|  |  |
| --- | --- |
| **CMSC 131** |  |
| **Sections 020X/040X** |

Assignment:     Project #5

Due Date:     Thursday 11/08, 11:00PM

Open/Closed policy:     CLOSED

**Poker Simulator**

****

**This is a closed project. Please review the** [**course policy on open/closed projects**](http://docs.google.com/openClosedPolicy.shtml) **before you begin.**

Overview

This project will simulate something close to a game of Texas Hold 'Em, which is a style of poker.  If you don't know how to play this game (or if you've never played cards before in your life), don't worry -- we'll explain everything!  Most of this program will be written for you.  We will provide the graphical interface and most of the infrastructure for the project.  You will supply two classes (and a series of JUnit tests).  The classes you will write are:  **Deck**, which represents a deck of cards, and  **PokerHandEvaluator**, which contains several static methods that evaluate a set of 5 cards to determine which "poker" hands are represented..

Cards

If you are not familiar with the standard deck of 52 playing cards and the terminology that goes along with it ("suit", "queen", "clubs", etc.), then please take a look at the [Playing Card Page](http://docs.google.com/playingCard.html).

Poker Hands

If you are not familiar with poker hands, such as "Full House" or "Straight Flush", then please read the [Poker Hand Page](http://docs.google.com/pokerHands.html).  By the way, to successfully implement this project you need to know the definitions of these hands very precisely, so unless you are a real expert it would be a good idea to read this information carefully!

(Partial) Rules of Texas Hold 'Em

If you are not familiar with this game, then please read the [Introduction to Texas Hold 'Em Page](http://docs.google.com/texasHoldEm.html).

Card Class (Provided)

Before you get started, you should read the [JavaDoc for the Card class.](http://docs.google.com/doc/index.html)  You will use this class while writing your classes.

Classes You Must Implement

There are two classes you must implement.  (You will also write a series of JUnit tests, described below.)  We have given you just "skeletons" for the classes to get you started.  The two classes you will write are:

### Deck.java

The Deck class represents a deck of cards.  It starts off with 52 cards, but as cards are dealt from the deck, the number of cards becomes smaller.

The class has one private instance variable that stores the Cards that are currently in the Deck:

private Card[] cards;

**Important Note:  The *first* Card in the array will be considered the card that is at the *top* of the deck.**  When cards are removed from the deck they will be removed from the *front* of the array.

**You must implement the following methods for the Deck class:**

* public Deck() -- This constructor initializes the Deck with 52 card objects, representing the 52 cards that are in a standard deck.  The cards must be ordered as in the diagram below:



**Important:  The FIRST card in your array should be the Ace of Spades, and the LAST card should be the King of Diamonds.  Note that in the picture above, the Ace of Spades is at the TOP of the deck, which is to the left.**

* public Deck(Deck other) -- Standard copy constructor.  Note that it is okay to make a shallow copy of the array of cards.  (The Card class is immutable, so aliasing Cards is not a problem.)

* public Card getCardAt(int position) -- Returns the card that is at the specified position in the array.  (Uses the usual 0-based indexing.)

* public int getNumCards() -- Returns the size of the array of Cards.  (It starts off equal to 52, but becomes smaller as cards are dealt from the deck.)

* public void shuffle() -- This method will re-arrange the cards that are in the deck.  The idea is that the deck will be divided into two "packets" -- the top half and the bottom half.  The new array of cards will consist of:  the first card from the top packet, followed by the first card from the bottom packet, followed by the second card from the top packet, followed by the second card from the bottom packet, etc.**Important:  If there are an odd number of cards, the top packet should have one more card than the bottom packet.** Remember that the top of the deck is considered to be the *front* of the array. In the pictures below, remember that the top of the deck appears at the left side.

Below  are two examples of how the shuffle should work:

**Example 1 (even number of cards):**

**Before Shuffling**

****

**After Shuffling**

****

**Example 2 (odd number of cards):**

**Before Shuffling**

****

**After Shuffling**

****

* public void cut(int position)-- This method divides the deck into two subpackets:  The part *above* the specified position, and the part that is at the specified position or below.  The two subpackets are reversed (the top packet is placed on the bottom and the bottom packet is placed on the top.)  The position value uses 0-based indexing as usual, so the card at the top of the deck (to the left in the diagrams) is at position 0.  Here is an example:

**Before cut**

****

**After cutting at position 4.**  (The 5 of spades was at position 4 in the diagram above.)



* public Card[] deal(int numCards)-- This method will remove the specified number of cards from the top of the deck and return them as an array.  For example, if the parameter is 4, then the first four cards in the deck will be returned as an array of size 4.  Important**:  The cards will be removed from the front of the "cards" array, not the back.**   
  **Hint:**  The "cards" array will need to be resized.  Here is the idea:
  + Make a new array (call it "smaller") that is the same size as the current deck of cards minus the number of cards being dealt.
  + Copy the cards that are not being dealt from the original array into the new one.
  + Assign the instance variable "cards" so that it refers to the new array.

**PokerHandEvaluator.java**

This class consists of several static methods that you will write.  The prototypes for the methods are:

* public static boolean hasPair(Card[] cards)
* public static boolean hasTwoPair(Card[] cards)
* public static boolean hasThreeOfAKind(Card[] cards)
* public static boolean hasStraight(Card[] cards)
* public static boolean hasFlush(Card[] cards)
* public static boolean hasFullHouse(Card[] cards)
* public static boolean hasFourOfAKind(Card[] cards)
* public static boolean hasStraightFlush(Card[] cards)

The parameter for each of these methods will be an array of exactly 5 cards.  Each method will return true or false, based on whether or not the given set of cards satisfies the poker hand being evaluated in the method.   For example, the hasTwoPair method will return true if the set of cards has two pairs of different values (e.g. a pair of 4's and a pair of Jacks).  If you need to review the various "poker hands" being tested in the methods, please take another look at the [Poker Hand Page](http://docs.google.com/pokerHands.html).

**Important:** Each of these methods checks whether or not the set of cards satisfies the given poker hand, but it does not care if it could also satisfy a *better*  hand.  For example, if the current hand is

<Ace of diamonds, Ace of spades, Ace of hearts, Jack of Spades, Jack of Diamonds>

then the results of calling each method should be as indicated below:

* hasPair -- true  (there is at least one pair)
* hasTwoPair -- true (there are two pairs of distinct values)
* hasThreeOfAKind -- true (there are three Aces)
* hasStraight -- false
* hasFlush -- false
* hasFullHouse -- true (there are three Aces and two Jacks -- that makes a "full house")
* hasFourOfAKind -- false
* hasStraightFlush -- false

**Important:**  In order for a hand to qualify as "Two Pair", it must have two pairs of *distinct* values -- in particular, having four-of-a-kind does *not* also count as two pair.  For example, if the current hand is

<2 of diamonds, 2 of spades, 2 of hearts, 2 of clubs, 9 of spades>

then the results of calling each method should be as indicated below:

* hasPair -- true  (there is at least one pair)
* hasTwoPair -- false (there are *not* two pairs of distinct values)
* hasThreeOfAKind -- true (there are three 2's)
* hasStraight -- false
* hasFlush -- false
* hasFullHouse -- false
* hasFourOfAKind -- true (there are four 2's)
* hasStraightFlush -- false

JUnit Tests

Be sure to put your JUnit tests into the class we have provided called "StudentTests.java". You should be using JUnit 4 (not JUnit 5, also known as "Jupiter").

The names of the Release tests on the submit server are intentionally ambiguous (testHandEvaluator1, testHandEvaluator2, etc.) We want to let you know whether or not your code is working, but you will need to rely on your own JUnit testing to find cases where your code does not behave correctly. Don't bother asking the TAs which kind of hands are involved with each of these tests -- they don't know either!

Be sure that for each method you have written some tests where the method returns "true", and some tests where the method returns "false".

Requirements

* As with all programming assignments:
  + You must use meaningful variable names and good indentation.
  + You must avoid code duplication, by calling appropriate methods (rather than cutting and pasting code). You may define your own private utility methods to perform often repeated tasks.
  + Because we are using an automatic testing system, it is important the you adhere closely to the specifications.
* You may not use any of the Java collections classes (including ArrayList, LinkedList, etc.)  If you don't know what these are, don't worry about it.

Running the Simulation

**To run the simulation, execute the main method in the Driver.java class.** Before clicking on the "Deal" button, you will usually want to do a series of "cuts" and "shuffles".  (You can cut the deck by clicking somewhere in the middle of the deck.  Shuffle is accomplished by clicking a button.)

Grading

Your grade will be computed as follows:

* Style:  10%
* Release Tests:  90%

[Web Accessibility](https://www.umd.edu/web-accessibility)