10-Inheritance

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).
- 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 the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class**: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

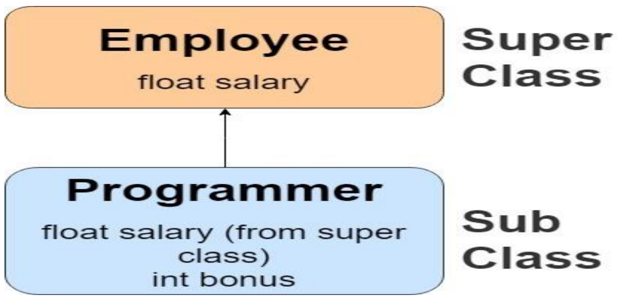
The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
        //methods and fields
}
```

- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example

- As displayed in the following figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is Programmer IS-A Employee. It means that Programmer is a type of Employee.



```
class Employee// parent class
{
        float salary=40000;
}
class Programmer extends Employee  //child class
{
        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);
        }
}
```

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

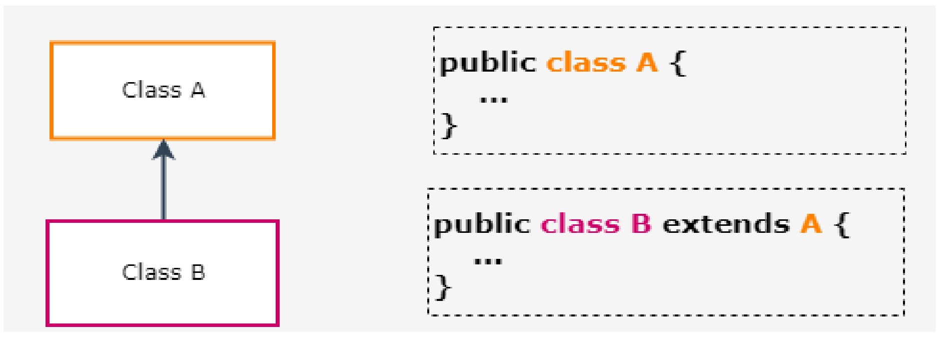
Types of inheritance in java

- On the basis of class, there can be three types of inheritance in java: **single, multilevel and hierarchical.**
- In java programming, **multiple and hybrid inheritance** is supported through **interface only**. We will learn about interfaces later.

Note: *Multiple inheritance is not supported in Java through class.*

(1) Single Inheritance Example

- When a class inherits another class, it is known as a single inheritance.



```
public class A {
   ...
}

public class B extends A {
   ...
}
```

- **Example:**In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

```
class Animal
{
    void eat()
    {
            System.out.println("eating...");
    }
}
class Dog extends Animal
{
    void bark()
    {
            System.out.println("barking...");
    }
}
class TestInheritance
{
    public static void main(String args[])
    {
            Dog d=new Dog();
            d.bark();
            d.eat();
    }
}
```

Output:

```
barking...
eating...
```

Eample-2

```
class Parent
{
    private int a=10;//a has private access in Parent can't be acceced in child class
    int b=20;
    void display()
    {
            System.out.println("hi papa");
    }
    void getParent()
    {
            System.out.println("hi parent");
            System.out.println("a= "+a+" b= "+b);
    }
}
class Child extends Parent
```

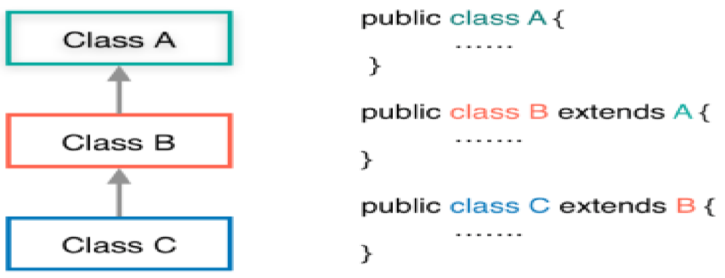
```
{
    int c=30;
    void print()
    {
            System.out.println("hello beta");
    }
    void getChild()
    {
            //System.out.println("in child a= "+a);
            System.out.println("in child b= "+b);
            System.out.println("in child c= "+c);

    }

}
class InnhEx1
{
    public static void main(String args[])
    {
            Parent p=new Parent();
            p.getParent();
            Child c=new Child();
            c.display();
            c.print();
            c.getChild();
    }
}
```

Output

```
E:\java ARP>java InnhEx1
hi parent
a= 10 b= 20
hi papa
hello beta
in child b= 20
in child c= 30
```

Multilevel Inheritance

- ➢ When there is a chain of inheritance, it is known as multilevel inheritance.



```
public class A {
        ......
 }
public class B extends A {
        .......
}
public class C extends B {
        .......
}
```

- **Example:1:**As you can see in the example given below**, BabyDog class inherits the Dog class which again inherits the Animal class**, so there is a multilevel inheritance.

```
class Animal
{
    void eat()
    {
            System.out.println("eating...");
    }
}
class Dog extends Animal
{
    void bark()
    {
            System.out.println("barking...");
    }
}
class BabyDog extends Dog
{
    void weep()
    {
            System.out.println("weeping...");
    }
}
class TestInheritance2
{
    public static void main(String args[])
    {
            BabyDog d=new BabyDog();
            d.weep();
            d.bark();
            d.eat();
    }
}
```

Output:

```
weeping...
barking...
eating...
```

Example-2

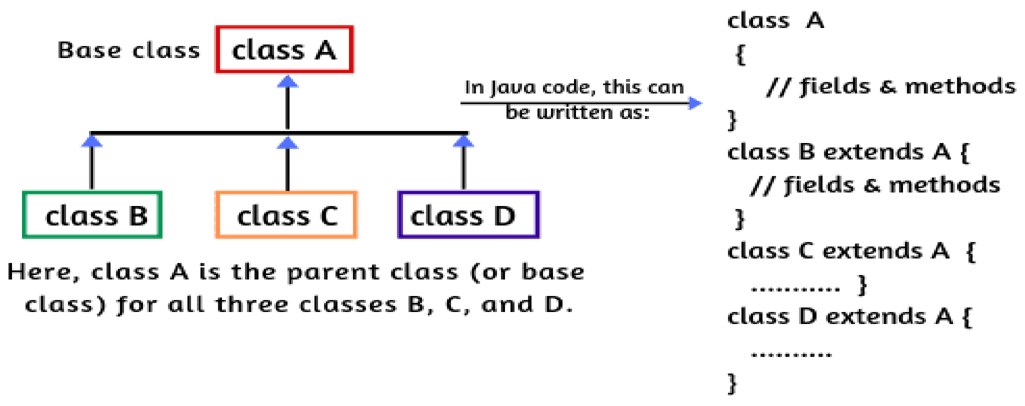
```
    class Parent
    {
            private int a=10;//a has private access in Parent cant be acceced in child class
            int b=20;
```

```
		void display()
		{
			System.out.println("hi papa");
		}
		void getParent()
		{
			System.out.println("a= "+a+" b= "+b);
		}
	}
	class Child1 extends Parent
	{
		int c=30;
		void print()
		{
			System.out.println("hello beta");
		}
		void getChild1()
		{
			System.out.println("in child1 b= "+b);//accessed from parent
			System.out.println("in child1 c= "+c);

		}
	}
	class Child2 extends Child1
	{

		void getChild2()
		{
			System.out.println("in child2 b= "+b);//accessed from parent child1
			System.out.println("in child2 c= "+c);//accessed from parent child1

		}
	}
	class InnhEx2
	{
		public static void main(String args[])
		{
			Child2 c2=new Child2();
			c2.getParent();
			c2.display();
			c2.print();
			c2.getChild1();
			c2.getChild2();

		}
	}
```

```
/*
E:\java ARP>java InnhEx2
a= 10 b= 20
hi papa
hello beta
in child1 b= 20
in child1 c= 30
in child2 b= 20
in child2 c= 30*/
```

Hierarchical Inheritance

- When two or more classes inherits a single class, it is known as hierarchical inheritance.


Base class
class A
In Java code, this can be written as:
class B
class C
class D
Here, class A is the parent class (or base class) for all three classes B, C, and D.

```
class  A
 {
    // fields & methods
}
class B extends A {
  // fields & methods
 }
class C extends A  {
  ..........  }
class D extends A {
  .........
}
```

- **Example**: In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

```
class Animal
{
    void eat()
    {
            System.out.println("eating...");
    }
}
class Dog extends Animal
{
    void bark()
    {
            System.out.println("barking...");
    }
}
class Cat extends Animal
{
    void meow()
    {
            System.out.println("meowing...");
    }
}
class TestInheritance3
{
    public static void main(String args[])
    {
            Cat c=new Cat();
            c.meow();
            c.eat();
            //c.bark();//C.T.Error
    }
}
Output:
meowing...
eating...
```

Method Overriding in Java

- Inheritance in java involves a relationship between parent and child classes. Whenever both the classes contain methods with the same name and arguments and parameters it is certain that one of the methods will override the other method during execution. The method that will be executed depends on the object.
- If the child class object calls the method, the child class method will override the parent class method. Otherwise, if the parent class object calls the method, the parent class method will be executed.

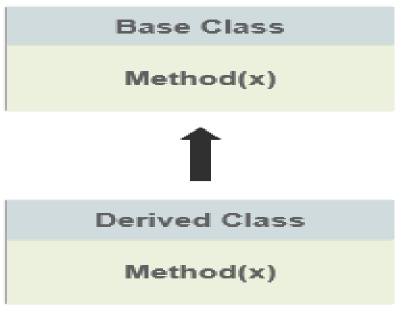
Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).
- A final method does not support method overriding.
- A static method cannot be overridden.
- Private methods cannot be overridden.
- The return type of the overriding method must be the same.
- We can call the parent class method in the overriding method using the super keyword.
- A constructor cannot be overridden because a child class and a parent class cannot have the constructor with the same name.

Lets Take an example to understand method overriding



```
class Parent
{
        void show()
        {
                System.out.println("parent class method");
        }
class Child extends Parent
{
        void show()
        {
                super.show();
                System.out.println("child class method");
        }
}
class Main
{
        public static void main(String args[])
        {
                Parent ob = new Child();
                ob.show();
        }
}
```

Output

```
parent class method
child class method
```

Lets take one more example of method overriding to understand it properly

```
        import java.util.*;
        /*Method over riding*/
        class Bank
        {
                int roi;
                void getRoi()
                {
                        System.out.println("Bank rate of interest");
                }
        }
```

```
class Icici extends Bank
{
	void getRoi()
	{
		System.out.println("Rate of Interest ICICI= "+roi+"%");
	}
}
class Hdfc extends Bank
{
	void getRoi()
	{
		System.out.println("Rate of Interest HDFC= "+roi+"%");
	}
}
class Over_ex1
{
	public static void main(String args[])
	{
		Icici i=new Icici();
		i.roi=7;
		i.getRoi();
		Hdfc h=new Hdfc();
		h.roi=6;
		h.getRoi();
	}
}
/*  E:\java ARP>java Over_ex1
Rate of Interest ICICI= 7%
Rate of Interest HDFC= 6%
*/
```

Super Keyword in Java

- The super keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

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

(1) super is used to refer immediate parent class instance variable.

- We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
//Parent class or Superclass or base class
    class Superclass
    {
            int num = 100;
    }
    //Child class or subclass or derived class
    class Subclass extends Superclass
    {
              /* The same variable num is declared in the Subclass
               * which is already present in the Superclass
               */
            int num = 110;
            void printNumber()
            {
                    System.out.println(num);
            }
    }
    Class Main
    {
            public static void main(String args[])
            {
                    Subclass obj= new Subclass();
                    obj.printNumber();
            }
    }
```

Output:

110

- Let's take the same example that we have seen above, this time in print statement we are passing **super.num** instead of num.

```
class Superclass
{
        int num = 100;
}

//Child class or subclass or derived class

class Subclass extends Superclass
{
  /* The same variable num is declared in the Subclass
   * which is already present in the Superclass
   */
        int num = 110;
        void printNumber()
        {
                System.out.println(num);
                System.out.println(super.num);

        }
    }
    Class Main
    {
        public static void main(String args[])
        {
                Subclass obj= new Subclass();
                obj.printNumber();
        }
    }
```

Output:

```
110
100
```

(2) super can be used to invoke parent class method

- The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Parentclass
{
    //Overridden method
    void display()
    {
        System.out.println("Parent class method");
    }
}
class Subclass extends Parentclass
{
    //Overriding method
    void display()
    {
        System.out.println("Child class method");
    }
    void printMsg()
    {
        //This would call Overriding method
        display();
        //This would call Overridden method
        super.display();
    }
}
class Main
{
    public static void main(String args[])
    {
        Subclass obj= new Subclass();
        obj.printMsg();
    }
}
```

Output

```
/*E:\java ARP>java Main
Child class method
Parent class method
*/
```

(3) Super is used to invoke parent class constructor.

- The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:
- When we create the object of sub class, the new keyword invokes the constructor of child class, which implicitly invokes the constructor of parent class.
- So the order to execution when we create the object of child class is: parent class constructor is executed first and then the child class constructor is executed. It happens because compiler itself adds super()(this invokes the no-arg constructor of parent class) as the first statement in the constructor of child class.

```
class Parentclass
{
        Parentclass()
        {
                System.out.println("Constructor of parent class");
        }
}
class Subclass extends Parentclass
{
        Subclass()
        {
                /* Compile implicitly adds super() here as the
                 *  first statement of this constructor.
                 */
                System.out.println("Constructor of child class");
        }
        Subclass(int num)
        {
                /* Even though it is a parameterized constructor.
                 * The compiler still adds the no-arg super() here
                 */
                System.out.println("arg constructor of child class");
        }
        void display()
```

```
	{
		System.out.println("Hello!");
	}
}
class Main
{
	public static void main(String args[])
	{
		/* Creating object using default constructor. This
		 * will invoke child class constructor, which will
		 * invoke parent class constructor
		 */
		Subclass obj= new Subclass();

		//Calling sub class method
		obj.display();

		/* Creating second object using arg constructor
		 * it will invoke arg constructor of child class which will
		 * invoke no-arg constructor of parent class automatically
		 */
		Subclass obj2= new Subclass(10);
		obj2.display();
	}
}
/*
E:\java ARP>java Main
Constructor of parent class
Constructor of child class
Hello!
Constructor of parent class
arg constructor of child class
Hello!*/
```

- **Parameterized super() call to invoke parameterized constructor of parent class**
- We can call super() explicitly in the constructor of child class, but it would not make any sense because it would be redundant. It's like explicitly doing something which would be implicitly done otherwise.
- However when we have a constructor in parent class that takes arguments then we can use parameterized super, like super(100); to invoke parameterized constructor of parent class from the constructor of child class.
- Let's see an example to understand this:

```
class Parentclass
{
	//no-arg constructor
```

```
        Parentclass()
        {
                System.out.println("no-arg constructor of parent class");
        }
        //arg or parameterized constructor
        Parentclass(String str)
        {
                //this();
                System.out.println("parameterized constructor of parent class");
        }
}
class Subclass extends Parentclass
{
        Subclass()
        {
          /* super() must be added to the first statement of constructor
             otherwise you will get a compilation error.
           *Another important point to note is that when we explicitly use super in constructor
             the compiler doesn't invoke the parent constructor automatically.
        */
                super("Hahaha");
                System.out.println("Constructor of child class");
        }
        void display()
        {
                System.out.println("Hello");
        }
}
class Main
{
        public static void main(String args[])
        {
                Subclass obj= new Subclass();
                obj.display();
        }
}
```

Output

```
/*E:\java ARP>java Main
parameterized constructor of parent class
Constructor of child class
Hello*/
```

Final Keyword In Java

- The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1) variable
2) method
3) class

- The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only.
- The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these.
- Let's first learn the basics of final keyword.

(1) Java final variable

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

/*There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.*/

```
class Bike9
{
        final int speedlimit=90;//final variable
        void run()
        {
         speedlimit=400;
        }
}
Class Main
{
        public static void main(String args[])
        {
        Bike9 obj=new  Bike9();
        obj.run();
        }
}//end of class
```

Output:Compile Time Error

(2) Java final method

- If you make any method as final, you cannot override it.

Example of final method

```
class Bike
{
        final void run()
        {
                System.out.println("running");
        }
}
class Honda extends Bike
{
        void run()
        {
                System.out.println("running safely with 100kmph");
        }
}
Class Main
{
        public static void main(String args[])
        {
                Honda honda= new Honda();
                honda.run();
        }

}
```

Output:Compile Time Error

(3) Java final class

➢ If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike
{
        System.out.println("Drive Safely");

}
class Honda1 extends Bike
{
        void run()
        {
                System.out.println("running safely with 100kmph");
        }
}

class Main
{
        public static void main(String args[])
        {
                Honda1 honda= new Honda1(); honda.run();
        }
}
```

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
class Bike
{
        final void run()
        {
                System.out.println("running...");
        }
}
class Honda2 extends Bike
{
        public static void main(String args[])
        {
                new Honda2().run();
        }
}
```

Output:running...

Q) What is blank or uninitialized final variable?

- A final variable that is not initialized at the time of declaration is known as blank final variable.
- If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.
- It can be initialized only in constructor.

Example of blank final variable

```
class Student
{
        int id;
        String name;
        final String PAN_CARD_NUMBER;
        ...
}
```

Q) Can we initialize blank final variable?

- Yes, but only in constructor. For example:

```
class Bike10
{
        final int speedlimit;//blank final variable
```

```
	Bike10()
	{
		speedlimit=70;
		System.out.println(speedlimit);
	}


	public static void main(String args[])
	{
		new Bike10();
	}
}
```

Output: 70

static blank final variable

- A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
class A
{
	static final int data;//static blank final variable
	static
	{
		data=50;
	}
	public static void main(String args[])
	{
		System.out.println(A.data);
	}
}
```

Q) What is final parameter?

- If you declare any parameter as final, you cannot change the value of it.

```
class Bike11
{
	int cube(final int n)
	{
		n=n+2;//can't be changed as n is final n*n*n;
	}
	public static void main(String args[])
	{
		Bike11 b=new Bike11();
		b.cube(5);
	}
}
```
Output: Compile Time Error

Q) Can we declare a constructor final?

- No, because constructor is never inherited.