ASSIGNMENT-6

Q1). Method Overloading
Task: Create a MathOperations class with overloaded methods: add(int a, int b) to add two integers. add(double a, double b) to add two doubles. Expected Outcome: Show how the compiler differentiates methods based on parameter types.

ANS -

```java
class MathOperations {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Method to add two doubles
    double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        MathOperations operations = new MathOperations();

        // Adding integers
        System.out.println("Sum of integers: " + operations.add(5, 10));

        // Adding doubles
        System.out.println("Sum of doubles: " + operations.add(5.5, 10.5));
    }
}
```

OUTPUT:

```
Sum of integers: 15
Sum of doubles: 16.0
```

Q2). This Keyword

Task: Create a class Student with fields: name and rollNumber. Write a constructor that uses this to initialize these fields. Add a display method to print the details. Expected Outcome: Use this to refer to instance variables and see its impact in constructors.

ANS -

The `this` keyword in Java is a reference to the current object (the object that is calling the method or constructor). It is used in various scenarios to avoid ambiguity, refer to instance variables, or call other methods/constructors within the same class.

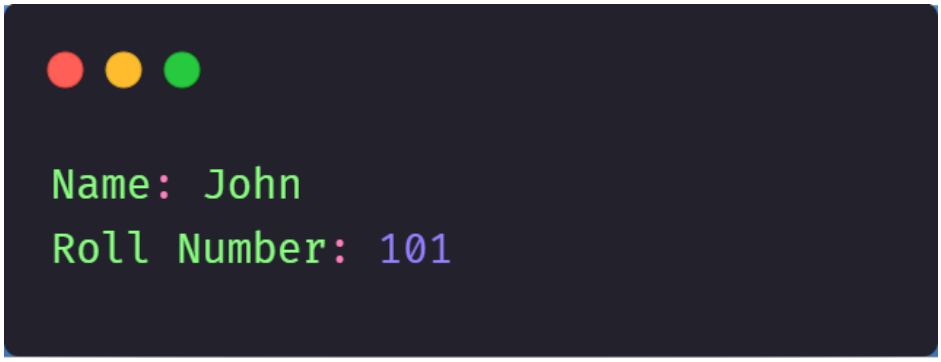
```java
class Student {
    String name;
    int rollNumber;

    // Constructor using this keyword
    Student(String name, int rollNumber) {
        this.name = name;         // Refers to the instance variable 'name'
        this.rollNumber = rollNumber; // Refers to the instance variable 'rollNumber'
    }

    // Method to display details
    void display() {
        System.out.println("Name: " + this.name);         // Refers to the instance variable 'name'
        System.out.println("Roll Number: " + this.rollNumber); // Refers to the instance variable 'rollNumber'
    }

    public static void main(String[] args) {
        Student student = new Student("John", 101); // Creating an object
        student.display(); // Calling the display method
    }
}
```

OUTPUT:



Q3)Super Keyword

Task: Create a parent class Vehicle with a method describe that prints "This is a vehicle." Create a child class Car that overrides describe to print "This is a car." Use super.describe() in the child class to also print the parent method's output. Expected Outcome: Understand how super calls the parent class methods.

ANS - The `super` keyword in Java is used to refer to the immediate parent class. It serves several purposes:

1. Call the Parent Class's Method:
    - You can use `super.methodName()` to call a method defined in the parent class, even if it has been overridden in the child class.
2. Access Parent Class's Members:
    - You can use `super.variableName` to access variables in the parent class if they are shadowed by variables in the child class.
3. Call the Parent Class Constructor:
    - You can use `super()` to explicitly call the parent class's constructor.

Explanation:

1. Parent Class (`Vehicle`):

The `Vehicle` class defines a method `describe()` that prints:
`This is a vehicle.`

2. Child Class (`Car`):
    - The `Car` class extends `Vehicle` and overrides the `describe()` method.
    - Inside the overridden `describe()` method, `super.describe()` is used to call the parent class's version of `describe()`.

After calling `super.describe()`, the child class adds its own implementation:
`This is a car.`

3. Execution Flow:
    - When you call `car.describe()`, the overridden method in the child class is executed.
    - First, `super.describe()` is executed, which calls the `describe()` method from the parent class (`Vehicle`).
    - Then, the child class-specific statement (`System.out.println("This is a car.");`) is executed.

CODE:

```java
class Vehicle {
    void describe() {
        System.out.println("This is a vehicle.");
    }
}

class Car extends Vehicle {
    @Override
    void describe() {
        super.describe(); // Calls the parent class's describe method
        System.out.println("This is a car."); // Child class-specific implementation
    }

    public static void main(String[] args) {
        Car car = new Car();
        car.describe();
    }
}
```

OUTPUT:

```
This is a vehicle.
This is a car.
```

Q4). Constructor

Task: Create a Person class with: A default constructor initializing name as "Unknown." A parameterized constructor to initialize name. Print the name from both constructors. Expected Outcome: Learn constructor overloading and object initialization.

ANS-Constructor overloading in Java means creating multiple constructors within a class, each having a different parameter list. This allows objects of the class to be initialized in different ways based on the type or number of arguments passed during object creation.

CODE:

```java
class Person {
    String name;

    // Default constructor
    Person() {
        this.name = "Unknown";  // Default initialization
    }

    // Parameterized constructor
    Person(String name) {
        this.name = name;       // Assigning value to the instance variable
    }

    public static void main(String[] args) {
        // Object created using default constructor
        Person person1 = new Person();

        // Object created using parameterized constructor
        Person person2 = new Person("Alice");

        // Displaying values
        System.out.println("Name from default constructor: " + person1.name);
        System.out.println("Name from parameterized constructor: " + person2.name);
    }
}
```

OUTPUT:

```
Name from default constructor: Unknown
Name from parameterized constructor: Alice
```

Q5). **Static Keyword**

Task: Create a class Counter with: A static variable count. A static method increment() to increase the count. A static method display() to print the count. Call these methods without creating an object.
Expected Outcome: Understand the use of static variables and methods.

ANS- The `static` keyword in Java is used to define:

1. **Static Variables**: Shared among all instances of a class (class-level).
2. **Static Methods**: Belong to the class rather than to any specific object and can be called without creating an instance.
3. **Static Blocks**: Run only once when the class is loaded into memory.

When We use `static`, you're associating the member (variable or method) with the **class itself**, not with specific objects of the class.

What I Have done in the Code?

1. **Static Variable (`count`):**
   - `static int count` is shared among all instances of the class.
   - It is initialized only **once** when the class is loaded, and all instances refer to the same `count`.
2. **Static Methods:**
   - `increment()` and `display()` are `static`, so they belong to the class, not to any object.
   - You can call them directly using the class name (`Counter.increment()`), without creating an instance of the `Counter` class.
3. **Execution Steps:**
   - **Step 1:** `Counter.increment()` is called, so `count` increases from 0 to 1.
   - **Step 2:** `Counter.increment()` is called again, so `count` increases from 1 to 2.

- **Step 3:** `Counter.display()` is called, which prints the current value of `count`.

CODE:

```java
class Counter {
    static int count = 0; // Static variable shared by all instances

    // Static method to increment the count
    static void increment() {
        count++; // Increases the static variable
    }

    // Static method to display the count
    static void display() {
        System.out.println("Count: " + count);
    }

    public static void main(String[] args) {
        // Calling static methods without creating objects
        Counter.increment(); // Increment count
        Counter.increment(); // Increment count again
        Counter.display();   // Display the count
    }
}
```

OUTPUT:

```
Count: 2
```

Q6. Encapsulation

Task: Create a BankAccount class with private fields: accountNumber and balance.

ANS - Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It is the practice of bundling **data (fields)** and **methods (functions)** that operate on the data into a single unit (class) while restricting direct access to some of the object's components. This is achieved using **access modifiers** (like `private`, `public`, etc.).

Encapsulation helps achieve **data hiding**, ensuring that critical parts of an object's internal state are protected from unintended interference and misuse.

In Java, **encapsulation** is implemented by:

1. Declaring class fields (variables) as `private`.
2. Providing `public` getter and setter methods to access and update the private fields safely.

CODE:

```java
class BankAccount {
    // Private fields to restrict direct access
    private String accountNumber;
    private double balance;

    // Constructor to initialize account details
    public BankAccount(String accountNumber, double balance) {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    // Getter for accountNumber (read-only access)
    public String getAccountNumber() {
        return accountNumber;
    }

    // Getter for balance (read access)
    public double getBalance() {
        return balance;
    }

    // Setter for balance (write access, with control)
    public void setBalance(double balance) {
        if (balance ⩾ 0) { // Validation: ensure non-negative balance
            this.balance = balance;
        } else {
            System.out.println("Balance cannot be negative.");
        }
    }

    public static void main(String[] args) {
        // Create a BankAccount object
        BankAccount account = new BankAccount("123456", 1000.0);

        // Access account details using getters
        System.out.println("Account Number: " + account.getAccountNumber());
        System.out.println("Balance: " + account.getBalance());

        // Update balance using the setter
        account.setBalance(1500.0);
        System.out.println("Updated Balance: " + account.getBalance());

        // Attempt to set a negative balance
        account.setBalance(-500.0); // Will trigger validation
    }
}
```

OUTPUT:

```
Account Number: 123456
Balance: 1000.0
Updated Balance: 1500.0
Balance cannot be negative.
```

Execution Steps:

1. Private Fields:
    - `accountNumber` and `balance` are declared `private` to prevent direct access from outside the class. This ensures they can only be accessed or modified via the methods provided by the class.
2. Constructor:
    - Initializes the fields (`accountNumber` and `balance`) when the object is created.
3. Getter Methods (`getAccountNumber()` and `getBalance()`):
    - Allow controlled, read-only access to `accountNumber` and `balance` from outside the class.
4. Setter Method (`setBalance()`):
    - Provides controlled write access to `balance`.
    - Includes validation to prevent setting a negative balance.
5. Main Method:
    - Demonstrates:
        - Reading `accountNumber` and `balance` using getter methods.
        - Updating `balance` using the setter method with proper validation.