Observer Design Pattern

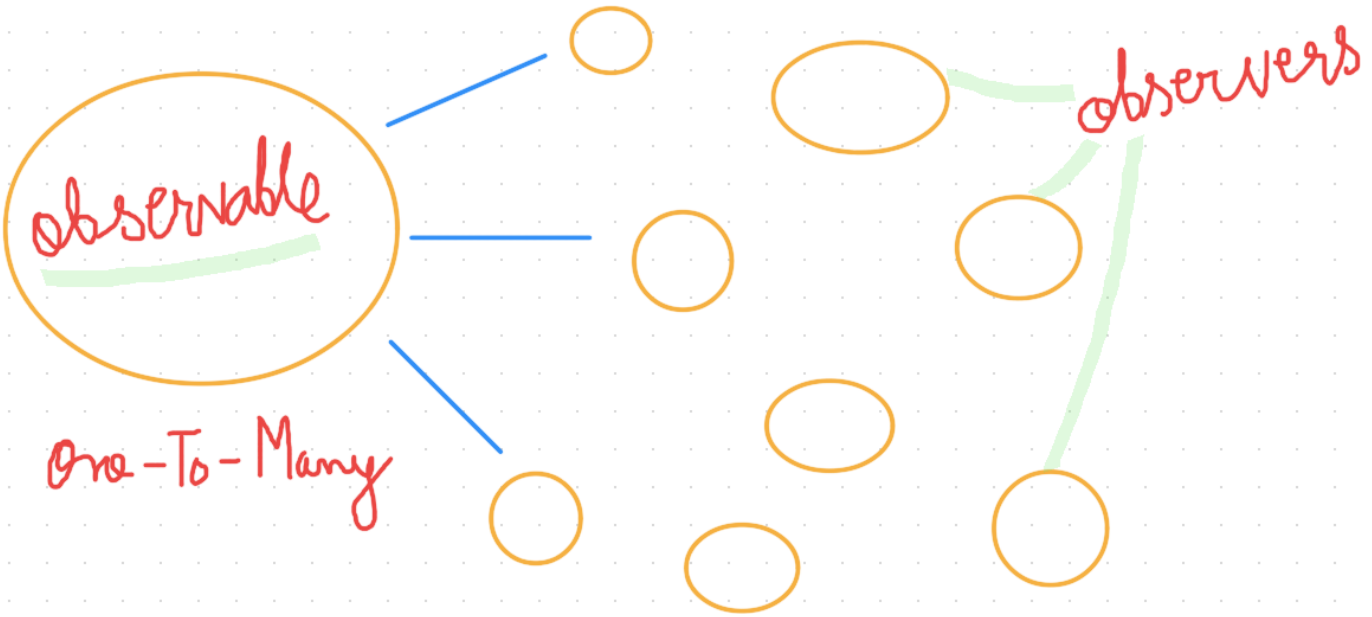
Notes by Bhavuk Jain

Definition: The **Observer pattern** is a design pattern in which an object, called as an ***observable***, maintains a list of its dependents, called ***observers***, and **notifies** them automatically of any state changes, usually by calling one of their methods.

Example: A popular use-case of this pattern is the **Notify Me** feature that is present on E-Commerce websites. If a user wants to buy something that is currently *out of stock*, he can use **Notify Me** feature so that he is **automatically informed** whenever that particular item is *back in stock*!

Core Concept

There is an ***observable*** (something that is being observed by others) and there are many ***observers*** (who observe the observable continuously). There can be **n** number of observers who are interested in knowing some kind of information that the observable holds. Imagine it like this:-



WHAT & WHY?

Taking our previous **E-commerce example**, let's say you want to buy an **Iphone**. You go to an E-commerce website and there you find that Iphone is currently ***out of stock***. So, in order to get notified whenever it comes back in stock, you press the **Notify Me** button. Like you, there are **thousands of other users** who are **interested to buy** the Iphone and they too, click the *Notify Me* button.

In this case, you are acting as an **Observer** who is observing the status of availability of Iphone. Similarly, Iphone is acting as an **Observable** (because one or many users are observing it)

When you click on the **Notify Me** button, you get added to the **interested users list** maintained by the Iphone seller. Now, whenever the **Iphone is back in stock**, all the

interested users (**observers**) will be **automatically *notified***about it.

PULL v/s PUSH

As, you can notice here, once you have **subscribed** to the **notification service**, you don't have to check the availability of Iphone again and again because you know, you will be automatically notified via email or push notifications whenever Iphone is back in stock.

Now, let's say , there was **no notification service**, then you will have to check the availability status of Iphone **again and again** by visiting that E-commerce website. This is called a **PULL** based seeking of information because the user has to ***PULL*** (make efforts) to know the status.

On, the other hand, **Notification service** is facilitating **PUSH** based mechanism because it will automatically ***PUSH*** notifications/information to all the users (**observers**) whenever Iphone is back in stock. The ***user doesn't need to PULL*** (make effort) again and again.

Code Snippets

(Tip: Read through the code comments to get a better perspective)

STEPS:

1.) We create an **Observable Interface**, called **IObservable** which would have the abstract methods as below:

2.) We create an **Observer Interface**, called **IObserver**, which would have the abstract method, as shown below:

IObserver.java

```java
package observerpattern;

public interface IObserver {

    // Observer will update itself when notified
    public void update();
}

```

3.) To provide implementation of the IObservable interface, we will create a class, called **IphoneObservable** which will provide concrete definition to the abstract methods of **IObservable** interface.

IphoneObservable.java

```java
package observerpattern;

import java.util.*;

public class IphoneObservable implements IObservable{
    private int stockCount = 0;    // Initial Stock Count
    private List<IObserver> list = new ArrayList<>();    // List to store the observers

    // Add a new observer to the list
    public void add(IObserver observer) { list.add(observer); }

    // Remove an observer from the list
    public void remove(IObserver observer) { list.remove(observer); }

    /* Set the new stock count and notify the observers
     if previously Iphone was out of stock */
    public void setStockCount(int stockCount){
        if(this.stockCount == 0){
            this.notifyObservers();
        }

        this.stockCount=stockCount;
    }


    // Returns the current stock count
    public int getStockCount() { return this.stockCount; }

    /* Iterate through the list of observers and notify them
    about the change in stock */
    public void notifyObservers(){
        for(IObserver observer : list){
            observer.update();
        }
    }
}
```

4.) To provide implementation of the IObserver interface, we will create a class, called **User** which will provide concrete definition to the abstract method of **IObserver** interface.

User.java

```
1  package observerpattern;
2
3  public class User implements IObserver{
4      // Reference variable for observable
5      IObservable observable;
6
7      // Assigning the observable object through constructor injection
8      public User(IObservable observable){
9          this.observable = observable;
10     }
11
12     // Perform some operations when the user is notified
13     public void update(){
14         System.out.println("User is notified that iphone is back in stock");
15     }
16 }
17
```

5.) Lastly, to test our implementation, we will **create an Iphone object** which is acting as an **Observable** and we will **create 3 users** who will act as **observers**.
We will then add these 3 users to the notification list of Iphone and then we will set the stock count of Iphone.

Main.java

```java
package observerpattern;

public class Main {
    public static void main(String[] args) {
        // Create a new observable
        IphoneObservable iphone = new IphoneObservable();

        // Creating new Observers
        User user1 = new User(iphone);
        User user2= new User(iphone);
        User user3 = new User(iphone);

        // Adding observers to the notification list
        iphone.add(user1);
        iphone.add(user2);
        iphone.add(user3);

        // Updating stock count for Iphone
        iphone.setStockCount(10);
        iphone.setStockCount(0);
        iphone.setStockCount(20);
    }
}

```

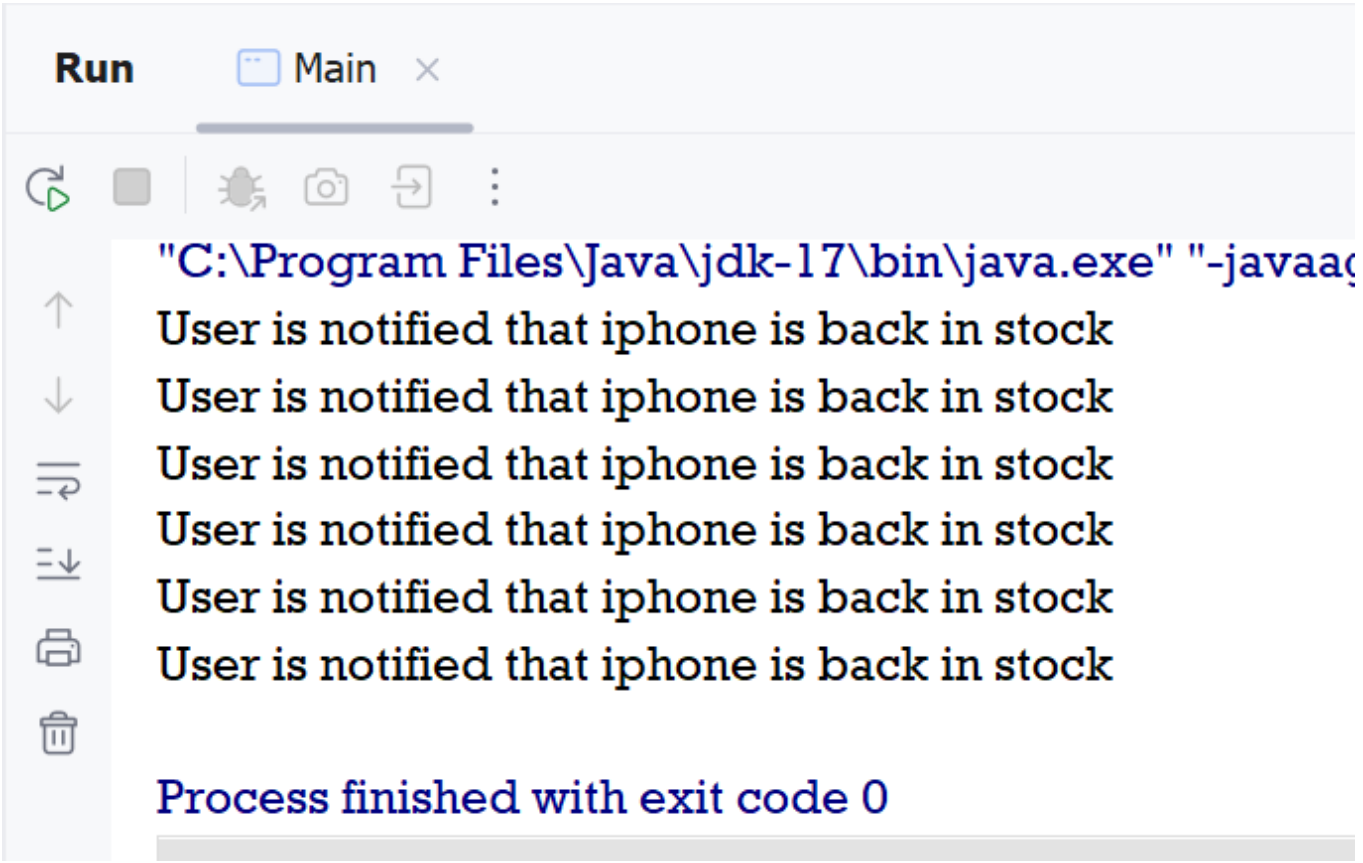
Predicting the output

a) As the **initial stock count is 0**, so setting the **stock count to 10**, **will notify all the users** about the availability of Iphone.

b) After that, we will again **set stock count to 0** which **will not notify** the users.

c) Lastly, we will **again set the stock count to 20**, which will **notify the users** again about the availability of Iphone.
So, in total, the **users will be notified twice** about the availability of Iphone when it is back in stock. And, as we have **3** users in total, **all of them are notified twice**. (That's why, we'll get a **total of 6 lines** of output)

Output:



Hope you find it helpful. Thanks for reading!

Signing off,
Bhavuk Jain