CMP5015Y: Programming 2

Coursework 2: 2016-2017

TITLE: CHEAT!
SET: 24/11
DUE: Weds Semester 2, Week 3
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 Cheat. You will first implement general classes that could be used in any card game, then implement players for Cheat 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`**, **`Hand`** and **`Deck`** that can be used for a variety of card games, and **`CardTest`** to test the required functionality.
Question 1 is worth 50% of the total marks for this coursework

Question 2: Implement
`BasicStrategy,BasicPlayer,BasicCheat, HumanStrategy,ThinkerStrategy, MyStrategy, StrategyFactory.` for the card game cheat
Question 2 is worth 50% 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 **`getNext`**, which returns the next **`enum`** value. So, for example, if the method is called on FOUR, FIVE should be returned. If the method is called on ACE, TWO should be returned.
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 ascending order (*see footnote). You should make proper use of generics for this.
5. Implement accessor methods **`getRank()`**`and` **`getSuit()`** that simply return the rank and suit.
6. Add a **`toString()`**method that informatively displays a card.
7. Add a static method called **`difference`** that returns the difference in **ranks** between two cards (so the difference between TEN and QUEEN is 2).
8. Add a static method called **`differenceValue`** that returns the difference in **values** between two cards (so the difference between TEN and QUEEN is 0).
9. Add two **`Comparator`** nested classes. One, called **`CompareDescending`**, should be used to sort the cards into descending order by rank (*see footnote), the other, **`CompareSuit`**, should be used to sort into ascending order of suit, i.e. all the clubs sorted by rank, then all the diamonds, then the hearts and finally the spades.
10. Write a main method that demonstrates your code is correct by calling all the methods you have implemented with informative output to the console.

Comparing Cards:
Note that you sort first by rank, then by suit. So a List
10 Diamonds, 10 Spades, 2 Clubs, 6 Hearts
sorts to ascending order as
2 Clubs, 6 Hearts, 10 Diamonds, 10 Spades
and into descending order as
10 Diamonds, 10 Spades, 6 Hearts, 2 Clubs
i.e. for the descending sort the rank order is reversed, but the suit order is maintained.

Class **`Deck`**

1. **Deck** should contain a list of Cards.
2. The **Deck** constructor should create the list and initialise all the cards in the deck. A **Deck** should start with all possible 52 cards.
3. Write a method to **shuffle** the deck that randomises the cards. To gain full marks for this section you should write your own method to shuffle rather than use the built in **Collections** method.
4. Implement a method **deal** that **removes** the top card from the deck and returns it.
5. Add methods **size** (returns number of cards remaining in the deck) and a final method **newDeck** (which reinitialises the deck)
6. Add a nested **Iterator** class called **OddEvenIterator** that traverses the Cards by first going through all the cards in odd positions, then the ones in even positions. So a deck

10 Diamonds, 10 Spades, 2 Clubs, 6 Hearts

would iterate in the order **10 Diamonds, 2 Clubs, 10 Spades, 6 Hearts (**this part of exercise is just to show you understand iterators, it does not require you to clone the deck).

7. Make the class **Iterable,** so that by default it traverses in the order they will be dealt.
8. Make the class **Serializable** with serialisation ID 101. Make it so that the deck is saved with the cards in **OddEvenIterator** order (this may seem a strange thing to do, but it is an exercise to demonstrate you understand Serialization).
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 copies all the cards to this hand.
2. Hand should be **Serializable** with serialisation ID 102.
3. A **Hand** should store a count of the number of each rank and suit that is currently in the hand. These counts should be stored in an array (representing a histogram) and modified when cards are added or removed from the hand.
4. A **Hand** should store the total value(s) of the 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. Note this should still be possible even if the sort routines (part 8 and 9 below) have been called.

Hand should also have the following methods

8. **sortAscending** to sort a **Hand** into ascending (using **Card compareTo**),
9. **sortDescending** descending order (using **CompareDescending)**.
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. **handValue** that returns the total rank values of the cards in the hand (Jack, Queen and Kings count for 10, Aces count 11).
13. **toString** displays the hand.
14. **isFlush** that returns true if all the cards in the hand are the same suit.
15. **isStraight** that returns true if all the cards are in consecutive order (with no duplicates). Thus **10 Diamonds, 10 Spades, 8 Clubs, 9 Hearts** is not a straight but **10 Spades, 8 Clubs, 9 Hearts** is.
16. 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: Cheat!*

(but don't cheat)

Question 2 involves you designing and implementing a simulation of the card game Cheat using the classes from Question 1. See http://bestuff.com/stuff/cheat or http://www.wikihow.com/Play-Cheat The rules for cheat for your implementation are described at the end of this document, any questions please ask on the discussion board. I strongly suggest you actually play cheat, online or amongst yourselves, before attempting this implementation.

You should use your classes from part 1. You are provided with the following interfaces/classes for this part of the exercise. Spend time understanding the structure of the code before beginning your implementation.

1. Interface **`CardGame.`** Simple interface with three abstract methods.
2. Class **`Bid`**. At any turn, a player makes a Bid, which consists of a Hand of cards, and a rank.
3. 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 two methods. **`playHand`** and **`callCheat`**. **`playHand`** is passed the last **`Bid`** made by another player and returns this players **`Bid`**. **`callCheat`** is passed the last **`Bid`** made by another player and returns true if this player wants to call cheat.
4. Interface **`Strategy.`** Contains three methods: **`cheat`**, **`chooseBid`** and **`callCheat.`** The strategy should be contained within a player and used to decide on the return values for the methods in interface **`Player`**
5. Class **`BasicCheat`**: this is my basic implementation of the **`CardGame`** interface that you can use for this question. It does not print out much about what is happening, so you can enhance it in any way you wish, it is just a guide. It is up to you to decide if the game needs enhancing. For example, currently it simply asks the players in turn to call cheat (line 30). This is not a good model as it favours the early players.

*Please note that although the name of this coursework is Cheat! this refers to the game you have to implement, not to how you should proceed in finding a solution. Plagiarism will be detected and punished

Classes you need to implement

Class **`BasicStrategy.`** You should create a new class **`BasicStrategy`** that implements the **`Strategy`** interface provided. Basic strategy should be:

1. Never cheat unless you have to. If a cheat is required, play a single card selected randomly;
2. If not cheating, always play the maximum number of cards possible of the lowest rank possible;
3. Call another player a cheat only when certain they are cheating (based on your own hand).

Class **`BasicPlayer`**: You should create a new class **`BasicPlayer`** that implements the **`Player`** interface provided. The player should contain a **`Hand`**, a **`Strategy`** and a reference to the **`CardGame`** the player is competing in. The basic methods for playing the game are described in the interface file provided. The basic player need store no information about the game.

At this point, **`BasicCheat`** should run. Write a static method to test the execution of the code so far. You may also want to include debugging test harness main methods in each class.

Your next task is to make the game more playable. To do this you have to implement three new strategies, all of which can be used in the **`BasicPlayer`** class. All the strategies should implement the **`Strategy`** interface or extend the **`BasicStrategy`**. Note some of the details are left for you to decide. You should comment your code to describe how it works.

Strategy 1: `HumanStrategy`

The first new strategy should be called **`Human`**. This should interact with the user via the console. The user should make the decision of whether to cheat or not and of what cards to play. The class should check the data entered is valid.

Strategy 2: `ThinkerStrategy`

The second new strategy should be called **`Thinker`**. This strategy should implement the following:

1. **Decision on whether to cheat.** The **`Thinker`** should of course cheat if it has to. It should also occasionally cheat when it doesn't have to.
2. **Choose hand.** If cheating, the **`Thinker`** should be more likely to choose higher cards to discard than low cards. If not cheating, it should usually play all its cards but occasionally play a random number.
3. **Calling Cheat.** The **`Thinker`** should attempt to make an informed decision to call cheat on another player. It should store all of its own cards it has placed in the discard, then examine this record (in conjunction with the current hand) to decide on whether to call cheat. It should always call cheat if the bid is not possible based on previous known play. It should then call cheat with a small probability *p* (set as a parameter) dependent on how many of the current rank are in the current discard pile. The exact way you implement this is up to you.

Strategy 3: `MyStrategy`

The third strategy should be called **`MyStrategy`** and you should decide on how it works. You can do it any way you want, although please make it somewhat different to the other two strategies and make it interesting and non-trivial. Consider using other information about the current hand composition, on other players past bids and on the number of cards the players have remaining.

`StrategyFactory`

The type of Strategy for a player should then be determined by a Strategy Factory class that inputs a String or an enum and returns the correct type of Strategy. Call this class **`StrategyFactory`.** Use the factory to set the player strategies with the player method **`setStrategy`**.

Demonstrate the use of your strategies in the class BasicCheat.

Put the classes for question 2 in a package called question2. Duplicate the classes from question 1, to keep things clear.

The Game of Cheat (called BS in the states)

All 52 cards in the deck are dealt out to the players (requires three or more players). The objective of the game is to get rid of all your cards.

The player with the 2 of clubs starts the game. A player's turn consists of selecting one or more cards to discard face down onto the discard pile. The first person has to declare a discard of at least one 2, the second person at least one 2 or at least one 3, the next at least one 3or at least one 4 etc, until it gets to Aces, at which point it returns to 2. When a person discards they state the number of cards they are discarding and the rank. Neither have to be true. After a player has played the cards and declared, all the other players get the chance to declare "cheat". If cheat is declared, the last cards played are examined by all the players. If declaration was honest, the person declaring cheat picks up all the cards in the discard pile. If the declaration was a cheat, the person who played the cards picks up the whole discard pile. Play continues until someone has successfully discarded all their cards.

So for example, imagine it is player 1 to play next and the three players have the following cards, where 3xK means three Kings. This is a partial deck for clarity, in the game all cards are dealt.

Player 1: 3xK, 2xQ, 4x10, 2x8, 2x7, 4x6
Player 2: 3xA, 2xJ, 2x5, 4x4.
Player 3: 2xQ, 2x6, 2x5, 4x3

And the previous play was 2x7. The next declaration of rank must be 7 or 8. Player 1 has to declare they are playing between one and four 7 cards or one and four 8 cards. This we call a bid. The actual cards they play can be accurate or not. So for example, the player could declare 2x8 but actually play a K and a Q. Let us suppose the player decides not to cheat and plays 2x8. These are removed from his hand and placed on the discard pile. Player 2 and 3 are offered the opportunity to call him a cheat. If they do so, the two cards played are examined and the entire discard pile is passed either to the person who made the call (if indeed they did cheat) or the person who made the challenge (if the call was honest). Whoever picks up the cards is the next to play, and must start again with 2. So for this example, suppose no one calls cheat. Player 2 is next. The last play was an 8, so they have to declare a number of 8s or 9s. Since they have no valid cards to play, they will have to cheat. Suppose they play one nine and neither of the others calls them a cheat. It then moves onto player 3, who has to play 9s or 10s. They cannot play honestly, so decide to get rid of a queen, and call 1 ten. Player 1 knows he is cheating since he has all the tens, so will no doubt call it, and Player 3 will have to pick up all the cards and start again with the 2s.

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
CardTest.java
BasicStrategy.java
BasicPlayer.java
BasicCheat.java
HumanStrategy.java
ThinkerStrategy.java
MyStrategy.java
StrategyFactory.java

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, so that the syntax highlighting is maintained. You should also tidy your code up, removing unnecessary white space and commented out code.
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 (via Blackboard)

Submit via Blackboard.
Submit a zipped Netbeans project via blackboard that contains two packages, called question1 and question2. Duplicate your code from question1 in the package question2. If you do not use Netbeans, I will accept just the source code, but you must put it in the packages.