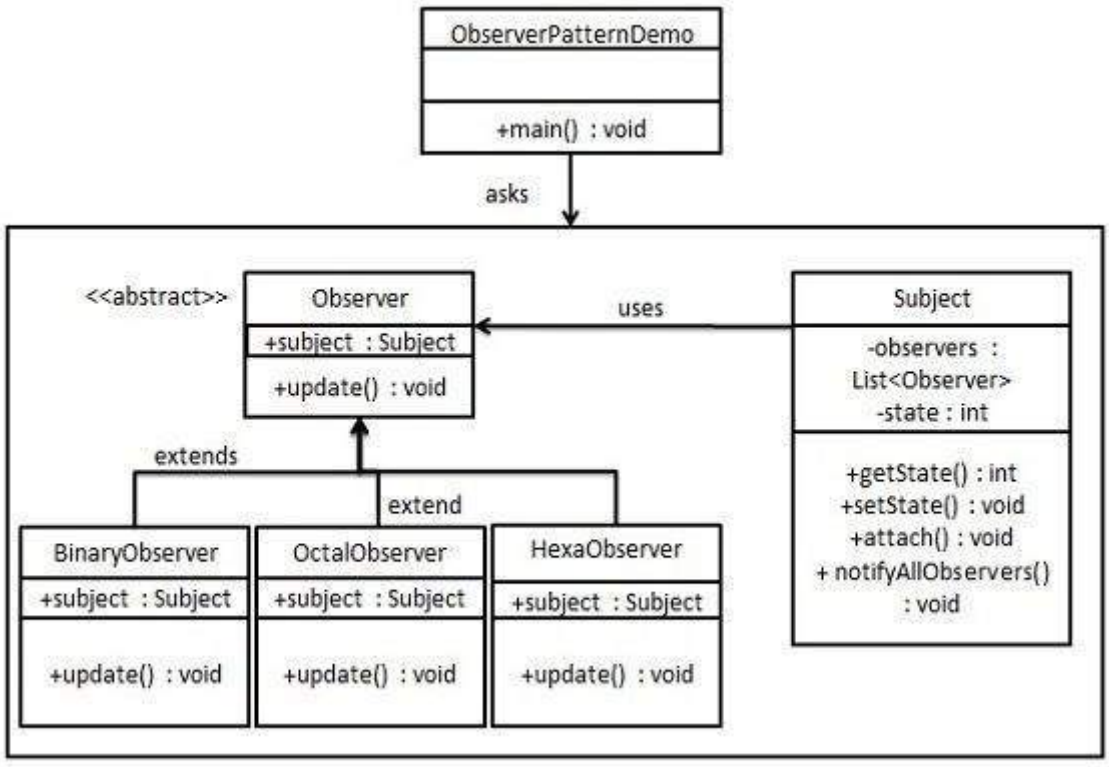
Modèles de conception - Modèle d'observateur

Le modèle d'observateur est utilisé lorsqu'il existe une relation un-à-plusieurs entre les objets, par exemple si un objet est modifié, ses objets dépendants doivent être notifiés automatiquement. Le modèle d'observateur appartient à la catégorie du modèle de comportement.

la mise en oeuvre

Le modèle d'observateur utilise trois classes d'acteurs. Sujet, observateur et client. Le sujet est un objet ayant des méthodes pour attacher et détacher des observateurs à un objet client. Nous avons créé un *observateur de* classe abstraite et un *sujet de* classe concret qui étend l' *observateur de* classe .

ObserverPatternDemo , notre classe de démonstration, utilisera l' objet de classe *Subject* et concret pour montrer le modèle d'observateur en action.



Étape 1

Créer une classe de sujet.

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();
      }
   }
}
```

Étape 2

Créez la classe Observer.

Observer.java

```
public abstract class Observer {
   protected Subject subject;
   public abstract void update();
}
```

Étape 3

Créer des classes d'observateurs concrètes

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() ).toUpperC
   }
}
```

Étape 4

Utilisez des objets *sujets* et des observateurs concrets.

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);
   }
}
```

Étape 5

Vérifiez la sortie.

```
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
```