Go Fish:

An Array List

Implementation

Noah Constable

Kyle Day

Cole Maxwell

CSCI 2101, 001

Hussam Ghunaim

November 20, 2020

# Overview

For our final project, our group chose to replicate the game of GoFish in Java through the utilization of the ArrayList data structure. ArrayList was designed by Oracle and is one possible way of storing data in arrays. We applied this structure to the deck of cards that human and AI players draw cards from, the hand in which players store their cards in, and the pool which is used to generate random names of AI players. Throughout this report, we may refer to ArrayList as simply an array, but note that in this case, they are interchangeable.

# In Concept

In the beginning, we considered doing either a hospital simulation or a card game for our final project. Deeming card games to be more fun and intuitive, we then went on to choosing which game to implement with the data structures presented in the course Go Fish was our final choice, as it is, on the surface, relatively simple to play. We could use stacks to simulate a deck of cards, since in the game Go Fish, cards are drawn from the deck from the top, just as a stack is in a computer program. Meanwhile, we could use bags for the hand, since in real life, drawing from and adding to the hand is done in an unsorted manner. These two structures, combined with the various steps involved in playing Go Fish, presented a challenge that we were excited to undertake.

Once we agreed to move forward with this concept, we made a brief outline of the project. The game would be played by four players, three of which are AI, and one of which is human. We decided not to implement human multiplayer for two reasons. Firstly, due to the amount of information that would be printed out in the console, the game would become quickly cluttered with all of the statements required for a human to understand what is happening in the game. Secondly, there would be no way to discreetly show players their cards; cheating would be fairly easy, as players can scroll up and look at each other's cards. With AI players, however, the player would not be required to see their cards in order to play against them. Meanwhile, the AI players would only need to know their values in memory, not needing a visual aid to tell their information apart from the human player’s.

Proceeding forward, we needed to decide on the basic rules of the game. Our initial concept of these rules was that the AI would ask for the card that it needs the least to complete a book, and that it would ask the player that previously asked it for cards. We knew we had to define a new player class for these and we also needed a way to define cards. We tossed around the idea of creating a class to create a card object with a value and suit, and importing the stack structures we’ve worked on previously to simply store the cards in one.

With a player and card class outlined, we also outlined a general GoFish class, which would drive the main gameplay and turn mechanics. At that point, we had planned on including an interface, not having considered running the game in that class at that point in time. Furthermore, in the outline, we shifted from using ArrayBags to ArrayLists, providing a more efficient means of storing the cards in the hand. By the time the outline was fully ready, we had ideas for the player class, the card class, the Go Fish/driver class, and how we would integrate everything. We set out to work.

# In Practice

Quickly after beginning to code, we realized that we could restructure the project to make implementing some aspects of the game much simpler. The deck was changed to utilize an ArrayList instead of ArrayStack, keeping structural method calls more consistent with that of the player’s hand.Furthemore, we simplified the card class to use enumeration. As a result, suits were no longer accounted for, since the enumeration is based on the card values alone. The change was largely inconsequential, however, since suits only serve as a visual way of differentiating between cards in the real life version of Go Fish. They do not alter the value of any card or have special stipulations in play.

Additionally, we rethought our approaches to coding the player and GoFish classes, as we realized that most methods we thought were going to belong to the GoFish class would need to belong to the player class. That being said, when we began shifting some GoFish methods over to the player class, we ran into a problem. If we used an interface, we would have to rewrite the code twice, since both the AI and human players would share the same logic in terms of giving and asking for cards. At the same time, the AI and human players were inherently different, and required different methods to be used for taking turns. After doing some research, Cole proposed the idea of using abstract classes. Unlike interfaces, abstract classes allow for both method headers and fully implemented methods to be written within the main class, letting subclasses inherit some logic while having the freedom to utilize other forms of logic when necessary.

After making this new abstract Player class, we created two subclasses: the AIPlayer and HumanPlayer classes. These two classes define different types of players, a human player and an AI player, respectively. Some methods, such as the name methods, fish(), giveAll(), and askFor() are the same. The main distinction in the subclasses arises when considering that an AI player needs to query the computer to determine what actions to take, while the computer queries the human player to determine the outcomes of their actions. For this reason, both of these types of players have different haveTurn() methods.

For each round in the game, every player has a turn. While having their turn, a player decides who to ask for a card type, what card type to ask for, and receive a card at the end of their turn, either through taking a card from the player they asked or drawing a card from the deck, if the deck is not empty. All of these steps have countermeasures against invalid inputs as well, preventing human players from calling on players or card types that do not exist. Meanwhile, the AI player accomplishes their tasks by randomly generating an index and then checking that index against the array of players in the game. As long as it has not generated its own index, it targets the player at that index. To decide what card to ask for, it simply picks a random card from its own hand and asks for that card type from a random player.

However, the AI’s method of taking a turn was not always this way. In an earlier phase of the project, we chose to implement the AI’s turn by having it select the last player that asked it for a particular card. The AI would then ask the player for that card. As we play-tested the code in the GoFish class, we discovered that the AI’s actions were rather problematic. Having the AI constantly ask the last player for the card they asked made the AI predictable and very prejudiced against a particular player. Additionally, asking for the card that would complete its books would get very repetitive if it did not get that card.

# In Testing

Over the course of working on the project, we tested and debugged the project through the use of the GoFish class, which contains a main method used to run the whole project in one location. Due to widespread use of the driver’s static methods and the player methods in the other classes, it was not always possible to create J-Unit tests for the classes. Ultimately, the only test class of this kind is related to the NamePool class, testing to different cases for file sizes in the case where the user modified the text with more or less names. Since the makePool() method does not rely on external classes within the project for code, we could safely create a J-Unit test method for it. In regards to the driver class, we began testing the methods of the Player (with subclasses), GoFish, and Card classes with simply a human player and one AI. This way, we can easily play the game and query the computer for different cases of input. Additionally, we could also plainly see if the AI was doing something wrong, since it prints out its turn and fishing actions. At the very end of the game, we could also see if the results of who won were accurate or not. There are three cases that have to be considered for this: a win for the human, a tie, and a loss for the player. With just two players, we could intentionally perform well, perform to match the AI’s progression, and perform poorly to reach each of these cases.

Once we added two more AI players, we focused on testing how names were used by the game. Initially, we had hard coded three names (Stu, Pat, and Virginia) into the code, but with the NamePool class, we chose the option to randomize the names. We ended up implementing the pool class into the driver, and tested to see if the AI’s names were different by examining the printed messages produced in the console. This was the time when we modified the player classes to examine the names of each player, rather than basing their string names on ordinal position within the ArrayList.

# In Total

After testing, debugging, and fine-tuning the code to our liking, we arrived at the end product: a project consisting of six primary classes, one J-Unit test class with 3 and 54-name .txt files for testing, and one .txt file with fifty-two names. Here’s how the project works in its finest form: when the game starts, a deck of fifty-two cards is created using randomization and the enumeration. The deckSize is counted and set aside for the fish() method to modify. Four player objects are created and added to an array of players, one HumanPlayer and three AIPlayers. In this step the player’s hands are also dealt. Some text appears and prompts the player to enter a name for themselves, and then confirm it. That name is assigned to the HumanPlayer object while the AI are randomly generated a name from the list of fifty-two. Some more text is printed out saying the other player’s names and the game begins. The human goes first. The game checks if they contain enough of any type of card in their hand to create a book at the beginning of every turn. If they do have enough, a book is created. This means that in reality the cards are just deleted from the player’s hand and their score is incremented by one. Next, they must choose an AI player and a card from their hand. That player is asked for that card, if they have it it is transferred to the recipient’s hand and deleted from the donor’s hand. If the donor does not have the card asked for, the recipient draws one card from the deck and their turn ends. Now it is the AIs’ turns and they all go sequentially. First, the AI randomly generates a target by picking an index that is not their own using Math.random(). Then it selects a card in its hand, again using Math.random(). It then takes its randomly generated target and randomly generated card and asks that target for that card. If the donor has the card, it is given. If it does not, the card is not given, just like with the human player. This aspect is the same for all players. At the end of the turn, given the deck is not empty, the player draws one. This iterates through all the AI players and back around to the human player where the cycle repeats. During the game, the player is shown their hand by way of printing out the cards in their hand array. They are also shown what cards they drew from the deck, what cards they received from the AI, what cards the AI asked for, and what cards the AI took from each player. It does not explicitly tell the player what each player’s hand is, but just like regular Go Fish if the player is paying attention they can figure out pieces quite easily. Once all thirteen possible books are created the game ends and each player’s score is compared. The player with the highest score wins, unless there is a tie. After the winner is declared and the score is displayed the game prompts the user if they would like to play another game. If the user accepts the whole cycle repeats and if they decline the program terminates.

# Contributions

Truly, the project is a result of cooperative, cumulative work. All of us were involved in testing each aspect of the game, and we modified the code as needed to produce the best results, regardless of what we initially decided to work on. However, understandingly, one may ask which parts of the project were introduced by each person. The abstract Player class and its subclasses were begun by Cole, who saw the potential in using shared and differing logic to fulfill the player’s tasks in the game of GoFish. He also introduced card enumeration. From there, Noah introduced the turn logic, AI logic, and considered the inputs that the player might possibly give the game at different times. Meanwhile, Kyle created the NamePool, its test class, the player creation and name assignment processes in the GoFish class, along with the looping of the game, and the processes that take in input from the player. As stated previously, from these introductions, we all worked to bring our individual aspects of the code closer together, producing the best game we believe that we possibly could utilizing the ArrayList data structure.

As an external resource, we referred to the Bicycle Playing Cards [guide](https://bicyclecards.com/how-to-play/go-fish/) to Go Fish in order to get a better sense of what we needed to account for in the game. Additionally, the ArrayList, Random, Scanner, and file utilities all came from Oracle, the corporation responsible for Java. Meanwhile, the JUnit team is responsible for the testing framework that we utilized to test the NamePool class.

# In Depth

In this section, we will explain some of the methods found within the project in depth to better explain their internal processes.

**Player.getName()** - gets the name of the player.

**Player.setName()** - sets the name of the player.

**Player.hasCard()** - returns whether or not the player has the given card type in their hand.

**Player.giveCard()** - iterates through the players hand and returns every instance of the given card type. Creates an array *givenCardArray* for the cards to be given away and iterates through the player’s hand. Every time it comes across an instance of the given card type it adds to the *givenCardArray* and deletes it from the player’s hand. Returns the array of cards.

**Player.askFor()** - takes a card and value and uses giveCard() to ask that player for that card. Iterates through the array returned by giveCard and adds those cards to the current player’s hand, then returns true. If there are no cards to add, returns false.

**Player.fish()** - draws a card from the deck and adds it to the player’s hand. Checks if the deckSize() is greater than zero, and draws a card if it is. Then adds this card to the players hand and stores it for the return value. If the deckSize() is not greater than zero, prints out that the deck is empty. Returns the *fishing* card, which is null if the deck was empty.

**Player.getNumBooks()** - returns the *numBooks* value for the player.

**Player.checkForbooks()** - checks the player’s hand for enough of a card type to create a book. Iterates through the hand of the player and for each value creates a *num* integer. Every time it encounters this value in the hand it increments the *num* counter by one. If the counter reaches four it creates a book, deleting those cards from the hand, incrementing the *numBooks* value of the player, and returning the card type that created the book.

**AIPlayer.haveTurn()** - simulates the turn from the AI’s perspective. Prints out who’s turn it is, checks their hand for books, prints out if they got a book, prints out if their hand is empty (only if it is empty), calls aiRequestLogic(), calls aiRequestTarget() with the argument of the index of the current player, prints out who they ask and what card they asked for, calls askFor() with the target player and card, prints out if they received cards from that player, and finally lets the AI go fishing.

**AIPlayer.aiRequestLogic()** - randomly selects a card from the current players hand to ask for. Uses random to use nextInt() given the size of the players hand and returns the card at that index in the hand.

**AIPlayer.aiRequestTarget()** - randomly selects a player to target. Randomly generates an integer between 0 and 3, the range of indexes of the players in the game and checks that it did not generate its own index. If it has it retries until it gets an index that is not its own. Returns the player at that index.

**HumanPlayer.haveTurn(**) - runs the human players turn much like the AI’s but with prompts and text instead of randomization. Uses a do{...} while() loop with a locally defined boolean *playing*. Checks for books, returns if the player has enough cards for a book, checks their hand with size() to make sure it’s not empty. If it is not it prints out their hand using displayHand(). Prompts the user for a target player and finds that player through a combination of contains() and iterating through the *Players* array, using getName() to check for a name matching what the user gave as input. If the player is in the game then it continues, if not it repeats. Prompts the user for a card to ask the donor for and makes sure that card exists by checking the entered value against the enumerated values for the cards using Card.valueOf(). Catches an IllegalArgumentException and loops while the entered value is invalid. Once a valid value is entered it checks that the player has that card in their hand using hand.contains(). If they have it it continues otherwise it loops. Once a valid target and card are established it asks the donor for that card using askFor() and sets the *playing* boolean as the return value. Prints out if the recipient received any cards from the donor and if the turn loops and the player can ask again a player for a card. Once the player fails to receive any cards from a donor then they go fish() and what they drew is printed out. Their turn then ends.

**HumanPlayer.displayHand()** - displays the hand of the player. Sequentially prints out the values of the cards in the player’s hand.

**HumanPlayer.contains()** - checks the *Players* array for the given player. A for loop is used to iterate through the array and find the given player. Returns true if the player is found and false otherwise.

**NamePool.makePool()** - adds the names from the Names.txt file to an ArrayList. Iterates through the file using the Scanner object, reads the name, and puts it into an array *pool* to be used later. At the end, the size of the ArrayList is returned..

**NamePool.distributeNames()** - distributes names from the *pool* array to the AI players in the game. Checks the human player’s name and if it is in the pool it is removed (so that an AI cannot have the same name). Then uses setName() to assign names randomly to the AI players.

**NamePool.retrieveName()** - retrieves a name from the *pool*. Uses a random *r* object to use nextInt() on the *pool.size()*. As long as there are enough names in the *pool* it will retrieve a name. Returns a string.

**NamePool.setName()** - sets the name of the given AIPlayer. Uses the retrieveName() method to set *aiName*. As long as *aiName* is not null, it sets the given AI’s name as the *aiName* using the AIPlayer’s setName().

**GoFish.draw()** - randomly selects a card from the deck. Returns the result of using a *pickRandom* Random object to find the nextInt() on the *deckOfCards*.size() and then removing that card from the *deckOfCards*.

**GoFish.deckSize()** - returns the size of the deck using *deckOfCards*.size().

**GoFish.restart()** - restarts the game. Uses a scanner object *input* to collect a string from the user. Uses a switch to compare the input to different values. “Y” restarts the game, “N” ends the game, and anything else loops the method. If the result is “Y” a boolean value *result* is set to true and at the end of the method *result* is returned.

**GoFish.main()** - runs the game. Creates a *deckOfCards* array object. Iterates through the enumerated values of the Card class four times, once to simulate each suit, and adds a card of that value to the *deckOfCards* array. Creates a HumanPlayer object *human* and three AIPlayer objects, *ai1, ai2,* and *ai3.* Stores the player objects in a *Player's* array. Prompts the user for a name and then asks you to confirm what you entered as your name. If you confirm it sets the given name as the *human* player’s name and starts the body of the game, otherwise loops until you enter a name. Starts a while loop checking that the result of *getNumBooks* on each player object is less than the total number of books that can be created (because if all the books that can be created are created then the game cannot continue). While this loop runs, it executes the haveTurn() method for each player. Then creates four integer objects to hold the scores for each player, using the *getNumBooks()*. Compares all these scores and determines the winner. Prints out if the *human* player was the winner, the loser, or tied and then shows the scores of all players. Finally, calls restart() and if *true* is returned the main() method loops, otherwise it ends.