In observer design pattern multiple observer objects registers with a subject for change notification. When the state of subject changes, it notifies the observers. Objects that listen or watch for change are called observers and the object that is being watched for is called subject.

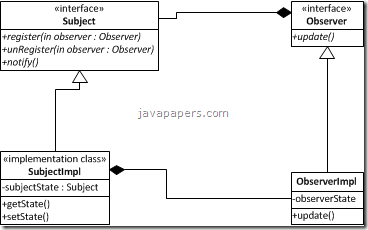
Pattern involved is also called as publish-subscribe pattern.

## Important Points on Observer Design Pattern

* Subject provides interface for observers to register and unregister themselves with the subject.
* Subject knows who its subscribers are.
* Multiple observers can subscribe for notifications.
* Subject publishes the notifications.
* Subject just sends the notification saying the state has changed. It does not pass any state information.
* Once the notification is received from subject, observers call the subject and get data that is changed.

The above last two points are not strictly followed in observer design pattern implementation. Along with the notification, state is also passed in some implementation so that the observer need not query back to know the status. It is better not to do this way.

**Observer pattern UML**



## Scenario for Observer Pattern Usage

* When multiple objects depend on state of one object and all these multiple objects should be in sync with the state of this one object then observer design pattern is the right choice to apply.
* Consider an excel sheet, data is shown to the user in different views. Generally data is is shown in grid cells and as required different graphs, charts can be created for same data. Underlying data is same and when that data (subject) state changes all the different view are updated.

## Observer and Observable Java API in jdk

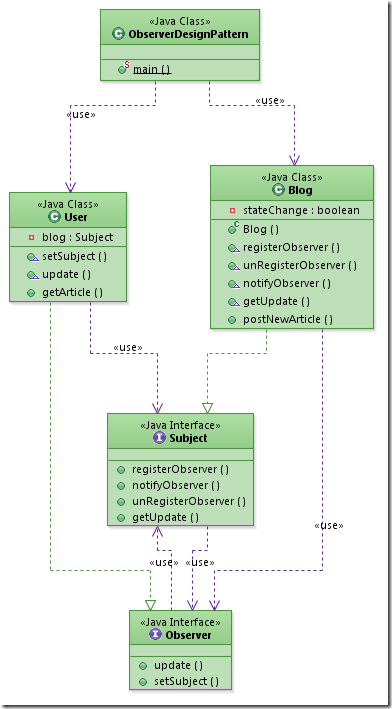
JDK provides Observer and Observable classes as part of util package. This is not something new and these classes were available since JDK 1.0. Observer is an interface which needs to be implemented to observe the state change in a observable subject. Observable is a class which should be extended by a subject. Observable provides implementation for methods to register or unregister an Observer and to notify the Observer objects.

Using these APIs, Observable and Observer in real time application may not be possible most of time. As we all know, [java does not support multiple inheritance](https://javapapers.com/core-java/why-multiple-inheritance-is-not-supported-in-java/). I can extend only one class and I do not want to exhaust that option with this Observable class. In a real time application I may need that option to be used for business inheritance.

Prefer composition over inheritance is a golden rule. So I thought of using Observable in composition than inheritance. That option is also ruled out because *setChanged()* method has access specified as *protected*. So as per [Java access speficier](https://javapapers.com/core-java/access-modifiers-in-java-explain/) rules, protected attributes cannot be accessed from outside the package. So the only option left is to do a custom implementation for Observer pattern. It is not a tedious one.

## Example Observer Design Pattern

Let us take a blog and subscriber example for observer design pattern sample implementation. Assume that there is a blog and users register to that blog for update. When a new article is posted in the blog, it will send update to the registered users saying a new article is posted. Then the user will access the blog and read the new article posted. In this example, blog is the subject and user is the observer.



/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.util.ArrayList;

import java.util.List;

/\*

\* Observer.java

\*/

interface Observer{

public void update();

public void setSubject(Subject subject);

}

/\*

\* User.java

\*/

class User implements Observer{

private String article;

private Subject blog;

public void setSubject(Subject blog) {

this.blog = blog;

article = "No new Article!!";

}

public void update() {

System.out.println("State change reported by subject.");

}

public String getArticle() {

return article;

}

}

/\*

\* Subject.java

\*/

interface Subject{

public void registerObserver(Observer observer);

public void notifyObserver();

public void unRegisterObserver(Observer observer);

public Object getUpdate();

}

/\*

\* Blog.java

\*/

class Blog implements Subject{

List<Observer> observerList;

private boolean stateChange;

public Blog() {

this.observerList = new ArrayList();

stateChange = false;

}

public void registerObserver(Observer observer) {

observerList.add(observer);

}

public void unRegisterObserver(Observer observer) {

observerList.add(observer);

}

public void notifyObserver() {

if(stateChange) {

for(Observer observer : observerList) {

observer.update();

}

}

}

public Object getUpdate() {

Object changedState = null;

//should have logic to send the state change to querying observer

if(stateChange) {

changedState = "Observer Design Pattern";

}

return changedState;

}

public void postNewArticle() {

stateChange = true;

notifyObserver();

}

}

public class TestObserver {

public static void main(String[] args) {

Blog blog = new Blog();

User user1 = new User();

User user2 = new User();

blog.registerObserver(user1);

blog.registerObserver(user2);

user1.setSubject(blog);

user2.setSubject(blog);

System.out.println(user1.getArticle());

blog.postNewArticle();

System.out.println(user1.getArticle());

}

}