Exercise on Design Patterns

Short Form Answers

1. *Write down three differences between abstract classes and interfaces in Java 8. Provide examples to illustrate your answer.*

a. An abstract class can have non-final member variables. E.g. interfaces can only have constants.
b. Only public methods in interfaces, abstract classes can use Protected. (visibility / accessibility modifiers)
c. An abstract class can only extend one class, where an interface can extend multiple interfaces. I.e. interfaces support multiple inheritance.
d. Abstract class can have a constructor, e.g. default constructor.
e. State
f. Abstract classes inherits from Object. Hence, has reflection.

2. *Are the following true or false?*

a. *Every interface must have at least one method.* **False**. An interface could extend other interfaces without adding additional methods. A marker class (see cloneable) to signify functionality. Bundle interfaces.
b. *An interface can declare instance fields that an implementing class must also declare*. **False**. Compiler error observed.
c. *Although you can't instantiate an interface, an interface definition can declare constructor methods that require an implementing class to provide constructors with given signatures*. **False**.

3. *Provide 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.*

See observer pattern. Register interest and receive asynchronous call.

An interface (in Java 8) can have a default method to provide an implementation, taking responsibility away from the implementing class.

4. *What is the value of a stub class like* `WindowAdapter` *which is composed of methods that do nothing?*

Supplies default stubs for all methods allowing implementing classes to only implement some of the methods in the original interface.

5. How can you prevent other developers from constructing new instances of your class? Provide appropriate examples to illustrate your answer.

Use the singleton design pattern. Set the constructor as private and provide a static getter to retrieve the instance when needed.

```
public class singleton {

    // private member variable
    private static singleton instance = new singleton();

    // private constructor
    private singleton() {}

    // public getter
    public static singleton getInstance() {return instance;}

}
```

6. Why might you decide to lazy-initialise a singleton instance rather than initialise it in its field declaration? Provide examples of both approaches to illustrate your answer.

Save load time and space if not used. Requires null check?

```
public class singleton {

    // private member variable
    private static singleton instance;

    // private constructor
    private singleton() {}

    // public getter
    public synchronise static singleton getInstance() {

        if (instance == null)
            instance = new singleton();

        return instance;
    }

}
```

7. Using the java.util.Observable and java.util.Observer classes/interfaces show how one object can be informed of updates to another object.

```
public class Subject extends Observable {

    private boolean toggle;

    public Subject() {
        this.toggle = true;
    }

    public void toggleToggle() {
        if (this.toggle)
            this.toggle = false;

        setChanged();
        notifyObservers();

    }

}
```

```
public class Watcher implements Observer {

    @Override
    public void update(Observable o, Object arg) {
```

```
        System.out.println("Subject changed");
    }
}
```

```
public class ObserverApp {

    public static void main(String[] args) {

        Observer observer = new Watcher();
        Subject observable = new Subject();
        observable.addObserver(observer);
        observable.toggleToggle();
        observable.toggleToggle();

    }
}
```

8. "The Observer pattern supports the MVC pattern". State if this statement is true or false and support your answer by use of an appropriate example.
   True.
   The "view" or components of the viewer can implement a listenable interface as the subject. The controller can work as an observer, so when input mechanisms on the view are changed the controller can respond appropriately.

9. Provide examples of two commonly used Java methods that return a new object.

   `java.lang.reflect.Constructor.newInstance()` and `Class.newInstance()`

10. What are the signs that a Factory Method is at work?

    When instantiation of multiple objects which share a superclass is deferred to separate class then the factory pattern is probably work.

11. If you want to direct output to System.out instead of to a file, you can create a Writer object that directs its output to System.out:

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

    Write a code example to define a Writer object that wraps text at 15 characters, centres the text, sets the text to random casing, and directs the output to System.out. Which design pattern are you using?

    Decorator…

```
import java.io.IOException;
import java.io.PrintWriter;
import java.io.Writer;
```

```
public class PrintWriterDecorator extends WriterDecorator {



	public PrintWriterDecorator() {
		super(new PrintWriter(System.out));
	}

	@Override
	public void write(char[] cbuf, int off, int len) throws
IOException {
		StringBuilder sb = new StringBuilder();
		sb.append(cbuf);
		for (int i = 0; i < len; i++){
			if (Math.random() < 0.5)
				sb.setCharAt(i,
Character.toUpperCase(sb.charAt(i)));
			else
				sb.setCharAt(i,
Character.toLowerCase(sb.charAt(i)));
		}

		int i = 15;
		while (i < len){
			sb.insert(i, "\n");
			len += 2;
			i += 17;
		}

		super.write(sb.toString().toCharArray(), off, len);

		super.flush();
	}
}

import java.io.IOException;
import java.io.PrintWriter;
import java.io.Writer;


public abstract class WriterDecorator extends Writer {

	protected Writer writer;

	public WriterDecorator(Writer decoratedWriter){
		this.writer = decoratedWriter;
```

```
	}


	@Override
	public void write(char[] cbuf, int off, int len) throws
IOException {
		writer.write(cbuf, off, len);
	}

	@Override
	public void flush() throws IOException {
		writer.flush();

	}

	@Override
	public void close() throws IOException {
		writer.close();
	}

}



import java.io.IOException;
import java.io.Writer;


public class DecoRunner {

	public static void main(String[] args) {
		Writer writer = new PrintWriterDecorator();
		String test = "testing";
		try {
			writer.write(test.toCharArray(), 0, test.length());
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}

	}
```