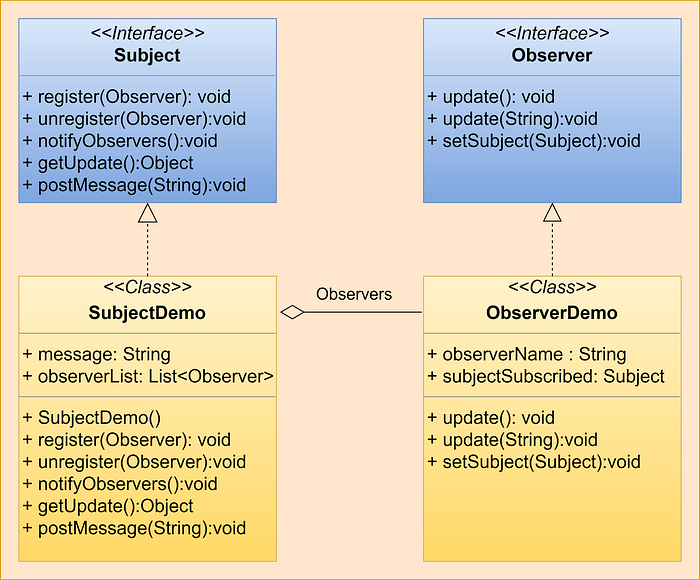
**Observer Design Pattern in Java**

A behavioral design pattern for Flexible Notification-Registration Interaction.



Observer design pattern architecture

In this article, we will see what is the intent of the Observer design pattern, what problems it solves, and its applicability.

Before we move to the technicality of this topic, let me give one of the most basic use cases of this design pattern. We mostly tend to subscribe to the YouTube channels that we love watching. So in this example, the channel is an example of a Subject and the users subscribing to it are Observers. Whenever there is an update on the channel all the subscribed users get notified.

**The Intent of the Observer Design Pattern Is To:**

***“Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”***

**The Observer Design Pattern Solves Problems Like:**

* How can a one-to-many dependency between objects be defined without making the objects tightly coupled?
* How to create a model to update all dependent objects when the state of one object changes?

**A Bit About Tight and Loose Coupling:**

* Coupling is the degree to which objects depend on each other.
* Tightly coupled objects are hard to implement, change, test, and reuse because they depend on (refer to and know about) many different objects (having different interfaces).
* Loosely coupled objects are easier to implement, change, test, and reuse because they have only minimal dependencies on other objects and they use a common interface.

**How Does the Observer Design Pattern Solve the Problem:**

1. It defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
2. The key idea in this pattern is to establish a flexible Notification-Registration mechanism

**An Unusual Technique To Solve the Problem:**

Check the code snippet below:

class Subject {  
 String message;  
 Observer1 observer1;  
 Observer2 observer2;  
 public Subject(Observer1 observer1, Observer2 observer2) {   
 this.observer1 = observer1;  
 this.observer2 = observer2;  
 }  
}

* This is an inflexible way to define a one-to-many dependency between objects. Here we have defined one object (**Subject**) that updates the dependent objects (**Observer1**, **Observer2**,**…**).
* This leads to the **Subject** to particularly depend on different **Observers**and makes it impossible to change the **Subject**(to add new observers or remove existing observers) independently (without having to change) the subject. It stops the subject from being reusable and makes the subject hard to test. The **Subject** needs to be updated for every addition or removal of **Observers** .
* That’s the kind of approach we want to avoid if we want to keep the objects in a one-to-many dependency loosely coupled.

**A Better Approach:**

The key idea in this pattern is to establish a flexible notification-registration interaction by notifying (calling update on) all registered observers automatically when an event of interest occurs.

• Define Subject and Observer objects:

**– Subject** : defines an interface for registering and unregistering observers (**register(observer o)**, **unregister(observer o)**) and for notifying observers (**notifyObservers()**), i.e calling **update()** on all registered observers.

**– Observer** : defines an interface for updating the state (**update()**), i.e., synchronizing the observer’s state with the subject’s state.

• When a subject changes state, all registered observers are notified and updated automatically (for each o in observers: o.update()).

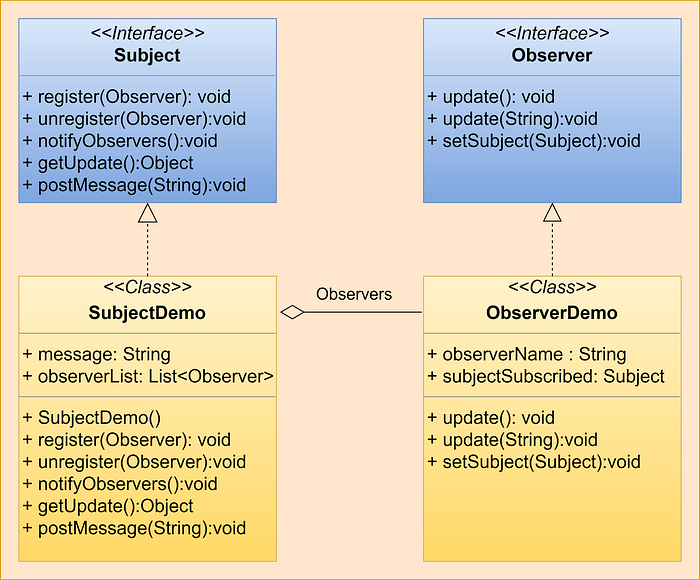
• This enables loose coupling between the subject and observers.

*Subject and observers have no explicit knowledge of each other. An open-ended number of observers can observe a subject. New observers can be added to and existing ones can be removed from the subject independently and dynamically.*

**When To Use It?**

We should consider using this pattern in our application whenever multiple objects are dependent on the state of one object as it provides a neat and well-tested design for the same.

**Observer Design Pattern Architecture:**

[[](https://javarevisited.blogspot.com/2018/02/top-5-java-design-pattern-courses-for-developers.html)](https://javarevisited.blogspot.com/2018/02/top-5-java-design-pattern-courses-for-developers.html)

Observer design pattern architecture

**Observer Design Pattern Example:**

**Subject Interface:**an interface to provide basic methods for implementing Subject classes.

Subject interface

**Observer Interface:**an interface to provide basic methods for implementing Observer classes.

Observer interface

**Subject Interface Implementation:**A subject class to update all its observers when the state of the subject changes.

Subject Interface implementation

**Observer Interface Implementation:**Observer class to receive updates from the subject once its state changes.

Observer interface implementation

**Driver Class To Test Observer Design Pattern:**

driver class for observer design pattern

**Output:**

Observer-1 No new message received!!!  
Message posted on topic : Hello Java — message posted by Vikram  
Observer-1 New message received: Hello Java — message posted by Vikram  
Observer-2 New message received: Hello Java — message posted by Vikram  
Observer-3 New message received: Hello Java — message posted by Vikram  
Observer-4 New message received: Hello Java — message posted by Vikram  
Observer-5 New message received: Hello Java — message posted by Vikram

**Practical Examples:**

* It is heavily used in GUI toolkits and event listeners. In Java, the button(subject) and onClickListener(observer) are modeled with the observer pattern.
* Social media, RSS feeds, and email subscriptions allow you to follow or subscribe and receive the latest notification.
* All users of an app on the Play Store get notified if there is an update.