Project 2: Card game conundrum

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

(Last updated March 24, 2019)

1 Introduction

Over spring break, my friend and I were playing a card game. The game was kind of a strange mix of Uno and War, played with a standard deck of playing cards. Each player has a hand of 5 cards. The game starts by dealing a card from the deck onto a pile in the middle. Players then take turns playing a card from their hand. The card they play must have the same suit, as the card in the middle, or the same or higher rank. Play goes back and forth until one of the players is unable to play a card. At that point the other player gets one point. The first player to 10 points wins.

As far as things go this is a rather silly game, but there is some strategy to it. In fact my roommate was winning pretty regularly. Therefore I came up with a cunning plan: I would program a version of this card game in java and test several simple AIs against each other. By seeing what types of AIs perform the best I would learn what strategies work the best.

1.1 Learning Goals

This assignment is designed with a few learning goals in mind:

- Implement several special purpose array-based data structures
- Implement a substantial example of polymorphism through interfaces
- Implement several simpler objects in java (you can never have enough practice with the basics!)
- See how AI driven simulations can provide insight into competitive strategies

2 The UnoWar card game

The card game you will be implementing is a rough mix of Uno and War. The game is played with a standard deck of cards. Each player has a 5 card hand. After playing a card each

player can draw another card. When the deck is empty it is re-shuffled (In our computer simulation we will reshuffle with a new set of 52 cards, however, in real-life play only the cards not being used would be shuffled)

Each round of the game starts with a card being drawn from the deck and put in the middle. Then players take turns playing a card into the middle from their hand. A player can only play a card that has equal or higher rank (the number of the card, aces, twos, and threes are low, 10, jacks, queens, and kings are high, and so-forth.) than the card in the middle, or cards with the same suit as the card in the middle. If a player cannot play a card they can pass their turn, in which case the other player wins the round and plays first in the next round. Play goes back and forth in this way until one player has won 10 rounds. The first player to get 10 points in this way is the winner.

We will be implementing and testing three different simple strategies for this game. The first plays a random valid card. While this is unlikely to be a good strategy, it is easy to implement and serves as a baseline strategy to compare others against.

The second strategy is to play the largest valid card every time. The idea here is to finish a round quickly and decisively, intimidating your foe into defeat.

The final strategy is to play the smallest valid card each round. The idea here is to wait and let your opponent exhaust their good cards before you use any of yours. While this may be worse in the short term, it saves up good cards for the long term.

At the end of this assignment you will have a program that can tell you roughly how often one strategy beats the other.

3 Random Number generation in Java

The Java class library provides a class called `Random` that implements a random number generator. You will use it to help shuffle the deck. To use `Random`, you must put the following line at the start of your program.

```
import java.util.Random;
```

You need only `Random`'s constructor and its method `nextInt`. The constructor call `new Random()` returns a new instance of the class `Random`. If `r` is an instance of `Random`, then `r.nextInt(10)` returns a randomly generated int between 0 (inclusive) and 10 (exclusive). You can pass whatever upper bound you want into `nextInt` to generate numbers from 0 to any specified upper bound.

4 Implementation

The design for this program is very typical of an Object-Oriented design. This is to say, it has many classes, most of which are small and serve one well defined purpose:

- `Card` - represents one playing card
- `Deck` - represents a deck of playing cards

- `Hand` - represents a hand of playing cards
- `CardPile` - represents a pile of playing cards
- `UnoWarMatch` - represents a match-up of two AIs at UnoWar (one match-up may be settled by thousands of actual games, this object handles both single games, and multiple game series)
- `AI` - a Java Interface describing what methods an UnoWar AI has.
- `RandomCardAI` - an AI that plays the first valid card it finds in its hand.
- `SmallestCardAI` - an AI that plays the lowest-rank valid card in its hand.
- `BiggestCardAI` - an AI that plays the highest-rank valid card in its hand.
- `Tournament` - a driver class with a main method that reports the win-rate for every possible pair of AIs.

While this may sound like many classes to write, most of them are relatively simple on their own. By splitting the behavior up like this you can also focus on single classes at a time. You could, for example, try to write one (and only one) class a day and be done in a little over a week.

4.1 Card

The `Card` class represents a single playing card. There are MANY ways to represent a playing card in any modern programming language. We are going to represent these with two integers: `rank` (the number on the card) and `suit`. For rank we will use 1 to represent "Ace", 2 to represent "Two", and so-forth until we hit 11 (Jack) 12 (Queen) and 13 (King). For suit we will use 1 to represent Spades, 2 to represent Hearts, 3 to represent Clubs, and 4 to represent Diamonds.

Your card object should be immutable, meaning that once constructed there should be no way to change it. Your Card class can have any other variables or methods you want, but it must have all of the following methods. These methods must have these exact names, and parameter types (although you can name the parameters differently)

- `public Card(int rank, int suit)`
  Constructor - the first `int` should indicate the rank of the card (1 = Ace, 2 = Two, ..., 11 = Jack, 12 = Queen, 13 = King) The second `int` indicates the suit 1 = Spades, 2 = Hearts, 3 = Clubs, 4 = Diamonds. This constructor should validate it's inputs – if an invalid suit or rank is given it should throw an `IllegalArgumentException`.
- `public int getRankNum()`
  This method should return the number representation of the cards rank.

- `public String getRankName()`
  This method should return the string naming the cards rank. (I.E. "Ace", "Four", "Ten", "Queen" etc.) All rank names should be capitalized

- `public String getSuitName()`
  This method should return the string naming the cards suit ("Spades", "Hearts", "Clubs", or "Diamonds")

- `public String toString()`
  Your Card should override the default toString method to one that prints a human readable name for the card. Once written this will help you when using print to debug since it provides a human readable description of the card. Examples of the type of output we are looking for can be found in the test files.

- `public boolean equals(Object obj)`
  You card class should override the default equals method. A Card should only be equal to other instances of the Card class, and then only other cards that have the same rank and suit.

4.2 Deck

The `Deck` class represents a deck of cards. It must use an array of type cards (length 52) to represent the cards and their current order. Other variables may be useful as well.

We are going to implement Deck so that if you draw from an empty deck it automatically reshuffles. In the real world reshuffling a deck of cards would only replace cards not otherwise in use. To make life simpler, however, we will reshuffle a new deck of 52 cards. While this may lead to certain cards being duplicated in our game, it shouldn't substantially effect the quality of our simulation.

Your `Deck` can have any other variables or methods you want, but it must have all of the following methods. These methods must have these exact names, and parameter types (although you can name the parameters differently)

- `public Deck()`
  Constructor - creates a new deck. Makes an array containing 52 different Cards. You must use one or more loops: you will receive no points if you just write 52 assignment statements. The order of Card's within the array does not matter. The last line of your constructor should call your shuffle method so that all decks are shuffled by default.

- `public void shuffle()`
  Shuffle the deck of Card's that is represented by the array you made in the constructor. The easiest way is the Durstenfeld-Fisher-Yates[1] algorithm, named after its inventors.

[1]For more information on this algorithm I recommend the Wikipedia page (`https://en.wikipedia.org/wiki/Fisher-Yates_shuffle`— it is both informative, and currently free of actual Java source code (looking up source code for this algorithm would be cheating, so be careful what you search)

The algorithm exchanges randomly chosen pairs of array elements, and works in $O(n)$ time for an array of size n. You must use the following pseudocode for this algorithm.

Do the following steps for the integer values of i starting from the length of the array minus one, and ending with 1.

1. Let j be a random integer between 0 and i, inclusive.
2. Exchange the array elements at indexes i and j.

- `public Card draw()`
  Draw and return the next card. Whatever card is drawn should not be drawn again until the deck is shuffled again. This should decrease the number of cards remaining. You need not pick a `Card` at random, because the `Card`'s in the array will already be shuffled into an appropriate order. This method must work in $O(1)$ time (unless it needs to reshuffle). If the deck is empty it should shuffle the cards and then deal the new top of the deck.

- `public int cardsRemaining()`
  Returns the number of cards remaining before the next reshuffle. This should return 52 after constructor or a call to shuffle, and decrease with calls to draw down to 0.

- `public boolean isEmpty()`
  Returns whether or not the deck is empty. If this returns true, the next call to draw will trigger a reshuffle.

4.3 Hand

The `Hand` class represents a hand full of cards. To make this class easier to use, we will make the hand class automatically draw cards from a deck as cards are removed from the hand. While we will only use one hand size for this specific program, the `Hand` class should be designed to work with any sized hand. Designing it this way will let this class be reused in the future.

The `Hand` class must represent the hand of cards using an array of cards.

- `public Hand(Deck deck, int size)`
  Constructor. This should create an array to store cards of the given size, and then draw it full of cards using the supplied deck.

- `public int getSize()`
  Get the size of the hand.

- `public Card get(int i)`
  Get the card at the given index in this hand. If the index is 0 it would be the first, card, 1 should give the second card, etc. If an index is given that is out of bounds this method should throw a `IllegalArgumentException`.

- `public boolean remove(Card card)`
  This method should remove a given card from the hand. If the card is found in the hand it should be removed, and a replacement card should be drawn from the deck. In this case the method should return true. If the card is not found in the hand it should return false. You should design this method to work if the input card is null (you can, however, safely assume that the cards in the hand are not null)

4.4 CardPile

The `CardPile` class represents the pile of cards that players play onto, and is where several of the rules of the game are implemented. We *could* use an array or linked stack to represent the card pile (cards are always placed on the top, and only the top is referenced). However, if you think carefully you may find that this is more work than is needed for the behavior of this object, given that we never need to reference anything but the current top card.

The methods the card pile should implement are described below:

- `public CardPile(Card topCard)`
  Constructor. This should create a new card pile with the given card as the initial top card.

- `public boolean canPlay(Card card)`
  This method should check if the input card is legal to play on the current stack. As a reminder of the rules for this: a card can be played if it has a higher rank than the current top card, has the same rank as the current top card, or if it has the same suit as the top card.

- `public void play(Card card)`
  Adds another card to the card pile, making this the new top card. If the input card is not legal to play on the top of the card pile, this method should throw a `IllegalArgumentException`.

- `public int getNumCards()`
  Gets the number of cards in the `CardPile`.

- `public Card getTopCard()`
  Gets the current top card for this `CardPile`

4.5 AI interface

The `AI` (short for artificial intelligence) interface should be a java interface, not a class. This interface describes all the behavior needed to represent a strategy for the UnoWar game. Classes that implement this, therefore, are AIs for the UnoWar game. If you want, you could also implement a human interface to this game as an implementation of the `AI` interface.

The AI interface has one method:

- `public Card getPlay(Hand hand, CardPile cardPile)`
  This method takes two parameters, the hand, full of cards the AI is allowed to play, and the cardPile that the AI is playing on. The AI should pick a card from the hand and return it to mark it as the card the AI intends to play. The AI can return null to indicate that they have no card that can be played on this card pile. The AI is not responsible for removing the card from the hand – this will be managed by another class. The AI should not return a card that is not in the input hand.

While this does not need to be indicated in the AI interface (it certainly can, but it doesn't have to) each AI should also implement the `toString` method.

4.6 UnoWarMatch

The `UnoWarMatch` class contains the majority of the code involving playing a game of UnoWar between two strategies that are implemented as instances of the `AI` interface. Broadly, this class does not represent one game of UnoWar, instead it represents a match-up between two AIs for this game. Therefore it not only has a function to play a single game of UnoWar, but also a function to play many games of UnoWar and compute the win-rate of one AI over another.

Note, the only instance variables for this class should be two AI objects. While there are other attributes such as a deck of cards, hands, or win/loss rates that you will be tempted to make instance variables, all of these are essentially local to one single method. Adding these as instance variables will make your life much more complicated, and make it much easier for you to write incorrect code.

While we only mandate the following three methods for UnoWarMatch, we recommend that you make several other private methods. The process of playing a single game of UnoWar is complicated enough that decomposing it into methods, or possibly even further classes, may be useful.

- `public UnoWarMatch(AI ai1, AI ai2)`
  Constructor. This takes the two AIs that this `UnoWarMatch` class is intended to compare.

- `public boolean playGame()`
  Play a single game of Uno War. This should play the UnoWar game as described earlier in the write-up until one of the AIs has won 10 rounds. The return value should be true if ai1 wins, and false if ai2 wins.

  As a brief reminder: The basic flow of the game is done in rounds, with each round being worth one point. Each round the AIs take turns playing a card from their hand into the card pile. The round ends when one AI has no valid card in their hand, at which point the other AI gets a point, and plays first in the next round.

  Some details on this method:

  – This method should start by constructing a new deck, hand, and card pile objects for the current match-up. (These objects should not be re-used game-to-game).

- This method should make sure cards are removed from the AIs hands once they are chosen for play. We do this here instead of in the AI class so that we can't write a cheating AI.
- Remember, the AI can chose to play null if no card is valid.
- AI 1 should play first in the first round.

- `public double winRate(int nTrials)`
  This method should have the AIs play eachother `nTrials` times, and report the percent of times AI 1 beat AI2 as a double. The return value should be between 0 and 1 inclusive. Since the winner of this game is partially determined by chance (the best AI can't win if it only gets bad cards) the game may need to be repeated thousands of times to get a precise estimate of the chance of one AI beating another.

4.7 AI Implementations

You are required to implement 3 further classes implementing basic AIs. All three classes should implement the AI interface. None of these classes should need any instance variables. All three interfaces should implement the AI interface method, and `toString` with toString simply returning the name of the class: "Random Card AI", "Smallest Card AI", and "Biggest Card AI".

The three AIs and their expected behavior as as follows:

- `RandomCardAI`
  This AI should play a random valid card from its hand. Since we do not sort the hand in any way, one easy way to implement this is simply to return the first valid card. While we don't expect this strategy to be particularly effective, it is traditionally to compare against the "random baseline" to help understand if an AI implements a strategy that is worse than playing with no strategy at all.

- `SmallestCardAI`
  This AI should play the lowest-rank valid card in its hand every time. (Ties can be broken arbitrarily - we do not require any specific policy)

- `BiggestCardAI`
  This AI should play the largest-rank valid card in its hand every time. (Ties can be broken arbitrarily - we do not require any specific policy)

4.8 Tournament

This is the driver class for this program, meaning it only needs to have a main method. The main method should print the win rate of every pair of AIs. The output of my program is given below: Your code does not need to output this exactly (infact, I would be surprised if it did, as estimated winRate is somewhat random) but it should contain all this information

```
Random Card AI vs. Random Card AI winRate: 0.499
Random Card AI vs. Smallest Card AI winRate: 0.002
Random Card AI vs. Biggest Card AI winRate: 0.842
Smallest Card AI vs. Random Card AI winRate: 0.998
Smallest Card AI vs. Smallest Card AI winRate: 0.499
Smallest Card AI vs. Biggest Card AI winRate: 0.999
Biggest Card AI vs. Random Card AI winRate: 0.156
Biggest Card AI vs. Smallest Card AI winRate: 0.0
Biggest Card AI vs. Biggest Card AI winRate: 0.491
```

5 Limitations

There are a few limitations for this project. First and foremost, as usual this is NOT a collaborative project. You should not show your code to anyone, or look at any other code for this problem, nor should you closely discuss solution details with anyone else. If you need help on any part of this project please email the course staff and we can provide assistance.

Secondly, Card and Deck classes are not particularly uncommon programming assignments. You should be very careful if you search for assistance with this project not to search for implementation of these classes. Looking at ANY other implementation of these classes will be considered a breach of academic integrity. If you need help on any part of this project please email the course staff and we can provide assistance.

Thirdly, you should not use any of the standard java library classes other than Random and String. In particular you are forbidden from using the java collections (I.E List, Stack, Queue, etc.) the goal of the `Deck`, `Hand`, and `CardPile` classes is to implement these from scratch using only arrays. The only standard java classes you are permitted to use are `java.util.Random` and `String`. If you want to use any other pre-built class please email Daniel and ask for permission, I will address these on a case-by-case basis. While this is not perfectly reflective of industrial programming, using the pre-built classes defeats the purpose of the assignment in this case.

6 Testing and incremental development

While this project has a substantial number of things to program. Each of these can be tackled relatively individually. The idea behind a design like this is to spread the total complexity of the program enough that no one part is actually that complicated. Therefore you should plan enough time to write and test each method from each class. But in doing so remember, that you can focus on only one class a time. So long as it works correctly, you will not need to worry too much about how other classes use it.

To make this process easier I have uploaded five test drivers for various parts of the program. While these do not test every behavior that I expect, they should serve as a good starting point for your own personal testing process. **These are not intended to replace**

your own personal tests. You should test every piece of code you write, using these files as a base.

Since some of these tests rely on the behavior of other classes (you can't test Deck without a working Card class, for example) there is an intended order to these tests. The intended order of completing these tests is as follows:

1. `CardTest.java`
2. `DeckTest.java`
3. `HandTest.java`
4. `CardPileTest.java`
5. `AITest.java`

Note, these tests only cover some of the simpler classes. You will want to design your own tests to be sure the behavior of the final few classes is correct.

7 Deliverables

For this project you should submit the following files:

- `AI.java`
- `BiggestCardAI.java`
- `Card.java`
- `CardPile.java`
- `Deck.java`
- `Hand.java`
- `RandomCardAI.java`
- `SmallestCardAI.java`
- `Tournament.java`
- `UnoWarMatch.java`

As well as the file for any helper classes you may write. We should be able to run your code given only the files you submit, so do not forget to include any necessary file.

I've been told that canvas does allow multiple file submissions, However, to make sure there are no errors on our part in managing these files, we ask that you submit a zip file

containing these files. Please email or attend office hours if you need assistance submitting a zip file.

Please name your submission `<YourName>_Project2.zip` before uploading it to Canvas. (We will handle unzipping the files and renaming the file for testing as needed.) This will help us resolve any issues with identifying files during automated testing that might arise. Your work must be submitted on Canvas by Monday, April 15th at 6:00PM.