Ass1 = Q1. You are given an AudioPlayer class that only plays MP3 files. Extend its functionality to support playing both MP4 and VLC files by using the Adapter Pattern.

➔1.interface MediaPlayer =

```
public interface MediaPlayer {
 void play(String audioType, String fileName);
}
```

2.Interface AdvanceMediaPlayer =

```
public interface AdvancedMediaPlayer {
        void playMp4(String fileName);
        void playVlc(String fileName);
       }
```

3.VlcPlayer.java =

```
public class VlcPlayer implements AdvancedMediaPlayer {
        @Override
        public void playMp4(String fileName) {
        }
        @Override
        public void playVlc(String fileName) {
           System.out.println("Playing vlc file. Name: " + fileName);
        } }
```

4.Mp4Player.java =

```
public class Mp4Player implements AdvancedMediaPlayer {
 @Override
 public void playMp4(String fileName) {
    System.out.println("Playing mp4 file. Name: " + fileName);
 }
 @Override
 public void playVlc(String fileName) {
 } }
```

5.MediaAdapter.java =

```
public class MediaAdapter implements MediaPlayer {
 AdvancedMediaPlayer advancedMusicPlayer;
 public MediaAdapter(String audioType) {
    if(audioType.equalsIgnoreCase("mp4")) {
       advancedMusicPlayer = new Mp4Player();
    } else if(audioType.equalsIgnoreCase("vlc")) {
       advancedMusicPlayer = new VlcPlayer();
    } }
 @Override
 public void play(String audioType, String fileName) {
    if(audioType.equalsIgnoreCase("mp4")) {
       advancedMusicPlayer.playMp4(fileName);
    } else if(audioType.equalsIgnoreCase("vlc")) {
       advancedMusicPlayer.playVlc(fileName);
    } } }
```

6.AudioPlayer.java =

```
import java.util.Scanner;
public class AudioPlayer implements MediaPlayer {
```

```
MediaAdapter mediaAdapter;
@Override
public void play(String audioType, String fileName) {
   if(audioType.equalsIgnoreCase("mp3")) {
      System.out.println("Playing mp3 file. Name: " + fileName);
   }
   else if(audioType.equalsIgnoreCase("mp4") || audioType.equalsIgnoreCase("vlc")) {
      mediaAdapter = new MediaAdapter(audioType);
      mediaAdapter.play(audioType, fileName);
   } else {
      System.out.println("Invalid media. " + audioType + " format not supported");
   } }
public static void main(String[] args) {
   AudioPlayer audioPlayer = new AudioPlayer();
   Scanner scanner = new Scanner(System.in);
   System.out.println("Enter audio type (mp3, mp4, vlc): ");
   String audioType = scanner.nextLine();
   System.out.println("Enter file name: ");
   String fileName = scanner.nextLine();
   audioPlayer.play(audioType, fileName);
} }
```

Q2. Implement a simple system where you have Employee objects. An employee can either be an individual worker or a manager who supervises a team of employees. Use the Composite Pattern to treat both individual employees and managers uniformly in terms of their reporting structure.

➔1.interface Employee =

```
public interface Employee {
 void showEmployeeDetails();
}
```

2.Developer.java =

```
public class Developer implements Employee {
 private String name;
 private String id;
 public Developer(String name, String id) {
   this.name = name;
   this.id = id;
 }
 @Override
 public void showEmployeeDetails() {
   System.out.println("Developer [Name: " + name + ", ID: " + id + "]");
 } }
```

3.Manager.java =

```
import java.util.ArrayList;
import java.util.List;
public class Manager implements Employee {
 private String name;
 private String id;
 private List<Employee> subordinates;
```

```
 public Manager(String name, String id) {
    this.name = name;
    this.id = id;
    this.subordinates = new ArrayList<>();
 }
 public void addEmployee(Employee employee) {
    subordinates.add(employee);
 }
 @Override
 public void showEmployeeDetails() {
    System.out.println("Manager [Name: " + name + ", ID: " + id + "]");
    for (Employee employee : subordinates) {
       employee.showEmployeeDetails();
    } } }
```

4.compositepattern.java =

```
import java.util.Scanner;
public class CompositePatternDemo {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      Manager generalManager = new Manager("General Manager", "GM001");
      System.out.print("Enter the number of developers: ");
      int numDevelopers = scanner.nextInt();
      scanner.nextLine();
      for (int i = 0; i < numDevelopers; i++) {
         System.out.print("Enter Developer Name: ");
         String devName = scanner.nextLine();
         System.out.print("Enter Developer ID: ");
         String devId = scanner.nextLine();
         Developer developer = new Developer(devName, devId);
         generalManager.addEmployee(developer);
      }
      System.out.print("Enter the number of managers: ");
      int numManagers = scanner.nextInt();
      scanner.nextLine(); // consume newline
      for (int i = 0; i < numManagers; i++) {
         System.out.print("Enter Manager Name: ");
         String mgrName = scanner.nextLine();
         System.out.print("Enter Manager ID: ");
         String mgrId = scanner.nextLine();
         Manager manager = new Manager(mgrName, mgrId);
         System.out.print("Enter the number of developers under this manager: ");
         int numSubordinates = scanner.nextInt();
         scanner.nextLine(); // consume newline
         for (int j = 0; j < numSubordinates; j++) {
            System.out.print("Enter Subordinate Developer Name: ");
            String subName = scanner.nextLine();
            System.out.print("Enter Subordinate Developer ID: ");
            String subId = scanner.nextLine();
```

```
            Developer subDeveloper = new Developer(subName, subId);
            manager.addEmployee(subDeveloper);
          }
          generalManager.addEmployee(manager);
      }
      System.out.println("\nOrganization Structure:");
      generalManager.showEmployeeDetails();
   } }
```

Q3. Develop a Java application where you have different types of shapes (e.g., Circle, Rectangle) and draw implementations (e.g., DrawingAPI1, DrawingAPI2). Use the Bridge pattern to separate shape logic from drawing logic.

➔1.shape.java =

```
public abstract class Shape {
   protected DrawingAPI drawingAPI;
   protected Shape(DrawingAPI drawingAPI) {
      this.drawingAPI = drawingAPI;
   }
   public abstract void draw();
   public abstract void resizeByPercentage(double pct);
}
```

2.DrawingAPI1.java =

```
public class DrawingAPI1 implements DrawingAPI {
 @Override
 public void drawCircle(double x, double y, double radius) {
    System.out.println("API1.circle at (" + x + "," + y + ") with radius " + radius);
 }
 @Override
 public void drawRectangle(double x, double y, double width, double height) {
    System.out.println("API1.rectangle at (" + x + "," + y + ") with width " + width + " and height " +
height);
 } }
```

3.DrawingAPI2.java =

```
public class DrawingAPI2 implements DrawingAPI {
          @Override
          public void drawCircle(double x, double y, double radius) {
             System.out.println("API2.circle at (" + x + "," + y + ") with radius " + radius);
          }
          @Override
          public void drawRectangle(double x, double y, double width, double height) {
             System.out.println("API2.rectangle at (" + x + "," + y + ") with width " + width + " and
height " + height);
          } }
```

4.Rectangleshape.java =

```
public class RectangleShape extends Shape {
 private double x, y, width, height;
 public RectangleShape(double x, double y, double width, double height, DrawingAPI drawingAPI) {
    super(drawingAPI);
```

```
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
 }
 @Override
 public void draw() {
    drawingAPI.drawRectangle(x, y, width, height);
 }
 @Override
 public void resizeByPercentage(double pct) {
    width *= (1.0 + pct / 100.0);
    height *= (1.0 + pct / 100.0);
 } }
```

5.interface DrawingAPI =

```
public interface DrawingAPI {
   void drawCircle(double x, double y, double radius);
   void drawRectangle(double x, double y, double width, double height);
}
```

6.circle.java =

```
public class CircleShape extends Shape {
 private double x, y, radius;
 public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI) {
    super(drawingAPI);
    this.x = x;
    this.y = y;
    this.radius = radius;
 }
 @Override
 public void draw() {
    drawingAPI.drawCircle(x, y, radius);
 }
 @Override
 public void resizeByPercentage(double pct) {
    radius *= (1.0 + pct / 100.0);
 } }
```

7.bridgepatterndemo =

```
import java.util.Scanner;
public class BridgePatternDemo {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Select Drawing API: ");
      System.out.println("1. DrawingAPI1");
      System.out.println("2. DrawingAPI2");
      int apiChoice = scanner.nextInt();
      DrawingAPI drawingAPI = (apiChoice == 1) ? new DrawingAPI1() : new DrawingAPI2();
      while (true) {
         System.out.println("Select Shape to draw: ");
```

```
        System.out.println("1. Circle");
        System.out.println("2. Rectangle");
        System.out.println("3. Exit");
        int shapeChoice = scanner.nextInt();
        if (shapeChoice == 3) {
            break;
        }
        Shape shape = null;
        switch (shapeChoice) {
            case 1:
                System.out.print("Enter x coordinate: ");
                double xCircle = scanner.nextDouble();
                System.out.print("Enter y coordinate: ");
                double yCircle = scanner.nextDouble();
                System.out.print("Enter radius: ");
                double radius = scanner.nextDouble();
                shape = new CircleShape(xCircle, yCircle, radius, drawingAPI);
                break;
            case 2:
                System.out.print("Enter x coordinate: ");
                double xRectangle = scanner.nextDouble();
                System.out.print("Enter y coordinate: ");
                double yRectangle = scanner.nextDouble();
                System.out.print("Enter width: ");
                double width = scanner.nextDouble();
                System.out.print("Enter height: ");
                double height = scanner.nextDouble();
                shape = new RectangleShape(xRectangle, yRectangle, width, height, drawingAPI);
                break;
            default:
                System.out.println("Invalid choice! Please try again.");
        }
        if (shape != null) {
            shape.draw();
            System.out.print("Resize by percentage: ");
            double pct = scanner.nextDouble();
            shape.resizeByPercentage(pct);
            shape.draw();
        } }
        scanner.close();
    } }
```

Q4. Create a basic Pizza class in Java and then implement decorators to add different toppings (like cheese, olives, and mushrooms) to the pizza. Each topping should be an additional class that extends a base decorator.

➔1.interface pizza =

```
public interface Pizza {
    String getDescription();
```

```
   double getCost();
}
```
2.olives.java =
```
public class Olives extends PizzaDecorator {
   public Olives(Pizza tempPizza) {
      super(tempPizza);
   }
   @Override
   public String getDescription() {
      return tempPizza.getDescription() + ", Olives";
   }
   @Override
   public double getCost() {
      return tempPizza.getCost() + 15.0;
   } }
```
3.mushrooms.java =
```
public class Mushrooms extends PizzaDecorator {
   public Mushrooms(Pizza tempPizza) {
      super(tempPizza);
   }
   @Override
   public String getDescription() {
      return tempPizza.getDescription() + ", Mushrooms";
   }
   @Override
   public double getCost() {
      return tempPizza.getCost() + 25.0;
   } }
```
4.cheese.java =
```
public class Cheese extends PizzaDecorator {
   public Cheese(Pizza tempPizza) {
      super(tempPizza);
   }
   @Override
   public String getDescription() {
      return tempPizza.getDescription() + ", Cheese";
   }
   @Override
   public double getCost() {
      return tempPizza.getCost() + 20.0;
   } }
```
5.PizzaDecorator.java =
```
public abstract class PizzaDecorator implements Pizza {
   protected Pizza tempPizza;
   public PizzaDecorator(Pizza tempPizza) {
      this.tempPizza = tempPizza;
   }
```

```
    @Override
    public String getDescription() {
        return tempPizza.getDescription();
    }
    @Override
    public double getCost() {
        return tempPizza.getCost();
    } }
```

6.PlanPizza.java =

```
public class PlainPizza implements Pizza {
    @Override
    public String getDescription() {
        return "Plain Pizza";
    }
    @Override
    public double getCost() {
        return 50.0;
    } }
```

7.pizzaorder.java =

```
import java.util.Scanner;
public class PizzaOrder {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Pizza myPizza = new PlainPizza();
        System.out.println("Plain Pizza selected. Base cost: " + myPizza.getCost());
        while (true) {
            System.out.println("Choose a topping to add:");
            System.out.println("1. Cheese");
            System.out.println("2. Olives");
            System.out.println("3. Mushrooms");
            System.out.println("4. Finish");
            System.out.print("Your choice: ");
            int choice = scanner.nextInt();
            switch (choice) {
                case 1:
                    myPizza = new Cheese(myPizza);
                    break;
                case 2:
                    myPizza = new Olives(myPizza);
                    break;
                case 3:
                    myPizza = new Mushrooms(myPizza);
                    break;
                case 4:
                    System.out.println("Order complete!");
                    System.out.println("Description: " + myPizza.getDescription());
                    System.out.println("Total cost: " + myPizza.getCost());
                    scanner.close();
```

```
            return;
          default:
            System.out.println("Invalid choice! Please select again.");
        } } } }
```

Ass2 =Q1.Create a Java class DatabaseConnection using the Singleton design pattern. The class should have a method getConnection() that returns a single instance of the connection. Ensure that only one instance of DatabaseConnection is created even if multiple threads try to access it.

➔1.DatabaseConnection.java =

```
public class DatabaseConnection {
  private static DatabaseConnection instance;
  private static Object mutex = new Object();
  private DatabaseConnection() {
  }
  public static DatabaseConnection getInstance() {
    if (instance == null) {
      synchronized (mutex) {
        if (instance == null) {
          instance = new DatabaseConnection();
        } } }
    return instance;
  }
  public void connect() {
    System.out.println("Connecting to the database...");
  }
  public void disconnect() {
    System.out.println("Disconnecting from the database...");
  } }
```

2.singletondemo.java =

```
import java.util.Scanner;
public class SingletonDemo {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    DatabaseConnection connection = null;
    while (true) {
      System.out.println("Choose an action:");
      System.out.println("1. Get Database Connection");
      System.out.println("2. Connect to Database");
      System.out.println("3. Disconnect from Database");
      System.out.println("4. Exit");
      System.out.print("Your choice: ");
      int choice = scanner.nextInt();
      switch (choice) {
        case 1:
          connection = DatabaseConnection.getInstance();
          System.out.println("Database connection instance obtained.");
          break;
        case 2:
```

```
            if (connection != null) {
                connection.connect();
            } else {
                System.out.println("No connection instance found. Obtain it first.");
            }
            break;
        case 3:
            if (connection != null) {
                connection.disconnect();
            } else {
                System.out.println("No connection instance found. Obtain it first.");
            }
            break;
        case 4:
            System.out.println("Exiting...");
            scanner.close();
            return;
        default:
            System.out.println("Invalid choice! Please select again.");
    } } } }
```

Q2.Implement a VehicleFactory class in Java using the Factory Method design pattern. The factory should be able to create objects of Car and Bike classes based on the input provided.

➔1.interface vehicle =

```
public interface Vehicle {
    void drive();
}
```

2.bike.java =

```
public class Bike implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Bike is driving");
    }
}
```

3.car.java =

```
public class Car implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Car is driving");
    }
}
```

4.bikefactory.java =

```
class BikeFactory extends VehicleFactory {
    @Override
    public Vehicle createVehicle() {
        return new Bike();
    }
}
```

5.carfactory.java =

```
class CarFactory extends VehicleFactory {
   @Override
   public Vehicle createVehicle() {
      return new Car();
   }
}
```

6.vehiclefactory.java =

```
public abstract class VehicleFactory {
   public abstract Vehicle createVehicle();
   public static VehicleFactory getFactory(String type) {
      if (type.equalsIgnoreCase("Car")) {
         return new CarFactory();
      } else if (type.equalsIgnoreCase("Bike")) {
         return new BikeFactory();
      }
      return null;
   } }
```

7.factorymethoddemo.java =

```
import java.util.Scanner;
public class FactoryMethodDemo {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      while (true) {
         System.out.println("Enter the type of vehicle (Car/Bike) or 'exit' to quit: ");
         String input = scanner.nextLine();
         if (input.equalsIgnoreCase("exit")) {
            break;
         }
         VehicleFactory factory = VehicleFactory.getFactory(input);
         if (factory != null) {
            Vehicle vehicle = factory.createVehicle();
            vehicle.drive();
         } else {
            System.out.println("Invalid vehicle type. Please try again.");
         } }
      scanner.close();
   } }
```

Q3.Implement a Shape interface with a method clone(). Create concrete classes Circle and Rectangle that implement this interface. Use the Prototype pattern to create a new instance of Circle or Rectangle by cloning the existing objects.

➔interface shape =

```
public interface Shape extends Cloneable {
   void draw();
   Shape clone();
}
```

2.rectangle.java =

```
public class Rectangle implements Shape {
        private int width;
        private int height;
        public Rectangle(int width, int height) {
           this.width = width;
           this.height = height;
        }
        @Override
        public void draw() {
           System.out.println("Drawing a Rectangle with width: " + width + " and height: " + height);
        }
        @Override
        public Shape clone() {
           try {
              return (Shape) super.clone();
           } catch (CloneNotSupportedException e) {
              e.printStackTrace();
              return null;
       } } }
```

3.circle.java =

```
public class Circle implements Shape {
 private int radius;
 public Circle(int radius) {
    this.radius = radius;
 }
 @Override
 public void draw() {
    System.out.println("Drawing a Circle with radius: " + radius);
 }
 @Override
 public Shape clone() {
    try {
       return (Shape) super.clone();
    } catch (CloneNotSupportedException e) {
       e.printStackTrace();
       return null;
    } }
```

4.prototypedemo.java =

```
import java.util.Scanner;
public class PrototypeDemo {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      Shape shape = null;
      Shape clonedShape = null;
      while (true) {
         System.out.println("Choose a shape to create:");
         System.out.println("1. Circle");
```

```
        System.out.println("2. Rectangle");
        System.out.println("3. Clone the last created shape");
        System.out.println("4. Exit");
        System.out.print("Your choice: ");
        int choice = scanner.nextInt();
        switch (choice) {
          case 1:
            System.out.print("Enter the radius of the Circle: ");
            int radius = scanner.nextInt();
            shape = new Circle(radius);
            shape.draw();
            break;
          case 2:
            System.out.print("Enter the width of the Rectangle: ");
            int width = scanner.nextInt();
            System.out.print("Enter the height of the Rectangle: ");
            int height = scanner.nextInt();
            shape = new Rectangle(width, height);
            shape.draw();
            break;
          case 3:
            if (shape != null) {
              clonedShape = shape.clone();
              System.out.println("Cloning the last created shape:");
              clonedShape.draw();
            } else {
              System.out.println("No shape to clone! Create a shape first.");
            }
            break;
          case 4:
            System.out.println("Exiting...");
            scanner.close();
            return;
          default:
            System.out.println("Invalid choice! Please select again.");
        } } } }
```

Ass3 = Q1. The Product class is currently responsible for both holding product information and calculating tax. Refactor the code by separating these responsibilities into distinct classes, ensuring that each class has only one responsibility.

➔1.product.java =

```
public class Product {
  private String name;
  private double price;
  public Product(String name, double price) {
    this.name = name;
    this.price = price;
  }
```

```
    public String getName() {
        return name;
    }
    public double getPrice() {
        return price;
    }
    @Override
    public String toString() {
        return "Product{name='" + name + "', price=" + price + "}";
    } }
```

2.taxcalculator.java =

```
public class TaxCalculator {
    private static final double TAX_RATE = 0.2;  // 20% tax rate
    public double calculateTax(Product product) {
        return product.getPrice() * TAX_RATE;
    }
    public double calculateTotalPrice(Product product) {
        return product.getPrice() + calculateTax(product);
    } }
```

3.Refractorcodedemo.java =

```
import java.util.Scanner;
public class RefactoredCodeDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter product name: ");
        String name = scanner.nextLine();
        System.out.print("Enter product price: ");
        double price = scanner.nextDouble();
        Product product = new Product(name, price);
        TaxCalculator taxCalculator = new TaxCalculator();
        System.out.println("Product Information: " + product);
        double tax = taxCalculator.calculateTax(product);
        double totalPrice = taxCalculator.calculateTotalPrice(product);
        System.out.println("Tax: " + tax);
        System.out.println("Total Price: " + totalPrice);
        scanner.close();
    } }
```

Q2. Refactor the ShapeCalculator that calculates areas for Circle and Rectangle. Allow it to support new shapes (like Triangle) without modifying the existing code. Make it easy to add future shapes.

➔1.interface shape =

```
public interface Shape {
    double calculateArea();
}
```

2.triangle.java =

```
public class Triangle implements Shape {
        private double base;
```

```
        private double height;
        public Triangle(double base, double height) {
            this.base = base;
            this.height = height;
        }
        @Override
        public double calculateArea() {
            return 0.5 * base * height;
        } }
```

3.rectangle.java =

```
public class Rectangle implements Shape {
        private double width;
        private double height;
        public Rectangle(double width, double height) {
            this.width = width;
            this.height = height;
        }
        @Override
        public double calculateArea() {
            return width * height;
        } }
```

4.circle.java =

```
public class Circle implements Shape {
 private double radius;

 public Circle(double radius) {
    this.radius = radius;
 }
 @Override
 public double calculateArea() {
    return Math.PI * radius * radius;
 } }
```

5.shapefactory.java =

```
public class ShapeFactory {
   public Shape getShape(String shapeType, double... dimensions) {
      if (shapeType == null) {
         return null;
      }
      switch (shapeType.toLowerCase()) {
         case "circle":
            return new Circle(dimensions[0]);
         case "rectangle":
            return new Rectangle(dimensions[0], dimensions[1]);
         case "triangle":
            return new Triangle(dimensions[0], dimensions[1]);
         default:
            return null;
      } } }
```

6.shapecalculator.java =

```
import java.util.Scanner;
public class ShapeCalculator {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      ShapeFactory shapeFactory = new ShapeFactory();
      while (true) {
         System.out.println("Choose a shape to calculate its area:");
         System.out.println("1. Circle");
         System.out.println("2. Rectangle");
         System.out.println("3. Triangle");
         System.out.println("4. Exit");
         System.out.print("Your choice: ");
         int choice = scanner.nextInt();
         Shape shape = null;
         switch (choice) {
            case 1:
               System.out.print("Enter the radius of the Circle: ");
               double radius = scanner.nextDouble();
               shape = shapeFactory.getShape("circle", radius);
               break;
            case 2:
               System.out.print("Enter the width of the Rectangle: ");
               double width = scanner.nextDouble();
               System.out.print("Enter the height of the Rectangle: ");
               double height = scanner.nextDouble();
               shape = shapeFactory.getShape("rectangle", width, height);
               break;
            case 3:
               System.out.print("Enter the base of the Triangle: ");
               double base = scanner.nextDouble();
               System.out.print("Enter the height of the Triangle: ");
               double triHeight = scanner.nextDouble();
               shape = shapeFactory.getShape("triangle", base, triHeight);
               break;
            case 4:
               System.out.println("Exiting...");
               scanner.close();
               return;
            default:
               System.out.println("Invalid choice! Please select again.");
         }
         if (shape != null) {
            System.out.println("The area is: " + shape.calculateArea());
         } } } }
```

Q3. The SavingsAccount class alters how interest is calculated compared to the Account class, breaking the Liskov Substitution Principle (LSP). Refactor the code so that SavingsAccount can replace Account without affecting functionality. Keep interest calculation logic uniform for all account types.

➔1.account.java =

```
public class Account {
   private String accountNumber;
   private double balance;
   protected InterestStrategy interestStrategy;
   public Account(String accountNumber, double balance, InterestStrategy interestStrategy) {
      this.accountNumber = accountNumber;
      this.balance = balance;
      this.interestStrategy = interestStrategy;
   }
   public double calculateInterest() {
      return interestStrategy.calculateInterest(balance);
   } }
```

2.accountdemo.java =

```
import java.util.Scanner;
public class AccountDemo {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      Account account = null;
      while (true) {
         System.out.println("Choose account type to create:");
         System.out.println("1. Regular Account");
         System.out.println("2. Savings Account");
         System.out.println("3. Exit");
         System.out.print("Your choice: ");
         int choice = scanner.nextInt();
         if (choice == 3) {
            break;
         }
         System.out.print("Enter account number: ");
         String accountNumber = scanner.next();
         System.out.print("Enter initial balance: ");
         double balance = scanner.nextDouble();
         switch (choice) {
            case 1:
               account = new Account(accountNumber, balance, new RegularInterestStrategy());
               break;
            case 2:
               account = new SavingsAccount(accountNumber, balance);
               break;
            default:
               System.out.println("Invalid choice! Please select again.");
               continue;
         }
```

```
        System.out.println("Account created: " + accountNumber);
        System.out.println("Initial balance: " + balance);
        System.out.println("Calculated interest: " + account.calculateInterest());
      }
      scanner.close();
   } }
```

3.interface interestratergy =

```
public interface InterestStrategy {
   double calculateInterest(double balance);
}
```

4.regularinterestratergy =

```
public class RegularInterestStrategy implements InterestStrategy {
 @Override
 public double calculateInterest(double balance) {
    return balance * 0.02;
 } }
```

5.savingaccount.java =

```
public class SavingsAccount extends Account {
   public SavingsAccount(String accountNumber, double balance) {
      super(accountNumber, balance, new SavingsInterestStrategy());
   } }
```

6.savingintreststatergy.java =

```
public class SavingsInterestStrategy implements InterestStrategy {
           @Override
           public double calculateInterest(double balance) {
              return balance * 0.03;
           } }
```

Q4. Refactor the TaskManager interface, which has methods like createTask(),deleteTask(), assignTask(), and markTaskAsComplete(). Break it into smaller interfaces: TaskCreation, TaskAssignment, and TaskCompletion. Ensure classes only implement the interfaces they actually need.

➔1.interface taskassignment =

```
public interface TaskAssignment {
   void assignTask(String taskName, String assignee);
}
```

2.interface taskcomplition =

```
public interface TaskCompletion {
   void markTaskAsComplete(String taskName);
   void deleteTask(String taskName);
}
```

3.interface taskcreation =

```
public interface TaskCreation {
   void createTask(String taskName);
}
```

4.taskmanager.java =

```
public class TaskManager implements TaskCreation, TaskAssignment, TaskCompletion {
```

```
    @Override
    public void createTask(String taskName) {
        System.out.println("Task created: " + taskName);
    }
    @Override
    public void assignTask(String taskName, String assignee) {
        System.out.println("Task assigned: " + taskName + " to " + assignee);
    }
    @Override
    public void markTaskAsComplete(String taskName) {
        System.out.println("Task marked as complete: " + taskName);
    }
    @Override
    public void deleteTask(String taskName) {
        System.out.println("Task deleted: " + taskName);
    } }
```

5.taskmanagerdemo.java =

```
import java.util.Scanner;
public class TaskManagerDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        TaskManager taskManager = new TaskManager();
        while (true) {
            System.out.println("Choose an action:");
            System.out.println("1. Create Task");
            System.out.println("2. Assign Task");
            System.out.println("3. Mark Task as Complete");
            System.out.println("4. Delete Task");
            System.out.println("5. Exit");
            System.out.print("Your choice: ");
            int choice = scanner.nextInt();
            scanner.nextLine();
            switch (choice) {
                case 1:
                    System.out.print("Enter task name: ");
                    String taskName = scanner.nextLine();
                    taskManager.createTask(taskName);
                    break;
                case 2:
                    System.out.print("Enter task name: ");
                    taskName = scanner.nextLine();
                    System.out.print("Enter assignee: ");
                    String assignee = scanner.nextLine();
                    taskManager.assignTask(taskName, assignee);
                    break;
                case 3:
                    System.out.print("Enter task name: ");
                    taskName = scanner.nextLine();
```

```
            taskManager.markTaskAsComplete(taskName);
            break;
        case 4:
            System.out.print("Enter task name: ");
            taskName = scanner.nextLine();
            taskManager.deleteTask(taskName);
            break;
        case 5:
            System.out.println("Exiting...");
            scanner.close();
            return;
        default:
            System.out.println("Invalid choice! Please select again.");
    } } } }
```

Ass4 = Q1.Create a directory structure using the Composite pattern. You should be able to add File and Folder objects. The Folder can contain multiple File objects and other Folder objects.

Hint: • Create a Component interface with methods like showDetails().

• Implement File and Folder classes based on this interface.

➔1.interface component =

```
public interface Component {
   void showDetails();
}
```

2.file.java =

```
import java.util.ArrayList;
import java.util.List;
public class File implements Component {
   private String name;
   public File(String name) {
      this.name = name;
   }
   @Override
   public void showDetails() {
      System.out.println("File: " + name);
   } }
```

3.folder.java =

```
import java.util.ArrayList;
import java.util.List;
public class Folder implements Component {
   private String name;
   private List<Component> components = new ArrayList<>();
   public Folder(String name) {
      this.name = name;
   }
   public void addComponent(Component component) {
      components.add(component);
   }
   public void removeComponent(Component component) {
```

```
        components.remove(component);
    }
    public String getName() {
        return name;
    }
    @Override
    public void showDetails() {
        System.out.println("Folder: " + name);
        for (Component component : components) {
            component.showDetails();
        } }
```

4.directorystructure.java =

```
import java.util.Scanner;
public class DirectoryStructure {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Folder rootFolder = new Folder("root");
        while (true) {
            System.out.println("Choose an action:");
            System.out.println("1. Add File");
            System.out.println("2. Add Folder");
            System.out.println("3. Show Details");
            System.out.println("4. Exit");
            System.out.print("Your choice: ");
            int choice = scanner.nextInt();
            scanner.nextLine(); // Consume newline
            switch (choice) {
                case 1:
                    System.out.print("Enter file name: ");
                    String fileName = scanner.nextLine();
                    File file = new File(fileName);
                    rootFolder.addComponent(file);
                    break;
                case 2:
                    System.out.print("Enter folder name: ");
                    String folderName = scanner.nextLine();
                    Folder folder = new Folder(folderName);
                    rootFolder.addComponent(folder);
                    addSubComponents(scanner, folder);
                    break;
                case 3:
                    rootFolder.showDetails();
                    break;
                case 4:
                    System.out.println("Exiting...");
                    scanner.close();
                    return;
                default:
```

```
                System.out.println("Invalid choice! Please select again.");
            } } }
    private static void addSubComponents(Scanner scanner, Folder parentFolder) {
        while (true) {
            System.out.println("Add sub-component to folder: " + parentFolder.getName());
            System.out.println("1. Add File");
            System.out.println("2. Add Folder");
            System.out.println("3. Go Back");
            System.out.print("Your choice: ");
            int choice = scanner.nextInt();
            scanner.nextLine(); // Consume newline
            switch (choice) {
                case 1:
                    System.out.print("Enter file name: ");
                    String fileName = scanner.nextLine();
                    File file = new File(fileName);
                    parentFolder.addComponent(file);
                    break;
                case 2:
                    System.out.print("Enter folder name: ");
                    String folderName = scanner.nextLine();
                    Folder folder = new Folder(folderName);
                    parentFolder.addComponent(folder);
                    addSubComponents(scanner, folder);
                    break;
                case 3:
                    return;
                default:
                    System.out.println("Invalid choice! Please select again.");
            } } } }
```

Q2. Implement a bridge pattern where you have a Vehicle interface with methods startEngine() and stopEngine(). You also have two types of Engine classes: PetrolEngine and ElectricEngine. Implement two vehicle types: Car and Motorbike. Each vehicle should be able to use either type of engine, demonstrating the flexibility of the bridge pattern.
Hint: • Vehicle interface will use Engine interface methods.
➔1.vehicle.java =

```
public abstract class Vehicle {
    protected Engine engine;
    public Vehicle(Engine engine) {
        this.engine = engine;
    }
    public abstract void startEngine();
    public abstract void stopEngine();
}
```

2.interface engine =

```
public interface Engine {
    void start();
```

```
   void stop();
}
```
3.motorbike.java =
```
public class Motorbike extends Vehicle {
        public Motorbike(Engine engine) {
           super(engine);
        }
        @Override
        public void startEngine() {
           System.out.println("Motorbike: ");
           engine.start();
        }
        @Override
        public void stopEngine() {
           System.out.println("Motorbike: ");
           engine.stop();
        } }
```
4.petrolengine.java =
```
public class PetrolEngine implements Engine {
 @Override
 public void start() {
    System.out.println("Petrol engine starting...");
 }
 @Override
 public void stop() {
    System.out.println("Petrol engine stopping...");
 } }
```
5.electricengine.java =
```
public class ElectricEngine implements Engine {
        @Override
        public void start() {
           System.out.println("Electric engine starting...");
        }
        @Override
        public void stop() {
           System.out.println("Electric engine stopping...");
        } }
```
6.car.java =
```
public class Car extends Vehicle {
 public Car(Engine engine) {
    super(engine);
 }
 @Override
 public void startEngine() {
    System.out.println("Car: ");
    engine.start();
 }
```

```
 @Override
 public void stopEngine() {
    System.out.println("Car: ");
    engine.stop();
 } }
```

7.bridgepatterndemo.java =

```
import java.util.Scanner;
public class BridgePatternDemo {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      Vehicle vehicle = null;
      while (true) {
         System.out.println("Choose a vehicle type:");
         System.out.println("1. Car");
         System.out.println("2. Motorbike");
         System.out.println("3. Exit");
         System.out.print("Your choice: ");
         int choice = scanner.nextInt();
         if (choice == 3) {
            break;
         }
         System.out.println("Choose an engine type:");
         System.out.println("1. Petrol Engine");
         System.out.println("2. Electric Engine");
         System.out.print("Your choice: ");
         int engineChoice = scanner.nextInt();
         Engine engine = (engineChoice == 1) ? new PetrolEngine() : new ElectricEngine();
         switch (choice) {
            case 1:
               vehicle = new Car(engine);
               break;
            case 2:
               vehicle = new Motorbike(engine);
               break;
            default:
               System.out.println("Invalid choice! Please select again.");
               continue;
         }
         vehicle.startEngine();
         vehicle.stopEngine();
      }
      scanner.close();
   } }
```

Q3. You are tasked with designing an organizational structure using the composite pattern. Create a Employee class and a Manager class. A Manager can have multiple employees reporting to them. Implement the structure where a Manager can delegate tasks to Employee objects and also manage other Manager objects.

Hint: • Use the composite pattern to represent managers and employees as a tree structure.

➔1.interface employee =

```
public interface Employee {
   void showDetails();
   void performTask(String task);
}
```

2.employeeimpl.java =

```
public class EmployeeImpl implements Employee {
   private String name;
   public EmployeeImpl(String name) {
      this.name = name;
   }
   @Override
   public void showDetails() {
      System.out.println("Employee: " + name);
   }
   @Override
   public void performTask(String task) {
      System.out.println(name + " is performing task: " + task);
   } }
```

3.manager.java =

```
import java.util.ArrayList;
import java.util.List;
public class Manager implements Employee {
   private String name;
   private List<Employee> employees = new ArrayList<>();
   public Manager(String name) {
      this.name = name;
   }
   public void addEmployee(Employee employee) {
      employees.add(employee);
   }
   public void removeEmployee(Employee employee) {
      employees.remove(employee);
   }
   public String getName() {
      return name;
   }
   @Override
   public void showDetails() {
      System.out.println("Manager: " + name);
      for (Employee employee : employees) {
         employee.showDetails();
      } }
```

```
    @Override
    public void performTask(String task) {
        System.out.println(name + " is delegating task: " + task);
        for (Employee employee : employees) {
            employee.performTask(task);
        } } }
```

4.orgnizationstructure.java =

```
import java.util.Scanner;
public class OrganizationStructure {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Manager rootManager = new Manager("CEO");
        while (true) {
            System.out.println("Choose an action:");
            System.out.println("1. Add Employee");
            System.out.println("2. Add Manager");
            System.out.println("3. Show Organization Structure");
            System.out.println("4. Delegate Task");
            System.out.println("5. Exit");
            System.out.print("Your choice: ");
            int choice = scanner.nextInt();
            scanner.nextLine(); // Consume newline
            switch (choice) {
                case 1:
                    System.out.print("Enter employee name: ");
                    String employeeName = scanner.nextLine();
                    Employee employee = new EmployeeImpl(employeeName);
                    rootManager.addEmployee(employee);
                    break;
                case 2:
                    System.out.print("Enter manager name: ");
                    String managerName = scanner.nextLine();
                    Manager manager = new Manager(managerName);
                    rootManager.addEmployee(manager);
                    addSubordinates(scanner, manager);
                    break;
                case 3:
                    rootManager.showDetails();
                    break;
                case 4:
                    System.out.print("Enter task: ");
                    String task = scanner.nextLine();
                    rootManager.performTask(task);
                    break;
                case 5:
                    System.out.println("Exiting...");
                    scanner.close();
                    return;
```

```
            default:
                System.out.println("Invalid choice! Please select again.");
        } } }
    private static void addSubordinates(Scanner scanner, Manager parentManager) {
        while (true) {
            System.out.println("Add sub-ordinate to manager: " + parentManager.getName());
            System.out.println("1. Add Employee");
            System.out.println("2. Add Manager");
            System.out.println("3. Go Back");
            System.out.print("Your choice: ");
            int choice = scanner.nextInt();
            scanner.nextLine();
            switch (choice) {
                case 1:
                    System.out.print("Enter employee name: ");
                    String employeeName = scanner.nextLine();
                    Employee employee = new EmployeeImpl(employeeName);
                    parentManager.addEmployee(employee);
                    break;
                case 2:
                    System.out.print("Enter manager name: ");
                    String managerName = scanner.nextLine();
                    Manager manager = new Manager(managerName);
                    parentManager.addEmployee(manager);
                    addSubordinates(scanner, manager);
                    break;
                case 3:
                    return;
                default:
                    System.out.println("Invalid choice! Please select again.");
            } } } }
```

Q4. Implement a proxy pattern to control access to a BankAccount class. The real class BankAccount has methods like deposit() and withdraw(). The BankAccountProxy will control access to these methods, ensuring that withdrawals can only be made if the account has sufficient balance.

Hint: • The proxy class can check if the account has enough balance before allowing the withdrawal.

➔1.interface banckaccount =

```
public interface BankAccount {
    void deposit(double amount);
    void withdraw(double amount);
    double getBalance();
}
```

2.bankaccountproxy.java =

```
public class BankAccountProxy implements BankAccount {
    private RealBankAccount realBankAccount;
    public BankAccountProxy() {
```

```
        this.realBankAccount = new RealBankAccount();
    }
    @Override
    public void deposit(double amount) {
        realBankAccount.deposit(amount);
    }
    @Override
    public void withdraw(double amount) {
        if (realBankAccount.getBalance() >= amount) {
            realBankAccount.withdraw(amount);
        } else {
            System.out.println("Proxy: Insufficient balance to withdraw: " + amount);
        } }
    @Override
    public double getBalance() {
        return realBankAccount.getBalance();
    } }
```

3.proxypatterndemo.java =

```
import java.util.Scanner;
public class ProxyPatternDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        BankAccount bankAccount = new BankAccountProxy();
        while (true) {
            System.out.println("Choose an action:");
            System.out.println("1. Deposit");
            System.out.println("2. Withdraw");
            System.out.println("3. Check Balance");
            System.out.println("4. Exit");
            System.out.print("Your choice: ");
            int choice = scanner.nextInt();
            switch (choice) {
                case 1:
                    System.out.print("Enter amount to deposit: ");
                    double depositAmount = scanner.nextDouble();
                    bankAccount.deposit(depositAmount);
                    break;
                case 2:
                    System.out.print("Enter amount to withdraw: ");
                    double withdrawAmount = scanner.nextDouble();
                    bankAccount.withdraw(withdrawAmount);
                    break;
                case 3:
                    System.out.println("Current balance: " + bankAccount.getBalance());
                    break;
                case 4:
                    System.out.println("Exiting...");
                    scanner.close();
```

```
                return;
            default:
                System.out.println("Invalid choice! Please select again.");
        } } } }
```

4.realbankaccount.java =

```
public class RealBankAccount implements BankAccount {
    private double balance;
    @Override
    public void deposit(double amount) {
        balance += amount;
        System.out.println("Deposited: " + amount);
    }
    @Override
    public void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println("Withdrew: " + amount);
        } else {
            System.out.println("Insufficient balance to withdraw: " + amount);
        } }
    @Override
    public double getBalance() {
        return balance;
    } }
```