OOPs in JAVA

What is Object-Oriented Programming?

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

Object-oriented programming (OOP) is a fundamental programming paradigm based on the concept of “objects”. These objects can contain data in the form of fields (often known as attributes or properties) and code in the form of procedures (often known as methods).

The core concept of the object-oriented approach is to break complex problems into smaller objects.

What is a Class in Java?

A class is a structure that defines the data and the methods to work on that data. When you write programs in the Java language, all program data is wrapped in a class, whether it is a class you write or a class you use from the Java platform API libraries.

Classes in the Java platform API libraries define a set of objects that share a common structure and behavior.

A class is defined as a collection of objects. You can also think of a class as a blueprint from which you can create an individual object.

To create a class, we use the keyword class.

Syntax:

```
class ClassName {
  // fields
  // methods
}
```

What is an Object in Java?

Object is termed as an instance of a class, and it has its own state, behavior and identity.

An object is an entity in the real world that can be distinctly identified.

An object consists of:

- **A unique identity:** Each object has a unique identity, even if the state is identical to that of another object.
- **State/Properties/Attributes:** State tells us how the object looks or what properties it has.
- **Behavior:** Behavior tells us what the object does.

Examples of object states and behaviors in Java:

Example 1:

- Object: car.
- State: color, brand, weight, model.
- Behavior: break, accelerate, turn, change gears.

Example 2:

- Object: house.
- State: address, color, location.
- Behavior: open door, close door, open blinds.

Syntax of an object:

```
public class Number {

        int y = 10;

        public static void main(String[] args) {

                Number myObj = new Number();

                System.out.println(myObj.y);

        }

}
```

Method Overloading

Method Overloading allows a class to have more than one method with the same name, as long as their parameter lists are different. This is a way of implementing polymorphism at compile time. Overloading increases the readability of the program.

Rules for Method Overloading

- **Different Parameters**: The methods must differ in the type or number of their parameters.
- **Return Type**: The return type can be different but is not considered in the method signature for overloading.
- **Access Modifier**: It can be different, but it is not considered in the method signature for overloading.

Example of Method Overloading

```
class OverloadExample {
    // Method with 2 int parameters
    public int add(int a, int b) {
        return a + b;
    }

    // Method with 3 int parameters
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method with 2 double parameters
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        OverloadExample example = new OverloadExample();

        // Calling add method with different parameters
        System.out.println("Sum of 2 and 3: " + example.add(2, 3));             // Outputs 5
        System.out.println("Sum of 1, 2 and 3: " + example.add(1, 2, 3));       // Outputs 6
        System.out.println("Sum of 2.5 and 3.5: " + example.add(2.5, 3.5));     // Outputs 6.0
    }
}
```

What are Constructors in Java?

A constructor in Java is similar to a method that is invoked when an object of the class is created.

Unlike Java methods, a constructor has the same name as that of the class and does not have any return type.

Syntax:

```
class Test {
  Test() {
    // constructor body
  }
}
```

Example:

```
class Main {
  private String name;

  // constructor
  Main() {
    System.out.println("Constructor Called:");
    name = "PK";
  }

  public static void main(String[] args) {

    // constructor is invoked while
    // creating an object of the Main class
    Main obj = new Main();
    System.out.println("The name is " + obj.name);
  }
}
```

Types of Constructors:

In Java, constructors can be divided into three types:

- No-Arg Constructor
- Parameterized Constructor
- Default Constructor

1. Java No-Arg Constructors

It is Similar to methods, a Java constructor may or may not have any parameters (arguments).

If a constructor does not accept any parameters, it is known as a no-argument constructor.

Syntax:

```
private Constructor() {
  // body of the constructor
}
```

Example: Java Private No-arg Constructor

```
class Main {
 int i;
 // constructor with no parameter
 private Main() {
  i = 5;
  System.out.println("Constructor is called");
 }

 public static void main(String[] args) {
  // calling the constructor without any parameter
  Main obj = new Main();
  System.out.println("Value of i: " + obj.i);
 }
}
```

Once a constructor is declared private, it cannot be accessed from outside the class. So, creating objects from outside the class is prohibited using the private constructor. Here, we are creating the object inside the same class. Hence, the program is able to access the constructor.

However, if we want to create objects outside the class, then we need to declare the constructor as public.

```
class Company {
  String name;
  // public constructor
  public Company() {
    name = "Programiz";
  }
}

class Main {
  public static void main(String[] args) {
    // object is created in another class
    Company obj = new Company();
    System.out.println("Company name = " + obj.name);
  }
}
```

2. Java Parameterized Constructor

A Java constructor can also accept one or more parameters. Such constructors are known as parameterized constructors (constructors with parameters).

Based on the argument passed, the language variable is initialized inside the constructor.

Example: Parameterized Constructor

```
class Main {
  String languages;
  // constructor accepting single value
  Main(String lang) {
    languages = lang;
    System.out.println(languages + " Programming Language");
  }
```

```
  public static void main(String[] args) {
    // call constructor by passing a single value
    Main obj1 = new Main("Java");
    Main obj2 = new Main("Python");
    Main obj3 = new Main("C");
  }
}
```

3. Java Default Constructor

If we do not create any constructor, the Java compiler automatically creates a no-arg constructor during the execution of the program.

This constructor is called the default constructor.

Example: Default Constructor

```
class Main {
  int a;
  boolean b;
  public static void main(String[] args) {
    // calls default constructor
    Main obj = new Main();
    System.out.println("Default Value:");
    System.out.println("a = " + obj.a);
    System.out.println("b = " + obj.b);
  }
}
```

The default constructor initializes any uninitialized instance variables with default values.

Type	Default Value
boolean	false
byte	**0**
short	**0**
int	**0**
long	**0L**
char	\u0000
float	**0.0f**
double	**0.0d**
object	Reference null

Important Notes on Java Constructors

- Constructors are invoked implicitly when you instantiate objects.
- The two rules for creating a constructor are:
  1. The name of the constructor should be the same as the class.
  2. A Java constructor must not have a return type.
- If a class doesn't have a constructor, the Java compiler automatically creates a default constructor during run-time. The default constructor initializes instance variables with default values. For example, the int variable will be initialized to 0
- Constructor types:
  No-Arg Constructor - a constructor that does not accept any arguments
  Parameterized constructor - a constructor that accepts arguments
  Default Constructor - a constructor that is automatically created by the Java compiler if it is not explicitly defined.
- A constructor cannot be abstract or static or final.
- A constructor can be overloaded but cannot be overridden.

Constructor Overloading

Constructor Overloading in Java is a technique where a class has more than one constructor with different parameter lists. The compiler differentiates these constructors by the number of parameters and their types.

- Overloaded constructors have the same name but different parameter lists.
- They enable different ways to initialize objects of a class.

Example:

```
class Box {
   double width, height, depth;

   // Constructor with no parameters
   Box() {
      width = height = depth = 0;
   }

   // Constructor with three parameters
   Box(double w, double h, double d) {
      width = w;
      height = h;
      depth = d;
   }

   // Constructor with one parameter (cube)
   Box(double len) {
      width = height = depth = len;
   }

   double volume() {
      return width * height * depth;
   }
```

```
    public static void main(String[] args) {
        // Creating Box objects using different constructors
        Box box1 = new Box();
        Box box2 = new Box(10, 20, 30);
        Box box3 = new Box(5);

        System.out.println("Volume of box1: " + box1.volume());
        System.out.println("Volume of box2: " + box2.volume());
        System.out.println("Volume of box3: " + box3.volume());
    }
}
```

Output:

Volume of box1: 0.0

Volume of box2: 6000.0

Volume of box3: 125.0