CSc 127B — Introduction to Computer Science II
Fall 2015 (McCann)

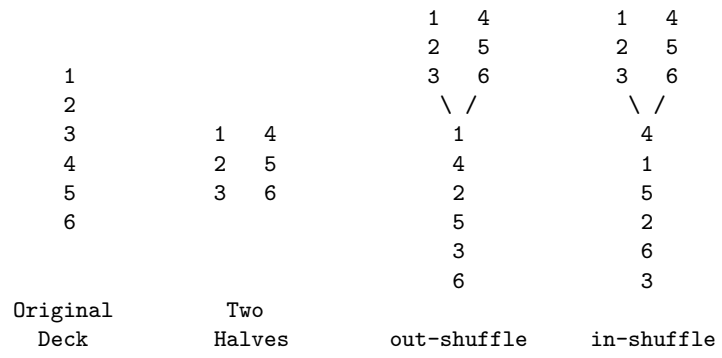
http://www.cs.arizona.edu/classes/cs127b/fall15/

Program #8: Faro Shuffling

Due Date: November 5th, 2015, at 9:00 p.m. MST

Overview: When shuffling a deck of playing cards by hand, a common technique is to divide the deck approximately in half, flip through the halves with your thumbs rapidly such that the cards form an interleaved pile, and repeat a few more times. This technique is called a *Faro Shuffle*, apparently because a skilled shuffler could thereby cheat at the game of Faro.

If the interleaving alternates single cards, one from each pile, that's a *perfect Faro shuffle*, and comes in two forms. An *out-shuffle* creates the interleaving with the original top and bottom cards at the top and bottom of the interleaving. An *in-shuffle* places the top and bottom cards as the second and second-to-last cards. The following figure demonstrates these two outcomes on a deck of six cards.



To keep things simple, decks may contain only an even number of cards. Note that 0 is an even number. A deck will have no more than 104 legal cards (to support games that combine two decks).

For this assignment, you will be creating your own card and deck classes. A deck will be represented by a linked list of node objects, each of which references a card object. A deck's methods will include out-shuffle and in-shuffle operations. Details are given below.

Assignment: Write a complete, well-documented Java program that consists of at least these three classes. You may add private methods to any of these as you see a need for them.

1. **Deck**. Deck is really a linked-list class with limited operations, with list elements that are Node objects (see below). Here are the public constructors and instance methods that you must support:

   (a) `Deck(String)` — Given the name of a file, create a deck object and populate it with the cards found in the file, in the order the cards appear. That is, the first card in the file is the top of the deck, the second card is the second in the deck, etc. See the Input section, below, for file content details.

   (b) `Deck(StringBuilder)` — Given the name of a file, create a deck object and populate it with the cards found in the file, but insert them into the deck's list using the following ordering scheme: Ace of Clubs < 2 of Clubs < ... < King of Clubs < Ace of Diamonds < ... < Ace of Hearts < ... < Ace of Spades < ... < King of Spades. Thus, if the file contains the 3 of Hearts, the 7 of Spades, 2 of Hearts, and the Jack of Clubs, the list will hold them in this order: Jack of Clubs, 2 of Hearts, 3 of Hearts, and 7 of Spades. As this example suggests, a file may contain just a subset of the standard deck's 52 cards, so long as the subset has an even number of cards. This constructor takes a StringBuilder object to distinguish it from the first constructor.

(Continued ...)

(c) `void outShuffle()` — Performs one perfect Faro out-shuffle on the deck's content. This method changes the state of the current object; that is, it does not return a new deck object, it only shuffles the content of the current deck. If the deck contains no cards, the state is not changed.

(d) `void inShuffle()` — Same as outShuffle(), except that it performs an in-shuffle on the deck's content.

(e) `toString()` — returns a String containing a representation of the deck's content. Each card is represented by two characters, and cards are separated by a single space. The first character is the rank ('A', 2, ..., 10, J, Q, or K) and the second is the suit ('C', 'D', 'H', or 'S'). For example, the ordered deck of four cards from above would be represented by the 11-character string "JC 2H 3H 7S".

2. **Card**. Each Card object represents a single card. Card must implement the `Comparable` interface.

   (a) `Card(int,String)` — Given the rank and suit name of a card, create a Card object representing that card.

   (b) `int getRank() and String getSuit()` — The getters. getRank() returns in integer in the range 1 - 13 (inclusive) and getSuit() returns the plural name of the suit with only the first letter capitalized (e.g., Clubs).

   (c) `toString()` — Returns a String object containing the two-character card representation used in Deck's toString() method. For example, the Queen of Diamonds is represented with "QD".

   (d) `compareTo()` — As specified by `Comparable`.

3. **Node**. Each Node object references the Card object it is representing and the next Node object in the linked list of Node objects that is representing the content of a Deck object.

   (a) `Node(Card)` — Creates a Node object that holds the given Card object reference. The Node object's Node reference field is set to null.

   (b) `Card getData(), Node getNext(), void setData(Card), void setNext(Node)` — The getters and setters. They should be self-explanatory.

Data: A file containing a deck of cards will be a text file with one card per line. Each card is represented with a pair of values. First is the rank, an integer from 1 through 13 (inclusive), where 1 = Ace, 11 = Jack, 12 = Queen, and 13 = King. Second is the suit name, one of 'Clubs', 'Diamonds', 'Hearts', and 'Spades', where capitalization is irrelevant (and no quotes). The rank is separated from the suit by one or more spaces, and spaces may appear before the rank and after the suit. Here's a sample deck file matching the content used earlier, plus an illegal card:

```
3 hearts
12 monkeys
7 SPADES
2      HeArTs
   11 Clubs
```

If an illegal card is found (e.g., one with an invalid rank, suit, or both), ignore the card and continue to the next card in the file. If a file contains an odd number of legal cards, place all but the last legal card into the deck. If the file contains more than 104 legal cards, ignore the extras; that is, use only the first 104 legal cards. Your Deck class must be able to support a deck with duplicates (e.g., two Aces of Spades).

Output: None required! You may have noticed that there is no specification for a `Prog8.java` file. We strongly recommend that you create one, to exercise your Deck, Card, and Node classes (that is, to test them thoroughly), but that file isn't a 'deliverable' for this assignment. You're welcome to turn in your testing program with the other files, but we won't grade it.

(Continued ...)

Turn In: For this assignment, you must place each of your classes in its own `.java` file (no `.zip` files, please!). Be sure to submit them all to the `cs127bsXp08` directory at any time before the stated due date and time. Of course, you can turn them in late if you still have late days to use, or don't mind losing 20% per day if your late days are exhausted.

For More Information:

- Wikipedia has a page on Faro shuffling . . .
  `http://en.wikipedia.org/wiki/Faro_shuffle`
- . . . and one on the game of Faro after which the shuffle appears to be named:
  `http://en.wikipedia.org/wiki/Faro_(card_game)`
- Perfect Faro shuffling is a lousy choice for actually shuffling a deck. If you ever find yourself writing a card game and need an algorithm to shuffle the deck, the Fisher–Yates algorithm is a much better choice:
  `http://en.wikipedia.org/wiki/Fisher-Yates_shuffle`

Hints, Reminders, and Other Requirements:

- You may use any file classes you care to use to read the deck files – no restrictions.
- You have to create your own Deck (a.k.a. linked list) class, with its own operations, for this assignment. The idea is to give you practice working with a linked structure. If you were to just use Java's version as a Deck representation, you wouldn't get any practice at all with manipulating node references to maintain a linked list.
- If you're thinking that it would be nice to be able to represent the two halves of the deck as Deck objects for the shuffling operations, you're starting to think like an object-oriented programmer. To keep this assignment manageable, we didn't include operations that would support this. *You may add public deck methods to support additional list operations!* We won't test such additional methods, just those specified in this handout. Note that you can do the shuffles without additional methods.
- Seems like there was something else . . . oh, right: Start early!