Exercises on Design Patterns

The Basics

Adapter, Decorator, Factory Method, Observer (MVC), and Singleton design patterns examined

You should use Java 8 to answer the following exercises.
The source code examples can be found on the repository under the `exercise-dp` folder.

Short form questions

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

2. Are the following *true* or *false*?
   (a) Every interface must have at least one method.
   (b) An interface can declare instance fields that an implementing class must also declare.
   (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.

   Provide examples to illustrate your answers.

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.

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

"stub/WindowListener.java"

```
package stub;

public interface WindowListener {
    void windowOpened();
    void windowClosing();
    void windowClosed();
    void windowIconified();
    void windowDeiconified();
    void windowActivated();
    void windowDeactivated();
}
```

"stub/WindowAdapter.java"

```
package stub;

public class WindowAdapter implements WindowListener {
    @Override
    public void windowOpened() {}

    @Override
    public void windowClosing() {}

    @Override
    public void windowClosed() {}

    @Override
    public void windowIconified() {}

    @Override
    public void windowDeiconified() {}

    @Override
    public void windowActivated() {}

    @Override
    public void windowDeactivated() {}
}
```

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

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.

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

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.

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

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

Long form questions

1. The FACTORY METHOD design pattern.

   The FACTORY METHOD pattern gives us a way to encapsulate the instantiations of concrete types; it encapsulates the functionality required to select and instantiate an appropriate class, inside a designated method referred to as a *factory method*. The factory method selects an appropriate class from a class hierarchy based on the application context and other contributing factors and it then instantiates the selected class and returns it as an instance of the parent class type.

   The advantage of this approach is that the application objects can make use of the factory method to gain access to the appropriate class instance. This eliminates the need for an application object to deal explicitly with the varying class selection criteria.

   You are required to implement the following classes:

   `Product` defines the interface of objects the factory method creates.

   `ConcreteProduct` implements the `Product` interface.

   `Creator` declares the factory method, which returns an object of type `Product`. `Creator` may also define a default implementation of the factory method that returns a default `ConcreteProduct` object. We may call the factory method to create a `Product` object.

   `ConcreteCreator` overrides the factory method to return an instance of a `ConcreteProduct`.

   Factory methods therefore eliminate the need to bind application-specific classes into your code. The code only deals with the `Product` interface (in this case); therefore it can work with any user-defined `ConcreteProduct` classes.

2. The SINGLETON design pattern.

   If you didn't provide implementations of a lazy and eager singleton pattern in Question 6 do so now. (You should provide a static `getInstance` method.

   Imagine that we now wish to use the code in a *multi-threaded environment*. Two threads concurrently access the class, thread `t1` gives the first call to the `getInstance()` method, it will check if the static variable that holds the reference to the singleton instance is `null` and then gets interrupted due to some reason. Another thread `t2` calls the `getInstance()` method successfully passes the instance check and instantiates the object. Then, thread `t1` wakes and it also creates the object. At this time, there would be two objects of this class which was supposedly a singleton.

   (a) How could we use the `synchronized` keyword to the `getInstance()` method to operate correctly.

   (b) The synchronised version comes with a price as it will decrease the performance of the code — why?

   (c) If the call to the `getInstance()` method isn't causing a substantial overhead for your application, then you can forget about it.

   (d) If you want to use synchronisation (or need to), then there is another technique known as *double-checked locking* which reduces the use of synchronisation. With

double-checked locking, we first check to see if an instance is created, and if not, then we synchronise.

Provide a sample implementation of this technique.

There are some other ways to break the singleton pattern:

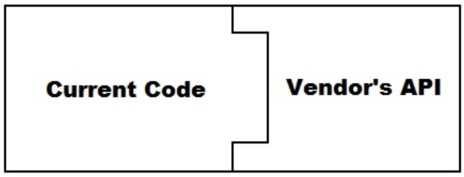
- If the class is `Serializable`.
- If it is `Cloneable`.
- It can be broken by reflection.
- If the class is loaded by multiple *class loaders*.

Try and write a class `SingletonProtected` that addresses some (all?) of these issues.

3. The ADAPTER design pattern.

   A software developer, Max, has worked on an e-commerce website. The website allows users to shop and pay online. The site is integrated with a third party payment gateway, through which users can pay their bills using their credit card. Everything was going well, until his manager called him for a change in the project.

   The manager has told him that they are planning to change the payment gateway vendor, and Max has to implement that in the code. The problem that arises here is that the site is attached to the `Xpay` payment gateway which takes an `Xpay` type of object. The new vendor, `PayD`, only allows the `PayD` type of objects to allow the process. Max doesn't want to change the whole set of a hundred classes which have reference to an object of type `XPay`. He cannot change the third party tool provided by the payment gateway. The problem arises due to the incompatible interfaces between the two different parts of the code. To get the process to work, Max needs to find a way to make the code compatible with the vendor's provided API.



   The current code interface is not compatible with the new vendor's interface. What Max needs here is an ADAPTER which can sit in between the code and the vendor's API, enabling the transaction to proceed.

```
package xpay;

public interface Xpay {
    String getCreditCardNo();

    void setCreditCardNo(String creditCardNo);

    String getCustomerName();

    void setCustomerName(String customerName);

    String getCardExpMonth();

    void setCardExpMonth(String cardExpMonth);

    String getCardExpYear();
```

```
    void setCardExpYear(String cardExpYear);

    Short getCardCVVNo();

    void setCardCVVNo(Short cardCVVNo);

    Double getAmount();

    void setAmount(Double amount);
}
```

The `Xpay` interface contains setter and getter methods to get the information about the credit card and customer name. The interface is implemented in the following code which is used to instantiate an object of this type, and exposes the object to the vendor's API.

```
package xpay;

public class XpayImpl implements Xpay {
    private String creditCardNo;
    private String customerName;
    private String cardExpMonth;
    private String cardExpYear;
    private Short cardCVVNo;
    private Double amount;

    @Override
    public String getCreditCardNo() {
        return creditCardNo;
    }

    @Override
    public void setCreditCardNo(String creditCardNo) {
        this.creditCardNo = creditCardNo;
    }

    @Override
    public String getCustomerName() {
        return customerName;
    }

    @Override
    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }

    @Override
    public String getCardExpMonth() {
        return cardExpMonth;
    }

    @Override
    public void setCardExpMonth(String cardExpMonth) {
        this.cardExpMonth = cardExpMonth;
    }

    @Override
    public String getCardExpYear() {
        return cardExpYear;
    }

    @Override
    public void setCardExpYear(String cardExpYear) {
        this.cardExpYear = cardExpYear;
    }

    @Override
    public Short getCardCVVNo() {
        return cardCVVNo;
    }
```

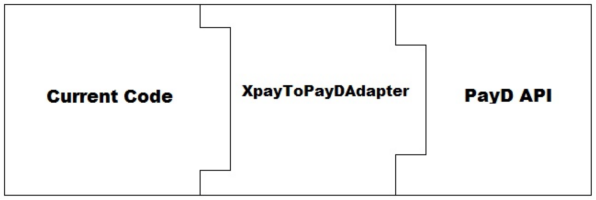
```
    @Override
    public void setCardCVVNo(Short cardCVVNo) {
        this.cardCVVNo = cardCVVNo;
    }

    @Override
    public Double getAmount() {
        return amount;
    }

    @Override
    public void setAmount(Double amount) {
        this.amount = amount;
    }
}
```

New vendor's interface looks like this:

```
package xpay;

public interface PayD {
    String getCustCardNo();

    void setCustCardNo(String custCardNo);

    String getCardOwnerName();

    void setCardOwnerName(String cardOwnerName);

    String getCardExpMonthDate();

    void setCardExpMonthDate(String cardExpMonthDate);

    Integer getCVVNo();

    void setCVVNo(Integer cVVNo);

    Double getTotalAmount();

    void setTotalAmount(Double totalAmount);
}
```

As you can see, this interface has a set of different methods which need to be implemented in the code. However, `Xpay` objects are created by most parts of the code, and it is difficult (and risky) to change the entire set of classes. We need some way, that's able to fulfil the vendor's requirement to process the payment and also make less or no change to the current code base.

Your are required to use the Adapter pattern to implement a `XpayToPayDAdapter` class to meet the requirements.



4. The Observer design pattern.

   Sports Lobby is a sports website targeted at sport lovers. They cover almost all kinds of sports and provide the latest news, information, matches scheduled dates, information about a particular player or a team. Now, they are planning to provide

live commentary or scores of matches as an SMS service, but only for their premium users. Their aim is to SMS the live score, match situation, and important events after short intervals. As a user, you need to subscribe to the package and when there is a live match you will get an SMS to the live commentary. The site also provides an option to unsubscribe from the package whenever a user wants to.

As a developer, the Sport Lobby has asked you to provide this new feature for them. The reporters of the Sport Lobby will sit in the commentary box in the match, and they will update live commentary to a commentary object. As a developer your job is to provide the commentary to the registered users by fetching it from the commentary object when it's available. When there is an update, the system should update the subscribed users by sending them the SMS.

This situation clearly indicates a *one-to-many* mapping between the match and the users, as there could be many users subscribed to a single match. The OBSERVER design pattern is best suited to this situation — you should implement this feature for Sport Lobby using the OBSERVER pattern.

Remember that there are four participants in the OBSERVER pattern:

`Subject` which is used to register observers. Objects use this interface to register as observers and also to remove themselves from being observers.

`Observer` defines an updating interface for objects that should be notified of changes in a subject. All observers need to implement the `Observer` interface. This interface has a method `update()`, which gets called when the `Subject`'s state changes.

`ConcreteSubject` stores the state of interest to `ConcreteObserver` objects. It sends a notification to its observers when its state changes. A concrete subject always implements the `Subject` interface. The `notifyObservers()` method is used to update all the current observers whenever the state changes.

`ConcreateObserver` maintains a reference to a `ConcreteSubject` object and implements the `Observer` interface. Each observer registers with a concrete subject to receive updates.

```
package observer;

public interface Observer {
    void update(String desc);

    void subscribe();

    void unSubscribe();
}
```

```
package observer;

public interface Subject {
    void subscribeObserver(Observer observer);

    void unSubscribeObserver(Observer observer);

    void notifyObservers();

    String subjectDetails();
}
```

```
package observer;

public interface Commentary {
    void setDesc(String desc);
}
```

```
package observer;

import observerpattern.CommentaryObject;
import observerpattern.SMSUsers;

import java.util.ArrayList;

public class TestObserver {

    public static void main(String[] args) {
        Subject subject = new CommentaryObject(new ArrayList<Observer>(), "Soccer Match [2014AUG24]");
        Observer observer = new SMSUsers(subject, "Adam Warner [New York]");
        observer.subscribe();

        System.out.println();

        Observer observer2 = new SMSUsers(subject, "Tim Ronney [London]");
        observer2.subscribe();

        Commentary cObject = ((Commentary) subject);
        cObject.setDesc("Welcome to live Soccer match");
        cObject.setDesc("Current score 0-0");

        System.out.println();

        observer2.unSubscribe();

        System.out.println();

        cObject.setDesc("It's a goal!!");
        cObject.setDesc("Current score 1-0");

        System.out.println();

        Observer observer3 = new SMSUsers(subject, "Marrie [Paris]");
        observer3.subscribe();

        System.out.println();

        cObject.setDesc("It's another goal!!");
        cObject.setDesc("Half-time score 2-0");

    }

}
```

5. The DECORATOR design pattern.

   You are commissioned by a pizza company make an extra topping calculator. A user can ask to add extra topping to a pizza and our job is to add toppings and increase its price using our classes.

   Please note: the main aim of the DECORATOR design pattern is to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality. The Decorator prevents the proliferation of subclasses leading to less complexity and confusion.

   For simplicity, let's create a simple `Pizza` interface which contains only two methods:

```
package decorator;
```

```
public interface Pizza {
    String getDesc();

    double getPrice();
}
```

The `getDesc` method is used to obtain the pizza's description whereas the `getPrice` is used to obtain the price.

Provide two implementations of the `Pizza` interface:

- `SimplyVegPizza`
- `SimplyNonVegPizza`

The decorator wraps the object whose functionality needs to be increased, so it needs to implement the same interface. Provide an abstract decorator class which will be extended by all the concrete decorators.

```
  public abstract class PizzaDecorator implements Pizza
```

Now provide several implementations of `PizzaDecorator` and exercise your classes with the given test class.

- `Ham extends PizzaDecorator`
- `Cheese extends PizzaDecorator`
- `Chicken extends PizzaDecorator`
- `FetaCheese extends PizzaDecorator`
- ...

```
package decorator;

import java.text.DecimalFormat;

public class TestDecoratorPattern {
    private static DecimalFormat dformat;

    static {
        dformat = new DecimalFormat("#.##");
    }

    public static void main(String[] args) {
        Pizza pizza = new SimplyVegPizza();
        pizza = new RomaTomatoes(pizza);
        print(pizza);

        pizza = new GreenOlives(pizza);
        print(pizza);

        pizza = new Spinach(pizza);
        print(pizza);

        pizza = new SimplyNonVegPizza();
        print(pizza);

        pizza = new Meat(pizza);
        print(pizza);

        pizza = new Cheese(pizza);
        print(pizza);

        pizza = new Ham(pizza);
```

```
        print(pizza);
    }

    private static void print(Pizza pizza) {
        System.out.println("Desc: " + pizza.getDesc());
        System.out.println("Price: " + dformat.format(pizza.getPrice()));
    }
}
```

The above code will result in the following output:

```
Desc: SimplyVegPizza (230), Roma Tomatoes (5.20), Green Olives (5.47), Spinach (7.92)
Price: 248.59
Desc: SimplyNonVegPizza (350), Meat (14.25), Cheese (20.72), Cheese (20.72), Ham (18.12)
Price: 423.81
```