Object Oriented Programming – Java OOPs Concepts With Examples

Object Oriented programming is a programming style which is associated with the concepts like class, object, Inheritance, Encapsulation, Abstraction, Polymorphism. Most popular programming languages like Java, C++, C#, Ruby, etc. follow an object-oriented programming paradigm.

What is Object-Oriented Programming?

Object-Oriented programming (OOP) refers to a type of programming in which programmers define the data type of a data structure and the type of operations that can be applied to the data structure.

As **Java** being the most sought-after skill, we will talk about object-oriented programming concepts in Java. An object-based application in Java is based on declaring classes, creating objects from them and interacting between these objects. I have discussed Java Classes and Objects which is also a part of object-oriented programming concepts, in my previous blog.

What are the four basic principles/ building blocks of OOP (object oriented programming)?

The building blocks of object-oriented programming are Inheritance, Encapsulation, Abstraction, and Polymorphism. Let's understand more about each of them in the following sequence:

1. Inheritance
2. Encapsulation
3. Abstraction
4. Polymorphism

What are the benefits of Object Oriented Programming?

1. Improved productivity during software development
2. Improved software maintainability
3. Faster development sprints
4. Lower cost of development
5. Higher quality software

However, there are a few challenges associated with OOP, namely:

1. Steep learning curve
2. Larger program size
3. Slower program execution
4. Its not a one-size fits all solution

Let's get started with the first Object Oriented Programming concept, i.e. Inheritance.

Object Oriented Programming : Inheritance

In OOP, computer programs are designed in such a way where everything is an object that interact with one another. Inheritance is one such concept where the properties of one class can be inherited by the other. It helps to reuse the code and establish a relationship between different classes.

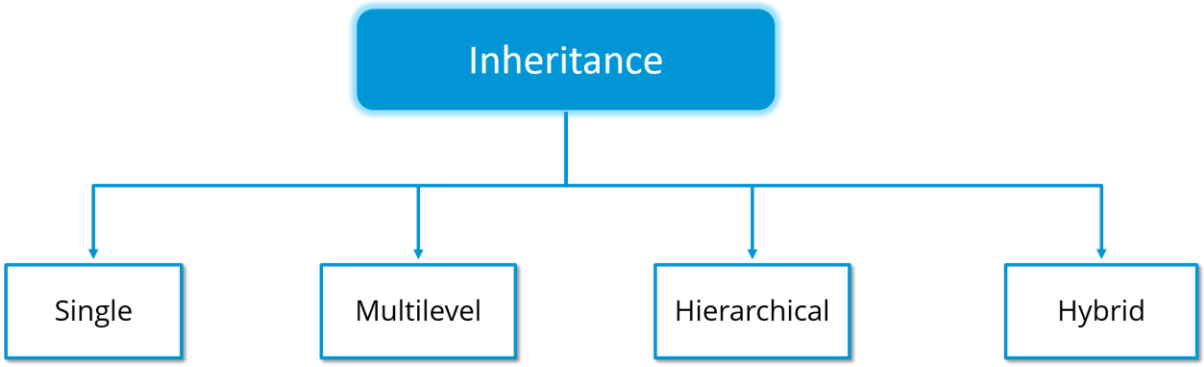
As we can see in the image, a child inherits the properties from his father. Similarly, in Java, there are two classes:

1. Parent class (Super or Base class)

2. Child class (Subclass or Derived class)

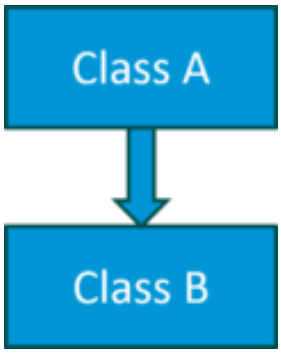
A class which inherits the properties is known as Child Class whereas a class whose properties are inherited is known as Parent class.

Inheritance is further classified into 4 types:



So let's begin with the first type of inheritance i.e. Single Inheritance:

1. **Single Inheritance:**



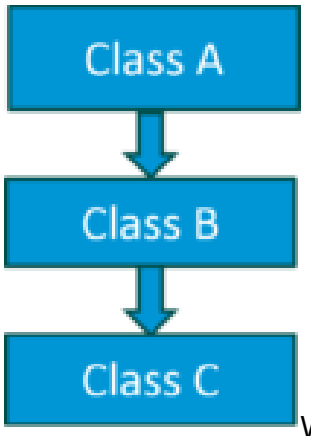
In single inheritance, one class inherits the properties of another. It enables a derived class to inherit the properties and behavior from a single parent class. This will in turn enable code reusability as well as add new features to the existing code.

Here, Class A is your parent class and Class B is your child class which inherits the properties and behavior of the parent class.

Let’s see the syntax for single inheritance

```
1Class A
2{
3---
4}
5Class B extends A {
6---
7}
```

2. Multilevel Inheritance:



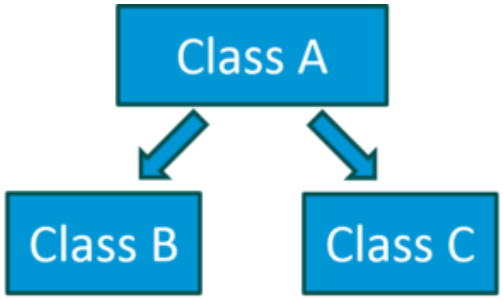
When a class is derived from a class which is also derived from another class, i.e. a class having more than one parent class but at different levels, such type of inheritance is called Multilevel Inheritance.

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.

Let’s see the syntax for multilevel inheritance in Java:

```
1Class A{
2---
3}
4Class B extends A{
5---
6}
7Class C extends B{
8---
9}
```

3. Hierarchical Inheritance:



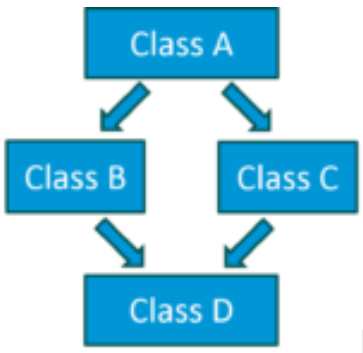
When a class has more than one child classes (sub classes) or in other words, more than one child classes have the same parent class, then such kind of inheritance is known as **hierarchical**.

If we talk about the flowchart, Class B and C are the child classes which are inheriting from the parent class i.e Class A.

Let's see the syntax for hierarchical inheritance in Java:

```
1Class A{
2---
3}
4Class B extends A{
5---
6}
7Class C extends A{
8---
9}
```

4. **Hybrid Inheritance:**



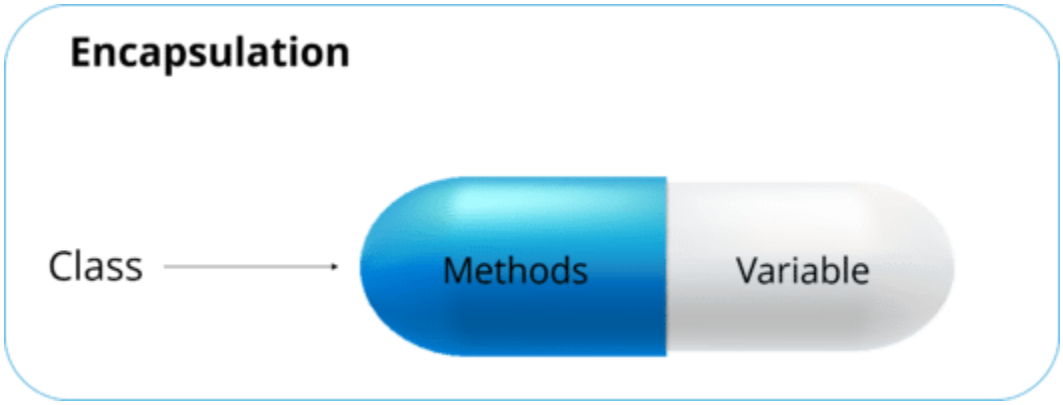
Hybrid inheritance is a combination of *multiple* inheritance and *multilevel* inheritance. Since multiple inheritance is not supported in Java as it leads to ambiguity, so this type of inheritance can only be achieved through the use of the interfaces.

If we talk about the flowchart, class A is a parent class for class B and C, whereas Class B and C are the parent class of D which is the only child class of B and C.

Now we have learned about inheritance and their different types. Let's switch to another object oriented programming concept i.e Encapsulation.

Object Oriented Programming : Encapsulation

Encapsulation is a mechanism where you bind your data and code together as a single unit. It also means to hide your data in order to make it safe from any modification. What does this mean? The best way to understand encapsulation is to look at the example of a medical capsule, where the drug is always safe inside the capsule. Similarly, through encapsulation the methods and variables of a class are well hidden and safe.



We can achieve encapsulation in Java by:

- Declaring the variables of a class as private.
- Providing public setter and getter methods to modify and view the variables values.

Let us look at the code below to get a better understanding of encapsulation:

```
 1public class Employee {
 2 private String name;
 3 public String getName() {
 4 return name;
 5 }
 6 public void setName(String name) {
 7 this.name = name;
 8 }
 9 public static void main(String[] args) {
10 }
11}
```

Let us try to understand the above code. I have created a class Employee which has a private variable **name.** We have then created a getter and setter methods through which we can get and set the name of an employee. Through these

methods, any class which wishes to access the name variable has to do it using these getter and setter methods.

Let's move forward to our third Object-oriented programming concept i.e. Abstraction.

Object Oriented Programming : Abstraction

Abstraction refers to the quality of dealing with ideas rather than events. It basically deals with hiding the details and showing the essential things to the user. If you look at the image here, whenever we get a call, we get an option to either pick it up or just reject it. But in reality, there is a lot of code that runs in the background. So you don't know the internal processing of how a call is generated, that's the beauty of abstraction. Therefore, abstraction helps to reduce complexity. You can achieve abstraction in two ways:

a) Abstract Class

b) Interface

Let's understand these concepts in more detail.

Abstract class: Abstract class in Java contains the 'abstract' keyword. Now what does the abstract keyword mean? If a class is declared abstract, it cannot be instantiated, which means you cannot create an object of an abstract class. Also, an abstract class can contain abstract as well as concrete methods. *Note*: You can achieve 0-100% abstraction using abstract class.

To use an abstract class, you have to inherit it from another class where you have to provide implementations for the abstract methods there itself, else it will also become an abstract class.

Let's look at the syntax of an abstract class:

```
1Abstract class Mobile {   // abstract class mobile
2Abstract void run();      // abstract method
```

Interface: Interface in Java is a blueprint of a class or you can say it is a collection of abstract methods and static constants. In an interface, each method is public and abstract but it does not contain any constructor. Along with abstraction, interface also helps to achieve multiple inheritance in Java. *Note*: You can achieve 100% abstraction using interfaces.

So an interface basically is a group of related methods with empty bodies. Let us understand interfaces better by taking an example of a 'ParentCar' interface with its related methods.

```
1public interface ParentCar {
2public void changeGear( int newValue);
3public void speedUp(int increment);
4public void applyBrakes(int decrement);
5}
```

These methods need be present for every car, right? But their working is going to be different.

Let's say you are working with manual car, there you have to increment the gear one by one, but if you are working with an automatic car, that time your system decides how to change gear with respect to speed. Therefore, not all my subclasses have the same logic written for *change gear*. The same case is for *speedup*, now let's say when you press an accelerator, it speeds up at the rate of 10kms or 15kms. But suppose, someone else is driving a super car, where it increment by 30kms or 50kms. Again the logic varies. Similarly for *applybrakes*, where one person may have powerful brakes, other may not.

Since all the functionalities are common with all my subclasses, I have created an interface 'ParentCar' where all the functions are present. After that, I will create a child class which implements this interface, where the definition to all these method varies.

Next, let's look into the functionality as to how you can implement this interface. So to implement this interface, the name of your class would change to any

particular brand of a Car, let's say I'll take an "Audi". To implement the class interface, I will use the 'implement' keyword as seen below:

```
 1public class Audi implements ParentCar {
 2int speed=0;
 3int gear=1;
 4public void changeGear( int value){
 5gear=value;
 6}
 7public void speedUp( int increment)
 8{
 9speed=speed+increment;
10}
11public void applyBrakes(int decrement)
12{
13speed=speed-decrement;
14}
15void printStates(){
16System.out.println("speed:"+speed+"gear:"+gear);
17}
18public static void main(String[] args) {
19// TODO Auto-generated method stub
20Audi A6= new Audi();
21A6.speedUp(50);
22A6.printStates();
23A6.changeGear(4);
24A6.SpeedUp(100);
25A6.printStates();
26}
27}
```

Here as you can see, I have provided functionalities to the different methods I have declared in my interface class. Implementing an interface allows a class to become more formal about the behavior it promises to provide. You can create another class as well, say for example BMW class which can inherit the same interface 'car' with different functionalities.

So I hope you guys are clear with the interface and how you can achieve abstraction using it.

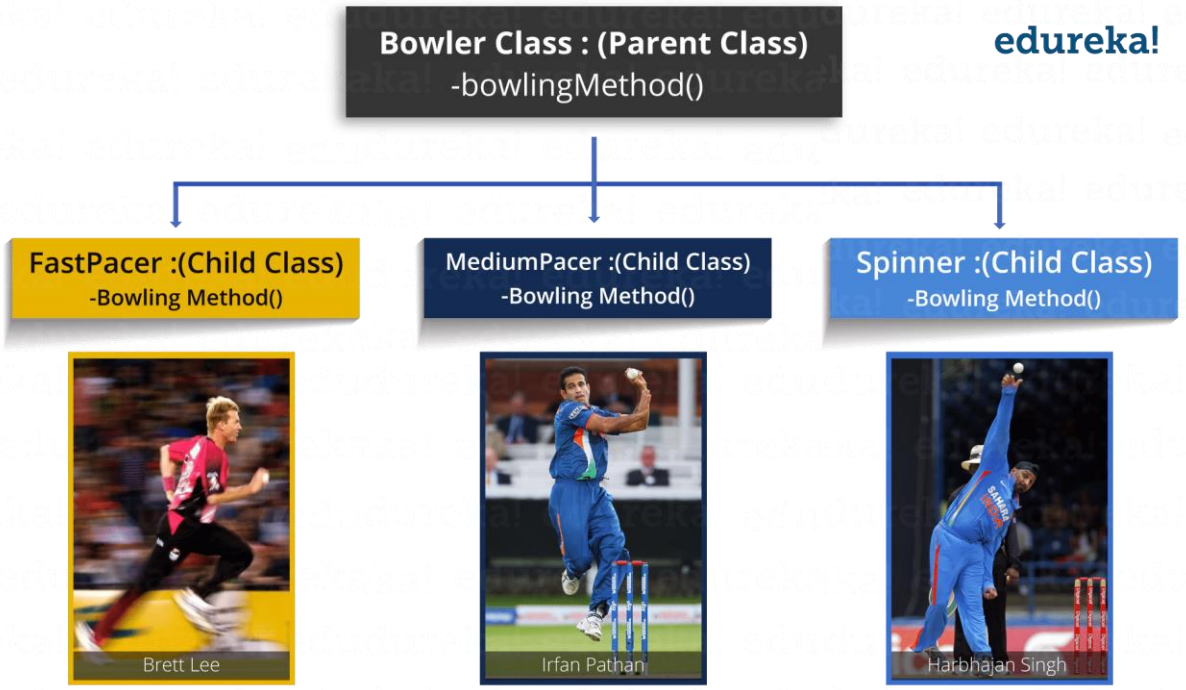
Finally, the last Object oriented programming concept is Polymorphism.

Object Oriented Programming : Polymorphism

Polymorphism means taking many forms, where 'poly' means many and 'morph' means forms. It is the ability of a variable, function or object to take on multiple

forms. In other words, polymorphism allows you define one interface or method and have multiple implementations.

Let's understand this by taking a real-life example and how this concept fits into Object oriented programming.



Let's consider this real world scenario in cricket, we know that there are different types of bowlers i.e. Fast bowlers, Medium pace bowlers and spinners. As you can see in the above figure, there is a parent class- **BowlerClass** and it has three child classes: **FastPacer**, **MediumPacer** and **Spinner**. Bowler class has **bowlingMethod()** where all the child classes are inheriting this method. As we all know that a fast bowler will going to bowl differently as compared to medium pacer and spinner in terms of bowling speed, long run up and way of bowling, etc. Similarly a medium pacer's implementation of **bowlingMethod()** is also going to be different as compared to other bowlers. And same happens with spinner class.
The point of above discussion is simply that a same name tends to multiple forms. All the three classes above inherited the **bowlingMethod()** but their implementation is totally different from one another.

Polymorphism in Java is of two types:

1. Run time polymorphism
2. Compile time polymorphism

Run time polymorphism: In Java, runtime polymorphism refers to a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this, a reference variable is used to call an overridden method of a superclass at run time. Method overriding is an example of run time polymorphism. Let us look the following code to understand how the method overriding works:

```
1public Class BowlerClass{
2void bowlingMethod()
3{
4System.out.println(" bowler ");
5}
6public Class FastPacer{
7void bowlingMethod()
8{
9System.out.println(" fast bowler ");
10}
11Public static void main(String[] args)
12{
13FastPacer obj= new FastPacer();
14obj.bowlingMethod();
15}
16}
```

Compile time polymorphism: In Java, compile time polymorphism refers to a process in which a call to an overloaded method is resolved at compile time rather than at run time. Method overloading is an example of compile time polymorphism. Method Overloading is a feature that allows a class to have two or more methods having the same name but the arguments passed to the methods are different. Unlike method overriding, arguments can differ in:

1. Number of parameters passed to a method
2. Datatype of parameters
3. Sequence of datatypes when passed to a method.

Let us look at the following code to understand how the method overloading works:

```
1class Adder {
2Static int add(int a, int b)
3{
4return a+b;
5}
6static double add( double a, double b)
7{
8return a+b;
9}
```

```
10
11public static void main(String args[])
12{
13System.out.println(Adder.add(11,11));
14System.out.println(Adder.add(12.3,12.6));
15}
16}
```

I hope you guys are clear with all the object oriented programming concepts that we have discussed above i.e inheritance, encapsulation, abstraction and polymorphism. Now you can make your Java application more secure, simple and re-usable using Java OOPs concepts. Do read my next blog on ***Java String*** where I will be explaining all about Strings and its various methods and interfaces.