Inheritance and Interfaces:

Content

Inheritance in Java

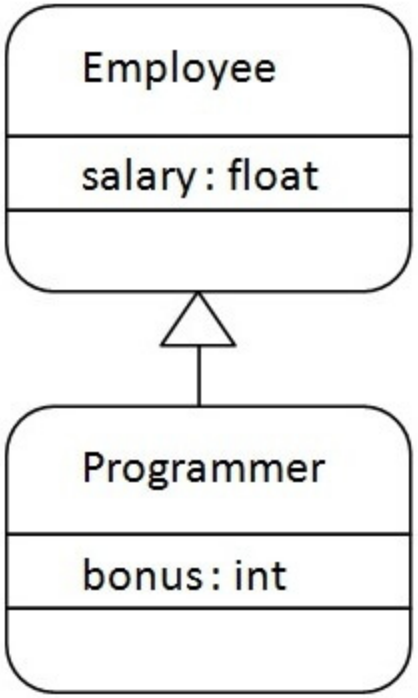
- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.
- Why use inheritance in java
  - For Method Overriding
  - 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

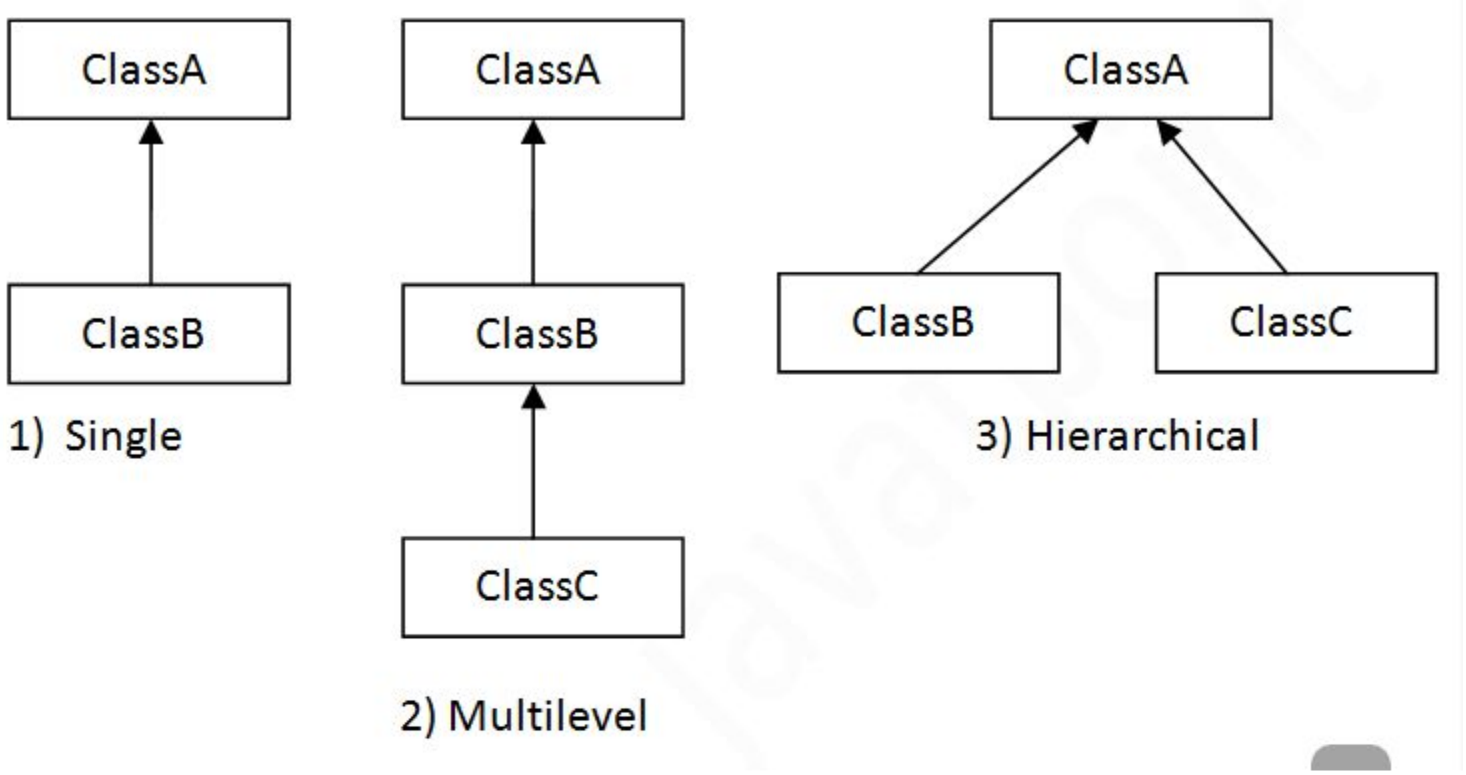
```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

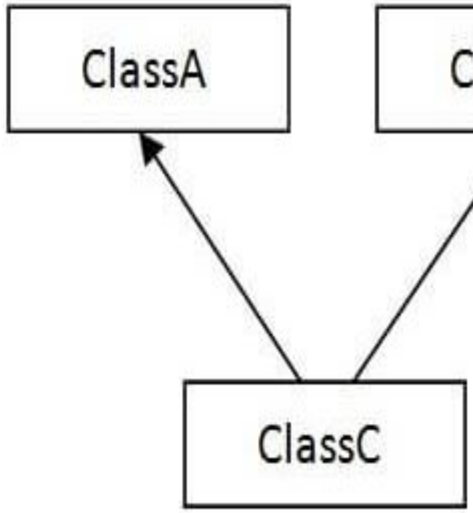
- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.



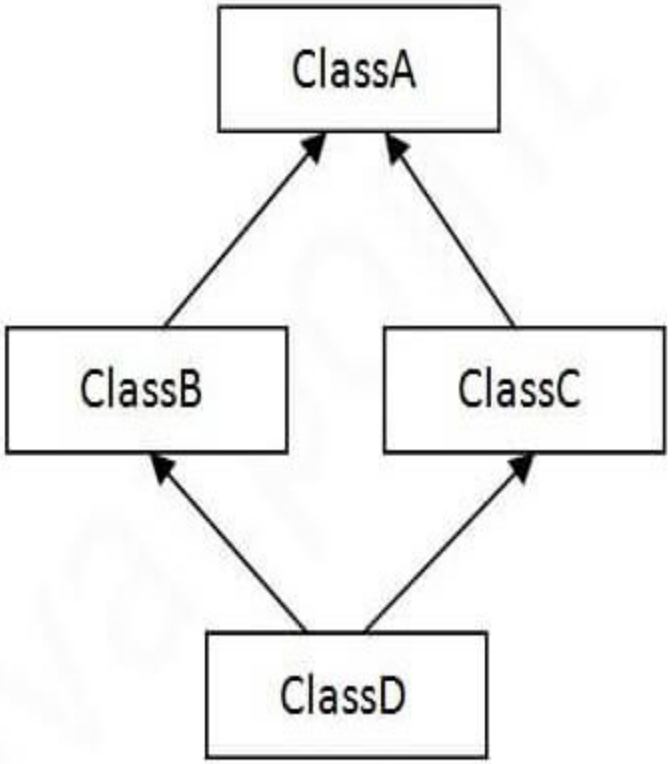
```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary i
s:"+p.salary);
   System.out.println("Bonus of Programm
er is:"+p.bonus);
}
}
```

- Types of inheritance in java
- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only.





4) Multiple



5) Hybrid

Single Inheritance Example

```
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();
}}
Output: barking…
Eating…
```

Why Constructors are not inherited in Java?

- Constructor is a block of code that allows you to create an object of class and has same name as class with no explicit return type.
- Whenever a class (child class) extends another class (parent class), the sub class inherits state and behavior in the form of variables and methods from its super class but it does not inherit constructor of super class because of following reasons:
- Constructors are special and have same name as class name. So if constructors were inherited in child class then child class would contain a parent class constructor which is against the constraint that constructor should have same name as class name

```
class Parent {
     Parent()
    {
    }

    public void print()
    {
    }
}

public class Child extends Parent {
     Parent()
    {
    }
    public void print()
    {
    }

    public static void main(String[] args)
    {
        Child c1 = new Child(); // allowed
        Child c2 = new Parent(); // not allowed
    }
}
```

- If we define Parent class constructor inside Child class it will give compile time error for return type and consider it a method. But for print method it does not give any compile time error and consider it a overriding method.
- Now suppose if constructors can be inherited then it will be impossible to achieving encapsulation. Because by using a super class’s constructor we can access/initialize private members of a class.
- A constructor cannot be called as a method. It is called when object of the class is created so it does not make sense of creating child class object using parent class constructor notation. i.e. Child c = new Parent();
- A parent class constructor is not inherited in child class and this is why super() is added automatically in child class constructor if there is no explicit call to super or this.

Multilevel Inheritance Example

```
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();
Dog d1 = new Dog();
D1.eat();
}}
```

Output:

```
Weeping….
Barking…
Eating….
```

- Method overriding is an example of runtime polymorphism. In method overriding, a subclass overrides a method with the same signature as that of in its superclass. During compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.
- We can override a method at any level of multilevel inheritance

```
class Animal {
  public void move() {
    System.out.println("Animals can move");
  }
}
class Dog extends Animal {
  public void move() {
    System.out.println("Dogs can walk and run");
  }
}
class Puppy extends Dog {
  public void move() {
    System.out.println("Puppy can move.");
  }
}
public class Tester {
  public static void main(String args[]) {
    Animal a = new Animal(); // Animal reference and object
    Animal b = new Puppy(); // Animal reference but Puppy
object
    a.move(); // runs the method in Animal class
    b.move(); // runs the method in Puppy class
  }
}
```

- **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?
}
}
```

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
  - 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.

- super is used to refer immediate parent class instance variable.
  - We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
// black, white
```

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

- super can be used to invoke parent class method
  - The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

```
Eating…
Eating bread
barking
```

- super is used to invoke parent class constructor.
- The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

- Output: animal is created
- dog is created

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:
  - Variable
  - method
  - 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.

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

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}
Output:Compile Time Error
```

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

Example of final method

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

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();
  }
}
```

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

Example of 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();
 }
}
```

- Is final method inherited?
- Yes, final method is inherited but you cannot override it. For Example:

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

- **What is blank or uninitialized final variable?**
- A final variable that is not initialized at the time of declaration is known as blank final variable.
- If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.
- It can be initialized only in constructor.

Example of blank final variable

```
class Student{
int id;
String name;
final String PAN_CARD_NUMBER;
...
}
```

Runtime Polymorphism in Java

- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.
- Let's first understand the upcasting before Runtime Polymorphism.
- Upcasting
  - If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.

```
class A{}
class B extends A{}
A a=new B();//upcasting
```

```
class Bike{
 void run(){System.out.println("running");}
}
class Splendor extends Bike{
 void run(){System.out.println("running safely with
   60km");}

 public static void main(String args[]){
   Bike b = new Splendor();//upcasting
   b.run();  //running safely with 60km
 }
}
```

```
class Bank{
float getRateOfInterest(){return 0;}
}
class SBI extends Bank{
float getRateOfInterest(){return 8.4f;}
}
class ICICI extends Bank{
float getRateOfInterest(){return 7.3f;}
}
class AXIS extends Bank{
float getRateOfInterest(){return 9.7f;}
}
```

```
class TestPolymorphism{
public static void main(String args[]){
Bank b =new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());  //8.4f
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());  //7.3f
b=new AXIS();
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());  //9.7f
}
}
```

Java Abstract Class

- Abstract class 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).
- Before learning the Java abstract class, let's understand the abstraction in Java first.
- Abstraction in Java
- **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.
- Abstraction lets you focus on what the object does instead of how it does it.
- Ways to achieve Abstraction
- There are two ways to achieve abstraction in java
  - Abstract class (0 to 100%)
  - Interface (100%)

- Abstract class in Java
- 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.

- **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();
}
}
```

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}}
```

Interface in Java

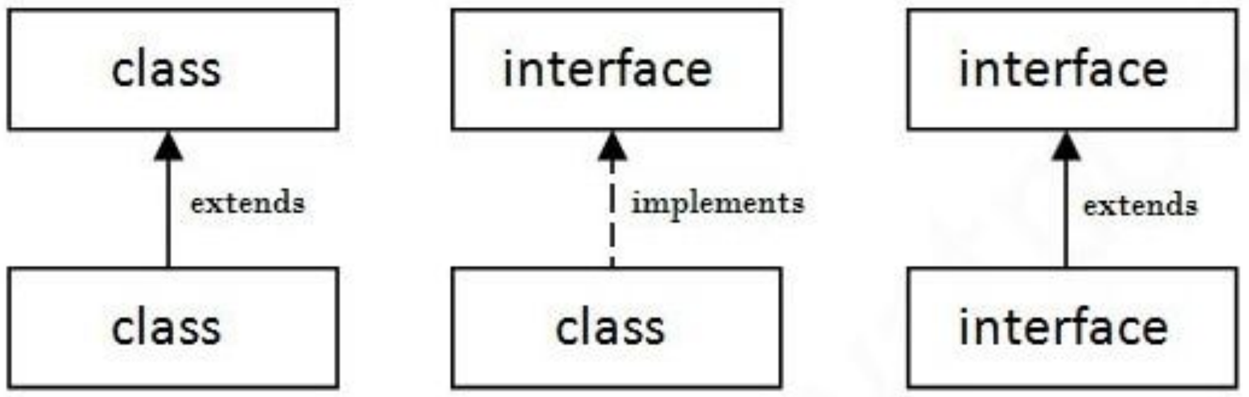
- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in 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.
- Since Java 8, we can have **default and static methods** in an interface.
- Since Java 9, we can have **private methods** in an interface.

- **Why use Java interface?**
- There are mainly three reasons to use interface. They are given below.
- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.
- **How to declare an interface?**
- An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.
- Syntax:

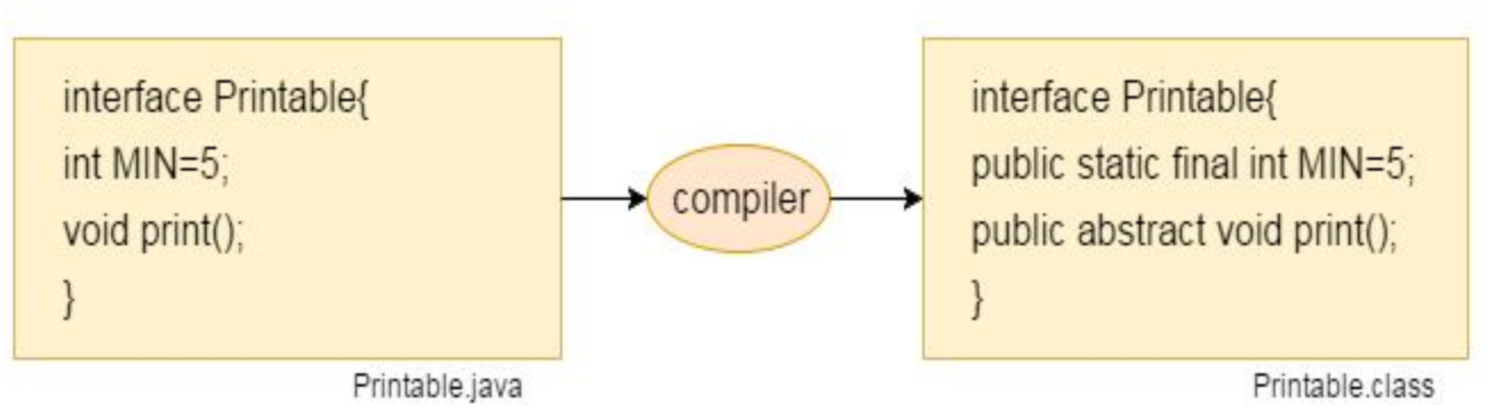
```
interface <interface_name>{

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

- 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**.



- **Internal addition by the compiler**
- 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



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

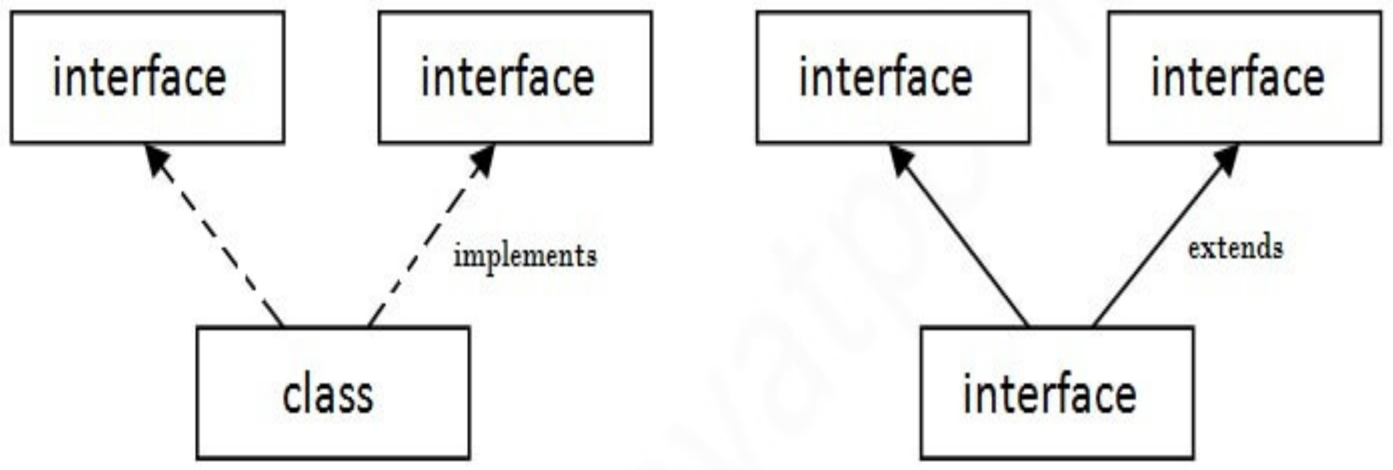
```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

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

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

```
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 TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

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



Multiple Inheritance in 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 is not supported through class in java, but it is possible by an interface, why?**
- As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

```
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();
 }
}
```

- **Interface inheritance**
- A class implements an interface, but one interface extends another interface.

```
interface Printable{
void print();
Void abc();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
Public void abc(){}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

- **Java 8 Default Method in Interface**
- Since Java 8, we can have method body in interface. But we need to make it default method

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}
```

- **Java 8 Static Method in Interface**
- Since Java 8, we can have static method in interface

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

Java instanceof operator

- The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).
- The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

```
class Simple1{
 public static void main(String args[]){
 Simple1 s=new Simple1();
 System.out.println(s instanceof Simple1);//true
 }
}
```

- An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

```
class Animal{}
class Dog1 extends Animal{//Dog inherits Animal

 public static void main(String args[]){
 Dog1 d=new Dog1();
 System.out.println(d instanceof Animal);//true
Animal a = new Animal();
//a instanceof Dog ⍰ false
 }
}
```

- **instanceof in java with a variable that have null value**
- If we apply instanceof operator with a variable that have null value, it returns false. Let's see the example given below where we apply instanceof operator with the variable that have null value.

```
class Dog2{
 public static void main(String args[]){
  Dog2 d=null;
   Dog2 d1 = new Dog2();//true
  System.out.println(d instanceof Dog2);//false
 }
}
```

- **Downcasting with java instanceof operator**
- When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error.
- If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.
  - **Dog d=new Animal();//Compilation error**
- If we perform downcasting by typecasting, ClassCastException is thrown at runtime.
  - Dog d=(Dog)**new** Animal();
  - //Compiles successfully but ClassCastException is thrown at runtim

- Possibility of downcasting with instanceof

```
class Animal { }

class Dog3 extends Animal {
  static void method(Animal a) {
    if(a instanceof Dog3){
       Dog3 d=(Dog3)a;//downcasting
       System.out.println("ok downcasting performed");
    }
  }

  public static void main (String [] args) {
    Animal a=new Dog3(); //upcasting
    Dog3.method(a);
  }


 }
```

- Downcasting without the use of java instanceof
  - Downcasting can also be performed without the use of instanceof operator as displayed in the following example:

```
class Animal { }
class Dog4 extends Animal {
 static void method(Animal a) {
     Dog4 d=(Dog4)a;//downcasting
     System.out.println("ok downcasting performed");
 }
  public static void main (String [] args) {
   Animal a=new Dog4();  //upcasting
   Dog4.method(a);
 }
}
```

```
interface Printable{}
class A implements Printable{
public void a(){System.out.println("a method");}
}
class B implements Printable{
public void b(){System.out.println("b method");}
}
```

```
class Call{
void invoke(Printable p){//upcasting
if(p instanceof A){
A a=(A)p;//Downcasting
a.a();
}
if(p instanceof B){
B b=(B)p;//Downcasting
b.b();
}

}
}//end of Call class

class Test4{
public static void main(String args[]){
Printable p=new B();  //upcasting
Call c=new Call();
c.invoke(p);
}
}
```

Understanding of Java Object Class

- The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.
- Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object. For example:
  - Object obj=getObject();//we don't know what object will be returned fr om this method
- The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.

public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Object Cloning in Java

- The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.
- The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.
- The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:
  - **protected** Object clone() **throws** CloneNotSupportedException

- Why use clone() method ?
  - The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.
- Advantage of Object cloning
  - Although Object.clone() has some design issues but it is still a popular and easy way of copying objects. Following is a list of advantages of using clone() method:
  - You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.
  - It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
  - Clone() is the fastest way to copy array.

- Disadvantage of Object cloning
  - To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
  - We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
  - Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
  - Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
  - If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
  - Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

```
class Student18 implements C
    loneable{
int rollno;
String name;

Student18(int rollno,String na
    me){
this.rollno=rollno;
this.name=name;
}

public Object clone()throws C
    loneNotSupportedExceptio
    n{
return super.clone();
}
```

```
public static void main(String args[]){
try{
Student18 s1=new Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

System.out.println(s1.rollno+" "+s1.name); //1
01 amit
System.out.println(s2.rollno+" "+s2.name); //
101  amit

}catch(CloneNotSupportedException c){}

}
}
```

Comparison between Abstract Class and interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.

7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

```
interface A{
void a();//bydefault, public and a
     bstract
void b();
void c();
void d();
}

//Creating abstract class that pro
     vides the implementation of o
     ne method of A interface
abstract class B implements A{
public void c(){System.out.println
     ("I am C");}
  public void a(){}
public void b(){}
public void d(){}
}

//Creating subclass of abstract class, now we n
eed to provide the implementation of rest of t
he methods
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

//Creating a test class that calls the methods of
 A interface
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```