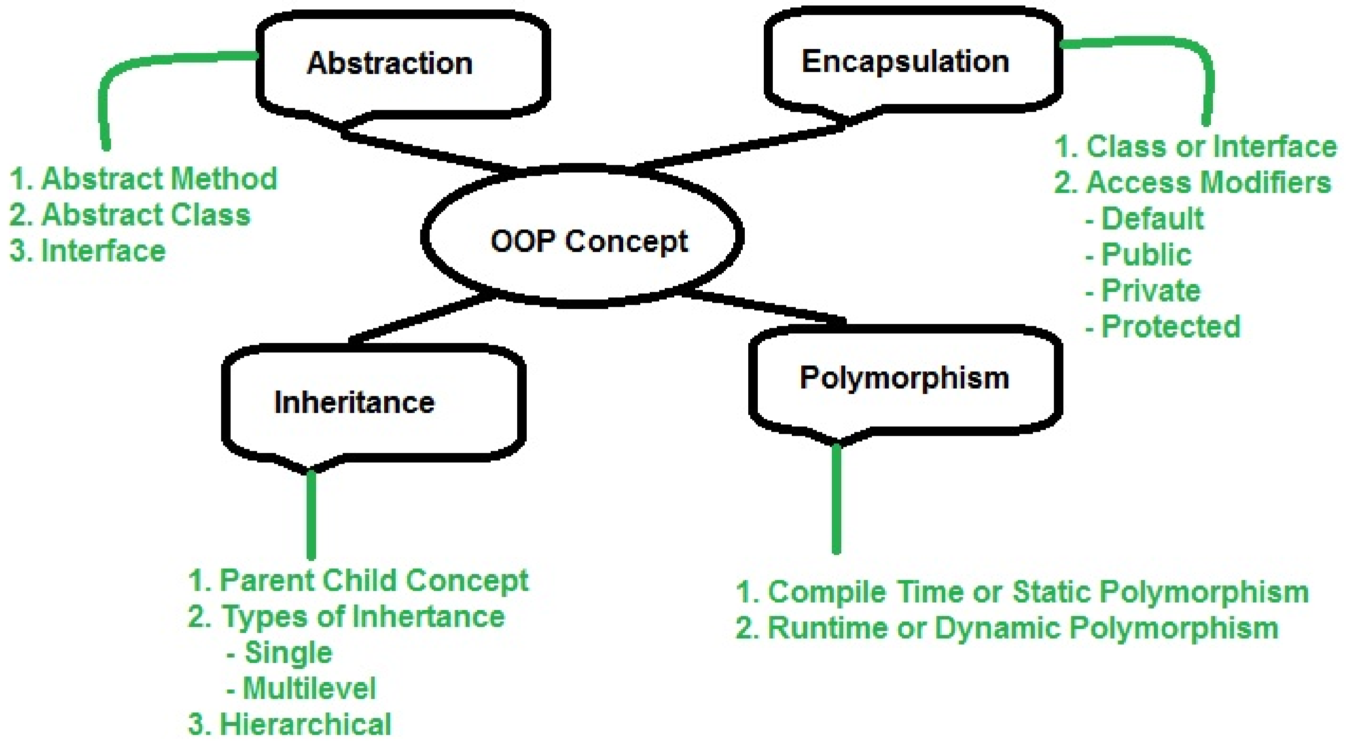
Java | Object Oriented Programming (OOPs) Concept

November 26, 2017 Ashok Kumar 3 Comments

> *Object Oriented Programming is a concept based on class and objects. A **Class** is like a repository which contains all data members and methods. On the other hand, an **Object** is an instance of a class which acquires all the properties(data members) and behavior(methods) of the class.*

OOPs, concepts revolve around following four points:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



OOPS Concept

- **Abstraction**

Abstraction and Encapsulation look similar to each other by its definition. The standard definition of abstraction is showing only necessary features to the user and hide its implementation. On the other hand, Encapsulation is hiding the complexity from the outer world.

To understand Abstraction we will start from the basic**meaning of Abstraction**. Abstract means something, which is in thoughts but not it exists in reality. It is like that we know that what should be done, but we don’t know how it would be done.

Take an example, there is a 'Shape' class which contains a method 'area()' but we cannot implement area() method here as we do not know what will be the shape(a square, a rectangle, a triangle or any other shape) in future. In that case, we will declare area() in the Shape class and will implement the same method in our subclasses as per the requirement.

Example:

```
public abstract class Shape
{
        public abstract void area(int var); // Abstract method
}

public class Circle extends Shape
{
        int radius;
        double pi=3.14;
        public void area(int var)
        {
                radius=var;
                System.out.println("Arear of Circle: "+(pi*radius*radius));
        }
}

public class Square extends Shape
{
        int side;
        public void area(int var)
        {
                side=var;
                System.out.println("Area of Square: "+side*side);
        }
}
public class Result
{
        public static void main(String[] args)
        {
                Shape obj=new Square();
                obj.area(4);

                Shape obj1=new Circle();
                obj1.area(3);
        }
}
```

In JAVA, Abstraction can be achieved using the abstract class, abstract methods, and interface. Using the abstract class we can achieve the different level of abstraction, but using the interfaces we can achieve 100% abstraction.

Abstract Class:
1. An abstract class would be declared using ‘abstract’ keyword.
2. Creation of object is not possible of an abstract class.
3. It may contain data members, methods, abstract methods, constructors.
4. Data members can’t be abstract.
5. It must be inherited by the subclass(es) using ‘extends’ keyword.
6. The subclass should implement each and every abstract method declared in Super Class. Otherwise, subclass would act like abstract class which needs to be extended further.

Abstract Method:
1. An abstract method can be placed within an abstract Class only.
2. It would be declared using ‘abstract’ keyword.
3. The abstract method should be declared only without any implementation.
4. It can be implemented only in the subclass(concrete class) using ‘extends’ or ‘implements’.

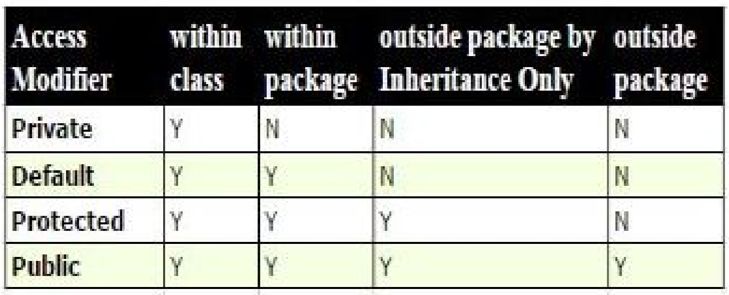
Interface:
1. Created by using ‘interface’ keyword.
2. All the methods would be abstract by default.
3. Interfaces can be implemented by subclass using ‘implements’.
4. All the methods should be implemented in subclass(concrete class).

- **Encapsulation**

In general, *Encapsulation is the process of binding the properties in a* ***single unit****. It can be achieved* ***by using the Class or Interface****.* It can be done with or without hiding any data. Fully encapsulation can be achieved only by **hiding data**. Hiding data is mainly focuses on providing the various level of access rights to the data and properties of a class. It basically protects our data from outside.

We can say that this process provides access to the data according to the requirement and as per the user. For example, taking an example of a Bank website. All the bank related information were kept in a single class ‘BankInfo’. A normal user can only see the limited details of the bank. But a user with admin rights can change the bank information.

To implement the different level of the encapsulation we have to use ‘**Access Modifier**s’ as mentioned below. If we define any data member or method as ‘private’, it will be accessible only to the class where it has been declared.



Access Modifier	within class	within package	outside package by Inheritance Only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Access Modifiers

For example, if we declare any data member as ‘private’ in a class we have to use **getter and setter** methods to providing read or write access to the outer world.
Example

```
package EncapsulationExample;
public class ABCBank
{
        private String BankName="ABC Bank Pvt. Ltd.";
        private static String BankAddress="Plot No. 6, New Delhi.";

        protected void setBankAddress(String newAddress)  // protected acces
        {
                BankAddress=newAddress;
                System.out.println("Address changed.");
        }
        public static void getBankAddress()              // public accessifi
        {
                System.out.println("Bank Address: "+BankAddress);
        }
}

package EncapsulationExample;     // Access details within package
public class AdminUser extends ABCBank
{
        public static void main(String[] args)
        {
        // Protected Methods can be accessed by subclass within and outside
        // Outside the package(Without inheritance) the user can only access
                String newAddress="Plot No. 7, New Delhi.";
                ABCBank obj=new AdminUser();
                obj.setBankAddress(newAddress);
                obj.getBankAddress();
        }
}

package Encapsulation_Use;            // Access the protected method outside
import EncapsulationExample.ABCBank;  // Use Encapsulation package for norma
public class ABCBank_User
{
        public static void main(String[] args)
        {
                ABCBank obj=new ABCBank();
                obj.getBankAddress(); // User has rights to access only publ
        }
}
```

We can make data only readable by removing setter method. Also, we can make data only writeable by removing getter method from the class.

- **Inheritance**

*Inheritance works on **Parent-Child concept**. As a child get some features from their parents and some of its own. Inheritance means acquiring the properties of the parent class in the child class using **'extends'** keyword.*
'Extends' means increasing or extending the feature of parent class by using the child class.

Example:

```
package InheritanceExample;
public class CompanyDetails
{
        // Defining Parent Class which have some data members and Methods.
        String CompanyName="XYZ PVT. Ltd.";
        String CompanyAddress="Sec-58, New Delhi.";
        String CompanyPhone="02145879352";

        public void displayCompanyDetails()
        {
                System.out.println(CompanyName);
                System.out.println(CompanyAddress);
                System.out.println(CompanyPhone);
        }
}

package InheritanceExample;
public class EmployeeInfo extends CompanyDetails
{
// Defining Child class which extends the Parent, so that we can use methods
        String EmployeeCode="12345";
        String EmployeeDept="Sales";

        public void displayEmployeeInfo()
        {
                System.out.println(EmployeeCode);
                System.out.println(EmployeeDept);
        }
        public static void main(String[] args)
        {
                EmployeeInfo obj=new EmployeeInfo();
                obj.displayCompanyDetails();  // Method from Parent Class.
                obj.displayEmployeeInfo();    // Method from Child Class.
        }
}
```

The main idea behind the inheritance is as:

1. To increase the reusability of the code and feature.
2. To support runtime polymorphism

Types of Inheritance:
1. Single
    Class Parent
    Class Child extends Parent
2. Multilevel
    Class SupParent
    Class Parent extends SupParent
    Class Child extends Parent
3. Hierarchical
    Class Parent
    Class Child_1 extends Parent
    Class Child_2 extends Parent
4. Multiple (Not supported in JAVA)
    Class Parent_1
    Class Parent_2
    Class Child extends Parent_1, Parent_2
5. Hybrid (Not supported in JAVA)
    Class SupParent
    Class Parent_1 extends SupParent
    Class Parent_2 extends SupParent
    Class Child extends Class Parent_1, Class Parent_2

Note:
To avoid the ambiguity and complexity java doesn't support Multiple and Hybrid inheritance.

- **Polymorphism**

*Polymorphism means that same thing **exists in many forms** to perform various actions.*

There are two types of Polymorphism:

1. Compile Time or Static: This can be achieved by using **Method Overloading**. This is call compile time or static because compiler determines which methods need to be called during compilation of code itself. Method overloading would be done within a class. To achieve method overloading we have to declare and define multiple methods with the **same name but having different signatures** like the number of parameters, type of parameters type etc.

Example:

```
package PolymorphismExample;
public class StaticPolymorphism
{
        public void getArea(int side)
        {
                System.out.println("Area of Square: "+side*side);
        }
        public void getArea(int length,int breadth)
        {
                System.out.println("Area of Rectangle: "+length*breadth);
        }

        public static void main(String[] args)
        {
                StaticPolymorphism obj=new StaticPolymorphism();
                obj.getArea(5);
                obj.getArea(5, 3);
        }
}
```

2. Runtime or Dynamic: This can be achieved by using **Method Overriding**. This is called runtime or dynamic because compiler determines at runtime, which methods need to be called. Method Overriding can be done **using inheritance and upcasting techniques**.In this methods with **same name and signature** can be implemented many times to achieve various goals and functionality.

Example:

```
package PolymorphismExample;
public class DynamicExample
{
        public void area(int side)
        {
                System.out.println("Area of Square: "+side*side);
        }
}

package PolymorphismExample;
public class DynamicExampleResult extends DynamicExample
{
        public void area(int radius)
        {
                System.out.println("Area of Circle: "+(3.14*radius));
        }
        public static void main(String[] args)
        {
                DynamicExample obj=new DynamicExampleResult(); // Upcasting
```

```
                    obj.area(5);  // It will call above method mentioned in 'Dyn

                    // If we comment the area() method in this class, then it wi
                    Superclass(DynamicExample).
            }
    }
```

Upcasting: Upcasting is a process to call the overridden method using the reference object of the superclass.

```
Class Parent
{
    /// Lines of code.
}
Class Child extends Parent
{
      Parent obj=new Child();// Upcasting child object to parent
      Child objb=new Class Parent()// Error
}
```