Task 1.

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

```java
package Day23;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class DatabaseConnection {
    private static DatabaseConnection instance;
    private Connection connection;
    private Statement statement;

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

    public static synchronized DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }

    public void connect(String dbUrl) throws SQLException {
        if (connection == null) {
            connection = DriverManager.getConnection(dbUrl);
            statement = connection.createStatement();
        }
    }

    public void close() throws SQLException {
        if (connection != null) {
            connection.close();
            connection = null;
            statement = null;
        }
    }

    public void executeQuery(String query) throws SQLException {
        if (connection != null) {
            statement.execute(query);
        } else {
            throw new SQLException("Database connection is not established.");
        }
    }

    public ResultSet fetchAll(String query) throws SQLException {
        if (connection != null) {
            return statement.executeQuery(query);
        } else {
            throw new SQLException("Database connection is not established.");
        }
    }
}
```

```
package Day23;

public class Main{}
public static void main(String[] args) {
    try {
        // Creating the Singleton instance and connecting to a database
        DatabaseConnection db = DatabaseConnection.getInstance();
        db.connect("jdbc:sqlite:example.db");

        // Execute a query to create a table
        db.executeQuery("CREATE TABLE IF NOT EXISTS example (id INTEGER PRIMARY KEY, name TEXT)");

        // Insert data
        db.executeQuery("INSERT INTO example (name) VALUES ('John Doe')");

        // Fetch data
        ResultSet rs = db.fetchAll("SELECT * FROM example");
        while (rs.next()) {
            System.out.println("ID: " + rs.getInt("id") + ", Name: " + rs.getString("name"));
        }

        // Close the connection
        db.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Task2:

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

Ans;

```
// Shape.java
public interface Shape {
    void draw();
}

// Circle.java
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

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

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

// ShapeFactory.java
public class ShapeFactory {

    // Factory Method
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;
        }

        switch (shapeType.toUpperCase()) {
            case "CIRCLE":
                return new Circle();
            case "SQUARE":
                return new Square();
            case "RECTANGLE":
                return new Rectangle();
            default:
                return null;
        }
    }
}

// FactoryPatternDemo.java
public class FactoryPatternDemo {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        // Get an object of Circle and call its draw method
        Shape shape1 = shapeFactory.getShape("CIRCLE");
        shape1.draw();

        // Get an object of Square and call its draw method
        Shape shape2 = shapeFactory.getShape("SQUARE");
        shape2.draw();

        // Get an object of Rectangle and call its draw method
        Shape shape3 = shapeFactory.getShape("RECTANGLE");
        shape3.draw();
    }
}
```

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.

Ans;

```
// SensitiveObject.java
public class SensitiveObject {
    private String secretKey;

    public SensitiveObject(String secretKey) {
        this.secretKey = secretKey;
    }

    public String getSecretKey() {
        return secretKey;
    }
}
// SecretAccess.java
public interface SecretAccess {
    String getSecretKey(String password);
}
// SensitiveObjectProxy.java
public class SensitiveObjectProxy implements SecretAccess {
    private SensitiveObject sensitiveObject;
    private String correctPassword;

    public SensitiveObjectProxy(String secretKey, String correctPassword) {
        this.sensitiveObject = new SensitiveObject(secretKey);
        this.correctPassword = correctPassword;
    }

    @Override
    public String getSecretKey(String password) {
        if (password.equals(correctPassword)) {
            return sensitiveObject.getSecretKey();
        } else {
            return "Access Denied: Incorrect Password";
        }
    }
}
// ProxyPatternDemo.java
public class ProxyPatternDemo {
    public static void main(String[] args) {
        String secretKey = "MySecretKey123";
        String correctPassword = "password123";

        SensitiveObjectProxy proxy = new SensitiveObjectProxy(secretKey, correctPassword);

        // Try to access the secret key with the correct password
        String key = proxy.getSecretKey("password123");
        System.out.println("Access with correct password: " + key);

        // Try to access the secret key with an incorrect password
        key = proxy.getSecretKey("wrongpassword");
        System.out.println("Access with incorrect password: " + key);
    }
}
```

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

Ans:

```
// SortingStrategy.java
public interface SortingStrategy {
    void sort(int[] numbers);
}

// BubbleSortStrategy.java
public class BubbleSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        System.out.println("Sorting using Bubble Sort");
        // Actual implementation of Bubble Sort
        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;
                }
            }
        }
    }
}

// QuickSortStrategy.java
public class QuickSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        System.out.println("Sorting using Quick Sort");
        // Actual implementation of Quick Sort
        quickSort(numbers, 0, numbers.length - 1);
    }

    private void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }
    private int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }
}
// Context.java
public class Context {
    private SortingStrategy sortingStrategy;

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

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

```
}
// StrategyPatternDemo.java
public class StrategyPatternDemo {
    public static void main(String[] args) {
        int[] numbers = {5, 1, 4, 2, 8};

        Context context = new Context();

        // Use Bubble Sort strategy
        context.setSortingStrategy(new BubbleSortStrategy());
        context.performSort(numbers.clone()); // Cloning to keep the original array

        System.out.println("After Bubble Sort:");
        printArray(numbers);

        // Use Quick Sort strategy
        context.setSortingStrategy(new QuickSortStrategy());
        context.performSort(numbers.clone()); // Cloning again for another sort

        System.out.println("After Quick Sort:");
        printArray(numbers);
    }

    private static void printArray(int[] arr) {
        for (int i : arr) {
            System.out.print(i + " ");
        }
        System.out.println();
    }}
```