## Types of inheritance

Refer following posts:

* [**Types of inheritance in java**](http://beginnersbook.com/2013/05/java-inheritance-types/)
* [**multilevel inheritance**](http://beginnersbook.com/2013/12/multilevel-inheritance-in-java-with-example/)
* [**multiple inheritance**](http://beginnersbook.com/2013/05/java-multiple-inheritance/)
* [**hybrid inheritance**](http://beginnersbook.com/2013/10/hybrid-inheritance-java-program/)
* [**hierarchical inheritance**](http://beginnersbook.com/2013/10/hierarchical-inheritance-java-program/)

#### Constructors and Inheritance

The [**constructor**](http://beginnersbook.com/2013/03/constructors-in-java/) in the superclass is responsible for building the object of the superclass and the constructor of the subclass builds the object of subclass. When the subclass constructor is called during object creation, it by default invokes the default constructor of super-class. Hence, in inheritance the objects are constructed top-down. The superclass constructor can be called explicitly using the keyword super, but it should be first statement in a constructor. The keyword super always refers to the superclass immediately above of the calling class in the hierarchy. The use of multiple super keywords to access an ancestor class other than the direct parent is illegal.

class Shape {

private int length;

private int breadth;

public int getBreadth() {

return breadth;

}

public int getLength() {

return length;

}

public void setBreadth(int i) {

breadth = i;

}

public void setLength(int i) {

length = i;

}

// default Constructor

Shape() {

length = 0;

breadth = 0;

System.out.println("Inside default constructor of Shape ");

}

// Parameterized Constructor

Shape(int len, int bdth) {

length = len;

breadth = bdth;

System.out.println("Inside constructor of Shape ");

System.out.println("length : " + length);

System.out.println("breadth : " + breadth);

}

}

// A subclass which extends for shape

class Rectangle extends Shape {

private String type;

// default Constructor

Rectangle() {

super();

type = null;

System.out.println("Inside default constructor of rectangle ");

}

// Parameterized Constructor

Rectangle(String ty, int len, int bdth) {

super (len, bdth);

System.out.println("Inside constructor of rectangle ");

System.out.println("length : " + len);

System.out.println("breadth : " + bdth);

System.out.println("type : " + type);

}

public String getType() {

return type;

}

public void setType(String string) {

type = string;

}

}

// A subclass which extends for rectangle

class ColoredRectangle extends Rectangle {

private String color;

/\* default Constructor\*/

ColoredRectangle() {

super();

color = null;

System.out.println("Inside default constructor of coloredRectangle");

}

// Parameterized Constructor

ColoredRectangle(String c, String ty, int len, int bdth) {

super (ty, len, bdth);

System.out.println("Inside constructor of coloredRectangle ");

System.out.println("length : " + len);

System.out.println("breadth : " + bdth);

System.out.println("type : " + ty);

}

public String getColor() {

return color;

}

public void setColor(String string) {

color = string;

}

}

public class Test {

public static void main(String args[]) {

ColoredRectangle CR = new ColoredRectangle();

ColoredRectangle CR2 = new ColoredRectangle("Red","Big", 5, 2 );

}

}

**The output is:**

Inside default constructor of Shape

Inside default constructor of rectangle

Inside default constructor of coloredRectangle

Inside constructor of Shape

length : 5

breadth : 2

Inside constructor of rectangle

length : 5

breadth : 2

type : null

Inside constructor of coloredRectangle

length : 5

breadth : 2

type : Big

#### Inheritance and Method Overriding

By using super we can access the [**overridden method**](http://beginnersbook.com/2014/01/method-overriding-in-java-with-example/) in the super class.

class Shape {

private int length;

private int breadth;

// default Constructor

Shape() {

length = 0;

breadth = 0;

}

// Parameterized Constructor

Shape(int len, int bdth) {

length = len;

breadth = bdth;

}

void showattributes() {

System.out.println("length : " + length);

System.out.println("breadth : " + breadth);

}

}

// A subclass which extends for shape

class Rectangle extends Shape {

private String type;

/\* default Constructor

\*/

Rectangle() {

type = null;

}

// Parameterized Constructor

Rectangle(String ty, int len, int bdth) {

super(len,bdth);

type = ty;

}

void showattributes() {

// showattributes() of class Shape is called

super.showattributes();

System.out.println("type : " + type);

}

}

public class Test {

public static void main(String args[]) {

Rectangle rect = new Rectangle("Blue",5,7);

// showattributes() in rectangle is called

rect.showattributes();

}

}

The output is :

length : 5

breadth : 7

type : Blue

#### Inheritance & Abstract Classes

The superclasses are more general than their subclasses. Usually, the superclasses are made abstract so that the objects of its prototype cannot be made. So the objects of only the subclasses can be used. To make a class abstract, the keyword abstract is used in the class definition.

Abstract methods are methods which do not have method statements. The subclasses provides the method statements. The methods provided by the superclass needs to be overridden by the subclass. The class that has at least one abstract method should be made abstract. The [**abstract class**](http://beginnersbook.com/2013/05/java-abstract-class-method/) can not be instantiated because it does not define a complete implementation.

public abstract class {

….

}

**Using Final with methods**: We can prevent a method from being overridden by using the keyword final at the start of its declaration. Final methods can not be overridden.

public abstract void methodname();

class Shape {

final void showattributes() {

System.out.println("Inside class shape ");

}

}

// A subclass which extends for shape

class Rectangle extends Shape {

void showattributes() { // Cannot override the final method

System.out.println("Inside class rectangle");

}

}

The method showattributes() cannot be overridden in the class rectangle because it is declared as final in class shape. It shows an error when we try to override it.

**Using Final with class**: We can also prevent inheritance by making a class final. When a class is declared as final, its methods also become final. An abstract class cannot be declared as final because an abstract class is incomplete and its subclasses need to provide the implementation.

final class shape {

void showattributes() {

System.out.println("Inside class shape ");

}

}

/\* A subclass which extends for shape

\*/

class rectangle extends shape {

// The type rectangle cannot subclass the final class shape

void showattributes() {

System.out.println("Inside class rectangle");

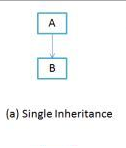
}

}

The class shape cannot be inherited because it is declared as final. It will show an error when we try to inherit it.

### 1) Single Inheritance

**Single inheritance** is damn easy to understand. When a class extends another one class only then we  call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a**parent class** of B and B would be  a **child class** of A.

**Single Inheritance example program in Java**

Class A

{

public void methodA()

{

System.out.println("Base class method");

}

}

Class B extends A

{

public void methodB()

{

System.out.println("Child class method");

}

public static void main(String args[])

{

B obj = new B();

obj.methodA(); //calling super class method

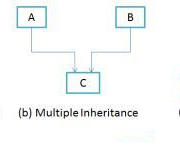
obj.methodB(); //calling local method

}

}

### 2) Multiple Inheritance

“**Multiple Inheritance**” refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with “multiple inheritance” is that the derived class will have to manage the dependency on two base classes.

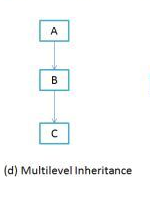
**[](http://beginnersbook.com/wp-content/uploads/2013/05/Multiple-Inheritance.png)**

Note 1: Multiple Inheritance is very rarely used in software projects. Using Multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.

Note 2: Most of the new OO languages like **Small Talk, Java, C# do not support Multiple inheritance**. Multiple Inheritance is supported in C++.

### 3) Multilevel Inheritance

**Multilevel inheritance** refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A

**[](http://beginnersbook.com/wp-content/uploads/2013/05/Multilevel-Inheritance.png)**

**Multilevel Inheritance example program in Java**

Class X

{

public void methodX()

{

System.out.println("Class X method");

}

}

Class Y extends X

{

public void methodY()

{

System.out.println("class Y method");

}

}

Class Z extends Y

{

public void methodZ()

{

System.out.println("class Z method");

}

public static void main(String args[])

{

Z obj = new Z();

obj.methodX(); //calling grand parent class method

obj.methodY(); //calling parent class method

obj.methodZ(); //calling local method

}

}

It’s pretty clear with the diagram that in Multilevel inheritance there is a concept of grand parent class. If we take the example of above diagram then class C inherits class B and class B inherits class A which means B is a parent class of C and A is a parent class of B. So in this case class C is implicitly inheriting the properties and method of class A along with B that’s what is called multilevel inheritance.

#### Example Program:

In this example we have three classes –  Car, Maruti and Maruti800. We have done a setup – class Maruti extends Car and class Maurit800 extends Maurti. With the help of this Multilevel hierarchy setup our Maurti800 class is able to use the methods of both the classes (Car and Maruti).

package beginnersbook.com;

class Car{

public Car()

{

System.out.println("Class Car");

}

public void vehicleType()

{

System.out.println("Vehicle Type: Car");

}

}

class Maruti extends Car{

public Maruti()

{

System.out.println("Class Maruti");

}

public void brand()

{

System.out.println("Brand: Maruti");

}

public void speed()

{

System.out.println("Max: 90Kmph");

}

}

public class Maruti800 extends Maruti{

public Maruti800()

{

System.out.println("Maruti Model: 800");

}

public void speed()

{

System.out.println("Max: 80Kmph");

}

public static void main(String args[])

{

Maruti800 obj=new Maruti800();

obj.vehicleType();

obj.brand();

obj.speed();

}

}

Output:

Class Car

Class Maruti

Maruti Model: 800

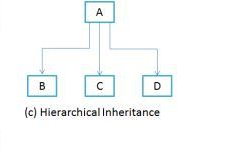
Vehicle Type: Car

Brand: Maruti

Max: 80Kmph

### 4) Hierarchical Inheritance

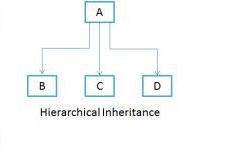
In such kind of inheritance one class is inherited by many**sub classes**. In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D.

**[](http://beginnersbook.com/wp-content/uploads/2013/05/Hierarchical-Inheritance.png)**

**Does Java supports hierarchical inheritance?**

Yes, It is quite possible to have hierarchical inheritance in java.

**What is hierarchical inheritance?**



As you can see in the above diagram that when a class has more than one child classes (sub classes) or in other words more than one child classes have the same parent class then such kind of inheritance is known as hierarchical.

**Let’s have a look at the below example program to understand it better –**

I’m using the above figure for implementing hierarchical inheritance in the below example-

Class A

{

public void methodA()

{

System.out.println("method of Class A");

}

}

Class B extends A

{

public void methodB()

{

System.out.println("method of Class B");

}

}

Class C extends A

{

public void methodC()

{

System.out.println("method of Class C");

}

}

Class D extends A

{

public void methodD()

{

System.out.println("method of Class D");

}

}

Class MyClass

{

public void methodB()

{

System.out.println("method of Class B");

}

public static void main(String args[])

{

B obj1 = new B();

C obj2 = new C();

D obj3 = new D();

obj1.methodA();

obj2.methodA();

obj3.methodA();

}

}

The above would run perfectly fine with no errors and the **output** would be –  
method of Class A  
method of Class A  
method of Class A

### 5) Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of**Single** and **Multiple inheritance.** A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces. yes you heard it right. By using **interfaces** you can have multiple as well as **hybrid inheritance** in Java.

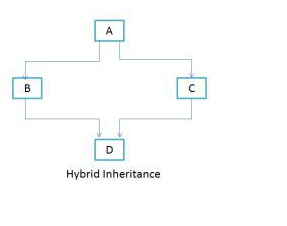
**Does java supports hybrid inheritance?**

Yes and No. If you are using only classes then this is not allowed in java, however using interfaces it’s possible to have [**hybrid inheritance in java**](http://beginnersbook.com/2013/05/java-inheritance-types/). We will see this in below example programs.

**Hierarchical inheritance and Hybrid inheritance are different?**

Yes, Hierarchical inheritance is different than hybrid inheritance. Hierarchical inheritance is possible to have in java even using the classes alone itself as in this type of inheritance two or more classes have the same parent class or in other words a single parent class has two or more child classes, which is quite possible to have in java.

## Hybrid Inheritance in Java



As you can see in the above diagram that it’s a combine form of single and multiple inheritance. Since java doesn’t support multiple inheritance, the hybrid inheritance is also not possible.

Case 1:  Using classes: If in above figure B and C are classes then this inheritance is not allowed as a single class cannot extend more than one class (Class D is extending both B and C). Reason explained below!!

Case 2: Using Interfaces: If B and C are interfaces then the above hybrid inheritance is allowed as a single class can implement any number of interfaces in java.

**Let’s understand the above concept with the help of examples:**

**Example program 1: Using classes to form hybrid**

public class A

{

public void methodA()

{

System.out.println("Class A methodA");

}

}

public class B extends A

{

public void methodA()

{

System.out.println("Child class B is overriding inherited method A");

}

public void methodB()

{

System.out.println("Class B methodB");

}

}

public class C extends A

{

public void methodA()

{

System.out.println("Child class C is overriding the methodA");

}

public void methodC()

{

System.out.println("Class C methodC");

}

}

public class D extends B, C

{

public void methodD()

{

System.out.println("Class D methodD");

}

public static void main(String args[])

{

D obj1= new D();

obj1.methodD();

obj1.methodA();

}

}

**Output:**

Error!!

**Why?**Most of the times you will find the following explanation of above error – Multiple inheritance is not allowed in java so class D cannot extend two classes(B and C).  **But do you know why it’s not allowed?**Let’s look at the above code once again, In the above program class B and C both are extending class A and they both have overridden the methodA(), which they can do as they have extended the class A. But since both have different version of methodA(), **compiler is confused** which one to call when there has been a call made to methodA() in child class D (child of both B and C, it’s object is allowed to call their methods), this is a ambiguous situation and to avoid it, such kind of scenarios are not allowed in java. In C++ it’s allowed.

**What’s the solution? Hybrid inheritance implementation using interfaces.**

interface A

{

public void methodA();

}

interface B extends A

{

public void methodB();

}

interface C extends A

{

public void methodC();

}

class D implements B, C

{

public void methodA()

{

System.out.println("MethodA");

}

public void methodB()

{

System.out.println("MethodB");

}

public void methodC()

{

System.out.println("MethodC");

}

public static void main(String args[])

{

D obj1= new D();

obj1.methodA();

obj1.methodB();

obj1.methodC();

}

}

Output:

MethodA

MethodB

MethodC

Note: Even though class D didn’t implement interface “A” still we have to define the methodA() in it. It is because interface B and C extends the interface A.

The above code would work without any issues and that’s how we implemented hybrid inheritance in java using interfaces.

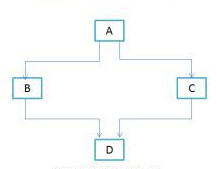
# Does Java support Multiple inheritance?

Earlier we discussed [**types of inheritance**](http://beginnersbook.com/2013/05/java-inheritance-types/). In this post we will discuss why java doesn’t support **multiple inheritance**. We would also see how to achieve this using [**interfaces in Java**](http://beginnersbook.com/2013/05/java-interface/).

### Why Java doesn’t support multiple inheritance?

C++ , Common lisp and few other languages supports multiple inheritance while java doesn’t support it. It is just to **remove ambiguity**, because**multiple inheritance** can cause ambiguity in few scenarios. One of the most common scenario is **Diamond problem.**

**What is diamond problem?**  
Consider the below diagram which shows multiple inheritance as Class D extends both Class B & C. Now lets assume we have a method in class A andclass B & C overrides that method in their own way. **Wait!! here the problem comes**– Because D is extending both B & C so if D wants to use the same method which method would be called (the overridden method of B or the overridden method of C). Ambiguity. That’s the main reason why Java doesn’t support multiple inheritance.



### How to achieve multiple inheritance in Java using interfaces?

interface X

{

public void myMethod();

}

interface Y

{

public void myMethod();

}

class Demo implements X, Y

{

public void myMethod()

{

System.out.println(" Multiple inheritance example using interfaces");

}

}

As you can see that the class implemented two interfaces. A class can implement any number of interfaces. In this case there is no ambiguity even though both the interfaces are having same method. Why? Because methods in an interface are always [**abstract**](http://beginnersbook.com/2013/05/java-abstract-class-method/) by default, which doesn’t let them to give their implementation (or method definition ) in interface itself.

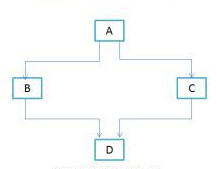
# Does Java support Multiple inheritance?

 Earlier we discussed [**types of inheritance**](http://beginnersbook.com/2013/05/java-inheritance-types/). In this post we will discuss why java doesn’t support **multiple inheritance**. We would also see how to achieve this using [**interfaces in Java**](http://beginnersbook.com/2013/05/java-interface/).

### Why Java doesn’t support multiple inheritance?

C++ , Common lisp and few other languages supports multiple inheritance while java doesn’t support it. It is just to **remove ambiguity**, because**multiple inheritance** can cause ambiguity in few scenarios. One of the most common scenario is **Diamond problem.**

**What is diamond problem?**  
Consider the below diagram which shows multiple inheritance as Class D extends both Class B & C. Now lets assume we have a method in class A andclass B & C overrides that method in their own way. **Wait!! here the problem comes**– Because D is extending both B & C so if D wants to use the same method which method would be called (the overridden method of B or the overridden method of C). Ambiguity. That’s the main reason why Java doesn’t support multiple inheritance.



### How to achieve multiple inheritance in Java using interfaces?

interface X

{

public void myMethod();

}

interface Y

{

public void myMethod();

}

class Demo implements X, Y

{

public void myMethod()

{

System.out.println(" Multiple inheritance example using interfaces");

}

}

As you can see that the class implemented two interfaces. A class can implement any number of interfaces. In this case there is no ambiguity even though both the interfaces are having same method. Why? Because methods in an interface are always [**abstract**](http://beginnersbook.com/2013/05/java-abstract-class-method/) by default, which doesn’t let them to give their implementation (or method definition ) in interface itself.