CHAPTER 2

Observer Patterns

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

Concept

In this pattern, there are many observers (objects) which are observing a particular subject (object). Observers are basically interested and want to be notified when there is a change made inside that subject. So, they register themselves to that subject. When they lose interest in the subject they simply unregister from the subject. Sometimes this model is also referred to as the Publisher-Subscriber model.

Real-Life Example

We can think about a celebrity who has many fans. Each of these fans wants to get all the latest updates of his/her favorite celebrity. So, he/she can follow the celebrity as long as his/her interest persists. When he loses interest, he simply stops following that celebrity. Here we can think of the fan as an observer and the celebrity as a subject.

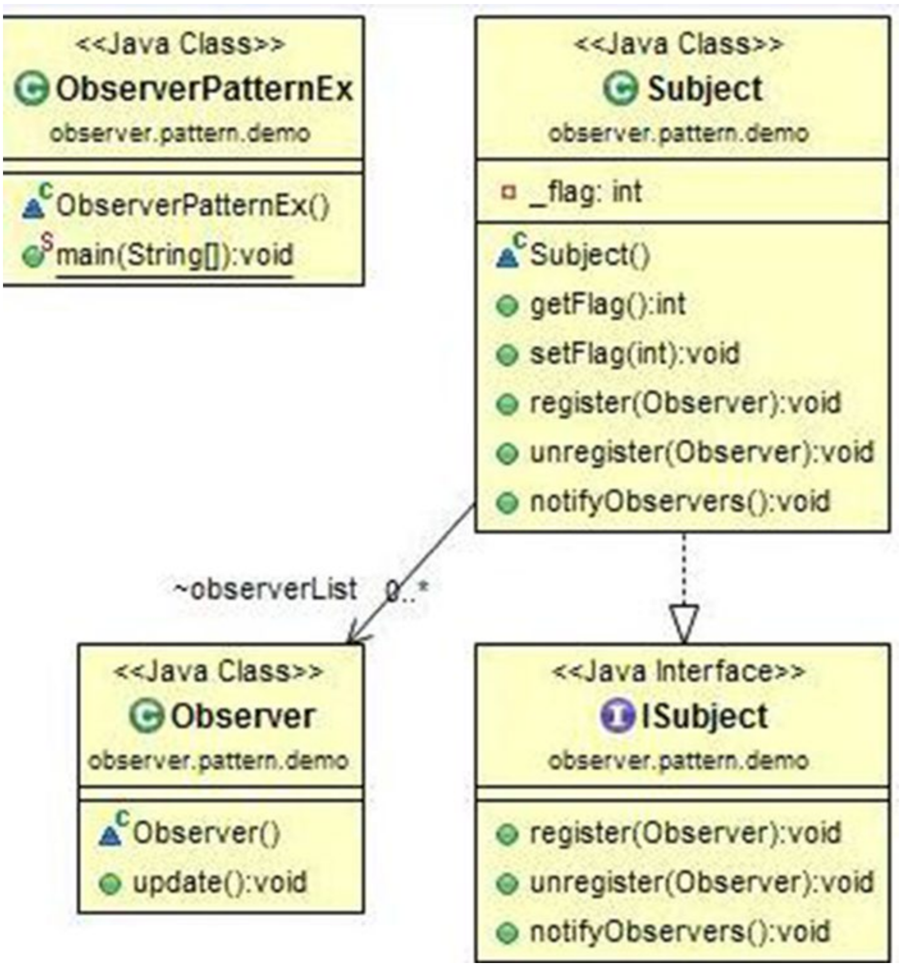
Computer World Example

In the world of computer science, consider a simple UI-based example, where this UI is connected with some database (or business logic). A user can execute some query through that UI and after searching the database, the result is reflected back in the UI. In most of the cases we segregate the UI with the database. If a change occurs in the database, the UI should be notified so that it can update its display according to the change.

Illustration

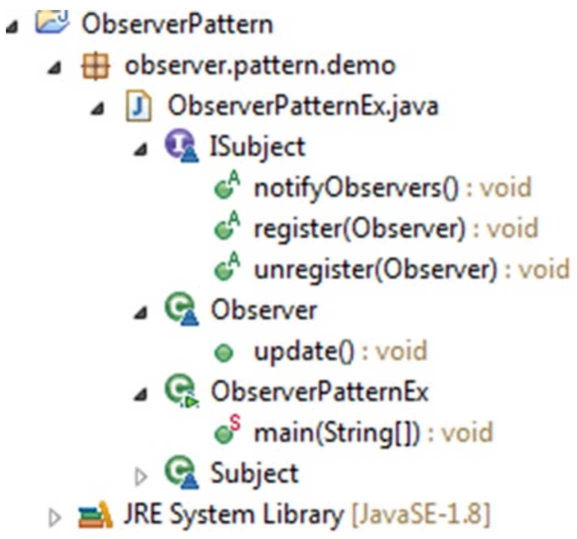
Now let us directly enter into our simple example. Here I have created one observer (though you can create more) and one subject. The subject maintains a list for all of its observers (though here we have only one for simplicity). Our observer here wants to be notified when the flag value changes in the subject. With the output, you will discover that the observer is getting the notifications when the flag value changed to 5 or 25. But there is no notification when the flag value changed to 50 because by this time the observer has unregistered himself from the subject.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:

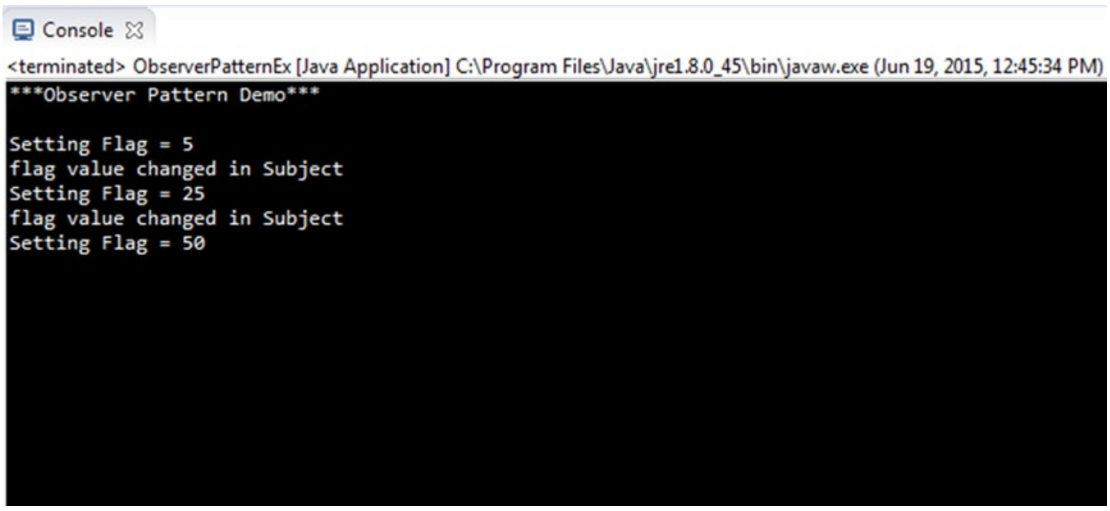


```
ObserverPattern
  observer.pattern.demo
    ObserverPatternEx.java
      ISubject
        notifyObservers() : void
        register(Observer) : void
        unregister(Observer) : void
      Observer
        update() : void
      ObserverPatternEx
        main(String[]) : void
      Subject
  JRE System Library [JavaSE-1.8]
```

Implementation

```
package observer.pattern.demo;
import java.util.*;

class Observer
    {
        public void update()
        {
                System.out.println("flag value changed in Subject");
        }
    }

    interface ISubject
    {
        void register(Observer o);
        void unregister( Observer o);
        void notifyObservers();
    }
    class Subject implements ISubject
    {
        List<Observer> observerList = new ArrayList<Observer>();
        private int _flag;
        public int getFlag()
        {
           return _flag;
        }
        public void setFlag(int _flag)
        {
          this._flag=_flag;
          //flag value changed .So notify observer(s)
                 notifyObservers();
        }
        @Override
        public void register(Observer o)
        {
            observerList.add(o);
        }
        @Override
        public void unregister(Observer o)
        {
            observerList.remove(o);
        }
        @Override
        public void notifyObservers()
        {
           for(int i=0;i<observerList.size();i++)
                {
                    observerList.get(i).update();
                }
        }
    }
class  ObserverPatternEx
{
        public static void main(String[] args)
    {
                System.out.println("***Observer Pattern Demo***\n");
        Observer o1 = new Observer();
        Subject sub1 = new Subject();
        sub1.register(o1);
        System.out.println("Setting Flag = 5 ");
        sub1.setFlag(5);
        System.out.println("Setting Flag = 25 ");
        sub1.setFlag(25);
        sub1.unregister(o1);
        //No notification this time to o1 .Since it is unregistered.
        System.out.println("Setting Flag = 50 ");
        sub1.setFlag(50);
    }
}
```

Output



```
<terminated> ObserverPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Jun 19, 2015, 12:45:34 PM)
***Observer Pattern Demo***

Setting Flag = 5
flag value changed in Subject
Setting Flag = 25
flag value changed in Subject
Setting Flag = 50
```

Note

The above example is one of the simple illustrations for this pattern. Let us consider a relatively complex problem. Let us assume the following:

1. *Now we need to have a multiple observer class.*
2. *And we also want to know about the exact change in the subject. If you notice our earlier implementation, you can easily understand that there we are getting some kind of notification but our observer does not know about the changed value in the subject*

Obviously now we need to make some change in the above implementation. Note that now we cannot use concrete observers like the following:

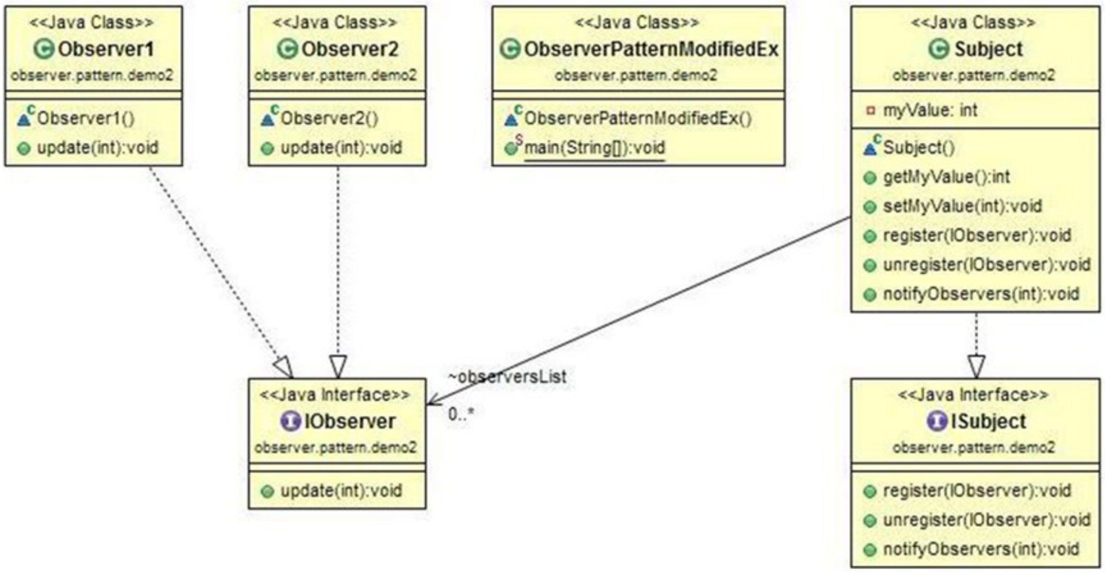
```
List<Observer> observerList = new ArrayList<Observer>();
```

Instead we need to use :

```
List<IObserver> observersList=new ArrayList<IObserver>();
```

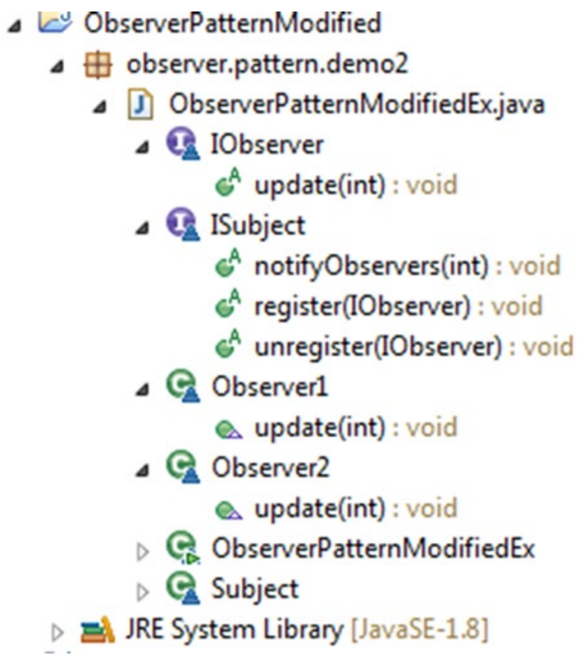
And so we need to replace Observer with IObserver in corresponding places also. We have also modified our update function to see the changed values in the Observers themselves.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the modified program is as follows:

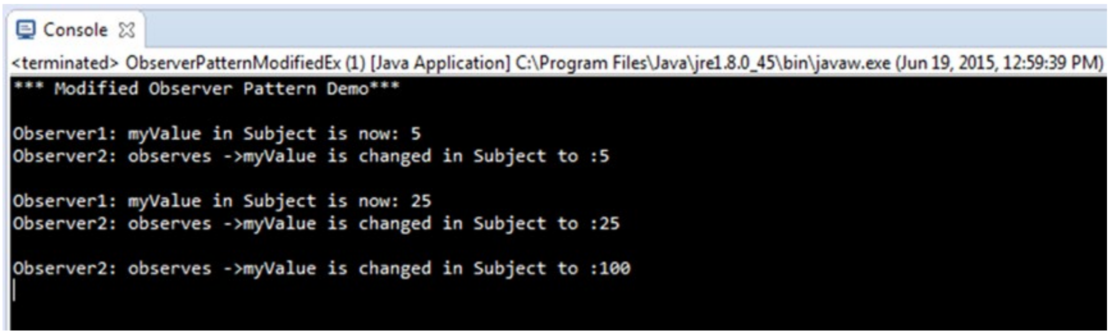


```
ObserverPatternModified
    observer.pattern.demo2
        ObserverPatternModifiedEx.java
            IObserver
                update(int) : void
            ISubject
                notifyObservers(int) : void
                register(IObserver) : void
                unregister(IObserver) : void
            Observer1
                update(int) : void
            Observer2
                update(int) : void
            ObserverPatternModifiedEx
            Subject
    JRE System Library [JavaSE-1.8]
```

Implementation

```
package observer.pattern.demo2;
import java.util.*;

    interface IObserver
    {
        void update(int i);
    }
    class Observer1 implements IObserver
    {
        @Override
                public void update(int i)
                {
                System.out.println("Observer1: myValue in Subject is now: "+i);
                }
    }
    class Observer2 implements IObserver
    {
        @Override
                public void update(int i)
                {
                        System.out.println("Observer2: observes ->myValue is changed in
                        Subject to :"+i);
                }
    }

    interface ISubject
    {
        void register(IObserver o);
        void unregister(IObserver o);
        void notifyObservers(int i);
    }

    class Subject implements ISubject
    {
        private int myValue;

        public int getMyValue() {
                        return myValue;
                }

                public void setMyValue(int myValue) {
                        this.myValue = myValue;
                         //Notify observers
                        notifyObservers(myValue);
                }
                List<IObserver> observersList=new ArrayList<IObserver>();

                @Override
        public void register(IObserver o)
        {
                observersList.add(o);
        }
                @Override
        public void unregister(IObserver o)
        {
                observersList.remove(o);
        }
        @Override
        public void notifyObservers(int updatedValue)
        {
           for(int i=0;i<observersList.size();i++)
                {
                        observersList.get(i).update(updatedValue);
                }
        }
    }

    class ObserverPatternModifiedEx
    {
        public static void main(String[] args)
        {
                System.out.println("*** Modified Observer Pattern Demo***\n");
            Subject sub = new Subject();
            Observer1 ob1 = new Observer1();
            Observer2 ob2 = new Observer2();

            sub.register(ob1);
            sub.register(ob2);

            sub.setMyValue(5);
            System.out.println();
            sub.setMyValue(25);
            System.out.println();

            //unregister ob1 only
            sub.unregister(ob1);
            //Now only ob2 will observe the change
            sub.setMyValue(100);
        }
    }
```

Output



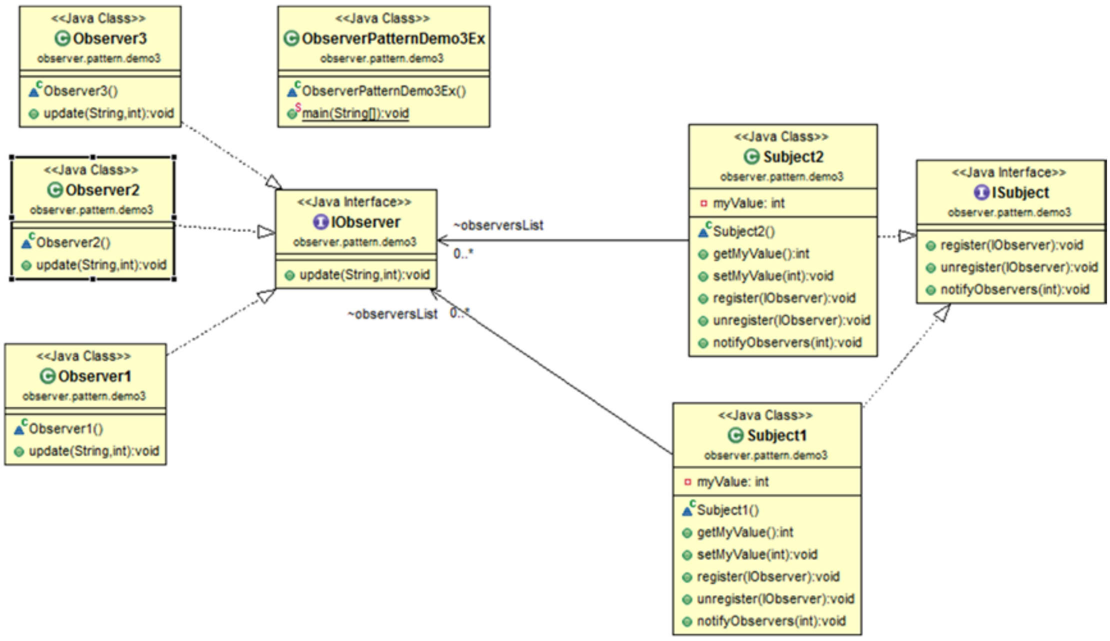
```
<terminated> ObserverPatternModifiedEx (1) [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Jun 19, 2015, 12:59:39 PM)
*** Modified Observer Pattern Demo***

Observer1: myValue in Subject is now: 5
Observer2: observes ->myValue is changed in Subject to :5

Observer1: myValue in Subject is now: 25
Observer2: observes ->myValue is changed in Subject to :25

Observer2: observes ->myValue is changed in Subject to :100
```

Assignment

Implement an observer pattern where you have multiple observers and multiple subjects.

■ **Note** I always suggests that you do the assignments by yourself. Once you complete your task, come back here to match the output.

UML Class Diagram

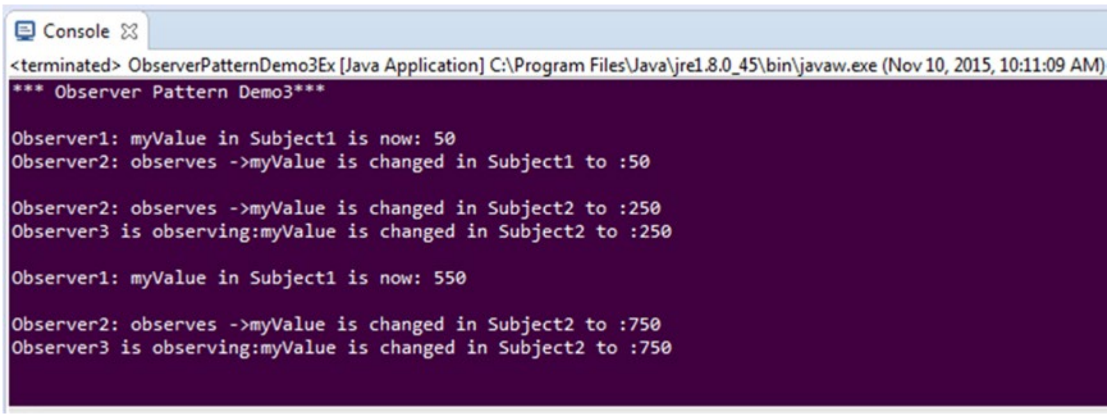


Implementation

```
package observer.pattern.demo3;

import java.util.*;

    interface IObserver
    {
        void update(String s,int i);
    }
    class Observer1 implements IObserver
    {
        @Override
                public void update(String s,int i)
                {
                System.out.println("Observer1: myValue in "+ s+ " is now: "+i);
                }
    }
    class Observer2 implements IObserver
    {
        @Override
                public void update(String s,int i)
                {
                        System.out.println("Observer2: observes ->myValue is changed in
                        "+s+" to :"+i);
                }
    }
    class Observer3 implements IObserver
    {
        @Override
                public void update(String s,int i)
                {
                        System.out.println("Observer3 is observing:myValue is changed in
                        "+s+" to :"+i);
                }
    }

    interface ISubject
    {
        void register(IObserver o);
        void unregister(IObserver o);
        void notifyObservers(int i);
    }

    class Subject1 implements ISubject
    {
        private int myValue;
    public int getMyValue() {
                return myValue;
            }

            public void setMyValue(int myValue) {
                this.myValue = myValue;
                 //Notify observers
                notifyObservers(myValue);
            }

            List<IObserver> observersList=new ArrayList<IObserver>();

            @Override
    public void register(IObserver o)
    {
            observersList.add(o);
    }
            @Override
    public void unregister(IObserver o)
    {
            observersList.remove(o);
    }
    @Override
    public void notifyObservers(int updatedValue)
    {
       for(int i=0;i<observersList.size();i++)
            {
                observersList.get(i).update(this.getClass().getSimpleName(),
                updatedValue);
            }
    }
}
class Subject2 implements ISubject
{
    private int myValue;

    public int getMyValue() {
                return myValue;
            }

            public void setMyValue(int myValue) {
                this.myValue = myValue;
                 //Notify observers
                notifyObservers(myValue);
            }

            List<IObserver> observersList=new ArrayList<IObserver>();
```

```
                @Override
        public void register(IObserver o)
        {
                observersList.add(o);
        }
                @Override
        public void unregister(IObserver o)
        {
                observersList.remove(o);
        }
        @Override
        public void notifyObservers(int updatedValue)
        {
           for(int i=0;i<observersList.size();i++)
                {
                        observersList.get(i).update(this.getClass().getSimpleName(),
                        updatedValue);
                }
        }
    }

    class ObserverPatternDemo3Ex
    {
        public static void main(String[] args)
        {
                System.out.println("*** Observer Pattern Demo3***\n");
            Subject1 sub1 = new Subject1();
            Subject2 sub2 = new Subject2();

            Observer1 ob1 = new Observer1();
            Observer2 ob2 = new Observer2();
            Observer3 ob3 = new Observer3();

          //Observer1 and Observer2 registers to //Subject 1
            sub1.register(ob1);
            sub1.register(ob2);
          //Observer2 and Observer3 registers to //Subject 2
            sub2.register(ob2);
            sub2.register(ob3);
          //Set new value to Subject 1
          //Observer1 and Observer2 get //notification
            sub1.setMyValue(50);
            System.out.println();
          //Set new value to Subject 2
          //Observer2 and Observer3 get //notification
            sub2.setMyValue(250);
            System.out.println();
            //unregister Observer2 from Subject 1
            sub1.unregister(ob2);
        //Set new value to Subject & only //Observer1 is notified
          sub1.setMyValue(550);
          System.out.println();
          //ob2 can still notice change in //Subject 2
          sub2.setMyValue(750);

      }
  }
```

Output



Console ☒

<terminated> ObserverPatternDemo3Ex [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 10, 2015, 10:11:09 AM)

```
*** Observer Pattern Demo3***

Observer1: myValue in Subject1 is now: 50
Observer2: observes ->myValue is changed in Subject1 to :50

Observer2: observes ->myValue is changed in Subject2 to :250
Observer3 is observing:myValue is changed in Subject2 to :250

Observer1: myValue in Subject1 is now: 550

Observer2: observes ->myValue is changed in Subject2 to :750
Observer3 is observing:myValue is changed in Subject2 to :750
```

CHAPTER 3

Singleton Patterns

GoF Definition: Ensure a class only has one instance, and provide a global point of access to it.

Concept

A particular class should have only one instance. We will use only that instance whenever we are in need.

Real-Life Example

Suppose you are a member of a cricket team. And in a tournament your team is going to play against another team. As per the rules of the game, the captain of each side must go for a toss to decide which side will bat (or bowl) first. So, if your team does not have a captain, you need to elect someone as a captain first. And at the same time, your team cannot have more than one captain.

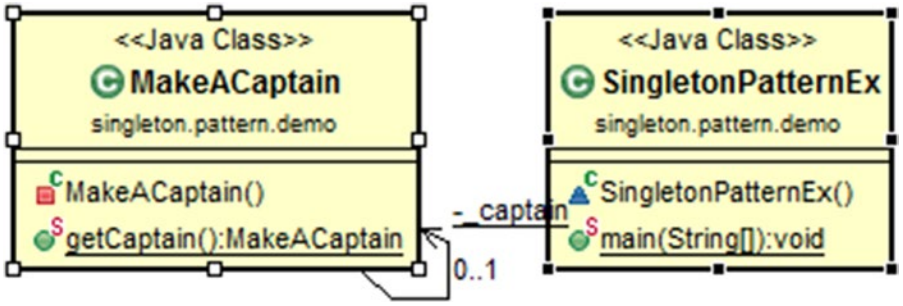
Computer World Example

In a software system sometimes we may decide to use only one file system. Usually we may use it for the centralized management of resources.

Illustration

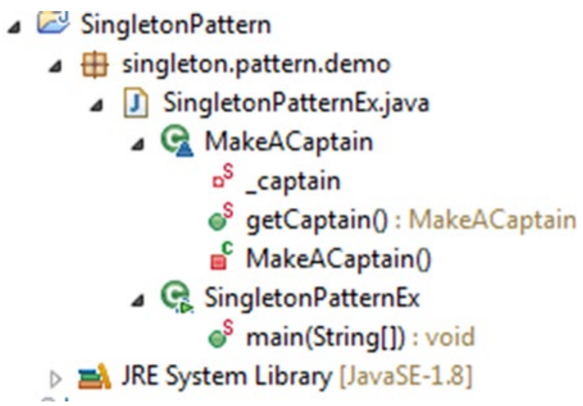
In this example, we have made the constructor private first, so that we cannot instantiate in normal fashion. When we attempt to create an instance of the class, we are checking whether we already have one available copy. If we do not have any such copy, we'll create it; otherwise, we'll simply reuse the existing copy.

UML Class Diagram



Package Explorer view

High-level structure of the parts of the program is as follows:

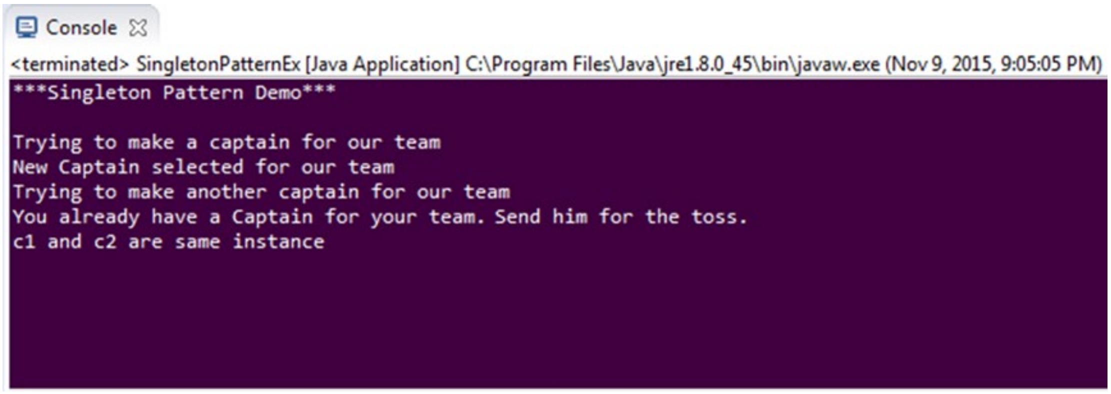


```
SingletonPattern
  singleton.pattern.demo
    SingletonPatternEx.java
      MakeACaptain
        _captain
        getCaptain() : MakeACaptain
        MakeACaptain()
      SingletonPatternEx
        main(String[]) : void
  JRE System Library [JavaSE-1.8]
```

Implementation

```
package singleton.pattern.demo;
class MakeACaptain
{
        private static MakeACaptain _captain;
        //We make the constructor private to prevent the use of "new"
        private MakeACaptain() { }
    public static MakeACaptain getCaptain()
    {

                // Lazy initialization
                if (_captain == null)
                { _captain = new MakeACaptain();
                   System.out.println("New Captain selected for our team");
                }
                else
                {
                        System.out.print("You already have a Captain for your team.");
                        System.out.println("Send him for the toss.");
                }
                return _captain;

        }
  }
class SingletonPatternEx
{
        public static void main(String[] args)
    {
                System.out.println("***Singleton Pattern Demo***\n");
                System.out.println("Trying to make a captain for our team");
                MakeACaptain c1 = MakeACaptain.getCaptain();
                System.out.println("Trying to make another captain for our team");
                MakeACaptain c2 = MakeACaptain.getCaptain();
                        if (c1 == c2)
                        {
                                System.out.println("c1 and c2 are same instance");
                        }
                }
}
```

Output



```
Console
<terminated> SingletonPatternEx [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 9, 2015, 9:05:05 PM)
***Singleton Pattern Demo***

Trying to make a captain for our team
New Captain selected for our team
Trying to make another captain for our team
You already have a Captain for your team. Send him for the toss.
c1 and c2 are same instance
```

Note

Why in the code have we used the term "Lazy initialization"?

Because, the singleton instance will not be created until the }getCaptain() method is called here.

Point out one improvement area in the above implementation?

The above implementation is not thread safe. So, there may be a situation when two or more threads come into picture and they create more than one object of the singleton class.

So what do we need to do to incorporate thread safety in the above implementation?

There are many discussions on this topic. People came up with their own solutions. But there are always pros and cons. I want to highlight some of them here:
Case (i): Use of "synchronized" keyword:

```
public static synchronized MakeACaptain getCaptain()
{
  //our code
}
```

With the above solution we need to pay for the performance cost associated with this synchronization method.
Case (ii): There is another method} called "Eager initialization" (opposite of "Lazy initialization" mentioned in our original code) to achieve thread safety.

```
class MakeACaptain
      {
          //Early initialization
          private static MakeACaptain _captain = new MakeACaptain();
          //We make the constructor private to prevent the use of "new"
          private MakeACaptain() { }

          // Global point of access //MakeACaptain.getCaptain() is a public static //method
          public static MakeACaptain getCaptain()
          {
              return _captain;
          }
      }
```

In the above solution an object of the singleton class is always instantiated.
Case (iii): To deal with this kind of situation, Bill Pugh came up with a different approach:

```
class MakeACaptain
{
        private static MakeACaptain _captain;
        private MakeACaptain() { }

      //Bill Pugh solution
      private static class SingletonHelper{
      //Nested class is referenced after getCaptain() is called
```

```
        private static final MakeACaptain _captain = new MakeACaptain();
    }

    public static MakeACaptain getCaptain()
    {
        return SingletonHelper._captain;
    }

}
```

This method does not need to use the synchronization technique and eager initialization. It is regarded as the standard method to implement singletons in Java.