**Master Inheritance In Java With Examples**

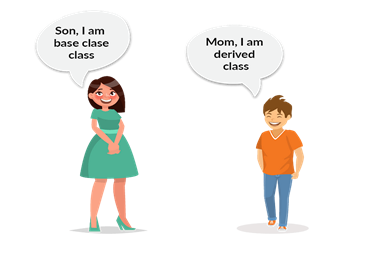
* Introduction to Inheritance in Java
* Types of Inheritance in Java

1. Single Inheritance
2. Multi-level Inheritance
3. Hierarchical Inheritance
4. Hybrid Inheritance

* Rules of Inheritance in Java

# Introduction To Inheritance in Java

In OOP, computer programs are designed in such a way where everything is an object that interacts with one another. Inheritance is an integral part of Java OOPs which lets the properties of one class to be inherited by the other. It basically, helps in reusing the code and establish a relationship between different classes.



As we know, a child inherits the properties of his parents. A similar concept is followed in Java, where we have two classes:

1. Parent class ( Super or Base class )

2. Child class ( Subclass or Derived class )

A class that inherits the properties is known as Child Class whereas a class whose properties are inherited is known as Parent class.

**Syntax:**

Now, to inherit a class we need to use extends keyword. In the below example, class Son is the child class and class Mom is the parent class. The class Son is inheriting the properties and methods of Mom class.

class Son extends Mom  
{  
//your code  
}

Let’s see a small program and understand how it works. In this example, we have a base class Teacher and a subclass HadoopTeacher. Since class HadoopTeacher extends the properties from the base class, we need not declare these properties and method in the subclass.

class Teacher {

    String designation = "Teacher";

    String collegeName = "Gla";

    void does() {

        System.out.println("Teaching");

    }

}

public class PythonTeacher extends Teacher {

    String mainSubject = "Spark";

    public static void main(String args[]) {

        PythonTeacher obj = new PythonTeacher ();

        System.out.println(obj.collegeName);

        System.out.println(obj.designation);

        System.out.println(obj.mainSubject);

        obj.does();

    }

}

**Output:**

Gla

Teacher

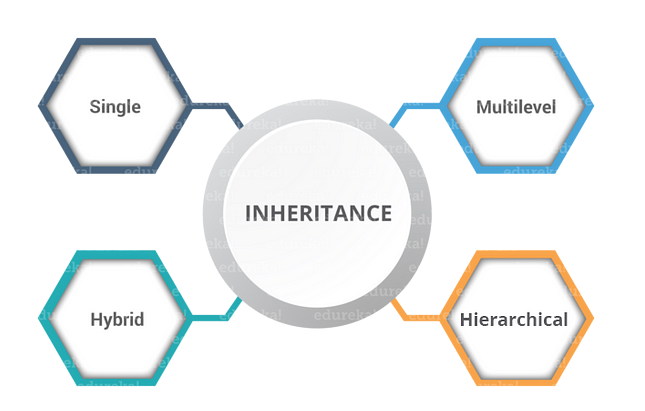
Spark

Teaching

Now let’s move further and see the various types of Inheritance supported by Java.

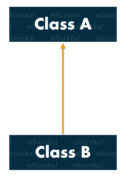
# Types of Inheritance in Java

Below figure depicts the types of Inheritance:



## Single Inheritance

In single inheritance, one class inherits the properties of another. It enables a derived class to inherit the properties and behavior from a single parent class. This will, in turn, enable code reusability as well as add new features to the existing code.



Here, Class A is your parent class and Class B is your child class which inherits the properties and behavior of the parent class. A similar concept is represented in the below code:

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();

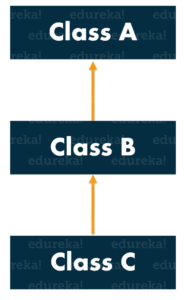
        d.eat();

    }

}

## Multi-level Inheritance

When a class is derived from a class which is also derived from another class, i.e. a class having more than one parent class but at different levels, such type of inheritance is called Multilevel Inheritance.



If we talk about the flowchart, class B inherits the properties and behavior of class A and class C inherits the properties of class B. Here A is the parent class for B and class B is the parent class for C. So in this case class C implicitly inherits the properties and methods of class A along with Class B. That’s what is multilevel inheritance.

class Animal {

void eat(){System.out.println(“eating…”);}

}

class Dog extends Animal {

void bark(){System.out.println(“barking…”);}

}

class Puppy extends Dog {

void weep(){System.out.println(“weeping…”);}

}

class TestInheritance2 {

    public static void main(String args[]) {

        Puppy d = new Puppy();

        d.weep();

        d.bark();

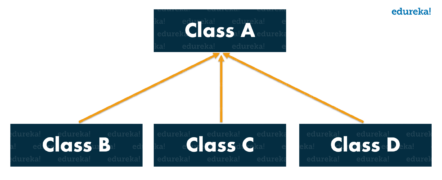
        d.eat();

    }

}

## Hierarchical Inheritance

When a class has more than one child classes (subclasses) or in other words, more than one child classes have the same parent class, then such kind of inheritance is known as **hierarchical**.



In the above flowchart, Class B and C are the child classes which are inheriting from the parent class i.e Class A.

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();

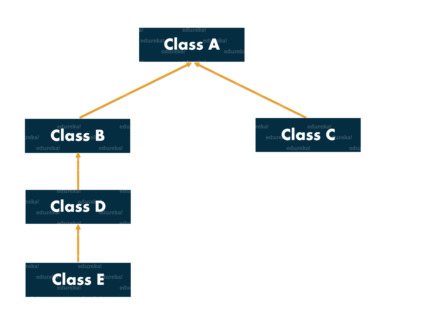
        c.eat();

    }

}

## Hybrid Inheritance

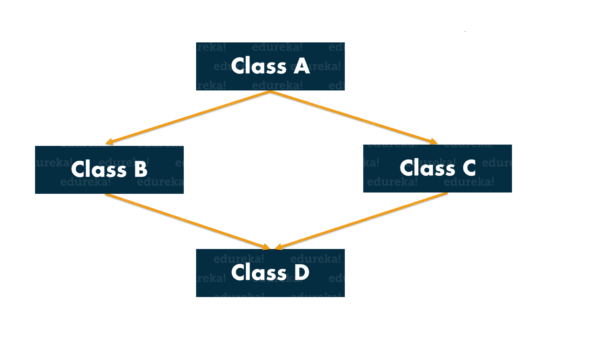
Hybridinheritance is a combination of two or more types of inheritance.



Now that we know what is Inheritance and its various types, let’s move further and see some of the important rules that should be considered while inheriting classes.

# Rules of Inheritance in Java

## RULE 1: Multiple Inheritance is NOT permitted in Java.



Multiple inheritance refers to the process where one child class tries to extend more than one parent class. In the above illustration, Class A is a parent class for Class B and C, which are further extended by class D. This is results in Diamond Problem. Why? Say you have a method show() in both the classes B and C, but with different functionalities. When Class D extends class B and C, it automatically inherits the characteristics of B and C including the method show(). Now, when you try to invoke show() of class B, the compiler will get confused as to which show() to be invoked ( either from class B or class C ). Hence it leads to ambiguity.

**For Example:**

class Demo1 {

    // code here

}

class Demo2 {

    // code here

}

class Demo3 extends Demo1, Demo2

{

//code here

}

class Launch

{

public static void main(String args[])

{

//code here

}

}

In the above code, Demo3 is a child class which is trying to inherit two parent classes Demo1 and Demo2. This is not permitted as it results in a diamond problem and leads to ambiguity.

NOTE: Multiple inheritance is not supported in Java but you can still achieve it using interfaces.

## RULE 2: Cyclic Inheritance is NOT permitted in Java.

It is a type of inheritance in which a class extends itself and form a loop itself. Now think if a class extends itself or in any way, if it forms cycle within the user-defined classes, then is there any chance of extending the Object class. That is the reason it is not permitted in Java.

**For Example:**

class Demo1 extends Demo2 {

    // code here

}

class Demo2 extends Demo1 {

    // code here

}

In the above code, both the classes are trying to inherit each other’s characters which is not permitted as it leads to ambiguity.

## RULE 3: Private members do NOT get inherited.

**For Example:**

class You

{

private int an;

private int pw;

You{

an =111;

pw= 222;

}

}

class Friend extends You {

void change

    Data()

{

an =8888;

pw=9999;

}

}

    void disp() {

        System.out.println(an);

        System.out.println(pw);

    }

}

class Launch {

    public static void main(String args[]) {

        Friend f = new Friend();

        f.change.Data();

        f.disp();

    }

}

When you execute the above code, guess what happens, do you think the private variables an and pw will be inherited? Absolutely not. It remains the same because they are specific to the particular class.

## RULE 4: Constructors cannot be Inherited in Java.

A constructor cannot be inherited, as the subclasses always have a different name.

class A {

    A();

}

class B extends A {

    B();

}

You can do only:

B b = new B(); // and not new A()

Methods, instead, are inherited with “the same name” and can be used. You can still use constructors from A inside B’s implementation though:

class B extends A {

    B() {

        super();

    }

}

## RULE 5: In Java, we assign parent reference to child objects.

Parent is a reference to an object that happens to be a subtype of Parent, i.e.a Child Object. Why is this used? Well, in short, it prevents your code to be tightly coupled with a single class. Since the reference is of a parent class, it can hold any of its child class object i.e., it can refer to any of its child classes.

It has the following advantages:-

1. Dynamic method dispatch allows Java to support overriding of a method, which is central for run-time polymorphism.
2. It allows a class to specify methods that will be common to all of its derivatives while allowing subclasses to define the specific implementation of some or all of those methods.
3. It also allows subclasses to add its specific methods subclasses to define the specific implementation of some.

Imagine you add getEmployeeDetails to the class Parent as shown in the below code:

public String getEmployeeDetails() {

    return "Name: " + name;

}

We could override that method in Child to provide more details.

@Override

public String getEmployeeDetails() {

    return "Name: " + name + " Salary: " + salary;

}

Now you can write one line of code that gets whatever details are available, whether the object is a Parent or Child, like:

parent.getEmployeeDetails();

Then check the following code:

Parent parent = new Parent();

parent.name = 1;

Child child = new Child();

child.name = 2;

child.salary = 2000;

Parent[] employees = new Parent[] { parent, child };

for (Parent employee : employees) {

    employee.getEmployeeDetails();

}

This will result in the following output:

Name: 1   
Name: 2 Salary: 2000

Here we have used a Child class as a Parent class reference. It had a specialized behavior which is unique to the Child class, but if we invoke getEmployeeDetails(), we can ignore the functionality difference and focus on how Parent and Child classes are similar.

## RULE 6: Constructors get executed because of super() present in the constructor.

As you already know, constructors do not get inherited, but it gets executed because of the super() keyword. ‘super()’ is used to refer the extended class. By default, it will refer to the Object class. The constructor in Object does nothing. If a constructor does not explicitly invoke a super-class constructor, then Java compiler will insert a call to the no-argument constructor of the super-class by default.

This brings us to the end of this article on “Inheritance in Java”. Hope, you found it informative and it helped in adding value to your knowledge.

If you wish to check out more articles on the market’s most trending technologies like Artificial Intelligence, DevOps, Ethical Hacking, then you can refer to Edureka’s official site.

Do look out for other articles in this series which will explain the various other aspects of Java.