Introduction to Computer Science – Honors I

CS 181 – Fall 2013

Assignment 3 (**due 10/17**)

Complete the following subjects and submit your answers/code electronically through moodle.

Under the Stevens Honor System, you are required to sign the Honor pledge in all material that you hand in for grading. The Stevens Honor pledge is shown below:
"I pledge my honor that I have abided by the Stevens Honor System."
Please copy the pledge to the top of your answer sheet and type your name. Programming assignments should also include the above in a comment block at the top of your source file(s).

```
/*
 * QuickSort.java
 *
 * "I pledge my honor that I have abided by the Stevens Honor System."
 *
 * John Smith (c) 2013
 *
 */
```

Maximum points = 100

The game

Uno is a popular card game in which each player holds a hand of cards, and tries to be the first one to "go out," or play all of their cards. Players are seated in a ring, and in the middle of this ring is an "up card," or a card placed on the table face up. Players take turns in sequence, clockwise around the ring. When it is your turn, you have the opportunity to play one of your cards on this up card (which will then become the new up card) and thus reduce the size of your hand. The card you play, however, must be playable on the up card, according to the following rules:

1. Most cards have a color -- red, green, blue, or yellow -- and you may play a card if it has the same color as the up card.
2. The colored cards each have a rank -- either a number from 0-9, or else a special "skip," "reverse," or "draw 2" rank. You may play a card if it has the same rank as the up card, even if it is of a different color.
3. There are two kinds of "wild" cards: ordinary wilds, and "draw 4" wilds. Either kind can be played on *any* up card. When you do so, you "call" a new color, specifying what color the next player must play.

Some special cards have an effect after being played, namely:

- If a "skip" card is played, the next player in sequence is skipped.
- If a "draw two" card is played, the next player in sequence must draw two cards from the deck, and is then skipped.
- If a "wild draw four" card is played, the next player in sequence must draw four cards from the deck, and is then skipped. (The player who played the "wild draw four" must then call a color as with a normal wild.)
- If a "reverse" card is played, the sequence of players is reversed to counterclockwise (or back to clockwise, after an even number of reverses.)

The object of the game is to run out of cards. When this happens, the player going out is awarded points based on **the cards remaining in the opponents' hands.** These points are calculated as follows:

1. For every numbered card held by an opponent, the winner of the round gets points equal to that number. (5 points for a 5, no points for a 0, etc.)
2. For every "special" colored card (draw two, reverse, and skip), the winner of the round gets 20 points.
3. For every "wild" card (either normal, or draw four), the winner of the round gets 50 points.

Normally, players continue playing hand after hand until one player reaches 500 points, and is declared the overall winner of the game.

Your project

We have a Uno simulation game. It simulates shuffling a deck, dealing hands to players, drawing an initial up card, enforcing all of the rules above, declaring a winner, and calculating scores. The only thing it does not do is actually choose a card to play from a hand (or call a color if a wild is played.)

Every student in the class will be writing their own code to do those two things: play a card from a hand, and call a color in case a wild was played. The simulation program will run the game, and then at the appropriate points, call your method(s) to do those two things. In this way, your program will be able to "play" Uno against your classmates in a tournament. Whoever has the best algorithm for playing a card should win.

You may object to that last statement, claiming that luck is a major factor. This would be true except for one thing: We are not going to pit your Uno program against your fellow students' programs in just one game, but in *50,000 straight games*. This will admittedly take a few seconds. But over that many games, any "lucky deals" that any one player might get will even out over time, leaving a superior algorithm with the lead.

Getting started

Do the following to get your project set up:

1. Create a new project. Call it "Uno." (Please do not call it anything else. Please call it "Uno," capitalized.)
2. Import all the provided java classes into your project.
3. One file is named “youruserid_UnoPlayer.java”. Name this class "jsmith_UnoPlayer,java" if "jsmith" is your Stevens userid.
4. Open up the jsmith_UnoPlayer.java (or whatever) file you just created. Change the word "youruserid" to your userid in the "public class" line.
5. There is a class named Main.java. This represents one of many scenarios (hands of cards plus chosen "up card") that your code will need to address properly and strategically. As before, replace " youruserid " with your actual Stevens userid.

You are now ready to begin **your mission of implementing the play() and callColor() methods** in **youruserid_UnoPlayer.java class**. You can use the main method defined in Main.java to test your code. Feel free to modify this method to test more scenario.

The UnoPlayer.java file

The UnoPlayer.java file contains what is called a "Java interface." This is an advanced topic that you will learn later; lucky for you, you don't need to understand anything about it right now except what I'm telling you on this page. What *is* important about this file is the two lines that begin with "public enum."

These two "enumerated types" represent the colors and the ranks of the cards in the program. Essentially, what is done here is add to the basic list of Java data types (int, double, etc.) Now, in addition to having a variable of type "int" or type "double," you can have a variable of type "Color" or "Rank."

The way you specify a value of one of these types is to prefix one of the capitalized words with "Color." or "Rank." For instance, here is some legal code:

```
int x = 5;
double y = 3.14;
Color myFavoriteColor = Color.BLUE;
Rank aPowerfulRank = Rank.WILD_D4;
```

There's really nothing more to it than that. Just be aware that the way you say "green" in the program is "Color.GREEN", and you'll be fine.

Uno player: methods

Your youruserid_UnoPlayer.java file has detailed comments describing the **play()** and **callColor()** methods you are to write code for. ***Reading*** **these detailed comments is an excellent and praiseworthy idea.** Note that the play() method takes four parameters. Here

is its method signature:

```
    public int play(List<Card> hand, Card upCard, Color
calledColor, GameState state);
```

Collectively, these arguments tell your method (1) what cards are in your hand, (2) what card is the "up card," (3) what color was called (this argument only has relevance if the "up card" is a wild), and (4) a way to find out other miscellaneous things about the state of the game, for your use in building a sophisticated strategy.

Your job is to write code that returns the integer of the card you wish to play. In the event that you cannot play any card, you should return -1 from this method. (Note that returning a -1 for a hand in which you *can* legally play is an error.)

If you wish, you may call methods on the GameState object passed to access detailed information about the state of the game. This object supports the following methods:

- `int[] getNumCardsInHandsOfUpcomingPlayers()` - the array returned by this method will have length equal to the number of players minus 1 (in a normal tournament game, this will be 3.) It tells you *in order* how many cards the next player to play after you has (at index 0), how many the player across from you has (at index 1), and how many the player who just played has (at index 2.) Note that when I say "the next player to play after you," that presumes, of course, that you do not play a skip or a reverse, in which case the player represented at index 1 or index 2, respectively, will be the next player.
- `int[] getMostRecentColorCalledByUpcomingPlayers()` - this array follows the same format as the previous, except it contains Colors, not ints. It tells you the most recent color each player called when they played a wild. (The value will be "Color.NONE" if that player has not yet played a wild card this round.)
- `List<Card> getPlayedCards()` - this method returns a list of the cards that have been played, in order, since the last deck remix. (A deck "remix" occurs if/when the draw pile becomes exhausted, and all of the cards in the discard pile are reshuffled and turned face down to become the new draw pile.) Interesting note: just from experimenting with my simulator, deck remixes are pretty uncommon. It seems that a large majority of games complete without ever requiring a deck remix, even when the players have the dumbest possible strategy (just play a random matching card.)
- `int[] getTotalScoreOfUpcomingPlayers()` - finally, this array tells you the total cumulative score for each player (in the grand 50,000-game Uno match), in order of their presumed turns, in the same way that arrays from the first two method calls in this list represented that order.

You can take advantage of the game state object by choosing to call any of these methods on it that you choose. Note that you are also free to ignore any of them if they're not of interest to you in developing your strategy.

The callColor() method is simpler. It takes only one parameter, telling you what's in your hand:

```
public Color callColor(List<Card> hand);
```

Our code will call this method of yours when you have just played a wild and we need to know what color you want to call. It must return one of the four valid Color values (*not* Color.NONE.)

Interface List<E>

A list is an ordered collection. The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. We have not covered Java interfaces yet, so you can just consider it as a customizable class. You can invoke Interface methods the same way as with classes.

Some commonly used methods of List are given as follows. You may find some of these methods helpful in your implementation.

`int size()`
Returns the number of elements in this list.

`boolean isEmpty()`
Returns `true` if this list contains no elements.

`boolean add(E e)`
Appends the specified element to the end of this list (optional operation).

`boolean remove(Object o)`
Removes the first occurrence of the specified element from this list, if it is present (optional operation). If this list does not contain the element, it is unchanged.

`E get(int index)`
Returns the element at the specified position in this list.

`E set(int index,E element)`
Replaces the element at the specified position in this list with the specified element (optional operation).

`void add(int index, E element)`
Inserts the specified element at the specified position in this list (optional operation). Shifts the element currently at that position (if any) and any subsequent

elements to the right (adds one to their indices).

`E remove(int index)`
Removes the element at the specified position in this list (optional operation). Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list.

More information on List<E> can be found here http://docs.oracle.com/javase/7/docs/api/java/util/List.html.

Grading

We will look at your **well-commented, compellingly documented** code to judge whether you attempted a non-trivial, intelligent approach. Your comments should narrate your algorithm completely and transparently. They should shed light on what your code is doing, and why, and explain the theory behind your approach.

If your code works correctly (**returns a correct answer**), without following any particular strategy, basically returning a random valid card, you will receive **70** points (**50 points for the correctness of the code and 20 explaining the code using comments and for using javadoc**). By correct answer we mean your play() method must always return the index of a card that can in fact be played on the up card, and must always return -1 only in the case where no card can in fact be legally played. Your callColor() method must simply always return a valid Color. Beyond that, it doesn't matter how simple your strategy is, or if your virtual player losses: they only must play according to the rules.

The remaining **30** points will be awarded for implementing a strategy for playing the card that will most likely (this game also includes chance after all) lead you to victory. The more information you use from the information made available to you during play, the higher your grade. For instance, picking the card to play based on its points, the current number of cards on each player's hand, the points of each player, use of special cards, etc. Some information on strategy can be found here http://www.unorules.com/best-strategies-to-win-uno/. Feel free to look for additional information on strategies online.