Card Classes II

Purpose

To continue exploring class related features: `enums`, inheritance, overriding `Object` class methods, and recursion as begun in Lab 4.

Due Date

The completed lab assignment is due by 15:30 Friday, 2023-02-17.

More enum types

Additional functions

It is possible to add member functions. For instance, in cribbage, each card has a count, which is the face value, except that jacks, queens, and kings all count as 10.

```
public enum Rank {
    Ace, Two, Three, /* ... */ Ten, Jack, Queen, King ;

    public int count() { return Math.min(ordinal()+1,10) ; }
    // ordinal is inherited.  It starts at 0 for Ace, ...
}
```

It is even possible to add state (private member variables) to an `emum` class. See Appendix A of the book for details.

More finite classes: two `Card` classes

Here is where actual code requirements start.

⇒ Reuse your `public enum Rank` from last lab, but add methods `public int count()` (or `getCount()` if you prefer), and `public String toSthortString()` (that returns a length 1 string).

⇒ Reuse your `public enum Suit` from last lab, but add method `public String toSthortString()` (that returns a length 1 string).

⇒ Create an immutable `Card` class with the same interface as Lab 4. However, override and overload the `.equals` methods with signatures

```
@Override
public boolean equals(Object o) { ... }

public boolean equals(Card c) { ... }
```

Make the former call the latter when appropriate.

⇒ In a parallel package re-implement your `Card` class to use a different internal state, but the same interface (like Lab 3). For instance, consider using a single byte as your state.

Faking enums, another `Card` class

If the `satic Card getCard` methods are public, but the constructors are private, we can control the production of `Card`s, and ensure that there are only 52 `Card` objects. We need two ideas: an array of values

```
private static Card [] theCards = new Card [52] ;
```

Next we use our `getCard()` methods to lazily initialize the array, as in

```
public Card getCard(int i) {
  if (theCards[i]==null) { theCards[i] = new Card(i) ; }
  return theCards[i] ;
  }
```

A variant of this technique is known as the *singleton pattern*, and is used when want to have a class with exactly one object.

⇒ In another parallel package re-implement your `Card` class using this idiom. Comment in you code on whether overriding the `.equals(...)` method is necessary in this case.

Testing

⇒ Write methods that

(a) count the number of pairs (pairs of cards with the same rank, for instance Q and Q♡) in an array list of cards;
(b) determine whether all of the cards are in the same suit.

⇒ Write a program that loops 40 times. For each loop, it picks a random combination of 5 cards; prints them, prints the number of pairs, and prints whether or not the cards are a flush. Output should look something like:

```
5C 6H 5S 5D 6H : 4 pair(s), no flush
3C 8H QC AC 9D : 0 pair(s), no flush
5C JC AC 2C 3C : 0 pair(s),    flush
...
```

Run this test on all three of your card classes.