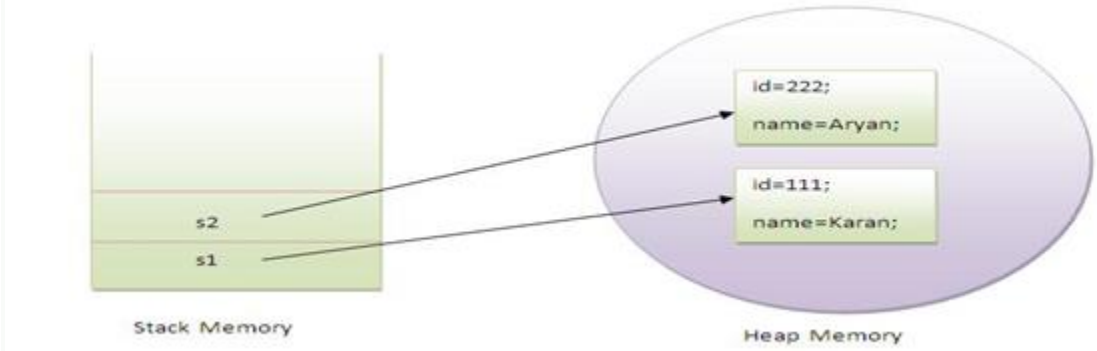
```
    void display(){
        System.out.println( id+" "+name);
    }
```

```
Output:   0 null
          0 null
```

In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable Solution of the above problem by this keyword

```
class Stud1{
    int id;
    String name;

    Stud1(int id,String name){
    this.id = id;
    this.name = name;
    }
   void display(){
         System.out.println(id+" "+name);
   }
```

```
    public static void main(String args[]){
        Stud1  s1 = new Stud1(111,"Karan");
        Stud1  s2 = new Stud1(222,"Aryan");
        s1.display();
        s2.display();
} }

Output    : 111 Karan
            222 Aryan
```



Where this keyword is not required : If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

```
class Stud2{
    int id;
    String name;
    Stud2(int i,String n){
        id = i;
        name = n;
    }
    void display(){
        System.out.println(id+" "+name);
   }
```

```
public static void main(String args[]){
        Stud2 e1 = new Stud2(111,"karan");
        Stud2 e2 = new Stud2(222,"Aryan");
        e1.display();
        e2.display();
}};

Output:   111 Karan
          222 Aryan
```

2) this() can be used to invoked current class constructor : The this() constructor call can be used to invoke the current class constructor (constructor chaining). This approach is better if you have many constructors in the class and you want to reuse that constructor.

//Program of this() constructor call (constructor chaining)

```
class Stud3{
    int id;
    String name;
    Stud3(){
        System.out.println("default constructor is invoked");
        }

    Stud3(int id,String name){
```

```
public static void main(String args[]){
    Stud3 e1 = new Stud3(111,"karan");
    Stud3 e2 = new Stud3(222,"Aryan");
    e1.display();
    e2.display();
   }
}
```

```
        this ();//it is used to invoked current class construct
or.
        this.id = id;
        this.name = name;
    }
    void display(){        System.out.println(id+" "+name);
        }
```

```
Output:
      default constructor is invoked
      default constructor is invoked
      111 Karan
      222 Aryan
```

Q: Where to use this() constructor call?

Ans :constructor chaining

The this() constructor call should be used to call the constructor in the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
class  H {
H(){
  this(44.66f);
   System.out.println("def. cons. of H");
}
H(int    p){
  this();
  System.out.println("int para cons. of H");
}
H(float  p){
   this("hi");
   System.out.println("float para cons. of H");
}
```

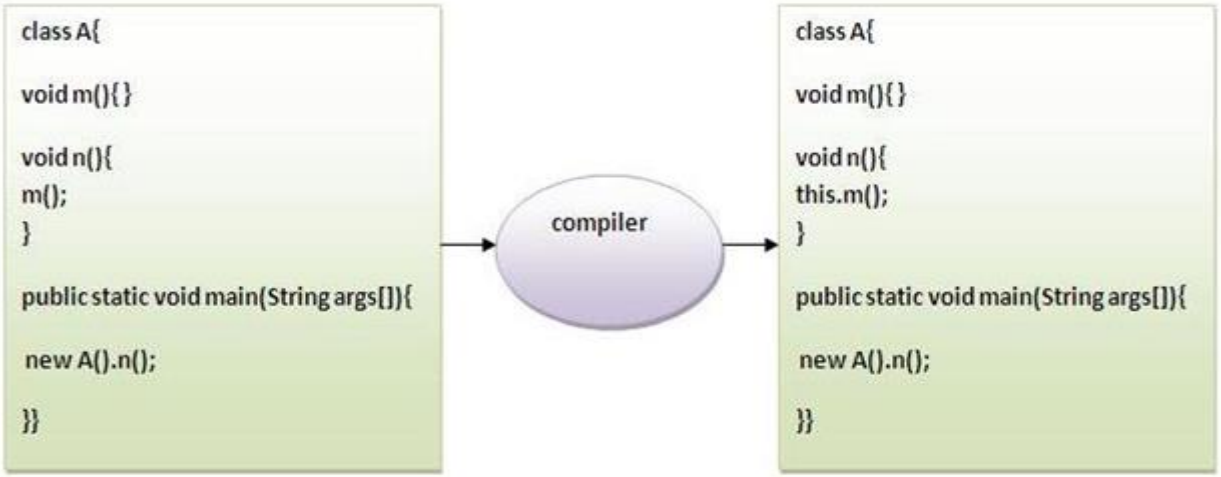
```
H(String  p){
   System.out.println("String para cons. of H");
}};

class  G {
 public static void main(String args[]) {
          H  h1= new  H(12);
}};
```

this() - It invokes default constructor of same class

this(parameter) - It invokes parmeterized constructor of same class.

When we use this() / this(para) to invoke other constructor of same calss, then it must be first executable statement in a constructor, hence more than one this()/ this(para) cannot be used in a constructor.

3)The this keyword can be used to invoke current class method (implicitly): You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example




```
class S{
  void m(){
        System.out.println("method is invoked");
  }
  void n(){
```

```
  public static void main(String args[]){
        S s1 = new S();
        s1.p();
  }}
```

```
        this.m();
//no need because compiler does it for you.
  }
void p(){
        n();
//complier will add this to invoke n() method as this.n()
  }
```

Output: method is invoked

4) this keyword can be passed as an argument in the method. The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
class S2{
  void m(S2 obj){
        System.out.println("method is invoked");
  }
  void p(){
        m(this);
  }
```

```
public static void main(String args[]){
        S2 s1 = new S2();
        s1.p();
}}
```

Output: method is invoked

5) The this keyword can be passed as argument in the constructor call. We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
class B{
  A4 obj;
  B(A4 obj){
    this.obj=obj;
  }
  void display(){
    System.out.println(obj.data);//using data member of A4 class
}}
```

```
class A4{
  int data=10;
  A4(){
   B b=new B(this);
   b.display();
  }
  public static void main(String args[]){
   A4 a=new A4();
}}
```

Output:1

6) The this keyword can be used to return current class instance : We can return the this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example of this keyword that you return as a statement from the method

```
class A{
A getA(){
        return this;
}
void msg(){
        System.out.println ("Hello java");
}}
```

```
class Test1{
public static void main(String args[]){
        new A().getA().msg();
} }
```

What are the difference between Conctructor & methods ?

	Constructors	**Methods**
Purpose	Create an instance of a class i.e. Constructor is used to initialize the state of an object	Group Java statements. i.e., Method is used to expose behavior of an object.
Modifiers	Cannot be abstract, final, native, static, or synchronized	Can have any access modifiers
Return Type	No return type, not even void	void or a valid return type
Name	Same name as the class (first letter is capitalized by convention) -- usually a noun	Any name except the class. Method names begin with a lowercase letter by convention -- usually the name of an action
this	this()/this(values) refers to another constructor	Refers to an instance of the owning class. Cannot

	in the same class. If used, it must be the first line of the constructor	be used by static methods.
super	super()/super(values) calls the constructor of the parent class. If used, must be the first line of the constructor	Calls an overridden method in the parent class
Inheritance	Constructors are not inherited	Methods are inherited
Compiler	The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Invocation	Constructor is invoked implicitly.	Method is invoked explicitly.

ACCESS SPECIFIER : It is a keyword used to hide or make available a component from the other components.

Eg. public → it can be accessed by anybody.
private → it can be accessed by same class.
protected → it can be accessed by child class.
Default/package → when we do not specify any access specifier then it will be default/ package and it can be accessible by anybody within same folder/package.

ACCESS MODIFIER : - It is a keyword which is used to modify the functionality to a component.

Eg. final:-- It makes read only attribute.
static:-- it can be accessed class name as well as by the object.
abstract:- only method declaration is available without the definition.
synchronized:-- multiple thread cannot be accessed a synchronized method simultaneously.
transient:-- The value of a variable will not persist even if the object is serialized.

FINAL VARIABLE :-- Declaring the final keyword before the variable makes it a constant in java(i.e. the value of the variable will not be changed) and it is called final variable.

e.g final int x=54;
---
x=34; //error, because final variable x is already initialized.

Declaring the final variable without any value makes it a blank final variable. The value of the variable can be assigned only once.

e.g. final int y; // y is a blank final variable , because it has no value
---
y=56; // ok, because blank final variable y has no value and we can initialize the value only once
---
y=42; // error; because y has already initialized.

Note : The position of access specifier & access modifier can always interchanged in java.

Java static keyword

The static keyword in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

STATIC VARIABLE/ CLASS LEVEL VARAIBLE:--

- Declaring the static keyword before instance variables makes it a static variable or class level variable in java.
- The static variable can be accessed by the object and by the class name in other classes.
- Each object will get separate copy of the data member of the class where as each object will never get separate copy of static variables because instance variables belongs to object where as static variable belongs to class(i.e. static variable are available per class basis, where as the instance variables are available per object basis).
- Each object can access or share or manipulate the value of static variable.
- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.
- Java static property is shared to all objects.

Advantage of static variable : It makes your program memory efficient (i.e it saves memory).

Ex: Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created .All student have its unique rollno and name so instance data member is good. Here, college refers to the common property of all objects. If we make it static, this field will get memory only once.

STATIC METHOD:--

- Declaring the static keyword before the return type of the method makes the method as static method.
- It can be invoked by the class name as well as by the class object in other class.
- A static method can't access non static member of the same class, hence it cannot access this/super, but it can access the static members of the same class.

```
class A{
int x;
static int cnt;
A() {   x = 4;   cnt++;    }
A(int p) {    x = p;    cnt++;  }
void show(){
    System.out.println( x  +  " : "  + cnt);
}
static void objCount(){
    System.out.println ("the no. of object :"+ cnt );
}};
```

```
class B{
    public static viod main(String args[]){
        System.out.println(A.cnt);
        A  a1 =  new  A();
        A.objCount();
        A  a2 =  new  A(13);
        a1.show();
        a2.show();
        new A().objCount();
        a2.show();
        System.out.println(a1.cnt);
    }};
```

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

Note : A local variable/parameter of a method/constructor can't be preceded by any access specifiers or by static modifier, otherwise it will generate compilation error.

public static void main(String args[]) :--

- Even if main is the method from which the program execution will start, still it is a member of any user defined class. Hence the main method can be accessed by anybody/interpreter only if it is public.
- Since main is a member, anybody or interpreter can access it by the object, but main method is static Hence we/other programmer/interpreter can invoke the main method by the class name.
- The args is the array of String objects which is a parameter of the main method & it is used to receive the command line argument.

```
class A{
  public static void main(int  ar){
     System.out.println("user define main of A");
  }
  static public void main(String main[]){
     System.out.println("original main of A");
     main(5);
}};
```

```
class B{
static public void main(String rgas[]){
   System.out.println("original main of B");
   String  p[]  =  { };
   A.main(p);
}};
```

If we remove public/static/string array from the main method , then the complier will compile the main() in a hope that it defined by the programmer, but the execution will never start from the main(), rather it will generate runtime exception.

Explanation of System.out.println() : --

```
class   A  {
 void show(){
    System.out.println("show of  A");
}};
class   B  {
  static  int  z =123;
  static   A  ob= new  A();
};
class C {
static   public  void  main(String   args[]){
   System.out.println(B.z);
   B.ob.show();
}};
```

```
public class   PrintStream ..... {
 public void println(String  k){
   //statement to print
}
 ----------
 ----------
 ----------
};
public class   System ..... {
  static   PrintStream  out= new  PrintStream(----, -
----);
 ----------
 ----------
 ----------
};
class C {
static   public  void  main(String   args[]){
   System.out.println("hello");
}};
```

Since out is a static variable of the system class hence we can write System.out . The data type of out must be a class(PrintStream class).

```
public class system------{
public static PrintStream out;
- - - - - - - - - - - - - - - -
- - - - - - - - - - - - - - - -
```

Since out is an object of PrintStream class, hence System.out return an object of PrintStream class by which we can invoke the println()/print()/printf() method because these method are defined and overloaded in PrintStream class.

Non- Static/Instance Variable	Static Variable

These variable should not be preceded by any static keyword Example:
class A {
  int x;
}
	These variables are preceded by static keyword.
	class A {
	  static int z;
	}

Memory is allocated for these variable whenever an object is created	Memory is allocated for these variable at the time of loading of the class.
Memory is allocated multiple time whenever a new object is created.	Memory is allocated for these variable only once in the program.

Non-static variable also known as instance variable while because memory is allocated whenever instance is created.	Memory is allocated at the time of loading of class so that these are also known as class variable.
Non-static variable are specific to an object	Static variable are common for every object that means there memory location can be **sharable** by every object reference or same class.
Non-static variable can access with object in other class.	Static variable can access with class reference or by object in other class.

Static block:--

- It is a block preceded by static keyword. It is used to initialize the static variable of the class.
- The program execution always starts from the static block then the main() will execute.
- It is executed before main method at the time of classloading.
- If we provide more than one static block in a class then they will execute sequentially.

```
class  Nomain1  {
        static  int  z;
        public  static  void  main(String  args[]){
            System.out.println("main method);
        }
        static  {
            z=123;
            System.out.println("val:"+z);}
        static  {
           System.out.println("2nd static block");
}};
```

If we do not provide the main() in a class then the program generate runtime exception. To avoid it we can write exit() of System class to terminate the execution .

```
class  Nomain2  {
        static  int  r;
        static  {
        r  =  444;
        System.out.println("The value  is : "+r);
        System.exit(0);
}}
```

Q : What is the difference between main() & static block ? Which one should be used in which case ?

```
class  A {
  A(){
    System.out.println("constructor of A");
  }
  static  {
      System.out.println("static block of A");
}};
```

```
class  B {
public static void main(String args[]) {
  A  a1  = new  A();
  A  a2  = new  A();
  A  a3  = new  A();
}};
```

Non static block : Declaring a block without any keyword in a class makes it a non-static block, it'll always execute before the constructor.

```
class  A {
  A(){
    System.out.println("constructor of A");
  }
```

```
class  B {
public static void main(String args[]) {
  A  a1 = new  A();
  A  a2 = new  A();
```

```
  {
     System.out.println("non-static block of A");
  }
 void test(){
       System.out.println("test in A");
}};
```

```
 a1.test();
 a2.test();
}};
```

Pass by value vs Pass by Reference:

```
class  A {
int  x,  y;
A(){  x=5;   y=9;  }
};
class  B{
static  void   alter(A  aa){
    aa.x=10;     aa.y=20;
}
static  void   change(int  m){
     m=10;
}
```

```
public static void main(String args[]){
  A   a1 = new  A();
  System.out.println("a1  before  :   "  +a1.x+  ",  "
+a1.y);
  alter(a1);   // pass by reference
  System.out.println("a1   after  :    "  +a1.x+  ",  "
+a1.y);

  int  p=123;
 System.out.println("p before :  " +p);
 change(p);   //pass by value
 System.out.println("p after :  " +p);
}};
```

Type of class:-- In java we have 3 types of classes known as Concrete class (CC), Abstract class (Ac) and Inner class (IC).

Concrete class (cc):- It is a class whose abject can be created independently is known as CC. it implements lower level of abstraction.

Abstract class (ac):- It is a class whose object cannot be created. It implements higher level of abstraction.

```
Syntax:--
[ access  specifier ]  abstract  class  classname {
              Data members.
              Constructors
              Concrete methods
              Abstract methods
         };
```

```
Ex  abstract Class A{
         int x,y:
         A() {  x  =  5;     y  =  7;  }
         A( int  p,  int  q )  {  x  =  p;    y  =  q;    }
         abstract void  diff();
         abstract int square(int z);
         void   add() {
         System.out.println("sum : " + ( x + y )  );
     }};
```

Abstract method :-- It is a method preceded by abstract keyword and that does not contain the method definition but contains method declaration.

Abstract class (abstract class) :--

- A class which is preceded by abstract keyword is known as abstract class.
- It may or may not contain any abstract method. Even if abstract class does not contain any abstract method still we cannot create its object.
- In case of concrete class only instance variables's are abstract because they hold different value for different objects. Therefore the abstract property of a class is define by the instance variables, hence a class known as abstract data type.
- In case of abstract class along with instance variables's some methods are abstract hence abstract class implements higher level of abstraction than concrete class.
- Both abstract class & concrete class help to achieve the code reuse and polymorphism during inheritance but abstract class forces the child class to define abstract method by overriding but CC can't force the child class to override a method. abstract class is like virtual class of c++.

INTERFACE:--It is a component whose object cannot be created. It implements highest level of abstraction. An interface, generally consisting of final variables & abstract methods.

SYNTAX:--	e.g. interface A{
[access specifier] interface interfacename {	int x = 23;
[public] [static] [final] variables	void add();
[public] [abstract] method	int diff();
};	};

By default the variables declared in an interface are public, static & final. Similarly the method in an interface are by default public & abstract. Hence no need to provide these keywords before the variables or methods defined in an interface. It is like pure virtual class of c++.

Concrete Class/abstract class	INTERFACE
1) We can define constant as well as variable. 2) We can define constructer and concrete method in Concrete Class & abstract class, as well as can declare abstract method in abstract class. 3) Java does not allow multiple inheritances from Concrete Class & abstract class. 4) abstract class is more abstract than Concrete Class because abstract class contains abstract method. 5) Concrete Class & abstract class allows us to achieve code reuse & polymorphism during inheritance	1) We can declare only constant. 2) We can declare only abstract method. 3) Java allows multiple inheritances from interfaces. 4) Compared to Concrete Class & abstract class, the interfaces has no concrete definition like. constructors or methods. Hence it has higher level of abstraction 5) Even if it has no concrete definition still it supports polymorphism during inheritance.

Interface VS Abstract class VS Concrete class

1. If we don't know anything about implementation just we have requirement specification (100% Abstraction) then we should go for interface.
2. If we are talking about implementation but not completely (partial implementation) then we should go for Abstract class.
3. If we are talking about implementation completely and ready to provide service then we should go for concrete class.

When should I use abstract classes and when should I use interfaces?

Use Interfaces when…

- You see that something in your design will change frequently.
- If various implementations only share method signatures then it is better to use Interfaces.
- You need some classes to use some methods which you don't want to be included in the class, then you go for the interface, which makes it easy to just implement and make use of the methods defined in the interface.

Use Abstract Class when…

- If various implementations are of the same kind and use common behavior or status then abstract class is better to use.
- When you want to provide a generalized form of abstraction and leave the implementation task with the inheriting subclass.
- Abstract classes are an excellent way to create planned inheritance hierarchies. They're also a good choice for nonleaf classes in class hierarchies.

INNER CLASS:- When a class B is defined within another class A then class B is known as inner class of A.& the class A known as outer class.

TYPE OF INNER CLASS:--

1) Non static inner class/Inner class /member inner class(MIC).

2) Static inner class(SIC).
3) Nested inner class(NIC)
3') Anonymous Inner Class

Member Inner Class (MIC):- When a class B is define at the same scope or level where other member of another class A is defined, then the class B is known as MIC of A. The MIC can access all the member of the outer class. The object of MIC cannot be created without the help of outer class object.

```
class A {
        int x , y ;
        A() { x = 7; y = 8; }
        A( int p, int q){ x = p; y = q; }
        class B{
           int z
           B() { z = 9; }
           B(int r) { z = r; }
           void show(){
        System.otu println(x + " : " + y + ": "+ z );
}} //EOF MIC B
void test( int k ) {
            B bb = new B(k);
            bb.show();
}} //end of outer class A
```

```
class C {
  public static void main(String args[]){
        A a1 = new A();
        A.B b1 = a1.new B();
        b1.show();
        A a2 = new A(32,24);
        a2.test(13);
}}
```

Staic Inner Class(SIC):-- When MIC is preceded by static key word then it become SIC. The SIC can access the static member o the outer class, but it cannot access non static members of the outer class. The object of SIC can be created by outer class name.

```
class A {
        int x;
        static int y ;
        A() { x = 7; y = 8; }
        A( int p, int q){ x = p; y = q; }
        static class B {
           int z
           B() { z = 9; }
           B(int r) { z = r; }
           void show(){
                System.otu println(y + ": "+ z );
}} //EOF SIC B
void test( int k ) {
            B bb = new B(k);
            bb.show();
}} //end of outer class A
```

```
class C {
  public static void main(String args[]){
        A.B b1 = new A.B();
        b1.show();
        A a2 = new A(32,24);
        a2.test(13);
}}
```

Nested inner class(NIC):-- When a class B is defined within the constructer or method for another class A then the class B is known as NIC.

The NIC can access those member of the outer class whatever is accessed to its enclosing constructer or method where it is defined. The NIC cannot access any local variable or parameter of its enclosing constructer or method where it is defined but it can access the final local variable or the final parameter of the method or constructer, this works upto JDK 1.7, but from JDK 1.8 the NIC can access all local variable or parameter of it's enclosing constructor or method. As a local variable/parameter of a method/constructer cannot be preceded by any access specifier or by static modifier similarly the NIC being local to a constructor/method cannot be preceded by any access specifier or by static modifier. The object of NIC can't be created outside of its enclosing constructer or method.

```
class A {
int x;
static int y ;
```

```
System.otu println(x+" : " + y + ": "+ z + " : " + m);
}} //EOF NIC B
    B bb = new B(k);
```

```
A() {   x = 7;   y = 8;  }
A( int  p,  int  q){  x = p;   y = q;
void test( int  k )  {
   final  int   m = 123;
   class  B {
      int   z
      B() {  z = 9;   }
      B(int r)  {   z = r;   }
      void show(){
    bb.show();
}}  //end of outer class A class   C  {

public static void main(String args[]){
   A    a1  =  new   A(32,24);
   a1.test(13);
}}
```

Q.WAP to create class A consisting of class B as MIC the class B should contain class C as MIC. Create one object class C in class D so that we can invoke a method of class C which will access the data members of other classes, class D should remain outside of class A.

Creating an Object by using Factory Method :--

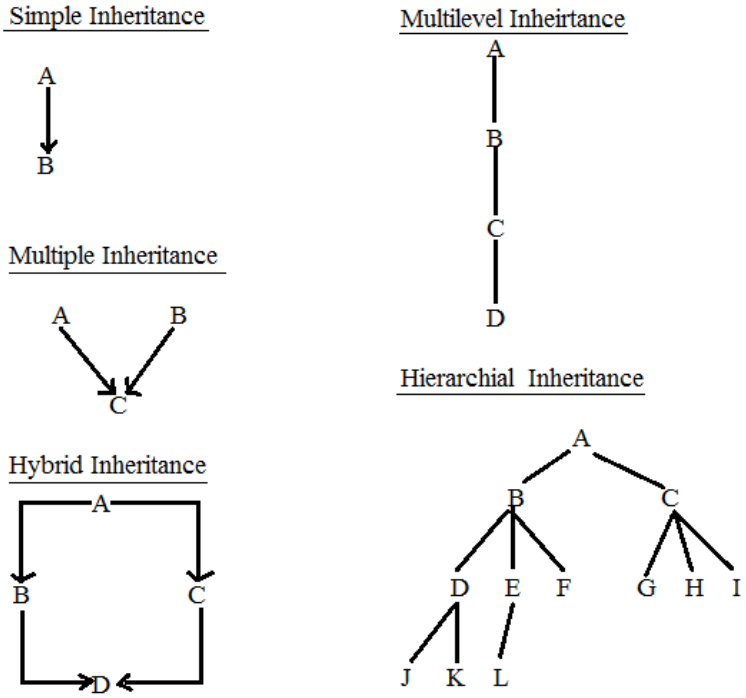
```
public final class FacMeth {
  /**  Static factory method returns an object of this class.     */
  public static FacMeth valueOf(String na, int ro) {
    return new FacMeth(na, ro);
  }

  /**  Caller cannot see this private constructor. The only way to build a ComplexNumber

is by calling the static  factory method.   */
  private FacMeth(String na, int ro) {
    name = na;
    roll = ro;
  }
  void  show(){
      System.out.println(name+", "+ roll);
 }
  private String  name;
  private int  roll;
};
class  Test {
public static void main(String stat[]) {
 FacMeth f1= FacMeth.valueOf("ram", 101);
 FacMeth f2= FacMeth.valueOf("sam", 102);
 FacMeth f3= FacMeth.valueOf("dam", 103);
 f1.show();
 f2.show();
 f3.show();
}}
```

Creating an Object by using newInstance() Method :--

```
class  A {
  void show() {
    System.out.println("show in A");
}};
class D {
 public static void main(String args[]) throws Exception {
    Class  c1 = Class.forName("A");
    A  a1 =(A) c1.newInstance();
    a1.show();
}};
```

INHERITANCE:--

- Using the properties & behavior of an existing component in a new component is known as inheritance.
- When a class 'B' will inherit from another class 'A' by **extends** keyword then the class 'B' is known as child class and the class A is known as parent class.
- All the property & behavior of the parent class will be inherit to the child class except the private properties & behavior of parent class. But the properties & behavior of the child class will never be inherited to the parent class.
- Java does not support multiple inheritances from classes, to avoid the ambiguity that occurs during multiple inheritances from classes as in c++, but it supports multiple inheritance from interfaces.
- We can define all 5 types of inheritances (simple, multiple, multilevel, hierarchical, hybrid) in java.



INHERITING FROM A CLASS:-- Java uses extends keyword to inherit from classes.

```
class A {
    void test() {
        System.out.println("test in A");
}}
class B extends A {
    void show() {
        System.out.println("show in B");
}}
```

```
class C{
    public static void main(String args[]) {
        A a1 = new A();
        B b1 = new B();
        a1.test();
        b1.test();
        b1.show();
        // a1.show();    /* parent has no idea
about method defined in child class*/
}}
```

CONSTRUCTER INHERANANCE RULE:--

Rule-1 : The default constructor of parent will be invoked implicitly from the 1st line of any constructer of the child class constructer. If we want to invoke the default constructor of the parent class from any constructer of child class then we have to write super() as the 1st executable statement in the child class constructer.

```
class A {
  int x;
  A() { x=5; }
}
```

```
class C{
public static void main(String args[]) {
  B b1 = new B();
```

```
class B extends A  {
  int y;
  B() {  y=10;  }
  B(int p) { y=p;  }
  void add()    {
     System.out.println("sum :"+(x+y));
}};
```

```
  B   b2  =   new   B(16);
  b1.add();
  b2.add();
}};
```

Rule -2 : The parameterized constructor of the parent class can be invoked explicitly from any constructer of the child class by writing super(parameters) as the 1st executable statement of any constructors of the child class, during this time the implicit call to default constructor will not be made.

```
class A{
         int x;
         A(int p){   x=p;    }   };
class B extends A  {
         int   y;
         B()   {    super(10);   y=8;   }
         B(int p; int q)   {    super(p);    y=q;   }
          void add(){
        System.out.println("sum:"+(x+y));       }
 } ;
```

```
class   C  {
   public static void main(String args[]){
         B   b1  =   new   B();
         B   b2  =   new   B(25,30);
         b1.add();
         b2.add();
  }
};
```

super <u>KEYWORD IN INHERITANCE :</u>

- The "super" keyword is used to refer the parent class object within the child class.
- If the name of the members of parent & child are similar, then to refer to parent class member within the child, we must use "super" keyword(i.e. use of super is mandatory in such case)
- When the name of the members of parent and child class are different, then we may or may not use "super" keyword to invoke the parent class member within the child class(i.e. use of super is optional in such case) .
- The super keyword is used only within the definition child class.

```
class    A  {
  int   x;
  A() {   x  =   5;  }
};
class  B  extends  A  {
  int    y;
  B() {   y  =   9;  }
  void  add()   {    System.out.println("sum : "  +  ( x + y));    }
};
class  C  extends  A  {
  int  x;
  C() {   x= 9;  }
  void  add1() {    System.out.println("sum : " + (x+x));   }
  void  add2() {    System.out.println("sum : " + (super.x+x) );
}};
class  D  {
  public static void main(String   args[]) {
     B   b1 = new    B();
     b1.add();
     C   c1 = new  C();
     c1.add1();
     c1.add2();
}};
```

<u>USE OF PRIVATE DATA MEMBERS : --</u>

```
class  A  {
  private  int  x, y;
  A() {   x= 5;  y = 9;  }
```

```
class  B  extends  A  {     };
class  C  {
  public static void main(String args[]) {
```

```
  void  add() {
        System.out.println("sum : " +(x+y));
}};
```

```
    B   b1 = new    B();
    b1.add();
    //System.out.println(b1.x);
}};
```

The private data member of the parent class will never be inherited to the child class. Hence a child class and it's object cannot use the private data members of the parent class, but the copy of private data members of the parent class will be implicitly available in the memory location of the object of the child class, which will be used for the parent class method invoked by the child class object.

REFERENCE CASTING: In case of inheritance the parent class is always considered as higher data type and the child class is always considered as lower data type.

```
class A {
    int  x , y ;
    A() {   x  =  5;      y  =  7;    }
    A(int  p,  int   q)  {   x  =  p;   y  =  q;     }
    void  add(){
        System.out.println("sum  :  " +  ( x + y)  );       }
};
class   B   extends    A {
    int  p,  q;
    B(){     p  =  11;     q  =  13;  }
    B(int   a, int b,  int  c, int  d ) {
        super(a, b);      p=c;          q=d;     }
    void   diff()  {   System.out.println("diff : " + (x+y-p- q ) );   }
};
class  C {
    public static void main(String  args[]){
       A    a1  =   new   A();
       B    b1  =   new   B(3, 4, 5, 6);
       a1.add();
       b1.add();
       b1.diff();
       b1  =  a1;
       B    b2  =  (B) a1;
       a1  =  b1;
       a1.add();
       a1.x  =  10;
       a1.y  =  12;
       // a1.diff();
       // a1.p  =  14;
       // a1.q  =  16;
       B   b3  =   (B) a1;
       b3.add();
       b3.diff();
}}
```

INHERITING FROM ABSTRACT CLASS :-

When a class will inherit from an abstract class, then the child class must override the abstract method of the parent abstract class, otherwise the compiler will not compile the source code, rather it will suggest to make the child class as an abstract class. The **reference** of parent abstract class can also hold the object of any child class(implicit reference casting).

```
abstract  class  A   {
   int  x, y;
   A() {   x=5;  y=  9;   }
   A(int  p,  int  q) {   x = p;  y = q;   }
```

```
class  C  {
   public static void main(String args[]) {
     B  b1 = new  B();
     b1.add();
```

```
   abstract  void   add();
   void  diff(int  p) {
       System.out.println("difference : " + (x+y-p));
}};
class  B extends A  {
   int  z ;
   B() {   z =11;   }
   B(int  a,  int  b, int  c) {  super( a, b);   z=c;   }
   void   add() {
     System.out.println("sum : " + (x+y+z));
   }
   void  mult() {
       System.out.println("mult : " + (z*y));
}};
```

```
      b1.diff(b1.z);
      b1.mult();
      B   b2 =  new   B(4, 5, 6  );
      A   a1 = b2;
      a1.add();
      a1.diff(b2.z);
    //   a1.mult();
}}
```

INHERITING FROM INTERFACES:-

When a class will inherit from an interface by **implements** keyword, then the child class must override the abstract method of the parent interface, otherwise the compiler will not compile the source code, rather it will suggest to make the child class as an abstract class.

When a class will inherit from more than one interface consisting of same method, then the child class will override the abstract method only once, similarly if a final variable(constant) is present in more than one interface with the same name, then the child class has to access the final variable with the interface name to avoid the ambiguity. The **reference** of parent interface will hold the object of any child class.

```
interface  A   {
   int  p =20;
   void   add();
}
interface  B   {
   int  p =40;
   void   add();
   int     diff(int  r);
}
class C  implements  A,  B  {
   int  x, y;
   C() {   x=5;  y=  9;   }
   C(int  p,  int  q) {   x = p;  y = q;   }
   public  void   add() {
     System.out.println("sum : " + (x+y+B.p));
   }
   public  int   diff(int  p) {
       return   x+ y- p –A.p;
  }
   void  mult() {
       System.out.println("mult : " + (x*y));
}};
```

```
class  D  {
   public static void main(String args[]) {
     C  c1 = new  C();
     c1.add();
     System.out.println("difference : " + c1.diff(5));
     c1.mult();
     C   c2 =  new   C(15, 16  );
     A   a1 = c2;
     a1.add();
//   System.out.println("difference : " + a1.diff(11));
//   a1.mult();
    B   b1 = c2;
    b1.add();
    System.out.println("difference : " + b1.diff(11));
//   b1.mult();
}}
```

Note :- A class can extends from another class at the same time it can implement from multiple interfaces.

Ex : class E extends A implements B, C, D { }

inheritance keyword		
Parent	**Keyword**	**Child**
Class	Extends	Class
Interface	Implements	Class
Interface	Extends	Interface
Class	impossible(na)	Interface

Enhancement from Java 1.8 : Method in an interface.

```
interface  A  {
   int   x=10;
   void  add(int  p, int q);
```

```
class  B implements   A{
  public void add(int  a, int b){
    System.out.println("total : "+(a+b));
```

```
  default void  show(){
    System.out.println("Java  1.8  allows  us  to
define a concrete function within an interface and
it should be preceded by default keyword, but the
access specifier of this method is public, it can be
used by child classs, & child class can override
it");
}};
```

```
}};
class  C  {
public static void main(String args[]) {
  B  b1= new   B();
  b1.add(5, 7);
  b1.show();
}};
```

POLYMERPHISM :- Same component providing multiple functionality is called polymorphism. It is divided into two parts: → Static linking /binding/ polymorphism (early binding)

→ Dynamic linking/ binding/ polymorphism (late binding).

STATIC LINKING:- The creation of link between method invocation and the method definition at the time of compilation is known as static linking. it is achieved by over loading. Java does not support operator overloading by a programmer but it has given overloaded operator such as +(sign/ concatenation/ addition operator), dot(. as member access operator as well as decimal point), but method overloading is supported for a programmer.

DYNAMIC LINKING :- The creation of link between the method invocation and the method definition at run time is known as dynamic linking. It can be achieved by overriding.

METHOD OVER LOADING:- Method with same name ,different signature if appears more than once in a class then it is known as over loaded method. Signature of a method defines the no of parameter and their data type.

Rules :

- The number of parameter should be different.
- If the number of parameter are equal then the data type must be different.
- If the number of parameters and their data type are equal then the sequence must be different.
- By changing only the return type of a method we cannot create over loaded method.

```
class Over{
  int x,y;
  Over() {  x =  5;   y  =  7;   }
  int add()  {
    System.out.print("1-")
    return x + y;
}
/* float add(){
     System.out.print("1-")
     return x + y;
}*/
int   add(int  a)  {
    System.out.print("2-");
    return x + y + a;
}
int add(int  a,   int  b)  {
   System.out.print("3-");
   return  x + y + a + b;
}
```

```
int add(int   a,  float   b)   {
   System.out.print("4-");
   return  (int)(x + y + a + b);
}
float  add( float  a,   int   b)   {
   System.out.print("5-");
  return   x  +  y  +  a  +  b;
}
float add( float   a,  float   b)  {
   System.out.print("6-");
   return  x  +  y  +  a  +  b;
}};
class D  {
public static void main(String args[]){
Over b1=new Over();
System.out.println(b1.add());
System.out.println(b1.add(5));
System.out.println(b1.add(4,3));
System.out.println(b1.add(4,3.3f));
System.out.println(b1.add(4.4f,3));
System.out.println(b1.add(4.4f,3.3f));
}};
```

METHOD OVERRIDING:- Method with same name, same signature& same type if appears in parent as well as in the child class then it is known as overridden method.

Rules:-

- A method can be overridden at most once in a child class, otherwise it will be considered as over loaded method.

- Method overridden by chain or subclasses(multilevel inheritance) cannot invoke each other directly by "super" keyword, but they can invoke the parent class method by "super".
- The access specifier of the overridden method in child class must remain the same as that of the parent class method or it may be less restrictive than the access specifier of the parent class method.
- The exception "throws" by the child class method may belong to any unchecked exception category but the overridden method in child cannot "throws" any checked exception if the parent has not "throws" any checked exception.

Ex-1 : **Rule-1**

```
class  A  {
  void  show()  {
     System.out.println("show in A");
}}
class  B  extends A {
  void  show()  {
     System.out.println("overridden show in B");
 }
 void  show(int  x)  {
    System.out.println("overloaded  show  in
B"+x);
 }};
```

```
class  C{
  public static void main(String args[]){
     A  a1 = new  A();
     B  b1 = new  B();
     a1.show();
     b1.show(33);
     b1.show();
     a1  =  b1;
     a1.show();
     // a1.show(33);
}};
```

Ex-2 : **Rule-2**

```
class  A {
public  void  show(){
   System.out.println("show in A");
}};
class  B  extends  A {
public  void  show(){
   System.out.println("overriden show in B");
}};
class  C  extends  B {
 void  show(){
   super.show();
   System.out.println("overriden show in C");
}};
```

```
class  D  extends  C {
 void  show(){
   super.show();
   System.out.println("overriden show in D");
}};
class  E {
 public static void main(String args[]) {
   D  d1= new  D();
   d1.show();
}};
```

Ex-3 : **Overidding example**

```
class  A {
 void  show(){
    System.out.println("show in A");
}};
class  B  extends  A {
 void  show(){
    System.out.println("show in B");
}};
class  C  extends  B {
 void  show(){
    System.out.println("show in C");
}};
class  D  extends  C {
 void  show(){
    System.out.println("show in D");
}};
```

```
class  E {
public static void main(String args[]) {
 A  ob;
 int  x= Integer.parseInt(args[0]);
 if(x>0 && x<100)
    ob= new  A();
 else if(x>100 && x<500)
    ob= new  B();  //implicit ref. casting
 else if(x>500 && x<1000)
    ob= new  C();  //implicit ref. casting
 else
    ob= new  D(); //implicit ref. casting
 ob.show();
}};
```

Ex-4 : **Overriding a static method** (A static method can be redefined in the child class, but still it is not considered as overridden method)

```
class  A {
```

```
class  C {
```

```
static void show(){
   System.out.println("show in A");
}};
class B extends A {
static void show(){
   System.out.println("show in B");
}};
```

```
public static void main(String args[]) {
   A a1= new A();
   B b1= new B();
   a1.show();
   b1.show();
   a1 = b1; //implicit reference casting
   a1.show();
}};
```

Ex-5 : **Overridding a private method** Upto java 1.7 does not allows us to redefine the private method of parent in child class, but java 1.8 allows us to define private method of parent in child class, still it is not considered as overriden method, rather they are considered as personal/separate method

```
class A {
  private void test(){
    System.out.println("personal test in A");
  }
  void demo(){   test(); }
};
class B extends A {
  private void test(){
    System.out.println("personal test in B");
}
void demo(){   test(); }
};
```

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

Ex-6 : **Covariant return type** : covariant return type of an overriding method means: the return type of overriding method in child class can be any subclass to that of the return type of overriden method of parent class.

```
class A {  String z="Parent"; }
class B extends A {  String z="Child"; }
class C {
  A getObject(){
    return new A();
}};
class D extends C {
  B getObject(){
    return new B();
}};
```

```
class E {
public static void main(String args[]) {
  C c1= new C();
  D d1= new D();
  A a1 = c1.getObject();
  System.out.println(a1.z);
  B b1 = d1.getObject();
  System.out.println(b1.z);
  c1 = d1;
  B b2= (B) c1.getObject();
  System.out.println(b2.z);
}};
```

What are the differences between method overloading and method overriding?

	Overloaded Method	Overridden Method
Arguments	Must change	Must not change
Return type	Can change	Can't change except for covariant returns
Exceptions	Can change	Can reduce or eliminate. Must not throw new or broader checked exceptions
Access specifiers	Can change	Must not make more restrictive (can be less restrictive)
Invocation	Reference type determines which overloaded version is selected. Happens at compile time.	Object type determines which method is selected. Happens at runtime.

Note :-

- A final class cannot be inherited by any child class.
- A final method cannot be overridden, but a child class can use the final method.

- A static method can be redefined in the child class but it is never considered as overridden method.

Can we create an object for an interface? Yes, it is always necessary to create an object implementation for an interface. Interfaces cannot be instantiated in their own right, so you must write a class that implements the interface and fulfill all the methods defined in it.

\Ex-1: **Defining a nested inner class in a method by inheriting from an interface.**

```
interface  A {
   void  add(int p,  int  q);
};
class   B {
static  A   getValue(){
  class  Z   implements  A {
      public  void   add(int  a, int  b){
           System.out.println("Sum : " +(a+b));
  }};//eof NIC Z
  Z  z1= new  Z();
  return  z1;
}};
```

```
class  C {
public  static void  main(String args[]){
   A   a1 =   B.getValue();
   a1.add(13, 11);
}};
```

Ex-2 : **Defining interface within a class**

```
class   Sarathi  {
   interface  Kirtan {
        void  bhajan();
  };
};
class  Chela  implements  Sarathi.Kirtan {
   public void bhajan(){
    System.out.println("baba  chakki  pissing,  chela
bampha neiing");
}};
```

```
class Jail {
  public static void main(String args[]) {
    Chela c1 = new  Chela();
    c1.bhajan();
 }};
```

Ex-3 : **Define a class in an interface**

```
interface  A  {
   void  show( );
    //by default class in an interface is public and
static
    class   B {
       int   x, y;
        B(){  x=5;  y=7;  }
        void  add(){
             System.out.println("total : "+(x+y));
    }};  //end of inner class
};
```

```
class  C  implements   A{
  public void show(){
    System.out.println("show of  A in C");
}};
class  D  {
public static void main(String args[]) {
  C.B   b1 = new  C.B();
   b1.add();
  C  c1= new   C();
   c1.show();
}};
```

Ex-3 : Using inner interface in a inner class & outer class

```
class  A {
  int x;
  A(){  x=5;  }
  A(int  p){  x=p;  }
  interface  B {
    void   show();
  };
  class   D  implements  B{
   public void  show(){
      System.out.println("show of  B in D");
  }};
};
```

```
class  C  implements  A.B {
   public void  show(){
       System.out.println("show of  B in C");
}};
class  E {
  public static void main(String args[]) {
    C   c1= new  C();
    c1.show();
    A.D   d1 = new  A().new   D();
    d1.show();
}};
```

Moral : **We can define anything within any other thing.**

Enhancement in Java 1.8: Using lambda expression (It will work only when the interface contains one method)

Ex-1 :	**Ex:** Using anonymous inner class & lambda

```
interface    A {
   void   disp();
};
class  C {
public  static void  main(String args[]){
  class  B   implements  A{
     public void   disp(){
          System.out.println("hello");
   }};    //B is Nested Inner Class
 B  b1 = new  B();
 b1.disp();

A   a1 = new   A()  {
     public void   disp() {
          System.out.println("hellooooooo");
 }};     //eof anonymous inner class
a1.disp();

  A  a2= () -> {
          System.out.println("hi  hello");
 };
  a2.disp();
}};
```

expression to inherit from interface & to override abstract method.

```
interface    A {
   int  add(int  p, int  q);
};
 class  C {
public  static void  main(String args[]){
   class   X   implements   A {
       public void  add(int a, int b){
            return   (a+b);
   }}; //eof  nested inner class
   X   x1 = new   X();
   x1.add(12, 15);

   A    x2= new   A()  {
       public int  add(int a, int b){
            return  (a+b);
   }}; //anonymous inner class
   System.out.println("sum : " +x2.add(12, 19));

  A  x3= (int a, int b) -> {  return (a+b);   };
  System.out.println("sum : " +x3.add(17,13));
}};
```

What is a marker interface ? Marker interfaces are those which do not declare any required methods, but signify their compatibility with certain operations. The **Serializable, Cloneable, Remote, EventListener** interface are typical marker interfaces. These do not contain any methods, but classes must implement this interface in order to be serialized and de-serialized.

IS-A Relationship: IS-A is a way of saying : This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
Ex:     public class Animal{          };
         public class Mammal extends Animal{    };
         public class Reptile extends Animal{     };
         public class Dog extends Mammal{     };
```

Now based on the above example, In Object Oriented terms following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are sub classes of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now if we consider the IS-A relationship we can say:

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence : Dog IS-A Animal as well

With use of the extends keyword the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

HAS-A relationship: These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

```
Ex:    public class Vehicle{      };
       public class Speed{       };
       public class Van extends Vehicle{
            private Speed sp;
       }
```

This shows that class Van HAS-A Speed. By having a separate class for Speed we do not have to put the entire code that belongs to speed inside the Van class., which makes it possible to reuse the Speed class in multiple applications.

In Object Oriented feature the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. SO basically what happens is the users would ask the Van class to do a certain action and the Vann class will either do the work by itself or ask another class to perform the action.

A very important fact to remember is that Java only supports only single inheritance. This means that a class cannot extend more than one class. Therefore following is illegal:

```
public class Dog extends Animal, Mammal{      };
```

However a class can implement one or more interfaces. This has made Java get rid of the impossibility of multiple inheritance

Association is a relationship where all objects have their own lifecycle and there is no owner. Let's take an example of Teacher and Student. Multiple students can associate with single teacher and single student can associate with multiple teachers, but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.

Aggregation is a specialized form of Association where all objects have their own lifecycle, but there is ownership and child objects can not belong to another parent object. Let's take an example of Department and teacher. A single teacher can not belong to multiple departments, but if we delete the department teacher object will not be destroyed. We can think about it as a "has-a" relationship.

Composition is again specialized form of Aggregation and we can call this as a "death" relationship. It is a strong type of Aggregation. Child object does not have its lifecycle and if parent object is deleted, all child objects will also be deleted.

Let's take again an example of relationship between House and Rooms. House can contain multiple rooms - there is no independent life of room and any room can not belong to two different houses. If we delete the house - room will automatically be deleted. Let's take another example relationship between Questions and Options. Single questions can have multiple options and option can not belong to multiple questions. If we delete questions options will automatically be deleted.

Difference between aggregation and composition

Composition is more restrictive. When there is a composition between two objects, the composed object cannot exist without the other object. This restriction is not there in aggregation. Though one object can contain the other object, there is no condition that the composed object must exist. The existence of the composed object is entirely optional. In both aggregation and composition, direction is must. The direction specifies, which object contains the other object.

Example: A Library contains students and books. Relationship between library and student is aggregation. Relationship between library and book is composition. A student can exist without a library and therefore it is aggregation. A book cannot exist without a library and therefore its a composition. For easy understanding I am picking this example. Don't go deeper into example and justify relationships!

Abstraction is specifying the framework and hiding the implementation level information. Correctness will be built on top of the abstraction. It gives you a blueprint to follow to while implementing the details. Abstraction reduces the complexity by hiding low level details.

Example : A wire frame model of a car.

Generalization uses a "is-a" relationship from a specialization class. Common structure and behavior are used from the specialization to generalized class. It is simply inheritance in a broader view.

Example: Consider a class named Person. A Student or a Faculty is a person. Therfore, the relationship between Student and Person, Faculty and Person is generalization.

Realization is a relationship between the blueprint class and object containing its respective implementation level details. The object is said to realize the blueprint class. We can think realization relationship between interface and implematation class.

Cohesion

The concept of **cohesion** shows to what degree a program's or a component's various tasks and responsibilities are **related to one another**, i.e. how much a program is focused on solving a single problem. Cohesion is divided into strong cohesion and weak cohesion.

Strong Cohesion : Strong cohesion indicates that the responsibilities and tasks of a piece of code (a method, class, component or a program) are **related to one another** and intended to **solve a common problem**. This is something we must always aim for. Strong cohesion is a typical characteristic of high-quality software.

Strong Cohesion in a Class : Strong cohesion in a class indicates that the class **defines only one entity**. As an entity can have many roles (Peter is a soldier, husband and a taxpayer). Each of these roles is defined in the same class. Strong cohesion indicates that the class solves only one task, one problem, and not many at the same time.

A class, which does **many things at the same time**, is difficult to understand and maintain. Consider a class, which implements a hash table, provides functions for printing, sending an e-mail and working with trigonometric functions all at once. How do we name such a class? If we find it difficult to answer this question, this means that we have failed to achieve **strong cohesion** and have to separate the class into several smaller classes, each solving a single task.

As an **example** of strong cohesion we can point out the **System.Math** class. It performs **a single task**: it provides mathematical calculations and constants:

- Sin(), Cos(), Asin()
- Sqrt(), Pow(), Exp()
- Math.PI, Math.E

Weak Cohesion

Weak cohesion is observed along with **methods, which perform several unrelated tasks**. Such methods take several different groups of parameters, in order to perform different tasks. Sometimes, this requires logically unrelated data to be unified for the sake of such methods. Weak cohesion is harmful and must be avoided!

Here is a sample class with weak cohesion:

```
public class Magic {
   public void printDocument(Document d) { … }
   public void sendEmail(String recipient, String subject, String text) { … }
   public void calculateDistanceBetweenPoints(int x1, int y1, int x2, int y2) { … }
}
```

Coupling

Coupling mostly describes the extent to which components / classes **depend on one another**. It is broken down into **loose coupling** and **tight coupling**. Loose coupling usually correlates with strong cohesion and vice versa.

Loose coupling is defined by a piece of code's (program / class / component) communication with other code through **clearly defined interfaces** (contracts). A change in the implementation of a loosely coupled component doesn't reflect on the others it communicates with. When you write source code, you **must not rely on inner characteristics** of components (specific behavior that is not described by interfaces).

The contract has to be maximally simplified and define only the required behavior for this component's work by hiding all unnecessary details. Loose coupling is a code characteristic you should aim for. It is one of the characteristics of high-quality programming code.

Here is an **example of loose coupling** between classes and methods:

```
class Report {
   public boolean loadFromFile(String fileName) { … }
   public boolean saveToFile(String fileName) { … }
}
class Printer {
   public static int print(Report  re) { … }
}
class Example {
   public static void main(String args[])    {
      Report r1 = new Report();
      r1.loadFromFile("DailyReport.xml");
      Printer.print(r1);
   }
}
```

In this example, **none of the methods depend on the others**. The methods rely only on some of the parameters, which are passed to them. Should we need one of the methods in a next project, we could easily take it out and reuse it.

Tight Coupling: We achieve tight coupling when there are many input parameters and output parameters; when we use undocumented (in the contract) characteristics of another component (for example, a dependency on static fields in another class); and when we use many of the so called control parameters that indicate behavior with actual data. **Tight coupling** between two or more methods, classes or components means that **they cannot work independently of one another** and that a change in one of them will also affect the rest. This leads to difficult to read code and big problems with its maintenance.

Tight Coupling – Example

Here is an **example of tight coupling** between classes and methods:

```
class MathParams{
   public static double operand;
   public static double result;
}
class MathUtil {
   public static void sqrt()   {
      MathParams.result = calcSqrt(MathParams.operand);
   }
}
class SpaceShuttle{
   public static void main(String args[])   {
      MathParams.operand = 64;
      MathUtil.sqrt();
      System.out.println(MathParams.result);
   }
}
```

STRUCTURE IN c :

```
#include<stdio.h>
#include<string.h>
struct stud {
        char name[20];
```

```
        int roll;
        float m1,m2,m3,avg;
};
int main(){
        struct stud s1,s2,s3;
        strcpy(s1.name,"Ram");
        s1.roll=101;
        s1.m1=99.9;
        s1.m2=99.9;
        s1.m3=99.9;

        strcpy(s2.name,"Sam");
        s2.roll=102;
        s2.m1=98.9;
        s2.m2=98.9;
        s2.m3=98.9;

        strcpy(s3.name,"Dam");
        s3.roll=103;
        s3.m1=96.9;
        s3.m2=96.9;
        s3.m3=96.9;

        printf("%s %d %f\n",s1.name, s1.roll, (s1.m1+s1.m2+s1.m3)/3);
        printf("%s %d %f\n",s2.name, s2.roll, (s2.m1+s2.m2+s2.m3)/3);
        printf("%s %d %f\n",s3.name, s3.roll, (s3.m1+s3.m2+s3.m3)/3);
        getchar();
        return 0;
}
```

STRUCTURE IN C++ :

```
#include<bits/stdc++.h>
using namespace std;
struct student {
        char name[20];
        int roll;
        float m1,m2,m3,avg;
        void set(){
                strcpy(name,"Rama");
                roll=101;
                m1=99.9;
                m2=99.9;
                m3=99.9;
        }
        void set(char n[],int r, float a, float b, float c){
                strcpy(name,n);
                roll=r;
                m1=a;
                m2=b;
                m3=c;
        }
        void print(){
                 cout<<name <<", " <<roll <<", " << (m1+m2+m3)/3 <<endl;
        }
};
```

```
int main(){
	struct student s1,s2,s3;
	s1.set();
	s2.set("Sama",102,98.9,98.9,98.9);
	s3.set("Dama",103,96.9,96.9,96.9);
	s1.print();
	s2.print();
	s3.print();

	getchar();
	return 0;
}
```

CLASS IN C++ :

```
using namespace std;
#include<iostream.h>
#include<string.h>
//struct stud {
class stud {
	public:
	char name[20];
	int roll;
	float m1,m2,m3,avg;
	//similar to void set() but it is called default constructor
	stud(){
		strcpy(name,"Ram");
		roll=101;
		m1=99.9;
		m2=99.9;
		m3=99.9;
	}

	//similar to void set(char n[],int r, float a, float b, float c) but it is called parameterised constructor
	stud(char n[],int r, float a, float b, float c){
		strcpy(name,n);
		roll=r;
		m1=a;
		m2=b;
		m3=c;
	}
	void print(){
		 cout<< name <<", " <<roll <<", " << (m1+m2+m3)/3 <<endl;
	}
};
int main(){
	stud s1; //s1 is an object and during this time dafult constructor is invoked implicitely
	stud s2("Sam",102,98.9,98.9,98.9); //s2 & s3 are objects created by parameterised constructor
	stud s3("Dam",103,96.9,96.9,96.9);
	s1.print();
	s2.print();
	s3.print();
	getchar();
	return 0;
}
```