Assessment 1 (25% of your final mark)

Provisional deadline: 9am, Monday 13th November 2017

For this assessment you are being asked to design, implement and document a simple card game in the Java programming language.

The first round of the game starts with all players having **the same number of cards in their deck**, and the cards are facing down (i.e. not visible to the players); all cards have a number of attributes, each attribute having a value from 0 up to 9 (inclusive).

Every player picks up a card from the top of their deck and looks at it.

The nominated first player then chooses an attribute, say Attribute i, to "play with"; all the players compare the values of this Attribute i on the cards they have picked up.

The player with the highest value wins the round, collects the cards that the other players had picked up and puts them (together with his/her card) facing down at the bottom of his/her deck.

2 The next round starts with every player picking up the card on top of his/her deck and looking at it, and a next player (or random player or the previous winner etc) chooses the attribute they wish to play with.

The game continues in the same fashion for an unlimited number of rounds until *only one player*, the *winner*, has cards left in their deck.

BACKGROUND: A CARD GAME

CardGame is intended to be an application that simulates the above described card game with the following features:

- In the simplest form, there is one human player (the user of the application) and one computer player.
- The user is prompted to enter his/her name.
- After entering their name, the user is then prompted to select the type of computer player they wish to play against. The options are:
  - a predictable computer, i.e., one that always chooses the same attribute in every round in which they play first, or
  - a random computer, i.e., one that randomly (uniformly at random) chooses one of the available attributes in every round in which they play first.

- After the players have been initialized, the game starts and the screen shows the names of the players and the number of cards in each player's deck.
- Then, the name of the player who plays first in the current round (i.e., current player) is shown on the screen, followed by the card at the top of their deck. If the current player is human, then there is a prompt on the screen for them to choose which attribute they wish to play with on the current round.
- After the attribute is selected, the winner is declared and shown on the screen, and the game moves on to the next round.
  Note: if you have ties in a round, simply choose as the winner the player who is closest to the first player that was entered in the game.
- This continues until there is only one player left with cards in their deck, and all other players have 0 cards.

TO DO

Implement the CardGame functionality in Java (for a pass):

Implement the card game outlined above; using objects and good class structure, create objects for the card, attribute player, game and of course the main class/method.

IMPORTANT: It is not necessary for this game to include any graphical output in the form of a GUI. The only output required is to the console/terminal as this is intended to be a text based game.

Card Class: This class should hold the attributes and any other properties of a card in the deck.

Attribute Class: This class should hold the name and value of the attribute; having it as a class will allow you to dynamically create as many attributes as you want for the cards in a deck. This could be useful if for example you wanted to give your user a choice of decks to play with, Bands or Cars for example, where the number of attributes that a card in each deck has might differ (e.g., for a Band you might want to have attributes such as "popularity", whereas for Cars you might want to have attributes such as "speed" or "price").

Player Class: This class should make the correct use of collections to hold a player's deck. It will also contain information about the player, for example the

type of the player (human or computer). It might be useful to also have methods in this class that can perform operations on any collections it might make use of, for example adding or removing cards from the player's deck.

***Extension of Classes*:** You should create classes that extend some of the ones above where appropriate. For example, you could extend the player class to create classes specifically for a human player or different types of computer players.

Game Class: You should create a class that initialises the game and contains all of the logic for the game itself.

Main Class: You should have a Main Class with a Main Method that will instantiate the game. It is good practice to have nothing else present here as it allows you to create attributes and functions that are not static and therefore allow for the correct use of multiple instances.

Must make use of:

- Data Structures, e.g. Queues
- Custom Objects, e.g. Card
- Object Inheritance, e.g., HumanPlayer class extends Player class.
- Take human input from the keyboard.

Document your Java Implementation (Needed for a pass):

Each class, method, field and constructor in your Java implementation should have a javadoc comment, even if its scope is private. You should use `@param` and `@return` tags where appropriate.

MORE BONUS CONTENT FOR A REALLY ADVANCED GAME (for > 80%)

- ✓ You can allow the game to have multiple human and multiple computer players. In this case, you should print a command on the console to ask the user what type they want each computer player to be, i.e., random, smart or predictable.
- ✓ You can set up some intelligence to the computer player(s):
  - A Smart Computer will be able to select the highest valued attribute on their card if it was their turn to select an attribute first.
  - A Predictable Computer will always select, say, the 1st attribute on their card if it was their turn to select an attribute first.

Submission

You should submit files, as described below, through the departmental on-line submission system (click here) in a single zip folder. The names below are only

suggestions, you can use your own descriptive versions. Please also do not include package declarations in your Java files.

1. A file, Attribute.java, containing your Java implementation for the card attributes.
2. A file, Card.java, containing your Java implementation for the cards of the game.
3. A file, Player.java, containing your Java implementation for the players of the game.
4. A file, Human.java, containing your Java implementation for the human player(s) of the game.
5. A file, RandomComputer.java, containing your Java implementation for the random computer player type.
6. A file, Game.java, containing your Java implementation for the game simulation.
7. A file, Main.java, containing your main method (this should only be a *very small* piece of code that creates a new object of type Game and invokes the method that runs the game).
8. If you have any other .java files, please also upload them; You may have more if you deliver the bonus content). *All files should have complete Javadoc comments.*

MARKING

This assessment contributes 25% of your final mark, and will be marked according to how far the following requirements are met:

- The Java code should be laid out according to a consistent format, and it should contain clear comments
- The Java code should correctly implement the functionality set out above.
- The javadoc documentation should be full (one document comment for each class and each method of each class), clear and informative.

A first-class solution (70+%) will meet all these requirements fully;

a 2.I solution (60-69%) will meet most but perhaps not all of these requirements (e.g., the code may not quite implement all the desired functionality, or may lack comments, or have an untidy layout);

a 2.II solution (50-59%) will have some more serious faults (e.g., the code may fall some way short of all the desired functionality, or may contain syntactic errors);

a third-class (40-49%) solution will have serious faults, though it should still show that a decent attempt was made (e.g., code that falls further short of being functional - though it shouldn't be too far away).

A solution getting a failing grade will simply be bad. Failure to hand in a solution will get a zero mark.

It might be possible to gain a higher mark - i.e., move above the 80% threshold by showing some originality and/or creativity - by thinking about what extra functionality might extend these basic requirements.

However it might make sense to think about this *only if you're satisfied that the work you've done on the basic requirements is above the 70% threshold* that you meet by turning in work of a *very high standard*.

In other words, if you don't do a good job on the basic required elements, you're unlikely to improve your marks by spending effort on implementing functionality that isn't asked for.

Note that "originality" and "creativity" could include clear comments or other indications of thinking clearly about the problem; showing reading outside the lecture notes, developing a neat class hierarchy.