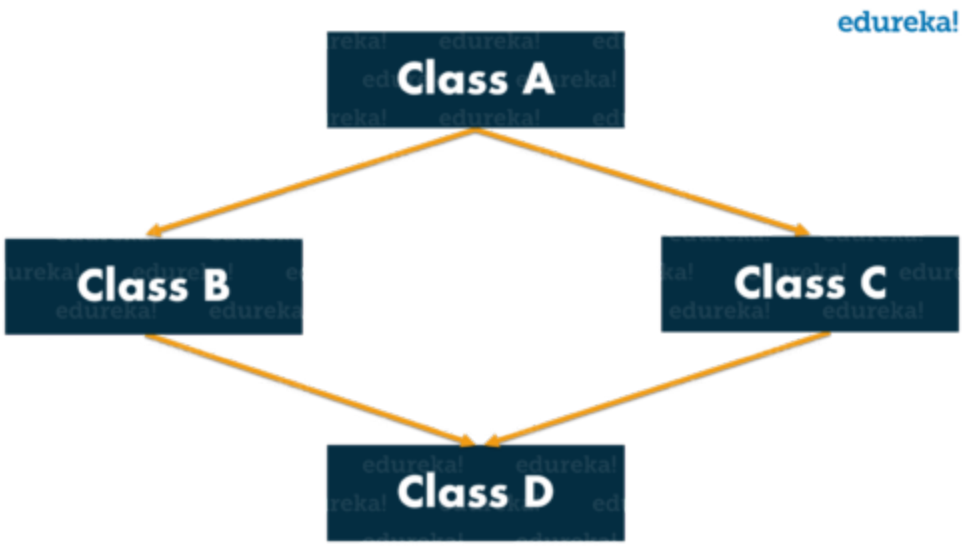
Rules of Inheritance in Java

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.