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

```
package wipro.com.assignment17;

public class SingletonExample {

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

        // Use the singleton instance to connect and disconnect
        dbManager.connect();
        dbManager.disconnect();
    }
}

public class DatabaseConnectionManager {

    // Private static instance variable
    private static DatabaseConnectionManager instance;

    // Private constructor to prevent instantiation outside the class
    private DatabaseConnectionManager() {
        // Initialize database connection setup here
        // For demonstration purposes, let's assume initialization code
        System.out.println("Initializing database connection...");
    }

    // Public static method to get the singleton instance
    public static synchronized DatabaseConnectionManager getInstance() {
        if (instance == null) {
            instance = new DatabaseConnectionManager();
        }
        return instance;
    }

    // Public method to provide database connection
    public void connect() {
        System.out.println("Connecting to database...");
        // Implement database connection logic here
    }

    // Public method to disconnect from database
    public void disconnect() {
        System.out.println("Disconnecting from database...");
        // Implement database disconnection logic here
    }

    // Other methods related to database operations can be added here
}
```

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:

```
package wipro.com.assignment17;
//Shape.java
public interface Shape {
 void draw();
}
```

Concrete Shape Implementations:

```
// Circle.java
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}

// Square.java
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}
```

```
// Rectangle.java

public class Rectangle implements Shape {

   @Override

   public void draw() {

      System.out.println("Inside Rectangle::draw() method.");

   }

}
```

To implement a Factory Method pattern for creating different Shape objects (Circle, Square, Rectangle), we'll create a `ShapeFactory` class that encapsulates the object creation logic. Each shape will implement a common `Shape` interface to ensure polymorphic behavior.

Shape Interface

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

Concrete Shape Implementations

```
java
Copy code
// Circle.java
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}

// Square.java
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}
```

```
// Rectangle.java
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}
```

ShapeFactory Class

```
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.

```
// SecretKey.java

public interface SecretKey {

   void accessKey(String password);

}
```

```
// RealObject.java
```

```
public class RealObject implements SecretKey {

    private String secretKey = "********"; // Actual secret key (can be any sensitive data)

    @Override
    public void accessKey(String password) {
        if (password.equals("password123")) { // Replace with your actual password check logic
            System.out.println("Access granted. Secret Key: " + secretKey);
        } else {
            System.out.println("Access denied. Incorrect password.");
        }
    }
}

// Proxy.java
public class Proxy implements SecretKey {

    private RealObject realObject;
    private String password;

    public Proxy(String password) {
        this.password = password;
    }
```

```
    @Override
    public void accessKey(String password) {
        if (realObject == null) {
            realObject = new RealObject();
        }
        realObject.accessKey(password);
    }
}

public class ProxyPatternDemo {

    public static void main(String[] args) {
        // Create a Proxy with a password
        SecretKey proxy = new Proxy("password123");

        // Access the secret key with correct password
        proxy.accessKey("password123");

        // Access attempt with incorrect password
        proxy.accessKey("wrongpassword");
    }
}
```

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

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

// BubbleSortStrategy.java
public class BubbleSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        // Implement Bubble Sort algorithm
        int n = array.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    // Swap array[j] and array[j+1]
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
        System.out.println("Sorting array using Bubble Sort.");
    }
}
```

```
// MergeSortStrategy.java
public class MergeSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        // Implement Merge Sort algorithm
        mergeSort(array, 0, array.length - 1);
        System.out.println("Sorting array using Merge Sort.");
    }

    private void mergeSort(int[] array, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            mergeSort(array, left, mid);
            mergeSort(array, mid + 1, right);
            merge(array, left, mid, right);
        }
    }

    private void merge(int[] array, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;
        int[] L = new int[n1];
        int[] R = new int[n2];
```

```
for (int i = 0; i < n1; ++i)
    L[i] = array[left + i];
for (int j = 0; j < n2; ++j)
    R[j] = array[mid + 1 + j];

int i = 0, j = 0;
int k = left;
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        array[k] = L[i];
        i++;
    } else {
        array[k] = R[j];
        j++;
    }
    k++;
}

while (i < n1) {
    array[k] = L[i];
    i++;
    k++;
}

while (j < n2) {
```

```
            array[k] = R[j];
            j++;
            k++;
        }
    }
}
```