# Inheritance in Java

**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](https://www.javatpoint.com/java-oops-concepts) (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new [classes](https://www.javatpoint.com/object-and-class-in-java) 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](https://www.javatpoint.com/method-overriding-in-java) (so [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) 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

{

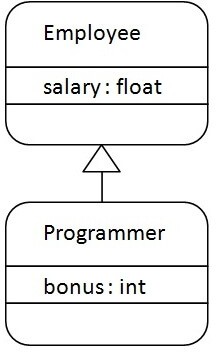
//methods and fields

}

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);

}

}

Programmer salary is:40000.0 Bonus of programmer is:10000

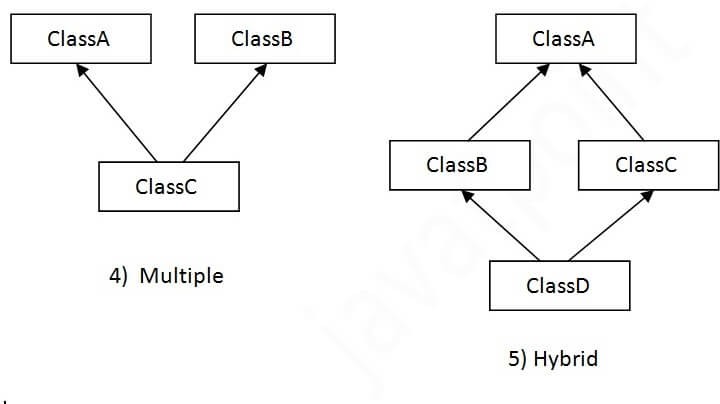
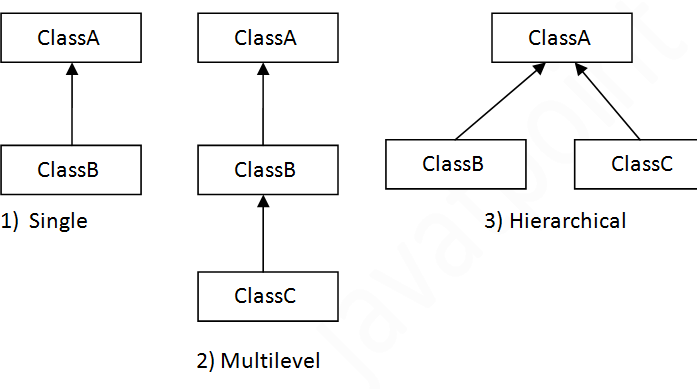
In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

**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:

## 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.

*File: TestInheritance.java*

**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.

*File: TestInheritance2.java*

**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.

*File: TestInheritance3.java*

**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...

## 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

# 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

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

1. **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();

}}

Output:

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();

bark();

}

}

**class** TestSuper2{

**public static void** main(String args[]){ Dog d=**new** Dog();

d.work();

}}

Output:

eating... barking...

In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

## 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:

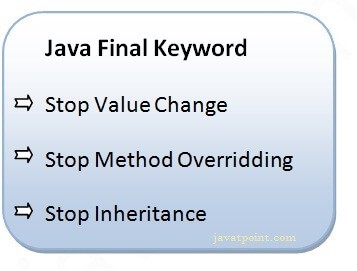
animal is created dog is created

# 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:

* 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.



## Java final variable

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

### 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

## 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

## 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();

}

}

Output:Compile Time Erro

#### 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.

# 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](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/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.

# Uses of 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.

# Declaring 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

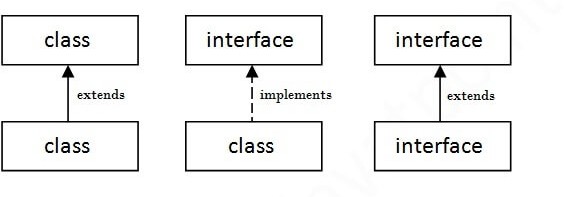
// declare methods that abstract

// by default.

}

# 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.



## Example

interface A

{

void display();

}

class B implements A

{

public void display()

{

System.out.println("Hello");

}

}

class MB

{

public static void main(String args[])

{

B obj = new B(); obj.display();

}

}

Output:

Hello

## Interface Example:

In this example, the interface A has only one method. Its implementation is provided by B and C classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

interface A

{

void display();

}

class B implements A

{

public void display()

{

System.out.println("Display method in B class");

}

}

class C implements B

{

public void display()

{

System.out.println("display method in C class");

}

}

class MainClass

{

public static void main(String args[])

{

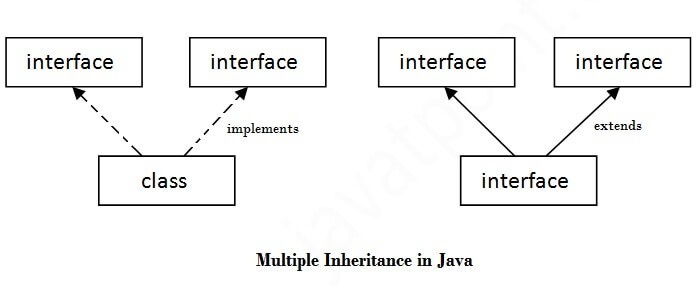
D obj=new D(); obj.draw();

}

}

# Multiple inheritance by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



interface A

{

void display();

}

interface B

{

void show();

}

class C implements A,B

{

public void display()

{

System.out.println("Hello");

}

public void show()

{

System.out.println("Welcome");

}

}

class MainClass

{

public static void main(String args[])

{

C obj = new C(); obj.display(); obj.show();

}

}

Output:

Hello Welcome

# Interface inheritance

A class implements an interface, but one interface extends another interface. interface A

{

void display();

}

interface B extends A

{

void show();

}

class C implements B

{

public void display()

{

System.out.println("Hello");

}

public void show()

{

System.out.println("Welcome");

}

}

Class MainClass

{

public static void main(String args[])

{

C obj = new C(); obj.display(); obj.show();

}

}

Output:

Hello Welcome

# 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

abstract class

{ }

# Abstract Method

A method which is declared as abstract and does not have implementation is known as an abstract method.

## Example

abstract void display(); //no method body and abstract

## Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

abstract class Bike

{

abstract void run();

}

class Honda extends Bike

{

void run()

{

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

}

}

class Car

{

public static void main(String args[])

{

Bike obj = new Honda(); obj.run();

}

}

running safely

# Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| 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 | The interface keyword is used to declare interface. |

|  |  |
| --- | --- |
| abstract class. |  |
| 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: | Example: |
| public abstract class Shape{ | public interface Drawable{ |
| public abstract void draw(); | void draw(); |
| } | } |