



---

# INHERITANCE -2

---

Velocity



AUGUST 3, 2022

VELOCITY  
pune

There are five types of inheritance as below

1. Single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hybrid inheritance
5. Hierarchical inheritance

### 1. Simple or Single inheritance

In single inheritance, a single subclass extends from a single superclass.



### 2. Multilevel Inheritance

In multilevel inheritance, a subclass extends from a superclass and then the same subclass acts as a superclass for another class.



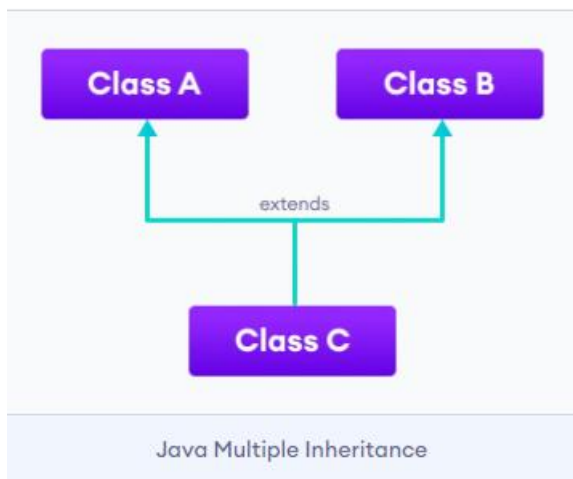
### 3. Hierarchical Inheritance

In hierarchical inheritance, multiple subclasses extend from a single superclass.



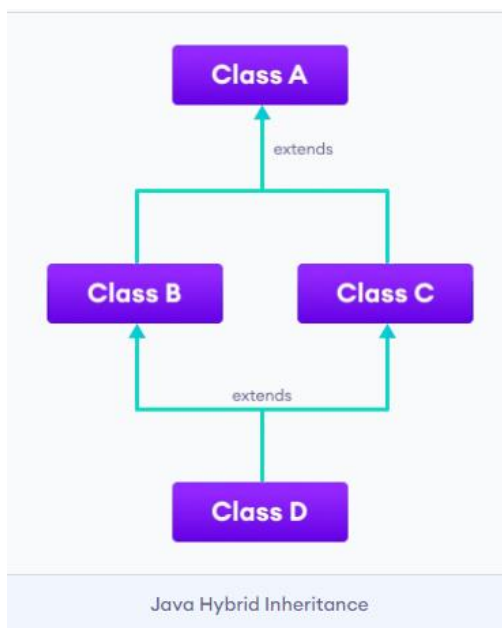
### 4. Multiple Inheritance

In multiple inheritance, a single subclass extends from multiple superclasses.



### 5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. For example,



Examples for single inheritance

```
package com.single.inheritance;
```

```
public class A {
```

```
    void m1() {  
        System.out.println("super class- m1 () method");  
    }
```

```
}
```

```
package com.single.inheritance;
```

```
public class B extends A {
```

```
    void m2() {  
        System.out.println("sub class - m2() method");  
    }
```

```
}
```

```
package com.single.inheritance;
```

```
public class TestMain {
```

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

```
        B b = new B();  
        b.m1();  
        b.m2();
```

```
    }
```

```
}
```

Example for multilevel inheritance

```
package com.multilevel.inheritance;
```

```
public class A {
```

```
    void m1() {  
        System.out.println("Class A- m1 () method");  
    }
```

```
}
```

```
package com.multilevel.inheritance;
```

```
public class B extends A{
```

```
    void m2() {  
        System.out.println("Class B- m2 method");  
    }
```

```
}
```

```
package com.multilevel.inheritance;
```

```
public class C extends B {
```

```
    void m3() {  
        System.out.println("Class c- m3 () method");  
    }
```

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

```
        C c= new C();  
        c.m1();  
        c.m2();  
        c.m3();
```

```
    }
```

```
}
```

Example for hierarchical inheritance

```
package com.hierachical.inheritance;
```

```
public class A {
```

```
    void m1() {  
        System.out.println("Class A- m1 () method");  
    }  
}
```

```
package com.hierachical.inheritance;
```

```
public class B extends A {
```

```
    void m2() {  
        System.out.println("Class B- m2() method");  
    }  
}
```

```
package com.hierachical.inheritance;
```

```
public class C extends A{
```

```
    void m3() {  
        System.out.println("Class c m3 method");  
    }  
}
```

```
package com.hierachical.inheritance;
```

```
public class D {
```

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

```
        B b = new B();  
        C c = new C();
```

```
        b.m1();  
        b.m2();  
        c.m3();
```

```
    }
```

```
}
```

Example for Multiple inheritance

```
package com.multiple.inheritance;
```

```
public class A {
```

```
    void m1() {
```

```
    }
```

```
}
```

```
package com.multiple.inheritance;
```

```
public class B {
```

```
    void m1() {
```

```
    }
```

```
}
```

```
package com.multiple.inheritance;
```

```
class C extends A,B {
```

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

```
        C c= new C();
```

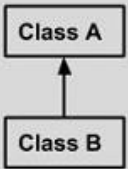
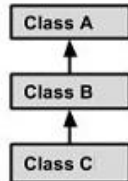
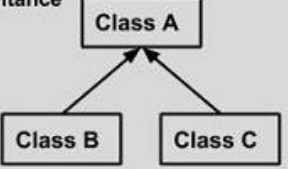
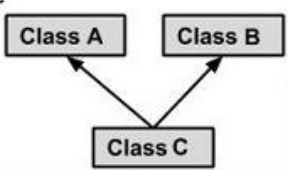
```
        c.m1();
```

```
    }
```

```
}
```

**Note-** it will get the compile time error.

## Different types of inheritance and their implementation in chart

<b>Single Inheritance</b>  <pre> graph BT     B[Class B] --&gt; A[Class A]         </pre>	<pre> public class A {     ..... } public class B extends A {     ..... }         </pre>
<b>Multi Level Inheritance</b>  <pre> graph BT     C[Class C] --&gt; B[Class B]     B --&gt; A[Class A]         </pre>	<pre> public class A { ..... } public class B extends A { ..... } public class C extends B { ..... }         </pre>
<b>Hierarchical Inheritance</b>  <pre> graph BT     B[Class B] --&gt; A[Class A]     C[Class C] --&gt; A         </pre>	<pre> public class A { ..... } public class B extends A { ..... } public class C extends A { ..... }         </pre>
<b>Multiple Inheritance</b>  <pre> graph BT     C[Class C] --&gt; A[Class A]     C --&gt; B[Class B]         </pre>	<pre> public class A { ..... } public class B { ..... } public class C extends A,B {     ..... } // Java does not support mutiple Inheritance         </pre>

## Important Points

- Code reuse is the most important benefit of inheritance because subclasses inherits the variables and methods of superclass.
- Private members of superclass are not directly accessible to subclass. But it can be indirectly accessible via getter and setter methods.
- Superclass members with default access is accessible to subclass ONLY if they are in same package.
- Superclass constructors are not inherited by subclass.
- If superclass doesn't have default constructor, then subclass also needs to have an explicit constructor defined. Else it will throw compile time exception. In the subclass constructor, call to superclass constructor is mandatory in this case and it should be the first statement in the subclass constructor.
- Java doesn't support multiple inheritance, a subclass can extends only one class.
- We can override the method of Superclass in the Subclass. However we should always annotate overridden method with `@Override` annotation. The compiler will know that we are overriding a method and if something changes in the superclass method, we will get a compile-time error rather than getting unwanted results at the runtime.
- We can call the superclass methods and access superclass variables using `super` keyword. It comes handy when we have the same name variable/method in the subclass but we want to access the superclass variable/method. This is also used when Constructors are



defined in the superclass and subclass and we have to explicitly call the superclass constructor.

- We can use instanceof instruction to check the inheritance between objects, let's see this with below example.

```
Cat c = new Cat();
Dog d = new Dog();
Animal an = c;

boolean flag = c instanceof Cat; //normal case, returns true

flag = c instanceof Animal; // returns true since c is-an
Animal too

flag = an instanceof Cat; //returns true because a is of type
Cat at runtime

flag = an instanceof Dog; //returns false for obvious reasons.
```

- We can't extend Final classes in java.

## Forms of Association in Java

There can be two types of relationships in OOPs:

-IS-A

-HAS-A

### IS-A (Inheritance)

The IS-A relationship is nothing but Inheritance. The relationships that can be established between classes using the concept of inheritance are called IS-A relations.

### HAS-A (association)

There are two forms of Association that are possible in Java:

a) Aggregation : It is a special form of Association where

- It represents Has-A relationship.
- It is a unidirectional association i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, both the entries can survive individually which means ending one entity will not effect the other entity

b) Composition

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

- It represents part-of relationship.
- In composition, both the entities are dependent on each other.
- When there is a composition between two entities, the composed object cannot exist without the other entity.

Aggregation	Composition
Aggregation allows child objects or contained objects to exist independently. <b>For example</b> in a relationship where School has Employees, if we delete School, Employees will remain and can function on its own.	In composition, the contained object cannot exist independently. <b>For example</b> a car has Engine. If we delete Car, Engine class cannot function on its own.
Aggregation is a 'has-a' relationship.	The composition is a form of 'has-a' relationship but viewed as part-of-a-whole' relation.
The aggregation has one to one association only.	Composition allows other relationships provided in the association.
The aggregation has a weak association between objects.	The composition has a strong association between objects.