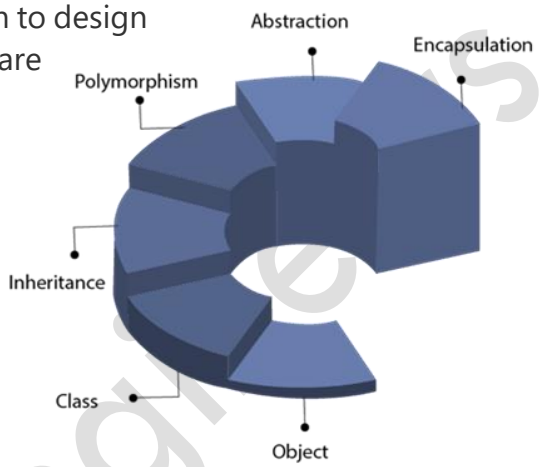
JAVA

Object Oriented Programming

Java
Object Oriented Programming

Object means a real-world entity such as a pen, chair, table, computer, watch, etc.

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:



- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Apart from these concepts, there are some other terms which are used in Object-Oriented design:

- Coupling
- Cohesion
- Association
- Aggregation
- Composition

Class

Collection of objects is called class. It is a logical entity.
A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Object

Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.
Example:

```
class Student {
   String name;
   int age;
   public void getInfo() {
       System.out.println("The name of this Student is " + this.name);
       System.out.println("The age of this Student is " + this.age);
   }
}
public class OOPS {
   public static void main(String args[]) {
       Student s1 = new Student();
       s1.name = "Aman";
       s1.age = 24;
       s1.getInfo();
        Student s2 = new Student();
```

Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as **subclass** (derived class, **child class**) and the class whose properties are inherited is known as **superclass** (base class, **parent class**).

extends Keyword

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

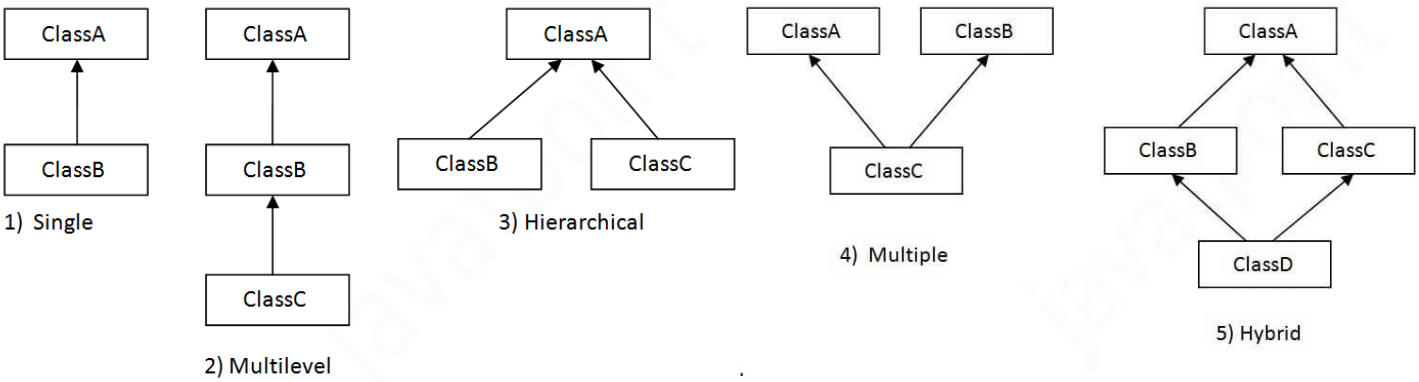
Syntax

```
class Super {
    .....
    .....
}
class Sub extends Super {
    .....
    .....
}
```

Types of inheritance in Java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only.



Types of Inheritance explained:

1. Single inheritance: When one class inherits another class, it is known as single level inheritance

```
class Shape {
   public void area() {
       System.out.println("Displays Area of Shape");
   }
}
class Triangle extends Shape {
   public void area(int h, int b) {
       System.out.println((1/2)*b*h);
   }
}
```

2. Hierarchical inheritance: Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

```
class Shape {
   public void area() {
        System.out.println("Displays Area of Shape");
   }
}
class Triangle extends Shape {
   public void area(int h, int b) {
        System.out.println((1/2)*b*h);
   }
}
class Circle extends Shape {
   public void area(int r) {
        System.out.println((3.14)*r*r);
   }
}
```

3. Multilevel inheritance: Multilevel inheritance is a process of deriving a class from another derived class.

```
class Shape {
   public void area() {
        System.out.println("Displays Area of Shape");
   }
}
class Triangle extends Shape {
   public void area(int h, int b) {
        System.out.println((1/2)*b*h);
   }
}
class EquilateralTriangle extends Triangle {
   int side;
}
```

4. Hybrid inheritance: Hybrid inheritance is a combination of simple, multiple inheritance and hierarchical inheritance.

Polymorphism

Polymorphism is the ability to present the same interface for differing underlying forms (data types). With polymorphism, each of these classes will have different underlying data. Precisely, Poly means 'many' and morphism means 'forms'.

Types of Polymorphism IMP

1. Compile Time Polymorphism (Static)
2. Runtime Polymorphism (Dynamic)

Compile Time Polymorphism: The polymorphism which is implemented at the compile time is known as **compile-time** polymorphism. Example - **Method Overloading**

Method Overloading: Method overloading is a technique which allows you to have more than one function with the same function name but with different functionality. Method overloading can be possible on the following basis:

1. The type of the parameters passed to the function.
2. The number of parameters passed to the function.

Code:

```
class Student {
   String name;
   int age;

   public void displayInfo(String name) {
       System.out.println(name);
   }

   public void displayInfo(int age) {
       System.out.println(age);
   }

   public void displayInfo(String name, int age) {
       System.out.println(name);
       System.out.println(age);
   }
}
```

Runtime Polymorphism: Runtime polymorphism is also known as **dynamic polymorphism**. **Function Overriding** is an example of **runtime** polymorphism.

Function Overriding means when the child class contains the method which is already present in the parent class. Hence, **the child class overrides the method of the parent class**. In case of function overriding, parent and child classes both contain the same function with a different definition. The call to the function is determined at runtime is known as runtime polymorphism.

Code:

```
class Shape {
   public void area() {
       System.out.println("Displays Area of Shape");
   }
}
class Triangle extends Shape {
   public void area(int h, int b) {
       System.out.println((1/2)*b*h);
   }
}
class Circle extends Shape {
   public void area(int r) {
       System.out.println((3.14)*r*r);
   }
}
```

Abstraction

Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user.

In other words, the user will have the information on what the object does instead of how it does it.

In Java, abstraction is achieved using Abstract classes and interfaces.

Abstraction is achieved in 2 ways:

- Abstract class
- Interfaces (Pure Abstraction)

Abstract Class

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Code:

```
abstract class Animal {
   abstract void walk();
   void breathe() {
       System.out.println("This animal breathes air");
   }
   Animal() {
       System.out.println("You are about to create an Animal.");
   }
}

class Horse extends Animal {
   Horse() {
       System.out.println("Wow, you have created a Horse!");
   }
   void walk() {
       System.out.println("Horse walks on 4 legs");
   }
}

class Chicken extends Animal {
   Chicken() {
       System.out.println("Wow, you have created a Chicken!");
   }
   void walk() {
       System.out.println("Chicken walks on 2 legs");
   }
}

public class OOPS {
   public static void main(String args[]) {
      Horse horse = new Horse();
      horse.walk();
      horse.breathe();
   }
}
```

Interfaces

- All the fields in interfaces are public, static and final by default.
- All methods are public & abstract by default.
- A class that implements an interface must implement all the methods declared in the interface.
- Interfaces support the functionality of multiple inheritance.

Code:

```
interface Animal {
   void walk();
}
class Horse implements Animal {
   public void walk() {
       System.out.println("Horse walks on 4 legs");
   }
}
class Chicken implements Animal {
   public void walk() {
       System.out.println("Chicken walks on 2 legs");
   }
}
public class OOPS {
   public static void main(String args[]) {
      Horse horse = new Horse();
      horse.walk();
   }
}
```

Encapsulation

Encapsulation is the process of combining data and functions into a single unit called class.
In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class. In simpler words, attributes of the class are kept private or public - **getter** and **setter** methods are provided to manipulate these attributes. Thus, encapsulation makes the concept of data hiding possible. (**Data hiding**: a language feature to restrict access to members of an object, reducing the negative effect due to dependencies. e.g., "protected", "private" feature in Java).

To achieve encapsulation in Java –

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

Code:

```
/* File name: EncapTest.java */
public class EncapTest {
   private String name;
   private String idNum;
   private int age;

   public int getAge() {
      return age;
   }

   public String getName() {
      return name;
   }

   public String getIdNum() {
      return idNum;
   }

   public void setAge( int newAge) {
      age = newAge;
   }

   public void setName(String newName) {
      name = newName;
   }

   public void setIdNum( String newId) {
      idNum = newId;
   }
}
/* The public setXXX() and getXXX() methods are the access points of the instance variables of
the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any
class that wants to access the variables should access them through these getters and setters. */
/* The variables of the EncapTest class can be accessed using the following program */

/* File name: RunEncap.java */
public class RunEncap {

   public static void main(String args[]) {
      EncapTest encap = new EncapTest();
      encap.setName("James");
      encap.setAge(20);
      encap.setIdNum("12343ms");

      System.out.print("Name: " + encap.getName() + " Age: " + encap.getAge());
   }
}
```

Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.

Advantages of Encapsulation:

- **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows 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 as read-only or write-only depending on our requirement. If we wish to make the variables as 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 as 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 easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

References,

- JavaTpoint
- GeeksForGeeks
- TutorialsPoint
- Oracle Java Documentations
- And Self Explored

Author: Raj Kumar

Must Visit,

- Explore More from YouTube@UneducatedEngineers