Task 1: Singleton
Implement a Singleton class that manages database connections. Ensure the class adheres strictly to the singleton pattern principles.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnectionManager {

    // Database URL, username and password
    private static final String DB_URL = "jdbc:mysql://localhost:3306/mydatabase";
    private static final String DB_USER = "username";
    private static final String DB_PASSWORD = "password";

    // Private constructor to prevent instantiation
    private DatabaseConnectionManager() {
    }

    // Static inner class - inner classes are not loaded until they are referenced
    private static class Holder {
        private static final DatabaseConnectionManager INSTANCE = new
DatabaseConnectionManager();
    }

    // Public method to provide access to the singleton instance
    public static DatabaseConnectionManager getInstance() {
        return Holder.INSTANCE;
    }

    // Method to get a database connection
    public Connection getConnection() {
        Connection connection = null;
        try {
            connection = DriverManager.getConnection(DB_URL, DB_USER,
DB_PASSWORD);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return connection;
```

```
    }

    public static void main(String[] args) {
        // Get the singleton instance of DatabaseConnectionManager
        DatabaseConnectionManager dbManager =
DatabaseConnectionManager.getInstance();

        // Get a database connection
        Connection connection = dbManager.getConnection();

        if (connection != null) {
            System.out.println("Database connection established.");
        } else {
            System.out.println("Failed to establish database connection.");
        }

        // Don't forget to close the connection after use
        try {
            if (connection != null && !connection.isClosed()) {
                connection.close();
                System.out.println("Database connection closed.");
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Task 2: Factory Method

Create a ShapeFactory class that encapsulates the object creation logic of different Shape objects like Circle, Square, and Rectangle.

```
// Shape interface
interface Shape {
    void draw();
}

// Concrete Circle class implementing Shape interface
class Circle implements Shape {
```

```
    @Override
    public void draw() {
        System.out.println("Drawing Circle");
    }
}

// Concrete Square class implementing Shape interface
class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Square");
    }
}

// Concrete Rectangle class implementing Shape interface
class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing Rectangle");
    }
}

// ShapeFactory class to create Shape objects
public class ShapeFactory {

    // Method to create Shape objects based on the shape type
    public Shape createShape(String shapeType) {
        if (shapeType == null) {
            return null;
        }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new Square();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        }
        return null;
    }
```

```
    // Main method to test the ShapeFactory
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        // Create a Circle and call draw method
        Shape circle = shapeFactory.createShape("CIRCLE");
        if (circle != null) {
            circle.draw();
        }

        // Create a Square and call draw method
        Shape square = shapeFactory.createShape("SQUARE");
        if (square != null) {
            square.draw();
        }

        // Create a Rectangle and call draw method
        Shape rectangle = shapeFactory.createShape("RECTANGLE");
        if (rectangle != null) {
            rectangle.draw();
        }
    }
}
```

Output
Drawing Circle
Drawing Square
Drawing Rectangle

Task 3: Proxy
Create a proxy class for accessing a sensitive object that contains a secret key. The proxy should only allow access to the secret key if a correct password is provided.

```
public class SecretKeyProxy {

    // Interface for the sensitive object
    interface SecretKeyManager {
        String getSecretKey();
    }
```

```
    // Concrete implementation of the sensitive object
    static class RealSecretKeyManager implements SecretKeyManager {
        private String secretKey = "mySecretKey123"; // Example secret key

        @Override
        public String getSecretKey() {
            return secretKey;
        }
    }

    // Proxy class that controls access based on a password
    static class SecretKeyAccessProxy implements SecretKeyManager {
        private SecretKeyManager secretKeyManager = new RealSecretKeyManager();
        private String password;

        public SecretKeyAccessProxy(String password) {
            this.password = password;
        }

        @Override
        public String getSecretKey() {
            if (authenticate()) {
                return secretKeyManager.getSecretKey();
            } else {
                System.out.println("Unauthorized access. Incorrect password.");
                return null;
            }
        }

        private boolean authenticate() {
            // Simulate password authentication (replace with your actual authentication
logic)
            return "correctPassword".equals(password);
        }
    }

    public static void main(String[] args) {
        // Create proxy with a correct password
        SecretKeyManager proxy = new SecretKeyAccessProxy("correctPassword");
```

```
        // Access the secret key through the proxy
        String secretKey = proxy.getSecretKey();
        if (secretKey != null) {
            System.out.println("Access granted. Secret key: " + secretKey);
        }

        // Try accessing with incorrect password
        SecretKeyManager proxyIncorrect = new
SecretKeyAccessProxy("incorrectPassword");
        String secretKeyIncorrect = proxyIncorrect.getSecretKey();
        if (secretKeyIncorrect != null) {
            System.out.println("Access granted. Secret key: " + secretKeyIncorrect);
        }
    }
}
```

Output
Access granted. Secret key: mySecretKey123
Unauthorized access. Incorrect password.

Task 4: Strategy
Develop a Context class that can use different SortingStrategy algorithms interchangeably to sort a collection of numbers

```
import java.util.Arrays;

// Interface for sorting strategy
interface SortingStrategy {
    void sort(int[] numbers);
}

// Bubble sort strategy
class BubbleSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        System.out.println("Sorting using Bubble Sort");
        // Bubble sort logic
        int n = numbers.length;
        for (int i = 0; i < n - 1; i++) {
```

```
            for (int j = 0; j < n - i - 1; j++) {
                if (numbers[j] > numbers[j + 1]) {
                    // Swap numbers[j] and numbers[j+1]
                    int temp = numbers[j];
                    numbers[j] = numbers[j + 1];
                    numbers[j + 1] = temp;
                }
            }
        }
    }
}

// Quick sort strategy
class QuickSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        System.out.println("Sorting using Quick Sort");
        // Quick sort logic (using Arrays.sort for simplicity)
        Arrays.sort(numbers);
    }
}

// Context class that uses different sorting strategies
public class Context {
    private SortingStrategy sortingStrategy;

    public void setSortingStrategy(SortingStrategy sortingStrategy) {
        this.sortingStrategy = sortingStrategy;
    }

    public void performSort(int[] numbers) {
        sortingStrategy.sort(numbers);
    }

    public static void main(String[] args) {
        Context context = new Context();

        // Sorting using Bubble Sort
        context.setSortingStrategy(new BubbleSortStrategy());
        int[] numbers1 = {5, 2, 7, 1, 3};
```

```
        context.performSort(numbers1);
        System.out.println("Sorted numbers using Bubble Sort: " +
Arrays.toString(numbers1));

        // Sorting using Quick Sort
        context.setSortingStrategy(new QuickSortStrategy());
        int[] numbers2 = {5, 2, 7, 1, 3};
        context.performSort(numbers2);
        System.out.println("Sorted numbers using Quick Sort: " +
Arrays.toString(numbers2));
    }
}
```

Output

```
Sorting using Bubble Sort
Sorted numbers using Bubble Sort: [1, 2, 3, 5, 7]
Sorting using Quick Sort
Sorted numbers using Quick Sort: [1, 2, 3, 5, 7]
```