**Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

The Singleton Pattern is a creational design pattern that ensures a class has only one instance, and provides a global point of access to that instance.

**🔹 Key Features of Singleton:**

1. **Single Instance**: Only one object of the class is ever created.
2. **Global Access Point**: That object can be accessed from anywhere in the application.
3. **Private Constructor**: Prevents instantiation from other classes.
4. **Static Method**: Used to get the instance of the class

In the below example, The DatabaseConnection class implements the Singleton design pattern to ensure only one instance of the database connection exists. It provides methods to simulate connecting and disconnecting from a database.

**🔸 DatabaseConnection.java**

package com.example.singleton;

public class DatabaseConnection {

private static DatabaseConnection instance;

// Private constructor to prevent instantiation

private DatabaseConnection() {

System.out.println("Database Connection Created.");

}

// Public method to provide access to the instance

public static DatabaseConnection getInstance() {

if (instance == null) {

synchronized (DatabaseConnection.class) {

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...");

}

}

**🔸 TestConnection.java**

package com.example.singleton;

public class TestConnection {

public static void main(String[] args) {

// Retrieve the singleton instance

DatabaseConnection connection1 = DatabaseConnection.getInstance();

DatabaseConnection connection2 = DatabaseConnection.getInstance();

System.out.println();

// Validate both instances are the same

if (connection1 == connection2) {

System.out.println("Both are the same database connection instance.");

} else {

System.out.println("Different instances detected!");

}

// Use the connection

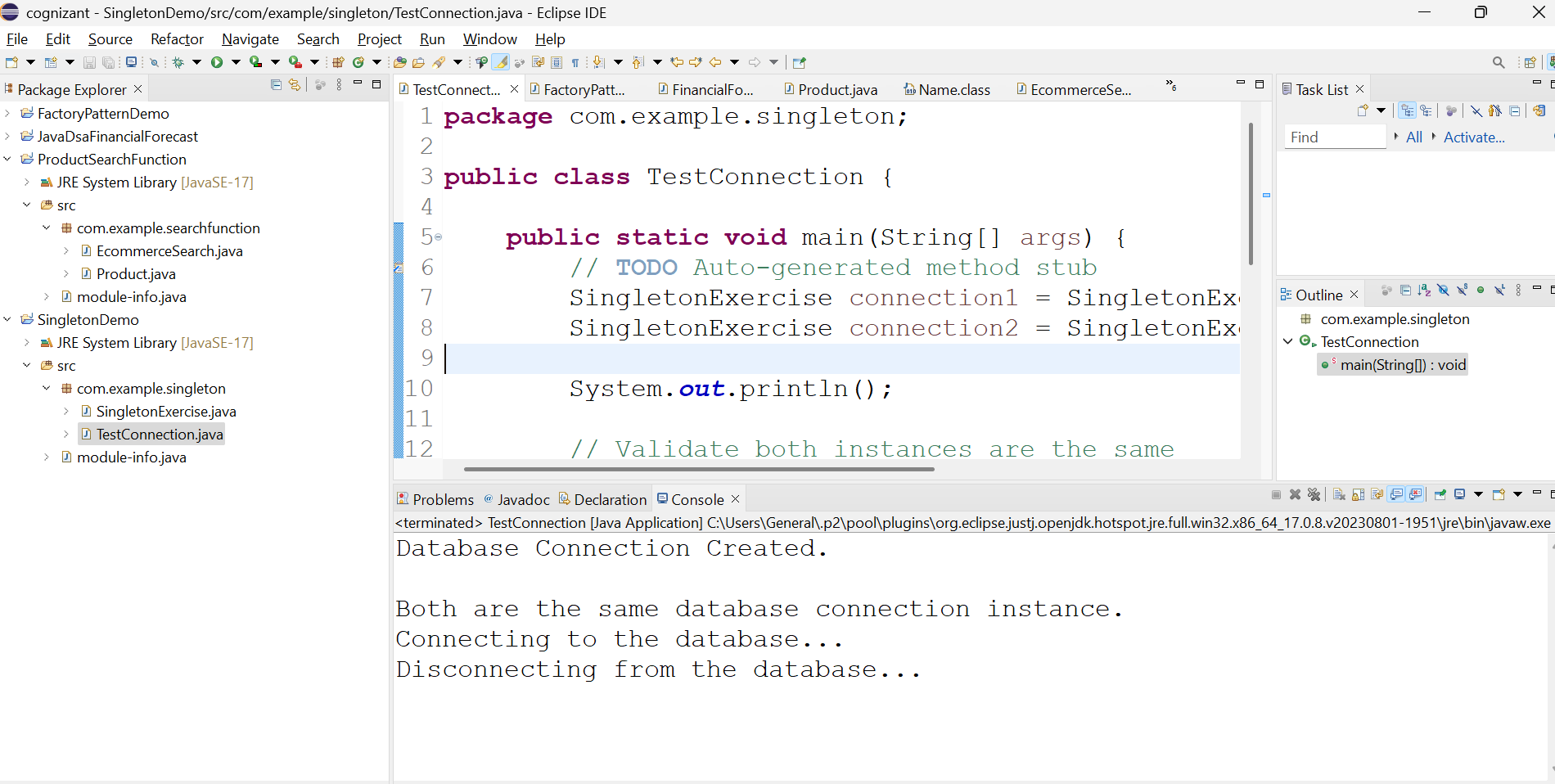
connection1.connect();

connection2.disconnect();

}

}

**OUTPUT:**

****

**2.** **Exercise 2: Implementing the Factory Method Pattern**

The Factory Method Pattern is a creational design pattern that provides an interface (or method) to create objects in a superclass, but allows subclasses or separate classes (factories) to decide which class to instantiate.

**🔷 Purpose:**

To **encapsulate object creation logic** and avoid using new keyword directly in the main code. This provides **loose coupling** between client code and the specific classes it uses.

Below example

* createNotification("EMAIL") is the **factory method**.
* The client code (**main method**) doesn’t use new EmailNotification() directly.
* The **factory method** decides which subclass (Email, SMS, or Push) to instantiate **based on input**.
* This encapsulates object creation logic in one place (NotificationFactory), not scattered throughout the code.

🔸 **1. Notification.java**

package com.example.factory;

public interface Notification {

void notifyUser();

}

🔸 **2. EmailNotification.java**

package com.example.factory;

public class EmailNotification implements Notification {

public void notifyUser() {

System.out.println("Sending an Email Notification");

}

}

🔸 **3. SMSNotification.java**

package com.example.factory;

public class SMSNotification implements Notification {

public void notifyUser() {

System.out.println("Sending an SMS Notification");

}

}

🔸 **4. PushNotification.java**

package com.example.factory;

public class PushNotification implements Notification {

public void notifyUser() {

System.out.println("Sending a Push Notification");

}

}

🔸 **5. NotificationFactory.java**

package com.example.factory;

public class NotificationFactory {

public Notification createNotification(String type) {

if (type == null || type.isEmpty())

return null;

if (type.equalsIgnoreCase("EMAIL"))

return new EmailNotification();

else if (type.equalsIgnoreCase("SMS"))

return new SMSNotification();

else if (type.equalsIgnoreCase("PUSH"))

return new PushNotification();

return null;

}

}

🔸 **6. FactoryPatternTest.java**

package com.example.factory;

public class FactoryPatternTest {

public static void main(String[] args) {

NotificationFactory factory = new NotificationFactory();

Notification n1 = factory.createNotification("EMAIL");

Notification n2 = factory.createNotification("SMS");

Notification n3 = factory.createNotification("PUSH");

n1.notifyUser();

n2.notifyUser();

n3.notifyUser();

}

}

**OUTPUT:**

