**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** in java is used to add additional functionalities to the existing class. Inheritance is used to extend the present class by adding some more properties to it. Inheritance is used to reuse the present tried and tested code so that you may not have to write them and compile them again.

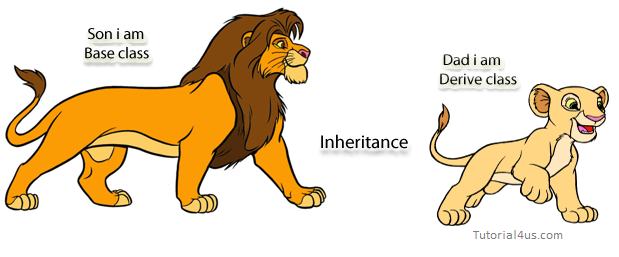
In inheritance the class which is giving properties is called as **parent class or super class or base class**

In inheritance the class which is taking properties is called as **child class or sub class or derived class**

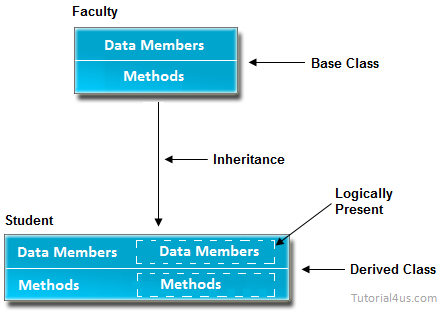
Inheritance represents the **IS-A relationship**, also known as “**parent-child relationship”**.

### Real life example of inheritance

The real life example of inheritance is child and parents, all the properties of father are inherited by his son.



**The following diagram use view about inheritance.**



In the above diagram data members and methods are represented in broken line are inherited from faculty class and they are visible in student class logically.

### 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*

*}*

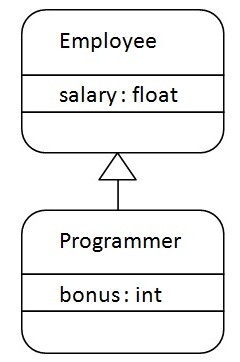
**Disadvantages of Inheritance**

1. Both classes (super and subclasses) are tightly-coupled.
2. As they are tightly coupled (binded each other strongly with extends keyword), they cannot work independently of each other.
3. Changing the code in super class method also affects the subclass functionality.
4. If super class method is deleted, the code may not work as subclass may call the super class method with super keyword. Now subclass method behaves independently.

**Java – Inheritance Basics :**

| **No** | **Term** | **Definition** |
| --- | --- | --- |
| 1 | **Inheritance** | **Inheritance** is a process where one object acquires the properties of another object |
| 2 | **Subclass** | Class which inherits the properties of another object is called as **subclass** |
| 3 | **Superclass** | Class whose properties are inherited by subclass is called as **superclass** |
| 4 | **Keywords Used** | extends and implements |

### Understanding the simple example of inheritance



Programmer is the subclass and Employee is the super-class. 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);*

*}*

*}*

***Output:***

*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**

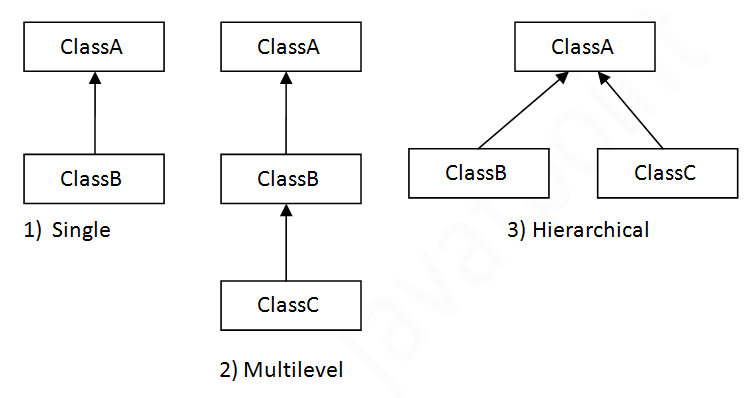
On the basis of class, there can be three types of inheritance in java:

***single,***

***multilevel and***

***hierarchical.***

In java programming, multiple and hybrid inheritance is supported through interface only.



#### Note: Multiple inheritance is not supported in java through class.

When a class extends multiple classes i.e. known as multiple inheritance.

**For Example:**

## multiple inheritance in java

1. S**ingle level inheritance**

 Single level inheritance means we contain 1 parent class and 1 child class.

2. **Multi level inheritance**

Multi level inheritance means we contain 1 parent class and many child classes with intermediate classes.

Intermediate classes means which sometimes work like parent class and some times like child classes.

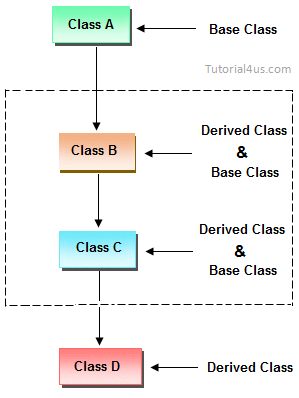


In Multilevel inheritances there exists single base class, single derived class and multiple intermediate base classes.

**Single base class + single derived class + multiple intermediate base classes.**

### Intermediate base classes

An intermediate base class is one in one context with access derived class and in another context same class access base class.



3. **Hierarchical inheritance**



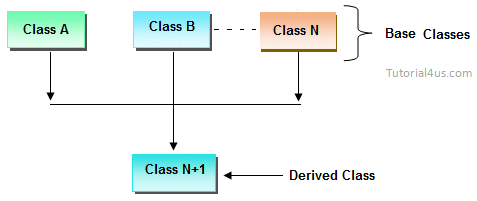
Hierarchical inheritance means we contain 1 parent class and many child classes without any intermediate classes.

4. **Multiple inheritance**



Multiple inheritance means we contain multiple parent classes and one child classes. but this multiple inheritance is not supported in java.

In multiple inheritance there exist multiple classes and singel derived class.



The concept of multiple inheritance is not supported in java through concept of classes but it can be supported through the concept of interface.

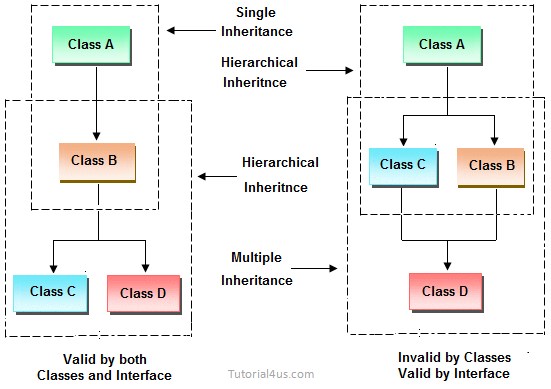
**Note:**

In Java multiple inheritance is not supported using classes but supported using interfaces.

**Note:**

**Combination of any inheritance type**

In the combination if one of the combination is multiple inheritance then the inherited combination is not supported by java through the classes concept but it can be supported through the concept of interface.



## 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*

*//CE: Syntax error, insert "ClassBody" to complete ClassDeclaration*

*Public Static* ***void*** *main(String args[]){*

*C obj=****new*** *C();*

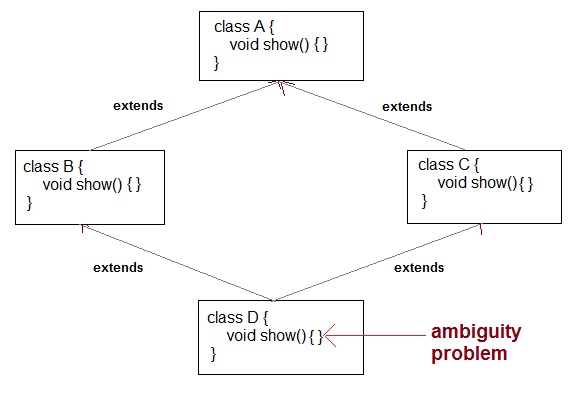
*obj.msg();*

*//Now which msg() method would be invoked?*

*}*

*}*

* To remove ambiguity.
* To provide more maintainable and clear design.



**//wap to demo on inheritance**

*class Parent{*

*int a = 10;*

*int b = 20;*

*void show(){*

*System.out.println("a="+a+"\tb="+b);*

*}*

*}*

*class Child extends Parent{*

*int c = 30;*

*int d = 40;*

*void display(){*

*a=300;*

*b=400;*

*System.out.println("a="+a+"\tb="+b);*

*System.out.println("c="+c+"\td="+d);*

*}*

*}*

*class Inheritance1{*

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

*Parent p = new Parent();*

*p.show();*

*//p.display();-invalid*

*Child c = new Child();*

*c.show();*

*c.display();*

*c.show();*

*}*

*}*

**Note:**

1. If we create an object for parent class then we can access only the members of Parent class

2. But if we create an object for child class then we can access the members of Both Parent class and Child class

**Rules**

1. In java we can create a class that extends only one class and we can not create a class that extends more than one class because java does not support multiple inheritance.

*class A{*

*}*

*class B{*

*}*

*class C extends A,B{ X-invalid*

*}*

1. In java cyclic inheritance is not supported.

*class Student extends Employee{ X-invalid & class Employee extends Employee{ X-invalid*

*}*

*}*

*class Employee extends Student{*

*}*

1. In java every predefined class or user defined class are child classes of Object class either directly or indirectly so that all members of the Object class we can directly use in any class.
2. When we declare a class and we are not extending from any class then compiler will create the class by extending from Object class automatically.

But we declare a class by extending from any one class then compiler wont extend from Object class, hence we call

Object class as java's super most class

*class A{ ----> class A extends java.lang.Object{*

*} }*

*class B extends A{ ----> class B extends A{*

*} }*

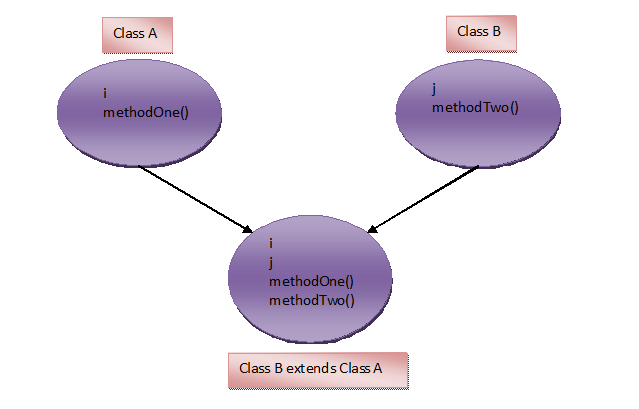
Inheritance in java is implemented by using **extend** keyword like below,

|  |
| --- |
| *class A{*  *int i;*  *void methodOne(){*  *System.out.println("From methodOne");*  *}*  *}*  *class B extends A{*  *int j;*  *void methodTwo(){*  *System.out.println("From methodTwo");*  *}*  *}* |

Here, **Class A** has two members – **‘i’** and **‘methodOne()’**. **Class B** also has two members – **‘j’** and **‘methodTwo()’**. **Class B** is extending **Class A**. Therefore, members of **class A** are inherited to **Class B**. Now, **Class B** will have two additional members inherited from **class A** along with its members. Have a look at the below diagram,

|  |
| --- |
| *class A{*  *int i;*  *void methodOne(){*  *System.out.println("From methodOne");*  *}*  *}*  *class B extends A{*  *int j;*  *void methodTwo(){*  *System.out.println("From methodTwo");*  *}*  *}* |

Here, **Class A** has two members – **‘i’** and **‘methodOne()’**. **Class B** also has two members – **‘j’** and **‘methodTwo()’**. **Class B** is extending **Class A**. Therefore, members of **class A** are inherited to **Class B**. Now, **Class B** will have two additional members inherited from **class A** along with its members. Have a look at the below diagram,



Here, **Class A** is called **super class** and **Class B** is called **sub class**. Here, you can analyse the inheritance like this – we are extending **class A** by adding additional properties to it through **Class B** or We are reusing properties of **Class A** in **Class B**.

## Points-To-Remember About Inheritance In Java :

Here are some points regarding inheritance in java.

* Constructors, SIB – Static Initialization Block and IIB – Instance Initialization Block of super class will not be inheriting to its sub class. But they are executed while creating an object to sub class. For example,

|  |
| --- |
| *class A{*  *int i;*  *static{*  *System.out.println("Class A SIB");*  *}*  *{*  *System.out.println("Class A IIB");*  *}*  *A(){*  *System.out.println("Class A Constructor");*  *}*  *}*  *class B extends A{*  *int j;*  *}*  *class MainClass{*  *public static void main(String[] args){*  *B b = new B();*  *}*  *}* |

In the above example, **Class B** is extending **Class A**. In the **MainClass**, We are creating an object to **Class B**. While creating this object, SIB, IIB and constructor of **Class A** are also executed. The output of the above program will be,

**Output :**  
*Class A SIB  
Class A IIB  
Class A Constructor*

* Static members of super class are inheriting to sub class as static members and non-static members are inheriting as non-static members only.

**Try to compile the following program,**

|  |
| --- |
| *class A{*  *int i;*  *A(int i){*  *System.out.println("Class A Constructor");*  *}*  *}*  *class B extends A{*  *int j;*  *}* |

You will get a compile time error saying implicit default constructor A() is undefined for Class A. Compiler will force you to write constructor in Class B. Because, we are not defining constructor for Class B. So, compiler will be providing default constructor. In that default constructor, first statement is super() – it is a calling statement to default constructor of Class A. But it is not defined in Class A. Therefore you will get a compile time error. To avoid this error, write the constructor for sub class. From that constructor call super class constructor explicitly. See the below code,

|  |
| --- |
| *class A{*  *int i;*  *A(int i){*  *System.out.println("Class A Constructor");*  *}*  *}*  *class B extends A{*  *int j;*  *public B(){*  *super(10);     //Explicitly Calling Class A constructor*  *System.out.println("Class B Constructor");*  *}*  *}* |

By default, every class is a sub class of **java.lang.Object** class. So, every class in java has properties inherited from Object class. Look at the below code,

|  |
| --- |
| *class A*  *{*  *//some satements*  *}* |

* Compiler will treat the above code as,

|  |
| --- |
| *class A extends Object*  *{*  *//some satements*  *}* |

* Any class can not extend itself i.e

|  |
| --- |
| *class A extends A*  *{*  *//It gives compile time error*  *}* |

* We can call super class constructor explicitly through **super()** calling statement from sub class constructor and we can call other constructors of the same class through **this()** calling statement but, we can’t call sub class constructor from super class constructor.

|  |
| --- |
| *class A{*  *A(){*  *//B();      There is no statement in java to call subclass constructor*  *System.out.println("Class A Constructor");*  *}*  *}*  *class B extends A{*  *B(){*  *super();        // calling statement to super class constructor*  *System.out.println("Class B Constructor");*  *}*  *}* |

## ****Effect of private, default, protected and public keyword on inheritance in java:****

* **private :**Private members can not be inherited to sub class.
* **Default** : Default members can be inherited to sub class within package.
* **protected** : protected members can be inherited to any sub class but usage of protected member is limited within package.
* **public** : public members are inherited to all sub classes.

**Let’s discuss this concept with example,**

|  |
| --- |
| *package com1;*  *public class A{*  *private int i;*  *int j;     protected int k;*  *public int m;*  *}*  *class B extends A{*  *void methodOfClassB(){*  *//****System.out.println(i);        Private member can not be inherited***  *System.out.println(j);* ***//Default member can be inherited within package***  *System.out.println(k);* ***//protected member can be inherited to any subclass***  *System.out.println(m);* ***//public member can be inherited to all sub classes***  *}*  *}*  *class C extends B{*  *void methodOfClassC(){*  *System.out.println(j);* ***//Default member can be inherited within package***  *System.out.println(k);* ***//protected member can be inherited to any subclass***  *System.out.println(m);****//public member can be inherited to any subclass***    *B b = new B();*  *System.out.println(b.j);* ***//Default member can be used within package***  *System.out.println(b.k);* ***//Protected member can be used anywhere in the package***  *System.out.println(b.m);* ***//Public member can be used anywhere***  *}*  *}*  *package com2;*  *import com1.A;*  *public class D extends A{*  *void methodOfClassD()    {*  *//System.out.println(j);* ***Default members can not be inherited outside package***  *System.out.println(k);* ***//Protected member can be inherited to any subclass***  *System.out.println(m);* ***//public member is always inherited to any subclass***    *A a = new A();*  *//System.out.println(a.i);* ***private member not visible outside the class***  *//System.out.println(a.j);* ***Default members are not visible outside package***  *//System.out.println(a.k);* ***Protected member can not be used outside the package.***  *System.out.println(a.m);****//public member can be used anywhere***  *}*  *}*    *class E extends D{*  *void methodOfClassE(){*  *System.out.println(k);****//Protected member can be inherited to any subclass***  *System.out.println(m);* ***//public member is always inherited***    *D d = new D();*  *//System.out.println(d.k);****Protected member can not be used outside the package.***  *System.out.println(d.m);****//public member can be used anywhere***  *}*  *}* |

**Note:**

**Hybrid Inheritance :**   
It is a combination of above types of inheritance.

There exist one more type of inheritance – **Multiple Inheritance.**

**Multiple Inheritance :** One class extends more than one class.

But, **Multiple Inheritance is not supported in java.** To avoid the ambiguity, complexity and confusion, multiple inheritance is not supported in java.

**Look at the below example,**

|  |
| --- |
| *class A{*  *void methodOne(){*  *System.out.println("From methodOfClassA");*  *}*  *}*  *class B{*  *void methodOne(){*  *System.out.println("From methodOfClassB");*  *}*  *}*  *class C extends A, B (If it is supported){*  *//Both the methods with same name are inherited to Class B*  *//This causes ambiguity and confusion. Therefore,*  *//Multiple Inheritance is not supported in java*  *}* |

In the above example, **Class A** also has **methodOne()** and **Class B** also has **methodOne()**. **Class C** is extending both the classes. So both the methods with same name are inheriting to **Class C**. It causes confusion and ambiguity for which method to use. Therefore, To avoid this, multiple inheritance is not supported in java.

**Types of relation ships**

If we want to access the information of one class inside another class then we can use any one of the following 2 relationships.

1. **Is-A relation**

If class is extending from another class then it is called as Is-A relation.

Here we can access class1 information inside the class2 directly without creating an object for class1.

**Eg:**

*class Person{*

*}*

*class Employee extends Person{*

*}*

*class Student extends Person{*

*}*

2. **Has-A relation**

If a class contains other class object then it is called as Has-A relation.

Here we can access class1 information inside the class2 only by using object of class1.

**Eg:**

*class Heart{*

*}*

*class Person{*

*Heart h = new Heart();*

*}*

**IS-A Relationship with Example :**

IS-A is a way of saying: This object is a type of that object.

***public******class*** *Vehicle{*

*}*

***public******class*** *FourWheeler* ***extends*** *Vehicle{*

*}*

***public******class*** *TwoWheeler* ***extends*** *Vehicle{*

*}*

***public******class*** *WagonR* ***extends*** *FourWheeler{*

*}*

**Conclusions from above Example :**

**From the above example:**

1. Vehicle is the **superclass**of TwoWheeler class.
2. Vehicle is the **superclass**of FourWheeler class.
3. TwoWheeler and FourWheeler are **sub classes** of Vehicle class.
4. WagonR is the **subclass**of both FourWheeler and Vehicle classes.

**IS-A relationship of above example is –**

*TwoWheeler IS-A Vehicle*

*FourWheeler IS-A Vehicle*

*WagonR IS-A FourWheeler*

Hence

*WagonR IS-A Vehicle*

**Java instanceOf keyword**

**Example :** IS-A Relationship Verification

***class*** *FourWheeler* ***extends*** *Vehicle{}*

***class*** *TwoWheeler* ***extends*** *Vehicle{}*

***class*** *WagonR* ***extends*** *FourWheeler{}*

***public******class*** *Vehicle {*

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

*FourWheeler v1 =* ***new*** *FourWheeler();*

*TwoWheeler v2 =* ***new*** *TwoWheeler();*

*WagonR v3 =* ***new*** *WagonR();*

***System****.out.println(v1* ***instanceof*** *Vehicle);*

***System****.out.println(v2* ***instanceof*** *Vehicle);*

***System****.out.println(v3* ***instanceof*** *Vehicle);*

***System****.out.println(v3* ***instanceof*** *FourWheeler);*

*}*

*}*

Output :

*true*

*true*

*true*

*true*

## InstanceOf Operator :

In the above example we have used this following statement –

***System****.out.println(v1* ***instanceof*** *Vehicle);*

The instanceof operator is used to check whether TwoWheeler is actually a Vehicle, and FourWheelet is actually a Vehicle.

### Invalid Use of InstanceOf Operator :

*System.out.println(v3 instanceof TwoWheeler);*

**Java extends keyword**

Suppose we are having the one already existing class, we call it as “**Parent Class**”. Suppose we need to create another class having exactly same attributes to that of the [parent class and some extra attributes](http://c4learn.com/javaprogramming/is-a-relationship-java-inheritance-tutorial/) then we use “**extend**” keyword to extend the class.

**Extends Keyword :**

We can extend a class by using the **extends keyword in a class declaration**after the class name and before the parent class.

***public******class*** *ParentClass {*

*}*

and we define child class like shown below –

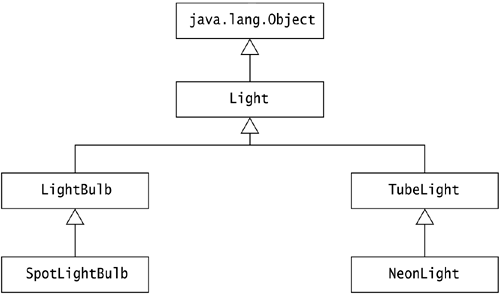
***public******class*** *ChildClass* ***extends*** *ParentClass {*

*}*

| Type | **Name** | **Explanation** |
| --- | --- | --- |
| Super Class | ParentClass | The class from which another class is derived is called the **superclass** |
| Sub Class | ChildClass | The derived class (the class that is derived from another class) is called a **subclass** |

**Important Note : Extends Keyword**

1. In the Java, all the classes are child classes of **java.lang.Object**
2. Object is the Super Class of all the classes in Java
3. Like C++, **multiple inheritance** is not allowed in Java. Class can only extend one class.



**Composition has-a Relationship**

One of the goals of OOP languages is **reusability**. Java gives good importance for reusability.

A method in Java can be called in two ways.

**1. With the objects of the same class (through composition)**

**2. With the objects of other classes (through inheritance)**

Here we discuss the composition.

*class Bird{*

*public void eat(){*

*System.out.println("All birds eat for their metabolic activities");*

*}*

*}*

*public class Parrot{*

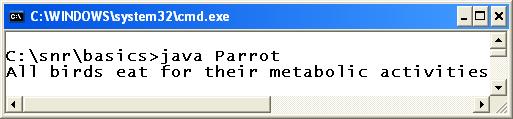
*public static void main(String args[]){*

*Bird b1 = new Bird();*

*b1.eat();*

*}*

*}*



The Parrot class, in its [main()](http://way2java.com/oops-concepts/public-static-void-mainstring-args/) method, created an object of **Bird**and called Bird's method**eat()**. Here, **Bird**and **Parrot**are no way connected or related (I mean, no inheritance). Just they happened to be in the same folder. This feature we call as "**composition**".

**Creating another class object in our class and calling other class methods is known as composition.**

###### **Composition has-a Relationship**

Composition uses the concept called "**has-a**" relationship.

We say, our class **has a** object of other class. Parrot "**has a**" object of **Bird**. Composition is meant for **reusability**.

When compiled **Parrot.java**, we get two .class files – **Bird.class**and**Parrot.class**. These classes work independently of each other even though they are coming from a single source file. Any other third class can make use of the methods of either both or anyone these classes by composition.

**Java HAS-A relationship: aggregation**

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

Consider a situation; Employee object contains much information such as id, name, email etc. It contains one more object named address, which contains its own information such as city, state, country, zip code etc. as given below.

***class*** *Employee{*

***int*** *id;*

*String name;*

*Address address;//****Address is a class***

*...*

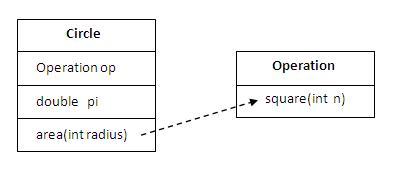
*}*

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 information 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*

Consider we are storing the information of the student, then we may create a class like below –

*Class Student {*

*int roll;*

***String*** *sname;*

*Address address;*

*}*

In the above class, we have entity reference “Address” which stores again its own information like street, city, state, zip. like below –

*Class Address {*

***String*** *street;*

***String*** *state;*

***String*** *zip;*

***String*** *city;*

*}*

So, we can say that **Student HAS-A Address** Thus if the class has entity reference then entity will represent the HAS-A relationship.

Consider the following example –

### Address.java

***public******class*** *Address {*

***String*** *street;*

***String*** *city;*

***String*** *state;*

***String*** *zip;*

***public*** *Address(****String*** *street,* ***String*** *city,* ***String*** *state,* ***String*** *zip) {*

***this****.street = street;*

***this****.city = city;*

***this****.state = state;*

***this****.zip = zip;*

*}*

*}*

### Student.java

***public******class*** *Student {*

*int roll;*

*Address address;*

*Student(int rollNo,Address addressDetail){*

*roll = rollNo;*

*address = addressDetail;*

*}*

*void printStudentDetails(Address address1) {*

***System****.out.println("Roll : " + roll);*

***System****.out.println("Street : " + address1.street);*

***System****.out.println("City : " + address1.city);*

***System****.out.println("State : " + address1.state);*

***System****.out.println("Zip : " + address1.zip);*

*}*

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

*Address address1;*

*address1 =* ***new*** *Address("1-ST","PN","Mah","41");*

*Student s1 =* ***new*** *Student(1,address1);*

*s1.printStudentDetails(address1);*

*}*

*}*

**Output :**

*Roll : 1*

*Street : 1-ST*

*City : PN*

*State : Mah*

*Zip : 41*

**Explanation : Advantages of Using Aggregation**

1. You can see the above code in which we already have the class “Address” which is used to store the details of address. Thus using aggregation we have reused the existing class.
2. [Inheritance](http://c4learn.com/javaprogramming/extends-keyword/) can be achieved only using the [IS-A relationship](http://c4learn.com/javaprogramming/is-a-relationship-java-inheritance-tutorial/) but using this HAS-A we can use existing code more efficiently.

**Java protected access specifier**

***package*** *com.c4learn.inheritance;*

***public******class*** *ProtectedExample {*

***public*** *void publicMethod(){*

*}*

***private*** *void privateMethod(){*

*}*

***protected*** *void protectedMethod(){*

*}*

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

*ChildClass c1 =* ***new*** *ChildClass();*

*c1.testMethods();*

*}*

*}*

***class*** *ChildClass* ***extends*** *ProtectedExample {*

***public*** *void testMethods() {*

*publicMethod();*

*protectedMethod();*

*privateMethod(); //Error*

*}*

*}*

**Output :**

*Compile Time Error*

**Explanation :**

1. From subclass we cannot access the private members of the superclass.
2. subclass you can access its superclass’s public and protected methods and fields
3. If the subclass and the superclass are in the same package, you can also access the superclass’s default methods and fields.

**Aggregation vs Composition**

**Aggregation**and **Composition** are very closely related terms where a novice gets confused and some examples are of good debate too.

**Observe the code.**

*class Test{*

*public void display() { System.out.println("Hello 1");*

*}*

*}*

*class Demo{*

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

*Test t1 = new Test();*

*t1.display();*

*}*

*}*

We say, class **Demo** has the object **t1** of Test class. Or say, Demo "has-a" object of **Test**. This is known as "**has-a**" **relationshi**p and is known as **composition** in Java. This, we know earlier in detail in "[Composition – “has-a” Relationship](http://way2java.com/oops-concepts/composition-has-a-relationship/)".

**Other name of composition is aggregation**.

Between composition and aggregation, a small difference exists. It is simply if **Demo** can exist without **Test**, it is known as aggregation and if **Demo** cannot

exist without **Test**object, it is known as **composition**.

After this introduction of Aggregation vs Composition, let us go with good examples to know better.

**1.** **Heart and human body.**   
Heart exists within a human body. It is an association between two. They are dependent. Can a heart exist without human body? Simply no. It is known as composition. That is, a child cannot exist without parent. Heart is completely controlled human body (organs).

**2.** **Library and book.**   
It is also an association. Can a book exist without library? Yes, it can; I have got only one book in my hand, it is possible. That is, delete (remove) the library, still book can exist. It is an **aggregation**. But a library cannot exist without books. So here, you should consider the direction also. From book to library it is aggregation but from library to book it is composition. Dependency matters.

In UML diagrams we represent aggregation as

[Aggregation vs Composition](http://way2java.com/wp-content/uploads/2013/08/image5.png)

and composition as

[Aggregation vs Composition](http://way2java.com/wp-content/uploads/2013/08/image6.png)

A **restricted aggregation is composition**.

The literal meaning of aggregation is collection of things together. An aggregation is nothing but a collection.

In brief, in aggregation association, child can exist independently without the parent. In composition association, a child cannot exist without parent. Child is completely controlled by parent.

Suppose take another example, say car. A car HAS-A engine, which states in car object we will define engine object.

Composition is a STRONGER relationship whereas Aggregation is a WEAKER relationship. Composition depicts dependency between objects but Aggregation depicts related objects can exist independently.

**Composition and Inheritance**

By virtue of inheritance, the subclass object can call super class methods and variables. But remember, the super class cannot call subclass members. In inheritance subclass is very rich because it can make use of its own methods and also super class methods. Following program explains.

*class Bird{*

*String mouth = "red";*

*public void eat(){*

*System.out.println("All birds eat for their metabolic activities");*

*}*

*}*

*public class Parrot extends Bird{*

*String body = "green";*

*public void food(){*

*System.out.println("Parrot eats seeds and fruits");*

*}*

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

*Parrot p1 = new Parrot();*

***// calling its own members with its object***

*System.out.println(p1.body);*

*p1.food();*

***// calling super class members with its object; inheritance***

*System.out.println(p1.mouth);*

*p1.eat();*

***//calling super class members with super class object; composition***

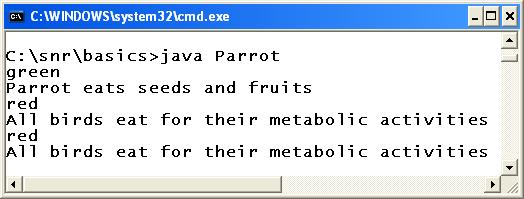
*Bird b1 = new Bird();*

*System.out.println(b1.mouth);*

*b1.eat();*

*}*

*}*



In the above code, the Parrot class uses both composition (as b1.eat()) and inheritance (as p1.eat()) to use super class members.

**Inheritance – "is-a" Relationship**

We know earlier, composition uses "[**has-a**](http://way2java.com/oops-concepts/composition-has-a-relationship/)" relationship. Now inheritance uses "**is-a**" relationship. We can feel as, a super class method is a subclass method; that is, the super class eat() method exists as if in the subclass; thereby the subclass object calls the super class method without any hesitation and straightway. This is known as "**is-a**" relationship. "**is-a**" relationship is achieved through inheritance.

**Constructors in inheritance**

If we have constructor only in child class then when ever we create an object for child class then it will call child class constructor and executed.

But if we have constructors in both parent class and child class and when ever we create an object for child class then first it will invoke child class constructor, child class constructor will invoke parent class constructor and this procedure will be continued up to java's super most class called Object class and next executed.

Invoking of constructors is done from bottom to top i.e., from child class to parent class but execution is done from top to bottom i.e., from parent class to child class

**//wap to demo on constructors in inheritance**

*class Parent{*

*Parent(){*

*System.out.println("Parent class constructor");*

*}*

*}*

*class Child1 extends Parent{*

*Child1(){*

*System.out.println("Child1 class constructor");*

*}*

*}*

*class Child2 extends Child1{*

*Child2(){*

*System.out.println("Child2 class constructor");*

*}*

*}*

*class Inheritance2{*

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

*Child2 c = new Child2();*

*}*

*}*

**o/p:**

*Parent class constructor*

*Child1 class constructor*

*Child2 class constructor*

**Static variables and static methods in inheritance**

If a Parent class contains static variables and static methods then we can access those members directly either by using Parent class name or Child class name.

**Eg:**

*class Parent{*

*static int a = 10;*

*static int b = 20;*

*static void show(){*

*System.out.println("a="+a+"\tb="+b);*

*}*

*}*

*class Child extends Parent{*

*static void display(){*

*a=300;*

*b=400;*

*System.out.println("a="+a+"\tb="+b);*

*}*

*}*

*class Inheritance3{*

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

*Child.show();*

*Child.display();*

*System.out.println("a="+Child.a+"\tb="+Child.b);*

*}*

*}*

M**ain() method in inheritance**

We can also write main() method with in the parent class or child class.

**Eg1:**

**Inheritance4.java**

*class Parent{*

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

*System.out.println("Parent class main() method");*

*}*

*}*

*class Child extends Parent{*

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

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

*}*

*}*

**Compilation**

javac Inheritance4.java

**Execution**

*>java Child*

*Child class main() method*

*>java Parent*

*Parent class main() method*

**Eg2:**

*class Parent{*

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

*System.out.println("Parent class main() method");*

*}*

*}*

*class Child extends Parent{*

*}*

**Execution**

*>java Parent*

*Parent class main() method*

*>java Child*

*Parent class main() method*

**Eg3:**

*class Parent{*

*}*

*class Child extends Parent{*

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

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

*}*

*}*

**Execution**

*>java Child*

*Child class main() method*

*>java Parent*

*NoSuchMethodError: main*

**Example:**

*class CompareEntertainmentsUsingInheritance{  
    public static void main(String arg[]){  
        Movie julai = new Movie();  
        julai.name = "Julai";  
        julai.director = "Trivikram";  
        julai.stuntMaster = "Peter Hein";  
        julai.numberOfArtists = 57;  
        julai.releaseDate = "15-Aug-2012";  
          
        julai.collectionsFirstWeek = 215467.8;  
        julai.collectionsRestOfTheDays = 541132.5;  
          
        Drama ramayan = new Drama();  
        ramayan.name = "Ramayana";  
        ramayan.writer = "Valmiki";  
        ramayan.stageSetter = "Anjaneya";  
        ramayan.numberOfArtists = 200000;  
        ramayan.releaseDate = "17-Mar-1659 BC";  
          
        ramayan.collectionsFirstWeek = 3282937242.86;  
        ramayan.collectionsRestOfTheDays = 93488272349.51;  
          
          
        Circus jumbo = new Circus();  
        jumbo.name = "Jumbo";  
        jumbo.ringMaster = "Antony";  
        jumbo.numberOfArtists = 316;  
        jumbo.releaseDate = "16-Dec-1997";  
          
        jumbo.collectionsFirstWeek = 2123132.21;  
        jumbo.collectionsRestOfTheDays = 234936725.09;  
          
        if((jumbo.getTotalCollections() > julai.getTotalCollections()) && (jumbo.getTotalCollections() > ramayan.getTotalCollections()))  
        {  
            jumbo.print();  
        }  
        else if (julai.getTotalCollections() > ramayan.getTotalCollections())  
        {  
            julai.print();  
        }  
        else  
        {  
            ramayan.print();  
        }      
    }  
}  
class Entertainment  
{  
    String name;  
    int numberOfArtists;  
    String releaseDate;  
    double collectionsFirstWeek;  
    double collectionsRestOfTheDays;  
    double getTotalCollections()    {  
        return collectionsFirstWeek + collectionsRestOfTheDays;  
    }  
    void printEntertainment() {  
        System.out.println( name + " got the following collections " );  
        System.out.println("First Week : " + collectionsFirstWeek);  
        System.out.println("Rest Of The Days : " + collectionsRestOfTheDays);  
        System.out.println("Total Collections : " + getTotalCollections());  
        System.out.println("Total Artists : " + numberOfArtists);  
        System.out.println("Release Date : " + releaseDate);  
    }  
}  
class Movie extends Entertainment{  
    String director;  
    String stuntMaster;  
  
    void print(){  
        printEntertainment();  
        System.out.println("Director : " + director);  
        System.out.println("Stunt Master : " + stuntMaster);  
    }  
}  
class Drama extends Entertainment{  
    String writer;  
    String stageSetter;  
    void print()    {  
        printEntertainment();  
        System.out.println("Writer : " + writer);  
        System.out.println("Stage Setter : " + stageSetter);  
    }  
}  
class Circus extends Entertainment{  
    String ringMaster;  
    void print(){  
        printEntertainment();  
        System.out.println("Ring Master : " + ringMaster);  
    }  
}*

**OUTPUT**

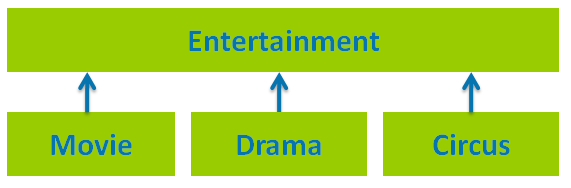
*Ramayana got the following collections  
First Week : 3.28293724286E9  
Rest Of The Days : 9.348827234951E10  
Total Collections : 9.677120959237E10  
Total Artists : 200000  
Release Date : 17-Mar-1659 BC  
Writer : Valmiki  
Stage Setter : Anjaneya*

**DESCRIPTION**

Here we have created a class called Entertainment, where the common variables and methods from related classes Movie, Drama and Circus are included. These classes (Movie, Drama and Circus) inherit those properties (variables and methods) from the class Entertainment. Here Entertainment is the *super-class* and the Movie, Drama and Cirucs are the *sub-classes*. 

In the main method, both the variables belonging the *super-class* and the *sub-class* can be accessed as if they belong to a single class. e.g., On the Drama object ramayan, we are accessing the super-class variables name, numberOfArtists, releaseDate along with the sub-class (Drama) variables writer and stageSetter. There is no difference in the way the super-class variables and the sub-class variables are accessed. 

Similarly there is no difference in accessing the methods belonging to super-class and sub-class. Also observe that we can access the methods from super-class in the sub-class as if they are in the same class. e.g., The printEntertainment() method of Entertainment class is called in the print() method of the Movie class.

The class hierarchy of this program is as shown below.   


**IS-A Relation ship in Java:**

The [Java Class Inheritance](http://java.meritcampus.com/core-java-topics/class-inheritance-in-java-with-example-program) supports the *'is-a relation'*. Every sub-class object is also a super-class object, but every super-class object need not be a sub-class object. e.g., Every Movie is an Entertainment, but every Entertainment need not be a Movie, similarly, every Drama is an Entertainment, but every Entertainment need not be a Drama.

Since every sub-class object is also a super-class object, we can use the sub-class object where we use the super-class object.

*public void printName(Entertainment e)  
{  
    System.out.println("Name of the entertainment is " + e.name);  
}  
Movie businessMan = new Movie();  
businessMan.name = "Business Man";  
businessMan.directorName = "Puri Jagannadh";*

*Drama devdas = new Drama();  
devdas.name = "Devdas";  
  
printName(businessMan);*

***// Calling printName method by passing the Movie object****printName(devdas);*

***// Calling printName method by passing the Drama object***

Here printName is a method which takes any object of type Entertainment as a parameter. Since businessMan is an object ofMovie, so it is also an object of Entertainment, so it can be passed as a parameter for the method printName. Similalry, devdasis also an object of Drama, which in turn is an object of Entertainment, so it can also be passed as a parameter.

*public void printMovie(Movie m){  
    System.out.println(m.directorName + " is the director for movie " + m.name);  
}  
Movie businessMan = new Movie();  
businessMan.name = "Business Man";  
businessMan.directorName = "Puri Jagannadh";  
  
Drama devdas = new Drama();  
devdas.name = "Devdas";  
  
printMovie(businessMan);*

***// Calling printMovie method by passing the Movie object****printMovie(devdas);*

***// THIS WON'T WORK SINCE DRAMA IS NOT A MOVIE***

Here we created a printMovie method, which takes any object of type Movie as a parameter. Since businessMan is an object of type Movie, it can be passed as a parameter. But devdas is an object of type Drama and since Drama is not a Movie, it can not be passed a parameter. Please note that passing parameter of different type will cause a compilation error. The example [Passing Sub Class Object As Super Class Reference](http://java.meritcampus.com/core-java-topics/pass-sub-class-object-as-super-class-reference) shows how the comparison of the entertainments can be done using a method.

**Passing sub class object as a super class reference:**

*class CompareEntertainmentsUsingMethods{  
    public static void main(String s[]) {  
        Movie julai = new Movie();  
        julai.name = "Julai";  
        julai.director = "Trivikram";  
        julai.stuntMaster = "Peter Hein";  
        julai.numberOfArtists = 57;  
        julai.releaseDate = "15-Aug-2012";  
      
        julai.collectionsFirstWeek = 215467.8;  
        julai.collectionsRestOfTheDays = 541132.5;  
      
        Drama ramayan = new Drama();  
        ramayan.name = "Ramayana";  
        ramayan.writer = "Valmiki";  
        ramayan.stageSetter = "Anjaneya";  
        ramayan.numberOfArtists = 200000;  
        ramayan.releaseDate = "17-Mar-1659 BC";  
      
        ramayan.collectionsFirstWeek = 3282937242.86;  
        ramayan.collectionsRestOfTheDays = 93488272349.51;  
      
        Circus jumbo = new Circus();  
        jumbo.name = "Jumbo";  
        jumbo.ringMaster = "Antony";  
        jumbo.numberOfArtists = 316;  
        jumbo.releaseDate = "16-Dec-1997";  
      
        jumbo.collectionsFirstWeek = 2123132.21;  
        jumbo.collectionsRestOfTheDays = 234936725.09;  
      
        compareEntertainments(julai, ramayan, jumbo);  
    }      
    public static void compareEntertainments(Entertainment ent1, Entertainment ent2, Entertainment ent3){  
    if((ent1.getTotalCollections() > ent2.getTotalCollections()) && (ent1.getTotalCollections() > ent3.getTotalCollections()))  
        {  
            System.out.println(ent1.name + " has the highest collections.");  
        }  
        else if (ent2.getTotalCollections() > ent3.getTotalCollections())  
        {  
            System.out.println(ent2.name + " has the highest collections.");  
        }  
        else  
        {  
            System.out.println(ent3.name + " has the highest collections.");  
        }  
    }  
}  
class Entertainment{  
    String name;  
    int numberOfArtists;  
    String releaseDate;  
    double collectionsFirstWeek;  
    double collectionsRestOfTheDays;  
    double getTotalCollections(){  
        return collectionsFirstWeek + collectionsRestOfTheDays;  
    }  
    void printEntertainment(){  
        System.out.println( name + " got the following collections " );  
        System.out.println("First Week : " + collectionsFirstWeek);  
        System.out.println("Rest Of The Days : " + collectionsRestOfTheDays);  
        System.out.println("Total Collections : " + getTotalCollections());  
        System.out.println("Total Artists : " + numberOfArtists);  
        System.out.println("Release Date : " + releaseDate);  
    }  
}  
class Movie extends Entertainment{  
    String director;  
    String stuntMaster;  
    void print(){  
        printEntertainment();  
        System.out.println("Director : " + director);  
        System.out.println("Stunt Master : " + stuntMaster);  
    }  
}  
class Drama extends Entertainment{  
    String writer;  
    String stageSetter;  
    void print(){  
        printEntertainment();  
        System.out.println("Writer : " + writer);  
        System.out.println("Stage Setter : " + stageSetter);  
    }  
}  
class Circus extends Entertainment{  
    String ringMaster;  
    void print(){  
        printEntertainment();  
        System.out.println("Ring Master : " + ringMaster);  
    }  
}*

**OUTPUT**

Ramayana has the highest collections.

**DESCRIPTION**

Here we have created a new method compareEntertainments in the class CompareEntertainmentsUsingMethods. This method takes three parameters and all of them are of type Entertainment. So this method can be used not only to compare Movie, Drama and Circus as we are doing here, it can be also used for comparing Movie, Movie and Drama orCircus, Circus and Circus etc.,

As discussed in [Java Class Inheritance](http://java.meritcampus.com/core-java-topics/class-inheritance-in-java-with-example-program), since every sub-class object is also a super-class object, we can assign a sub-class object to a super-class reference. This is also called as *downcasting*. e.g., the object of type Drama can be assigned to the reference of the type Entertainment.

*Entertainment e;  
Movie m = new Movie();  
e = m;****// LINE A****Drama d = new Drama();  
e = d;****// LINE B*** *m = d;****// LINE C - WON'T WORK, SINCE A DRAMA OBJECT IS NOT OF TYPE MOVIE.***

Here we created an Entertainment reference, and also created a Movie reference and object. At LINE A, we have assigned theMovie reference m to the Entertainment reference e. Similarly, we have created a Drama reference and object and assigned the same to the Entertainment reference e. 

The assignment at LINE C will not work, since we are assigning a reference of type Drama to a reference of type Movie. SinceDrama is not a Movie(i.e. the type of Drama and Movie are different), the assignment causes a compilation error. 

Look at [Assigning Super Class Reference To A Sub Class Reference In Java](http://java.meritcampus.com/core-java-topics/assigning-super-class-reference-to-a-sub-class-reference) which explains how the references of super-class type can be assigned to a sub-class reference.

[**Assigning Super Class Reference To A Sub Class Reference In Java**](http://java.meritcampus.com/core-java-topics/assigning-super-class-reference-to-a-sub-class-reference)

As discussed in [Is-A Relationship In Java](http://java.meritcampus.com/core-java-topics/class-inheritance-is-a-relationship-in-java) every sub-class object is also of type super-class. Although all of the super-class objects are not of the sub-class type, some of them are. So the super-class references can be assigned to sub-class reference, if we know that the super-class reference is holding an object of sub-class type.

*Entertainment ent1;  
Drama d1 = new Drama();  
ent1 = d1;  
Drama d2 = (Drama) ent1; // LINE A*

As shown here, we are first creating a reference of Entertainment called ent1, a reference of Drama called d1 and a Dramaobject. Since we know that ent1 is referring to an object of type Drama, we can up cast the reference as shown in LINE A and assign it to another Drama reference d2.   
Instead, if we try to up cast a reference which does not refer to the correct object type, then it throws a ClassCastExceptionduring run time.

*Entertainment ent1;  
Circus c1 = new Circus();  
ent1 = c1;  
Drama d2 = (Drama) ent1; //* ***LINE A - THROWS ClassCastException****Circus c2 = (Circus) ent1; //* ***LINE B - WILL WORK***

Here a ClassCastException is thrown at LINE A, since ent1 points to Circus object and hence can not be cast to a Drama. But the casting as done in LINE B will work since ent1 points to the Circus object.

[**Multilevel Inheritance In Java With Example Program**](http://java.meritcampus.com/core-java-topics/multilevel-inheritance-in-java-with-example-program)

As discussed in [Java Class Inheritance](http://java.meritcampus.com/core-java-topics/class-inheritance-in-java-with-example-program) Java supports inheritance to provide a hierarchy for class definition and hence avoiding duplicate code and increasing maintainability. There is no limit on the levels of hierarchy. We could have class D extend from class C, which in turn extends from class B, which in turn extends from class A. e.g., MountainBike extends from Bike, Bikeextends from RegisteredVehicle, RegisteredVehicle extends from Vehicle.

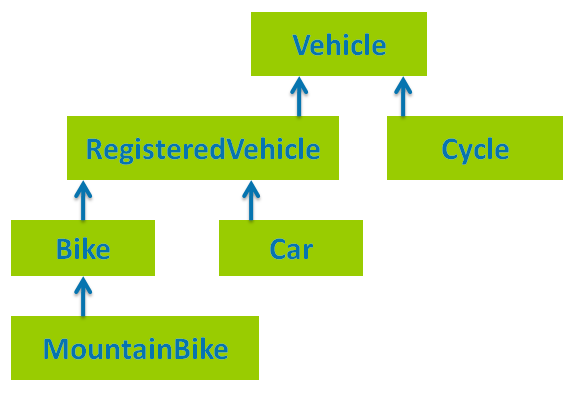
*public static void main(String arg[]){  
        MountainBike mb = new MountainBike();  
        mb.numberOfWheels = 2;  
        mb.registrationNumber = "APXX WWW";  
        mb.hasHelmet = true;  
        mb.maxElevation = 3000.0;  
          
        System.out.print("Mountain Bike with registration Number " + mb.registrationNumber);  
        System.out.println(" is supported till the elevation of " + mb.maxElevation + " feet.");      
    }  
}  
class Vehicle{  
    int numberOfWheels;  
}  
class RegisteredVehicle extends Vehicle{  
    String registrationNumber;  
}  
class Bike extends RegisteredVehicle* ***// LINE A****{  
    boolean hasHelmet;  
}  
class MountainBike extends Bike{  
    double maxElevation;  
}  
class Car extends RegisteredVehicle{  
    boolean hasAC;  
}  
class Cycle extends Vehicle{  
    boolean hasBackSeat;  
}*

**OUTPUT**

Mountain Bike with registration Number APXX WWW is supported till the elevation of 3000.0 feet.

**DESCRIPTION**

This program shows how to implement multi level inheritance. Initially Vehicle class is defined. RegisteredVehicle extends from Vehicle, Bike extends from RegisteredVehicle and MountainBike extends from Bike. So a MountainBike is a Bike, which in turn is a RegisteredVehicle, which in turn is a Vehicle. We also defined Car which extends from RegisteredVehicle and Cycle which extends from Vehicle.

The class hierarchy in the above program when shown in a diagram will look like.   
Please note that *multi-level inheritance* is different from *multiple inheritance*. Multiple inheritance which means inheriting from two classes is not supported in the Java. Although C++ supports multiple inheritance, it is intentionally not supported in java, since it adds lots of complexity.

# [Inheritance Of Inner Classes In Java](http://javaconceptoftheday.com/inheritance-inner-classes-java/)

**One inner class can extend another inner class of the same class.**

*class OuterClass{*

*class InnerClassOne{*

*int x = 10;*

*void methodOfInnerClassOne(){*

*System.out.println("From InnerClassOne");*

*}*

*}*

*class InnerClassTwo extends InnerClassOne{*

*//One Inner Class can extend another inner class*

*}*

*}*

*public class InnerClasses{*

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

*OuterClass outer = new OuterClass();* ***//Instantiating OuterClass***

*OuterClass.InnerClassTwo innerTwo = outer.new InnerClassTwo();* ***//Instantiating InnerClassTwo***

*System.out.println(innerTwo.x);*

***//Accessing inherited field x from InnerClassOne***

*innerTwo.methodOfInnerClassOne();*

***//calling inherited method from InnerClassOne***

*}*

*}*

An inner class can be extended by another class outside of it’s outer class. If you are extending static inner class (Static nested class), then it is a straight forward implementation. If you are extending non-static inner class, then sub class constructor must explicitly call super class constructor using an instance of outer class. Because, you can’t access non-static inner class without the instance of outer class.

|  |
| --- |
| *class OuterClass{*  *static class InnerClassOne{*  *//Class as a static member*  *}*  *class InnerClassTwo{*  *//Class as a non-static member*  *}*  *}*  ***//Extending Static inner class or static nested class***  *class AnotherClassOne extends OuterClass.InnerClassOne*  *{*  *//static nested class can be referred by outer class name,*  *}*    ***//Extending non-static inner class or member inner class***  *class AnotherClassTwo extends OuterClass.InnerClassTwo{*  *public AnotherClassTwo(){*  *new OuterClass().super();*  ***//accessing super class constructor through OuterClass instance***  *}*  *}* |

When an outer class is extended by it’s sub class, Member inner classes will not be inherited to sub class. To use inner class properties inside the sub class of outer class, sub class must also have an inner class and that inner class must extend inner class of the outer class. For example,

|  |
| --- |
| *class OuterClass{*  *int x;*  *void methodOfOuterClass(){*  *System.out.println("From OuterClass");*  *}*  *//****Class as a member***  *class InnerClass{*  *int y;*  *}*  *}*  *class AnotherClass extends OuterClass{*  ***//Only fields and methods are inherited.***  ***// To use inner class properties,***  ***//it's inner class must extend inner class of it's super class***  *class AnotherInnerClass extends InnerClass{*  ***//Inner Class of AnotherClass extends Inner Class of OuterClass***  *}*  *}*  *public class InnerClasses{*  *public static void main(String args[]){*  *AnotherClass anotherClass = new AnotherClass();*  ***//creating AnotherClass Object***  *System.out.println(anotherClass.x);*  ***//accessing inherited field x from OuterClass***  *anotherClass.methodOfOuterClass();*  ***//calling inherited method from OuterClass***  ***//Using the properties of InnerClass***  *AnotherClass.AnotherInnerClass anotherInnerClass = anotherClass.new AnotherInnerClass();*  ***//creating object to AnotherInnerClass***  *System.out.println(anotherInnerClass.y);*  ***//accessing inherited field y from InnerClass***  *}*  *}* |

Inner class can extend it’s outer class. But, it does not serve any meaning. Because, even the private members of outer class are available inside the inner class. Even though, When an inner class extends its outer class, only fields and methods are inherited but not inner class itself.

*class OuterClass{*

*int x;*

*void methodOfOuterClass(){*

*System.out.println("From OuterClass");*

*}*

*/****/Class as a member***

*class InnerClass extends OuterClass{*

*//only fields and methods are inherited, but not member Inner Classes*

*}*

*class InnerClassOne{*

*//another class as a member*

*}*

*}*

*public class InnerClasses{*

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

*OuterClass outer = new OuterClass();*

***//You have to create OuterClass object to access non-static inner class***

*OuterClass.InnerClass inner = outer.new InnerClass();*

***//creating object to InnerClass***

*System.out.println(inner.x);*

***//accesiing inherited field x***

*inner.methodOfOuterClass();*

***//accessing inherited method***

*}*

*}*

# [Java Interview Questions On Inheritance](http://javaconceptoftheday.com/java-interview-questions-on-inheritance/)

**1) What do you mean by inheritance.?**

Inheritance is one of the key features of object oriented programming. Through inheritance, a class (Sub Class) can inherit properties of another class (Super Class). Sub class can have it’s own properties along with the inherited properties from it’s super class.

**2) What are the types of inheritance.?**

There are 5 types of inheritance.

1). Single Inheritance : One class is extended by only one class.

2). Multilevel Inheritance : One class is extended by a class and that class in turn is extended by another class thus forming a chain of inheritance.

3). Hierarchical Inheritance : One class is extended by many classes.

4).Hybrid Inheritance : It is a combination of above types of inheritance.

5). Multiple Inheritance : One class extends more than one classes. (Java does not support multiple inheritance.)

**3) Can a class extend more than one classes or does java support multiple inheritance? If not, why?**

No, a class in java can not extend more than one classes or java does not support multiple inheritance. To avoid ambiguity, complexity and confusion, java does not supports multiple inheritance. For example, If Class C extends Class A and Class B which have a method with same name, then Class C will have two methods with same name. This causes ambiguity and confusion for which method to use. To avoid this, java does not supports multiple inheritance.

|  |
| --- |
| *class A{*  *void methodOne(){*  *System.out.println("From methodOfClassA");*  *}*  *}*  *class B{*  *void methodOne(){*  *System.out.println("From methodOfClassB");*  *}*  *}*  *class C extends A,B (If it is supported){*  ***//two same methods will be inherited to Class C.***  ***//This causes ambiguity and confusion.***  *}* |

**4) How do you implement multiple inheritance in java?**

Through interfaces, we can implement multiple inheritance in java. As classes in java cannot extend more than one classes, but a class can implement more than one interfaces.

|  |
| --- |
| *interface A{*  *}*  *interface B{*  *}*  *class C implements A, B{*  *//Class implementing two interfaces.*  *}* |

**5) You know that all classes in java are inherited from java.lang.Object class. Are interfaces also inherited from Object class.?**

No, only classes in java are inherited from Object class. Interfaces in java are not inherited from Object class. But, classes which implement interfaces are inherited from Object class.

**6) How do you restrict a member of a class from inheriting to it’s sub classes.?**

By declaring that member as a private. Because, private members are not inherited to sub classes.

**7) Can a class extend itself.?**

No, A class can not extend itself.

**8) Are constructors and initializers also inherited to sub classes.?**

No, Constructors and initializers(Static initializers and instance initializers) are not inherited to sub classes. But, they are executed while instantiating a sub class.

**9) What happens if both, super class and sub class, have a field with same name.?**

Super class field will be hidden in the sub class. You can access hidden super class field in sub class using super keyword.

**10) Are static members inherited to sub classes?**

Yes, Static members are also inherited to sub classes.

*class A{*

*static int i = 10;*

*static void method()   {*

*System.out.println("Static Method");*

*}*

*}*

*class B extends A{*

*}*

*public class StaticInitializers{*

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

*B.method();* ***//Calling inherited static method***

*System.out.println(B.i);* ***//printing inherited static field.***

*}*

*}*

***Note***

**Subclass can make use of super class members but super class cannot make use of subclass members**

## Que 1. Final Vs Non Final Method, Which is more efficient ?

### Answer : Final

1. Final methods execute more efficiently than non final method.
2. Compiler knows at compile time that a call to a final method won’t be overridden by some other method

## Let’s See, How ?

Consider the final method,

package com.c4learn.inheritance;

public class ShapeClass {

final **void** setShape() {

System.out.println("I am Inherited");;

}

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

Circle c1 = new Circle();

c1.setShape();

}

}

class Circle extends ShapeClass {

}

Whenever we create an object of the child class then compiler will check whether method written inside the parent class is overriden or not at run time.

final **void** setShape() {

System.out.println("I am Inherited");;

}

In this case compiler already knows that, method is declared as final so it does not waste time in checking whether method is final or not.

## Que 2. Can we inherit Final Method ?

### Answer : Yes

In case of final method, Method is inherited but cannot be overriden so always method from parent class will be executed.

Consider the following example –

package com.c4learn.inheritance;

public class ShapeClass {

final **void** setShape() {

System.out.println("I am Inherited");;

}

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

Circle c1 = new Circle();

c1.setShape();

}

}

class Circle extends ShapeClass {

}

**Output :**

I am Inherited

We can inherit the final method but cannot override the final method inside the child class.

## Que 3. Can Java have blank final Value ?

### Answer : No

package com.c4learn.inheritance;

public class BlankFinalValue {

final **int** myValue;

public BlankFinalValue(){

**this**.myValue = 3;

}

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

BlankFinalValue s1 = new BlankFinalValue();

}

}

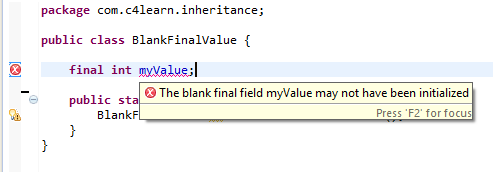
### Output :

Compiler Successfully

## Explanation :

1. We cannot have final Value which is uninitialized.
2. You will get compile time error if you did not provide initialization.
3. In this case we have to provide initialization in the constructor. (As shown in above program)

If final variable is not initialized in the constructor then it will throw **compile time error as below** –

[](http://javaimg.c4learn.com/2013/10/Uninitialized-Final-Value.png)

## If You have Multiple Constructor then –

we need to initialize final variable in each of the constructor like below program –

package com.c4learn.inheritance;

public class BlankFinalValue {

final **int** myValue;

public BlankFinalValue(){

**this**.myValue = 3;

}

public BlankFinalValue(**int** num){

**this**.myValue = num;

}

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

BlankFinalValue s1 = new BlankFinalValue();

}

}

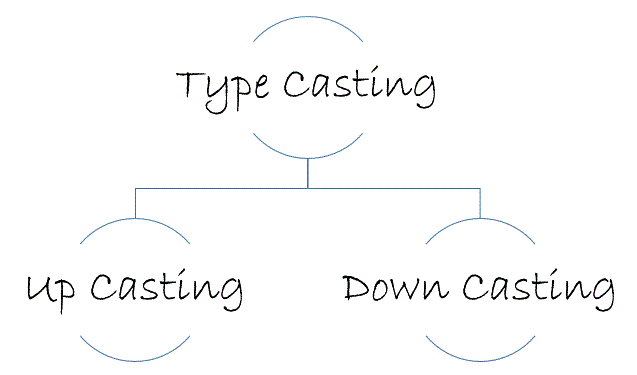
If you failed to initialize final variable in any of the constructor then you will get compile time error –

The blank final field myValue may not have been initialized

## Java type casting – Inheritance

Converting one type of value to another is called as **Type Casting**.

## Different Forms of Type Casting :

There are two types of type casting.  
[](http://javaimg.c4learn.com/2013/10/Type-Casting-Different-Forms.gif)

## Live Example of Type Casting in Inheritance :

**package** com.c4learn.inheritance;

**class** Vehicle {

**String** nameOfVehicle;

}

**class** TwoWheeler **extends** Vehicle {

**String** vehicleModel;

}

**public** **class** TypeCastExample {

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

Vehicle v1 = **new** Vehicle();

Vehicle v2 = **new** TwoWheeler();

TwoWheeler v3 = (TwoWheeler) **new** Vehicle();

TwoWheeler v4 = **new** TwoWheeler();

}

}

## A. Up Casting :

You can cast an instance of a child class to its parent class. Casting an object of child class to a parent class is called **upcasting**.

Consider the following example, Assuming that TwoWheeler is a subclass of Vehicle.

Vehicle v2 = new TwoWheeler();

## B. Down Casting :

Casting an object of a parent class to its child class is called **downcasting**.

Consider the following example, Assuming that TwoWheeler is a subclass of Vehicle. Following is an example of the down casting –

TwoWheeler v3 = (TwoWheeler) new Vehicle();

**Note:**In Inheritance the scope of access modifier increasing is allow but decreasing is not allow. Suppose in parent class method access modifier is default then it's present in child class with default or public or protected access modifier but not private(it decreased scope).

## Important Points for Inheritance:

* In java programming one derived class can extends only one base class because java programming does not support multiple inheritance through the concept of classes, but it can be supported through the concept of Interface.
* Whenever we develop any inheritance application first create an object of bottom most derived class but not for top most base class.
* When we create an object of bottom most derived class, first we get the memory space for the data members of top most base class, and then we get the memory space for data member of other bottom most derived class.
* Bottom most derived class contains logical appearance for the data members of all top most base classes.
* If we do not want to give the features of base class to the derived class then the definition of the base class must be preceded by final hence final base classes are not reusable or not inheritable.
* If we are do not want to give some of the features of base class to derived class than such features of base class must be as private hence private features of base class are not inheritable or accessible in derived class.
* Data members and methods of a base class can be inherited into the derived class but constructors of base class can not be inherited because every constructor of a class is made for initializing its own data members but not made for initializing the data members of other classes.
* An object of base class can contain details about features of same class but an object of base class never contains the details about special features of its derived class (this concept is known as scope of base class object).
* For each and every class in java there exists an implicit predefined super class called java.lang.Object. because it providers garbage collection facilities to its sub classes for collecting un-used memory space and improved the performance of java application.

### Difference between Java Inheritance and C++ Inheritance

The main difference between java Inheritance and C++ Inheritance is; Java doesn’t support multiple inheritance but C++ support.

**What is not possible using java Class Inheritance?**

1. Private members of the superclass are not inherited by the subclass and can only be indirectly accessed.
2. Since constructors and initializer blocks are not members of a Class, they are not inherited by a subclass.
3. A subclass can extend only one superclass
4. Members that have default accessibility in the superclass are also not inherited by subclasses in other packages, as these members are only accessible by their simple names in subclasses within the same package as the superclass.

## Disabling Inheritance

We can disable subclassing by declaring the class final.

A final class cannot be subclassed.

The following code declares a final class named MyClass:

**public** **final** **class** MyClass{

}

We can also declare a method as final. A final method cannot be overridden or hidden by a subclass.

**public** **class** A {

**public** **final** void m1() {

}

**public** void m2() {

}

}

***Note:***

Constructors are not members of a class and they are not inherited by subclasses.

They are used to initialize instance variables.

**class** CSuper {

**public** CSuper() {

System.out.println(**"Inside CSuper() constructor."**);

}

}

**class** CSub **extends** CSuper {

**public** CSub() {

System.out.println(**"Inside CSub() constructor."**);

}

}

**public** **class** Main {

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

CSub cs = **new** CSub();

}

}

The code above generates the following result.



## Example

The following shows how does Compiler Injection of a super() Call to Call the Immediate Ancestor's no-args Constructor

**class** CSuper {  
 **public** CSuper() {

**super**(); **// Injected by the compiler**

System.out.println(**"Inside CSuper() constructor."**);

}

}

**class** CSub **extends** CSuper {

**public** CSub() {

**super**(); **// Injected by the compiler**

System.out.println(**"Inside CSub() constructor."**);

}

}

**public** **class** Main {

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

CSub cs = **new** CSub();

}

}

The code above generates the following result.



The keyword super refers to the immediate ancestor of a class.

We can call superclass constructors using the super keyword only as the first statement inside a constructor.

## no-args constructor

We can call the no-args constructor or any other constructors of the superclass explicitly as the first statement inside constructors of a class.

The compiler injects the no-args constructor call only if you have not added one explicitly.

**class** Employee {

**private** String name = **"Unknown"**;

**public** Employee(String name) {

this.name = name;

}

**public** **void** setName(String name) {

this.name = name;

}

**public** String getName() {

**return** name;

}

}

**class** Manager **extends** Employee {

**public** Manager(String name) {

**super**(name);

}

}

**public** **class** Main {

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

Manager mgr = **new** Manager(**"Tom"**);

String name = mgr.getName();

System.out.println(**"Manager name: "** + name);

}

}

The code above generates the following result.

http://www.java2s.com/Tutorials/JavaImage/myResult/N/NO_ARGS_CONSTRUCTOR__FC89B1B30ED7D306DE17.PNG

Every class must call the constructor of its superclass from its constructors directly or indirectly.

If the superclass does not have a no-args constructor, we must call any other constructors of the superclass explicitly.

## Method Hiding

A class inherits all non-private static methods from its superclass.

Redefining an inherited static method in a class is known as method hiding.

The redefined static method in a subclass hides the static method of its superclass.

Redefining a non-static method in a class is called method overriding.

All rules about the redefined method (name, access level, return types, and exception) for method hiding are the same as for method overriding.

***class*** *MySuper {*

***public******static******void*** *print() {*

*System.out.println(****"Inside MySuper.print()"****);*

*}*

*}*

***class*** *MySubclass* ***extends*** *MySuper {*

***public******static******void*** *print() {*

*System.out.println(****"Inside MySubclass.print()"****);*

*}*

*}*

***public******class*** *Main {*

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

*MySuper mhSuper =* ***new*** *MySubclass();*

*MySubclass mhSub =* ***new*** *MySubclass();*

*MySuper.print();*

*MySubclass.print();*

*((MySuper) mhSub).print();*

*mhSuper = mhSub;*

*mhSuper.print();*

*((MySubclass) mhSuper).print();*

*}*

*}*

The code above generates the following result.



## Field Hiding

A field declaration, static or non-static, in a class hides the inherited field with the same name in its superclass.

The type of the field and its access level are not considered in the case of field hiding.

Field hiding occurs solely based on the field name.

***class*** *MySuper {*

***protected******int*** *num = 100;*

***protected*** *String name =* ***"Tom"****;*

*}*

***class*** *MySub* ***extends*** *MySuper {*

***public******void*** *print() {*

*System.out.println(****"num: "*** *+ num);*

*System.out.println(****"name: "*** *+ name);*

*}*

*}*

***class*** *MySub2* ***extends*** *MySuper {*

***// Hides num field in MySuper class***

***private******int*** *num = 200;*

***// Hides name field in MySuper class***

***private*** *String name =* ***"Jack"****;*

***public******void*** *print() {*

*System.out.println(****"num: "*** *+ num);*

*System.out.println(****"name: "*** *+ name);*

*}*

*}*

***public******class*** *Main {*

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

*MySub fhSub =* ***new*** *MySub();*

*fhSub.print();*

*MySub2 fhSub2 =* ***new*** *MySub2();*

*fhSub2.print();*

*}*

*}*

The code above generates the following result.



## Example

The following code shows how to Access Hidden Fields of Superclass Using the super Keyword

***class*** *MySuper {*

***protected******int*** *num = 100;*

***protected*** *String name =* ***"Tom"****;*

*}*

***class*** *MySub* ***extends*** *MySuper {*

***// Hides the num field in MySuper class***

***private******int*** *num = 200;*

***// Hides the name field in MySuper class***

***private*** *String name =* ***"Jack"****;*

***public******void*** *print() {*

*System.out.println(****"num: "*** *+ num);*

*System.out.println(****"super.num: "*** *+ super.num);*

*System.out.println(****"name: "*** *+ name);*

*System.out.println(****"super.name: "*** *+ super.name);*

*}*

*}*

***public******class*** *Main {*

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

*MySub s =* ***new*** *MySub();*

*s.print();*

*}*

*}*

The code above generates the following result.



Field hiding occurs when a class declares a variable with the same name as an inherited variable from its superclass.

Field hiding occurs only based on the name of the field.

A class should use the keyword super to access the hidden fields of the superclass.

The class can use the simple names to access the redefined fields in its body.