Unit – III

Inheritance and Interface

Inheritance

- 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. When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Inheritance represents the **IS-A** relationship which is also known as a parent-child relationship.

- **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.

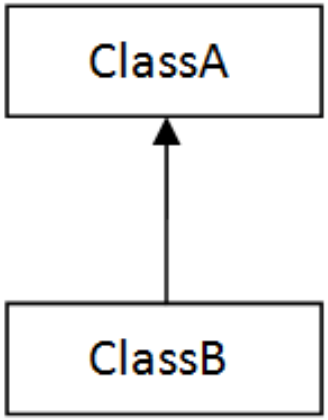
Inheritance Basics

- General class -> Specific class
- Super class -> Sub class

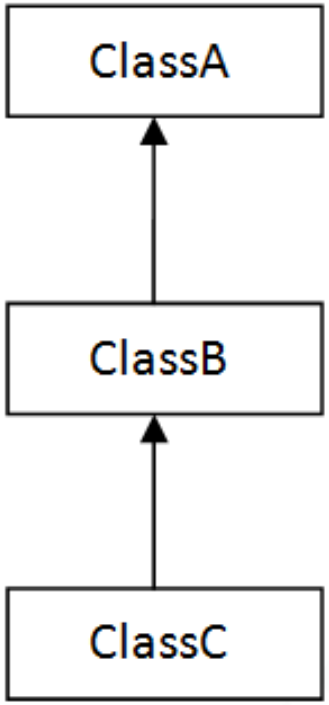
```
// Create a superclass.
class A
{

}

// Create a subclass by extending class A.
class B extends A
{

}
```

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.

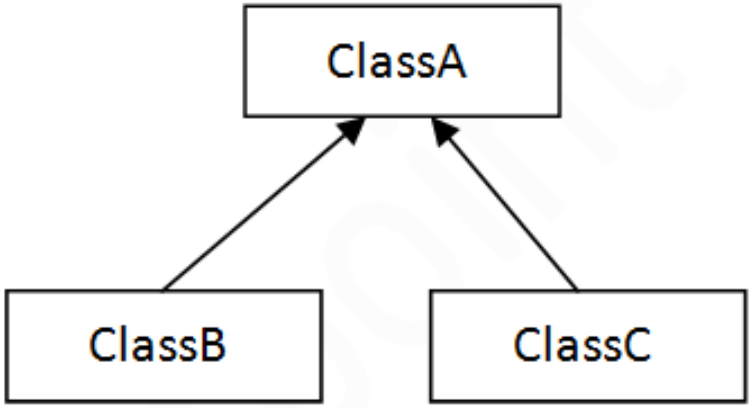
```
class Employee{
 float salary=43000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.sala
    ry);
   System.out.println("Bonus of Programmer is:"+p.b
    onus);
}
}
```



1) Single



2) Multilevel



3) Hierarchical

Single Inheritance

- 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();
}}
```

Multilevel Inheritance

- 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 TestInheritance {
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Hierarchical Inheritance

- 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.

```
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 TestInheritance {
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
c.bark();  //Compile Time Error
}}
```

multiple inheritance is not supported in java

- WHY?
- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

```
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?
}
}
```

- 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.

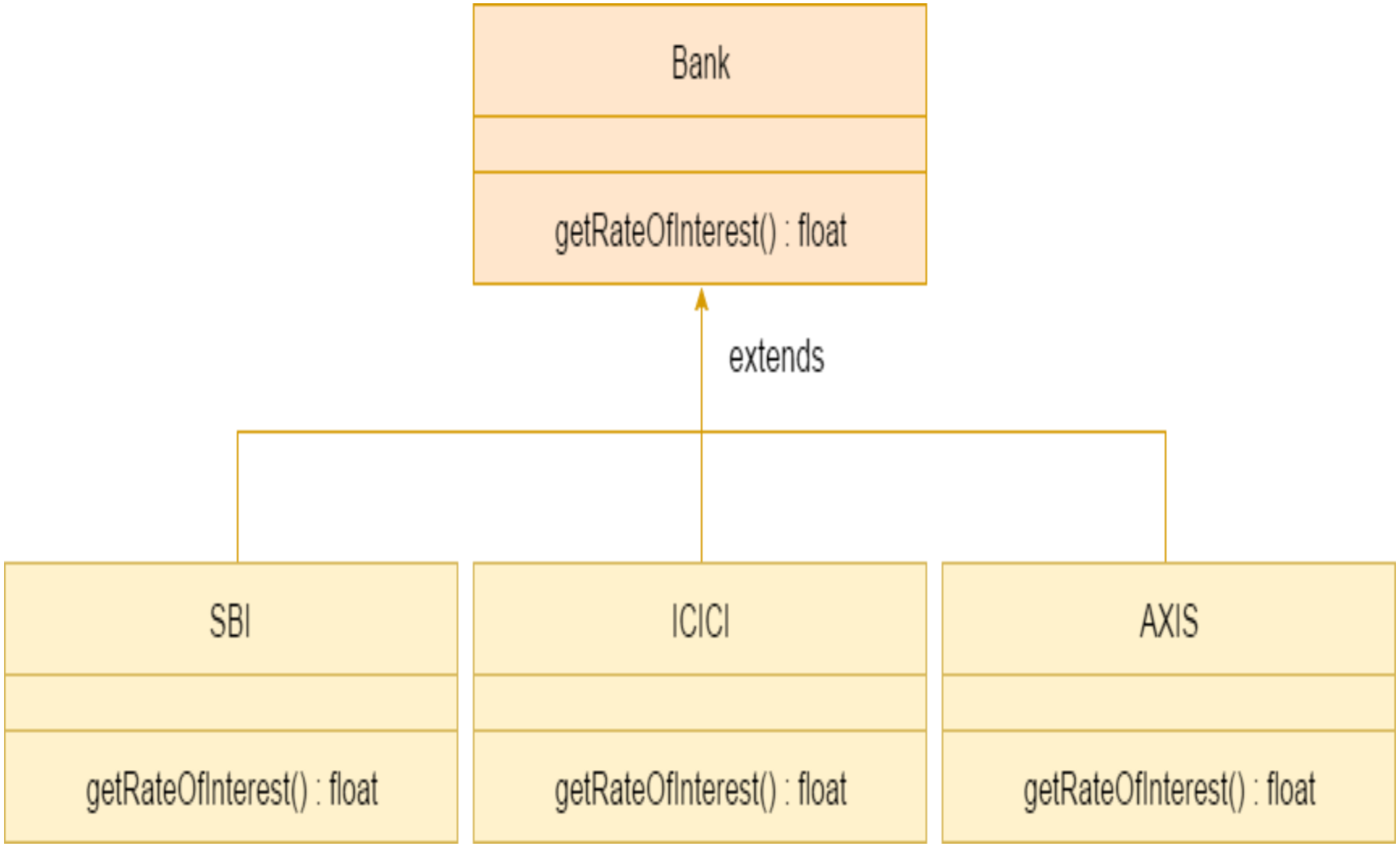
Access Protection

- Visibility of variables and methods within classes, subclasses, and packages.
- Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses
- The three access specifiers, **private, public,** and **protected,** provide a variety of ways to produce the many levels of access required by these categories.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

```
class OverloadingExample{
static int add(int a,int b)
{return a+b;}
static int add(int a,int b,int c){r
eturn a+b+c;}
}
```

```
//Java Program to illustrate the use of Java Method Overriding
 //Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}
   public static void main(String args[]){
  Bike obj = new Bike();//creating object
  obj.run();//calling method
  }
}
```


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

```
class Bank{
int getROI(){return 0;}
}
class SBI extends Bank{
int getROI(){return 8;}
}
class ICICI extends Bank{
int getROI(){return 7;}
}
class AXIS extends Bank{
int getROI(){return 9;}
}
class Test{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getROI());
System.out.println("ICICI Rate of Interest: "+i.getROI());
System.out.println("AXIS Rate of Interest: "+a.getROI());
}
}
```

Using super

- Two purpose
  - To call superclass constructor

```
super(arg-list);
```

  - To access a member of the superclass that has been hidden by a member of a subclass.
- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

```
class Vehicle{

  void run(){System.out.println("Vehicle is running");}
}

class Bike extends Vehicle{
    void run() { System.out.println("Bike is running safely");
Super.run();
}
   public static void main(String args[]){
  Bike obj = new Bike();//creating object
  obj.run();//calling method
   }
}
```

Why can we not override static method?

>It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

No, because the main is a static method.

Method Overloading	Method Overriding
Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Final Keyword

- Preventing method overriding
- Preventing inheritance

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

```
class Bike{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;  //compile time error
 }
 public static void main(String args[]){
 Bike obj=new  Bike();
 obj.run();
 }
}//end of class
```

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

```
class Bike{
  final void run(){System.out.println("running");}
}

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

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

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

```
final class Bike{}

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

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

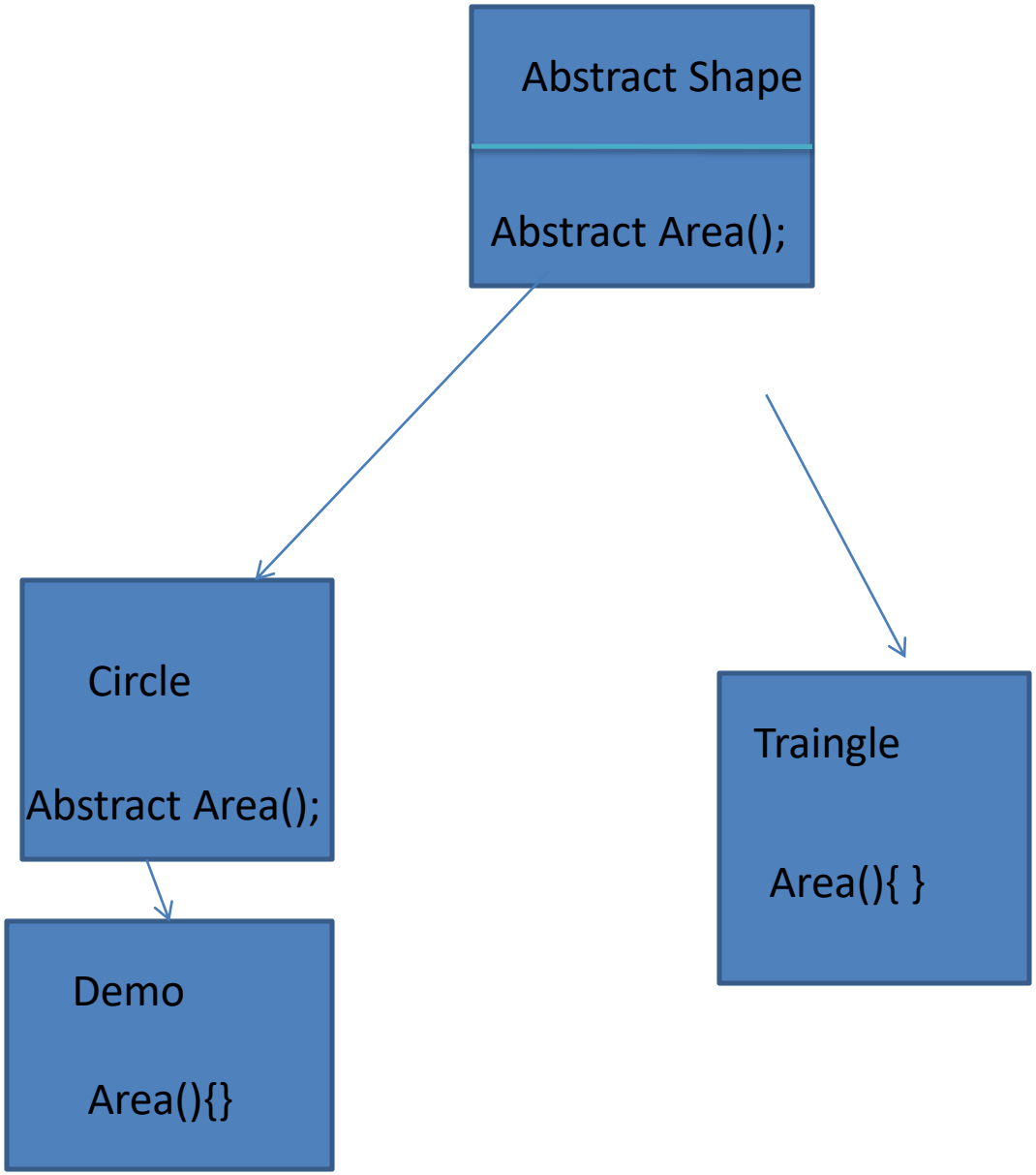
Is final method inherited?

Yes, final method is inherited but you cannot override it.

```
   class Bike{
  final void run(){System.out.println("running...");}
}
class Honda extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

Abstract Classes

- A superclass declares the structure of a given abstraction without providing a complete implementation of every method.
- A superclass that only defines a generalized form.
- Shared by all of its subclasses.
- Leaving it to each subclass to fill in the details.
- Superclass is unable to create a meaningful implementation for a method.


Abstract Shape
Abstract Area();
Circle
Abstract Area();
Traingle
Area(){ }
Demo
Area(){}

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

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

- A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.
- Points to Remember
- 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.

- You may have methods that must be overridden by the subclass in order for the subclass to have any meaning.
- You can require that certain methods be overridden by subclasses by specifying the **abstract type modifier.**
- A subclass must override them.

- To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

- Any class that contains one or more abstract methods must also be declared abstract.

- An abstract class cannot be directly instantiated with the **new operator.**
- You cannot declare abstract constructors, or abstract static methods.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract.**

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

Interfaces

- *interface* keyword is used
- Abstraction
  - Abstract class (0 to 100%)
  - Interface (100%)
- Syntactically similar to classes
- Lack instance variables, and methods are declared without any body.

- Once it is defined, any number of classes can implement an interface.
- One class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface.
- A class is free to determine the details of its own implementation.

Defining an Interface

```
access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    type final-varname1 = value;
    type final-varname2 = value;
    // ...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}
```

- Variables can be declared, they are implicitly **final and static**
  - They cannot be changed by the implementing class.
  - They must also be initialized.
- All methods and variables are implicitly **public.**

Implementing Interfaces

- one or more classes can implement the interface

```
class classname [extends superclass]
  [implements interface [,interface...]] {
   // class-body
}
```

- When you implement an interface method, it must be declared as **public.**

Syntax

```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

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

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

```
//Interface declaration: by first user
interface Drawable{
void draw();
}
//Implementation: by another user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by met
    hod e.g. getDrawable()
d.draw();
}}
```

```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface {
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

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

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

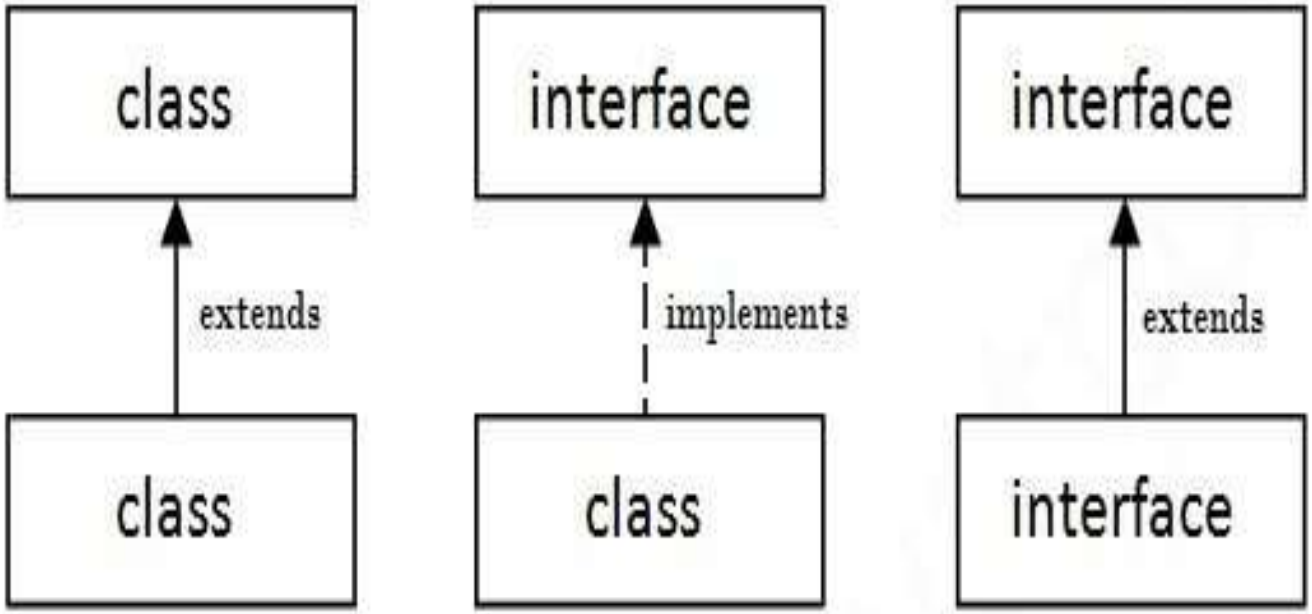
```
interface Line
{
   ...
}
interface Polygon
{
   ...
}
class Rectangle implements Line, Polygon
{
   ...
}
```

```
interface Line
{
   //members of Line interface
}

interface Polygon extends Line
{
   //members of Polygon interface and
Line interface
}
```

```
interface A
{
    ...
}
interface B
{
    ...
}

interface C extends A, B
{
    ...
}
```

Important points about interface

- We can’t create instance(interface can’t be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extends another interface or interfaces (more than one interface).
- A class that implements interface must implements all the methods in interface.
- All the methods are public and abstract. And all the fields are public, static, and final.
- It is used to achieve multiple inheritance.


class
extends
class
interface
implements
class
interface
extends
interface

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

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

Interface	Abstract Class
Pure abstract class	Incomplete class
It specify functionality which a class should have	It specify common attributes and behaviour that a class inherits
Contains only method declarations	Contains method declarations and definitions
Fields are implicitly final and static	Fields can be final, static as well as instance variables
Can not have constructor	Can have a constructor
Can be implemented by more than one class	Can be extended by more than one class
A class can implements more than one interface	A class can extend only one abstract class
An interface can be extended by another interface	An abstract class can be extended by another abstract as well as concrete class.