Objective:

This lab provides additional practice in writing classes. You will encounter a class that contains a collection of objects of another class you write. As we have done in previous labs, JUnit tests are used to test the code developed in the lab.

Introduction:

The activities of this lab are related to a simple solitaire card game called Elevens. You will learn the rules of Elevens, and will be able to play it by using the supplied Graphical User Interface (GUI). You will learn about the design and the Object Oriented Principles that suggested that design. In this lab you will begin to implement some of the code behind the game.

Instructions:

1. Set up your Eclipse environment by creating a new project, say Elevens or Lab7. Let’s start by playing the game. Copy the file elevens.jar into your project (do not place it in the src folder or the default package).

Here are the rules of the game:

The game of Elevens uses a deck of 52 cards, with ranks Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King, and suits Clubs, Diamonds, Hearts, and Spades. Here is how it is played.

1. The deck is shuffled, and nine cards are dealt “face up” from the deck to the board.
2. Then the following sequence of steps is repeated.
   a. The player removes each pair of cards (A, 2, . . ., 10) that total 11, e.g., an 8 and a 3, or a 10 and an A. An ace is worth 1, and the suits are ignored when determining cards to remove.
   b. Any triplet consisting of a J, a Q, and a K is also removed by the player. Suits are also ignored when determining which cards to remove.
   c. Cards are dealt from the deck if possible to replace the cards just removed.

(Note: This lab is taken from College Board materials for the AP Computer Science A course. This is one of the exemplar labs that is provided for this course.)
The game is won when the deck is empty and no cards remain on the table. Go ahead and play the
game several times by double-clicking the .jar file.

2. In the first activity you will write a Card class that will be used to represent card objects. In the
implementation of the Card class you will be required to include:
   - Three instance variables, one to store the card’s rank, one to store the card’s suit, and one to
     store the card’s point value.
   - A constructor that takes two String parameters that represent the card’s rank and suit, and an int
     parameter that represents the point value of the card.
   - Accessor methods for the card’s rank, suit, and point value.
   - A toString() method that uses the format:
     
     
     
     .

   - An equals() method to test equality between two card objects.

     Document the class with javadoc style comments that include a brief overview of the class, your
name(s), and the date. Document each method with javadoc style comments that include a brief
description of the method and any parameters or returns. Print the source code for your Card class
and hand it in with your lab.

3. Create a JUnit test class. Write tests to test the accessors, toString(), and equals(). Include at least six
tests.

   Run the tests. (You will need to fix the initial errors by adding JUnit 4 to the build path.) You should
get a green bar. Add a snapshot of this green bar to your lab report.

   Document the file as a whole and each test using Javadoc style. Print the source code for your Card
tester class and hand it in with your lab.

4. Now you will write the Deck class. A deck will be represented by a list of cards and an integer value
storing the size of the deck. So for instance variables you should have:

   private List<Card> cards;
   private int size;

   You will write a constructor, an isEmpty() method, a size() method, a deal() method, a
shuffle() method, and a toString() method.
Write a constructor for the `Deck` class that receives three arrays as parameters. The arrays contain the ranks, suits, and point values for each card in the deck. The constructor creates an ArrayList and then creates the specified cards and adds them to the list. For example, if `ranks = {"A", "B", "C"}`, `suits = {"Giraffes", "Lions"}`, and `values = {2, 1, 6}`, the constructor would create the following cards:

```
["A", "Giraffes", 2], ["B", "Giraffes", 1], ["C", "Giraffes", 6],
["A", "Lions", 2], ["B", "Lions", 1], ["C", "Lions", 6]
```

and would add each of them to `cards`. The parameter `size` would then be set to the size of `cards`, which in this example is 6.

Finally, the constructor should shuffle the deck by calling the `shuffle` method.

Write a `isEmpty()` method that returns true when the size of the deck is 0; false otherwise.

Write a `size()` method that returns the number of cards in the deck that are left to be dealt.

Write a `deal()` method that “deals” a card by removing a card from the deck and returning it, if there are any cards in the deck left to be dealt. It returns null if the deck is empty. There are a couple of ways of accomplishing this task. Here are two possible algorithms:

**Algorithm 1:** Because the cards are being held in an ArrayList, it would be easy to simply call the List method that removes an object at a specified index, and return that object. Removing the object from the end of the list would be more efficient than removing it from the beginning of the list. Note that the use of this algorithm also requires a separate “discard” list to keep track of the dealt cards. This is necessary so that the dealt cards can be reshuffled and dealt again.

**Algorithm 2:** It would be more efficient to leave the cards in the list. Instead of removing the card, simply decrement the `size` instance variable and then return the card at `size`. In this algorithm, the `size` instance variable does double duty; it determines which card to “deal” and it also represents how many cards in the deck are left to be dealt. **This is the algorithm that you should implement.**

Write a `shuffle()` method that does nothing at this point. A later exercise will have you write the shuffle method.

5. Create a JUnit test class to test the `Deck` class. Write tests to test the methods. (Hint: test the class on a small deck – not a 52-card deck.) Include at least six tests.

Run the tests. You should get a green bar. **Add a snapshot of this green bar to your lab report.**

Document the file as a whole and each test using Javadoc style.

6. Now we address how to shuffle a deck. Think about how you shuffle a deck of cards by hand. How well do you think it randomizes the cards in the deck?

A **shuffling** of a deck is a **permutation** of its cards into a random-looking sequence. A requirement of a good shuffling procedure is that any particular permutation has just as much chance of occurring as
any other. We will consider one shuffling algorithm here, called the *perfect shuffle*. A perfect shuffle involves splitting the deck in half and then interleaving the two half-decks by alternating cards from the two half-decks to form the shuffled deck. (Unfortunately, the perfect shuffle comes nowhere near generating all possible deck permutations. In fact, eight perfect shuffles of a 52-card deck in succession returns the deck to its original state!)

Here is the “perfect shuffle” algorithm that starts with an array named *cards* that contains 52 cards and creates an array named *shuffled*.

1. **Initialize** *shuffled* to contain 52 “empty” elements
2. Set *k* to 0
3. For *j* = 0 to 25,
   - Copy *cards*[j] to *shuffled*[k],
   - Advance *k* by 2.
4. Set *k* to 1
5. For *j* = 26 to 51,
   - Copy *cards*[j] to *shuffled*[k],
   - Advance *k* by 2.

This algorithm moves the first half of *cards* to the even index positions of *shuffled*, and it moves the second half of *cards* to the odd index positions of *shuffled*. The above algorithm shuffles 52 cards and can be easily modified to shuffle any deck with an even number of cards. If an odd number of cards is shuffled, the array to be shuffled has one more even-indexed position than odd-indexed positions. Therefore, the first loop must copy one more card than the second loop does. This requires rounding up when calculating the index of the middle of the deck.

Write the code for the `shuffle()` method using the perfect shuffle algorithm.

Add two tests to your deck tester. One that tests shuffling a deck with an even number of cards and one that tests shuffling a deck with an odd number of cards. *Once more, take a screen snapshot of the green bar that you get.*

Print the source code for your Deck class and the source code for your Deck tester class and hand them in with your lab.

7. **Extra credit.**
   A more interesting and effective shuffling algorithm is the “efficient selection shuffle”. For extra credit, code the efficient selection shuffle.

   First, consider the algorithm that starts with an array named *cards* that contains 52 cards and creates an array named *shuffled*. We will call this algorithm the “selection shuffle”.
Initialize shuffled to contain 52 “empty” elements
Then for \( k = 0 \) to 51,
  - Repeatedly generate a random integer \( j \) between 0 and 51, inclusive until \( cards[j] \) contains a card (not marked as empty),
  - Copy \( cards[j] \) to \( shuffled[k] \)
  - Set \( cards[j] \) to empty.

This approach finds a suitable card for the \( k \)th position of the deck. Unsuitable candidates are any cards that have already been placed in the deck. While this is a more promising approach than the perfect shuffle, its big defect is that it runs too slowly. Every time an empty element is selected, it has to loop again. To determine the last element of shuffled requires an average of 52 calls to the random number generator (since only one cell of \( cards \) remains non-empty).

A better version, the “efficient selection shuffle”, works as follows:

For \( k = 51 \) downto 1,
  - Generate a random integer \( r \) between 0 and \( k \), inclusive,
  - Exchange \( cards[k] \) to \( cards[r] \).

It may be helpful to think through how this is working. It really is a clever algorithm. It is quite a bit more efficient than the previous algorithm.

Code this algorithm in the shuffle() method. Rerun your JUnit tests. Print the source file for the Deck class and hand it in with your lab.

**Hand in:**

The write-up you hand in for this lab should include:
- output (snippets) from the tests in Steps 3, 5, 6.
- a printed copy of the (4)classes as requested in Steps 2, 3, 6.
- a copy of the Deck class from Step 7 if you are doing the extra credit.

**Help Policy:**

Help Policy in Effect for This Assignment: Group Project with Limited Collaboration

In particular, you may discuss the assignment and concepts related to the assignment with the following persons, in addition to an instructor in this course: any member of your group; any St. Bonaventure Computer Science instructor; and any student enrolled in CS 132.

You may use the following materials produced by other students: materials produced by members of your group.