From: **Alan Liang** alan.5100@yahoo.com
Subject: that lab tho
Date: October 20, 2016 at 2:52 PM
To: Winnie Liang winnieliang27@yahoo.com

lab08

lab08

Due tonight at 11:59 pm

All in package `lab08`

We'll be writing some playing-card based code today to get more practice with **o**bject-**o**riented **p**rogramming (*OOP*). We'll write the similar code in two different styles to compare and contrast them.

Card.java

Make a java class `Card`. Within `Card` create an enum `Suit` with 4 members:

```
public static enum Suit { Hearts, Diamonds, Clubs, Spades };
```

Enum stands for enumeration which Java uses to declare a new type where a variable of that type must have one of the values in the list. A Student's status (one of Freshman, Sophomore, ...) is another good candidate to be an enum.

`Card` has two private fields, a `Suit` named `suit` and a `int` named `value`.

Provide a simple constructor that initializes the two fields from the arguments. If `value` is out of range (either < 2 or > 11), throw an IllegalArgumentException with an error message.

Provide getters and setters for both fields. Remember, eclipse can do this all for you.

Now to override some methods. In Java, by default, a class that does not subclass any class will always have the Object class as its superclass. So when we override toString, we are overriding Object's toString which is to print the Object's memory location. Click on the link to see the javadocs for Object's other methods.

Override `public boolean equals(Object other)`. This tests the passed in object for equality with this object, returning true of false. Why is the type of `other` `Object` and not `Card`? What if we wanted to test a Card and String for equality? We must first check if `other` is a `Card`. If it is not, return false. To check if an object `o` is an instance of some class `Class`, you would use the `instanceof` operator: `o instanceof Class` which returns a boolean. If other is an instance of Card, declare a `Card` variable, initialized to `(Card)other`. This casts `other` to a `Card`. This is safe because we ensured in the if statement that other is an instance of `Card`. Now that we have two Card objects, they're equal if and only if their values are equals and their suits are equal.

Override the `hashCode` method (`public int hashCode()`). Whenever we override equals, it is a good idea to override hashCode too. `hashCode` strives to return a fairly unique integer to quickly characterize an object. Typically, hashcode is written as some combination of the hashcodes of its fields. We multiply by object. Typically, hashcode is written as some combination of the hashcodes of its fields. We multiply by primes numbers to better spread a class's hash codes. Have this hashCode method return `31 * suit.hashCode() + 7 * Integer.hashCode(value)`.

Override the `toString` method (`public String toString()`). Create an empty string. If the `value` is 11, append "Ace", else append `value`. Append the string " of " and `suit.toString()` as well, then return that string.

Write a main method inside Card that tests your class by creating several cards of varying values and suits and printing them. Since `Suit` was created inside the `Card` class we must refer to a `Suit` value as `Suit.Spades`, etc...
Note that a better `Card` class would distinguish between 10 and face cards but this is good enough for this lab's purposes.

CardHand.java

Recall abstract classes are good for creating base classes with some methods but also some unimplemented methods we want to force subclasses to implement. Abstract classes cannot be instantiated, they are merely blueprints.

Make an **abstract** java class `CardHand`. It has one member `cards`, instantiated to an `ArrayList` of `Card`. Leave `private` off so subclasses can access it. No constructor.

Write `public` methods `void addCard(Card c)`, `void clear()`, and `String toString()` that delegate to calling the corresponding `ArrayList` method on `cards`. Now for the OO - create an `abstract` method, `public abstract int value()`. Abstract methods have no body (`{ ... }`), just put a semi-colon (`;`). Any class that subclasses `CardHand` either has to itself be abstract or implement the `value` method.

BlackjackHand.java

Make a class `BlackjackHand` which extends (subclasses) `CardHand`. It is going to implement the `value()` method to return a hand value according to blackjack logic. The method should return the largest value under 21, 21, or the smallest value over 21. This is complicated by the fact that Aces can have a value of 1 or 11. Have the method only return 0 for now.

Before you implement `value()`, write a JUnit test case class `TestValue` that tests the following cases. Each case (bulletpoint) needs its own method. Why? Run your tests and make sure they all fail (because you haven't implemented value correctly yet). Now implement `value` (see hints below), testing as you go. Repeat until all tests pass. This is called **TDD** (test-driven development).

- [2, 4] -> 6
- [Ace, 10] -> 21
- [Ace, Ace] -> 12 (11 + 1 since 11 + 11 is over 21)
- [Ace, Ace, Ace] -> 13 (11 + 1 + 1)
- [10, 10, 2] -> 22
- [10, 10, Ace] -> 21

Hints for writing `value`:

- calculate the sum of all non-ace cards and count how many aces there are

- add all the ace values to the non-ace sum using the fact that an ace should count as 11 if adding it would not cause the sum to exceed 21, otherwise it should count as a 1.

While the logic would be more complicated, "all" that would be needed to plug-in the ability for Texas Hold'em/Bridge... hands would be to subclass `CardHand` and override `value()` with the appropriate logic. By subclassing, we inherit all the common boilerplate for "free".

ComposedHand

Extension (subclassing) is one aspect of object-oriented. When one class, like a BlackjackHand, extends another class, like a CardHand, we say that a BlackjackHand *is a* CardHand (with the further traits of a BlackjackHand).

On the other hand, we have composition, where one class holds an instance of another class as a field. In this case, one class *has a* other class. In last week's text editor lab, FileBuffer was a Buffer, while Editor had a Buffer.

Let's make some tweaks to our code to see what composition can do for us.

Create a new regular class `ComposedHand`. Copy the inside fields and methods of `CardHand` over into this class. Add a `private` field, named `method` of type `ComputationMethod`.

Create a constructor that takes a `ComputationMethod` argument and saves it in `method`.

A `ComposedHand` will be composed of the method it uses to compute the value. We are extracting the computation logic into its own class instead of being "trapped" in a subclass.

Remove the `abstract` modifier of `value()` and make the body simply

```
return method.compute(cards);
```

ComputationMethod

Make an **abstract** class `ComputationMethod` with one method `public abstract int compute(List<Card> cards)`.

BlackjackMethod

Make a class, `BlackjackMethod` that extends `ComputationMethod`. In order to subclass `ComputationMethod`, we must implement the methods it demands, namely compute.

To implement `compute`, copy your implementation of `BlackjackHand`'s `value` method over. Note that the two methods are slightly different - one holds `cards` internally because its in the same class, while one is given `cards` as a parameter.

If we wanted other ways of valuing hands, we'd have another class extend `ComputationMethod` with its own implementation, i.e. StudPokerMethod, HoldEmMethod.

Driver

Create a `Driver` class with a main method. Create a `BlackjackHand` and add a few cards to it. Print the hand's value.

Next, create a `ComposedHand`. Since `ComposedHand` is composed with (has a) `ComputationMethod`, we'll need an instance of a subclass of `ComputationMethod` to give to our `ComposedHand`. Create a `BlackjackMethod` object and pass that to the `ComposedHand`'s constructor. Add the same cards you added to your `BlackjackHand` to your `ComposedHand`. Print the hand's value.

Run your program to verify you they both give the same, correct number.

Discussion

We've seen two different ways of setting up and organizing our code in a object-oriented manner. We code specific logic in specific classes that inherit from general classes.

1. Extension (subclassing) is a more static, permanent way of customization
2. composition allows us to dynamically change behavior during runtime. In being dynamic, we could change the `ComputationMethod` of a `ComposedHand` during runtime to be some sort of other subclass of `ComputationMethod`.

At the end of the day, good programming style, regardless of programming language, is about separating what stays the same from those things that change.

In this lab's context a Card stays the same but the way we value those cards changes, so we have separated the logic in two different ways as well as making it easy to add new evaluations. Object-oriented programming achieves this partially through classes and subclasses.