Inheritance in Java

**Inheritance in java** is a mechanism in which one object acquires all the properties and behaviors of 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 parent class, and you can add new methods and fields also.

Inheritance represents the **IS-A relationship**, also known as parent-child relationship.

### **Why use inheritance in java**

* For Method Overriding (so runtime polymorphism can be achieved).
* For Code Reusability.

### **Syntax of Java Inheritance**

**class** Subclass-name **extends** Superclass-name

{

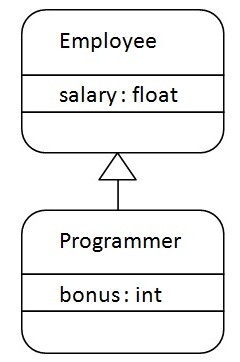
   //methods and fields

}

The **extends keyword** indicates that you are making a new class that derives from an existing class.

In the terminology of Java, a class that is inherited is called a super class. The new class is called a subclass.

### **Understanding the simple example of inheritance**



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. Relationship between 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);

}

}

Programmer salary is:40000.0

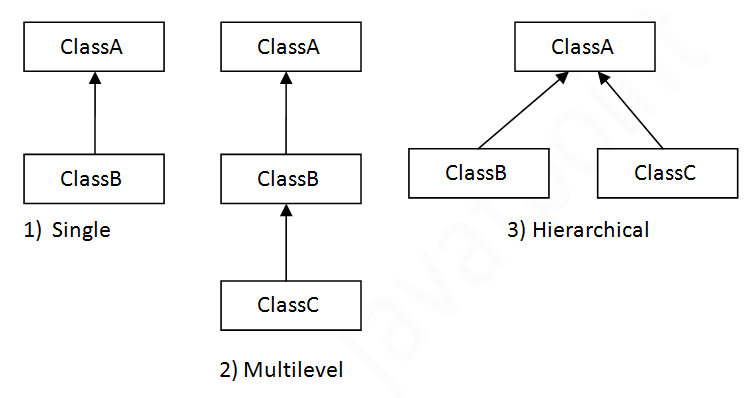
Bonus of programmer is:10000

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**

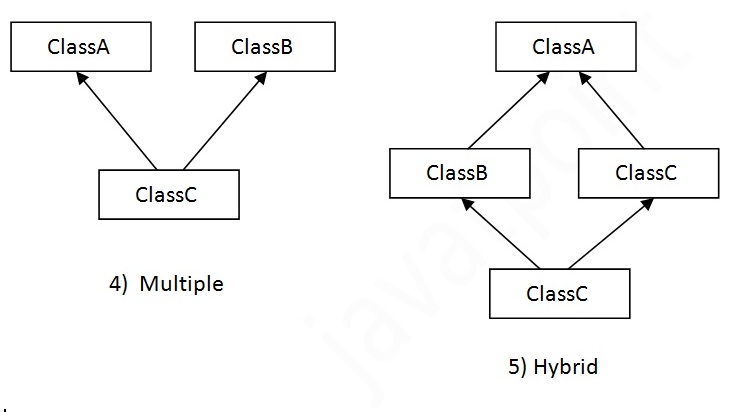
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

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 a class extends multiple classes i.e. known as multiple inheritance. For Example:



## **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 same method and you call it from child class object, there will be ambiguity to call 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 now.

**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

# Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

**class** Employee{

**int** id;

String name;

Address address;//Address is a class

...

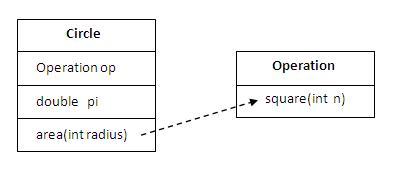
}

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

### Why use Aggregation?

* For Code Reusability.

### **Simple Example of Aggregation**



In this example, we have created the reference of Operation class in the Circle class.

**class** Operation{

**int** square(**int** n){

**return** n\*n;

  }

}

**class** Circle{

  Operation op;//aggregation

**double** pi=3.14;

**double** area(**int** radius){

    op=**new** Operation();

**int** rsquare=op.square(radius);//code reusability (i.e. delegates the method call).

**return** pi\*rsquare;

  }

**public** **static** **void** main(String args[]){

    Circle c=**new** Circle();

**double** result=c.area(5);

    System.out.println(result);

  }

}

Output:78.5

### **When use Aggregation?**

* Code reuse is also best achieved by aggregation when there is no is-a relationship.
* Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

### **Understanding meaningful example of Aggregation**

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

#### Address.java

**public** **class** Address {

String city,state,country;

**public** Address(String city, String state, String country) {

**this**.city = city;

**this**.state = state;

**this**.country = country;

}

}

Emp.java

**public** **class** Emp {

**int** id;

String name;

Address address;

**public** Emp(**int** id, String name,Address address) {

**this**.id = id;

**this**.name = name;

**this**.address=address;

}

**void** display(){

System.out.println(id+" "+name);

System.out.println(address.city+" "+address.state+" "+address.country);

}

**public** **static** **void** main(String[] args) {

Address address1=**new** Address("gzb","UP","india");

Address address2=**new** Address("gno","UP","india");

Emp e=**new** Emp(111,"varun",address1);

Emp e2=**new** Emp(112,"arun",address2);

e.display();

e2.display();

}

}

Output:111 varun

gzb UP india

112 arun

gno UP india

# Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

### **Usage of Java Method Overriding**

* Method overriding is used to provide specific implementation of a method that is already provided by its super class.
* Method overriding is used for runtime polymorphism

#### **Rules for Java Method Overriding**

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

### Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

**class** Vehicle{

**void** run(){

System.out.println("Vehicle is running");

}

}

**class** Bike **extends** Vehicle{

**public** **static** **void** main(String args[]){

   Bike obj = **new** Bike();

   obj.run();

   }

}

Output:Vehicle is running

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

### **Example of method overriding**

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

**class** Vehicle{

**void** run(){

System.out.println("Vehicle is running");

}

}

**class** Bike2 **extends** Vehicle{

**void** run(){

System.out.println("Bike is running safely");

}

**public** **static** **void** main(String args[]){

Bike2 obj = **new** Bike2();

obj.run();

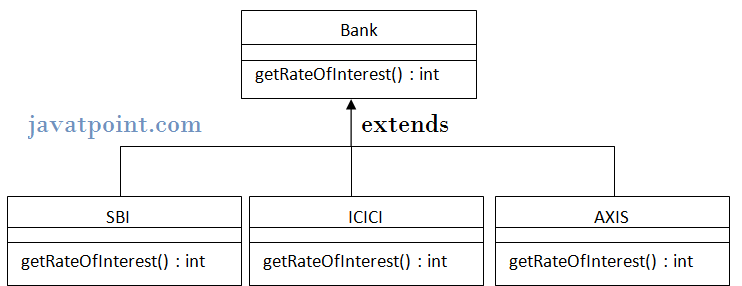
}

}

Output:Bike is running safely

### **Real example of Java Method Overriding**

Consider a scenario, Bank is a class that provides functionality to get rate of interest. But, rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7% and 9% rate of interest.



**class** Bank{

**int** getRateOfInterest(){

**return** 0;

}

}

**class** SBI **extends** Bank{

**int** getRateOfInterest(){

**return** 8;

}

}

**class** ICICI **extends** Bank{

**int** getRateOfInterest(){

**return** 7;

}

}

**class** AXIS **extends** Bank{

**int** getRateOfInterest(){

**return** 9;

}

}

**class** Test2{

**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.getRateOfInterest());

System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());

System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());

}

}

Output:

SBI Rate of Interest: 8

ICICI Rate of Interest: 7

AXIS Rate of Interest: 9

### **Can we override static method?**

No, static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

### **Why we cannot override static method?**

because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

### **Can we override java main method?**

No, because main is a static method.

# Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:

#### Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.

### **Simple example of Covariant Return Type**

**class** A{

A get(){**return** **this**;}

}

**class** B1 **extends** A{

B1 get(){**return** **this**;}

**void** message(){System.out.println("welcome to covariant return type");}

**public** **static** **void** main(String args[]){

**new** B1().get().message();

}

}

Output:welcome to covariant return type

As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

# super keyword in java

The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

## **Usage of java super Keyword**

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

## **1) super is used to refer immediate parent class instance variable.**

***Problem without super keyword***

**class** Vehicle{

**int** speed=50;

}

**class** Bike3 **extends** Vehicle{

**int** speed=100;

**void** display(){

   System.out.println(speed);//will print speed of Bike

  }

**public** **static** **void** main(String args[]){

   Bike3 b=**new** Bike3();

   b.display();

}

}

Output:100

|  |
| --- |
| In the above example Vehicle and Bike both class have a common property speed. Instance variable of current class is refered by instance bydefault, but I have to refer parent class instance variable that is why we use super keyword to distinguish between parent class instance variable and current class instance variable. |

***Solution by super keyword***

//example of super keyword

**class** Vehicle{

**int** speed=50;

}

**class** Bike4 **extends** Vehicle{

**int** speed=100;

**void** display(){

   System.out.println(**super**.speed);//will print speed of Vehicle now

  }

**public** **static** **void** main(String args[]){

   Bike4 b=**new** Bike4();

   b.display();

}

}

Output:50

## **2) super is used to invoke parent class constructor.**

|  |
| --- |
| The super keyword can also be used to invoke the parent class constructor as given below: |

**class** Vehicle{

  Vehicle(){System.out.println("Vehicle is created");}

}

**class** Bike5 **extends** Vehicle{

  Bike5(){

**super**();//will invoke parent class constructor

   System.out.println("Bike is created");

  }

**public** **static** **void** main(String args[]){

   Bike5 b=**new** Bike5();

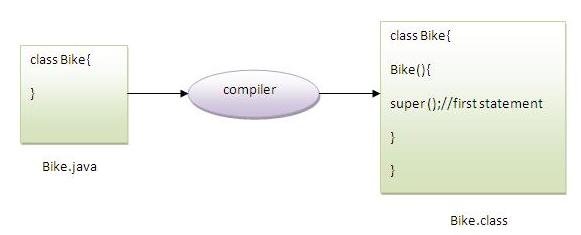
}

}

Output:Vehicle is created

Bike is created

#### Note: super() is added in each class constructor automatically by compiler.



As we know well that default constructor is provided by compiler automatically but it also adds super() for the first statement.If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

#### Another example of super keyword where super() is provided by the compiler implicitly.

**class** Vehicle{

  Vehicle(){System.out.println("Vehicle is created");}

}

**class** Bike6 **extends** Vehicle{

**int** speed;

  Bike6(**int** speed){

**this**.speed=speed;

    System.out.println(speed);

  }

**public** **static** **void** main(String args[]){

   Bike6 b=**new** Bike6(10);

 }

}

Output:Vehicle is created

10

## **3) 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 in case subclass contains the same method as parent class as in the example given below: |

**class** Person{

**void** message(){System.out.println("welcome");}

}

**class** Student16 **extends** Person{

**void** message(){System.out.println("welcome to java");}

**void** display(){

message();//will invoke current class message() method

**super**.message();//will invoke parent class message() method

}

**public** **static** **void** main(String args[]){

Student16 s=**new** Student16();

s.display();

}

}

Output:welcome to java

welcome

|  |
| --- |
| In the above example Student and Person both classes have message() method if we call message() method from Student class, it will call the message() method of Student class not of Person class because priority is given to local. |

|  |
| --- |
| In case there is no method in subclass as parent, there is no need to use super. In the example given below message() method is invoked from Student class but Student class does not have message() method, so you can directly call message() method. |

### Program in case super is not required

**class** Person{

**void** message(){System.out.println("welcome");}

}

**class** Student17 **extends** Person{

**void** display(){

message();//will invoke parent class message() method

}

**public** **static** **void** main(String args[]){

Student17 s=**new** Student17();

s.display();

}

}

Output:welcome

# Instance initializer block:

|  |
| --- |
| **Instance Initializer block** is used to initialize the instance data member. It run each time when object of the class is created. |
| The initialization of the instance variable can be directly but there can be performed extra operations while initializing the instance variable in the instance initializer block. |

#### Que) What is the use of instance initializer block while we can directly assign a value in instance data member? For example:

**class** Bike{

**int** speed=100;

}

## **Why use instance initializer block?**

|  |
| --- |
| Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc. |

### **Example of instance initializer block**

|  |
| --- |
| Let's see the simple example of instance initializer block the performs initialization. |

**class** Bike7{

**int** speed;

    Bike7(){System.out.println("speed is "+speed);}

    {speed=100;}

**public** **static** **void** main(String args[]){

    Bike7 b1=**new** Bike7();

    Bike7 b2=**new** Bike7();

    }

}

Output:speed is 100

speed is 100

|  |
| --- |
| There are three places in java where you can perform operations:   1. method 2. constructor 3. block |

## **What is invoked firstly instance initializer block or constructor?**

**class** Bike8{

**int** speed;

    Bike8(){System.out.println("constructor is invoked");}

    {System.out.println("instance initializer block invoked");}

**public** **static** **void** main(String args[]){

    Bike8 b1=**new** Bike8();

    Bike8 b2=**new** Bike8();

    }

}

Output:instance initializer block invoked

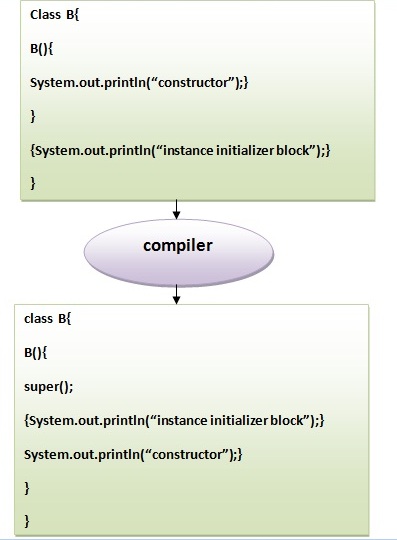
constructor is invoked

instance initializer block invoked

constructor is invoked

|  |
| --- |
| In the above example, it seems that instance initializer block is firstly invoked but NO. Instance intializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement super(). So firstly, constructor is invoked. Let's understand it by the figure given below: |

#### Note: The java compiler copies the code of instance initializer block in every constructor.



## **Rules for instance initializer block :**

|  |
| --- |
| There are mainly three rules for the instance initializer block. They are as follows: |

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

## **Program of instance initializer block that is invoked after super()**

**class** A{

A(){

System.out.println("parent class constructor invoked");

}

}

**class** B2 **extends** A{

B2(){

**super**();

System.out.println("child class constructor invoked");

}

{System.out.println("instance initializer block is invoked");}

**public** **static** **void** main(String args[]){

B2 b=**new** B2();

}

}

Output:parent class constructor invoked

instance initializer block is invoked

child class constructor invoked

## **Another example of instance block**

**class** A{

A(){

System.out.println("parent class constructor invoked");

}

}

**class** B3 **extends** A{

B3(){

**super**();

System.out.println("child class constructor invoked");

}

B3(**int** a){

**super**();

System.out.println("child class constructor invoked "+a);

}

{System.out.println("instance initializer block is invoked");}

**public** **static** **void** main(String args[]){

B3 b1=**new** B3();

B3 b2=**new** B3(10);

}

}

Output:parent class constructor invoked

instance initializer block is invoked

child class constructor invoked

parent class constructor invoked

instance initializer block is invoked

child class constructor invoked 10

# 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.

## **1) Java final variable**

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

### final keyword in java**Example of final variable**

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

**class** Bike9{

**final** **int** speedlimit=90;//final variable

**void** run(){

  speedlimit=400;

 }

**public** **static** **void** main(String args[]){

 Bike9 obj=**new**  Bike9();

 obj.run();

 }

}//end of class

Output:Compile Time Error

## **2) 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();

   }

}

Output:Compile Time Error

## **3) 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** Honda();

  honda.run();

  }

}

Output:Compile Time Error

### **Q) Is final method inherited?**

Ans) 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...

### **Q) 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;

...

}

### Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

**class** Bike10{

**final** **int** speedlimit;//blank final variable

  Bike10(){

  speedlimit=70;

  System.out.println(speedlimit);

  }

**public** **static** **void** main(String args[]){

**new** Bike10();

 }

}

Output:70

### **static blank final variable**

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

### **Example of static blank final variable**

**class** A{

**static** **final** **int** data;//static blank final variable

**static**{ data=50;}

**public** **static** **void** main(String args[]){

    System.out.println(A.data);

 }

}

### Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

**class** Bike11{

**int** cube(**final** **int** n){

   n=n+2;//can't be changed as n is final

   n\*n\*n;

  }

**public** **static** **void** main(String args[]){

    Bike11 b=**new** Bike11();

    b.cube(5);

 }

}

Output:Compile Time Error

### **Q) Can we declare a constructor final?**

No, because constructor is never inherited.