Object-Oriented Programming (OOP) Concepts in Java

OOP is a programming paradigm that uses **objects** and **classes** to structure programs into reusable code blocks. Java is inherently designed to support OOP principles, making it easy to create modular, scalable, and maintainable software.

Key OOP Concepts in Java

1. Class

- **Definition**: A blueprint or template for creating objects.
- **Purpose**: Encapsulates data (fields) and behaviors (methods).

2. Object

- **Definition**: An instance of a class that represents a real-world entity.
- **Purpose**: To interact with the class's properties and methods.

Class and Object

Program: Employee Management System

```
class Employee {
   private int employeeId;
   private String employeeName;
public Employee(int id, String name) {
      this.employeeId = id;
      this.employeeName = name;
   }
public void displayEmployeeDetails() {
      System.out.println("Employee ID: " + employeeId);
      System.out.println("Employee Name: " + employeeName);
   }
}
public class Main {
   public static void main(String[] args) {
      Employee emp1 = new Employee(101, "Alice");
      Employee emp2 = new Employee(102, "Bob");
emp1.displayEmployeeDetails();
      emp2.displayEmployeeDetails();
   }
}
```

Explanation:

1. **Employee Class**: Stores employee details (employeeId and employeeName).
2. **Constructor**: Initializes employee details while creating objects.
3. **displayEmployeeDetails()**: Prints employee information.
4. **Main Class**: Creates two Employee objects and displays their details.

Output:

```
Employee ID: 101
Employee Name: Alice
Employee ID: 102
Employee Name: Bob
```

Encapsulation

- **Definition**: Wrapping data (variables) and methods (functions) together into a single unit, often by making fields private and providing public getter/setter methods.
- **Purpose**: Provides security, hides implementation, and controls access.

Encapsulation

Program: Bank Account Management

```
java
Copy code
class BankAccount {
   private String accountHolderName;
   private double balance;
public BankAccount(String accountHolderName, double initialDeposit) {
      this.accountHolderName = accountHolderName;
      this.balance = initialDeposit;
   }
public void deposit(double amount) {
      balance += amount;
   }
public void withdraw(double amount) {
      if (amount <= balance) balance -= amount;
      else System.out.println("Insufficient funds");
   }
public void displayBalance() {
      System.out.println(accountHolderName + "'s Balance: $" + balance);
   }
}
public class Main {
   public static void main(String[] args) {
      BankAccount account = new BankAccount("John Doe", 5000);
      account.deposit(1500);
      account.withdraw(2000);
      account.displayBalance();
   }
```

```
}
```
Explanation:

1. **Encapsulation**: Fields like balance are private to prevent unauthorized access.
2. **Methods**: deposit(), withdraw(), and displayBalance() allow controlled interactions.
3. **Main Class**: Creates a BankAccount object, updates balance, and displays it.

Output;

John Doe's Balance: $4500.0

Explanation:

- Initial deposit: $5000
- Deposit: $1500 → Total = $6500
- Withdrawal: $2000 → Final balance = $4500

Inheritance

- **Definition**: Allows a class (child) to inherit fields and methods from another class (parent).
- **Purpose**: Promotes code reuse and establishes a relationship between classes.

Program: Role-Based Access Control

```
class User {
   protected String username;
public User(String username) {
      this.username = username;
   }
public void login() {
      System.out.println(username + " logged in.");
   }
}
class Admin extends User {
   public Admin(String username) {
      super(username);
   }
public void manageUsers() {
      System.out.println(username + " is managing users.");
   }
}
class Member extends User {
   public Member(String username) {
      super(username);
   }
public void viewContent() {
      System.out.println(username + " is viewing content.");
   }
}
public class Main {
   public static void main(String[] args) {
      Admin admin = new Admin("Admin01");
      Member member = new Member("Member01");
admin.login();
      admin.manageUsers();
      member.login();
      member.viewContent();
   }
}
```

Explanation:

1. **Inheritance**: Admin and Member inherit common login behavior from the User class.
2. **Specialization**: Admin can manage users; Member can view content.
3. **Main Class**: Demonstrates role-based functionality using objects.

Output:

```
Admin01 logged in.
Admin01 is managing users.
Member01 logged in.
Member01 is viewing content.
```

Polymorphism

- **Definition**: The ability of a single interface to represent different underlying forms (behaviors).
    - **Compile-time (Method Overloading)**: Multiple methods with the same name but different parameters.
    - **Runtime (Method Overriding)**: A subclass provides a specific implementation of a method already defined in its superclass.
- **Purpose**: Promotes flexibility and extensibility.

Program: Payment Processing System

```
abstract class Payment {
   public abstract void processPayment(double amount);
}
class CreditCardPayment extends Payment {
   @Override
   public void processPayment(double amount) {
      System.out.println("Processing credit card payment of $" + amount);
   }
}
class PayPalPayment extends Payment {
   @Override
   public void processPayment(double amount) {
      System.out.println("Processing PayPal payment of $" + amount);
   }
}
public class Main {
   public static void main(String[] args) {
      Payment payment1 = new CreditCardPayment();
```

```
    Payment payment2 = new PayPalPayment();
payment1.processPayment(250.75);
    payment2.processPayment(100.50);
  }
}
```

Explanation:

1. **Polymorphism**: processPayment() has different implementations in CreditCardPayment and PayPalPayment.
2. **Abstraction**: Payment is abstract, providing a common interface for payments.
3. **Main Class**: Processes payments using specific payment methods.

Output:

```
Processing credit card payment of $250.75
Processing PayPal payment of $100.50
```

5. Abstraction

- **Definition**: Hiding implementation details and exposing only essential functionalities using abstract classes or interfaces.
- **Purpose**: Provides a clear separation between the interface and the implementation.

Program: Vehicle Management System

```
abstract class Vehicle {
  private String model;
public Vehicle(String model) {
    this.model = model;
  }
public String getModel() {
    return model;
  }
public abstract void startEngine();
}
class Car extends Vehicle {
  public Car(String model) {
    super(model);
  }
@Override
  public void startEngine() {
    System.out.println("Starting car engine for model: " + getModel());
  }
}
class Bike extends Vehicle {
  public Bike(String model) {
    super(model);
  }
@Override
  public void startEngine() {
    System.out.println("Starting bike engine for model: " + getModel());
  }
}
public class Main {
  public static void main(String[] args) {
    Vehicle car = new Car("Tesla Model S");
    Vehicle bike = new Bike("Yamaha R15");
car.startEngine();
    bike.startEngine();
  }
}
```

Explanation:

1. **Abstraction**: Vehicle hides implementation details of startEngine().
2. **Specialization**: Car and Bike provide specific implementations for startEngine().
3. **Main Class**: Demonstrates engine functionality for different vehicles.

Abstraction

Output:

```
Starting car engine for model: Tesla Model S
Starting bike engine for model: Yamaha R15
```

Advantages of OOP in Java

1. **Reusability**: Code can be reused through inheritance and abstraction.
2. **Scalability**: Easy to extend and maintain.
3. **Security**: Encapsulation protects sensitive data.
4. **Flexibility**: Polymorphism allows dynamic method invocation.