**1. Encapsulation - Banking System**

**Real-Time Scenario**: Imagine you have a **Bank Account**. You don't want anyone to directly access or change the account balance. Instead, you provide methods to deposit, withdraw, and view the balance, while keeping the balance hidden from direct access.

**Example:**

class BankAccount {

// Private variable (balance is hidden)

private double balance;

// Constructor to initialize balance

public BankAccount(double initialBalance) {

if (initialBalance >= 0) {

balance = initialBalance;

} else {

balance = 0;

System.out.println("Initial balance cannot be negative.");

}

}

// Getter method to view the balance (public access)

public double getBalance() {

return balance;

}

// Method to deposit money into the account (public method)

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

} else {

System.out.println("Deposit amount must be positive.");

}

}

// Method to withdraw money from the account (public method)

public void withdraw(double amount) {

if (amount > 0 && amount <= balance) {

balance -= amount;

} else {

System.out.println("Invalid withdrawal amount.");

}

}

}

public class Main {

public static void main(String[] args) {

// Creating a BankAccount object

BankAccount account = new BankAccount(1000);

// Using methods to interact with the balance

System.out.println("Initial Balance: " + account.getBalance());

account.deposit(500);

System.out.println("Balance after deposit: " + account.getBalance());

account.withdraw(300);

System.out.println("Balance after withdrawal: " + account.getBalance());

}

}

**Explanation:**

* **Encapsulation**: The balance is a private variable, and access is only allowed via getter and setter methods (getBalance, deposit, and withdraw).
* This ensures the balance is protected from direct modification and can only be modified in a controlled manner.

**2. Aggregation - School and Students**

**Real-Time Scenario**: A **School** can have multiple **Students**. However, students can exist independently from the school, and even if the school closes, students still exist.

**Example:**

class Student {

String name;

int age;

public Student(String name, int age) {

this.name = name;

this.age = age;

}

}

class School {

private Student[] students;

public School(Student[] students) {

this.students = students;

}

public void showStudents() {

for (Student student : students) {

System.out.println("Student Name: " + student.name + ", Age: " + student.age);

}

}

}

public class Main {

public static void main(String[] args) {

// Creating Student objects

Student student1 = new Student("Alice", 15);

Student student2 = new Student("Bob", 16);

// Creating School object and passing students

Student[] students = {student1, student2};

School school = new School(students);

// Displaying students in the school

school.showStudents();

}

}

**Explanation:**

* **Aggregation**: The **School** has multiple **Students**, but **Students** can exist without the school. If the school closes, the students still exist independently.

**3. Composition - Car and Engine**

**Real-Time Scenario**: A **Car** is made up of various **Components**, and one of the essential components is the **Engine**. If the car is destroyed, the engine will no longer exist, as it's tightly coupled with the car. The **Car** cannot function without the **Engine**, demonstrating **Composition**.

**Example (Car and Engine):**

class Engine {

String engineType;

public Engine(String engineType) {

this.engineType = engineType;

}

public void start() {

System.out.println("The " + engineType + " engine is starting.");

}

}

class Car {

private Engine engine; // Composition: Car has an engine

public Car(String engineType) {

this.engine = new Engine(engineType); // Engine is created as part of Car

}

public void startCar() {

engine.start(); // The car uses its engine to start

}

}

public class Main {

public static void main(String[] args) {

// Creating a Car object, which has an Engine

Car car = new Car("V8");

// Starting the car

car.startCar();

}

}

**Explanation:**

* **Composition**: The **Car** object has an **Engine** object, and the **Engine** is an essential part of the **Car**. The **Car** cannot function without the **Engine**. When the **Car** object is destroyed, the **Engine** object is also destroyed.

**4. Inheritance - Employee and Manager**

**Real-Time Scenario**: In an **Organization**, a **Manager** is an **Employee** with additional responsibilities. Both **Employee** and **Manager** share common attributes like name, ID, and salary, but the **Manager** also has specific actions like managing teams. This is a perfect example of **Inheritance**.

**Example (Employee and Manager):**

class Employee {

String name;

int id;

double salary;

public Employee(String name, int id, double salary) {

this.name = name;

this.id = id;

this.salary = salary;

}

public void work() {

System.out.println(name + " is working.");

}

public void showDetails() {

System.out.println("Employee ID: " + id + ", Name: " + name + ", Salary: $" + salary);

}

}

class Manager extends Employee {

String department;

public Manager(String name, int id, double salary, String department) {

super(name, id, salary); // Call to Employee constructor

this.department = department;

}

public void manageTeam() {

System.out.println(name + " is managing the " + department + " team.");

}

@Override

public void work() {

super.work(); // Calling Employee work method

System.out.println(name + " is also managing the team.");

}

}

public class Main {

public static void main(String[] args) {

// Creating an Employee object

Employee employee = new Employee("John", 101, 50000);

employee.showDetails();

employee.work();

System.out.println();

// Creating a Manager object (which is an Employee with more responsibilities)

Manager manager = new Manager("Alice", 102, 70000, "Sales");

manager.showDetails();

manager.work(); // Calling overridden work method

manager.manageTeam(); // Manager-specific method

}

}

**Explanation:**

* **Inheritance**: The **Manager** class **inherits** from the **Employee** class, meaning it has all the properties (name, id, salary) and methods (like work()) of an **Employee**. Additionally, **Manager** adds specific behavior such as managing a team (manageTeam()), demonstrating that a **Manager** "is an Employee" but with added responsibilities.

**Comparison of Real-Time Scenarios:**

| **Concept** | **Scenario** | **Explanation** |
| --- | --- | --- |
| **Encapsulation** | **Bank Account** | Data (balance) is hidden and can only be accessed or modified using methods. |
| **Aggregation** | **School and Students** | A school has students, but students exist independently of the school. |
| **Composition** | **Car and Engine** | A **Car** "Has-A" **Engine**. The **Engine** cannot exist without the **Car**. |
| **Inheritance** | **Employee and Manager** | A **Manager** "Is-A" **Employee**. A **Manager** inherits all features of an **Employee** and adds new responsibilities (managing a team). |