Class 07

Introduction to Java Class, Object, and Object-Oriented Principles

Introduction to Object-Oriented Programming with Java

- What is Object-Oriented Programming (OOP)?
- Advantages of OOP

What is Object-Oriented Programming (OOP)? Object-Oriented Programming, commonly known as OOP, is a programming paradigm or methodology that organizes and structures code based on real-world objects and their interactions. In OOP, software is designed by modeling it as a collection of objects that represent various entities, both physical and conceptual, within the application's domain.

Key concepts of OOP include:

1. **Objects:** These are instances of classes and represent real-world entities. Objects encapsulate data (attributes) and behaviors (methods).
2. **Classes:** Classes are blueprint or templates for creating objects. They define the structure and behavior of objects.
3. **Encapsulation:** Encapsulation is the concept of bundling data (attributes) and the methods (functions) that operate on that data into a single unit called a class. It restricts direct access to some of an object's components, providing data security.
4. **Inheritance:** Inheritance allows a class (subclass or derived class) to inherit the properties and behaviors of another class (superclass or base class). It promotes code reuse.
5. **Polymorphism:** Polymorphism enables objects of different classes to be treated as objects of a common superclass. It allows different classes to have methods with the same name, but each class provides its implementation.
6. **Abstraction:** Abstraction involves simplifying complex reality by modeling classes based on their essential features. It hides the complex reality while exposing only the necessary parts.

Advantages of OOP

OOP offers several advantages, making it a popular programming paradigm:

1. **Modularity:** OOP promotes modularity by breaking down complex systems into smaller, manageable parts (objects). Each object is responsible for specific functionality, making code easier to understand and maintain.

2. **Reusability:** Through inheritance and polymorphism, OOP allows for the reuse of code. Existing classes and their functionalities can be extended and reused in new classes, saving development time.
3. **Encapsulation:** Encapsulation provides data security by restricting direct access to an object's attributes. It ensures that data is accessed and modified through well-defined methods, preventing unintended data corruption.
4. **Flexibility and Extensibility:** OOP systems are highly adaptable. New classes can be created by extending existing ones, and modifications can be made without affecting other parts of the code.
5. **Clear and Understandable Code:** OOP encourages modeling real-world scenarios, resulting in code that mirrors the problem domain. This makes the code easier to understand, even for those not familiar with the program.
6. **Maintenance and Scalability:** OOP simplifies maintenance and scalability. Changes can be made to specific classes without affecting the entire system, making it easier to manage and update.

In summary, Object-Oriented Programming (OOP) is a programming idea that uses objects and classes to design and structure software. It offers numerous benefits, including modularity, reusability, encapsulation, flexibility, readability, and easier maintenance. Java is one of the most widely used languages that follows the OOP principles, making it a powerful tool for developing complex and robust applications.

2. **Java Class and Object**
    - Definition of a Class
    - Creating Objects from a class
    - Instance Variables and Methods

Class & Object:

In Java, a class is a blueprint or a template for creating objects. It defines the structure and behavior that objects created from that class will have. A class can contain fields (variables) and methods (functions) that describe the properties and actions of its objects.

Here's a simple example:

Java code

```
// Defining a class named "Car"

public class Car {

    // Fields (instance variables)

    String make;
```

```java
    String model;
    int year;

    // Constructor (a special method to create objects)
    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }
    // Method to display information about the car
    public void displayInfo() {
        System.out.println("Make: " + make);
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
    }
}
public class Main {
    public static void main(String[] args) {
        // Creating an object of the "Car" class
        Car myCar = new Car("Toyota", "Camry", 2023);
        // Accessing the object's methods and fields
        myCar.displayInfo();
    }
}
```

In this example:

We define a class named "Car" with fields (make, model, and year), a constructor to initialize those fields when creating an object, and a method (displayInfo) to display information about the car. the main method, we create an object called myCar from the "Car" class. This object has its own set of fields and can call the methods defined in the class.

We set the values of the fields for myCar using the constructor and then call the displayInfo method to print the car's information.

So, in summary, a class is like a blueprint (Car) that defines the structure, and an object (myCar) is an instance of that blueprint with its specific data.

Instance Variables: Instance variables, also known as member variables or fields, are attributes associated with an object. These variables represent the object's state or characteristics. Each object created from a class has its own set of instance variables. Instance variables are declared within a class but outside of any method or constructor. They define the properties that an object of that class will have. For example, in a Car class, you might have instance variables like color, model, and year.

```
public class Car {

    String color;  // Instance variable

    String model;  // Instance variable

    int year;      // Instance variable

}
```

Instance Methods: Instance methods, also known as member methods, are functions defined within a class that operate on the instance variables of an object. These methods represent the behaviors or actions that an object can perform. They are called on specific objects and can access and modify the object's state through instance variables. For example, a Car class might have methods like start(), accelerate(), and stop().

```
public class Car {

    // Instance variables

    void start() {

        // Method logic to start the car

    }

void accelerate() {

        // Method logic to accelerate the car

    }

    void stop() {

        // Method logic to stop the car

    }

}
```

*** Constructors in java

In Java, a constructor is a special method within a class that is used to initialize objects. When you create an instance (object) of a class, the constructor is automatically called to set up the initial state of the object. Here are some key points to understand about constructors:

1. **Initialization**: Constructors are primarily used for initializing the instance variables (fields) of an object when it is created. This ensures that the object starts with a meaningful and consistent state.
2. **Same Name as Class**: A constructor has the same name as the class it belongs to. It doesn't have a return type, not even **void**.
3. **Multiple Constructors**: A class can have multiple constructors with different parameter lists. This is called constructor overloading.
4. **Default Constructor**: If you don't define any constructors in your class, Java provides a default constructor with no arguments. However, if you define any constructors, the default one won't be automatically available.

Here's a simple example:

```
public class Person {
String name; int age; // Parameterized constructor
public Person(String n, int a) {
name = n; age = a;
}
 // Default constructor (provided by Java if no constructors are defined)
public Person() {
name = "Unknown"; age = 0;
}
 public void displayInfo() {
System.out.println("Name: " + name);
System.out.println("Age: " + age);
}
public static void main(String[] args) {
 // Creating objects using constructors
Person person1 = new Person("Alice", 25);

Person person2 = new Person(); // Calling a method to display information
person1.displayInfo(); person2.displayInfo();

} }
```

In this example:

- We have a **Person** class with two constructors: a parameterized constructor and a default constructor.
- When we create **person1** using the parameterized constructor, we pass the name and age values.
- When we create **person2**, it uses the default constructor, and we don't pass any values, so it initializes to "Unknown" and 0.

Constructors are fundamental for initializing objects with specific data, making them a crucial part of Java classes.

Code Example:

java code

```
public class Student {

// Instance Variables

String name;

int age;

// Constructor

public Student(String n, int a)

{

name = n;

age = a;

}

// Method

public void displayInfo() {

 System.out.println("Name: " + name);

System.out.println("Age: " + age);

}
}
```

3. Polymorphism

- Polymorphism Concept
- Method Overloading
- Method Overriding

Polymorphism is a fundamental concept in object-oriented programming and is a key feature of the Java programming language. It allows objects of different classes to be treated as objects of a common superclass. In simpler terms, polymorphism enables you to work with objects of different types through a common interface.

There are two main types of polymorphism in Java:

Compile-Time Polymorphism (Static Binding):

Compile-time polymorphism, also known as static binding or method overloading, occurs when multiple methods in the same class have the same name but different parameters (different number or types of parameters). The compiler determines which method to call based on the method's signature at compile time. Here's an example:

java code

```
class Calculator {
int add(int num1, int num2) {
return num1 + num2; }
double add(double num1, double num2) {
 return num1 + num2; }
String add(String str1, String str2) {
 return str1 + str2; } }
public class CompileTimePolymorphism {
public static void main(String[] args) {
Calculator calculator = new Calculator();
int sum1 = calculator.add(5, 7); // Calls the int version of add
double sum2 = calculator.add(3.14, 2.71); // Calls the double version of add
String result = calculator.add("Hello, ", "world!"); // Calls the String version of add
System.out.println("Sum1: " + sum1);
System.out.println("Sum2: " + sum2);
System.out.println("Concatenated String: " + result);
} }
```

Run-Time Polymorphism (Dynamic Binding):

Polymorphism is a fundamental concept in object-oriented programming, allowing objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different types of objects, providing flexibility and extensibility to your code. In Java, polymorphism is typically achieved through method overriding and interfaces. Also known as method overriding.

Occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.

The correct method to be executed is determined at runtime based on the actual type of the object.

This is achieved by using the @Override annotation and creating a method with the same signature in the subclass.

Here's a simple Java example that demonstrates polymorphism:

java code

```
class Animal {
void makeSound() {
System.out.println("Some Animal sound"); } }
class Dog extends Animal {
@Override void makeSound() {
System.out.println("Dog Sound Bark"); } }
class Cat extends Animal {
@Override void makeSound() {
 System.out.println("Cat Sound Meow"); } }
public class PolymorphismExample {
public static void main(String[] args) {
Animal myDog = new Dog();
Animal myCat = new Cat();
myDog.makeSound(); // Calls Dog's makeSound method
myCat.makeSound(); // Calls Cat's makeSound method
}
}
```

In this example:

1. We have a superclass **Animal** with a method **makeSound()**.
2. Two subclasses, **Dog** and **Cat**, extend the **Animal** class and override the **makeSound()** method to provide their own implementations.
3. In the **PolymorphismExample** class, we create objects of type **Animal** but instantiate them as **Dog** and **Cat**.
4. When we call the **makeSound()** method on these objects, it invokes the overridden method of the actual object type (polymorphism). This allows us to call **makeSound()** on an **Animal** reference, but the specific implementation is determined at runtime based on the object's actual type.

4. Inheritance

- Inheritance Concept:
- Superclass and Subclass
- Method Overriding

Code Example:

```
class Vehicle {
void start() {
System.out.println("Vehicle started");
}
}
class Car extends Vehicle {
@Override void start() {
System.out.println("Car started");
}
 }
```

5. Encapsulation

- Encapsulation Concept

- Private Variables and Public Methods
- Getters and Setters

Code Example:

java code

```
public class Person {
private String name;
public String getName() {
 return name; }
public void setName(String n) {
 name = n; }
 }
```

Encapsulation:

************Access Modifiers**

Access modifiers, also known as access specifiers, are keywords in Java that define the visibility or accessibility of classes, variables, methods, and constructors. They determine which parts of your code can access or modify a particular element. Java provides four main access modifiers:

public: The element (class, variable, method, etc.) is accessible from any other class.

private: The element is only accessible within the same class. It cannot be accessed from outside the class.

protected: The element is accessible within the same class, within subclasses, and within the same package.

default (package-private): If no access modifier is specified, the element is accessible within the same class and within the same package (package-level access).

Here's an example demonstrating the use of access modifiers:

```
public class MyClass {
   public int publicVar;       // Accessible from anywhere
   private int privateVar;     // Accessible only within this class
   protected int protectedVar; // Accessible within this class and subclasses
   int defaultVar;             // default Access within this class and package
}
```

Access modifiers help control the level of encapsulation and visibility of class members, contributing to the principles of encapsulation and data hiding in Object-Oriented Programming.

6. **Abstraction**

- Abstraction Concept : if any method is not properly defined within the class, that means it has only the method signature that is not defined then we can call this class as Abstract Class. Any class that extends that abstract class must implement that abstract method. If it does not define it then it must declare as an abstract class.
- Abstract Classes and Methods
- Interface

Code Example:

java code

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