Observer Design pattern in java

*Observer design pattern in Java* is very important pattern and as name suggest it’s used to observe things. Suppose you want to notify for change in a particular object than you observer that object and changes are notified to you. Object which is being observed is refereed as Subject and classes which observe subject are called Observer**. This is beautiful pattern and used heavily along with Model View Controller Design pattern where change in model is propagated to view so that it can render it with modified information.** Observer pattern is also a very [popular Java interview questions](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html) and mostly asked on Senior or mid senior level.

For implementation of this pattern java makes our task very easy, developer need not to do so much for implementing this pattern .In **java.util**package we can find interfaces ,classes and methods for implementing this pattern.

**Public Interface Observer:**

Any class who implements this interface must be notified when subject or observable object change its status.

**Update (Observable Ob, Object arg): This method is called when subject is changed.**

**Class Observable:**

It’s a subject to whom observer wants to observe.

**Some Important Method:**

addObserver(Observer o):add Observers in the set of observers for this subject or observalbel object.

deleteObserver(Observer o): delete Observers in the set of observers .

hasChanged():check if object has changed.

clearChanged():this method will indicate that subject has no changes or all the observers has been notified when changes is made.

notifyObservers(): notify all the observers if object has changed .

Read more: <http://javarevisited.blogspot.com/2011/12/observer-design-pattern-java-example.html#ixzz36RrjEpdl>

### Code Example of Observer design pattern in Java:

**Observer Design pattern** is generic than how it is implemented in Java. You are free to choose java.util.Observable or java.util.Observer or *writing your own Subject and Observer interface*. I prefer having my own Subject and Observer interface and this is how I am going to write my Code Example of Observer design Pattern in java. Mine Example is very Simple you have a Loan on which interest rate is subject to change and when it changes, Loan notifies to Newspaper or Internet media to display new loan interest rate. To implement this we have a **Subject** interface which contains methods for adding, removing and notifying Observers and an **Observer** interface which contains update(int interest) method which will be called by Subject implementation when interest rate changes.

**import** java.util.ArrayList;

**interface** Observer {

**public** **void** update(**float** interest);

}

**interface** Subject {

**public** **void** registerObserver(Observer observer);

**public** **void** removeObserver(Observer observer);

**public** **void** notifyObservers();

}

**class** Loan **implements** Subject {

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

**private** String type;

**private** **float** interest;

**private** String bank;

**public** Loan(String type, **float** interest, String bank) {

**this**.type = type;

**this**.interest = interest;

**this**.bank = bank;

       }

**public** **float** getInterest() {

**return** interest;

       }

**public** **void** setInterest(**float** interest) {

**this**.interest = interest;

              notifyObservers();

       }

**public** String getBank() {

**return** **this**.bank;

       }

**public** String getType() {

**return** **this**.type;

       }

       @Override

**public** **void** registerObserver(Observer observer) {

              observers.add(observer);

       }

       @Override

**public** **void** removeObserver(Observer observer) {

              observers.remove(observer);

       }

       @Override

**public** **void** notifyObservers() {

**for** (Observer ob : observers) {

                     System.*out*

                                  .println("Notifying Observers on change in Loan interest rate");

                     ob.update(**this**.interest);

              }

       }

}

**class** Newspaper **implements** Observer {

       @Override

**public** **void** update(**float** interest) {

              System.*out*.println("Newspaper: Interest Rate updated, new Rate is: "

                           + interest);

       }

}

**class** Internet **implements** Observer {

       @Override

**public** **void** update(**float** interest) {

              System.*out*.println("Internet: Interest Rate updated, new Rate is: "

                           + interest);

       }

}

**public** **class** ObserverTest {

**public** **static** **void** main(String args[]) {

              // this will maintain all loans information

              Newspaper printMedia = **new** Newspaper();

              Internet onlineMedia = **new** Internet();

              Loan personalLoan = **new** Loan("Personal Loan", 12.5f,

                           "Standard Charterd");

              personalLoan.registerObserver(printMedia);

              personalLoan.registerObserver(onlineMedia);

              personalLoan.setInterest(3.5f);

       }

}

**Advantage of Observer Design Pattern in Java:**

Main advantage is **loose coupling** between objects called observer and observable. The subject only know the list of observers it don’t care about how they have their implementation. All the observers are notified by subject in a single event call as **Broadcast communication**.

**Disadvantage of Observer Design Pattern in Java:**

          The disadvantage is that the sometime if any problem comes, [debugging](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html) becomes very difficult because flow of control is implicitly between **observers** and **observable** we can predict that now observer is going to fire and if there is chain between observers then debugging become more complex.

          Another issue is Memory management because subject will hold all the reference of all the observers if we not unregister the object it can create the memory issue.

That’s all on **Observer Pattern in Java**. Share your thought how and when you have used Observer Pattern in your Project and any issue you have faced?

Read more: <http://javarevisited.blogspot.com/2011/12/observer-design-pattern-java-example.html#ixzz39e6j8oBh>