Inheritance in Java

← prev next →

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.

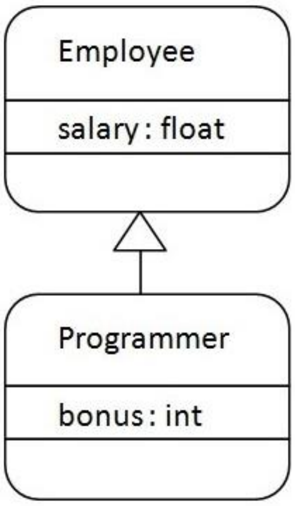
The syntax of Java Inheritance

```
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);
}
}
```

Test it Now

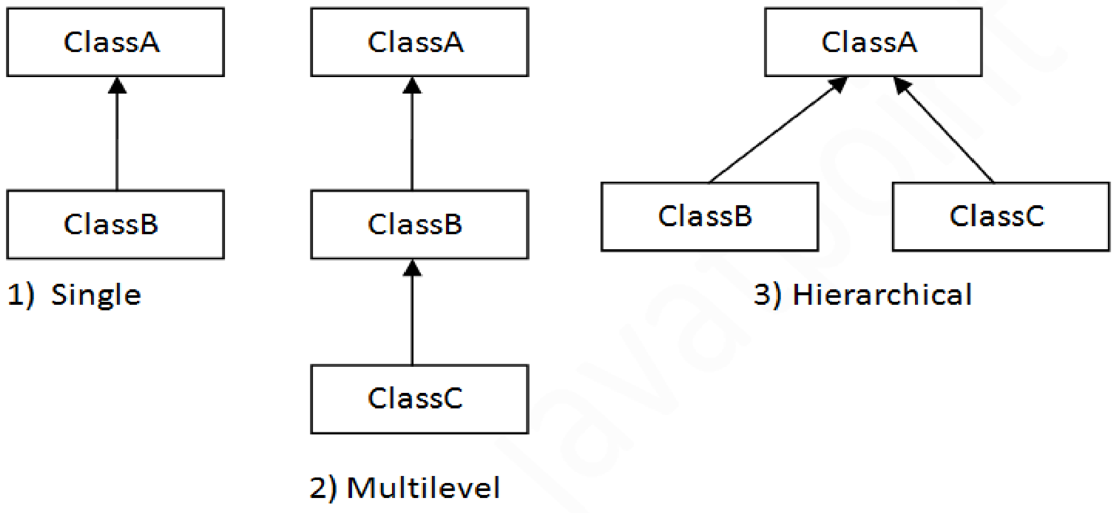
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

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

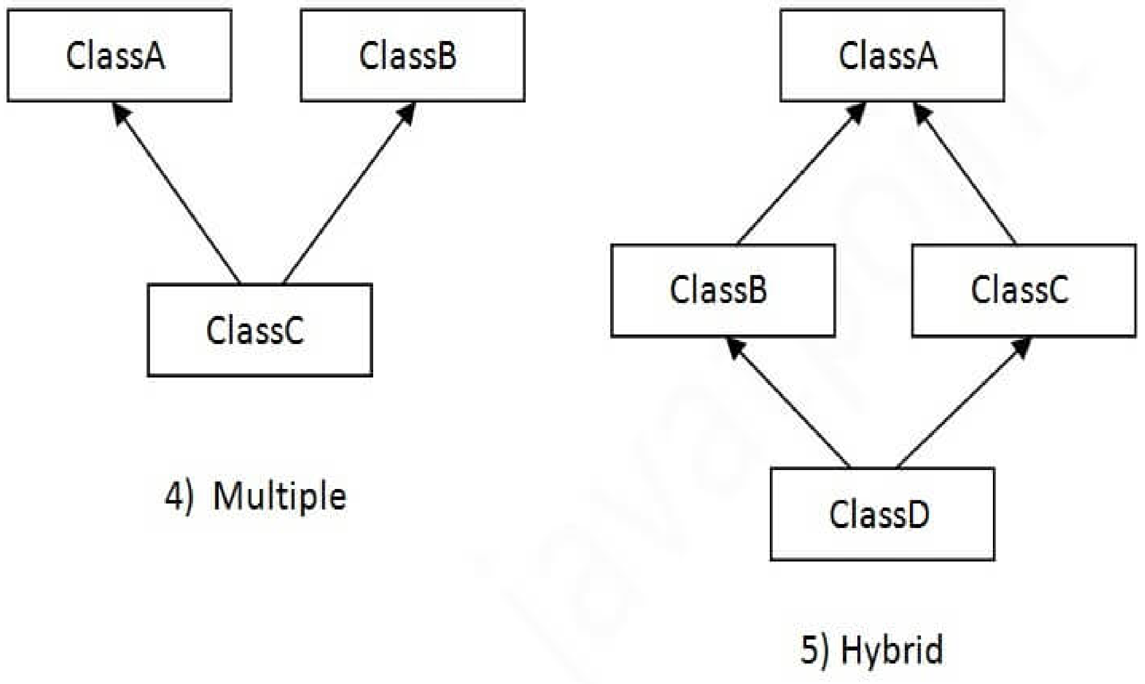
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

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



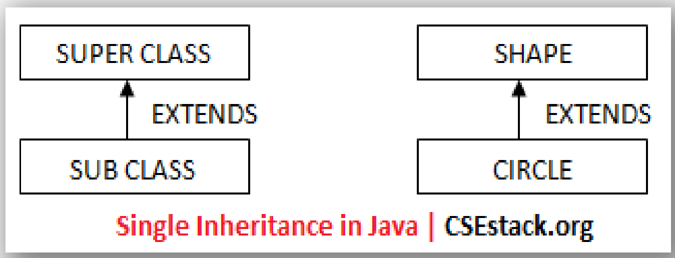
Note: Multiple inheritance is not supported in Java through class.

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



1) Single Inheritance:

When a single child class extends the properties of the parent class, then it is said to showcase single inheritance. Single inheritance can be represented as follows.



Single Inheritance in Java | CSEstack.org

For example,

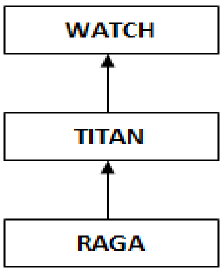
```
class Shape{
    void draw()
    {
        System.out.println("Draw shape");
    }
}
class Circle extends Shape
{
    void drwaCircle(){
        System.out.println("Draw Circle");
    }
 public static void main(String args[])
{
    Circle c  = new Circle();
    c.draw();
    c.drawCircle();
}
}
```

Output:

```
Draw shape
Draw Circle
```

2) Multilevel Inheritance:

When classes extend the properties of each other level by level is known as multilevel inheritance. For instance, class Z extends Y and class Y extends X. The pictorial representation below will help you understand the gist of multi-level inheritance better.



Multilevel Inheritance in Java | CSEstack.org

Above is a pictorial representation of Multi-Level Inheritance.

A simple example is followed:

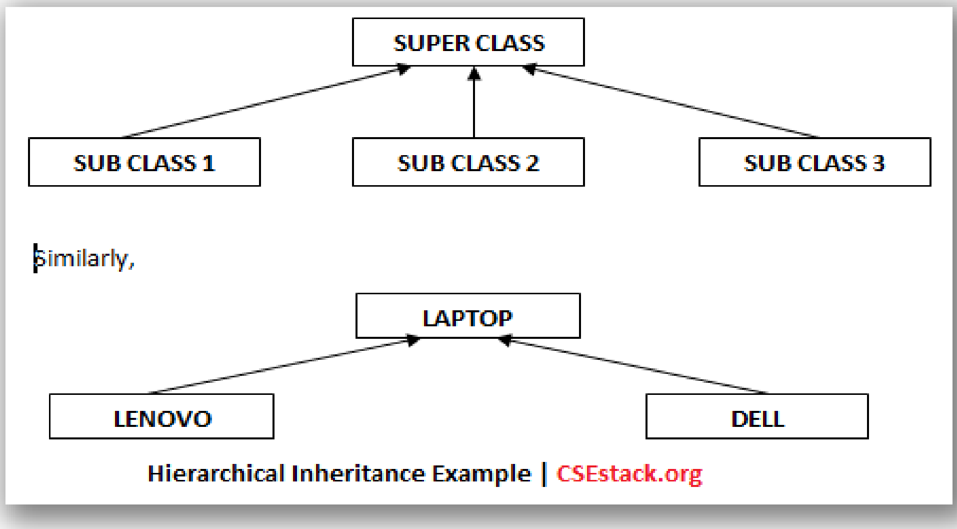
```
class Watch
{
    void display()
    {
        System.out.println("WATCH");
    }
class Titan extends Watch
{
    void property()
    {
        System.out.println("TITAN");
    }
}
class Raga extends Titan
{
    void type()
    {
        System.out.println("RAGA");
        System.out.println("Classic Collection");
    }
}

class TestDemo
{
    public static void main(String args[])
    {
        Raga r = new Raga();
        r.display();
r.property();
r.type();
    }
}
```

Output:

```
WATCH
TITAN
RAGA
Classic Collection
```

3) Hierarchical Inheritance:

When one single class is inherited by multiple subclasses is known as hierarchical inheritance. Before we discuss an example of hierarchical inheritance, let us look at the pictorial representation of hierarchical inheritance first.



Hierarchical Inheritance Example | CSEstack.org

A simple example is discussed below:

```java
class Laptop
{
    void display()
    {
        System.out.println("Working...");
    }
class Dell extends Laptop
{
    void print()
    {
        System.out.println("Dell Inspiron");
    }
}
class Lenovo extends Laptop
{
    void show()
    {
        System.out.println("Lenovo YOGA");
    }
}
class TestDemo
{
    public static void main(String args[]){
        Dell d = new Dell();
        d.print();
        d.display();
        Lenovo l = new Lenovo();
        l.show();
        l.display();
    }
}
```

Output:

```
Dell Inspiron
Working...
Lenovo YOGA
Working...
```

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.

```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?
}
}
```

Test it Now

```
Compile Time Error
```

Super Keyword

Uses of super keyword

1. To call methods of the superclass that is overridden in the subclass.
2. To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.
3. To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor.

Let’s understand each of these uses.

---

1. Access Overridden Methods of the superclass

If methods with the same name are defined in both superclass and subclass, the method in the subclass overrides the method in the superclass. This is called method overriding.

Example 1: Method overriding

```java
class Animal {

  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }

  public void printMessage(){
    display();
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

Output

```
I am a dog
```

Example 2: super to Call Superclass Method

```java
class Animal {

   // overridden method
   public void display(){
      System.out.println("I am an animal");
   }
}

class Dog extends Animal {

   // overriding method
   @Override
   public void display(){
      System.out.println("I am a dog");
   }

   public void printMessage(){

      // this calls overriding method
      display();

      // this calls overridden method
      super.display();
   }
}

class Main {
   public static void main(String[] args) {
      Dog dog1 = new Dog();
      dog1.printMessage();
   }
}
```

Output

```
I am a dog
I am an animal
```

2. Access Attributes of the Superclass

The superclass and subclass can have attributes with the same name. We use the `super` keyword to access the attribute of the superclass.

Example 3: Access superclass attribute

```
class Animal {
  protected String type="animal";
}

class Dog extends Animal {
  public String type="mammal";

  public void printType() {
    System.out.println("I am a " + type);
    System.out.println("I am an " + super.type);
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printType();
  }
}
```

When we run this program, the output will be:

```
I am a mammal
I am an animal
```

In this example, we have defined the same instance field `type` in both the superclass `Animal` and the subclass `Dog`.

We then created an object `dog1` of the `Dog` class. Then, the `printType()` method is called using this object.

Inside the `printType()` function,

- `type` refers to the attribute of the subclass `Dog`.
- `super.type` refers to the attribute of the superclass Animal.

3. Use of super() to access superclass constructor

As we know, when an object of a class is created, its default constructor is automatically called.

To explicitly call the superclass constructor from the subclass constructor, we use `super()`. It's a special form of the `super` keyword.

`super()` can be used only inside the subclass constructor and must be the first statement.

Example 4: Use of super()

```java
class Animal {

  // default or no-arg constructor of class Animal
  Animal() {
    System.out.println("I am an animal");
  }
}

class Dog extends Animal {

  // default or no-arg constructor of class Dog
  Dog() {

    // calling default constructor of the superclass
    super();

    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

Output

```
I am an animal
I am a dog
```

Example 5: Call Parameterized Constructor Using super()

```java
class Animal {

  // default or no-arg constructor
  Animal() {
    System.out.println("I am an animal");
  }

  // parameterized constructor
  Animal(String type) {
    System.out.println("Type: "+type);
  }
}

class Dog extends Animal {

  // default constructor
  Dog() {

    // calling parameterized constructor of the superclass
    super("Animal");

    System.out.println("I am a dog");
  }
}

class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
  }
}
```

Output

```
Type: Animal
I am a dog
```

The compiler can automatically call the no-arg constructor. However, it cannot call parameterized constructors.

If a parameterized constructor has to be called, we need to explicitly define it in the subclass constructor.