**Elevens**

In this lab you will be coding a card game that is a variation on classic solitaire. The game is called Elevens, and its rules are pretty simple:

* Nine cards are dealt ("on the board") at a time
* Cards have the following values:
  + Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10 have a point value of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
  + Jack, queen, and king have no point value.
* Cards can be removed from the board if:
  + Two cards' values sum to 11 (e.g. 4 and 7, or Ace and 10)
  + Three face cards are showing and they are a jack, a queen, and a king
* Cards that have been removed from the board are replaced (dealt) from the deck – if you make it through the entire deck, you win. If there are no more legal moves on the board, you lose

You will be writing the core logic of the game - a working GUI has been provided as well.

**Elevens**

1. Play the game of Elevens for a bit to see how it works by running the **Elevens.jar** file in the lab folder (the rules were listed previously).
2. Import the necessary files to begin the project by doing the following:
   1. **BlueJ**
      1. Download the ***BlueJ starter files*** folder and extract it (right-click -> **Extract all**).
      2. Copy the entire *unzipped* folder to your H: drive. Rename it Lab16-Elevens.
      3. Open by double-clicking the **package.bluej** file.
   2. **Eclipse**
      1. Download the ***Eclipse starter files*** folder and extract it (right-click -> **Extract all**).
      2. In Eclipse, make a new Java project called Lab16-Elevens.
      3. **Drag and drop** the *unzipped* materials as follows:
         1. Drag the source (.java) files into the "src" folder
         2. Drag the "cards" folder into the project folder (*not* the "src" folder).
3. Note that the classes responsible for creating the graphics (those containing the letters "GUI") won't compile at the moment. *Don't freak out.* They will compile later, when you have completed the classes they rely on. All other classes should compile and be testable (in a Runner class, for example).
4. At this point in the year this exercise would be trivial, so the Card and Deck classes have been (nearly) completed for you. You should be familiar with their methods and instance variables. A couple notes:
   1. Card objects are added to the Deck, and the deck "deals" a Card at the ***current index*** (rather than actually removing the Card object from the Deck). The current index is maintained with the size instance variable (size represents the number of un-dealt cards in the deck).
   2. The toString() methods have already been overridden for both - you can print a Card or Deck object, when testing your methods.
5. In the Deck class, complete the shuffle() method that will shuffle the cards in the deck. Use a shuffle algorithm similar to a "selection sort" (***not*** an API method). Pseudo-code:

For i = 51 (deck size - 1) down to 1:

− Generate a random integer r between 0 and i, inclusive

− Swap the card at r and the card at i (the current index)

//note that *cards* is an ArrayList in the Deck class!

Don't forget to reset the size instance variable - after shuffling, the deck should be reset.

1. Complete the ElevensBoard class (started for you).
   1. ***Before looking at the code***, think about what would be required to play a game of Elevens at your desk. List everything you'd need, then compare what you came up with to the instance variables in the ElevensBoard class.
   2. Complete the cardIndexes() method, that returns a List<Integer> of all **indexes** in the cards array with non-null references (i.e. the indexes of all locations on the board that have a card). For a new game, this method should return a list containing [0,1,2,3,4,5,6,7,8], as the board will be full. Use the following variable initialization:

List<Integer> cardIndexes = new ArrayList<>();

/\* List is an abstract super-type (an interface, which we'll learn about soon). You can use the more general type of List<Integer> to store any ArrayList<Integer> object, because ArrayList *is-a* List \*/

* 1. Complete the containsPairSum11() method, that returns true if ***any*** of the Cards at the indexes in the List parameter sum to 11 (the list may contain more than 2 cards – this method does not check for a "legal" move). Use the cardAt(int k) method (returns a Card at index k), rather than accessing the cards array directly (this will help later).
  2. Complete the containsJQK() method, that returns true if ***any*** of the Cards at the indexes in the List parameter form a trio of jack, queen, and king. A triple-nested for loop is *not* the best way of going about this.
  3. Complete the isLegal() method. A move is legal iff (if and only if):
     1. Two cards are selected, and their values sum to 11.
     2. Three cards are selected, and they are a jack, a queen, and a king.
  4. Complete the anotherPlayPossible() method that returns true if, given the current state of the board, the player has another legal move possible.

1. A few other variations on solitaire exist that are very similar to Elevens. For example, the game of Thirteens has the following slightly different rule set:
   * Uses a 10-card board
   * A, 2 … 10, J, Q correspond to the point values of 1, 2 … 10, 11, 12.
   * Pairs of cards with point values that add up to 13 can be selected and removed
   * Kings can be selected and removed individually

Keeping in mind the differences between Elevens and Thirteens, refactor your code in the ElevensBoard class into a more reusable ***abstract*** class called Board. The Board class should be used to store code common to these types of card games. You will never instantiate a Board, rather you will instantiate its children, actual game instances (e.g. ElevensBoard).

* 1. Think about which variables will and won't change for sub-types (other game variations, like Thirteens). Variables shared amongst game types should be inherited, while variables that are specific to a particular game should *not* be inherited.

For example, the pointValues array will be specific to each game. The ranks and suits of the cards in the deck won't change, and can reasonably be moved to the Board class. Every game type will have an array of cards - this variable should also be inherited (though it may be instantiated with different sizes by sub-classes).

* 1. Methods that won't change from game to game should be ***concrete*** (fully implemented), while methods that exist in all game variations but whose implementations will change should be made ***abstract*** (such that extending classes will *have* to implement them).
  2. Modify the ElevensBoard class, which should now extend Board. Override the necessary methods; make sure to put a call to the super-class' constructor in ElevensBoard's constructor (and make sure to supply Board the information it needs to initialize its variables).
  3. Test that your ElevensBoard class works by running the main() method in the **ElevensGUIRunner.java** file. Thoroughly test your code – attempt to remove items that should and shouldn't be removable, make sure the number of un-dealt cards at the beginning of a game is correct, etc. Fix any mistakes you may have.

1. (Riddle) What is the difference between a dollar and a half and thirty five-cents?
2. Perform the steps necessary to extend Board a second time and implement the ThirteensBoard class (the rules were given previously).
   1. Ensure that your code works by modifying **ElevensGUIRunner.java**, such that it now uses a ThirteensBoard instance (object), rather than an ElevensBoard instance. Only one small change should be required.
   2. Marvel at the fact that all it took was a few minor changes to make an entirely different game. *Nothing wrong with being crafty.*