Design Patterns

Singleton Pattern & Observer Pattern

Prepared by: wahidullah mudaser

introduction

- This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

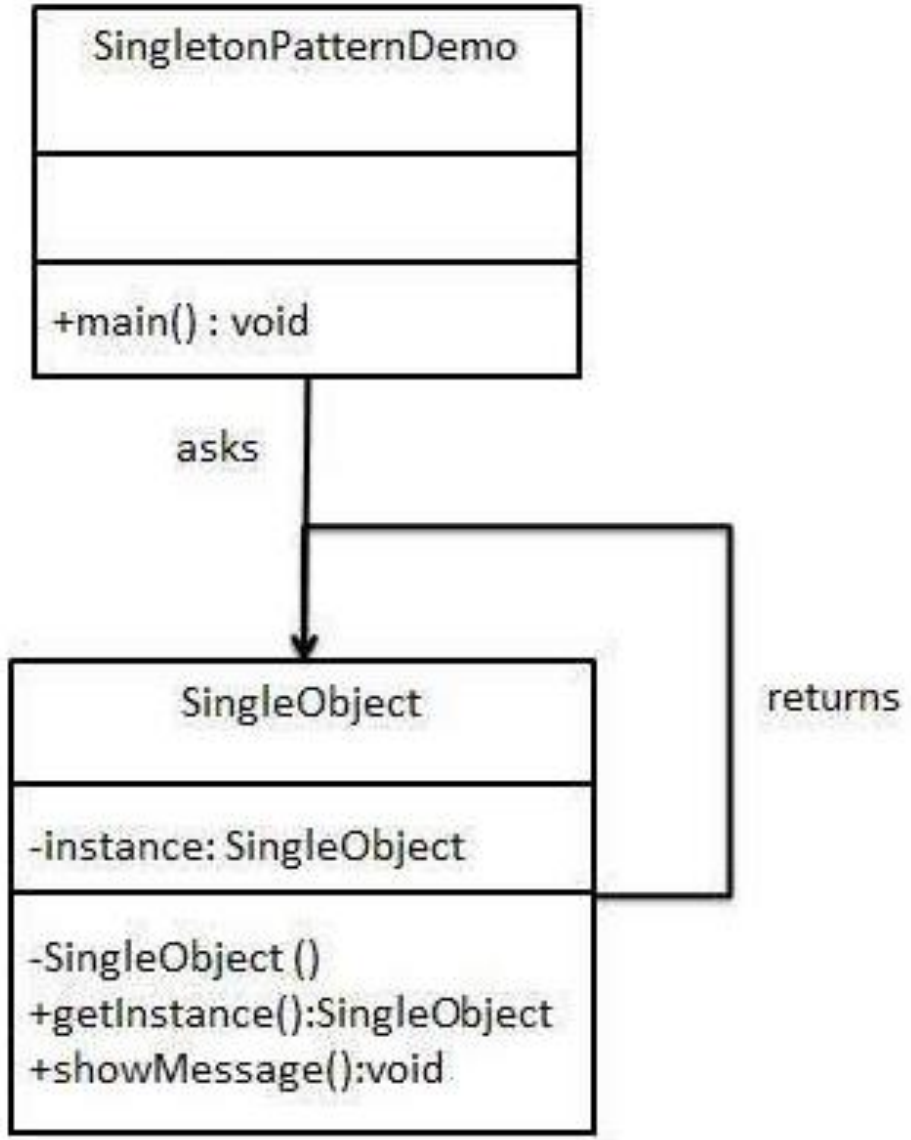
- This pattern involves a single class which is responsible to create an object while making sure that only single object gets created.

- This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Implementation

- We're going to create a *SingleObject* class.
- *SingleObject* class have its constructor as private and have a static instance of itself.
- *SingleObject* class provides a static method to get its static instance to outside world.
- *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.

Cont…


SingletonPatternDemo
+main() : void
asks
SingleObject
-instance: SingleObject
-SingleObject ()
+getInstance():SingleObject
+showMessage():void
returns

Step-1

- Create a Singleton Class.
- *SingleObject.java*

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Step-2

- Get the only object from the singleton class.
- *SingletonPatternDemo.java*

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Observer Pattern

Intro

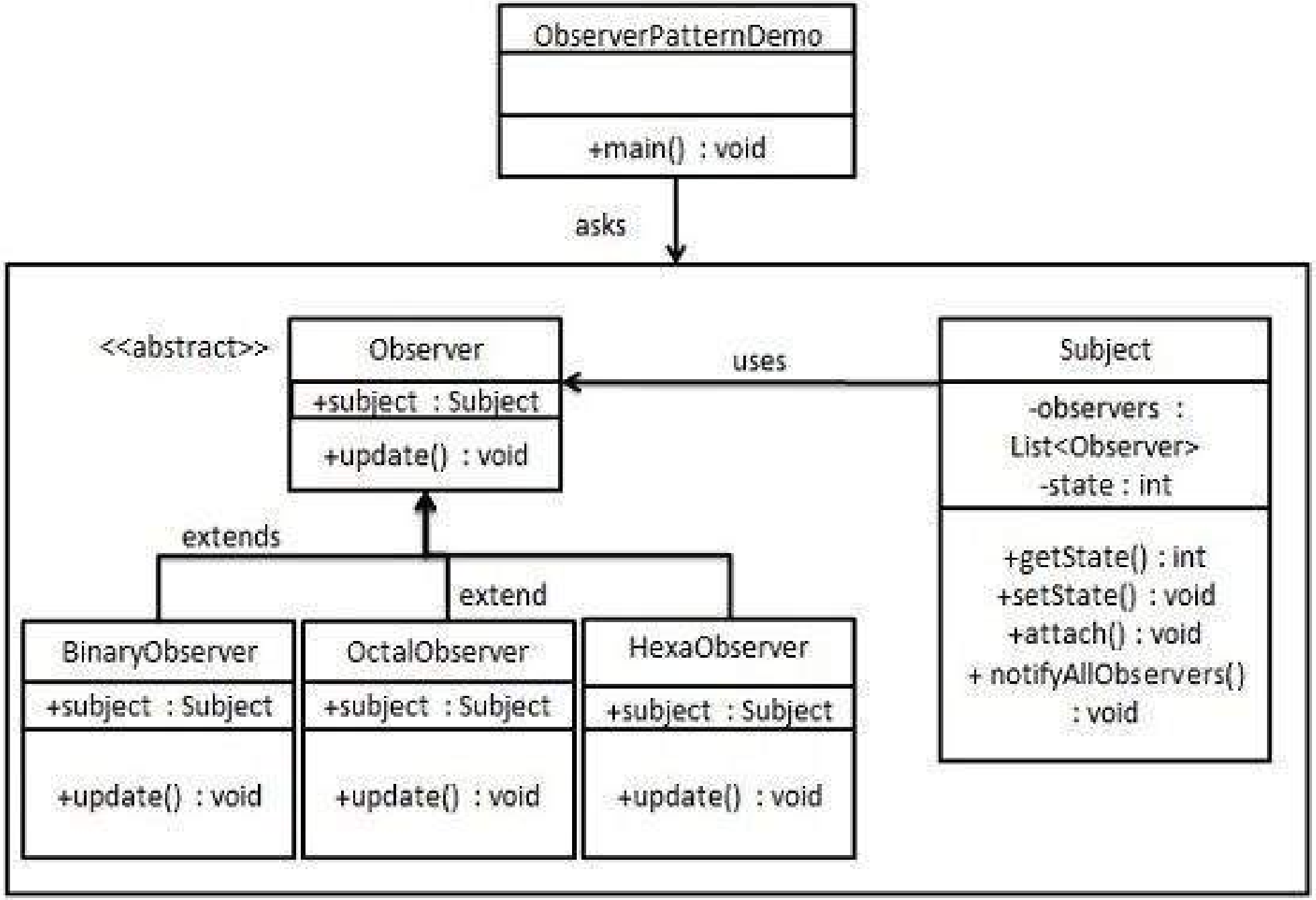
- Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.

- Observer pattern falls under behavioral pattern category.

Implementation

- Observer pattern uses three actor classes. Subject, Observer and Client.
- Subject is an object having methods to attach and detach observers to a client object.
- We have created an abstract class *Observer* and a concrete class *Subject* that is extending class *Observer*.
- *ObserverPatternDemo*, our demo class, will use *Subject* and concrete class object to show observer pattern in action.

C


ObserverPatternDemo
+main() : void
asks
<<abstract>>
Observer
+subject : Subject
+update() : void
uses
Subject
-observers : List<Observer>
-state : int
+getState() : int
+setState() : void
+attach() : void
+ notifyAllObservers() : void
extends
extend
BinaryObserver
+subject : Subject
+update() : void
OctalObserver
+subject : Subject
+update() : void
HexaObserver
+subject : Subject
+update() : void

Step-1

- Create Subject class.
- *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();
      }
   }
}
```

Cont…

- **Step-2**
  - Create Observer class.
  - *Observer.java*

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

- **Step-3**
  - Create concrete observer classes
  - *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() ) );
   }
}
```

Cont…

- *OcatalObserver.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() ).toUpperCase() );
   }
}
```

cont

- **Step-4**
  - *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);
   }
}
```

- **Step-5**
  - output

```
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
```

Any