Unit 3: Object-Oriented Programming Concepts (OOP) (9 Hours)

Fundamentals of Classes

Object-Oriented Programming (OOP) is a programming paradigm that is based on the concept of **objects**, which can contain data and methods. In Java, a **class** is the blueprint from which objects are created. This unit focuses on the **fundamentals of classes**, including creating simple classes, creating class instances (objects), adding methods to classes, and calling functions/methods.

---

1. Introduction to Classes

A **class** in Java is a blueprint or template for creating objects. It defines a set of attributes (variables) and methods (functions) that describe the behavior of the objects created from the class. An object is an instance of a class.

Basic Syntax of a Class:

```
public class ClassName {
    // Attributes (fields) or properties of the class
    int attribute1;
    String attribute2;

    // Constructor (optional)
    public ClassName() {
        // Initialize attributes
    }

    // Methods (functions) for the class
    public void method1() {
        System.out.println("Method 1 of the class.");
    }

    public int add(int a, int b) {
        return a + b;
    }
}
```

2. A Simple Class Example

Let’s start by creating a simple class called `Person`. This class will have attributes such as `name` and `age`, and a method `greet()` that prints a greeting message.

Lab 1: A Simple Class Example

```
public class Person {
    // Attributes (fields)
    String name;
    int age;

    // Constructor to initialize the attributes
    /*
```

```
   Features of a Constructor:
    1. Same Name as Class – The constructor name must match the class name.
    2. No Return Type – Constructors do not have a return type, not even void.
    3. Automatic Invocation – It is called automatically when an object is created.
    4. Used for Initialization – It initializes object properties.
    */
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method to print a greeting message
    public void greet() {
        System.out.println("Hello, my name is " + name + " and I am " + age + " years
old.");
    }
}
```

Explanation of the Example:

- **Class Declaration**: The class `Person` is declared with the keyword `public`. It contains two attributes: `name` (String) and `age` (int).
- **Constructor**: The constructor `public Person(String name, int age)` is used to initialize the `name` and `age` attributes when an object is created.
- **Method**: The `greet()` method prints a greeting message using the `name` and `age` attributes.

3. Creating Class Instances (Objects)

An **instance** of a class is an object that is created based on the class template. In Java, objects are created using the `new` keyword.

Lab 1: Creating Class Instances (Objects)

```
public class Main {
    public static void main(String[] args) {
        // Creating an object of the Person class
        Person person1 = new Person("Sharat", 25);

        // Calling the greet() method on person1
        person1.greet();
    }
}
```

Explanation of the Example:

- **Creating an Object**: `Person person1 = new Person("Sharat", 25);` creates an object `person1` of the class `Person` and initializes it with the values `"Sharat"` for the name and `25` for the age.
- **Calling a Method**: `person1.greet();` calls the `greet()` method of the `person1` object, which prints the greeting message.

Sample Output:

```
Hello, my name is Sharat and I am 25 years old.
```

4. Adding Methods to a Class

Methods in Java define the behavior of the objects created from a class. They can accept parameters, perform operations, and return values.

Method Syntax:

```
public returnType methodName(parameters) {
    // Method body
    // Perform operations and return a value if needed
}
```

Lab 2: Adding Methods for Calculations

Let's add a method to calculate the area of a rectangle in the `Rectangle` class.

```
public class Rectangle {
    // Attributes (fields)
    double length;
    double width;

    // Constructor to initialize the rectangle's dimensions
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    // Method to calculate the area of the rectangle
    public double calculateArea() {
        return length * width;
    }
}
```

Explanation of the Example:

- **Attributes**: The `Rectangle` class has two attributes: `length` and `width`.
- **Constructor**: The constructor initializes the `length` and `width` values when a new object of the class is created.
- **Method**: The `calculateArea()` method computes and returns the area of the rectangle (i.e., `length * width`).

Lab 2: Using the Rectangle Class:

```
public class Main {
    public static void main(String[] args) {

        // Creating an object of Rectangle class
        Rectangle rect = new Rectangle(5.0, 4.0);
```

```
        // Calling the method to calculate area
        double area = rect.calculateArea();

        // Printing the area
        System.out.println("The area of the rectangle is: " + area);

        /*
        Use of Scanner to input from user:
        Scanner scanner = new Scanner(System.in);
        double length = scanner.nextDouble();
        double breadth = scanner.nextDouble();

        scanner.close();    ->> close scanner object at the end.
         */
    }
}
```

Explanation of the Example:

- **Creating an Object**: The object `rect` of class `Rectangle` is created with the length `5.0` and width `4.0`.
- **Calling the Method**: `rect.calculateArea();` calculates the area of the rectangle and stores the result in the `area` variable.

Sample Output:

```
The area of the rectangle is: 20.0
```

Question: Lab 3:

Write a Java program to calculate the **Volume** and **Total Surface Area (TSA)** of a cuboid using functions. The program should:

1. Accept the **length**, **breadth**, and **height** of the cuboid as input from the user.
2. Use a function to calculate the **Volume**.
3. Use another function to calculate the **Total Surface Area (TSA)**.
4. Display the results.

Formulae:

1. **Volume of the cuboid (V)**: V = length * breadth * height
2. **Total Surface Area (TSA) of the cuboid**: TSA=2*(length * breadth + breadth * height + height * length)

Sample Output:

```
Enter the length of the cuboid: 5
Enter the breadth of the cuboid: 4
Enter the height of the cuboid: 3

Volume of the cuboid: 60
Total Surface Area (TSA) of the cuboid: 94
```

Explanation:

- **Volume** is calculated as: `5 * 4 * 3 = 60`
- **Total Surface Area** is calculated as: `2 * ( 5 * 4 + 4 * 3 + 3 * 5 ) = 94`

5. Calling Functions/Methods

To **call a method** in Java, you use the object reference for instance methods or the class name for static methods.

Calling Instance Methods:

An instance method belongs to an object. You must create an object to call the instance method.

```
objectReference.methodName();
```

Lab 4: Calling Instance Methods

```
public class Person {
    String name;

    public Person(String name) {
        this.name = name;
    }

    public void greet() {
        System.out.println("Hello, my name is " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of Person class
        Person person1 = new Person("Sharat");

        // Call the greet method
        person1.greet();  // Output: Hello, my name is Sharat
    }
}
```

Calling Static Methods:

A **static method** belongs to the class itself and can be called without creating an object.

```
ClassName.methodName();
```

Lab 5: Calling Static Methods

```
public class Calculator {
    // Static method for addition
    public static int add(int a, int b) {
        return a + b;
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        // Calling the static method without creating an object
        int result = Calculator.add(5, 3);

        // Printing the result
        System.out.println("The sum is: " + result);  // Output: The sum is: 8
    }
}
```

6. Access Modifiers in Java

Java provides **access modifiers** to control the visibility of classes, methods, and variables.

- `public`: Accessible from anywhere.
- `private`: Accessible only within the same class.
- `protected`: Accessible within the same package and by subclasses.
- Default (no modifier): Accessible within the same package.

Lab 6: Access Modifiers in Java

```
public class Person {
    private String name;  // Private field
    public int age;       // Public field

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Public method to access private field
    public String getName() {
        return name;
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person("Sharat", 30);

        // Accessing public field directly
        System.out.println("Age: " + person.age);

        // Accessing private field via a public method
        System.out.println("Name: " + person.getName());
    }
}
```

Sample Output:

```
Age: 30
Name: Sharat
```

---

7. Abstraction

Abstraction is the concept of hiding the implementation details and showing only the essential features of an object. In Java, abstraction is achieved using **abstract classes** and **interfaces**.

- **Abstract Class**: A class that cannot be instantiated and may have abstract methods (methods without implementation).
- **Interface**: A collection of abstract methods. A class that implements an interface must provide implementations for all its methods.

Lab 7: Abstraction

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

    public void sleep() {
        System.out.println("Animal is sleeping.");
    }
}

// Concrete class
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks.");
    }
}

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

---

8. Encapsulation

Encapsulation is the concept of wrapping data (variables) and methods that operate on the data into a single unit known as a class. It restricts direct access to some of the object's components and can prevent unintended interference and misuse of the data.

- **Private Variables**: Used to hide the data.
- **Public Methods**: Used to access and update the data (getters and setters).

Lab 8: Encapsulation

```java
public class Person {
    // Private variables
    private String name;
    private int age;

    // Getter method
    public String getName() {
        return name;
    }

    // Setter method
    public void setName(String name) {
        this.name = name;
    }

    // Getter method
    public int getAge() {
        return age;
    }

    // Setter method
    public void setAge(int age) {
        if (age >= 0) {
            this.age = age;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.setName("Sharat");
        p.setAge(25);

        System.out.println("Name: " + p.getName());  // Output: Name: Sharat
        System.out.println("Age: " + p.getAge());    // Output: Age: 25
    }
}
```

---

9. Using `this` Keyword

The **`this` keyword** is used within an instance method or constructor to refer to the current object. It is commonly used to refer to instance variables when they are shadowed by method parameters.

Lab 9: Using `this` Keyword

```java
public class Person {
    String name;

    public Person(String name) {
        this.name = name;  // 'this' refers to the current object's instance variable
```

```
    }

    public void printName() {
        System.out.println("Name: " + this.name);
    }

    public static void main(String[] args) {
        Person p = new Person("Sharat");
        p.printName();  // Output: Name: Sharat
    }
}
```

---

10. Constructors: Default and Parameterized

- **Default Constructor**: A constructor provided by Java when no constructors are defined in the class. It initializes object members with default values (e.g., `null` for objects, `0` for integers).
- **Parameterized Constructor**: A constructor that allows the initialization of object members with specific values when an object is created.

Lab 10: Constructors: Default and Parameterized

```
public class Car {
    String model;
    int year;

    // Default Constructor
    public Car() {
        this.model = "Unknown";
        this.year = 2020;
    }

    // Parameterized Constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    public void display() {
        System.out.println("Model: " + model + ", Year: " + year);
    }

    public static void main(String[] args) {
        Car car1 = new Car();  // Calls default constructor
        Car car2 = new Car("Tesla", 2022);  // Calls parameterized constructor

        car1.display();  // Output: Model: Unknown, Year: 2020
        car2.display();  // Output: Model: Tesla, Year: 2022
    }
}
```

---

11. Methods: Passing by Value and by Reference

- **Passing by Value**: In Java, primitive data types are passed by value. This means that a copy of the variable is passed to the method.
- **Passing by Reference**: For objects, Java passes the reference to the object (not the actual object), meaning that modifications made to the object in the method affect the original object.

Lab 11: Methods: Passing by Value and by Reference

```
// Passing by value (Primitive type)
public class Example {
    public static void changeValue(int num) {
        num = 50;
    }

    public static void main(String[] args) {
        int number = 10;
        changeValue(number);
        System.out.println("Value after method call: " + number);  // Output: 10
(value not changed)
    }
}

// Passing by reference (Object)
public class Example {
    static class Person {
        String name;
        Person(String name) {
            this.name = name;
        }
    }

    public static void changeName(Person p) {
        p.name = "Sharat";
    }

    public static void main(String[] args) {
        Person person = new Person("Sharat");
        changeName(person);
        System.out.println("Name after method call: " + person.name);  // Output: Name
after method call: Sharat
    }
}
```

---

12. Methods that Return Values

A method in Java can return a value. The return type is specified in the method declaration.

Lab 12: Methods that Return Values

```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int sum = calc.add(5, 3);
        System.out.println("Sum: " + sum);  // Output: Sum: 8
    }
}
```

---

13. Polymorphism and Method Overloading

- **Polymorphism**: The ability to take many forms. In Java, polymorphism allows us to use the same method name with different implementations.
- **Method Overloading**: The ability to define multiple methods with the same name but different parameter types or number of parameters.

Lab 13: Polymorphism and Method Overloading

```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 3));      // Output: 8
        System.out.println(calc.add(5.5, 3.5));  // Output: 9.0
    }
}
```

---

14. Recursion

Recursion occurs when a method calls itself in order to solve a problem. A base case is used to stop the recursion.

Lab 14: Factorial

```java
public class Factorial {
    public static int factorial(int n) {
        if (n == 0) {
            return 1;  // Base case
        } else {
            return n * factorial(n - 1);  // Recursive case
        }
```

```
    }

    public static void main(String[] args) {
        System.out.println("Factorial of 5: " + factorial(5));  // Output: 120
    }
}
```

---

15. Nested and Inner Classes

- **Nested Class**: A class defined within another class.
- **Inner Class**: A nested class that has access to the instance variables and methods of the outer class.

Lab 15: Nested and Inner Classes

```
public class OuterClass {
    private String outerVar = "Outer class variable";

    // Inner class
    class InnerClass {
        public void printOuterVar() {
            System.out.println(outerVar);  // Accesses outer class variable
        }
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();  // Create inner class
object
        inner.printOuterVar();  // Output: Outer class variable
    }
}
```

Conclusion: What Students Have Learned

After completing this unit on **Fundamentals of Classes** and other Object-Oriented Programming (OOP) concepts, students will have gained a thorough understanding of key OOP principles and Java programming techniques. Specifically, students will be able to:

1. **Understand the Fundamentals of Classes:**
    - Define and create **simple classes** in Java.
    - Instantiate **objects** of a class and work with them.
    - Add **methods** to a class and call them effectively, demonstrating the practical use of functions in object-oriented design.
2. **Apply Abstraction and Encapsulation:**
    - Implement **abstraction** to hide unnecessary details and expose only essential features through abstract classes or interfaces.
    - Use **encapsulation** to bundle data (variables) and methods that operate on the data within a class, restricting direct access to certain fields for better data integrity and security.

3. **Utilize the `this` Keyword:**
    - Understand how to use the `this` keyword to refer to the current instance of a class, especially in constructors and methods where variable names may conflict.
4. **Work with Constructors:**
    - Define and use **default constructors** that provide initial values for an object's attributes.
    - Create **parameterized constructors** to initialize objects with specific values at the time of creation.
5. **Understand Method Parameters and Value Passing:**
    - Learn how Java handles **pass-by-value** for primitive types and **pass-by-reference** for objects, enabling students to manipulate data within methods appropriately.
6. **Implement Access Control:**
    - Use **access modifiers** (public, private, protected) to control access to class members, ensuring proper encapsulation and security in the application.
7. **Work with Methods that Return Values:**
    - Develop the ability to create methods that perform calculations or operations and return values to the caller for further processing.
8. **Explore Polymorphism and Method Overloading:**
    - Grasp the concept of **polymorphism**, allowing methods to behave differently based on the object type.
    - Implement **method overloading** to define multiple methods with the same name but different parameter types or counts, enhancing flexibility and reusability of code.
9. **Master Recursion:**
    - Understand **recursion** as a problem-solving technique, where methods call themselves to solve smaller instances of a problem, and apply it effectively in various scenarios like calculating factorials.
10. **Use Nested and Inner Classes:**
    - Implement **nested** and **inner classes** to group related classes together, enabling better organization and encapsulation of functionality within a larger class.