CS 100 Project 2: Bridge Hands
Summer 2001
Due in lecture, Thursday August 2

Objectives

Completing this project will help you:
- Gain experience using multiple sorting routines
- Become familiar with the Comparable interface
- Manipulate one-dimensional arrays

First, skim the whole project. Then carefully read all instructions before starting.

Submission for this project's files will be done ONLINE. To submit your files, go to
http://submit.cs.cornell.edu/cs100/scripts/submit.pl
This link will also be available from the Assignments page of our website. For submission purposes, your username will be your Netid. Your password, however, is **not** the password you use for Bear Access. Instead you should use the password you received via email (to your netid@cornell.edu).

IMPORTANT! We are testing your programs by actually running them with different inputs. This means two things:

a) Make sure your program **compiles** and **runs** before submitting it. If there are parts of your program that are still causing problems, comment them out. A program that compiles and runs is vastly superior to one that does not. We may refuse to grade submissions that do not compile.
b) Your program should **conform exactly** to the specification we provide. We will make available sample output that shows exactly what your program should do, for a given input. Match this exactly so that we can easily compare it to the proper answer. Also, **do not change** any parts of a program that are indicated "do not modify".

1. Bridge Games

Bridge is a card game where four players (two partnerships) attempt to win by making "bids" that indicate how good their cards are and then taking "tricks" (having a card that is higher than the other three players' cards)[1]. Each player is dealt **thirteen** cards (their "hand") from a deck, and then the players enter a **bidding phase** where they are allowed to bid on their hands (and the hand they think their partner may have). When the bidding is done, they then enter a **playing phase** where they play out the hands and see how many tricks the highest bidder was able to take.

To play this game well, you have to understand the rules that dictate which hands are valued higher than others. The rules allow you to compare two hands and determine which one has a higher value. For this project, we'll ignore the playing phase and most of the bidding phase and concentrate on **valuing a bridge hand.**

In this project, you'll implement a the main() method of a BridgeHand class, which will deal bridge hands to a given number of players, sort the players' hands, and then announce which player has the "best" hand. To enable this, you'll also implement the rest of the BridgeHand class and two sorting methods. We will supply the Card class and the DeckOfCards class.

1. **Read** Section 2 of this handout and make sure you understand the bridge hand valuation rules provided there. These are what will enable you to compare two hands and decide which one is better.

2. Download **BridgeHand.java**, **DeckOfCards.java, Card.java**, **Sorting.java, Stat.java** and **SavitchIn.java** from the website. Create a new project and add all six files to it.

- **BridgeHand.java:** This class defines a hand of bridge (thirteen cards). You'll define a **constructor that takes in an array of 13 Cards**. Importantly, you'll be implementing a **compareTo()** method so that two BridgeHands can be compared to determine which should be valued higher. **toString()** has already been implemented for you. See the file for more details about what you need to implement.

[1] This is a massive oversimplification of the game of Bridge.

The **main()** method will start a game of bridge. There are always **four players**, and they each get thirteen cards (this uses up the entire deck). You should create a DeckOfCards, shuffle it, and deal out the desired number of bridge hands. Your program should then sort the player hands and print them out in sorted order, with the best hand listed first.

- **DeckOfCards.java:** This class defines a full deck of cards. The constructor will create a sorted deck by default (as if it had just come from the factory). The class also includes a **shuffle()** method which randomizes the order of the cards in the deck, and a **deal()** method which returns the top Card of the deck. If you deal() without first doing a shuffle(), you'll notice that the cards you get are still in sorted order. After a deck has been fully dealt, it will not be able to return any more cards. You can call shuffle() to reset the deck and then deal out another hand.

- **Card.java:** This class defines what a single card is. Each card has two private variables: **value** and **suit**. Value is an integer from 1 (Ace) to 13 (11 is the Jack, 12 is the Queen, and 13 is the King). Note that these are defined as public named constants in the Card class, so you can use them outside of the class if convenient. Suit is one of the following Strings: "Spades", "Hearts", "Diamonds", or "Clubs". The constructor takes in a value and suit and stores them in the Card object. The class offers four public methods: **getValue()** and **getSuit()**, which return the relevant Card information, **equals()**, which returns true if two Cards have the same value and suit and false otherwise, and **toString()**, which is used when you print out a Card.

- **Sorting.java:** This class will contain multiple sorting methods that you'll implement. You've already seen Selection Sort and Insertion Sort, and we briefly discussed Bubble Sort in lecture. Insertion Sort has already been implemented for you; you'll implement the other two. All three will run in such a way that you'll be able to compare their efficiency by looking at how much work each one does. More details on this can be found in Sorting.java, and will be discussed in lecture next week.

  Java supplies a sort() method which we will also cover in lecture. This will provide you with an excellent way to test your sorting routines and make sure they're sorting correctly; you can compare your sorted results with what you get when sorting with sort().

3. Test your newly implemented methods and classes. Remember you can use the main() method in a class to write testing code for that class. Use all tools at your disposal to track down (and fix) problems: use the CodeWarrior debugger, output to the screen, consultants, etc.

It is **your responsibility** to ensure that you've tested your code appropriately. We will be running our own set of additional "beta" tests on your submitted code. It is to your advantage to try to anticipate what you think we might test, and run those tests before handing in your homework.

4. Submit BridgeHand.java and Sorting.java at the submission website (see above).

2. Rules for Valuing Bridge Hands

For this project, you should use the following method to assign point values to a bridge hand:

- Each Ace is worth 4 points, each King is with 3 points, each Queen is worth 2 points, and each Jack is worth 1 point.
- Being void in a suit (not having any of that suit) is worth 3 points. Having just one card of a suit (singleton) is worth 2 points, and having just two cards of a suit (doubleton) is worth 1 point.

3. Bonus point: Poker Hands

Write an additional class, **PokerHand.java**, which is similar to BridgeHand.java but holds 5 cards (not 13) and does valuation for poker hands. For this class, you should use the following ranking scheme when comparing two poker hands:

- (highest)
- straight flush (all five cards have the same suit and they are a straight)
- four of a kind
- full house (two of a kind **and** three of another kind)
- flush (all five cards have the same suit)

- straight (the cards for a consecutive sequence, e.g. 4-5-6-7-8)
- three of a kind
- two (different) pair
- two of a kind (pair)
- high card
- (lowest)

To break ties, look at the value of the cards involved in the hand characterization. For example, for two hands that both have pairs, compare the value of the card used for each pair ("qualifying" cards). (Suit is never used to break ties!) Aces outrank Kings outrank Queens outrank Jacks outrank tens and so on. If the pairs are of the same value card, check the other ("nonqualifying") cards in the hands. For two full houses, compare the values of the tripled cards first, then the pairs. For straights, compare the highest card in each straight, and so on.

Your main method in PokerHand.java should set up a poker game for five players, then sort the hands and output them in ranked order, just as in BridgeHand.java.

To have your bonus work graded, submit PokerHand.java at the submission website.

Hint: you may find it useful to **sort the cards in the hand first**. Note that the Card class is also a Comparable class, so you can use the sorting methods in Sorting.java to sort, for example, an array of Cards.

You will probably also want to define some private helping functions in the PokerHand class, such as

```
boolean hasPair();
boolean hasTriple();
boolean isFullHouse();
```
etc.