DESIGN PATTERNS WORKSHEET

SHORT FORM QUESTIONS

1. Three differences between abstract classes and interfaces:
    - Abstract class can specify constructor
    - Interface specifies behaviour and not state
    - Interface can have default method

2. (a) Every interface must have at least one method : FALSE. Marker interfaces exist such as Serializable, Clonable.

   (b) An interface can declare instance fields that an implementing class must also declare: FALSE

   (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. Any of the listener classes
4. Provide default implementation to avoid wrong implementation.
5. Make constructor private. See singleton implementation in Long form questions part.
6. To preserve resources if creating instance at loading time consumes a lot of resources, such as memory. Also, it is possible that instance may never get used. See implementation in Long form questions part.
8. True
9. Constructor, getInstance.
10. Signs of Factory Pattern
    - No new keyword in client
    - Use of getInstance and static
    - Returning a type rather than implementation
11. Decorator

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

or

```
WrapFilter out = new Wrap(new Buffer(newCase(new PrintWriter(s.out))),15);
out.setCentre(true);
```

LONG FORM QUESTIONS

2(a) Synchronised version of singleton creates thread safe singleton class by making static getInstance method synchronised. This will ensure that only one thread will be executing in the code segment responsible for checking and creating instance of a class. Applying synchronised to static method locks the Class.

```
public class ThreadSafeSingleton {

  private static ThreadSafeSingleton instance;
  private ThreadSafeSingleton(){}
```

```
    public static synchronized ThreadSafeSingleton getInstance(){
        if(instance==null){
            instance = new ThreadSafeSingleton();
        }
        return instance;
    }
}
```

2(b) Synchronisation can be expensive in terms of performance because before a particular thread enters into synchronised code, it will acquire a lock on the Object. This lock is released after thread is done with the code. While the code is locked, other threads have to **wait** until the resource is unlocked, reducing benefits of multithreading.

2(c) If calling the getInstance() method isn't causing substantial overhead for your application, do not need to worry about it. Synchronising getInstance() is straightforward and effective. Need to keep in mind that synchronising a method can decrease performance by a factor of 100, so if a high traffic part of the code begins using getInstance(), might need to reconsider.

2(d) To reduce the overhead involved in synchronisation, **double checked locking** is used. In this implementation, the synchronisation is applied inside the if condition with an additional check to make sure that only one instance of singleton class is created. So first check to see if an instance is created, and if not, then synchronise. This way, only synchronise the first time through.

```
public class SingletonDoubleLocking {

    private static SingletonDoubleLocking instance;
    private SingletonDoubleLocking(){}

    public static SingletonDoubleLocking getInstance(){
        if(instance==null){
            synchronized (SingletonDoubleLocking.class) {
                if (instance == null) {
                    instance = new SingletonDoubleLocking();
                }
            }
        }
        return instance;
    }
}
```

But could just keep eager implementation, which is completely thread safe. Threads are not creating instance of the class. It has been created when the class is loaded. The JVM guarantees that instance will be created before any thread accesses the static *instance* variable.

BREAKING SINGLETON DESIGN PATTERN EXAMPLES

Example of code, which breaks Singleton using **Reflection**

```
import java.lang.reflect.Constructor;


public class ReflectionSingletonTest {
    public static void main(String[] args) {
        EagerSingleton instanceOne = EagerSingleton.getInstance();
        EagerSingleton instanceTwo = null;
        try {
            Constructor[] constructors = EagerSingleton.class.getDeclaredConstructors();
            for (Constructor constructor : constructors) {
                //Below code will destroy the singleton pattern
                constructor.setAccessible(true);
                instanceTwo = (EagerSingleton) constructor.newInstance();
                break;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println(instanceOne.hashCode());
        System.out.println(instanceTwo.hashCode());
    }

}
```

When run above code, will result in hashCodes being different, hence more than one instance of Singleton will be created.
To overcome this issue, private constructor should be implemented as follows:

```
private SingletonProtected() {
    if (instance != null) {
        throw new IllegalStateException("Already created.");
    }
}
```

Another example of breaking singleton design pattern is when it implements **Serializable** interface. Code below will result in two different instances of singleton classes.

```
public class SingletonSerializedTest {

    public static void main(String[] args) throws FileNotFoundException, IOException, ClassNotFoundException
{
        SingletonProtected instanceOne = SingletonProtected.getInstance();
        ObjectOutput out = new ObjectOutputStream(new FileOutputStream("filename.ser"));
        out.writeObject(instanceOne);
        out.close();

        //deserailize from file to object
        ObjectInput in = new ObjectInputStream(new FileInputStream("filename.ser"));
        SingletonProtected instanceTwo = (SingletonProtected) in.readObject();
        in.close();

        System.out.println("instanceOne hashCode="+instanceOne.hashCode());
        System.out.println("instanceTwo hashCode="+instanceTwo.hashCode());

    }
}
```

This happens when singleton instance is serialised and then deserialised, giving new instance of the same singleton class. By using java API, can implement readResolve() method in singleton class, which allows to override the instance read from the stream. This will insure that there is only one instance of singleton class. Therefore, whenever the singleton class is deserialised, it will return the same singleton instance.

```
private Object readResolve() throws ObjectStreamException {
   return instance;
}
```

Another way singleton pattern can be broken is when **Class Loaders** are used. Multiple class loaders are commonly used in many situations – including servlet containers. Using them may result in multiple singleton instances independent of how carefully singleton class is implemented. In order to ensure that same class loader loads the singleton, need to specify the class loader in singleton class.

```
private static Class getClass(String classname) throws ClassNotFoundException {
   ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
   if (classLoader == null)
      classLoader = SingletonProtected.class.getClassLoader();
   return (classLoader.loadClass(classname));
}
```

This method tries to associate the class loader with the current thread. If that class loader is null, the method uses the same class loader that loaded a singleton base class. This method can be used instead of Class.forName().

Finally, singleton pattern may be broken if singleton implements/extends class which implements **Cloneable** interface. This makes singleton clonable too and will result in two instances of singleton class if it is cloned. To avoid this issue, need to override clone() method of Object class and throw the CloneNotSupported exception.

```
@Override
public Object clone() throws CloneNotSupportedException {
   throw new CloneNotSupportedException("Singleton, cannot be cloned");
}
```