Started on	Saturday, 7 March 2020, 3:00 PM
State	Finished
Completed on	Saturday, 7 March 2020, 4:04 PM
Time taken	1 hour 3 mins
Marks	3.00/3.00
Grade	**10.00** out of 10.00 (**100**%)

Information

General Lab Information

In this lab we will be looking at the Observer pattern.

The goals for this lab are:

- To understand what the Observer pattern is
- To understand when to use the Observer pattern
- Be able to implement the Observer pattern

Information

The Observer Pattern

The Observer pattern is most often used in one-to-many dependency relationships, where there is a **subject** being observed by a number of objects that are interested in state changes to the subject.

- **Subject** - Maintains a list of observers, and notifies them of changes to it's state.
- **Observer** - Receives notifications from the Subject, and may act on these notifications

For example, Starman wants to avoid big galactic storms, as they make it hard to zoom around in space and might damage his sweet ride. He's signed up for the Galaxy Weather Service reports, and gets updates about weather all around the Galaxy. In this situation, Galaxy Weather Service is the Subject, which notifies Starman, the Observer, of changes that he may be interested in. Starman may choose to act on these updates, or he may ignore them, it's up to him.

Information

The Observer Pattern in Java

The Observer pattern is very commonly used, and in older versions, Java provided an Observable class, and an Observer interface to help implement this pattern. Unfortunately these are now deprecated. But they are relatively easy to implement yourself.

- **Observer interface** - Objects that observe other objects should implement the Observer interface, which required them to implement an update() method to receive updates from it's subject. This allows the subject being observed to maintain a list of their observers, and use their update() method to update them when changes occur.
- **Observable class** - A class which may be observed (aka, the **Subject**), should extend the Observable class, which provides methods for adding, deleting, and notifying observers of changes to state. It also provides a setChanged() method, which should be called whenever it undergoes a state change.

Below is the source code for the Observer interface and the Observable class. Use these for the following questions.

ZIP Link

Question **1**

Correct

Mark 1.00 out of 1.00

The Observable Class

Objects that may be observed should subclass the **Observable** class, which provides us with a number of handy methods:

- **addObserver**(Observer observer) - adds the given observer to the objects set of observers
- **setChanged**() - marks the objects as having changed state
- **hasChanged**() - indicates whether the state of the object has changed (set with the setChanged() method)
- **notifyObservers**() - notify observers of the object about the state change
- ... and a few others which you are welcome to explore in the Java documentation

Let's make a subject

Write a class for the Galaxy Weather Service, called **GalaxyWeather**, which keeps track of weather and informs observers of changes. This is what Starman will receive weather reports from.

It should have the following **private boolean** attributes, each values should have getter and setter, and should all be initialized as False.

- isSolarFlare
- isSpaceStorm
- isComets.

These indicate different types of "weather" that occur in space.

The class should extend the Observable class, and any time an attribute changes, then it should call setChanged() and notify it's observers using one of the methods provided by the Observable class. It should only setChanged and notify if the actual value given is different to what it is already set to (if a variable is currently false, and we call setX(false), it shouldn't trigger an update)

You can have a go at testing your class with this basic observer (we'll improve on this later):

```
public class SpaceMan implements Observer {

  @Override
  public void update(Observable o, Object arg) {
    System.out.println("Received update!");
  }
}
```

The Observable class provided to you will be available on the quiz server, so you only need to submit the GalaxyWeather class. **When submitting, exclude the import statement for the Observable class.**

For example:

Test	Result

`GalaxyWeather weather = new GalaxyWeather();`
`System.out.println(weather.getIsComets());`
	`false`

`GalaxyWeather weather = new GalaxyWeather();`
`SpaceMan spacey = new SpaceMan();`
`weather.addObserver(spacey);`
`weather.setIsSolarFlare(true);`
	`Received update!`

Answer: (penalty regime: 0, 10, 20, ... %)

```
public class GalaxyWeather extends Observable
{
    private boolean isSolarFlare;
    private boolean isSpaceStorm;
    private boolean isComets;

    public GalaxyWeather() {
        this.isSolarFlare = false;
        this.isSpaceStorm = false;
        this.isComets = false;
    }

    public void setIsSolarFlare(boolean isSolarFlare) {
        if (this.isSolarFlare != isSolarFlare) {
            this.isSolarFlare = isSolarFlare;
            setChanged();
            notifyObservers();
        }
    }

    public void setIsSpaceStorm(boolean isSpaceStorm) {
        if (this.isSpaceStorm != isSpaceStorm) {
```

	Test	Expected	Got	

✓	`GalaxyWeather weather = new GalaxyWeather();`
	`System.out.println(weather.getIsComets());`
		`false`	`false`	✓

✓	`GalaxyWeather weather = new GalaxyWeather();`
	`SpaceMan spacey = new SpaceMan();`
	`weather.addObserver(spacey);`
	`weather.setIsSolarFlare(true);`
		`Received update!`	`Received update!`	✓

✓	`GalaxyWeather weather = new GalaxyWeather();`
	`SpaceMan spacey = new SpaceMan();`
	`weather.addObserver(spacey);`
	`weather.setIsComets(true);`
		`Received update!`	`Received update!`	✓

✓	`GalaxyWeather weather = new GalaxyWeather();`
	`SpaceMan spacey = new SpaceMan();`
	`weather.addObserver(spacey);`
	`weather.setIsSpaceStorm(true);`
		`Received update!`	`Received update!`	✓

✓	`GalaxyWeather weather = new GalaxyWeather();`
	`SpaceMan spacey = new SpaceMan();`
	`weather.addObserver(spacey);`
	`weather.setIsSpaceStorm(false);`
				✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Question **2**

Correct

Mark 1.00 out of 1.00

Observers

Observers need to implement the Observer interface. This requires that the class has an update() method, which takes an Observable and an Object, which may be some information that is being passed through as the update. So when an Observer is notified of an update, it is passed a reference of the subject it is watching, so it can then check the state of the object for the information that it is interested in. This is useful because the subject doesn't need to know what each observer wants to know, all it needs to do is notify everyone when something changes.

An Observer for Starman

Create a class called Starman that implements the Observer interface. This means you will need to implement the update() method, so that Starman knows then changes occur to the weather. This should work with your GalaxyWeather class from the previous question, so have a go at testing them together. **Include the code for your GalaxyWeather class in your submission.**

A Starman object should react to updates from the GalaxyWeather in the following ways:

- If there is a solar flare, Starman has to stay home, print "Sigh... no adventuring today"
- If there is comets (but no solar flare), then print "I'll show those comets who's boss!"
- Otherwise, do nothing, Starman doesn't have to act on each update

Note that the update() method only takes a generic Observable o, so you will need to cast this to a GalaxyWeather to check its state.

The Observable class and Observer interface provided to you will be available on the quiz server. **When submitting, exclude the import statement for the Observable class and the Observer interface.**

For example:

Test	Result

`GalaxyWeather weather = new GalaxyWeather();`
`Starman spacey = new Starman();`
`weather.addObserver(spacey);`
`weather.setIsComets(true);`
	`I'll show those comets who's boss!`

Answer: (penalty regime: 0, 10, 20, ... %)

```java
public class Starman implements Observer {

      @Override
      public void update(Observable o, Object arg) {
        if (o instanceof GalaxyWeather) {
          if (((GalaxyWeather)o).getIsSolarFlare()) {
            System.out.println("Sigh... no adventuring today");
          } else if (((GalaxyWeather)o).getIsComets()) {
            System.out.println("I'll show those comets who's boss!");
          }
        }
      }
    }

public class GalaxyWeather extends Observable
{
    private boolean isSolarFlare;
    private boolean isSpaceStorm;
    private boolean isComets;

    public GalaxyWeather() {
      this.isSolarFlare = false;
```

	Test	Expected	Got	

✓	`GalaxyWeather weather = new GalaxyWeather();`
	`Starman spacey = new Starman();`
	`weather.addObserver(spacey);`
	`weather.setIsComets(true);`
		`I'll show those comets who's boss!`	`I'll show those comets who's boss!`	✓

✓	`GalaxyWeather weather = new GalaxyWeather();`
	`Starman spacey = new Starman();`
	`weather.addObserver(spacey);`
	`weather.setIsSolarFlare(true);`
		`Sigh... no adventuring today`	`Sigh... no adventuring today`	✓

	Test	Expected	Got	

✓	`GalaxyWeather weather = new GalaxyWeather();`
	`Starman spacey = new Starman();`
	`weather.addObserver(spacey);`
	`weather.setIsSpaceStorm(true);`
				✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Question **3**

Correct

Mark 1.00 out of 1.00

Adding and removing observers

Once we have a class that is Observable, and a class that can Observe, we need to actually begin the observation, and in some cases we will want to stop observing as well.

In the questions above, our code has done that for you using the Observable.addObserver() method. You can check what other method the class Observable provide.

Doing it yourself

Extending your existing Starman class, add two methods to allow the class to start and stop observing a GalaxyWeather instance.

- void startObserving(GalaxyWeather galaxyWeather) - This method should add itself to the registered observers in the given GalaxyWeather
- void stopObserving(GalaxyWeather galaxyWeather) - This method should remove itself from the registered observers in the given GalaxyWeather.

Include the code for your GalaxyWeather class in your submission. The Observable class and Observer interface provided to you will be available on the quiz server. **When submitting, exclude the import statement for the Observable class and the Observer interface.**

Answer: (penalty regime: 0, 10, 20, ... %)

```java
public class Starman implements Observer {

    public void startObserving(GalaxyWeather galaxyWeather) {
        galaxyWeather.addObserver(this);
    }

    public void stopObserving(GalaxyWeather galaxyWeather) {
        galaxyWeather.deleteObserver(this);
    }

    @Override
    public void update(Observable o, Object arg) {
        if (o instanceof GalaxyWeather) {
            if (((GalaxyWeather)o).getIsSolarFlare()) {
                System.out.println("Sigh... no adventuring today");
            } else if (((GalaxyWeather)o).getIsComets()) {
                System.out.println("I'll show those comets who's boss!");
            }
        }
    }
}
```

	Test	Expected	Got	

✓	`GalaxyWeather weather = new GalaxyWeather();`
	`Starman spacey = new Starman();`
	`spacey.startObserving(weather);`
	`weather.setIsComets(true);`
		`I'll show those comets who's boss!`	`I'll show those comets who's boss!`	✓

✓	`GalaxyWeather weather = new GalaxyWeather();`
	`Starman spacey = new Starman();`
	`spacey.startObserving(weather);`
	`weather.setIsSolarFlare(true);`
	`spacey.stopObserving(weather);`
	`weather.setIsComets(true);`
		`Sigh... no adventuring today`	`Sigh... no adventuring today`	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

◄ Quiz 6 - GUI Development (hidden)   Jump to...   Lab Resources (hidden) ►