Production Code

Card:

The Card class creates and validates card objects that will be used in the game. Although we initially chose to have the cards as integers, we decided to change the cards to objects as this allows more individuality between cards with the same values. This means that during the discard process our 'weighting' probabilities, which determine how likely a card is to be discarded, can be different for card with the same integer card value. This class will also handle the setting and incrementing of weightings used in later classes.

Deck:

This class creates and stores the card objects in a queue that will make up the decks of the game. An initial deck is made which will be checked for validity, e.g. 4 cards not 3, and then the cards will be put into a queue. We chose a queue over an array to store the cards as this would allow cards to be drawn from the top and placed at the bottom as per the specification. There are also methods to retrieve the contents of the deck, log contents, add card to the bottom, remove card from the top, and retrieve the final deck id which are all used in other classes dealing with their own deck instances.

PackHandler:

PackHandler is the class that uses the user inputs from the terminal to create the initial pack of cards that will then be used to make the decks and player hands. A user will input the number of players and the file location for the input pack of cards. If either value is invalid, e.g. file not found or negative number of players, or if the values do not match, e.g. 4 players should mean 32 cards, but the input pack only has 20, then the user is re-prompted for valid inputs that can successfully make the game. The input pack is iterated through, and each line is validated as a card value. The input pack is then split into two halves, player cards first and deck cards secondly. This is due to deck being initialized before players however we need to keep to the specification of filling player hands first. The deck and player halves are split into individual hands and decks in round robin fashion based on the number of players there are. Finally, a 'gameArray' is returned of the hands and decks of individual players and deck in a 3D array for the game loop to use.

Player:

This multi-threaded class creates the players using the constructor and gives the players the ability to pick up and discard cards along with a variety of methods all used during the game round. The constructor takes the initial hand that is validated and turned into the player's hand, Boolean winCheckArray which is false but returns true if that player has won, the ID of the winning player to tell the other players they have won and their respective pick up and discard decks. A hand however will have 5 elements rather than 4. During pick up the card is stored into the 5^{th} slot from pick up deck and its weighting is set, 0 for preferred and 1 for non-preferred cards. During discard, a non-preferred card is picked using the weighting probabilities and discarded to discard deck. This is determined by random where cards with higher weightings have a higher chance of being discarded, preferred cards have a weighting and probability of 0. The other cards will have their weightings increased by x2 to prevent stagnation compared to more recent cards with preferred cards being unchanged as 0x2=0. A win check is done after the initial hand and each round where a matching hand of the same value for all fours cards,

preferred or non by that player, is a winning hand returning true in the win check array of all players with true for the index of the winner.

CardGame:

This is the main class that all other run through which includes the game loop. There are many attributes, instances and arrays initialised at the beginning such as the 'gameArray' with the hands and decks. First decks are created using the array of decks and their contents. Player is created with the constructor and player hands then thread array is made based on the number of players. One thread per player allows for concurrency as the players do their actions in parallel. Threads are started and the game loop beings. It will constantly run while the 'winningPlayerId' == 0 which is the non-existant player 0. There are continual checks each loop if a winning player is found, interrupting threads at the end of each round when we detect every player is finished. If a player isn't finished, the finished player threads will sleep. Once all threads are finished and there is no winner, we clear the win check array and continue. Once a player is found, we find the index of the winner and return its id. The threads are stopped safely and the hands and final decks are logged. There are two 'Utility' static nested classes isWinCheckArrayFull and findWinningIndex. The former checks that there aren't any nulls within the win check array while the latter finds the index of the winning player once there is one. We chose sleeping the threads over waiting as we believed synchronization would lead to the code running more linearly than unsynchronized threads. However, all threads stop at a point at the end of the round as all threads wait for all player threads to finish. The programs speed will most likely be hindered. We felt this was worth it in return for the heightened robustness of the program.

OutputManager:

Class based around writing to a file allowing us to monitor the game as it runs. The constructor creates a new output file named after the deck of player it is respective to. Message throughout the game can be sent to output manager which can be logged and sent to the output file e.g. Player 1 draws a 2 from deck 3. Once connection is established to a file, the connection locks so other files cannot read or write to that specific file as two threads connected to one file at the same time could be problematic. There are also other methods that maintain the safe and secure opening, closing and writing to files along with logged exceptions thrown if errors occur e.g. writing to a closed logger. An issue occurred when we realised the output manager was not thread safe when working with the player and card game classes. Therefore, we decided to write another class that uses OutputManager that logs the threads events more safely.

ThreadedLogger:

This custom designed threaded class is used to log messages throughout the game. At any point in the code, you can open a thread for logging. The program will restrict the logging to one thread at a time, this is safer than opening multiple that may interfere with each other, repeatedly checking the queue 'message stream' for any potential log message which is repeated until a command is run for the specific thread which shuts it down. Once this command is run, it switches a flag which will tell the program to finish logging whatever is in the queue and then shut the program down. As it doesn't stop any code from running when opening or closing, this is much safer than only using our previous OutputManager class.

Testing Code

JDK 21.0.2

Maven 3.9.9

JUnit Version: 5.9.3

Card:

Valid/Invalid Values – A range of values are inputted into the two tests to discern whether an individual value can be a valid card. Non-zero positives are valid while negatives, zero and nulls are not and throw InvalidCardException.

Set Preferred/Non-Preferred Weighting – SetWeighting() sets the weighting used in determining cards to discard in Player.java. Cards are preferred by players if a card value matches the player number. These tests check that the weighting is correctly set. Preferred cards have a weighting of 0 while non-preferred cards will be set to 1.

Weighting Multiplier – To prevent stagnation of cards, the weighting probability of that card being discarded is increased every round by multiplying its weighting. Increment tests expect higher values after incrementWeighting() is used, unless that card is preferred in which case the multiplication will have no effect (0x2=0).

Deck:

Invalid Deck Length – Decks must have a length of 4 elements/cards or DeckLengthException is logged.

Invalid/Null Cards – Deck must contain valid values of cards or log InvalidCardException or NullPointerException.

Deck ID – Each new deck is tested to be the previous deck's id + 1.

Log File – Tests that each new deck has an existing corresponding output file for logging contents at the end of the game.

Closure Log – log is successfully closed at the end of the game to prevent data leaks.

Deck Getter – decks must be available for other classes to grab the contents of for running a successful game.

Deck Logger – with the complexity of the logger, a test is made to see that the internal private logging operates as intended.

Deck Getter Size – checks that the deck size is as expected as we don't want decks to have an abnormal number of cards.

Add/Remove – with the frequent number of changes made to decks, they are checked that cards are safely added/removed without unsafe changes.

Double Add/Remove/Closure – a deck cannot undergo the same operation twice in a way that would change the needed constraints of the deck. For example, remove two cards from a deck without adding another would reduce the deck size too much for the rest of the game loop to run.

Close Add/Remove – decks must be closed to prevent data leaks so tests are run to see that closed decks cannot be changed further after closure.

PackHandler:

Valid/Invalid Number of Players – The number of players must be a non-zero positive as you cannot have -1 players. These tests make sure that the inputted number is valid for the creation of the game, otherwise throw an IllegalArgumentException and re-prompt the user for a valid input. An extra check is made that the number of players isn't too high to make the pack size above the integer limit.

Valid/Invalid FilePath – Similar to Number of Players Tests, FilePaths must be valid file location to successfully create the game. If the location is an invalid location/not found then either FileNotFound or InvalidFile exceptions are thrown and the user is re-prompted for an input.

Line Value for Card – The numbers within the input pack must be valid for the card values. These tests iterate through each line of the input pack and throw an exception if an invalid value is found e.g. a non-integer.

Player:

Initial Hand Length – The player constructor needs a valid initial hand (received from packHandler) to begin the game. These tests check that an exception is thrown if the initial hand is invalid e.g. contains nulls. Although this is unlikely with the testing of previous code, these tests are implemented to catch any errors that may pass through the other test catches.

Winning Hand Check – winCheck() returns true if a player has a winning hand. Tests are ran to check that winCheck(0 returns the expected true/false statements for hands. This is needed as winCheck(0 is ran continually and is necessary for declaring a winner and ending the game.

IsOneNull() – pickUp() cannot happen if there is not at least one null in the player's hand to temporarily hold a card. IsOneNull() is tested to return the index of the null card to allow for successful pickup and that an exception is throw if there had been an error and all slots are already filled.

PickUp/Discard – These two significant methods are checked that they run successfully or that an exception is throw if an error occurs. E.g. too many cards to pick up another throws an exception in pickUp().

HandToString – Hands will need to be frequently logged to output files. It is necessary that the conversion of the hand array to readable print is successful therefor a test is ran of handToString() that output is as expected.

Get Non-Null Card Values – Although we have 5 elements in a hand with one being a null to allow for pickups, the null shouldn't be printed in the player hand output files. This test checks

that getNonNullCardValues() outputs the hand of a player without its null, otherwise throwing an exception.

Discard Index – A card is discarded based on its weighting. A higher weighting means a higher probability of being discard. Simply having the highest weighting discarded would lead to stagnation and infinite game loops thus random probability prevents both issues. These tests make sure that preferred cards are not choses but can also be run multiple times to check that the same card isn't discarded every time.

CardGame:

WinCheck – Several tests are ran based on the combinations of true/false statements that make up the Boolean array 'winCheckArray'. The variety of tests check that the returns are as expected based on the inputted winCheckArray in isWinCheckArrayFull() which returns false if there are any nulls in the array e.g. true for winCheckArray = {true, false, true, false} as there are no nulls.

findWinningIndex – More tests for findWinningIndex() which returns the index of the winning player once a winning hand is found. Similar to winCheck, several tests are ran based on the potential winCheckArrays e.g. index of -1 if there are no winning players yet or of 3 if the fourth player has won.

Development Log

Date and Time	Duration	Colum Role	Kadeem Role	Colum Signature	Kadeem Signature
16:00 9/11/2024	1 Hour 30 Minutes	Pilot	Observer	730024327	730036432
13:00 25/11/2024	2 Hours	Pilot	Observer	730024327	730036432
15:00 30/11/2024	2 hours	Pilot	Observer	730024327	730036432
15:00 1/12/2024	3 hours	Observer	Pilot	730024327	730036432
15:00 4/12/2024	2 hours	Observer	Pilot	730024327	730036432
13:00 7/12/2024	5 hours	Observer	Pilot	730024327	730036432
13:00 8/12/2024	5 hours	Observer	Pilot	730024327	730036432
13:00 9/12/2024	6 hours	Observer	Pilot	730024327	730036432