Exercises on Design Patters: Short form answers

The Basics

ADAPTER, DECORATOR, FACTORY METHOD, OBSERVER (MVC), and SINGLETON design patterns examined

1. Since the advent of Java 8, abstract classes and interfaces display an ever increasing number of similarities in that they both cannot be instantiated and they each contain a blend of declared methods (with or without implementing details). However, there are still differences that exist and three of them are outlined below:

   Abstract classes are ***extended*** by other subclasses, creating a top down hierarchy, whereas interfaces are ***implemented*** by concrete classes as an adherence to specification.

   Example:

   ```
   public class A extends B {}             //example of extending an abstract class
   public class A implements C {}          //example of implementing an interface
   ```

   A class can implement any number of interfaces, but can only extend one superclass.

   Example:

   ```
   public class A implements C, D, E {}
   ```

   An abstract class can contain methods that are public, private or protected whereas interfaces by definition only contain public methods.

   Example:

   ```
   public abstract class AbstractClass {
           public void aMethod();
           private int anotherMethod();
   }

   public interface Interface {
           void aMethod();
   }
   ```

2. The statement *every interface must have at least one method* is FALSE. An interface without any methods is utilised as an identifier, otherwise known as a marker interface. For example, an interface can extend from a superclass (that does have methods).

   The statement *an interface can declare instance fields that an implementing class must also declare* is FALSE. An interface can only declare constants i.e. fields that are static and final.

   The statement *although you can't instantiate an interface, an interface definition can declare constructor methods that require an implementing class to provide constructors with give signatures* is also FALSE. An interface cannot declare constructors, purely because there should be no means to instantiate an interface (at least as far as I know in Java).

3. A listener class is an example of an interface with methods that do not imply responsibility on the part of the implementing class to take action on behalf of the caller or to return a value. In this scenario, the interface does not define a pre-supposed contract, but is used to register / notice events that occur throughout the lifecycle of the program.

4. The WindowAdapter class is required to implement the interface WindowLisener, but each method is defined to perform no action whatsoever. This implementation of the interface enables further subclasses to extend WindowAdapter and to then only override those methods that the respective subclasses require utilisation of. Each subclass of WindowAdapter (and hence each indirect implementation of WindowListener) therefore does not need to provide an implementation for every declared method in the interface, but only those that they need.

5. You can prevent other developers from constructing new instances of a class by implementing the singleton pattern. This involves creating a private constructor for the class, storing a unique instance of the class in a static field, and declaring a static getInstance method to retrieve that instance. The class must contain no public constructors.

6. A lazy initialisation of a singleton will enable the instance to be computed only when it is first required in the program. This can be beneficial if the singleton is expensive to build, requires a high CPU running cost, or if the instantiation is dependent on other information that is otherwise not available at its static initialisation time.

7. Example of the use of java.util.Observable and java.util.Observer classes (obtained from Java Tutorials):

```
class ObservedObject extends Observable {

   private String watchedValue;


   public ObservedObject(String value) {

   watchedValue = value;

   }
```

```java
   public void setValue(String value) {
   // if value has changed notify observers
   if(!watchedValue.equals(value)) {
   System.out.println("Value changed to new value: "+value);
   watchedValue = value;

   // mark as value changed
   setChanged();
   // trigger notification
   notifyObservers(value);
   }
   }
}
public class ObservableDemo implements Observer {
   public static void main(String[] args) {
   // create watched and watcher objects
   ObservedObject watched = new ObservedObject("Original Value");
   // watcher object listens to object change
   ObservableDemo watcher = new ObservableDemo();

   // trigger value change
   watched.setValue("New Value");

   // add observer to the watched object
   watched.addObserver(watcher);

   // trigger value change
   watched.setValue("Latest Value");
   }

   public void update(Observable obj, Object arg) {
   System.out.println("Update called with Arguments: "+arg);
   }
}
```

8. The observer pattern supports the MVC pattern. This is TRUE because the View-Controller can be implemented as observers, and the Model as an observable.

9. Two examples of Java methods that return new objects include:

   a. toString()
   b. clone()

10. Signs that a factory method is at work:

    There exists the creation of a new object without the keyword **new**.

    The factory method returns a type that is abstract.

    The method is implemented by several classes.

11.

```
Writer out = new PrintWriter(System.out);

out = new WrapFilter(new BufferedWriter(out), 15);

((WrapFilter) out).setCenter(true);

out = new RandomCaseFilter(out);
```