OOP - Exercises 4/20 Solutions

1. The **Observer Pattern** is characterized by a Subject, an Observer, a concrete implementation of the Observer called a ConcreteObserver, and attaching and notification methods.

Observer Pattern

Context

1. An object (which we'll call the subject) is the source of events (such as "my data has changed").

2. One or more objects (called the observers) want to know when an event occurs.

Solution

1. Define an observer interface type. Observer classes must implement this interface type.

2. The subject maintains a collection of observer objects.

3. The subject class supplies methods for attaching observers.

4. Whenever an event occurs, the subject notifies all observers.

For a *JButton* (the subject), identify the Observer, ConcreteObserver objects, and the "attaching" and "notifying" methods.

Observer → ActionListener interface

ConcreteObserver → the class implementing the ActionListener interface

attach → addActionListener() method

notify() → actionPerformed() method

2. Suppose you want to provide access to a file myLogfile.txt that clients can write logs to. Implement a FileLogger class using the **Singleton Pattern** so that only one instance of FileLogger is available at any time. The client should be able to write logs to the file using a static method:

```
FileLogger fl=FileLogger.getInstance();
fl.writeLog(" The process terminated abnormally" );
```

Provide complete implementation of FileLogger class. You may assume that the text file myLogfile.txt exists.

```
import java.io.*;
public class FileLogger {
   private static FileLogger instance;
   private static File file;
   private FileLogger(){
            file=new File("logFile.txt");
   }
   public static synchronized FileLogger getInstance(){
            if (instance == null)
               instance = new FileLogger();
            return instance;
   }
   public static void writeLog(String s) throws Exception {
            FileWriter fw=new FileWriter(file,true);
            fw.write(s+System.getProperty("line.separator"));
            fw.close();
   }
}
```

```
/** Test the FileLogger class */
public class testFileLogger{
   public static void main(String[] args){
            FileLogger fl = FileLogger.getInstance();
            try{
               fl.writeLog(args[0]);
            }catch(Exception e){
               e.printStackTrace();
            }
   }
}
```

3. Consider the Calculator class (myCalculator.java) from the previous exercise. Use this class to write a Java program that accepts an arithmetic expression as a command-line argument string, and outputs its value. For example if I call my program class expressionEvaluator, then executing:

>*java expressionEvaluator "42*(23+7/9)+7*(90-22)/3"*

should give the value 1157.33 correct to two decimal places.

(This is the third use of the myCalculator class – MVC pattern in action!)

```
import java.text.DecimalFormat;
public class expressionEvaluator{
    public static void main(String[] args)throws Exception{
            myCalculator calc=new myCalculator();
            double d=(double)calc.evaluateExpression(args[0]);
            DecimalFormat df=new DecimalFormat("#.00");
            System.out.println(df.format(d));
    }
}
```

4. Write a program that takes a class name as command-line argument and prints all the interfaces that class implements, if any, as well as all of its superclasses and interfaces implemented by the super classes.

```
public class ShowSuperClassesAndInterfaces{
   public static void main(String[] args)throws Exception{
           Class c1=Class.forName(args[0]);
           printInterface(c1);
           while ((c1 = c1.getSuperclass()) != null) {
              System.out.println("extends " + c1.getName());
              printInterface(c1);
           }
   }
   public static void printInterface(Class c){
           Class[] theInterfaces = c.getInterfaces();
           for (int i = 0; i < theInterfaces.length; i++) {
              String interfaceName = theInterfaces[i].getName();
              System.out.println("Interface->"+interfaceName);
           }
   }
}
```

5. What are the three different ways of getting the Class associated with a Java class or its instance? What are the situations where each method is useful?