Object Oriented Programming with Java
Practical-4

Introduction to Inheritance

Java Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).
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 (so 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

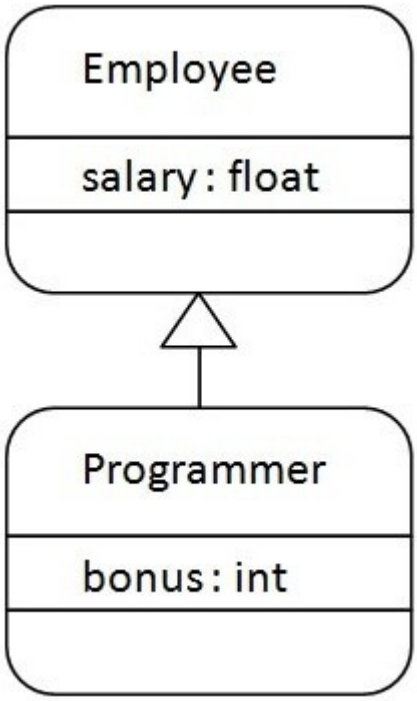
```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

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.

Java Inheritance Example



As displayed in the above figure, 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.

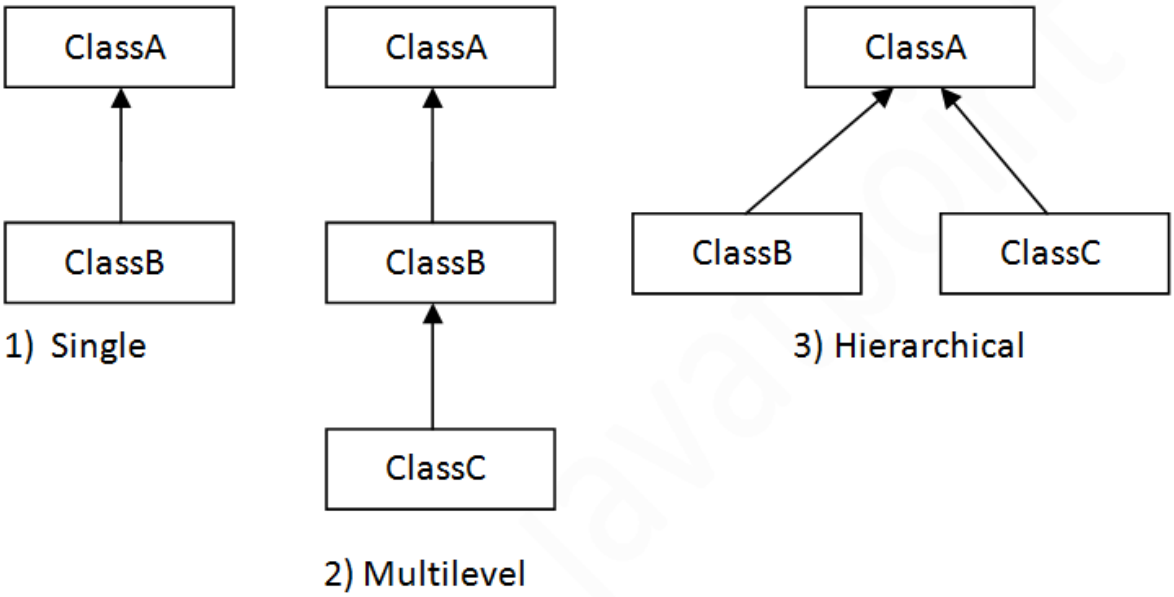
```
class Employee
{
      float salary=40000;
}
class Programmer extends Employee
{
      int bonus=10000;
      public static void main(String args[])
      {
         Programmer p=new Programmer();
         System.out.println("Programmer salary is:"+p.salary);
         System.out.println("Bonus of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

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

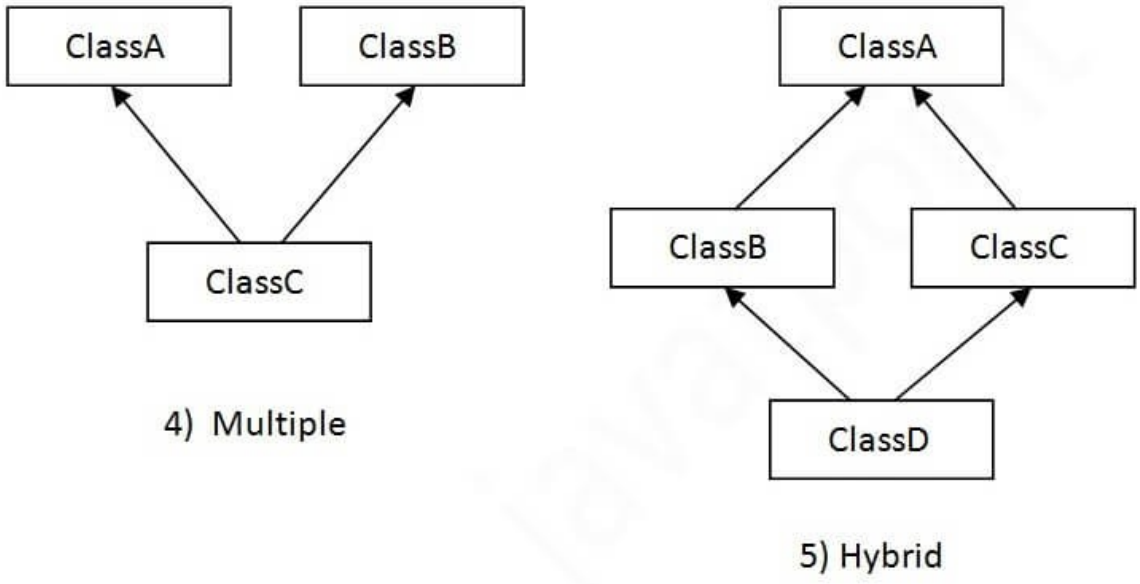
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



.

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();
}}
```

Output:

```
barking...
eating...
```

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();
}}
```

Output:

weeping...
barking...
eating...

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.

```
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
}}
```

Output:

meowing...
eating...

Q) 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?
}
}
```

Compile Time Error

is-a relationship

In Java, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an is-a relationship between two classes. For example,

- **Car** is a **Vehicle**
- **Orange** is a **Fruit**
- **Surgeon** is a **Doctor**
- **Dog** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Orange** can inherit from **Fruit**, and so on.

Method Overriding in Java Inheritance

However, if the same method is present in both the superclass and subclass, what will happen?

In this case, the method in the subclass overrides the method in the superclass. This concept is known as method overriding in Java.

Example 2: Method overriding in Java Inheritance

```
class Animal {

  // method in the superclass
  public void eat()
  {
        System.out.println("I can eat");
  }
}

// Dog inherits  Animal
class Dog extends
Animal {

  // overriding the eat()
```

```
 method @Override
 public void eat() { System.out.println("I eat dog ood");
 }

 // new method in subclass
 public void bark() {
  System.out.println("I can bark");
 }
}

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

  // create an object of the subclass
  Dog labrador = new Dog();

  // call the eat() method
  labrador.eat();
  labrador.bark();
 }
}
```

Output

```
I eat dog food
I can bark
```

In the above example, the eat() method is present in both the superclass *Animal* and the subclass
Dog.

Here, we have created an object *labrador* of *Dog*.

Now when we call eat() using the object *labrador*, the method inside *Dog* is called. This is because the method inside the derived class overrides the method inside the base class.

This is called method overriding.

super Keyword in Java Inheritance

Previously we saw that the same method in the subclass overrides the method in superclass.

In such a situation, the super keyword is used to call the method of the parent class from the method of the child class.

Example 3: super Keyword in Inheritance

```
class Animal {

  // method in the superclass
  public void eat() {
  System.out.println("I can
  eat");
  }
}

// Dog inherits  Animal
class Dog extends
Animal {

  // overriding the eat()
  method @Override
  public void eat() {

    // call method of
    superclass super.eat();
    System.out.println("I eat dog food");
  }

  // new method in subclass
  public void bark()
  {
        System.out.println("I can bark");
  }
}

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

    // create an object of the subclass
    Dog labrador = new Dog();

    // call the eat() method
    labrador.eat(); labrador.bark();
  }
}
```

Output

```
I can eat
I eat dog food
I can bark
```

In the above example, the eat() method is present in both the base class *Animal* and the derived class *Dog*. Notice the statement,

super.eat();

Here, the super keyword is used to call the eat() method present in the superclass.

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).

Example of Abstract class that has an abstract method

```
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();
}
}
```

Another example of Abstract class

```
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()+" %");
}}
```

Abstract class having constructor, data member and methods

```
//Example of an abstract class that has abstract and non-abstract methods abstract
 class Bike{
  Bike(){System.out.println("bike is created");}
  abstract void run();
  void changeGear(){System.out.println("gear changed");}
 }
//Creating a Child class which inherits
 Abstract class class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
//Creating a Test class which calls abstract and non-
 abstract methods class TestAbstraction2{
 public static void main(String
  args[]){ Bike obj = new
  Honda();
  obj.run();
  obj.changeGear();
 }
}
```

Method Overloading

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) Method Overloading: changing no. of arguments

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

2) Method Overloading: changing data type of arguments

```
class Adder{
```

```
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```