Welcome to the Elevens project. In this exercise you will be building a complete multi-class game based around the solitaire card game "Elevens".

This is another "Pair Programming" Assignment

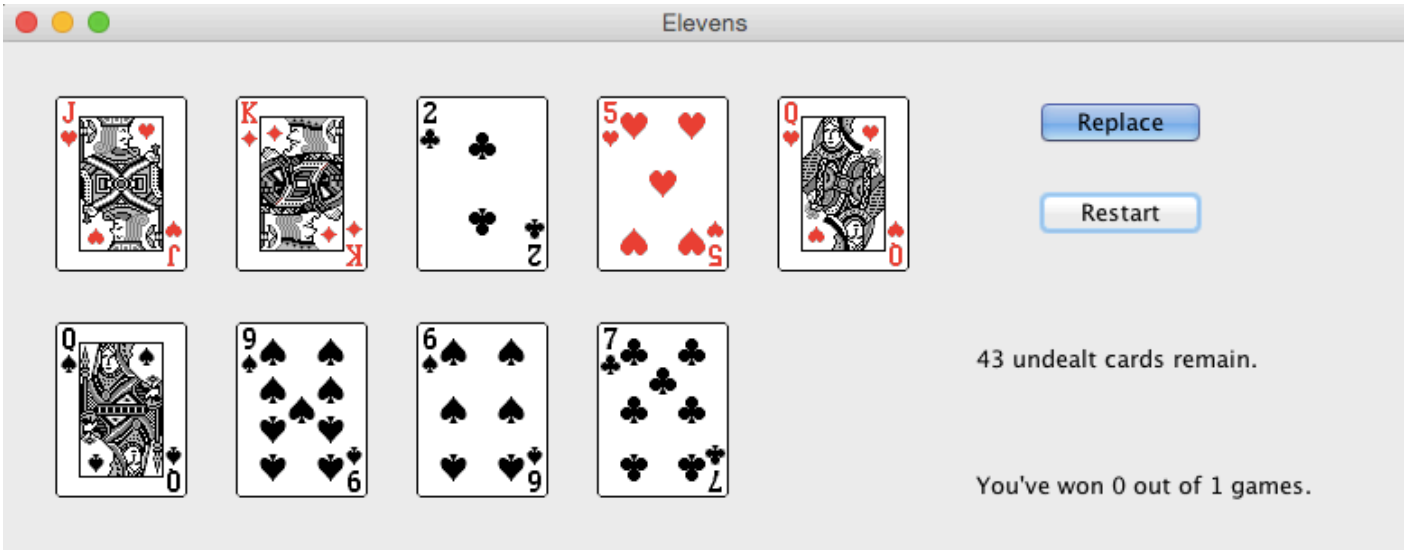
While you are working on this project, you will use the "Pair Programming" approach. In pair programming, two programmers share one computer. One student is the "driver," who controls the keyboard and mouse. The other is the "navigator," who observes, asks questions, suggests solutions, and thinks about slightly longer-term strategies. You should switch roles about every 20 minutes. For this assignment each partner will receive the same grade.

Be sure that both of you have duplicate saved copies of your work each day. You need to have something to work with tomorrow in the event your partner is absent.

The solitaire game of Elevens uses a deck of 52 cards, with ranks A (ace), 2, 3, 4, 5, 6, 7, 8, 9, 10, J (jack), Q (queen), and K (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 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.

The game is won when the deck is empty and no cards remain on the table. The game is lost when there are no more possible (i.e. legal) combinations of cards to be removed.



For example, if the cards above were dealt to the board, three moves would be possible. The 5 and the 6 could be removed. The 2 and the 9 could be removed. Lastly, the J, Q, and K could be removed. As cards are removed, they are replaced with new cards from the deck. The game continues until either all of the cards are removed, or there are no more possible moves to be made.

If you wish to experiment with playing the game yourself, here is an executable jar file of a complete Elevens game (you may need to install a Java JDK for it to run on your computer).

To get started on this assignment, download, extract and save the Elevens project folder to your computer. All of the starter code has been provided for you. It contains nine java classes and a no-source tester. The first four classes are partially completed (you will be completing the remainder).

1. `Card` objects represent a playing card.
2. `Deck` objects represent shuffled deck of `Card` objects.
3. `Shuffler` is a class where you can test your shuffle routines to make sure they are correct.
4. `ElevenBoard` represents the playing board of an Elevens game.
5. `ElevensTest` is a no-source tester class that will check your work as you proceed.
6. `Board` is an abstract class that contains common methods for many different type of games similar to Elevens. Note that `ElevensBoard` is a subclass of `Board`.
7. `CardGameGUI` is a class that represents a Graphical User Interface for card games like Elevens.
8. `ElevensGUIRunner` is a class specifically designed to run the GUI version of the Elevens game.
9. `ThirteensBoard` represents the playing board of a Thirteens game (to be discussed later). Similar to `ElevensBoard`, `ThirteensBoard` is also a subclass of `Board`.
10. `ThirteensGUIRunner` is a class specifically designed to run the GUI version of the Thirteens game.

1. Begin by completing the implementation of the provided `Card` class as follows.
   a. Open the `Card` class in the editor and look at the declaration of the instance variables near the top of the class. Then, complete the body of the constructor by using the parameters to set the instance variables.
   b. Complete the getter methods for the card's rank, suit, and value.
   c. Complete the `equals` method that tests the equality between two card objects. Be sure to use the proper way to compare the equality of objects, as well as the proper (only) way to compare the equality of primitives.

   A completed `toString` method has been provided for debugging purposes later on.

   Run the tester to make sure the `Card` class has been completed properly before you continue.

2. Next comes a more challenging part of the assignment. Complete the implementation of the Deck class as follows.
   a. Recall that `Card` objects can be created in the following manner:

```
Card card = new Card(rank, suit, value);
```

   where `rank` is the rank of the card (a `String`), `suit` is the suit of the card (a `String`), and `value` is the point values of the card (an `int`). Specifically, the statement

```
Card myCard = new Card("ace", "diamonds", 1);
```

would create the Ace of Diamonds as a `Card` object with a point value of 1. You will need to create individual `Card` objects as a part of making a deck of cards.

Now look at the `Deck` constructor, which receives three arrays as parameters. The arrays contain the ranks, suits, and values for each card in the deck. The constructor instantiates the instance variable array `cards` to hold the appropriate number (which is the number of ranks times the number of suits) of `Card` objects, and then creates the individual `Card` objects (one at a time) and adds them to the array. In order to make a deck of cards, you'll probably need a nested for loop to iterate through all the possible rank and suit combinations. (Note that each rank is associated with a particular value).

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 instance variable `size` would be set to the length of the array, which in this example is 6.

Finally, the constructor should shuffle the deck by calling the `shuffle` method. (Note that you will not be implementing the shuffle method until later).

Here is some pseudo-code to help you understand how to complete the `Deck` constructor. (Indentation indicates grouping.)

```
public Deck (ranks, suits, values)
    instantiate cards array for the proper amount of Card objects (ranks * suits)
    set instance variable size to zero
    set local variable position to zero
    for each rank in ranks
        for each suit in suits
            make new Card object (rank, suit, values[position])
            set cards[size] to the new Card object
            increment size
        increment position
    call the shuffle method
```

That should do it. Because the instance variable `size` gets incremented each time a new `Card` object is added to the `cards` array, `size` should contain the proper value when the constructor is finished. The tester will let you know if you completed everything correctly.

Run the tester to make sure the `Deck` constructor is working properly before you continue.

Now the next three methods are relatively easy.

b. Complete the `getSize` method. This method should return the instance variable `size`, which represents the number of cards that are left to be dealt.

c. Complete the `isEmpty` method. This method should return `true` when the `size` of the deck is 0; `false` otherwise.

d. Complete the `deal` method. This method "deals" a card by returning the next card to be dealt, if there are any cards left in the deck. It returns `null` if the deck `isEmpty`.

   Because we would like to easily reset the deck and shuffle all cards again from time to time, we are going to leave all of the cards in the deck, and indicate a dividing line between those cards that have been dealt, and those that have not. Cards are dealt one at a time from the *end* of the `cards` array, and the dividing line between dealt and un-dealt cards is indicated by the `size` instance variable (it represents the number of cards not yet dealt). At the beginning of the game, `size` should automatically indicate a full deck of cards (e.g. `cards.length`).

   So now, it is really simple to "deal" a card. First, check if the deck `isEmpty` (meaning there are no more cards to available to deal), and if so, return `null`. Otherwise, simply decrement the `size` instance variable, and then return the card located at position `size`. That's it!

   Furthermore, in order to easily reset the deck back to a full deck, you just have to reset the `size` instance variable to `cards.length`, and then shuffle the cards again.

   Run the tester to make sure the `Deck` class (not including the `shuffle` method) is working properly before you continue.

3. Now we are going to take a brief detour to explore the concept of shuffling. Think about how you shuffle a deck of cards by hand. How well do you think it randomizes the cards in the deck?

   Now consider the shuffling of a deck, that is, the permutation of its cards into a random-looking sequence. A requirement of the shuffling procedure is that any particular permutation has just as much chance of occurring as any other. We will be using the `Math.random` method to generate random numbers to produce these permutations.

   Several ideas for designing a shuffling method come to mind. Here we will consider two of them:

   Perfect Shuffle

   Card players often shuffle by splitting the deck in half and then interleaving the two half-decks, as shown at the right. This procedure is called a *perfect shuffle* if the interleaving alternates between the two half-decks.

   Unfortunately, the perfect shuffle comes nowhere near generating all possible deck permutations. In fact, eight perfect shuffles of a 52-card deck will return the deck to its original state!

   Consider the following "perfect shuffle" algorithm that starts with an array named `cards` that contains 52 cards and creates an array named `shuffled`. *Note that this is just an example for discussion purposes. Don't begin to write your shuffle methods until you have read the instructions.*

```
initialize shuffled to contain 52 "empty" elements
set k to 0
for j = 0 to 25
      set shuffled[k] to cards[j]
      set k to k+2
set k to 1
for j = 26 to 51
      set shuffled[k] to cards[j]
      set k to k+2
```

   This approach 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`.

Selection Shuffle

Consider the following algorithm that starts with an array named `cards` that contains 52 cards and shuffles by swapping selected elements in the same array. We will call this algorithm a *selection shuffle*.

```
for k = 51 down to 1
    Generate a random integer r between 0 and k, inclusive
    Exchange cards[k] and cards[r]
```

It is interesting to note this has the same structure as a selection sort:

```
for k = 51 down to 1
    Find r, the position of the largest value among cards[0] through cards[k]
    Exchange cards[k] and cards[r]
```

The selection shuffle algorithm, however, does not require to a loop to find the largest (or smallest) value to swap, so it works even more quickly than a selection sort.

Now, complete the following two methods in the `Shuffler` class. ***Note that in both cases, you are shuffling an array of integers of unknown length.***

a. Complete the `perfectShuffle` method, using the perfect shuffle algorithm outlined above. Note that this method returns a new, shuffled array that contains the same values as in the provided array, just in a newly specified order.

b. Complete the `selectionShuffle` method, using the selection shuffle algorithm outlined above. Note that this method does not return anything, but rather shuffles the existing array by reordering (swapping) specified values. When you generate a random index to be swapped, be sure that it is within the appropriate range of values. Also remember that in order to swap two values in an array, you have to use a temporary variable to first store one of the values.

Run the tester to make sure the `Shuffler` class is working properly before you continue.

4. Now it is time to take what you have learned about how to shuffle cards and put it to use.

a. Implement the `shuffle` method in the `Deck` class. Write a new selection shuffle algorithm similar to what you just completed in the previous exercise. The `shuffle` method also needs to reset the value of `size` to indicate that all of the cards can be dealt again.

When you have successfully completed the shuffle method of the `Deck` class, the tester will indicate that it is so.

Run the tester to make sure the entire `Deck` class (including the `shuffle` method) is working properly before you continue.

5. The Elevens game belongs to a set of related solitaire games. In this section you will learn about some of these related games. Then you will see how inheritance can be used to reuse the code that is common to all of these games without rewriting it.

Thirteens

A game related to Elevens, called Thirteens, uses a 10-card board. Ace, 2, … , 10, jack, queen correspond to the point values of 1, 2, …, 10, 11, 12. Pairs of cards whose point values add up to 13 are selected and removed. Kings are selected and removed singly.

Tens

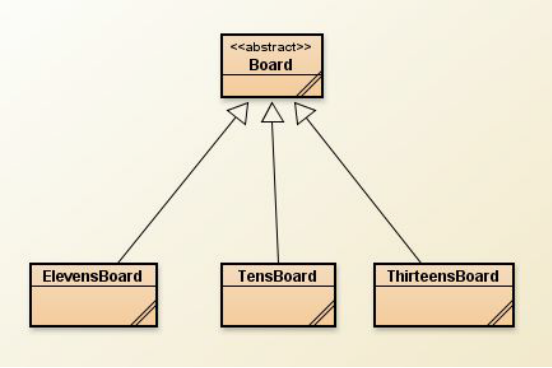
Another relative of Elevens, called Tens, uses a 15-card board. Pairs of cards whose point values add to 10 are selected and removed, as are quartets of kings, queens, jacks, and tens, all of the same rank (for example, K♠, K♥, K♦, and K♣).

Let's take a moment here to have a brief discussion about abstract classes. In reading the descriptions of Elevens and its related games, it is evident that these games share common state and behaviors. Each game requires:

- State (instance variables) — a deck of cards and the cards "on the" board.
- Behavior (methods) — to deal the cards, to remove and replace selected cards, to check for a win, to check if selected cards satisfy the rules of the game, to see if there are more legal selections available, and so on.

With all of this state and behavior in common, it would seem that inheritance could allow us to write code once and reuse it, instead of having to copy it for each different game. But how? If we use the "IS-A" test, a `ThirteensBoard` "IS-A" `ElevensBoard` is not true. They have a lot in common, but an inheritance relationship between the two does not exist. So how do we create an inheritance hierarchy to take advantage of the commonalities between these two related boards?

The answer is to use a common superclass. We've taken all the state and behavior that these boards have in common and put them into the `Board` class. Then we could have `ElevensBoard`, `TensBoard` (if we had one), and `ThirteensBoard` inherit from the `Board` class. This makes sense because each of them is just a different kind of board. An `ElevensBoard` "IS-A" `Board`, a `ThirteensBoard` "IS-A" `Board`, and a `TensBoard` "IS-A" `Board`. A diagram that shows the inheritance relationships of these classes is included below. Note that `Board` is shown as abstract. We'll discuss why later.



Let's see how this works out for dividing up the code. Because all these games need a deck and the cards on the board, all of the instance variables go into `Board`. Some methods, like `deal`, will work the same for every game, so they are in `Board` as well. Methods like `containsJQK` are Elevens-specific and should be in `ElevensBoard`. So far, so good.

But what should we do with the `isLegal` and `anotherPlayIsPossible` methods? Every Elevens-related game will have both of these methods, but they need to work differently for each different game. That's exactly why Java has abstract methods. Because each of these games needs `isLegal` and `anotherPlayIsPossible` methods, we include those methods in `Board`. However, because the implementation of these methods depends on the specific game, we make them abstract in `Board` and don't include their implementations there. Abstract methods have the word `abstract` in their header, and contain no method body at all, but instead are concluded with a semicolon. Listing an abstract method in a parent class means that it *must* be implemented in any and every child class of that parent. It is a way of requiring that a method be specifically defined in all subclasses, and in that manner, it acts much in the same fashion as an interface does.

Because `Board` now contains abstract methods, it must also be specified as an `abstract` class. Finally, we override each of these abstract methods in the subclasses to implement their specific behavior for that game. But if we have to implement `isLegal` and `anotherPlayIsPossible` in

each game-specific board class, why do we need to have the abstract methods in `Board`? Consider a class that uses a board, such either one of the GUI runner classes (`ElevensGUIRunner` or `ThirteensGUIRunner`). Such a class is called a *client* of the `Board` class.

The GUI runner does not actually need to know what kind of a game it is displaying! It only knows that the board that was provided "IS-A" `Board`, and it only "knows" about the methods in the `Board` class. The GUI program is only able to call `isLegal` and `anotherPlayIsPossible` because they are included in `Board`.

Finally, we need to understand how the GUI runner is able to execute the correct `isLegal` and `anotherPlayIsPossible` methods. When the GUI runner starts, it is provided an object of a class that inherits from `Board`. If you want to play Elevens, you provide an `ElevensBoard` object.

If you want to play `Tens`, you provide a `TensBoard` object. So, when the GUI runner uses that object to call `isLegal` or `anotherPlayIsPossible`, it automatically uses the method implementation included in that particular object. This is known as *polymorphism*.

Now, back to the task at hand. Since the `Card` and `Deck` classes have already been completed, the final class to finish is `ElevensBoard`. This class will contain the state (instance variables) and behaviors (methods) necessary to play the game of Elevens.

Notice how `ElevensBoard` specifically defines (as instance variables) the ranks, suits, and values to be used for a game of Elevens. Since they are not going to ever change (it will always be a deck of 52 standard playing cards), can you see why it is appropriate that they are declared as `final` (constant) variables?

`ElevensBoard` has four methods that need to be completed, namely `isLegal` and `anotherPlayPossible` (as required by the abstract methods of the same name in the parent class `Board`), as well as two helper methods `containsPairSum11` and `containsJQK`.

a. Start by completing `containsJQK` first. The passed parameter is a `List` of `Integer` objects (notice they are *not* `int` primitives). `Integer` objects are objects that contain (yup, you guessed it!) an integer value, but they are full-fledged objects (not primitives). `Integer` objects are part of a group called "wrapper" classes because they wrap a primitive value in an object. Because a `List` can only contain objects (and not primitives), wrapper classes must be used instead to store primitive values in a list. Fortunately, wrapper classes are 'auto boxed and unboxed', which means that they can be used implicitly as if they were primitives.

   The `List` of `Integer` objects represents the indices of the cards on the board to be checked. In our case, that will be somewhere between 0 and 8 inclusive (9 cards total on an Elevens board). Note that it may be less than 9 cards, as it may represent three cards that were selected by the user in playing the game. Regardless of how long the list is, you will need to iterate through all of it and see if a Jack, Queen, and King are there or not.

   Procedurally, you will probably want to set up three `boolean` variables such as `foundJack`, `foundQueen`, and `foundKing` and initially set them all to false. Then iterate through the list, and get the card at the given index, check the rank of the card and see if it is one of the three we are looking for. If so, set the appropriate `boolean` variable to `true`. Once you have finished processing all of the cards in the list, then see if all three `boolean` variables are `true` or not (return that value).

Note in the `Board` class, the instance variable array `deck` is `private`, so it cannot be directly accessed. You will need to use the `public` method `cardAt` that will return the card at the given index. Here is some sample code that you may find useful:

```
for(Integer i : selectedCards)
   if(cardAt(i).getRank().equals("jack"))
       foundJack = true;
```

Run the tester to make sure the `containsJQK` method is working properly before you continue.

b. Next, complete the `containsPairSum11` method. Again, the passed parameter is a `List` of `Integer` objects, which represent the card indices from the game board. Many times, the list will just contain two card indices (to check if those two cards sum to 11). Other times, however, it may be all the cards on the board (to see if there is still a legal play to be made). You will need to look at all the possible pairs of cards to see if at least one pair adds up to 11 or not. Most likely that means a nested loop to take one card, and compare it to all of the other ones. No need to compare a given pair of cards more than once, though, so iterate thought the list one card at a time, comparing that card to the remaining cards in the list after that one. In other words, the number of cards to be compared will get smaller and smaller as you proceed thorough the list.

Note that in order to properly get a `Card` object to check, you will need to make a call such as `cardAt(selectedCards.get(i))`. Then make a similar call for the other card.

Run the tester to make sure the `containsPairSum11` method is working properly before you continue.

c. Now with the two previous helper methods completed, the rest is relatively easy. Your task now is to complete the `anotherPlayIsPossible` method. To do so, just make a call to both `containsJQK` and `containsPairSum11` methods, passing to each the entire `cardIndexes` list. If either method returns `true`, then `anotherPlayIsPossible` is `true`.

Run the tester to make sure the `anotherPlayIsPossible` method is working properly before you continue.

d. Lastly, complete the `isLegal` method. Here you are checking the list of cards selected by the user to see if they are a legal combination or not. The are only two possible legal combinations, namely two cards that add up to 11, or three cards that are a Jack, Queen, and King. Therefore, if the list contains exactly two cards, then pass the list to `containsPairSum11` method to see if it is legal or not. Otherwise, if the list contains exactly three cards, then pass the list to `containsJQK` method to see if it is legal or not. For a list of *any* other size the combination is not legal, so return `false`.

Run the tester to make sure the `isLegal` method, and indeed the entire project is now working properly.

There you go. You are all done now. Congratulations on finishing a very challenging assignment. Please demonstrate your work by running the tester class for me to observe so that I will know you have completed the assignment.

You should now be able to play Elevens using the provided `ElevensGUIRunner` class. You should also be able to play Thirteens using the provided `ThirteenGUIRunner` class.

If you are so inclined (this part of the assignment is optional), why don't you write implement a game of Tens as well? You'll need to write from scratch a `TensBoard` class, and a `TensGUIRunner` class. It shouldn't be that difficult, though, as you can use the examples from Elevens and Thirteens as your guide. Have fun!