OBJECT ORIENTED PROGRAMMING SYSTEMS
JAVA

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

Class is a user-defined data type which defines its properties and its functions. Class is the only logical representation of the data. For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions. The class does not occupy any memory space till the time an object is instantiated.

Object is a run-time entity. It is an instance of the class. An object can represent a person, place or any other item. An object can operate on both data members and member functions.

Example 1:

```
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();
        s2.name = "Shradha";
        s2.age = 22;
        s2.getInfo();
    }
}
```

Example 2:

```
class Pen {
   String color;

   public void printColor() {
       System.out.println("The color of this Pen is " + this.color);
   }
}

public class OOPS {
   public static void main(String args[]) {
       Pen p1 = new Pen();
       p1.color = blue;

       Pen p2 = new Pen();
       p2.color = black;

       Pen p3 = new Pen();
       p3.color = red;

       p1.printColor();
       p2.printColor();
       p3.printColor();
   }
}
```

Note: When an object is created **using** a new keyword, then space is allocated for the variable in a heap, and the starting address is stored in the stack memory.

'this' keyword: 'this' keyword in Java that refers to the current instance of the class. In OOPS it is used to:

1. pass the current object as a parameter to another method
2. refer to the current class instance variable

Constructor: Constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally.

- Constructors have the same name as class or structure.
- Constructors don't have a return type. (Not even void)
- Constructors are only called once, at object creation.

There can be **three types** of constructors in Java.

1. Non-Parameterized constructor: A constructor which has no argument is known as non-parameterized constructor (or no-argument constructor). It is invoked at the time of creating an object. If we don't create one then it is created by default by Java.

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

   Student() {
       System.out.println("Constructor called");
```

```
    }
}
```

2. Parameterized constructor: Constructor which has parameters is called a parameterized constructor. It is used to provide different values to distinct objects.

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

   Student(String name, int age) {
       this.name = name;
       this.age = age;
   }
}
```

3. Copy Constructor: A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object. There is only a user defined copy constructor in Java (C++ has a default one too).

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

   Student(Student s2) {
       this.name = s2.name;
       this.age = s2.age;
   }
}
```

Note: Unlike languages like C++, Java has no Destructor. Instead, Java has an efficient garbage collector that deallocates memory automatically.

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)

Let's understand them one by one:

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.

```
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.

```
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);
   }
}
```

Inheritance

Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such a way, you can **reuse, extend or modify** the attributes and behaviors which are defined in other classes.
In Java, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Types of Inheritance:

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.

Package in Java

Package is a group of similar types of classes, interfaces and sub-packages. Packages can be built-in or user defined.

Built-in packages - java, util, io etc.

```
import java.util.Scanner;
```

```
import java.io.IOException;
```

Access Modifiers in Java

- **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

```
package newpackage;


class Account {
   public String name;
   protected String email;
   private String password;


   public void setPassword(String password) {
       this.password = password;
```

```
    }
}
public class Sample {
    public static void main(String args[]) {
        Account a1 = new Account();
        a1.name = "Apna College";
        a1.setPassword("abcd");
        a1.email = "hello@apnacollege.com";
    }
}
```

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 and 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).

Abstraction

We try to obtain an **abstract view**, model or structure of a real-life problem, and reduce its unnecessary details. With definition of properties of problems, including the data which are affected and the operations which are identified, the model abstracted from problems can be a standard solution to this type of problems. It is an efficient way since there are nebulous real-life problems that have similar properties.

In simple terms, it is hiding the unnecessary details & showing only the essential parts/functionalities to the user.

Data binding: Data binding is a process of binding the application UI and business logic. Any change made in the business logic will reflect directly to the application UI.

Abstraction is achieved in 2 ways:

- Abstract class
- Interfaces (Pure Abstraction)

1. **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.

```
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();
```

```
    }
}
```

2. **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.

```
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();

    }

}
```

Static Keyword Static can be:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

```
class Student {

    static String school;

    String name;

}

public class OOPS {
    public static void main(String args[]) {
        Student.school = "JMV";
        Student s1 = new Student();
        Student s2 = new Student();

        s1.name = "Meena";
        s2.name = "Beena";

        System.out.println(s1.school);
        System.out.println(s2.school);
    }
}
```