Assignment 5

Due Friday 7/25 at 10:00 AM

Sorting arrays (30 points): For this part of the assignment you will implement the sorting methods bubble sort and insertion sort. Download Sort.java from the assignments page and add your two sorting methods.

Specifications:

1. Your methods should be called **bubbleSort** and **insertionSort**, and should be public and static.
2. Each method should take as input an array of Comparable objects, like selectionSort does.
3. Each method should return the number of comparisons required to sort the array (as an int).
4. Use the algorithms I gave you in class on Thursday (7/17). These algorithms are also described in Savitch, pages 411,412 (in exercises 4 and 5 of chapter 6).
5. Add code to the **bubbleSort** method so that if the array is sorted after a pass through the array then the method terminates without any more comparisons.
6. The result of your two methods should be that the input array is sorted from "smallest" to "largest" (with the "smallest" entry in the array index 0).

You will want to test your methods, but you do not need to submit your testing code. Hand in a copy of Sort.java and submit a copy electronically.

Card games (30 points): For this part of the assignment you will create objects that will comprise a deck of playing cards. First you will create an object to represent a playing card, then you will create an object to represent a deck of playing cards.

Specifications for Card.java:

1. Your playing card objects should be defined in the class Card. Your Card objects should have a value (like 2,7, King) and a suit (Hearts, Spades, Diamonds, Clubs). Instance variables must be private. You may choose how to implement this class (what instance variables to use) but you must provide the methods described below.
2. Your Card class must implement the Comparable interface. That is, you must provide a compareTo method that returns an int and takes a Card object as input. Your method must return a negative number if the value of the calling Card object is less than the value of the input Card object, zero if the values are equal, and a positive number otherwise. The natural ordering on the values of the card is (from lowest to highest with Ace high): 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace. Your compareTo method should not distinguish between suits.
3. You must write a **toString** method that returns a string representing your Card object. The string must be "'value' of 'suit'" with 'value' being one of the strings in the list in number 2 above, and suit being one of the suits listed in number 1. Examples: Jack of Spades, 9 of Diamonds, 10 of Hearts, Queen of Clubs.
4. You must write a public method **getValue** that returns an integer. You should return an integer in the range 1 – 13 (inclusive) depending on the value of the card. You should return 1 if the card is a 2, 2 if the card is a 3, ..., 9 if the card is a 10, 10 if the card is a Jack, 11 if the card is a Queen, 12 if the card is a King, and 13 if the card is an Ace.
5. You must provide a public method **getSuit** that returns the suit of the card as a string ("Hearts", "Diamonds", "Clubs", or "Spades").
6. You should provide a default constructor that sets your card to be the Queen of Spades, and a constructor that takes as input an int and a string (in that order) and creates an appropriate new card. You may assume that the string is one of the four listed in number 5 above and that the int is in the range 1-13 (inclusive) representing the cards as in number 4 above. You should not write

mutator methods for your instance variables – there is no reason to change one playing card into a different playing card.

Specifications for DeckOfCards.java:

1. Your deck of cards object should be defined in the class DeckOfCards.
2. Your DeckOfCards object should have as a private instance variable an array of Card objects of length 52. You should also have a private integer instance variable to keep track of which card is the "top" card in the deck.
3. You should provide a default constructor which creates a standard deck of cards (one card of each value with each suit = 52 cards). Initially, you may assume that the "top" card in the deck is the card in your array at index 51. You do not need to provide any other constructors.
4. Provide a public method **shuffle** that shuffles the deck using the following algorithm: Pick two cards at random and swap their location in the deck, repeat 500 times. Your method should not return anything, it should just shuffle the DeckOfCards object. Make sure you test this method out well to make sure that your top and bottom card can be moved by your method.
5. Provide a public method **deal** that takes no input and returns a copy of the card object that is the current "top card" in your deck. You won't actually *remove* the card from your deck, just return a copy of the card and set your "top card" instance variable to point to the next card in the array so that the next card would be returned from the method **deal** if the method was called again. If there are no cards left to deal out from the deck your method should return *null.*

Hand in a copy of Card.java and DeckOfCards.java and submit a copy electronically.

Bonus (2 points): Now that you have a deck of cards to work with, write a program (please call it PlayCards.java) that facilitates a game of cards between two human players (or between one human player and the computer). Your program can play any card game you would like to write as long as it involves more than one card per player and as long as the human players get to make at least one "choice" during the game (i.e. to draw a card, or whatever). Your program must also determine the winner of the card game and print out a message saying which player has won. Include in the comment at the top of

PlayCards.java a description of the game you will be facilitating. Be extra descriptive if you suspect that your instructor does not know how to play your card game. Hand in a copy of PlayCards.java and submit a copy electronically.

Poker (40 points): You will write a class defining a PokerHand object (an object to hold five cards) and provide methods that would be useful if you wanted to write a program to play poker. Here is some background information and definitions in case you are unfamiliar with poker.

Poker Basics: (I found and copied this information from the web at http://members.aol.com/mgoodnight/poker.html)

Poker uses a standard pack of playing cards, 52 cards (there are some poker games that uses more or less depending on the variations such as adding wild cards like jokers). The card ranking is as follows Ace (the highest), King, Queen, Jack, 10, 9, 8, 7, 6, 5, 4, 3, 2 (the lowest).

There are four suits (spades, hearts, diamonds and clubs). No suit is higher than another. All poker hands contain five cards, the highest hand wins.

/* Note for CS 100: We will assume no wild cards for our poker hands. */

The use of Wild Cards depends on the variations, wild cards take on whatever rank or suit you want it to take. A wild card can either be a separate card added like a joker or you may specify a certain card in the standard deck to be wild like deuces, or whatever else.

Here are the rankings and descriptions of the different poker hands, in order from best to worst:

1.Five of a Kind -A five of a kind, only possible when using wild cards, is the highest possible hand. If more than one hand has five of a kind, the higher cards wins, five Aces will beat five kings, which beats five queens, and continues on by the ranking of the cards.

2.Straight Flush-A straight flush is the best natural hand. A straight flush is a straight (5 cards in order, such as 7-8-9-10-J) that are all of the same suit. As in a regular straight, you can have an ace either high (A-K-Q-J-T) or low (A-

2-3-4-5). You can not use the Ace in a wraparound (an example would be K-A-2-3-4, which is not a straight). An Ace high straight-flush is called a Royal Flush and is the highest natural hand.

3.Four of a Kind - Four cards of the same rank like four Aces or Four Kings. If there are two or more hands that qualify, the hand with the higher-rank four of a kind wins. Very rarely, I mean really rarely, if you are playing a game with alot of wild cards, you may have two four of a kinds with the same rank. In this case you use the High Card rule (number 10 on this list).

4.Full House - A full house is a three of a kind and a pair, such as K-K-K-2-2. When there are two full houses the tie is broken by the three of a kind. An example would be J-J-J-5-5 would beat 9-9-9-A-A. If for some reason the three of a kind cannot determine the victor then you go to the pair to decide (this would only happen in a game with wild cards). An example of this would be K-K-K-A-A would beat K-K-K-J-J.

5.Flush - A flush is a hand where all of the cards are the same suit, such as A-J-9-7-5, all of Diamonds. When flushes tie, follow the rules for High Card.

6.Straight - Five cards in rank order, but not of the same suit (it can be any combination of the four suits). An example of a straight is 2-3-4-5-6. The Ace can either be high or low card, either A-2-3-4-5 or 10-J-Q-K-A. Wraparounds are not allowed (an example being K-A-2-3-4). When two straights tie, the highest straight wins, K-Q-J-10-9 would beat 5-4-3-2-A. If two straights have the same value, AKQJT vs AKQJT, the pot is split.

7.Three of a Kind - Three cards of any rank with the remaining cards not being a pair (that would be a full house if it were). Once again the highest ranking three of a kind would win. K-K-K-2-4 would beat Q-Q-Q-2-3. If both are the same rank (only in a wild card game), then the High Card rule come into effect with the remaining two.

8.Two Pair - Two distinct pairs of cards and a 5th card. The highest ranking pair wins ties. If both hands have the same high pair, the second pair wins. If both hands have the same pairs, the high card wins.

9.Pair - One pair with three distinct cards. Highest ranking pair wins. High card breaks ties.

10.High Card - When a hand has none of the above qualifications listed above, nobody has even a pair or better, then it comes down to who is holding the highest ranking card. If there is a tie for the high card then the next high card determines the pot, if that card is a tie than it continues down till the third, fourth, and fifth card. The High card is also used to break ties when the high hands both have the same type of hand (pair, flush, straight, etc).

/* End of material from the web */

Specifications:

1. Your PokerHand object should have a private instance variable that is an array of Card objects of length 5. For this assignment you should assume that your PokerHand would NEVER contain a "Joker" or other "wild card".
2. You should write a default constructor that creates some kind of default poker hand (you choose).
3. You should write a constructor that takes as input five card objects (remember to set your instance variables to a new copy of the input objects for security).
4. You should write a constructor that takes as input an array of card objects of length 5 (remember to set your instance variables to a new copy of the input objects for security.)
5. Add to your constructors a call to one of your sorting methods in Sort. Since your card objects implement comparable, you can (and should) sort your poker hand's array of Card objects using one of your sorting methods (or use mine – selectionSort if you can't get your methods to work.) Having your cards in sorted order will make writing your methods below much easier.
6. Write a toString method that prints out the five cards in your hand separated by a comma and a space. Example: "5 of Hearts, 8 of Diamonds, 9 of Clubs, Jack of Spades, Ace of Spades"
7. You will now write some methods that would help determine "how good" your poker hand is. Write the following public methods that take no input and return a boolean: **hasTwoOfAKind, hasThreeOfAKind, hasFourOfAKind, hasFlush, hasStraight, hasFullHouse, hasStraightFlush, hasTwoPair .** Your methods should return *true* if the poker hand satisfies the conditions for having the hand that the name of the method suggests. If your poker hand has three of a kind, then hasTwoOfAKind should return true and

hasThreeOfAKind should return true also. Likewise if your poker hand has a straight flush then hasFlush, hasStraight, and hasStraightFlush should all return true.

Hand in a copy of PokerHand.java and submit a copy electronically.

Bonus (2 points): Make your PokerHand class implement Comparable. That is, you must write a compareTo method that would compare two poker hands. This will be challenging because you must handle all of the cases to "break ties." For example, if both hands have two of a kind the better hand is determined by the value of the pairs. If those values are the same, then the other three cards in the hand are used to determine the better hand. If the hands are truly a "tie" even after you check the "high card" rules, your compareTo method should return 0. Otherwise your method should return a negative number if the input hand is better than the calling hand and a positive number if the calling hand is better than the input hand.

If you can pull this off I'm convinced that you could write a program to play any card game.