**Definition:**

Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change. In observer pattern, the object that watch on the state of another object are called Observer and the object that is being watched is called Subject.

Observer pattern falls under behavioral pattern category.

**Subject** contains a list of observers to notify of any change in it’s state, so it should provide methods using which observers can register and unregister themselves.

The Observer Pattern defines a one to many dependency between objects so that one object changes state, all of its dependents are notified and updated automatically.

The observer pattern is also known as Dependents or Publish-Subscribe.

**Explanation:**

* One to many dependency is between Subject(One) and Observer(Many).
* There is dependency as Observers themselves don’t have access to data. They are dependent on Subject to provide them data.

**Benefits:**

* It describes the coupling between the objects and the observer.
* It provides the support for broadcast-type communication.

## Usage:

* When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
* When the framework we writes and needs to be enhanced in future with new observers with minimal changes.

**Example:**

Imagine that you have two types of objects: a Customer and a Store. The customer is very interested in a particular brand of product (say, it’s a new model of the iPhone) which should become available in the store very soon.

The customer could visit the store every day and check product availability. But while the product is still in route, most of these trips would be pointless.

On the other hand, the store could send tons of emails (which might be considered spam) to all customers each time a new product becomes available. This would save some customers from endless trips to the store. At the same time, it’d upset other customers who aren’t interested in new products.

It looks like we’ve got a conflict. Either the customer wastes time checking product availability or the store wastes resources notifying the wrong customers.

##  Solution

The Observer pattern suggests that you add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.

## UML for Observer Pattern:

****

**Implementation:**

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. We have created an abstract class *Observer* and a concrete class *Subject* that is extending class *Observer*.

*ObserverPatternDemo*, our demo class, will use *Subject* and concrete class object to show observer pattern in action.



## Step 1

Create Subject class.

*Subject.java*

import java.util.ArrayList;

import java.util.List;

public class Subject {

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

 private int state;

 public int getState() {

 return state;

 }

 public void setState(int state) {

 this.state = state;

 notifyAllObservers();

 }

 public void attach(Observer observer){

 observers.add(observer);

 }

 public void notifyAllObservers(){

 for (Observer observer : observers) {

 observer.update();

 }

 }

}

Step 2

Create Observer class.

*Observer.java*

public abstract class Observer {

 protected Subject subject;

 public abstract void update();

}

Step 3

Create concrete observer classes

*BinaryObserver.java*

public class BinaryObserver extends Observer{

 public BinaryObserver(Subject subject){

 this.subject = subject;

 this.subject.attach(this);

 }

 @Override

 public void update() {

 System.out.println( "Binary String: " + Integer.toBinaryString( subject.getState() ) );

 }

}

*OctalObserver.java*

public class OctalObserver extends Observer{

 public OctalObserver(Subject subject){

 this.subject = subject;

 this.subject.attach(this);

 }

 @Override

 public void update() {

 System.out.println( "Octal String: " + Integer.toOctalString( subject.getState() ) );

 }

}

*HexaObserver.java*

public class HexaObserver extends Observer{

 public HexaObserver(Subject subject){

 this.subject = subject;

 this.subject.attach(this);

 }

 @Override

 public void update() {

 System.out.println( "Hex String: " + Integer.toHexString( subject.getState() ).toUpperCase() );

 }

}

Step 4

Use *Subject* and concrete observer objects.

*ObserverPatternDemo.java*

public class ObserverPatternDemo {

 public static void main(String[] args) {

 Subject subject = new Subject();

 new HexaObserver(subject);

 new OctalObserver(subject);

 new BinaryObserver(subject);

 System.out.println("First state change: 15");

 subject.setState(15);

 System.out.println("Second state change: 10");

 subject.setState(10);

 }

}

Step 5

Verify the output.

First state change: 15

Hex String: F

Octal String: 17

Binary String: 1111

Second state change: 10

Hex String: A

Octal String: 12

Binary String: 1010