Oops concept in java

What is OOps:

OOPS stands for Object-Oriented Programming Language. The main purpose of the OOPS is to deal with the real-world entity using programming language.

To reduce the complexity of the program, the OOPS concept will be created.

OOPS features

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

1. **Class:**

   Class is a collection of objects and it does not take any space on the memory. Class is also called a blueprint of an object. There are two types of the class are there, predefined and user-defined.

   Class are of 2 types

   - Pre-define(scanner,system,String)
   - User-defined(A,Dog,Test)

2.Object:

Object is an instance of the class that executes a class once the object is created. It takes up the space like the other variable in memory.

Syntax:

Class_name obj_name=new class_name()

4. ABSTRACTION

Abstraction in Java is a fundamental concept in object-oriented programming that helps simplify complex systems by hiding unnecessary details and showing only the essential features to the user. Think of it as a way to focus on what something does, without worrying about how it does it.

Everyday Example of Abstraction

Imagine you're driving a car. You know how to start the car, accelerate, brake, and steer. You don't need to know how the engine works internally, how the fuel is combusted, or how the electrical system is wired. All those details are hidden from you, allowing you to focus on driving. This is what abstraction does in programming—it hides the complicated stuff and lets you work with a simpler interface.

How Abstraction Works in Java

1. **Abstract Classes**:
   - An **abstract class** is a special kind of class that can't be directly instantiated, meaning you can't create an object of it. Instead, it serves as a blueprint for other classes.
   - It can contain both:
     - **Abstract methods**: Methods that have no body; they only have a declaration. These methods must be implemented by subclasses (classes that extend the abstract class).
     - **Concrete methods**: Methods with a body that can be used as-is or overridden by subclasses.
   - Example:

```
// Abstract class
abstract class Animal {
   // Abstract method (no body)
```

```
    abstract void sound();

    // Concrete method
    void sleep() {
        System.out.println("Animal is sleeping");
    }
}

// Subclass (inheriting from Animal)
class Dog extends Animal {
    // Providing the implementation for the abstract method
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.sound(); // Outputs: Dog barks
        myDog.sleep(); // Outputs: Animal is sleeping
    }
}
```

- In this example, the Animal class is abstract and defines a method sound() without a body. The Dog class extends Animal and provides its own implementation of the sound() method. The sleep() method is available to Dog because it's inherited from Animal.

2. **Interfaces**:
   - An **interface** in Java is like a contract that a class can agree to. It defines a set of abstract methods that a class must implement if it agrees to the contract.
   - Unlike abstract classes, a class can implement multiple interfaces, which allows a form of multiple inheritance in Java.
   - Example:

```
// Interface
interface Animal {
    void sound(); // Abstract method
}

// Class implementing the interface
class Cat implements Animal {
    public void sound() {
        System.out.println("Cat meows");
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Cat myCat = new Cat();
        myCat.sound(); // Outputs: Cat meows
    }
}
```

- Here, the Animal interface declares a method sound(). The Cat class implements the Animal interface and provides its specific version of the sound() method.

- **Why Use Abstraction?**

- **Simplicity**: By using abstraction, you can work with objects at a high level, without needing to know the complex details of their implementation.
- **Reusability**: Abstract classes and interfaces allow you to reuse code. You can define common behaviors in a base class or interface, and then reuse those behaviors in multiple classes.
- **Maintainability**: Abstraction makes it easier to maintain and update your code. If the implementation details change, as long as the abstract methods and interfaces stay the same, other parts of your program won't be affected.

In short, abstraction in Java helps you write cleaner, more understandable, and more maintainable code by allowing you to focus on what an object does, without worrying about how it does it.

5. Encapsulation

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP) in Java. It is the concept of bundling the data (fields) and the methods (functions) that operate on the data into a single unit or class, while restricting access to some of the object's components. This means that the internal representation of an object is hidden from the outside world, and only a controlled interface is exposed.

Key Points of Encapsulation:

1. **Private Data Members:** Data (variables) are kept private so that they can't be accessed directly from outside the class.
2. **Public Methods:** Provide public methods (getters and setters) to access and update the value of the private data.

Real-Life Example of Encapsulation:

Let's consider a real-life example of a **Bank Account**.

- **Data:** A bank account has a balance.
- **Operations:** You can deposit money, withdraw money, and check the balance.

In Java, encapsulation is achieved by making the balance field private and providing public methods to interact with it.

EXAMPLE:

```
class A {

    // Private field - encapsulated data

    private int value;

    // Public method to set the value

    public void setvalue(int x) {

        value = x;

    }

    // Public method to get the value

    public int getvalue() {

        return ++value;  // Increment the value by 1 and then return it

    }

}

public class B {

    public static void main(String[] args) {

        A r = new A();  // Create an instance of class A

        r.setvalue(100);  // Set the value to 100

        System.out.println(r.getvalue());  // Get the incremented value and print it

    }

}
```