INTRODUCTION TO COMPUTER SCIENCE I
PROJECT 2
Blackjack

1 The game

For this assignment, we will be writing a program to play (a somewhat limited version of) the game of *Blackjack* (a.k.a., *21*). Specifically, this version of the game will pit a single player (the user) against the dealer (the machine). There will be no *splitting*, *insurance*, or *surrendering*. Here is a description of our simplified version of Blackjack:

- In this game, each card has a *value* based on its *rank* (but not its *suit*). Specifically, cards of rank 2 through 9 are worth their *face value*; cards 10 through King have a value of 10; the Ace has a value of 11.[1]
- Before the first hand is dealt, the dealer will *shuffle the deck*. This deck will then be used for repeated hands so long as at least 20 cards remain in the deck before dealing. When the deck has too few cards remaining, a fresh deck will be shuffled before starting a new hand.
- The game should begin with the player granted $100 (virtual). At the beginning of each hand, the player must *place a wager* of at least $1, and (of course) at most the amount that the player has remaining.
- The hand begins with the dealing of two cards each to the dealer and player. For the player, both cards are *face up* (the card's suit and rank are shown); the dealer, however, gets one card face up, the other *face down* (hidden).
- The player then plays out their hand, trying to get the cards in their hand as close to a combined value of 21 as possible *without going over*. The player will be given repeated opportunities to *hit* (take another card from the deck and add it to the hand) or to *stay* (leave the hand as-is, thus ending the player's turn). The value of the hand is the sum of the values of the cards, as described *supra*. If the value is over 21, the player has *busted*, the hand ends immediately (without the dealer playing out its own hand), and the player loses the wager.
- The dealer then plays out its hand by hitting until the value of its hand is *at least 17*.

[1] Remember that this is a simplified version of the game. Later, you can add a little complexity by allowing each Ace to be worth 1 or 11, as it would be in a real game of Blackjack.

- With both hands played out, the winner is determined as follows:

    1. If the dealer has 21, the dealer wins (even if the player also has 21), taking the wager.
    2. If the dealer has busted, then the player wins, being paid the value of the wager.[2]
    3. If the dealer's hand has a higher value than the player's, then the dealer wins, taking the wager.
    4. If the player's hand has a higher value than the dealer's, then the player wins, being paid the value of the wager.
    5. If both player's hands have the same value, then the hand is a *push* (no winner), returning the wager to the player.

2 Your assignment

Getting started: Create a new folder/directory for `project-2`, and open/change into it. Then go to the following link for the starting code:

`bit.ly/COSC-111-project-2-source`

You should save this file into your `project-2` directory with the name `Blackjack.java`. Then open the code into *Emacs/Aquamacs*. You will see the beginnings of a Blackjack program, and your job is to **complete the code** such that it plays a game of Blackjack as described above. A player should be allowed to play hands until they run out of (virtual) money or choose to end the game.

2.1 On the representation of the cards

This game assumes a standard 52-card deck, where each card is defined by two characteristics:

1. A **suit:** One of *spades, hearts, clubs, diamonds*.

2. A **rank:** One of *Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King*.

In order to make a game that uses such a (virtual) deck of cards, we must choose some way to represent each. For this assignment, we will assign a unique integer to represent each card—an *encoding*. Specifically, each card will be uniquely represented by an integer between 0 and 51 (inclusive), like so:

[2]For example, if the player wagered $10, then the player would keep that wagered $10 **and** receive an additional $10 from the dealer.

Rank	Suit	Encoding
Ace	Spades	0
2	Spades	1
3	Spades	2
⋮	⋮	⋮
Queen	Spades	11
King	Spades	12
Ace	Hearts	13
2	Hearts	14
⋮	⋮	⋮
King	Hearts	25
Ace	Clubs	26
⋮	⋮	⋮
King	Clubs	38
Ace	Diamonds	39
⋮	⋮	⋮
King	Diamonds	51

With this arrangement, the *rank* and *suit* can be determined by a little arithmetic based on the number of cards in each suit, 13. Specifically, for a given card encoding . . .

```
suitNumber = encoding / 13
rankNumber = encoding % 13
```

. . . where . . .

`suitNumber`	**suit**
0	Spades
1	Hearts
2	Clubs
3	Diamonds

`rankNumber`	**rank**
0	Ace
1	2
2	3
⋮	⋮
9	10
10	Jack
11	Queen
12	King

In this way, you can represent the deck as an array of 52 `int` values. Making a new deck requires only that you assign 0 to 51 into the array, thus representing one of each card.

2.2 On the shuffling of the deck

The task of *shuffling* an array—known mathematically as *randomly permuting* it—requires some thought, and there are many possible approaches. First, we should define the problem clearly: A *permutation* of an array is a re-ordering of the elements in that array. A *random permutation* of an array of n elements is a random selection of the $n!$ possible orderings. So, for a small-scale example where $n = 3$, consider an array whose contents are:

```
position: 0 1 2
value:    8 2 5
```

There are $3! = 1 \times 2 \times 3 = 6$ possible orderings of these elements:

```
position:   0 1 2
ordering 1: 2 5 8
ordering 2: 2 8 5
ordering 3: 5 2 8
ordering 4: 5 8 2
ordering 5: 8 2 5
ordering 6: 8 5 2
```

A *good shuffling algorithm*[3] will generate each of these $n!$ orderings with probability $\frac{1}{n!}$. Another way of stating this property is that, for a good shuffling algorithm, *each element has an equal probability ($\frac{1}{n}$) of ending up in each of the n positions.*

Your primary goal (for this part of the assignment) should be to devise a *good shuffling algorithm* and write a method to perform it. Your secondary goal, if you have the time for such a thing, is to make an *efficient* such algorithm that uses as little memory space and time as you can devise.

3 Submitting your work

Submit your `Blackjack.java` source code file with the CS submission system, using one of the two methods:

- **Web-based:** Visit the CS submission systems web page at `www.cs.amherst.edu/submit`.
- **Command-line based:** Use the `cssubmit` command at your shell prompt.
  (WARNING: This method works only on `remus/romulus`.)

This assignment is due on Thursday, Apr-04, 11:59 pm.

[3] We can later more sharply, quantitatively define *goodness*.