Inheritance in Java - It is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.

Important terminologies:

•Super/Base/Parent Class: The class whose features are inherited.

•Sub/Derived/Child Class: The class that inherits the other class. The subclass can add its own fields and methods in addition to the superclass fields and methods.

•Reusability: Inheritance supports the concept of "reusability", i.e. we can reuse the code of Parent Class by Inheriting it in Child Class.

- The keyword used for inheritance is **extends**.

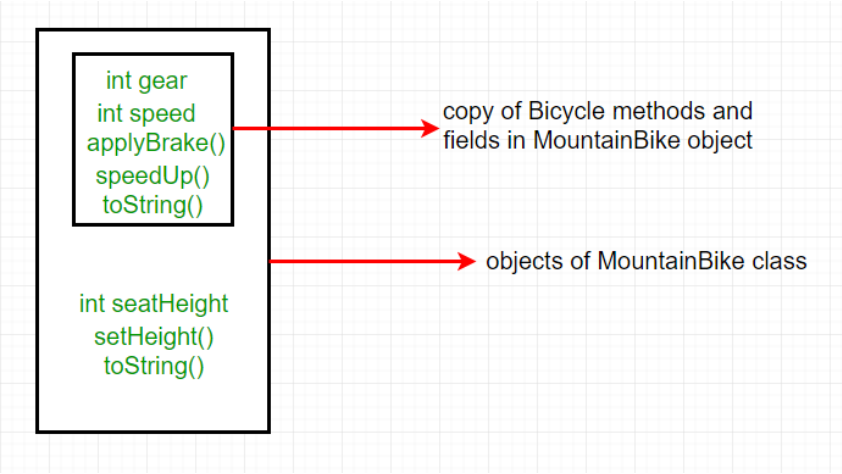
- In Java, constructor of parent class with no argument gets **automatically** called in child class constructor. But, if we want to call **parameterized contructor** of base class, then we can call it using **super()**.

- Parent class constructor call must be the **first line** in Child class constructor.

- When subclass object is created, a **separate object of super class** will **not be created** even when it's constructor is executed. So we can't blindly say that whenever a class constructor is executed, object of that class is created or not.

- When object of sub class is created, a **copy** of the all methods and fields of the superclass acquire memory in this object. **Object of superclass is not created**.

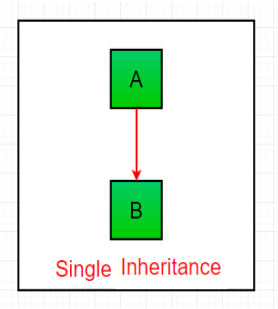
- Example: Assume Bicycle (Parent Class) and MountainBike (Child Class). On creating object of MountainBike,



Types of Inheritance in Java

Single Inheritance :

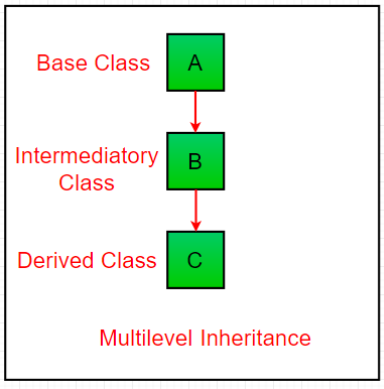
➢ In this, subclasses inherit the features of **one superclass**. In the image, the class A serves as a base class for the derived class B.



Multilevel Inheritance :

➢ A derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.

➢ In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.



➢ In **Java**, a class **cannot** directly access the **grandparent's members**. It is allowed in **C++** though. In C++, we can use **scope resolution operator (::)** to access any ancestor's member in inheritance hierarchy.

Example:

```
// Trying to directly access grandparent's members
class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    }
}
class Parent extends Grandparent {
    public void Print() {
        System.out.println("Parent's Print()");
    }
}
```

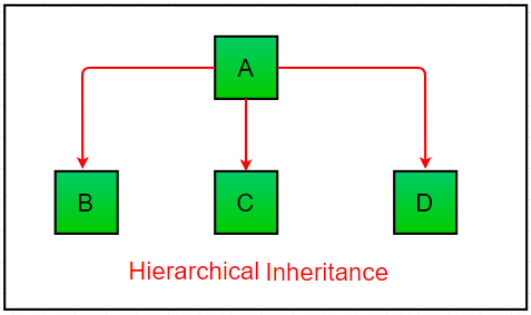
```
class Child extends Parent {
    public void Print() {
        super.super.Print();  // Trying to access Grandparent's Print() (ERROR LINE)
        System.out.println("Child's Print()");
    }
}
public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

Output: Compiler Error

- In Java, we can access grandparent's members only **through the parent class**.

Hierarchical Inheritance :

- In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.
- In below image, the class A serves as a base class for the derived class B,C and D.



Multiple Inheritance :

- Multiple Inheritance is a feature of object oriented concept, where a class can **inherit properties of more than one parent class.**

- The **problem** occurs when there exist methods with **same signature** in **both the super classes**. On calling the method (in subclass), the compiler cannot determine which class method to be called and even on calling which class method gets the priority. It will give **Compilation Error**.
- **The Diamond Problem**: In Multiple Inheritance,

```
      GrandParent
      /          \
  Parent_1      Parent_2
      \          /
       Child Class
```

➢ Suppose, both **Parent_1** and **Parent_2** have a function **fun()**, then there will be confusion. Therefore, in order to avoid such complications Java does not support multiple inheritance of classes.

Multiple Inheritance (using Interface) :

➢ Interface contains **abstract methods** (methods without implementation) which are to be overridden in Child Class to provide implementation.

➢ Therefore, even if Two Parent Interface contains method with same signature, it won't matter as their implementation is absent.

➢ Example:

```
// Java program to illustrate the concept of Multiple inheritance
import java.util.*;
import java.io.*;

interface one {
    public void print_geek();
}

interface two {
    public void print_for();
}

interface three extends one,two { // Multiple Inheritance
    public void print_geek();
}

class child implements three  {
    public void print_geek() { // Overriding methods from Parent Interface
        System.out.println("Geeks");
    }

    public void print_for() { // Overriding methods from Parent Interface
        System.out.println("for");
    }
}

public class Main {
    public static void main(String[] args) {
        child c = new child();
        c.print_geek();
        c.print_for();
        c.print_geek();
          }
}
```

Default Methods (Interface) and Multiple Inheritance –

➢ From Java 8, **Default Methods** can be defined in Interfaces. Before Java 8, interfaces could have only abstract methods.

➢ Default methods **allow** the interfaces to have methods with implementation.

Example:

```
interface TestInterface {
   public void square (int a);  // abstract method
   default void show() {   // default method
      System.out.println("Default Method Executed");
   }
}
```

➢ Solution: In case both the implemented interfaces contain default methods with same method signature, the implementing class should **explicitly specify** which default method is to be used **or** it should **override the default method** else we will get Compilation Error.

Example:

```
// A simple Java program to demonstrate multiple inheritance through default methods.
interface TestInterface1  {
   default void show() { // default method
      System.out.println("Default TestInterface1");
   }
}

interface TestInterface2 {
   default void show() {  // Default method
      System.out.println("Default TestInterface2");
   }
}

class TestClass implements TestInterface1, TestInterface2  {
   public void show()  {    // Overriding default show method

      TestInterface1.super.show(); // use super keyword to call the show method of TestInterface1 interface

      TestInterface2.super.show();   // use super keyword to call the show method of TestInterface2 interface
   }
   public static void main(String args[]) {
      TestClass d = new TestClass();
      d.show();
   }
}
```

Output:

```
Default TestInterface1
Default TestInterface2
```

➢ If there is a diamond through interfaces, then there is **no issue** if **none** of the **middle interfaces** provide **implementation** of **root interface**.

Example:

```
interface GPI  {
   default void show() { // default method
      System.out.println("Default GPI");
   }
}

interface PI1 extends GPI { }

interface PI2 extends GPI { }

class TestClass implements PI1, PI2 {
   public static void main(String args[])  {
      TestClass d = new TestClass();
      d.show();
   }
}
```

➢ If they **provide implementation**, then implementation can be accessed using **super keyword** (normally as already described earlier).

Important facts about inheritance in Java -

➢ Default superclass: Except **Object** class, which has no superclass, every class has **one and only one direct superclass** (single inheritance).

➢ In the absence of any other explicit superclass, every class is **implicitly a subclass** of **Object** class.

➢ A superclass can have any number of subclasses. But a subclass can have only one superclass.

➢ Inheriting Constructors: A subclass inherits all the members (fields, methods, and nested classes) from its superclass. **Constructors are not members**, so they are **not inherited by subclasses**, but the constructor of the superclass can be invoked from the subclass.

➢ Private member inheritance: A subclass **does not inherit the private members** of its parent class. Though they can be accessed through Public / Protected Getters and Setters.

➢ Final Keyword: During inheritance, we must declare methods with final keyword for which we want to follow the same implementation throughout all the derived classes.

➢ **Final Class** can **not** be subclassed i.e. no other class can extend it.

What all can be done in a Subclass?

- The inherited fields can be used directly, just like any other fields / new fields can be declared that are not in superclass.

- The **inherited methods** can be **used directly** as they are OR can be **overidden**.

- We can write a **new static method** in the subclass that has the same signature as the one in the superclass, thus **hiding** it.

- We can declare new methods in the subclass that are not in the superclass.

- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.