King's Corner

Design Document

Introduction

My name is *Rafael Grigorian* and I have implemented the classic card game *King's Corner* for my *Software Design (CS342)* course which I am currently taking at the *University of Illinois at Chicago*. This program is written in Java and uses a command line interface. This program is currently in *v.1.0.0*. The specific rules for the game that this program follows can be found in the link below this paragraph. This program uses ASCII art to make the game more visually appealing since it is only a command line interfaced game. For further information or assistance, one can reach me using the specified email below.

Prompt:	*https://www.cs.uic.edu/pub/CS342/AssignmentsS16/proj1.pdf*
Rules:	*http://www.pagat.com/domino/kingscorners.html*
Github:	*https://github.com/rdogg312/KingsCorner*
Email:	*rgrigo3@uic.edu*

Section #01 - Purpose

The purpose of this project is to break down the interface and functional components of the *King's Corner* game in a way that is self contained in necessary classes. This program is very susceptible to modularization using Java classes. The audience of this game is anyone who is interested and will be available for download on my Github account. It is important to see how each class works together as a subsystem to properly function as intended by the creator.

Section #02 - Modularized Classes

The following will give the reader a high level construct of what each class that is confined within this program does. For more detailed information refer to the *Inter-relational Concept* and *Intra-relational Concept* sections below.

KingsCorner.java: This class contains the main function and is the driver for the game. It contains the very abstract game loop that is needed to play multiple games, multiple rounds, and even to switch turns.

Board.java: This class contains all the game's data object instances within it's data members. The players, card piles, and everything to do with this game is held within this class.

Player.java: This class contains the user's hand, score, and function's that will be used to validate user input as well as preform various moves.

Computer.java: The Computer class extends from the Player class and it contains the same data members. Although the Computer class overrides the turn functionality that the Player class provides. It implements an AI system in order to determine what to do. This is important in the scheme of modularization so we can swap a computer player and human player interchangeably. This gives the program more room to grow in the future with less effort and chas.

CardPile.java: This class contains a pile name and size data member and is also implemented to act as a list that points to the top and bottom of nodes, or Card instances. It also contains functions similar to what regular linked list might have.

Deck.java: This class implements the CardPile class and contains all functionality of it. It also extends the functionality of a card pile by implementing a shuffle algorithm as well as a dealing function which pops and returns the top card of the deck.

Card.java: This class contains information about the cards rank, suit and color. This class acts like a node in a linked list. The list portion being implemented in the CardPile class.

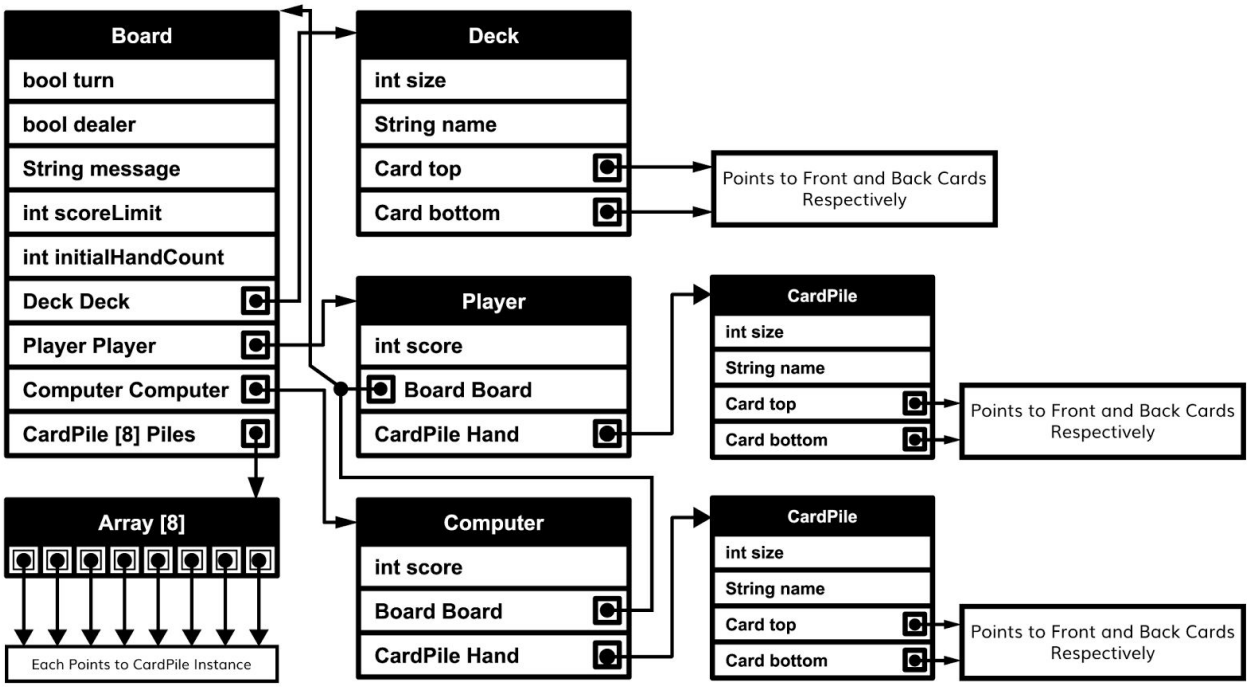
Rank.java: This class is a simple enum that assigns a card a rank. This rank is given in the conventional order of 'Ace is low', meaning that the ace counts as the lowest card in the game. Additional functions are there as static in order to convert to/from symbolic string and the Rank enum.

Suit.java: This class is a simple enum that assigns a card a suit. These suits include, in order of rank: Spades, Hearts, Diamonds, and Clubs. Additional functions are present to convert to/from symbolic strings and the Suit enum.

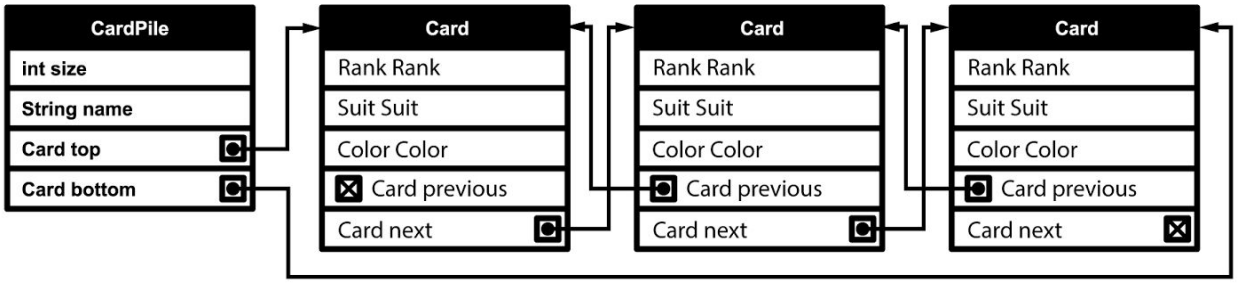
Color.java: This class is a simple enum that assigns a card a color. This enum is usually predefined within the Card class, since it needs to correspond correctly to a color. It also contains functions that convert to/from symbolic strings and the Color enum.

Section #04 - Model

Below we can see an instantiated occurrence of the game board. The Board class holds handles to the rest of the instantiated objects involved in this game such as deck, player, computer, and all eight card piles. The parts that are cut off in the preceding figure below are not present for the reason of abstraction, and repetitiveness. An example of the linked list structure that Card has with CardPile and Deck are shown in the figure below this one.



As mentioned above, the below figure shows the relationship between Card instances and CardPile and Deck instances. The Card objects act like notes in a linked list, where the Deck and CardPile objects act like the list that contains pointers to the front and back of the list, as well as list size and other miscellaneous information.



Section #03 - Inter-relational Usage

CardPile.java / Deck.java: As seen previously these classes implement a linked list LIST struct. This was very useful because I was able to arbitrarily handle and swap cards inter-relationally between data structures without having to worry about memory allocation. Other such data structures like arrays may need to manage memory, or have an extra abundance of unused memory that would be allocated. This approach eliminates that, and uses only the amount of memory that is necessary for the card piles to behave as they are supposed to inter-relationally.

Card.java: This is implemented to behave as a node, and it fits perfetly within the functions that are needed for a deck to have. A card pile must be able to be shuffled easily, behave as a stack at certain roles, and be able to be merged with ease. The list-node approach is definitely a superior implementation for this game.

Player.java / Computer.java: Since the Computer class implements the Player class, it is very easy for me to test the game by putting two AI computers to play against eachother and everything should run smoothly. Since they are build to have the same core functionality, but one uses an algorithm instead of user input, it is easy to see how this is a very modular approach to the problem. This is also very useful because we are able to add players more easily with this approach and we are able to scale up the game in terms of player sizes. One potential risk is that the user input is not parsed in the most secure way and security flaws might come into play.

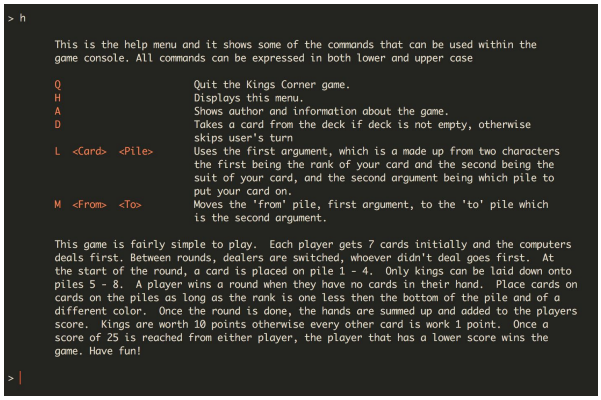
Board.java: This class is basicly the object that holds all other instances of other objects that relate to the game. It also operates on the game using functions that control the other components from a high level of abstraction, since it uses other user defined functions of said instances. This class will also be initiated within the KingsCorner class as a local variable.

Color.java / Suit.java / Rank.java: These enums are very useful in the sense that we can change an entire symbol through out the whole program and we only need to change it in the respective file. This allows for a simple upgrading procedure as well as a well organized class structure.

Section #03 - Interaction & User Input

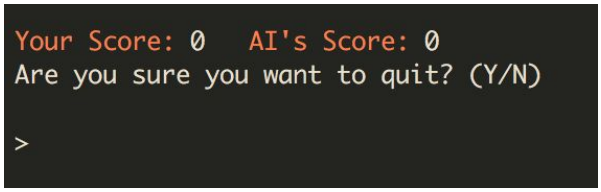
After compiling the program, you can run it by typing “java KingsCorner” into your console. The game should initiate automatically and begin to deal out the cards. You will notice that the game animates the dealing of the cards after anything on the board changes. After the board is rendered and all cards have been dealt out, then the program will ask you for user input.

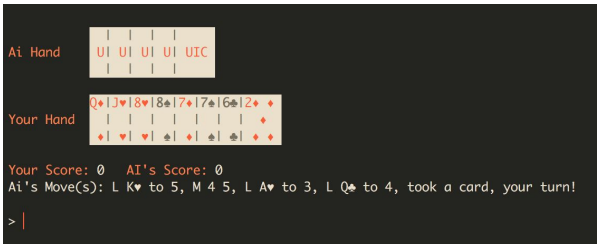
Pressing ‘h’ or ‘H’ when prompted for user input, you will display this screen to the user, prompting them for every possible command that is valid for input to the program.



Pressing ‘a’ or ‘A’ when prompted for user input will display the figure on the left. It displays to the user information about the author and project.

Pressing ‘q’ or ‘Q’ will prompt the user with a confirmation prompt asking if they are sure they want to quit. The result whichever input is obvious.



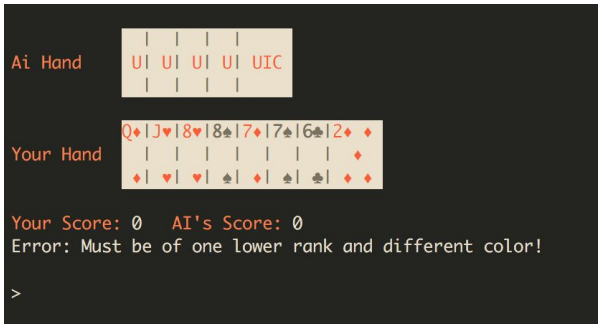


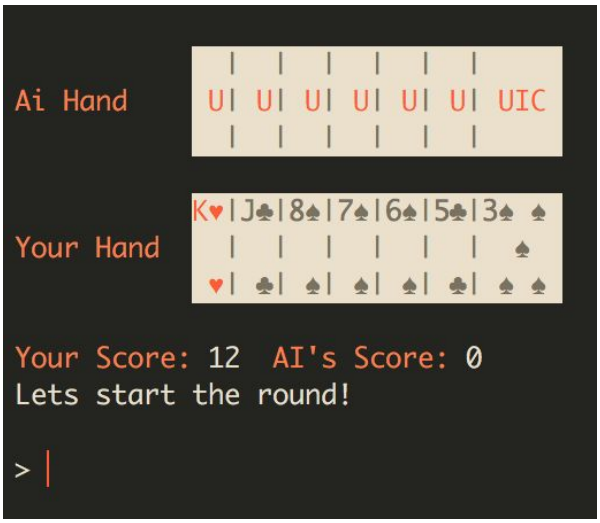
Pressing ‘d’ or ‘D’ will end your turn and take a card from the top of the deck. The message section on the screen will display every move that the AI computer has made. When deck is empty, then you just end your turn.

Pressing ‘l’ or ‘L’ followed by the card you would like to place from your hand and then the pile number as the second argument will place said card from your hand onto said pile if valid.

Pressing ‘m’ or ‘M’ followed by a from pile number, followed by a to pile number, will merge the from pile to the bottom of the ‘to’ pile.

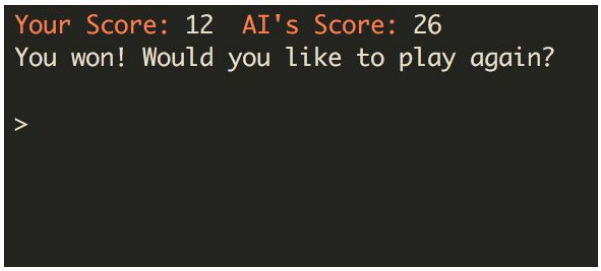
Inputting invalid user input, whether it is syntactic or semantic, an error message will show up in the message section of the game board.

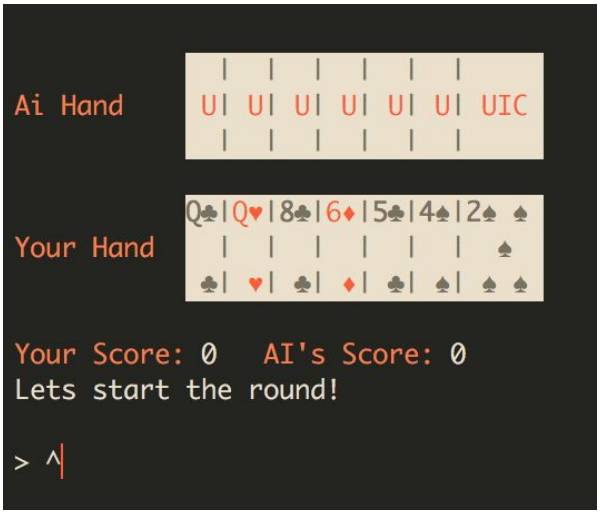




The score is displayed on top of the message section of the game board and displays the amount of penalty points a player has.

When a game finishes, the program will ask to restart the game. If you wish to play another game, everything will be reset as long as you input 'y' or 'Y', otherwise it will assume you don't want to. When this prompt is displayed, the final score should also appear on the screen.





Secret debugging mode! When playing a round, if you input '^', the game will replace you with another AI computer, and will finish the round for you. It may or may not win. It will keep the current score that you have going and will stop when the next round starts. This is used for debugging primarily, but is also a fun feature to watch play out.

Section #04 - Benefits and Risks

The benefits of using this design approach are as follows:

- Due to the modular approach of this program, it is easy to edit and update/upgrade the software in the future.
- The linked list structure that the Card / CardPile / Deck classes implement make the memory work efficiently since we only use what we need and is very compact and fast. It is also a benefit because there is no need to copy data across memory, we can just modify pointers.
- Since the Player and the Computer class is very modular and extend from one another, it is easy to create more dynamic player situations like AI v. AI. It is also much easier to scale up the game to include multiple players by just modifying the main game loop in KingsCorner.java.
- This design approach is very scalable and is easy to make the game bigger when talking about player size, and also more easy to implement more features since all classes contain all functionality about said class within itself and is not randomly scattered across multiple files.

- Easy to change symbols across the whole game since I use enums for color, suit and rank. because of this I would just need to go to one file to make a modification across the whole game.

There is also one notable security risk:

- Because there is user input from the human player, It is always a risk that many security breaches may occur from improper data parsing and insufficient safeguards.