Design Patterns Handson

# Implementing the Singleton Pattern

# Overview

The Singleton pattern ensures that only one instance of a class is created and provides a global point of access to it.

## Logger.java

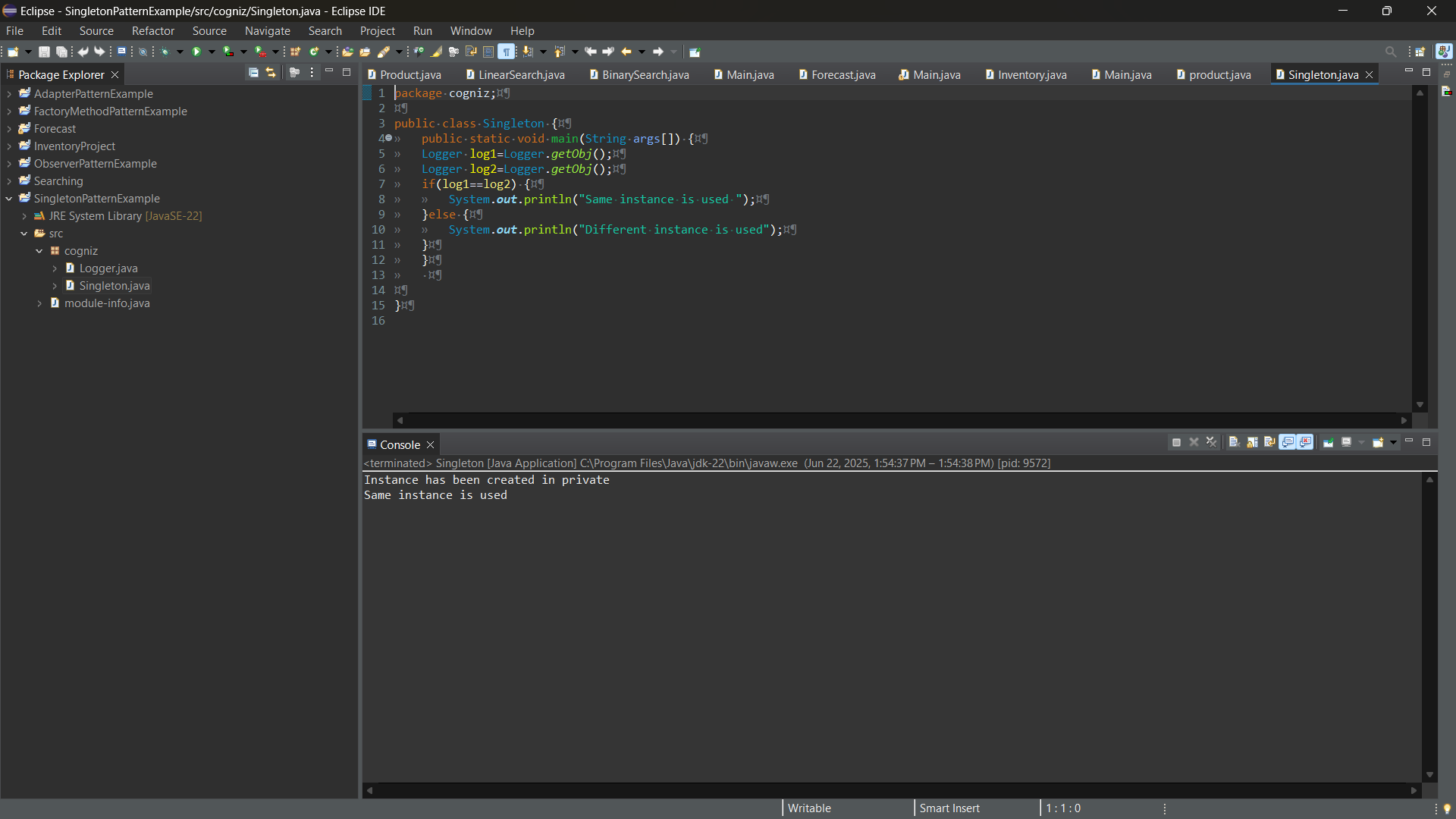
package cogniz;  
  
public class Logger {  
 private static Logger instance;  
   
 private Logger() {  
 System.out.println("Instance has been created in private");  
 }  
   
 public static Logger getObj() {  
 if (instance == null) {  
 instance = new Logger();  
 }  
 return instance;  
 }  
}

## 1 Singleton.java (Test Class)

package cogniz;  
  
public class Singleton {  
 public static void main(String args[]) {  
 Logger log1 = Logger.getObj();  
 Logger log2 = Logger.getObj();  
   
 if (log1 == log2) {  
 System.out.println("Same instance is used");  
 } else {  
 System.out.println("Different instance is used");  
 }  
 }  
}

## Output

Expected Output:  
Instance has been created in private  
Same instance is used



# 2. Implementing the Factory Method Pattern

# Overview

The Factory Method pattern defines an interface for creating objects but lets subclasses decide which class to instantiate.

## Document.java (Interface)

package factory;  
  
public interface Document {  
 String open();  
}

## DocumentFactory.java (Abstract Factory)

package factory;  
  
public abstract class DocumentFactory {  
 public abstract Document createDocument();  
}

## WordDocumentFactory.java

package factory;  
  
public class WordDocumentFactory extends DocumentFactory {  
 @Override  
 public Document createDocument() {  
 return new WordDocument();  
 }  
}  
  
class WordDocument implements Document {  
 public String open() {  
 return "creating word document";  
 }  
}

## PdfDocumentFactory.java

package factory;  
  
public class PdfDocumentFactory extends DocumentFactory {  
 @Override  
 public Document createDocument() {  
 return new PdfDocument();  
 }  
}  
  
class PdfDocument implements Document {  
 public String open() {  
 return "creating pdf document";  
 }  
}

## ExcelDocumentFactory.java

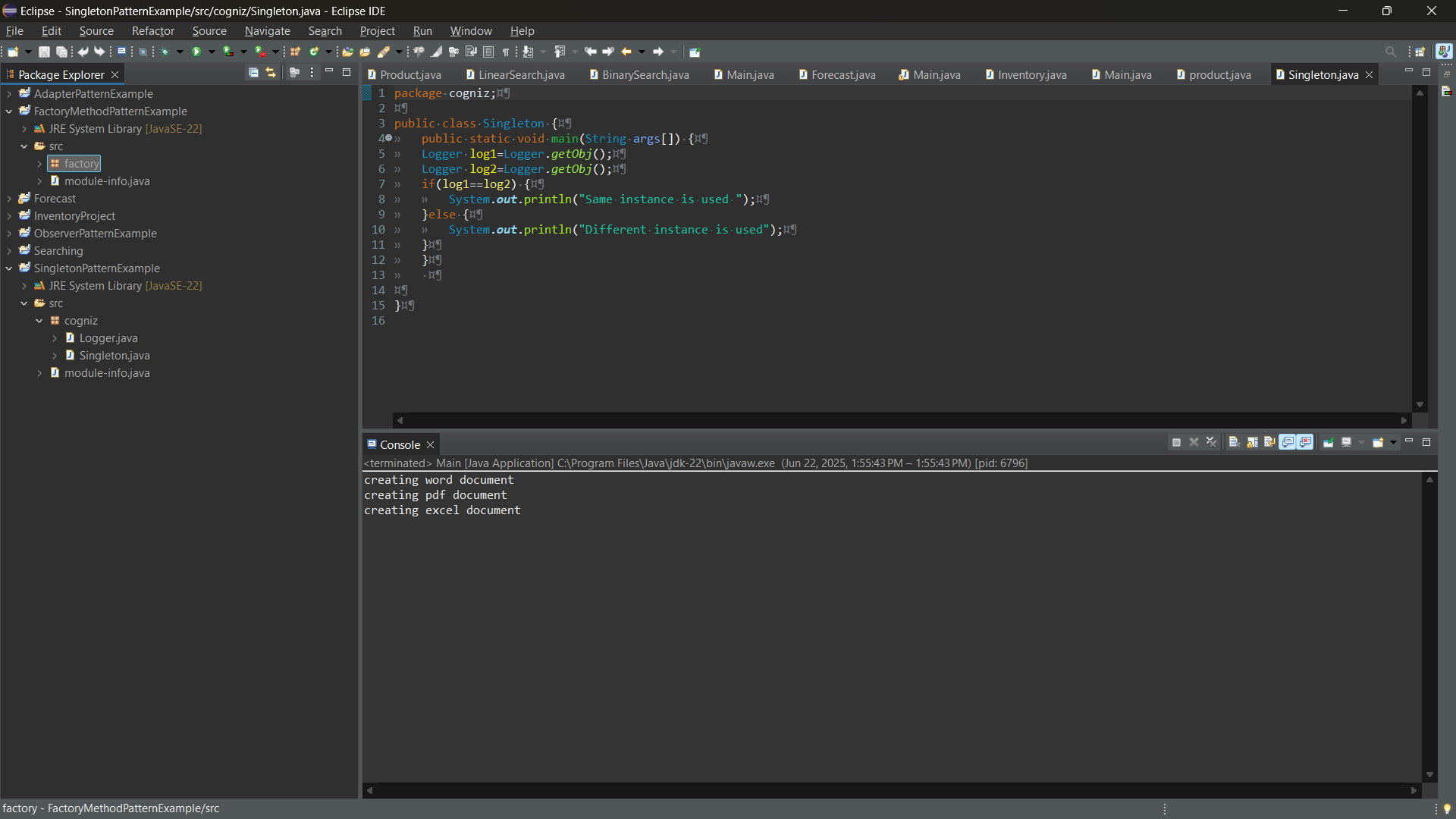
package factory;  
  
public class ExcelDocumentFactory extends DocumentFactory {  
 @Override  
 public Document createDocument() {  
 return new ExcelDocument();  
 }  
}  
  
class ExcelDocument implements Document {  
 public String open() {  
 return "creating excel document";  
 }  
}

## Main.java (Client Code)

package factory;  
  
public class Main {  
 public static void main(String[] args) {  
 DocumentFactory wordFactory = new WordDocumentFactory();  
 Document wordDoc = wordFactory.createDocument();  
 System.out.println(wordDoc.open());  
  
 DocumentFactory pdfFactory = new PdfDocumentFactory();  
 Document pdfDoc = pdfFactory.createDocument();  
 System.out.println(pdfDoc.open());  
  
 DocumentFactory excelFactory = new ExcelDocumentFactory();  
 Document excelDoc = excelFactory.createDocument();  
 System.out.println(excelDoc.open());  
 }  
}

## Output

Expected Output:  
creating word document  
creating pdf document  
creating excel document



## Implementing the Observer Pattern

## Overview

The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.

## Observer.java (Interface)

package observer.com;  
  
public interface Observer {  
 void update(double stockPrice);  
}

## Stock.java (Subject Interface)

package observer.com;  
  
public interface Stock {  
 void registerObserver(Observer o);  
 void deregisterObserver(Observer o);  
 void notifyObservers();  
}

## StockMarket.java (Concrete Subject)

package observer.com;  
  
import java.util.\*;  
  
public class StockMarket implements Stock {  
 private List<Observer> observers = new ArrayList<>();  
 private double stockPrice;  
  
 @Override  
 public void registerObserver(Observer o) {  
 observers.add(o);  
 }  
  
 @Override  
 public void deregisterObserver(Observer o) {  
 observers.remove(o);  
 }  
  
 @Override  
 public void notifyObservers() {  
 for (Observer o : observers) {  
 o.update(stockPrice);  
 }  
 }  
  
 public void setStockPrice(double price) {  
 this.stockPrice = price;  
 notifyObservers();  
 }  
}

## MobileApp.java (Concrete Observer)

package observer.com;  
  
public class MobileApp implements Observer {  
 private String name;  
  
 public MobileApp(String name) {  
 this.name = name;  
 }  
  
 public void update(double stockPrice) {  
 System.out.println(name + " - Mobile App: Stock price updated to " + stockPrice);  
 }  
}

## WebApp.java (Concrete Observer)

package observer.com;  
  
public class WebApp implements Observer {  
 private String name;  
  
 public WebApp(String name) {  
 this.name = name;  
 }  
  
 public void update(double stockPrice) {  
 System.out.println(name + " - Web App: Stock price updated to " + stockPrice);  
 }  
}

## Main.java (Client Code)

package observer.com;  
  
public class Main {  
 public static void main(String args[]) {  
 StockMarket st = new StockMarket();  
 Observer o1 = new MobileApp("Krishna");  
 Observer o2 = new WebApp("Prakash");  
  
 st.registerObserver(o1);  
 st.registerObserver(o2);  
  
 st.setStockPrice(100);  
 }  
}

## Output

Expected Output:  
Krishna - Mobile App: Stock price updated to 100.0  
Prakash - Web App: Stock price updated to 100.0

