Project 3: ¿Cuál es mejor?

Objectives:

- Learn Java syntax equivalents to Python constructs you already know
- Practice how different implementations can affect efficiency
- Practice with built-in Java classes

Overview

You're going to redo Project 2 in Java. Starter code for IntelliJ is in the shared Google drive folder. From the user's perspective, the game will play exactly the same as before. But there are two primary changes:

1. Change the implementation of a card so it uses two (**and only two**) instance variables – an int for the rank (2-14) and a string for the suit where the suit will be fully spelled out, like "Spades". You might have already done this in the previous project, but if you had more than 2 instance variables, reduce to just these 2 -- from which other information can be computed, of course.
2. Change the implementation of a deck so we can deal cards *without* removing them from the internal representation of the deck. This will improve the efficiency of several methods like `deal` and `gather`. Details about how to do this are in the "Dealing without Removing" section below.

Translate to Java

As you write your Java classes, remember to use Java naming conventions (camelCase for multi-word variables, methods, and parameters, CamelCase for multi-word classes, and ALL_CAPS for constants) and Javadoc formatted comments for all public methods and all classes, including class descriptions.

Here are the required classes and methods:

- **Card** which models a single playing card. It should have
    - a non-default constructor which takes a rank (int) and suit (String) as parameters, in that order
    - getter methods for getting the rank and suit
    - `toString` method to return the "Jack of clubs" readable version of the card
- **Deck** which models a deck of cards. Implement it using an ArrayList<Card> for the internal representation. It should have:

- a default constructor
- a `shuffle` method. You need to write this one yourself instead of depending on a built-in method. Do it by iterating through the ArrayList and for each index, swap the Card at that index with one from a random index. Look up the `nextInt` method in the **ThreadLocalRandom** class for an easy way of getting random numbers. **PITFALL ALERT**: Read the summary at the top of the Javadocs for how to call it.
- a `deal` method which returns the next undealt card or null if deck is empty
- a `size` method that returns the number of *undealt* cards in the deck
- a `gather` method which returns the deck to a state where all cards are undealt. It does **not** have to be in an unshuffled state.
- a `toString` method which returns all the *undealt* cards in the deck as a string

- **PokerHand** which models a 5-card hand of cards. It should contain
  - the following non-default constructor:

    ```
    public PokerHand(ArrayList<Card> cardList)
    ```

    where `cardList` is the list of cards that should be in the hand. Be sure to copy this list instead of just storing a reference to it.
  - an `addCard` method that will take a Card object as a parameter. It will add that card to the hand. Be robust. Nothing should happen if the hand already has 5 cards in it.
  - a `getCard` method that will take an index (int >= 0) as a parameter. It will return the Card object at that index. Return null if index is invalid.
  - a `toString` method
  - a `compareTo`, which compares this PokerHand to another PokerHand. Here's the required signature:

```
/**
 *  Determines how this hand compares to another hand, returns
 *  positive, negative, or zero depending on the comparison.
 *
 *  @param other The hand to compare this hand to
 *  @return a negative number if this is worth LESS than other, zero
 *  if they are worth the SAME, and a positive number if this is worth
 *  MORE than other
 */
public int compareTo(PokerHand other)
```

Dealing without Removing

Up until now, your Deck has been implemented as a Python list where the `deal` method probably called the built-in list method `pop` to return and remove a card from the deck. The length of the list therefore represents the number of cards in the deck.

This internal representation is simple and easy to implement. But it has one big drawback: when the `pop` method removes the card at index 0, it shifts all the other cards down by one index so that index 0 now points at what used to be the second card in the deck. It does this so that index 0 will always be pointing to the first thing in the list (as all lists should). This means `pop` has to do some work with all *n* cards in the deck, and that work increases linearly as *n* increases. Thus, the `pop` method has a linear (i.e. O(n)) running time.

We can improve this by changing the internal implementation of the deck *without* changing the externally visible behavior. Your deck will now have two instance variables:

- an ArrayList of 52 Card objects, from which we will never remove any cards.
- an int named `nextToDeal` that should be initialized to zero. This variable indicates which index in the ArrayList is the next to be dealt. Thus, it also acts as a dividing line between those cards already dealt and those yet to be dealt.

Rewrite your Deck class using this implementation. Note that this affects many Deck methods, not just `deal`. The `shuffle` method, for example, cannot just shuffle everything in the ArrayList now because some of those cards may have already been dealt! Make sure that all of the public methods still behave correctly.

Don't miss the big picture: calculate the efficiency of `deal` in Big-O notation after you're done. Is it better than O(n)?

Testing

Using the Testing class (already in the starter code), create unit tests in a class called `PokerComparisonTests` in `PokerComparisonTests.java`. This class should have a main method (public static void main(String[] args)) that calls other static methods, each of which tests one method. You should have, at minimum, tests of `compareTo`, but I encourage you to have additional tests for other public methods. Remember, write tests first and test as you go, not at the end!

In addition, you'll be using Gradescope that lets you submit your .java files (in this case, Card, Deck, and PokerHand) at any time and run *my* unit tests for compareTo. Your score for output will come directly from your Gradescope score, so submit early and often. You can submit to Gradescope as

often as you like, but whatever your score is when the assignment deadline hits, that's the points you've earned.

Grading

This project is worth 100 points. Here's the breakdown:
20 points for a working `compareTo` method. This score comes directly from Gradescope.
15 points for thorough unit testing
65 points for design including, but not limited to, information hiding, constants, private helper methods, complete Javadocs, meaningful names, good indentation, robust code where indicated above, good modularity, and having all required methods described in this document.

Turning it in

1. Put the honor code affirmation in PokerComparisonTests.java:

   I affirm that I have carried out the attached academic endeavors with full academic honesty, in accordance with the Union College Honor Code and the course syllabus.

2. Upload all of your .java files to gradescope (don't upload a zip file --- that will likely cause problems).

Gentle Reminder

Programming projects are *individual* projects. I encourage you to talk to others about the general nature of the project and ideas about how to pursue it. However, the technical work, the writing, and the inspiration behind these must be substantially your own. You must cite anyone else who contributes in any way to the project by adding appropriate comments to the code. Similarly, if you include information that you have gleaned from other sources, you must cite them as references. Looking at, and/or copying, other people's code is inappropriate, and will be considered an honor code violation.