ENCAPSULATION AND INHERITANCE

Encapsulation in Java

Encapsulation in Java is a fundamental concept in object-oriented programming (OOP) that refers to the bundling of data and methods that operate on that data within a single unit, which is called a class in Java. Java Encapsulation is a way of hiding the implementation details of a class from outside access and only exposing a public interface that can be used to interact with the class.

In Java, encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively. By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent.

Implementation of Java Encapsulation

Below is the example with Java Encapsulation:

```
// Java Program to demonstrate
// Java Encapsulation

// Person Class
class Person {
    // Encapsulating the name and age
    // only approachable and used using
    // methods defined
    private String name;
    private int age;

    public String getName() { return name; }
```

```
    public void setName(String name) { this.name = name; }

    public int getAge() { return age; }

    public void setAge(int age) { this.age = age; }
}

// Driver Class
public class Main {
    // main function
    public static void main(String[] args)
    {
        // person object created
        Person person = new Person();
        person.setName("John");
        person.setAge(30);

        // Using methods to get the values from the
        // variables
        System.out.println("Name: " + person.getName());
        System.out.println("Age: " + person.getAge());
    }
}
```

Output

Name: John

Age: 30

Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, that it is a protective shield that prevents the data from being accessed by the code outside this shield.

- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which it is declared.
- As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of a class private, and the class is exposed to the end-user or the world without providing any details behind implementation using the abstraction concept, so it is also known as a **combination of data-hiding and abstraction**.
- Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.
- It is more defined with the setter and getter method.

Advantages of Encapsulation

- **Data Hiding:** it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
- **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirements. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
- **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.
- **Freedom to programmer in implementing the details of the system:** This is one of the major advantage of encapsulation that it gives the programmer freedom in implementing the details of a system. The only constraint on the programmer is to maintain the abstract interface that outsiders see.

Disadvantages of Encapsulation in Java

- Can lead to increased complexity, especially if not used properly.
- Can make it more difficult to understand how the system works.
- May limit the flexibility of the implementation.

Examples Showing Data Encapulation in Java

Example 1:

Below is the implementation of the above topic:

```
// Java Program to demonstrate
// Java Encapsulation


// fields to calculate area
class Area {
    int length;
    int breadth;

    // constructor to initialize values
    Area(int length, int breadth)
    {
        this.length = length;
        this.breadth = breadth;
    }

    // method to calculate area
    public void getArea()
    {
        int area = length * breadth;
        System.out.println("Area: " + area);
    }
```

```
}

class Main {
    public static void main(String[] args)
    {

        Area rectangle = new Area(2, 16);
        rectangle.getArea();
    }
}
```

Output

Area: 32

Example 2:

The program to access variables of the class EncapsulateDemo is shown below:

```
// Java program to demonstrate
// Java encapsulation

class Encapsulate {
    // private variables declared
    // these can only be accessed by
    // public methods of class
    private String geekName;
    private int geekRoll;
    private int geekAge;

    // get method for age to access
    // private variable geekAge
```

```
    public int getAge() { return geekAge; }

    // get method for name to access
    // private variable geekName
    public String getName() { return geekName; }

    // get method for roll to access
    // private variable geekRoll
    public int getRoll() { return geekRoll; }

    // set method for age to access
    // private variable geekage
    public void setAge(int newAge) { geekAge = newAge; }

    // set method for name to access
    // private variable geekName
    public void setName(String newName)
    {
        geekName = newName;
    }

    // set method for roll to access
    // private variable geekRoll
    public void setRoll(int newRoll) { geekRoll = newRoll; }
}

public class TestEncapsulation {
    public static void main(String[] args)
    {
```

```
        Encapsulate obj = new Encapsulate();

        // setting values of the variables
        obj.setName("Harsh");
        obj.setAge(19);
        obj.setRoll(51);

        // Displaying values of the variables
        System.out.println("Geek's name: " + obj.getName());
        System.out.println("Geek's age: " + obj.getAge());
        System.out.println("Geek's roll: " + obj.getRoll());

        // Direct access of geekRoll is not possible
        // due to encapsulation
        // System.out.println("Geek's roll: " +
        // obj.geekName);
    }
}
```

Output

```
Geek's name: Harsh
Geek's age: 19
Geek's roll: 51
```

Example 3:

In the above program, the class Encapsulate is encapsulated as the variables are declared private. The get methods like getAge(), getName(), and getRoll() are set as public, these methods are used to access these variables. The setter methods like setName(), setAge(), setRoll() are also declared as public and are used to set the values of the variables.

Below is the implementation of the defined example:

```
// Java Program to demonstrate
// Java Encapsulation


class Name {
    // Private is using to hide the data
    private int age;
    // getter
    public int getAge() { return age; }
    // setter
    public void setAge(int age) { this.age = age; }
}


// Driver Class
class GFG {
    // main function
    public static void main(String[] args)
    {
        Name n1 = new Name();
        n1.setAge(19);
        System.out.println("The age of the person is: "+ n1.getAge());
    }
}
```

Output

The age of the person is: 19

Example 4:

Below is the implementation of the Java Encapsulation:

```
// Java Program to demonstrate
// Java Encapsulation

class Account {
    // private data members to hide the data
    private long acc_no;
    private String name, email;
    private float amount;
    // public getter and setter methods
    public long getAcc_no() { return acc_no; }
    public void setAcc_no(long acc_no)
    {
        this.acc_no = acc_no;
    }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getEmail() { return email; }
    public void setEmail(String email)
    {
        this.email = email;
    }
    public float getAmount() { return amount; }
    public void setAmount(float amount)
    {
        this.amount = amount;
    }
```

```
}

// Driver Class
public class GFG {
	// main function
	public static void main(String[] args)
	{
		// creating instance of Account class
		Account acc = new Account();
		// setting values through setter methods
		acc.setAcc_no(90482098491L);
		acc.setName("ABC");
		acc.setEmail("abc@gmail.com");
		acc.setAmount(100000f);
		// getting values through getter methods
		System.out.println(
			acc.getAcc_no() + " " + acc.getName() + " "
			+ acc.getEmail() + " " + acc.getAmount());
	}
}
```

Output

90482098491 ABC abc@gmail.com 100000.0

Inheritance in Java

Java, Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

Why Do We Need Java Inheritance?

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

Important Terminologies Used in Java Inheritance

- **Class:** Class is a set of objects which shares common characteristics/ behavior and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- **Super Class/Parent Class:** The class whose features are inherited is known as a **superclass(or a base class or a parent class).**
- **Sub Class/Child Class:** The class that inherits the other class is known as a **subclass(or a derived class, extended class, or child class)**. The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to Use Inheritance in Java?

The **extends keyword** is used for inheritance in Java. Using the extends keyword indicates you are derived from an existing class. In other words, “extends” refers to increased functionality.

Syntax :

```
class derived-class extends base-class
{
  //methods and fields
}
```

Inheritance in Java Example

Example: In the below example of inheritance, class Bicycle is a base class, class MountainBike is a derived class that extends the Bicycle class and class Test is a driver class to run the program.

```
// Java program to illustrate the
// concept of inheritance

// base class
class Bicycle {
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement)
    {
        speed -= decrement;
    }

    public void speedUp(int increment)
    {
```

```
        speed += increment;
    }

    // toString() method to print info of Bicycle
    public String toString()
    {
        return ("No of gears are " + gear + "\n" + "speed of bicycle is " + speed);
    }
}

// derived class
class MountainBike extends Bicycle {

    // the MountainBike subclass adds one more field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int gear, int speed,
                        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one more method
    public void setHeight(int newValue)
    {
        seatHeight = newValue;
```

```
    }
    // overriding toString() method
    // of Bicycle to print more info
    @Override public String toString()
    {
        return (super.toString() + "\nseat height is " + seatHeight);
    }
}

// driver class
public class Test {
    public static void main(String args[])
    {
        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());
    }
}
```

Output

```
No of gears are 3
speed of bicycle is 100
seat height is 25
```

In the above program, when an object of MountainBike class is created, a copy of all methods and fields of the superclass acquires memory in this object. That is why by using the object of the subclass we can also access the members of a superclass.

Please note that during inheritance only the object of the subclass is created, not the superclass. For more, refer to Java Object Creation of Inherited Class.

Example 2: In the below example of inheritance, class Employee is a base class, class Engineer is a derived class that extends the Employee class and class Test is a driver class to run the program.

```
// Java Program to illustrate Inheritance (concise)


import java.io.*;


// Base or Super Class
class Employee {
    int salary = 60000;
}


// Inherited or Sub Class
class Engineer extends Employee {
    int benefits = 10000;
}


// Driver Class
class Gfg {
    public static void main(String args[])
    {
        Engineer E1 = new Engineer();
        System.out.println("Salary : " + E1.salary + "\nBenefits : " + E1.benefits);
    }
}
```

Output

```
Salary : 60000
Benefits : 10000
```

In practice, inheritance, and polymorphism are used together in Java to achieve fast performance and readability of code.

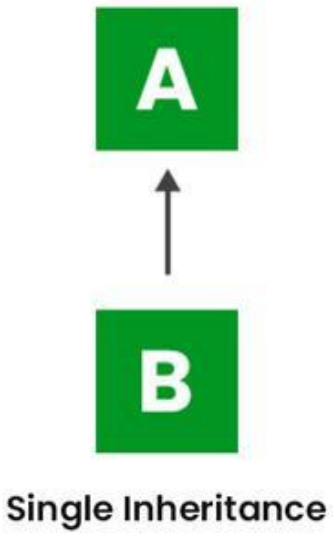
Java Inheritance Types

Below are the different types of inheritance which are supported by Java.

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

1. Single Inheritance

In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.



Single Inheritance

```
// Java program to illustrate the
// concept of single inheritance
import java.io.*;
import java.lang.*;
import java.util.*;
```

```
// Parent class
class One {
	public void print_geek()
	{
		System.out.println("Geeks");
	}
}

class Two extends One {
	public void print_for() { System.out.println("for"); }
}

// Driver class
public class Main {
	// Main function
	public static void main(String[] args)
	{
		Two g = new Two();
		g.print_geek();
		g.print_for();
		g.print_geek();
	}
}
```

Output

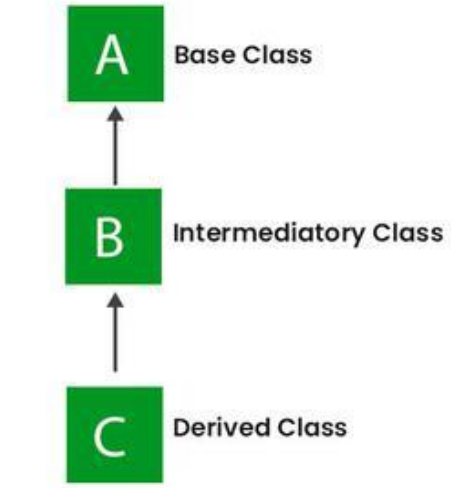
Geeks

for

Geeks

2. Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.



Multilevel Inheritance

```
// Java program to illustrate the
// concept of Multilevel inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

class One {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

class Two extends One {
    public void print_for() { System.out.println("for"); }
```

```
}

class Three extends Two {
	public void print_geek()
	{
		System.out.println("Geeks");
	}
}

// Drived class
public class Main {
	public static void main(String[] args)
	{
		Three g = new Three();
		g.print_geek();
		g.print_for();
		g.print_geek();
	}
}
```

Output

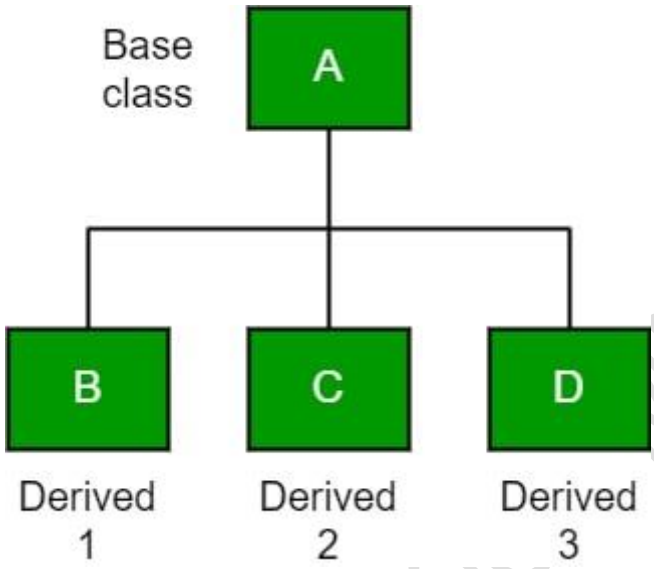
Geeks

for

Geeks

3. Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.




```
// Java program to illustrate the
// concept of Hierarchical inheritance

class A {
    public void print_A() { System.out.println("Class A"); }
}

class B extends A {
    public void print_B() { System.out.println("Class B"); }
}

class C extends A {
    public void print_C() { System.out.println("Class C"); }
}
```

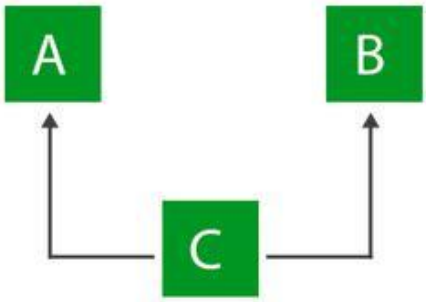
```
class D extends A {
	public void print_D() { System.out.println("Class D"); }
}
// Driver Class
public class Test {
	public static void main(String[] args)
	{
		B obj_B = new B();
		obj_B.print_A();
		obj_B.print_B();

		C obj_C = new C();
		obj_C.print_A();
		obj_C.print_C();

		D obj_D = new D();
		obj_D.print_A();
		obj_D.print_D();
	}
}
```

Output

```
Class A
Class B
Class A
Class C
Class A
Class D
```

4. Multiple Inheritance (Through Interfaces)

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.



Multiple Inheritance

```
// Java program to illustrate the
// concept of Multiple inheritance
import java.io.*;
import java.lang.*;
import java.util.*;

interface One {
        public void print_geek();
}

interface Two {
        public void print_for();
}

interface Three extends One, Two {
```

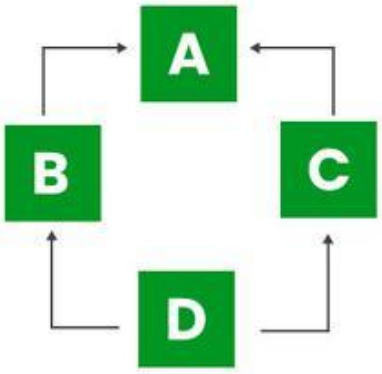
```
        public void print_geek();
}
class Child implements Three {
        @Override public void print_geek()
        {
                System.out.println("Geeks");
        }

        public void print_for() { System.out.println("for"); }
}

// Drived class
public class Main {
        public static void main(String[] args)
        {
                Child c = new Child();
                c.print_geek();
                c.print_for();
                c.print_geek();
        }
}
```

Output

```
Geeks
for
Geeks
```

5. Hybrid Inheritance

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance. However, it is important to note that Hybrid inheritance does not necessarily require the use of Multiple Inheritance exclusively. It can be achieved through a combination of Multilevel Inheritance and Hierarchical Inheritance with classes, Hierarchical and Single Inheritance with classes. Therefore, it is indeed possible to implement Hybrid inheritance using classes alone, without relying on multiple inheritance type.



Hybrid Inheritance

Java IS-A type of Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.

```
public class SolarSystem {

}

public class Earth extends SolarSystem {

}

public class Mars extends SolarSystem {

}

public class Moon extends Earth {

}
```

Now, based on the above example, in Object-Oriented terms, the following are true:-

- SolarSystem is the superclass of Earth class.
- SolarSystem is the superclass of Mars class.
- Earth and Mars are subclasses of SolarSystem class.
- Moon is the subclass of both Earth and SolarSystem classes.

```
class SolarSystem {
}
class Earth extends SolarSystem {
}
class Mars extends SolarSystem {
}
public class Moon extends Earth {
    public static void main(String args[])
    {
        SolarSystem s = new SolarSystem();
        Earth e = new Earth();
        Mars m = new Mars();

        System.out.println(s instanceof SolarSystem);
        System.out.println(e instanceof Earth);
        System.out.println(m instanceof SolarSystem);
    }
}
```

Output

```
true
true
true
```

What Can Be Done in a Subclass?

In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus overriding it (as in the example above, *toString()* method is overridden).
- We can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- We can declare new methods in the subclass that are not in the superclass.
- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Advantages Of Inheritance in Java:

1. Code Reusability: Inheritance allows for code reuse and reduces the amount of code that needs to be written. The subclass can reuse the properties and methods of the superclass, reducing duplication of code.
2. Abstraction: Inheritance allows for the creation of abstract classes that define a common interface for a group of related classes. This promotes abstraction and encapsulation, making the code easier to maintain and extend.
3. Class Hierarchy: Inheritance allows for the creation of a class hierarchy, which can be used to model real-world objects and their relationships.
4. Polymorphism: Inheritance allows for polymorphism, which is the ability of an object to take on multiple forms. Subclasses can override the methods of the superclass, which allows them to change their behavior in different ways.

Disadvantages of Inheritance in Java:

1. Complexity: Inheritance can make the code more complex and harder to understand. This is especially true if the inheritance hierarchy is deep or if multiple inheritances is used.
2. Tight Coupling: Inheritance creates a tight coupling between the superclass and subclass, making it difficult to make changes to the superclass without affecting the subclass.

Conclusion

Let us check some important points from the article are mentioned below:

- **Default superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of the Object class.
- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because Java does not support multiple inheritances with classes. Although with interfaces, multiple inheritances are supported by Java.
- **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.