27 Behavioral: Observer Pattern — News-Agency Push Notification Assignment

Build a mini **publish/subscribe** system: a NewsAgency (Subject) pushes headlines to any number of NewsSubscribers (Observers).  
 Include JavaDoc in every class and finish with a brief **Reflection** on benefits & trade-offs.

.

├── analysis

│ └── observer\_need.md ← why pub/sub beats hard-wired callbacks

├── src/main/java

│ └── news

│ ├── subject

│ │ ├── Subject.java

│ │ └── NewsAgency.java

│ ├── observer

│ │ ├── Observer.java

│ │ └── NewsSubscriber.java

│ └── ObserverDemo.java

├── src/test/java/news

│ ├── BroadcastTest.java

│ ├── RemoveObserverTest.java

│ └── NoObserverTest.java

├── reflection.md

└── README.md

#### **1 Subject.java**

package news.subject;

import news.observer.Observer;

/\*\*

\* <p>Observable entity in the classic Observer pattern.</p>

\* Provides operations to manage and notify {@link Observer}s.

\*/

public interface Subject {

/\*\* Registers a new observer. \*/

void addObserver(Observer o);

/\*\* Removes an existing observer. \*/

void removeObserver(Observer o);

/\*\* Notifies all registered observers of a change. \*/

void notifyObservers(String headline);

}

#### **2 NewsAgency.java – concrete subject**

package news.subject;

import news.observer.Observer;

import java.util.ArrayList;

import java.util.List;

/\*\* Concrete Subject that publishes headlines to its subscribers. \*/

public class NewsAgency implements Subject {

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

private String headline = "";

public void setHeadline(String headline) {

this.headline = headline;

notifyObservers(headline);

}

public String getHeadline() { return headline; }

@Override public void addObserver(Observer o) { observers.add(o); }

@Override public void removeObserver(Observer o) { observers.remove(o); }

@Override public void notifyObservers(String h) {

for (Observer o : observers) o.update(h);

}

}

#### **3 Observer.java**

package news.observer;

/\*\*

\* Listener that reacts to news published by a {@code Subject}.

\*/

public interface Observer {

/\*\* Called by the subject when a new headline is available. \*/

void update(String headline);

}

#### **4 NewsSubscriber.java – concrete observer**

package news.observer;

/\*\* Simple console-printing subscriber. \*/

public class NewsSubscriber implements Observer {

private final String name;

public NewsSubscriber(String name){ this.name = name; }

@Override public void update(String headline) {

System.out.println(name + " received: " + headline);

}

}

#### **5 Demo / client**

package news;

import news.subject.NewsAgency;

import news.observer.\*;

/\*\* Shows registration, publication, and dynamic removal. \*/

public class ObserverDemo {

public static void main(String[] args){

NewsAgency agency = new NewsAgency();

Observer alice = new NewsSubscriber("Alice");

Observer bob = new NewsSubscriber("Bob");

Observer chad = new NewsSubscriber("Chad");

agency.addObserver(alice);

agency.addObserver(bob);

agency.addObserver(chad);

agency.setHeadline("Breaking: Observer pattern rocks!");

agency.removeObserver(chad);

agency.setHeadline("Update: Chad unsubscribed.");

}

}

Console

Alice received: Breaking: Observer pattern rocks!

Bob received: Breaking: Observer pattern rocks!

Chad received: Breaking: Observer pattern rocks!

Alice received: Update: Chad unsubscribed.

Bob received: Update: Chad unsubscribed.

#### **6 Tests (outline)**

/\* BroadcastTest.java \*/

NewsAgency a = new NewsAgency();

StringBuilder log = new StringBuilder();

Observer s = log::append; // lambda observer

a.addObserver(s);

a.setHeadline("X");

assertEquals("X", log.toString());

/\* RemoveObserverTest.java \*/

a.removeObserver(s);

a.setHeadline("Y");

assertEquals("X", log.toString()); // unchanged

/\* NoObserverTest.java \*/

NewsAgency empty = new NewsAgency();

empty.setHeadline("None"); // should not throw

## **reflection.md**

**Observer** delivers *loose coupling*: subjects know nothing about subscribers beyond the Observer interface.  
 *Pros*

* Add or remove listeners at runtime → flexible event flow.
* Subject and observers evolve independently (Open/Closed).
* Suits GUI toolkits, messaging back-ends, micro-service events.

*Cons*

* **Notification storm** if many observers; consider filters / async queues.
* Ordering is not guaranteed—critical if dependencies exist.
* Risk of **memory leaks** when observers forget to unsubscribe.

For event-driven apps such as news feeds, stock tickers, or GUI widgets, Observer is lightweight yet powerful—just keep an eye on lifetime management and broadcast volume.