Stock Market Monitoring with Observer Pattern

import java.util.ArrayList;

import java.util.List;

public class ObserverPatternExample {

public static void main(String[] args) {

// Create stock market subject

StockMarket stockMarket = new StockMarket();

// Create observers

Observer mobileApp = new MobileApp("User1's iPhone");

Observer webApp = new WebApp("User1's Browser");

Observer desktopApp = new DesktopApp("User2's Desktop");

// Register observers

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

System.out.println("=== Stock Market Opens ===");

stockMarket.setStockPrice("AAPL", 150.25);

stockMarket.setStockPrice("GOOGL", 2750.50);

// Register another observer during trading

System.out.println("\n=== New Observer Joins ===");

stockMarket.registerObserver(desktopApp);

stockMarket.setStockPrice("MSFT", 300.42);

// Unregister an observer

System.out.println("\n=== Observer Leaves ===");

stockMarket.unregisterObserver(webApp);

stockMarket.setStockPrice("TSLA", 850.75);

// Price change with fewer observers

System.out.println("\n=== End of Day Trading ===");

stockMarket.setStockPrice("AAPL", 152.10);

}

// Subject Interface

interface Stock {

void registerObserver(Observer o);

void unregisterObserver(Observer o);

void notifyObservers();

}

// Concrete Subject

static class StockMarket implements Stock {

private List<Observer> observers = new ArrayList<>();

private String stockSymbol;

private double stockPrice;

public void setStockPrice(String stockSymbol, double stockPrice) {

this.stockSymbol = stockSymbol;

this.stockPrice = stockPrice;

notifyObservers();

}

@Override

public void registerObserver(Observer o) {

observers.add(o);

System.out.println("New observer registered: " + o.getName());

}

@Override

public void unregisterObserver(Observer o) {

observers.remove(o);

System.out.println("Observer unregistered: " + o.getName());

}

@Override

public void notifyObservers() {

System.out.println("\nStock update: " + stockSymbol + " = $" + stockPrice);

for (Observer observer : observers) {

observer.update(stockSymbol, stockPrice);

}

}

}

// Observer Interface

interface Observer {

void update(String stockSymbol, double stockPrice);

String getName();

}

// Concrete Observers

static class MobileApp implements Observer {

private final String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(String stockSymbol, double stockPrice) {

System.out.println(name + " received update - " + stockSymbol + ": $" + stockPrice);

}

@Override

public String getName() {

return name;

}

}

static class WebApp implements Observer {

private final String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(String stockSymbol, double stockPrice) {

System.out.println(name + " received update - " + stockSymbol + ": $" + stockPrice);

}

@Override

public String getName() {

return name;

}

}

static class DesktopApp implements Observer {

private final String name;

public DesktopApp(String name) {

this.name = name;

}

@Override

public void update(String stockSymbol, double stockPrice) {

System.out.println(name + " received update - " + stockSymbol + ": $" + stockPrice);

}

@Override

public String getName() {

return name;

}

}

}

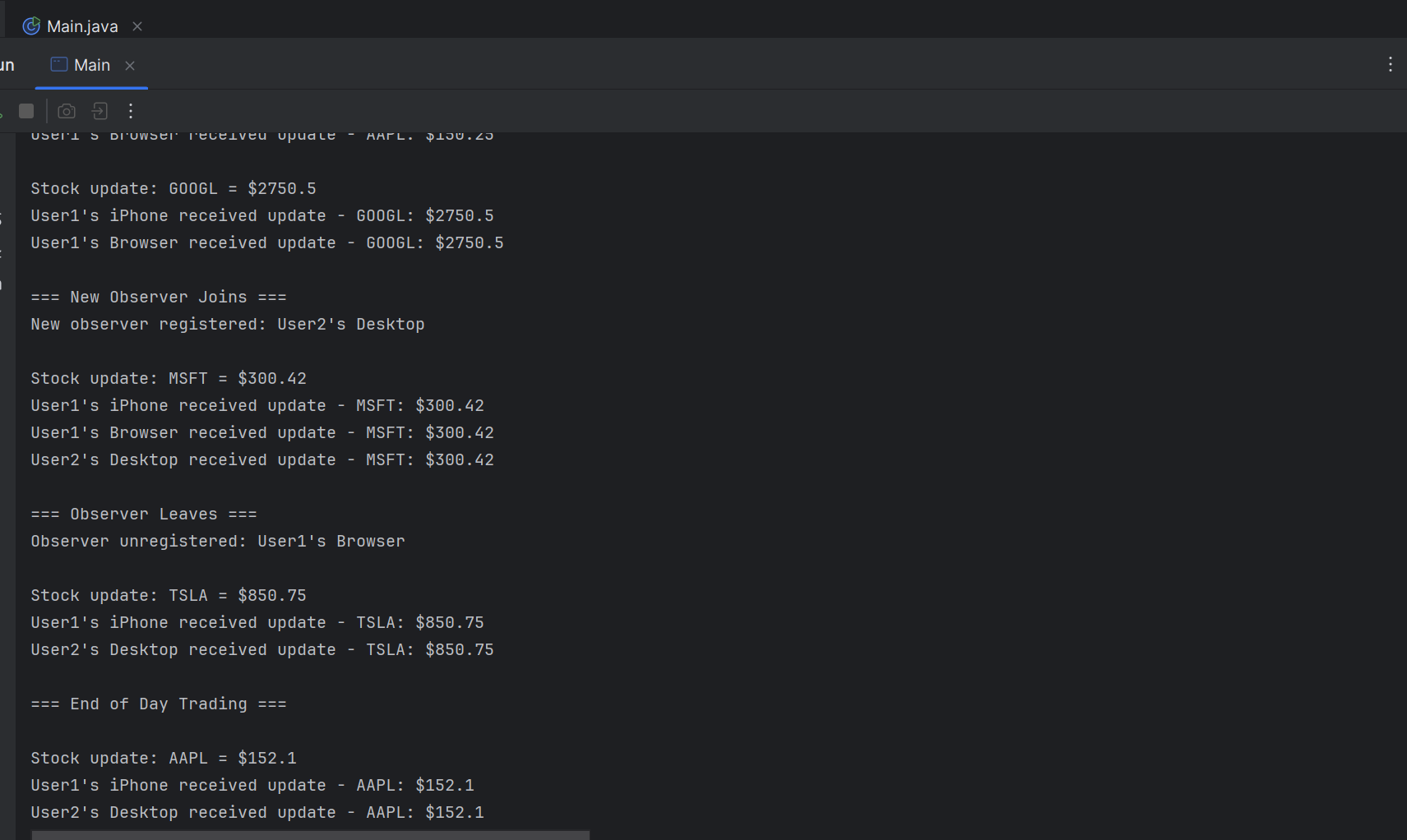
Key Features:

1. **Loose Coupling**: Subjects know nothing about concrete observers
2. **Dynamic Registration**: Observers can join/leave at runtime
3. **Multiple Observers**: Different types can observe the same subject
4. **Push Model**: Subject pushes updates to all observers
5. **Extensible**: Easy to add new observer types

Observer Pattern Benefits:

1. **Decoupling**: Subjects and observers are loosely coupled
2. **Flexibility**: Observers can be added/removed dynamically
3. **Broadcast Communication**: One change notifies multiple observers
4. **Open/Closed Principle**: New observers can be added without changing subject
5. **Event Handling**: Perfect for event-driven systems

**Output**

****