Programming 2 Java Coursework 2015

TITLE: WHIST
SET: 12/11/14
DUE: Thursday Semester 2, Week 2 (22/1/15)
SET BY: A. J. Bagnall
CHECKED BY: G. C. Cawley
MARKS: 25% of the total marks for this module

This coursework involves you designing and implementing a simulation of a card game called Whist. Whist is a game for four players based on taking tricks. You will first implement general classes that could be used in any card game, then implement players for whist that utilise different playing strategies. You should complete part 1 before attempting part 2.

Information on Cards:

There are two variables associated with a card:

Suit: CLUBS, DIAMONDS, HEARTS and SPADES
Rank:TWO, THREE, FOUR, FIVE, SIX,SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE.

Each rank has a value. TWO has value 2, THREE 3 etc. JACK, QUEEN, KING all count for 10, and ACE counts 11. There are 52 different possible cards.

Question 1: implement classes `Card`, `Deck` and `Hand`.
Question 1 is worth 50% of the total marks for this coursework.

Question 2: implement classes `Trick`, `BasicPlayer`, `BasicStrategy`, `HumanStrategy`, `AdvancedStrategy` and `BasicWhist`
Question 2 is worth 60% of the total marks for this coursework.

Question 1.
Classes for Card Games

Design and implement four classes to be used in the card game described in Question 2.

Class **`Card`**

1. Make the class **`Serializable`** with serialisation ID 100.
2. Use two **`enum`** types for **`Rank`** and **`Suit`**. The **`Rank enum`** should store the value of each card. The **`Rank enum`** should also have a method **`getPrevious`**, which returns the previous **`enum`** value in the list. So, for example, if the method is called on FOUR, THREE should be returned. If the method is called on TWO, ACE should be returned. **`Rank`** should also have a method **`getValue`** that returns the integer value of the card as defined above. The **`Suit enum`** method should have a method that returns a randomly selected suit.
3. The **`Card`** class should contain two variables called **`rank`** and **`suit`** of type **`Rank`** and **`Suit.`** It should have a single constructor with the **`Rank`** and **`Suit`** passed as arguments.
4. Make this class **`Comparable`** so that **`compareTo`** can be used to sort the cards into descending order (*see below).
5. Implement accessor methods **`getRank()`**and **`getSuit()`** that simply return the rank and suit.
6. add a **`toString()`**method.
7. Add a static method called **`max`** that uses your **`compareTo`** method to find and return the highest value card in a List of cards. Use an **`Iterator`** to traverse the list.
8. Add two **`Comparator`** classes. One, called **`CompareAscending`**, should be used to sort the cards into ascending order (*see below), the other, **`CompareRank`**, should be used to sort into ascending order of rank, i.e. all the twos first, then the threes etc.
9. Write a main method that demonstrates your code is correct by calling all the methods you have implemented with informative output to the console.

(*) Note that you sort first by suit, then by rank. So a List
10 Diamonds, 4 Spades, 10 Spades, 2 Clubs, 6 Hearts, 3 Clubs
Sorts into descending order as
10 Spades, 4 Spades, 6 Hearts, 10 Diamonds, 3 Clubs, 2 Clubs
i.e. suit order spades, hearts, diamonds then clubs, then by value. To Sort into ascending order, you keep the order of suit by reverse the rank, i.e.
4 Spades, 10 Spades, 6 Hearts, 10 Diamonds, 2 Clubs, 3 Clubs

Class **Deck**

1. **Deck** should contain a collection of Cards of fixed size.
2. Make the class **Serializable** with serialisation ID 200.
3. The **Deck** constructor should create the list, initialise all the cards in the deck and shuffle them. A **Deck** should start with all possible 52 cards.
4. Add methods **size** (returns number of cards remaining in the deck) and a final method **newDeck** (which reinitialises the deck).
5. Write a nested **Iterator** that traverses the deck in the order the cards are to be dealt. This should not use the built in List iterators. Instead, you should keep track of position yourself.
6. Implement a method **deal** that removes the top card from the deck and returns it that uses the iterator you have defined in the previous part.
7. Add a second nested **Iterator** class called **SecondCardIterator** that traverses the Cards in sorted order (as defined by the Card **compareTo** method). You can assume that the deck does not change from the time that an instance of this iterator is created **(**this part of exercise is just to show you understand iterators).
8. Make the class **Iterable,** so that by default it traverses in the order they will be dealt.
9. Write a main method that demonstrates your code is correct by calling all the methods you have implemented with informative output to the console.

Class **Hand**:

1. A **Hand** contains a collection of Cards. The class should provide a default constructor (creates an empty hand), a constructor that takes an array of cards and adds them to the hand and a constructor that takes a different hand and adds all the cards to this hand.
2. Hand should be **Serializable** with serialisation ID 300.
3. A **Hand** should store a count of the number of each suit that is currently in the hand. These counts should be modified when cards are added or removed from the hand.
4. A **Hand** should store the total value(s) of cards in the hand, with ACES counted high. So a **Hand <10 Diamonds, 10 Spades, 2 Clubs>** has total value 22, a **Hand <10 Diamonds, 10 Spades, Ace Clubs>** has total value of 31 and a **Hand <10 Diamonds, Ace Spades, Ace Clubs>** has total value 32.
5. **Hand** should have three **add** methods: **add** a single **Card**, **add** a **Collection** typed to **Card** and **add** a **Hand.**
6. **Hand** should have three **remove** methods: **remove** a single **Card** (if present), remove all cards from another hand passed as an argument (if present) and remove a card at a specific position in the hand. The first two methods should return a boolean (true if all cards passed were successfully removed), the last should return the removed card.
7. Hand should be **Iterable**. The **Iterator** should traverse the cards in order they were added.
8. **sort** method to sort a **Hand** into ascending order (using **CompareAscending**),
9. **sortByRank** to sort descending order (using **CompareRank)**.
10. **countSuit** that takes a suit as an argument and returns the number of cards of that suit.
11. **countRank** that takes a rank as an argument and returns the number of cards of that rank.
12. **hasSuit** that takes a **Suit** as argument and returns a boolean to indicate whether this hand contains any cards of the suit passed.
13. **toString** displays the hand.
14. Write a main method that demonstrates your code is correct by calling all the methods you have implemented with informative output to the console.

Question 2: Whist

You have to implement a variant of the card game whist with different playing strategies. You should use the classes you defined in question 1. You can if you wish add methods not specified in question 1, although try to use the existing ones if possible.

The rules for whist are described here and further details are readily available online.
https://en.wikipedia.org/wiki/Whist
https://www.whist-cardgame.com/

If you are unsure of the rules I suggest you play the online game via the link above after reading the description below.
The rules of your simulation are as follows: whist is a four player game, with two teams of two. Players in the same team sit opposite each other (i.e. a player only plays after the opposition has played). Each player is dealt 13 cards, and one suit is randomly selected as trumps. A round of play is called a trick. Each trick involves a player playing a card, and the other players following with cards of their own. Players must follow suit if they can. If they cannot, they can either trump the card (play any card of the trump suit) which wins the trick (unless another player plays a higher trump card), or they play a card of another suit. The trick is won by the highest trump or the highest card of the suit of the first card. The winner of a trick is the player to play next. After 13 tricks, the team that has the most tricks wins and is awarded a number of points equal to the number of tricks over 6 that the pair has won. Rounds are repeated until one team has seven or more points, at which point a winner is declared.

So, for example, suppose trumps is clubs and we have four players with five cards each.
Player 1:
<Ace Spades, Two Spades, Six Hearts, Five Hearts, Ten Diamonds>
Player 2:
<Three Spades, Queen Diamonds, Ace Clubs, Four Clubs, Two Clubs>
Player 3:
<King Hearts, Nine Diamonds, Eight Diamonds, King Clubs, Three Clubs>
Player 4:
<King Spades, Queen Hearts, Nine Hearts, Jack Diamonds, Two Diamonds>

It is player 3 to play. They lead with the King of Hearts. Player 4 has some hearts but cannot beat the King, so plays the nine of hearts. Player 1 is next, and plays the Five of Hearts. Player 2 has no hearts, so can either trump with a club or discard. They trump with the two of clubs. Hence the trick is
< **King Hearts, Nine Hearts, Five Hearts, Two Clubs>**
Winner: player two
Player 2 is next to play. They play the Ace of Clubs. Player 3 follows with the three, and player 4 has no clubs so discards Two Diamonds. Player 1 also has no Clubs, but also has no trumps, so plays the Six Hearts. Player 2 wins, and the game continues for four more tricks.

Code Structure

You are provided with a structure to follow. Please note this may not be how you would have done it, or indeed the best way. One of the things I want to assess is whether you can understand the structure and follow the requirements. This is a key skill in programming. You are provided with the classes and interfaces described below. **You cannot alter these interfaces.** Classes that implement the interfaces can add methods if required, but try to minimize this as it then makes the game implementation more complex and more tightly coupled.

1. Empty class **`Trick.`** You need to define the data structures and implement the methods for this class.
2. Interface **`Player`**. Each player needs methods to add and remove cards and hands, and to set the game and strategy. The game play is controlled by method **`playCard`**.
3. Interface **`Strategy.`** Contains methods: **`chooseCard`** and **`updateData.`** The strategy should be contained within a player and used to decide on the return value for the method **`playCard`** in interface **`Player`**
4. Class **`BasicWhist`**. This is my skeleton implementation of the game of whist. You can use it as the basis of your implementation if you wish. Alternatively, you can write your own implementation. However, **your game of whist must use the `Player` interface and `Trick` class.** The real complexity of this exercise is getting the modelling of class interaction correct. You need to think in object terms and resist the temptation to tightly couple all classes.

Part 2.1. Implement a simple simulation that uses **`Trick, Player and Strategy`**

1. Define the data structures and methods for the class **`Trick`**. A trick needs to store the cards of the players as they are played, and also needs to know what the lead card was.
2. Define **`BasicStrategy`** and **`BasicPlayer`** classes that implement the **`Strategy`** and **`Player`** interfaces. **`BasicPlayer`** should contain a **`BasicStrategy.`**
3. Implement a simulation of whist and run a game of **`BasicPlayer`** objects**.**
4. Handle incorrect data passing (such as trying to access a null pointer) by throwing exceptions.

Rules of **`BasicStrategy`**

Case 1: First Player
If the player is first to play, it plays its highest card. If there are two or more highest ranked cards, it chooses one randomly.
Otherwise, generally, there are two scenarios. Either it can follow suit or it cannot. If it can it needs to decide which card to play. If it cannot, it needs to decide whether to trump (if possible).

Case 2: Partner is winning the trick
It should check whether its partner is currently winning. If they are, it plays the lowest card it can, following suit if possible, discarding a non-trump if not.

Case 2 Example 1:
Trumps Spades:
Trick so far: (Partner first) Queen Hearts, Two Heart
Hand: Ten Hearts, Two Spades, Nine Clubs
Basic Strategy plays **Ten Hearts**
Case 2 Example 2
Trumps Spades:
Trick so far: (Partner Second) Queen Hearts, King Hearts, Two Heart
Hand: Ten Diamonds, Two Spades, Nine Clubs
Basic Strategy plays **Nine Clubs** (i.e. don't trump your partner!)

Case 3: Partner is not winning or has yet to play
If the partner is not winning or have yet to play it compares its hand to the current best card in the trick. If it can follow suit and can beat the current highest card, it does so with the highest card possible. If it cannot beat the current card it plays the lowest card possible. If it cannot follow suit, it trumps if it can.

Trumps Spades in all examples:
Case 3: Example 3
Trick so far: Queen Hearts,
Hand: Ace Hearts, Two Hearts, Nine Clubs
Basic Strategy plays **Ace Hearts**
Case 3: Example 4
Trick so far: (Partner Second) Queen Hearts, Three Hearts, Four Hearts
Hand: Ten Hearts, Two Hearts, Nine Clubs
Basic Strategy plays Two Hearts
Case 3: Example 5
Trick so far: (Partner Second) Queen Hearts, Three Hearts, Four Hearts
Hand: Ten Spades, Two Diamonds, Nine Clubs
Basic Strategy plays **Ten Spades (trump)**
Case 3: Example 6
Trick so far: (Partner Second) Queen Hearts, Three Hearts, Four Hearts
Hand: Ten Spades, Two Diamonds, Nine Clubs
Basic Strategy plays **Two Diamonds (discard lowest)**

Implement a basic whist game for four **`BasicPlayer`** that follows the structure given below:

Rules of **`BasicPlayer`**

The basic player simply follows the **`BasicStrategy`** for every card it plays.

Rules of **`BasicWhist`**

Player 1 always pairs with player 3.
Game continues until one team has 7 points.
Single hand

1. Choose trumps randomly.
2. Choose first player to play randomly.
3. For number of tricks loop
   - 4.1 Get cards from each player in the correct order by calling **`playCard`**
   - 4.2 Determine the winner of the trick and store this in the **`Trick`** class.
   - 4.3 Update the scores for this game and display
   - 4.4 Send the completed trick to each player with **`viewTrick`**.
5. Display the final score and offer to play another

The basic game is not responsible for checking players play correctly. In principle, a player could follow a strategy that involved cheating (e.g. trumping when they could have followed suit). Write a static method **`basicGame()`** that runs a game of four basic players with appropriate output to the console.

Part 2.2. Human Strategy

Implement a strategy where a human decides what cards to play called **`HumanStrategy`**. The decision of what card to play should be made via the console, and the human should be presented with the relevant information in order to make a decision.

Write a static method **`humanGame()`** where a single human plays with three that use basic strategy.

Part 2.3 AI Strategy

There are many more advanced ways of playing whist. The most important way to facilitate advanced play is to keep track of the cards that have been played. For example, if you know your Jack is now the high card, it may be best to play it now. The other important aspect is attempting to adapt play based on the cards you think your partner has. So, for example, if your partner is out of a suit it may be beneficial to play that suit so they can trump.

Your task is to design an algorithm for playing whist better then to experimentally test whether a player of your advanced players can beat a pair of basic players. To do this you should implement an **`AdvancedStrategy`** class, then write a static method in **`BasicWhist`** called **`advancedGame()`** that runs a simulation of **`BasicPlayer`** objects, two using **`BasicStrategy`** and two using **`AdvancedStrategy.`** Please write a brief description of how your algorithm works and provide graphical evidence of how well it performs against the basic player based on the output of the method **`advancedGame()`**.**`AdvancedStrategy`** can have any internal storage it needs, but can only interface with the player and game through receiving completed tricks via `updateData` and current tricks through **`chooseCard()`**

Note this part of the coursework is intentionally more open ended than the others. I want you to decide on the best way to go about this rather than tell you how to do it.

Submission Requirements

Please read this carefully, as you will lose marks if you do not follow the instructions to the letter. Also remember to comment your code.

Hard Copy (via the hub)

Submit printouts of the following classes **in this order.**
Card.java
Deck.java
Hand.java
Trick.java
BasicPlayer.java
BasicStrategy.java
HumanStrategy.java
AdvancedStrategy.java
BasicWhist.java
Advanced player description document

Each class should be printed in portrait, and should not over run the page. You may have to split some code to do this, but it is good practice. You should print directly from Netbeans, preferably in colour, so that the syntax highlighting is maintained.
Each class should be separately stapled together, and all of them should be put in a folder in the order given above. Do not try to staple them all together. Submit your folder of code printouts to the hub.

Electronic Copy of Code (via Blackboard)

Submit via the Blackboard assignment for this module via the menu item coursework (not by evision).

Submit a zipped Netbeans project via blackboard that contains two packages, called cards and whist. If you do not use Netbeans, I will accept just the source code, but you must put it in the packages. We do not need an electronic copy of the advanced player description.