Object-Oriented Programming (OOPS-1)

Introduction to OOPS

Object-oriented programming System(OOPs) is a programming paradigm based on the concept of *"objects"* and *"classes"* that contain data and methods. The primary purpose of OOP is to increase the flexibility and maintainability of programs. It is used to structure a software program into simple, reusable pieces of code **blueprints** (called *classes*) which are used to create individual instances of *objects*.

What is an Object?

The object is an entity that has a state and a behavior associated with it. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.
Objects have states and behaviors. Arrays are objects. You've been using objects all along and may not even realize it. Apart from primitive data types, objects are all around in java.

What is a Class?

A class is a **blueprint** that defines the variables and the methods (Characteristics) common to all objects of a certain kind.

Example: If **Car** is a class, then **Maruti 800** is an object of the **Car** class. All cars share similar features like 4 wheels, 1 steering wheel, windows, breaks etc. Maruti 800 (The **Car** object) has all these features.

Classes vs Objects (Or Instances)

Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.

In this module, you'll create a **Car** class that stores some information about the characteristics and behaviors that an individual **Car** can have.

A class is a blueprint for how something should be defined. It doesn't contain any data. The **Car** class specifies that a name and a top-speed are necessary for defining a **Car**, but it doesn't contain the name or top-speed of any specific **Car**.

While the class is the blueprint, an instance is an object that is built from a class and contains real data. An instance of the **Car** class is not a blueprint anymore. It's an actual car with a **name**, like Creta, and with a **top speed** of 200 Km/Hr.

Put another way, a class is like a form or questionnaire. An **instance** is like a form that has been filled out with information. Just like many people can fill out the same form with their unique information, many instances can be created from a single class.

Defining a Class in Java

All class definitions start with the `class` keyword, which is followed by the name of the class.
Here is an example of a **Car** class:

```
class Car{

}
```

Note: Java class names are written in *CapitalizedWords* notation by convention. **For example**, a class for a specific model of Car like the Bugatti Veyron would be written as **BugattiVeyron**. The first letter is capitalized. This is just a good programming practice.

The **Car** class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all Car objects should have. There are several properties that we can choose from, including **color**, **brand**, and **top-speed**. To keep things simple, we'll just use **color** and **top-speed**.

Constructor

- Constructors are generally used for instantiating an object.
- The task of a constructor is to initialize(assign values) to the data members of the class when an object of the class is created.
- In Java, constructor for a class must be of the same name as of class.
- In Java, constructors don't have a return type.

Syntax of Constructor Declaration

```
public Car(){
    // body of the constructor
}
```

Types of constructors

- **Default Constructor:** The default constructor is a simple constructor that doesn't accept any arguments.
- **Parameterized Constructor:** A constructor with parameters is known as a parameterized constructor. The parameterized constructor takes its arguments provided by the programmer.

Class Attributes or Data Members

Class attributes are attributes that may or may not have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `constructor.`

For example, the following **Car** class has a class attribute called `name` and `topSpeed` with the value `"Black"`:

```
class Car{
    String name;
    int topSpeed;

    public Car(String carName, int speed){
        name = carName;
        topSpeed = speed;
    }
}
```

- Class attributes are defined directly beneath the first line of the class name.
- When an instance of the class is created, the class attributes are automatically created.

The `this` Parameter

- The `this` parameter is a reference to the current instance of the class and is used to access variables that belong to the class.
- We can use `this` every time but the main use of `this` comes in picture when the attributes and data members share the same name.

Let's update the Car class with the `Car` method that creates `name` and `topSpeed` attributes:

```
class Car{
```

```
    String name;
    int topSpeed;

    public Car(String name, int topSpeed){
        this.name = name;
        this.topSpeed = topSpeed;
    }
}
```

In the body of `constructor`, two statements are using the self variable:

1. `this.name = name` assigns the value of the `name` parameter to `name` attribute.
2. `this.topSpeed= topSpeed` assigns the value of the `topSpeed` parameter to `topSpeed` attribute.

All **Car** objects have a `name` and a `topSpeed`, but the values for the `name` and `topSpeed` attributes will vary depending on the **Car** instance. Different objects of the **Car** class will have different names and top speeds.

Now that we have a **Car** class, let's create some cars!

Instantiating an Object in Java

Creating a new object from a class is called instantiating an object. Consider the previous simpler version of our **Car** class:

```
Car c1 = new Car("Creta", 200);
```

You can instantiate a new **Car** object by typing the name of the class, followed by opening and closing parentheses:

Now, instantiate a second **Car** object:

```
Car c2 = new Car("i 10", 190);
```

The new **Car** instance is located at a different memory address. That's because it's an entirely new instance and is completely different from the first **Car** object that you instantiated.

Even though **c1** and **c2** are both instances of the **Car** class, they represent two distinct objects in memory. So they are not equal.

After you create the **Car** instances, you can access their instance attributes using dot notation:

```
System.out.println(c1.name);
System.out.println(c2.topSpeed);
```

Output will be:

```
Creta
190
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. The values of these attributes ***can*** be changed dynamically:

```
c1.topSpeed= 250
c2.name = "Jeep"
```

In this example, you change the `topSpeed` attribute of the `c1` object to **250**. Then you change the `name` attribute of the `c2` object to `"Jeep"`.

Note:

- The key takeaway here is such custom objects are **mutable** by default i.e. their states can be modified.

Access Modifiers

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **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.
3. **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.
4. **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

We write the type of modifier before every method or data member.

Final Keyword

If you make any variable final, you cannot change the value of the final variable (It will be constant).

```
class Pen{
    final int price = 15;
}

public class MCQs {
    public static void main(String[] args) {
        Pen p = new Pen();
        p.price = 20;
        System.out.println(p.price);
    }

}
```

There is a final variable price, we are going to change the value of this variable, but it can't be changed because the final variable once assigned a value can never be changed. Therefore this will give a **compile time error**.

Static Keyword

The static variable gets memory only once in the class area at the time of class loading.

The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class rather than an instance of the class.

```
class Car{
    static int year;
    String company_name;

}

class NewCar{
    public static void main (String[] args) {
        Car c=new Car();
        Car.year=2018;
        c.company_name="KIA";
        Car d=new Car();
        System.out.print(d.year);

    }

}
```

Now in this code, when we look carefully, even when the new instance of Car is created, the year is defined by the first instance of the Car and it tends to remain

the same for all instances of the object. But here's the catch, we can change the value of this static variable from any instance. Here the output will be 2018 for every instance as long as it is not changed.

Static Functions

If you apply a static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data members and can change the value of it.

```
class Test{
    static int a = 10;
    static int b;
    static void fun(){
        b = a * 4;
    }
}

class NewCar{
    public static void main(String[] args) {
      Test t=new Test();
      t.fun();
      System.out.print(t.a + t.b);
    }

}
```

When t.fun() is called, it will simply change the value of b to 40.

Therefore the output of this code will be 50.