Object-Oriented Programming (OOP) in Java

What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to represent data and methods to manipulate that data. The main principles of OOP are:

1. **Class**
   A class is a blueprint for creating objects.

2. **Object**
   An object is an instance of a class.

3. **Encapsulation:**
   Bundling data (attributes) and methods (functions) that operate on the data within one unit (class). This hides the internal state and requires all interaction to be performed through an object's methods.

4. **Abstraction:**
   Hiding complex implementation details and showing only the essential features of an object.

5. **Inheritance:**
   Mechanism by which one class can inherit fields and methods from another class. This promotes code reusability.

6. **Polymorphism:**
   The ability of different classes to be treated as instances of the same class through a common interface. This allows methods to do different things based on the object it is acting upon.

Examples of OOP

1. Class and Object

A class is a blueprint for creating objects & an object is an instance of a class.

```
// Class definition
class Car
 {
        // Attributes
        String color;
        String model;

        // Method
        void displayInfo()
        {
                System.out.println("Car model: " + model + ", Color: " + color);
        }
}

// Using the class to create an object

public class Main
{
        public static void main(String[] args)
        {
                Car myCar = new Car();          // Creating an object
                myCar.color = "Red";            // Setting attributes
                myCar.model = "Toyota";
                myCar.displayInfo();            // Calling method
        }
}
```

---

2. Encapsulation

Encapsulation involves restricting direct access to some of an object's components.

```
class   BankAccount
{
        private double  balance;                    // Private variable

        // Constructor
        public   BankAccount(double initialBalance)
        {
                balance = initialBalance;
        }

        // Method to deposit money
        public void deposit(double amount)
        {
                 if (amount > 0)
                {
                        balance += amount;
                }
        }

        // Method to check balance
        public double getBalance()
        {
                return balance;
        }
}

// Using the class
public class Main
{
        public static void main(String[] args)
        {
                BankAccount account = new BankAccount(1000);
                account.deposit(500);
                System.out.println("Balance: " + account.getBalance()); // Output: Balance: 1500.0
   }
}
```

3. **Inheritance**

Inheritance allows a new class to inherit properties and methods from an existing class.

```
class Animal
{
	void sound()
	{
		System.out.println("Animal makes a sound");
	}
}

// Dog class inherits from Animal
class Dog extends Animal
{
	void sound()
	{
		System.out.println("Dog barks");
	}
}

// Using inheritance
public class Main
{
	public static void main(String[] args)
	{
		Dog myDog = new Dog();
		myDog.sound(); // Output: Dog barks
	}
}
```

4. **Polymorphism**

Polymorphism allows methods to be used in different ways depending on the object that calls them.

```
class Shape
{
        void draw()
        {
                System.out.println("Drawing a shape");
        }
}

class Circle extends Shape
{
        void draw()
        {
                System.out.println("Drawing a circle");
        }
}

class Square extends Shape
{
        void draw()
        {
                 System.out.println("Drawing a square");
         }
}

// Using polymorphism
public class Main
{
        public static void main(String[] args)
        {
                Shape myShape;

                myShape = new Circle();
                myShape.draw();                 // Output: Drawing a circle

                 myShape = new Square();
                myShape.draw();                 // Output: Drawing a square
   }
}
```

5. **Abstraction**

Abstraction allows us to define complex logic in a way that is hidden from the user.

```
abstract class Animal
{
	abstract void sound();		// Abstract method

	void sleep()
	{
		System.out.println("Sleeping...");
	}
}

class Cat extends Animal
{
	void sound()
	{
		System.out.println("Cat meows");
	}
}

// Using abstraction
public class Main
{
	public static void main(String[] args)
	{
		Animal myCat = new Cat();
		myCat.sound(); // Output: Cat meows
		myCat.sleep(); // Output: Sleeping...
	}
}
```

Summary

- **Encapsulation** : Protects data using access modifiers (private, public).
- **Inheritance** : Allows new classes to extend existing ones.
- **Polymorphism** : Enables methods to act differently based on the object type.
- **Abstraction** : Simplifies complex systems by exposing only necessary parts.

These concepts form the foundation of OOP in Java. As you practice, you'll get more comfortable with using these principles in your coding!

Access modifiers for classes or interfaces

- In Java, methods and data members can be encapsulated by the following four access modifiers. The access modifiers are listed according to their restrictiveness order.

  1) **private** (accessible within the class where defined)

  2) **default** or package-private (when no access modifier is specified)

  3) **protected** (accessible only to classes that subclass your class directly within the current or different package)

  4) **public** (accessible from any class)

  But, the classes and interfaces themselves can have only two access modifiers when declared outside any other class.
  1) **public**
  2) **default** (when no access modifier is specified)

  Note: *Nested interfaces and classes can have all access modifiers.*
  Note: *We cannot declare class/interface with private or protected access modifiers.*