Java Inheritance

Java Inheritance

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- In inheritance child class reuse methods and fields of the parent class and can add new methods and fields in child class
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.
- Why use inheritance in java
  - For Method Overriding (runtime polymorphism can be achieved).
  - For Code Reusability.

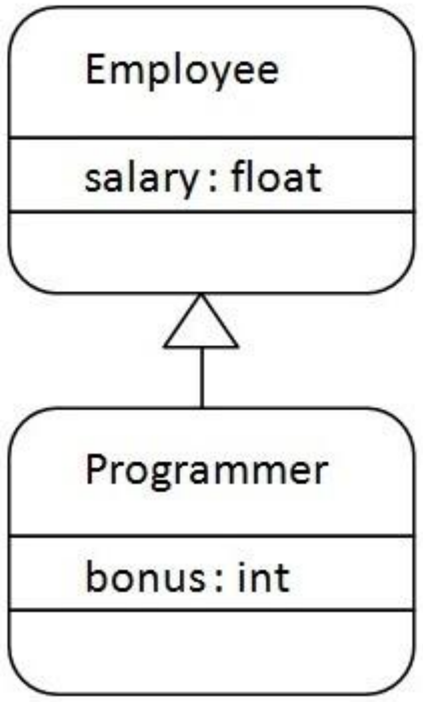
Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class

- The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

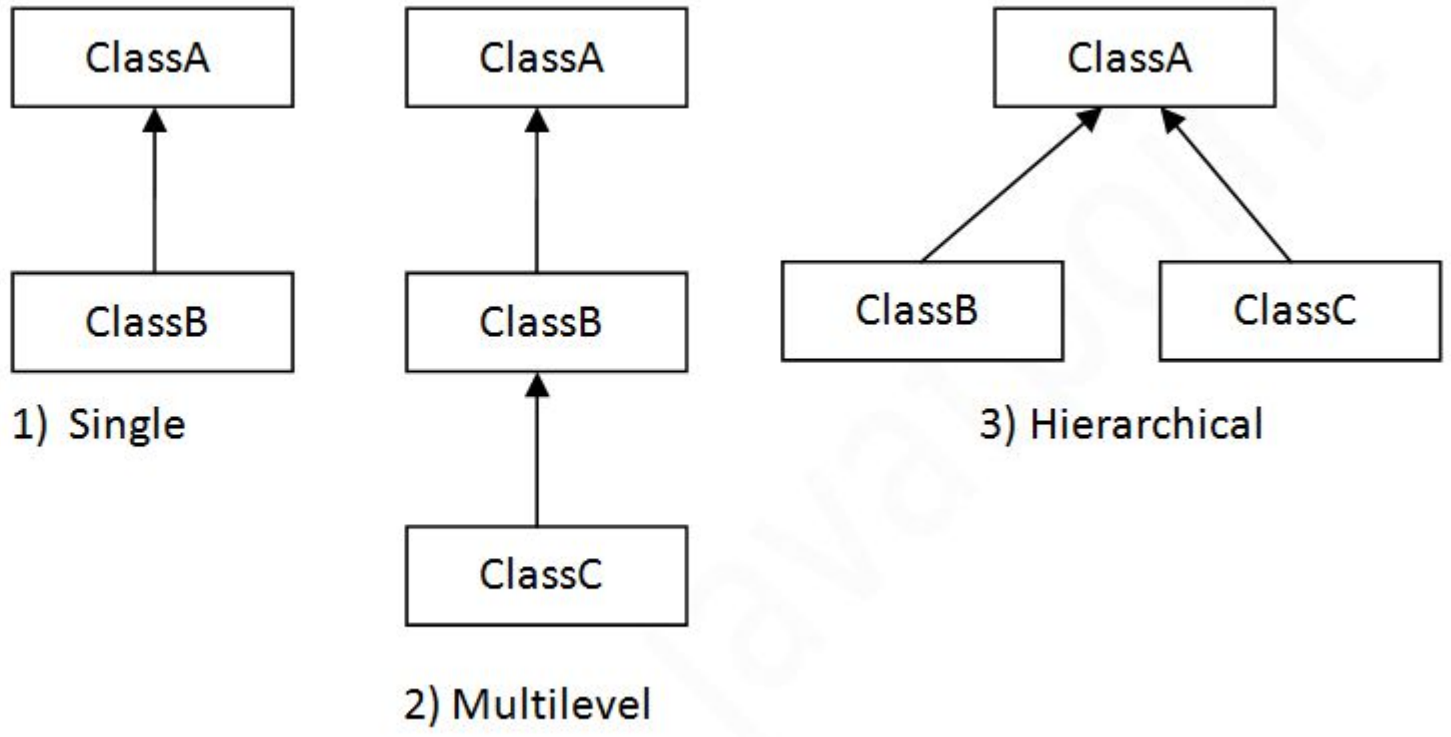
- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

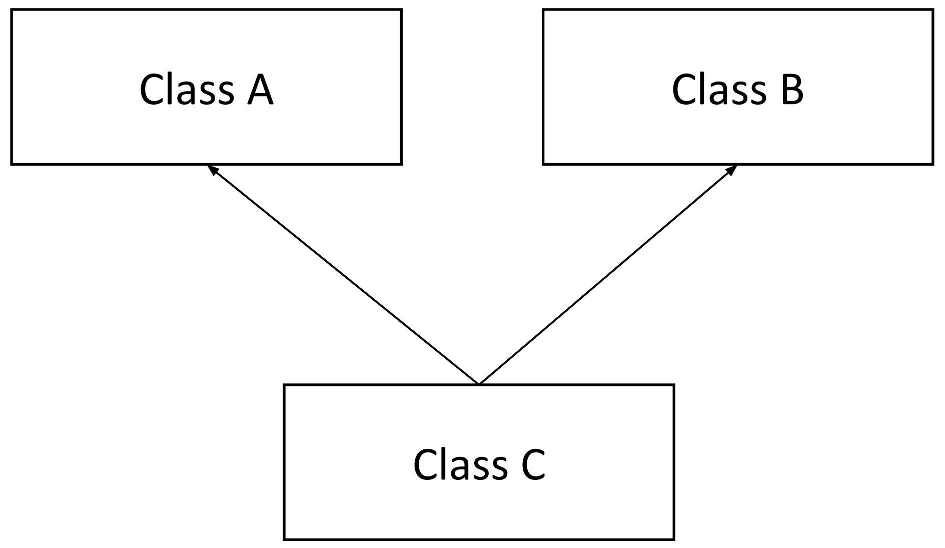


Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```java
class Employee{
float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
}
public class Inhert{
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Types of inheritance in java





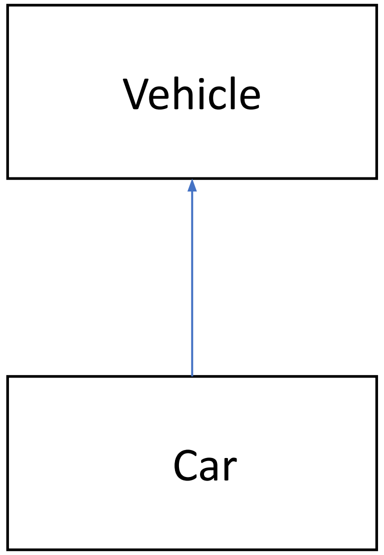
When one class inherits multiple classes, it is known as multiple inheritance.
Multiple inheritance is not supported in Java through class.

- Add one derived class for Car and Bike class (i10,bajaj)
- In i10 add one method i10method()// print –this is i10 car
- In bajaj add one method bajajmethod()// print –this is bajaj bike
- In main class main method create objects of i10 and bajaj class
- For bajaj-Access methods –bajajmethod(),getdata(,),printdata()
- For i10-Access methods –i10method(),getdata(,),printdata()
- Create one vehicle class (base class)

with one instance variable (color), add method-vehiclemethod()-print –this is base class method

- Single Inheritance Example
- When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```


Vehicle
Car

- Multilevel Inheritance Example
- When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

- Hierarchical Inheritance Example
- When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

```java
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Super Keyword in Java

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by **super reference variable.**

- Usage of Java super Keyword

1. super can be used to refer immediate parent class **instance variable.**
2. super can be used to invoke immediate **parent class method.**
3. super() can be used to invoke immediate **parent class constructor.**

- **Aim:** Write a program to demonstrate the use of method overriding.

Create class 'UG_student' inherited from 'Student' class with instance variables as branch and s_class. Override method print_data() to print the values of instance variables.

- **Post lab Question:**

1. Create a class named 'Shape' with two methods

   a. display() to print "Length and Breadth of shape".

   b. calculate_area() to print the area of the shape.

- Then create two other classes named 'Rectangle', 'Triangle' inheriting the Shape class. Both the classes should override the display() and calculate_area() method to print Length and Breadth of the respective shape and area of respective shape respectively.

- Now create a subclass 'Square' of 'Rectangle' and override the methods of superclass. Now call the methods display() and calculate_area() of each Rectangle, Triangle and Square by creating object of each of the class.

- Create Student class add instance variable –branch
- Create UG_student class add instance variable – s_class
- Add get_data method in Student class [void get_data(String branch)]
- Add get_data method in UG_student class

  [void get_data(String br,String s_class)]
- Add main class StudInher ,
- Add main method
- Create derived class object
- Add print_data() method in Student class
- Add print_data() method in UG_student class
- In main method – call get_data method and print_data method

Final Keyword in Java

- The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

- The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

Final Keyword in Java

1) Java final variable

- If you make any variable as final, you cannot change the value of final variable(It will be constant).

2) Java final method

- If you make any method as final, you cannot override it.

3) Java final class

- If you make any class as final, you cannot extend it.

Blank or uninitialized final variable

```
class Bike10{
  final int speedlimit;//blank final variable

  Bike10(){
  speedlimit=70;
  System.out.println(speedlimit);
  }

  public static void main(String args[]){
    new Bike10();
 }
}
```

Final class

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Abstract class in Java and Abstraction in Java

- A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Abstract class in Java

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example of abstract class

- **abstract class** A{}
- Abstract Method in Java
- A method which is declared as abstract and does not have implementation is known as an abstract method.
- **Example of abstract method**
- **abstract void** printStatus();//no method body and abstract

```
abstract class Bike{
  abstract void run();

}
class Honda4 extends Bike{
void run()
{
  System.out.println("running safely");
}
public static void main(String args[]){
 Bike obj = new Honda4();//
 obj.run();
}
}
```

- creates an instance of Honda4, but references it using the type Bike.
- When obj.run() is called, Java uses **dynamic dispatch** to execute Honda's run() method at runtime.

- Create abstract class Fruit
- Add abstract taste() method
- Create Mango class-derived class
- Add taste() method in mango class
- Create main class FruitClass
- Add main method
- Create object of derived class
- Fruit f=new Mango();
- Access taste() method
- Add derived class –Pear
- Add taste() method in Pear class

```java
abstract class Shape
{
abstract void draw();
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}

class TestAbstraction1{
public static void main(String args[]){
Shape c=new Circle1();
Shape r=new Circle1();
c.draw();
r.draw();
}  }
```

Why multiple inheritance is not supported in java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
- Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

Interface in Java

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.

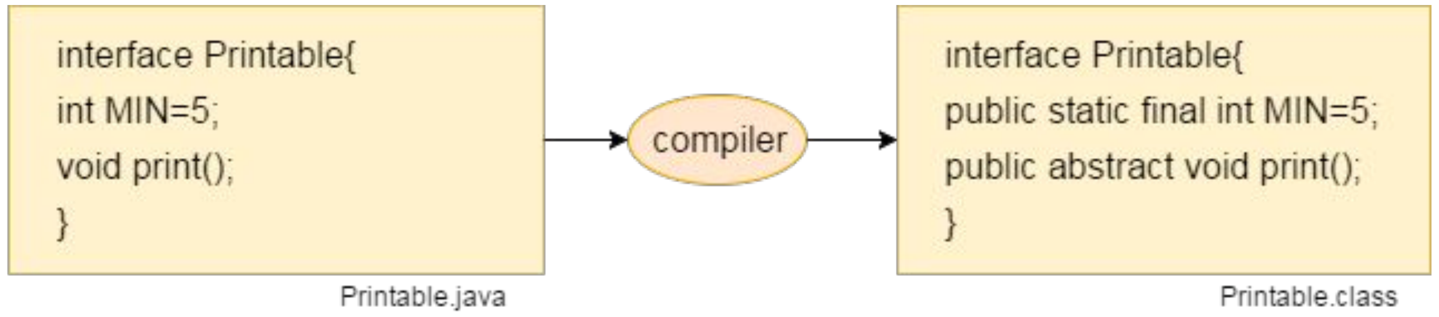
Why use Java Interface

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.

Interface Syntax

```
interface <interface_name>
{
    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

Internal addition by the compiler



```
interface Printable{
int MIN=5;
void print();
}
```

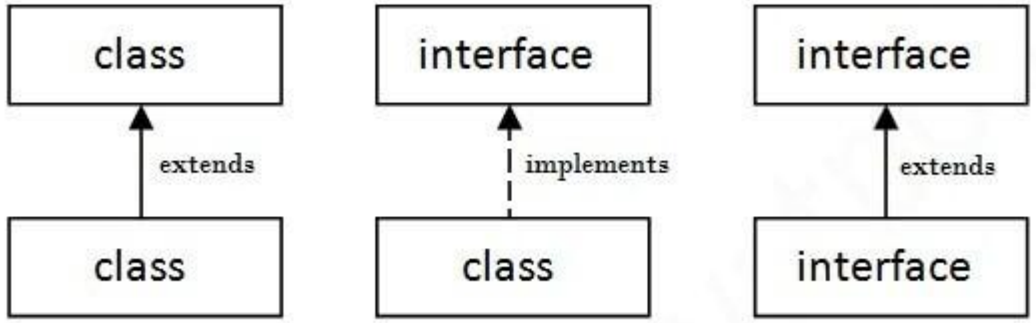
```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

- The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

- In other words, Interface fields are public, static and final by default, and the methods are public and abstract.

The relationship between classes and interfaces

- As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example

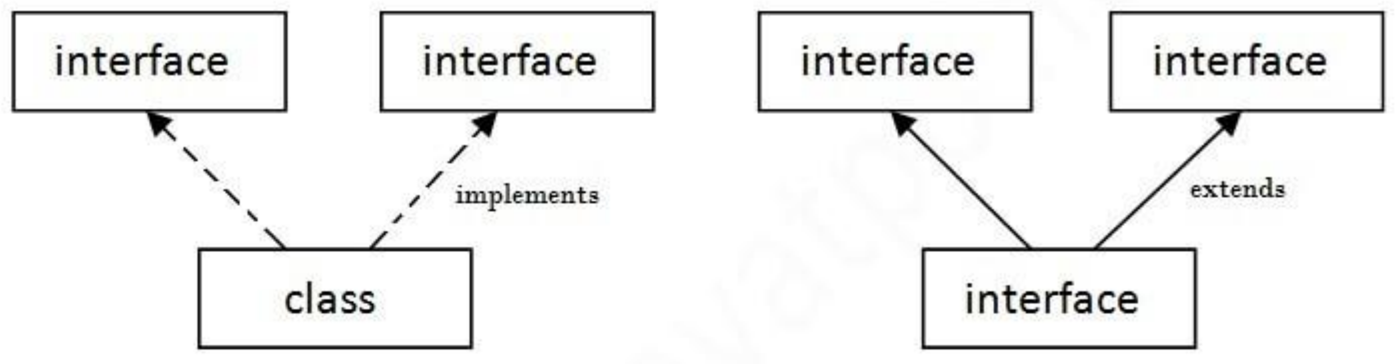
```
interface printable{
void print();  // public
}
class A6 implements printable{
public void print()
{System.out.println("Hello");
}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

- Create interface Bank
- Add abstract method -float rateOfInterect();
- Add default method void print() // non abstract method for interface bank
- Create derived class SBI for interface Bank
- add method public float rateOfInterest(){ return 9.15f;}
- Create derived class BOI for interface Bank
- add method public float rateOfInterest(){ return 10.2f;}
- Create main class –BankROI
- Add main method
- Create objects of SBI and BOI
- Access rateOfInterest()
- float a=s.rateOfInterest();
- Access default method

Multiple inheritance in Java

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```java
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Multiple inheritance in Java by interface

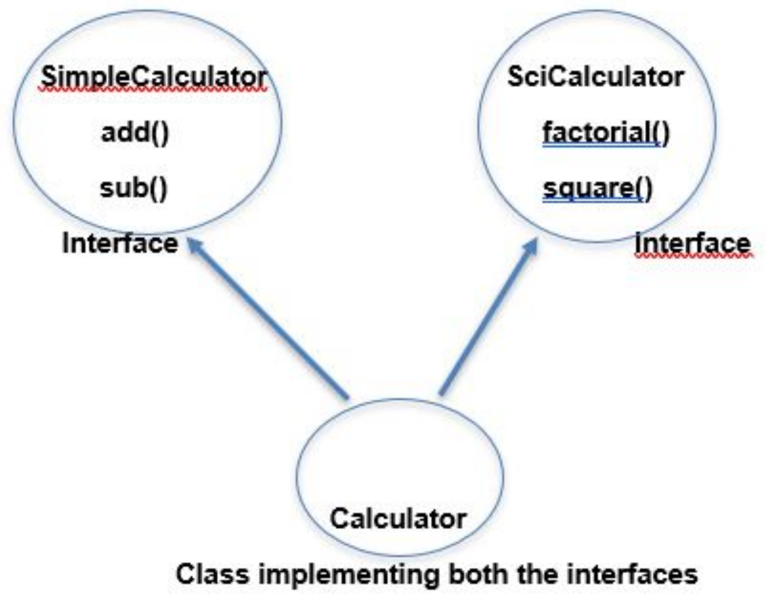
```
interface Printable{
void print();
}
interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

```java
interface Sayable{
    // Default method
    default void say(){
        System.out.println("Hello, this is default method");
    }
    // Abstract method
    void sayMore(String msg);
}
public class DefaultMethods implements Sayable{
    public void sayMore(String msg){        // implementing abstract method
        System.out.println(msg);
    }
    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        dm.say();   // calling default method
        dm.sayMore("Work is worship");  // calling abstract method

    }
}
```

- **Aim:** Write a program to Implement Multiple Inheritance using interface



- Create two interfaces ‘SimpleCalculator’ and ‘SciCalculator’ as shown in the above figure. Write a class ‘Calculator’ that implemets both the interfaces. Create object of class ‘Calculator’ and call the methods add(), sub(), factorial() and square() and display the result of each method.

- Create 2 interfaces A & B
- Add abstract method void show() method in class A& B
- Create derived class C
- Implement show() method in class C
- Print – “multiple inheritance using interfaces”.
- Create main class MultipleInhr
- Add main method
- Create object of class C
- Invoke show() method

Member access specifiers

- An access control defines a boundary to a member of a class. They are also known as access specifiers.
- There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
3. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.
4. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Varargs in java

- The varargs allows the method to accept zero or muliple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.
- Advantage of Varargs:
- We don't have to provide overloaded methods so less code.

- Ob.display(1);
- Ob.display(2,4);
- Ob.display(3,5,6);

- Ob.display(1,3,8,9,5);

- Ab.display(……..)

- Syntax of varargs:
- The varargs uses ellipsis(...) identifies a varible number of arguments. Syntax is as follows:

```
   return_type method_name(data_type ... variableName){}
e.g.  void print(int ... a)//internally varargs converted to arrays i.e.
                                                          int[] a
 ob.print(4,5);
```

Using index we can differentiate values

Rule for varargs:

- While using the varargs, you must follow some rules otherwise program code won't compile.

- The rules are as follows:
  - There can be only one variable argument in the method.

```
void method(String... a, int... b){}//Compile time error
void method(int... a, String b){}//Compile time error
```

Nested class

- In Java, it is possible to define a class within another class, such classes are known as *nested* classes. They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation, and creates more readable and maintainable code.
- The scope of a nested class is bounded by the scope of its enclosing class.
- Class A

```
{
//instance variable, method, class
}
```

Nested class syntax

```
class OuterClass

{

...

    class NestedClass

    {

        ...

    }

}
```

- Method : Call by object
- Syntax:
- Return_type method_name(Class_name obj)
- Void display(Class obj)
- Call the method
- Display(a) // where a is an object

- Object as a return type
- Class_name method_name();
- Class_name obj1_name= obj.method_name();
- Student cal_per(1289)
- {
- return obj name;
- }