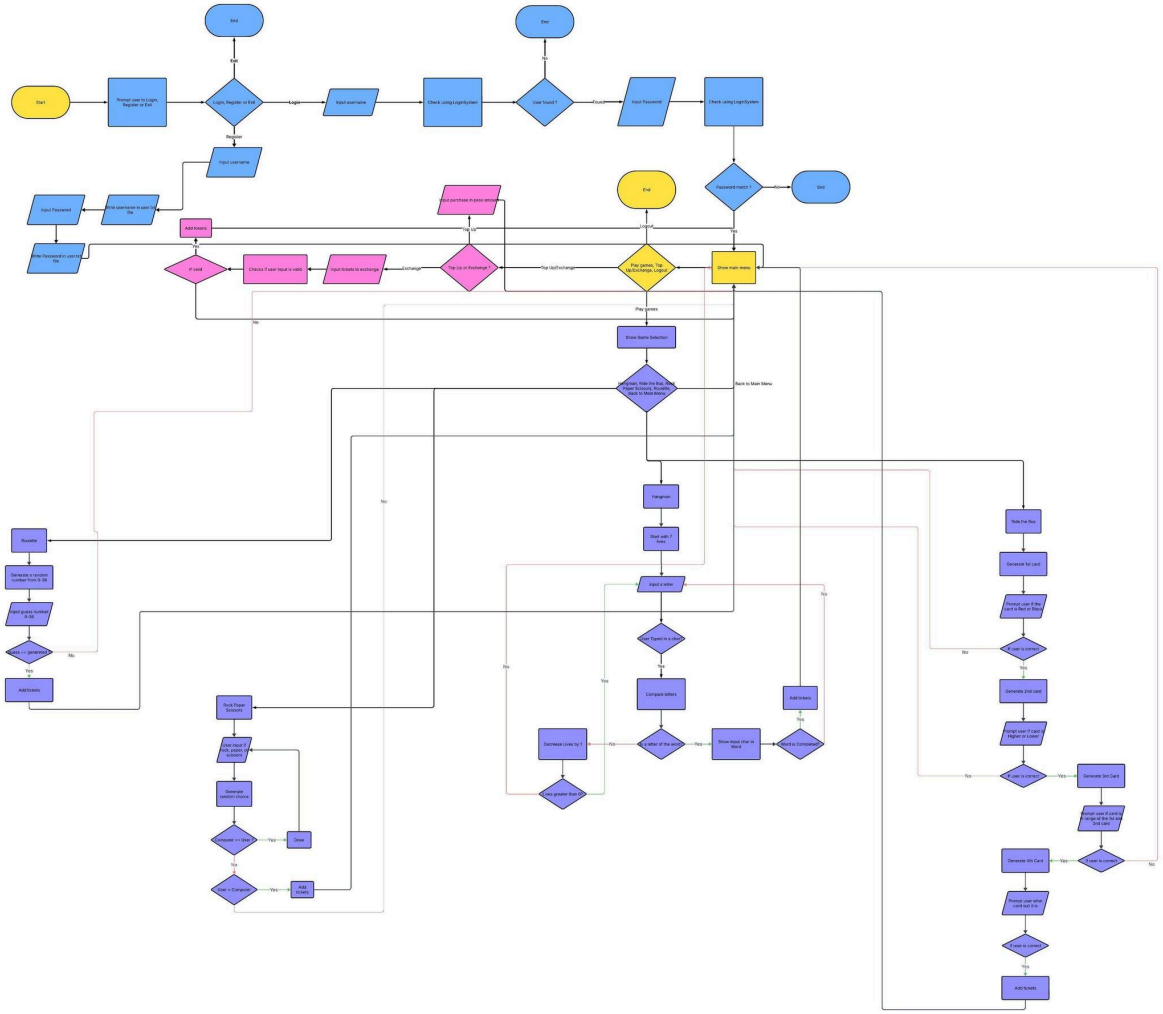
Documentation

Generalized Explanation

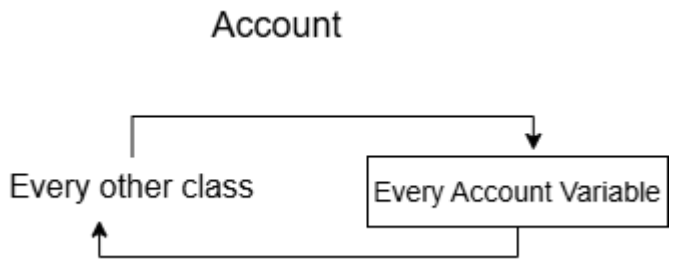
The Program is a simulation of an Arcade, including:

- Accounts, as well as its balance and transaction (Payments and Play History) records.
- Playable Games
- Retro Arcade UI

Overall Flowchart



Account



***Definition and Purpose*:**

- An abstract class that manages “Account” objects, essentially the base class in which “unique users” are created, defined, and managed.

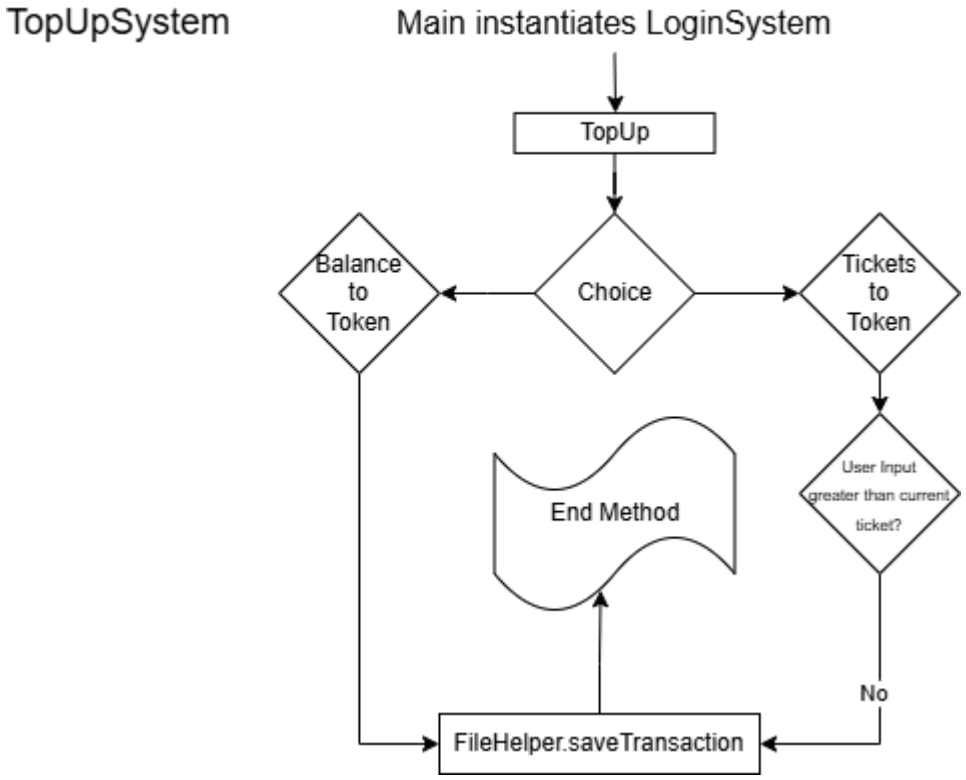
***Classes and Imports*:**

- N/A

Methods:

- Balance
  - **addBalance()** an overloaded method that takes in either an integer or double value and increases the balance of the current account object. Only used by the Account’s subclass, TopUpSystem.
- Tokens
  - **addTokens()** takes in an integer value and increases the number of tokens in the current account object by the specified amount. Used within TopUpSystem subclass to reflect token gains after a balance top-up or ticket exchange.
  - **spendTokens()** takes in an integer value and decreases the number of tokens in the current account object. Returns true if enough tokens are available or false otherwise. Used in the Game class to check and deduct tokens before allowing the player to play.
- Tickets
  - **addTickets()** takes in an integer value and increases the number of tickets in the current account object. Used in the Game class to reward players with tickets after a game, and in the TopUpSystem class.
  - **exchangeTickets()** takes in an integer and checks if the current account object has enough tickets to exchange. Returns true if the number of tickets is sufficient, allowing the exchange to proceed, returns false otherwise. Used only in TopUpSystem class.

TopUpSystem



Classes and Imports:

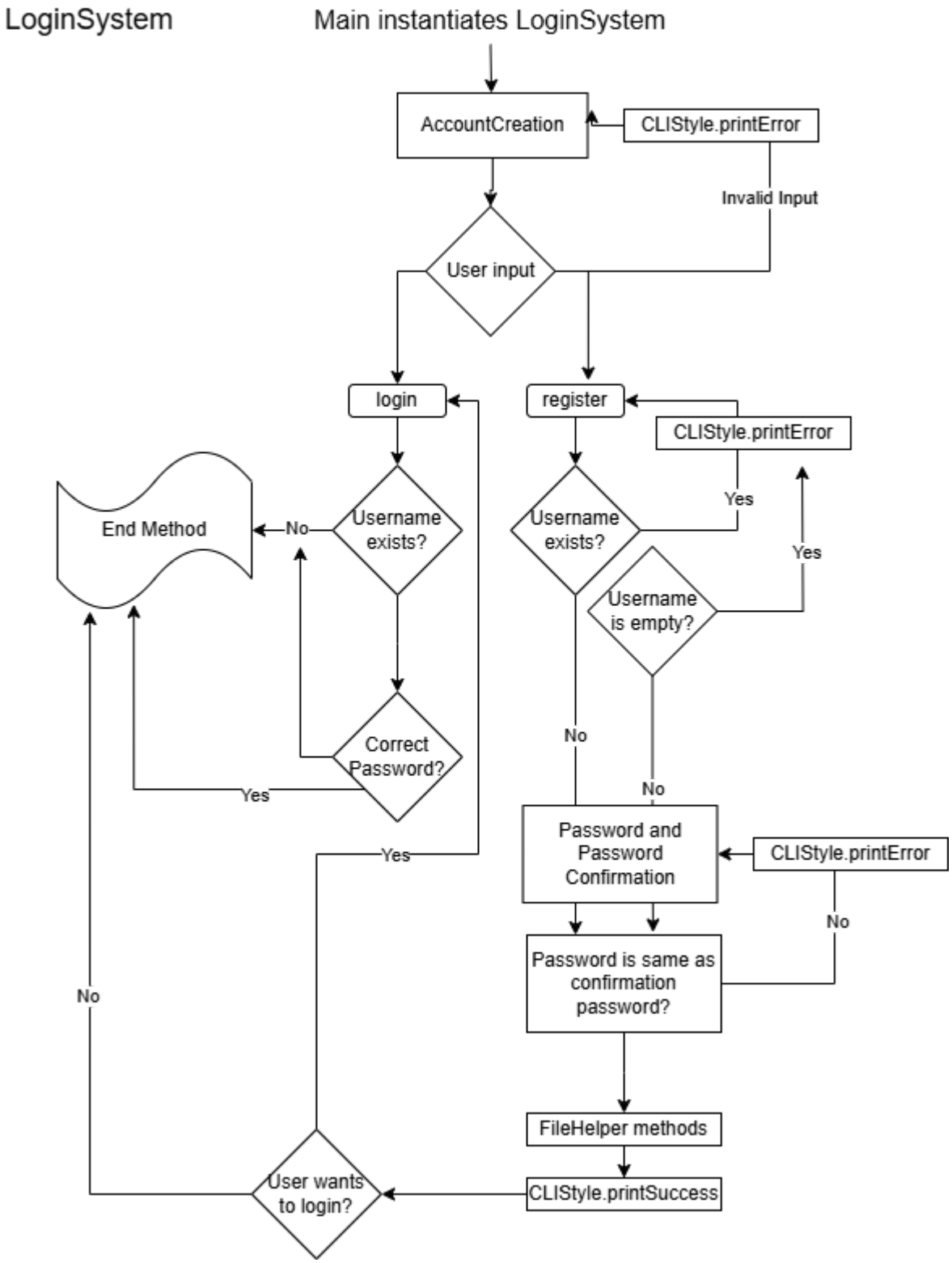
- Java.text package
  - **SimpleDateFormatter** is an instance class that allows the usage of user-defined patterns for date-time formatting.
  - **Format** is an abstract class that allows the formatting of locale-sensitive information (dates, numbers, etc.). In the context of the TopUpSystem class, an instance method “format” is used to take a date object as a parameter, returning a formatted String.
- Java.util package
  - **Date** is an instance class that takes in the current date and time of the machine. The default constructor of Date retrieves the system’s current timestamp which can be converted into String.
  - **Scanner** is an instance class that takes in either a file or an input stream. System.in is used as the argument in this class to read keyboard inputs from the user

Methods:

```
public void TopUp()
```

- **This `TopUp()` method is part of a class that extends `Account` (likely `TopUpSystem`). It allows a user to either top up their balance using money or exchange tickets for tokens.**

LoginSystem



Classes and Imports:

- Java.util package
    - **Scanner** is an instance class that takes in either a file or an input stream. System.in is used as the argument in this class to read keyboard inputs from the user. In this class specifically, it uses the Scanner object from the main

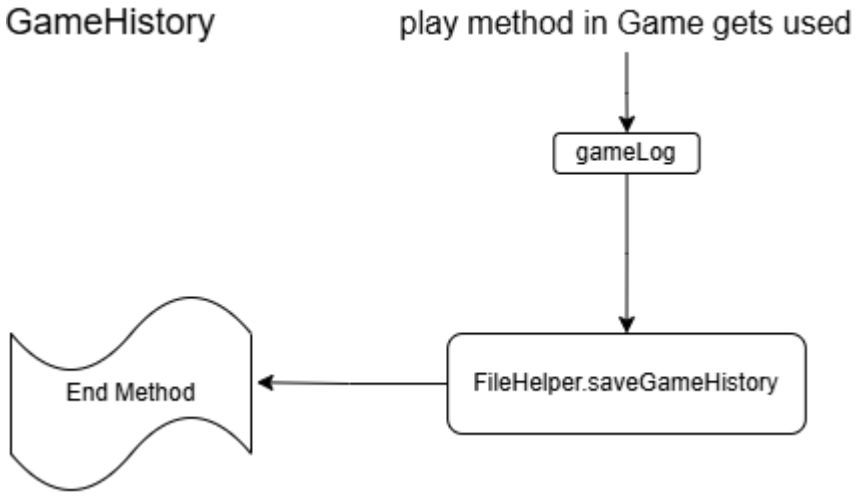
class to avoid conflict with closing the Scanner (issues such as LoginSystem already closing the scanner while the try-with-resources in the main class still requires the Scanner object).

- FileHelper is a utility class created in this program that is often referenced in the LoginSystem class. It is a class that handles all the file handling methods, which LoginSystem, needing to check pre-existing Accounts or creating Account objects, needs to access.

Methods:

- **AccountCreation()** displays the main menu for account setup, prompting the user to either log in or register. Handles invalid inputs and returns the resulting logged-in Account object or null if the operation fails or is cancelled.
- **login()** prompts the user for a username and password. Verifies the credentials by checking if the username exists and the password is correct. If valid, it returns the corresponding Account object; otherwise, it prints an error and returns null.
- **register()** guides the user through the registration process, including username validation (checks for duplicates and empty inputs) and password confirmation. Once a valid account is created, it is saved using FileHelper. Optionally, the user is asked if they want to log in immediately after registering.

GameHistory



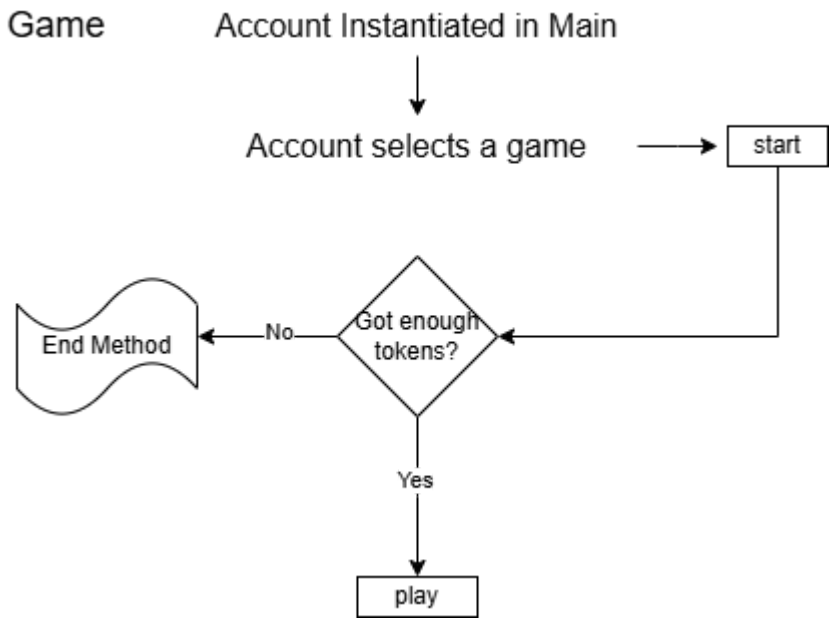
Classes and Imports:

- Java.text package
  - **SimpleDateFormatter** is an instance class that allows the usage of user-defined patterns for date-time formatting.
  - **Format** is an abstract class that allows the formatting of locale-sensitive information. In the context of the GameHistory class, an instance method “format” is used to take a date object as a parameter, returning a formatted String.
- Java.util package
  - **Date** is an instance class that takes in the current date and time of the machine. The default constructor of Date retrieves the system’s current timestamp which can be converted into String.

Methods:

- **gameLog(String gameName, boolean victoryStatus, int tickets)** Records a formatted log of a game session, noting the date, game name, result (win or loss), and ticket earnings. Saves the log to the correct account file using FileHelper, with the playerAccount passed as a parameter.

Game



Definition and Purpose:

- The superclass of all the playable games in the arcade. It includes the stats (such as game cost, rewards and victory status) of the player when they play a specific game.

Classes and Imports:

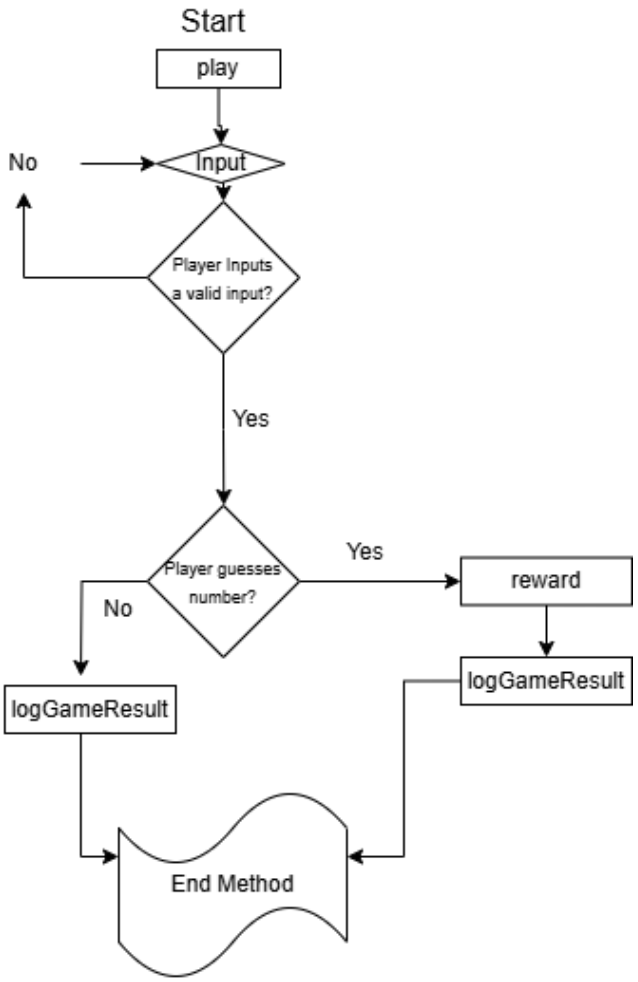
- N/A

Methods:

- **start()** A public void method that checks the user's (or Account object in more technical terms) if they have the right amount of tokens to start a game, if yes, then proceed to the play() method.
- **play()** an abstract void method in which the game functionality and flow are to be defined.
- **Game(Account)** the default constructor that requires an Account to be passed in order to update variables and log necessary information
- **reward(int)** a protected void method that displays the winning screen and updates the ticket amount of the account
- **logGameResult(Account, String, boolean, int)** a protected void method that calls the game logging in method in the GameHistory class.

Roulette

Roulette



Classes and Imports:

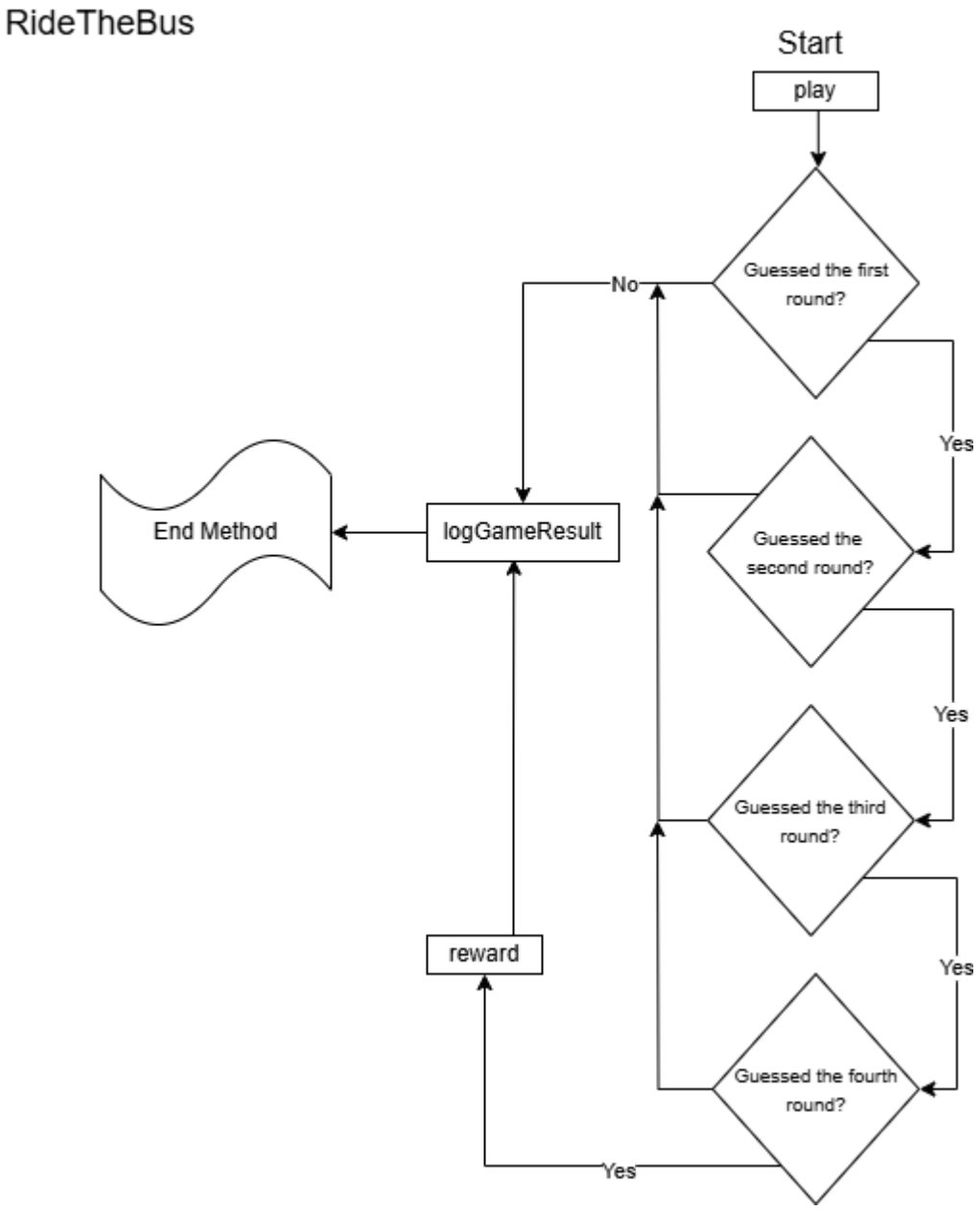
- **Random** is a class that allows the creation of an integer in a predefined range or the ability to randomly select an index in an array. In the context of this Game subclass, Random is used to randomly select between the numbers 0-36, to which then the player must guess.

Methods:

- **play()** overridden method from the Game superclass that contains the logic for playing Roulette.
    - Prompts the player to bet on a number between 0 and 36, generates a random winning number, and checks if the player's guess matches. If the player wins, they are awarded 30 tickets using the inherited reward() method; otherwise, a loss message is shown. Game results are logged through logGameResult(). Automatically called by start() after checking for available tokens.
- **reward(int ticketsWon)** an inherited protected method from Game that adds the specified number of tickets to the player's account using Account.addTickets(). Also displays a message informing the player of the reward. Called only when the player wins the game.

- **logGameResult(Account player, String gameName, boolean won, int tickets)** an inherited protected method from Game that logs the game result using GameHistory.gameLog(). Records whether the player won, the game name, and the number of tickets earned at the end whether the player wins or loses.

RideTheBus



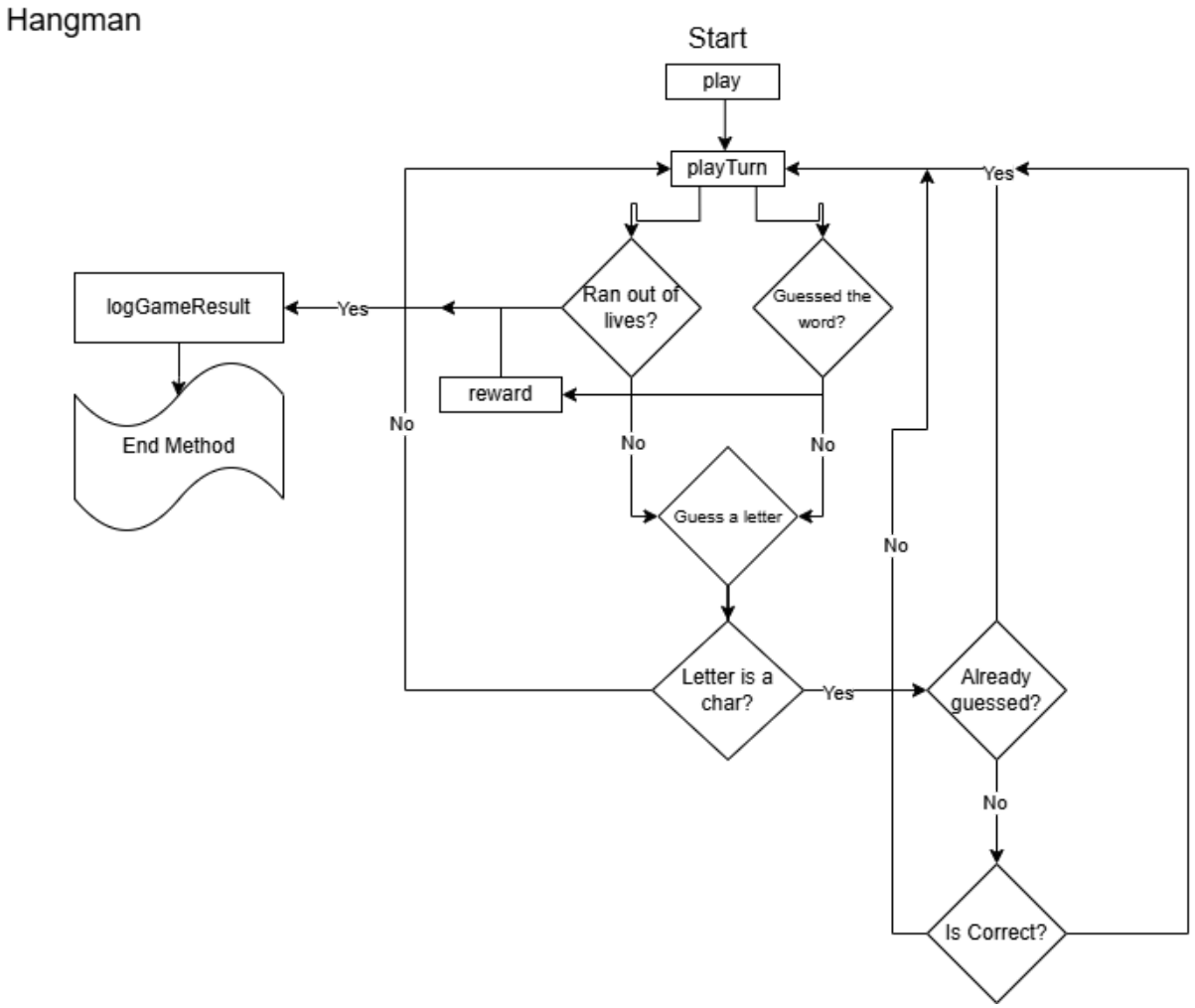
Classes and Imports:

- java.lang
    - **Math** is a class that allows the usage of static methods in manipulating or accessing numerical data.
        - **min** is a method of the Math class that returns the smallest value between two parameters. On the other hand, **max** returns the highest value between two parameters. In this program, both methods are used to find the proper range between two values consistently by accessing the smallest and biggest between two cards regardless of order.
- java.util
    - **Random** is a class that allows the creation of an integer in a predefined range or the ability to randomly select an index in an array. In the context of this Game subclass, Random is used for both, cycling through an Array to assign Card Suits and randomizing a set range of integers to assign Card Values.

Methods:

- **play()** overridden method from the Game superclass that contains the logic for playing Ride the Bus.
    - The player is asked to make guesses about four randomly drawn cards in 4 rounds. If the player guesses correctly in each round, they are rewarded with 20 tickets using the inherited reward() method and the result is logged with logGameResult().
- **losePause()** a private helper method that provides a pause after the player loses, requiring the player to press Enter before returning to the main menu. Called after any incorrect guess during the game.
- **reward(int ticketsWon)** an inherited protected method from Game that adds the specified number of tickets to the player's account using Account.addTickets(). Displays a message informing the player of the reward & called only when the player wins.
- **logGameResult(Account player, String gameName, boolean won, int tickets)** an inherited protected method from Game that logs the game result using GameHistory.gameLog(). Records whether the player won, the game name, and the number of tickets earned at the end whether the player wins or loses.
- losePause() turns the “pause before quitting the play() method” into a method itself since there are multiple cases in which the user can lose in this game. This pause trick is done across all the game subclasses after the program ends (through loss or victory) but they are all done once, RideTheBus has multiple scenarios of ending its play method.

Hangman



Classes and Imports:

- java.lang
  - **StringBuilder**, to put it simply, makes Strings mutable. Stringbuilder is a character array to begin with, and when the “String” gets modified, it just adjusts the array accordingly and parses it to String. Strings are not being mutated here, it's just constantly reassigned.
- java.util
  - **Random** is a class that allows the creation of an integer in a predefined range or the ability to randomly select an index in an array. In the context of this Game subclass, Random is used to randomly select words to be guessed by the user.
  - **Scanner** in this class is used to take in a line, however, the required input should only be a single character, that is done through constantly reminding the player and printing an error in the case of non-char inputs.

Methods:

public void play()

- **An overridden method from the Game superclass that contains the logic for playing Hangman.**
    - **The player is asked to guess letters in a word, and the game continues until they either guess the word or run out of lives. The player starts with 7 lives, and for each incorrect guess, they lose one life.**

private void playTurn(String word, StringBuilder wordDisplay, String guessedLetters, int lives)

- **A private helper method that handles one turn of the game. It recursively prompts the player for a letter guess and updates the game state (the word display and remaining lives). If the player guesses correctly, the display is updated, if they guess incorrectly, their lives are decremented.**

reward(int ticketsWon)

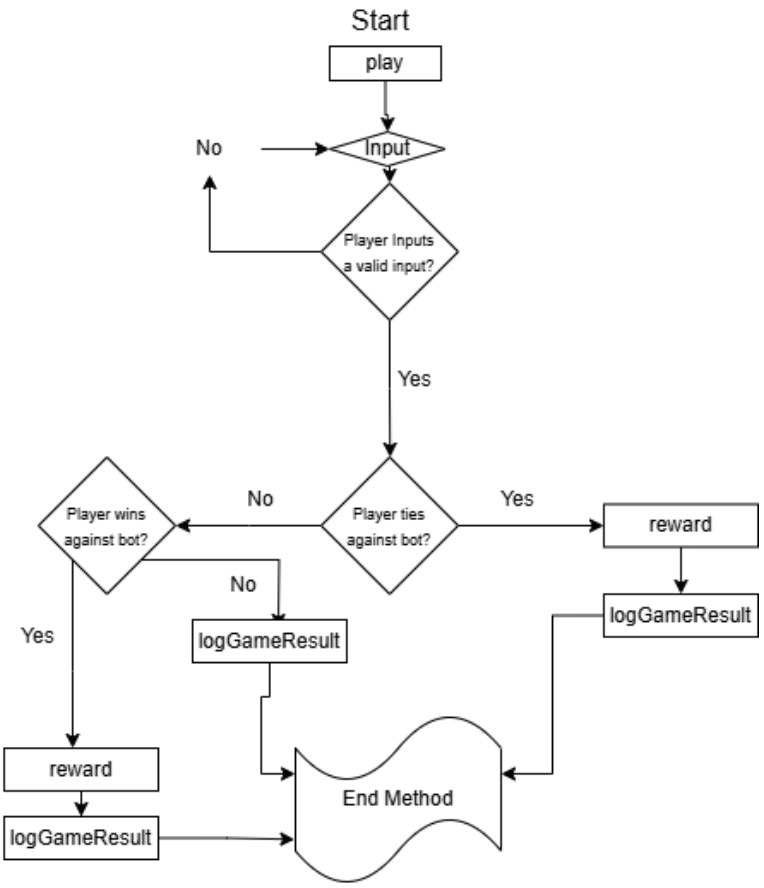
- **an inherited protected method from Game that adds the specified number of tickets to the player's account using Account.addTickets(). Displays a message informing the player of the reward & called only when the player wins.**

logGameResult(Account player, String gameName, boolean won, int tickets)

- an inherited protected method from Game that logs the game result using GameHistory.gameLog(). Records whether the player won, the game name, and the number of tickets earned at the end whether the player wins or loses.

RockPaperScissors

RockPaperScissors



Classes and Imports:

- **Random** is a class that allows the creation of an integer in a predefined range or the ability to randomly select an index in an array. In the context of this Game subclass, Random is used to randomly select the opponent's hand.

Methods:

public void play()

- **play() overridden method from the Game superclass that contains the logic for playing Rock Paper Scissors.**
  - **The player is asked to enter one of three valid moves: Rock(r), Paper(p), or Scissors(s), validates the input, and then randomly selects a move for the computer. If the player wins, they earn 10 tickets through reward(). The game result is logged using logGameResult(), and the game pauses for the player to press Enter before returning to the main menu.**

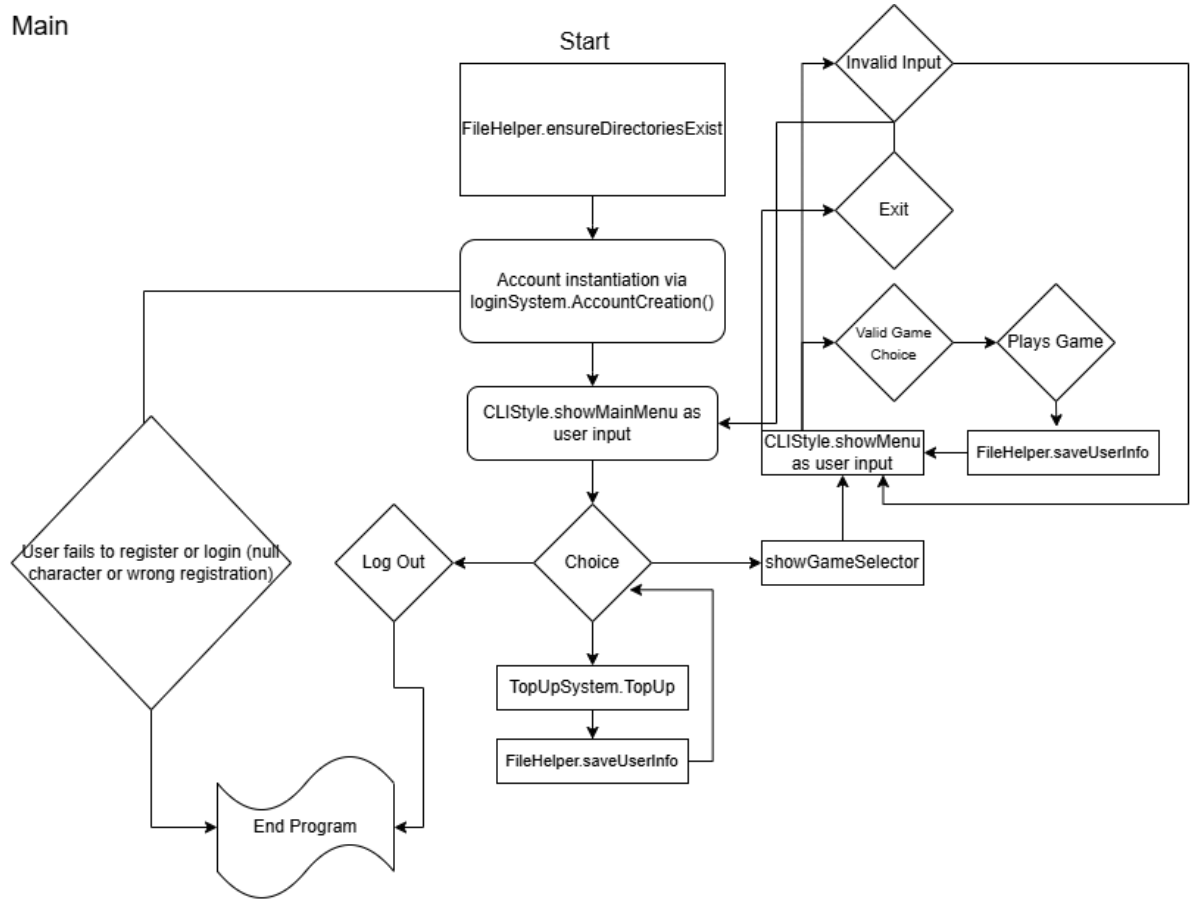
reward(int ticketsWon)

- **an inherited protected method from Game that adds the specified number of tickets to the player's account using Account.addTickets(). Displays a message informing the player of the reward & called only when the player wins.**

logGameResult(Account player, String gameName, boolean won, int tickets)

- **an inherited protected method from Game that logs the game result using GameHistory.gameLog(). Records whether the player won, the game name, and the number of tickets earned at the end whether the player wins or loses.**

Main



Definition and Purpose:

- The entry point of execution in the Java program. It starts the program and manages interactions between objects (or classes for that matter).

Classes and Imports:

- Java.util package
  - **Scanner** is used in the main class to allow the user to select in the main menu and games
- Exception
  - While not exactly a package, **Exception** along with its error subclasses **IOException** (missing file or directory) and **ClassCastException** (when an

object is cast with a wrong datatype) are also used in the main class for extra error handling (Most error handling are done in the classes themselves) .

- Arcade package (supposedly root directory of the program, but removed as a concept, calling the program's classes as belonging to Arcade is just used for clarity)
    - **Account** class is used for account creation and modification.
    - **Game** abstract class is used to allow the users to select games.
    - **FileHelper** utility class is used to check directory issues before the program begins in the main class specifically.
    - **CLIStyle** utility class handles all the terminal styling (front-end). Most of its methods are used in the main class including Menu Displays, Choice Displays, and Terminal Clearing.

Methods:

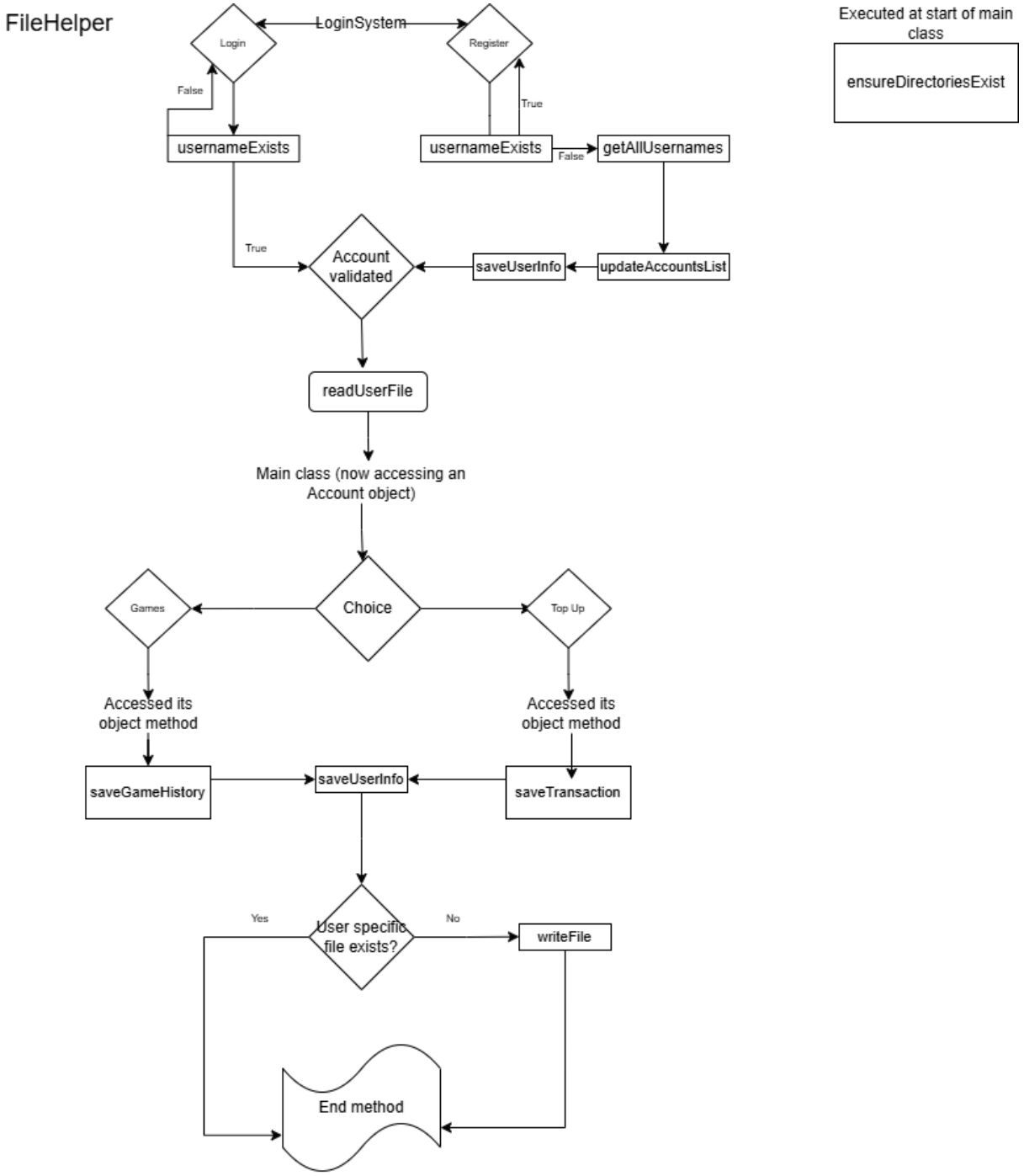
`private static void showGameSelector(Account account, Scanner scanner)`

- **The showGameSelector method displays a menu of available games for the user to choose from, then launches the selected game and saves the updated user information afterward.**

`public static void main(String[] args)`

- **This main method is the entry point of the arcade application. It handles initial setup, user login/registration, and provides a main menu loop for playing games or topping up.**

FileHelper



Definition and Purpose:

- A utility class often referenced in the Account subclasses. It is a class that handles all the file handling methods including checking for file integrity, file information, file content, directory integrity, modifying and creating text files.

Classes and Imports:

- Java.io package

    - **File** is a class that creates, manipulates, and verifies file system objects such as files and directories. It represents the directory path or file if used as an object.
        - **File.separator** is a platform independent way to get a correct file path separator (/ or \), it is a final string, in the case of FileHelper, it is used several times to create or verify a directory.
        - **mkdirs()** or **mkdir()** is a boolean method, returning true if it creates a specified file path/directory and returning false otherwise. mkdir creates a single directory while mkdirs creates multiple.
    - **FileWriter** is a subclass of OutputStreamWriter created for writing an output stream onto a specified file, aside from that, its only job is connecting to a file, it only has simple writing capabilities. It is used as a parameter for PrintWriter since it is a Writer.
    - **PrintWriter** enhances file writing by adding formatting capabilities such as println() and printf(). It does not directly manage file connections, instead relying on a Writer to handle file access. By wrapping FileWriter, PrintWriter allows formatted text to be written to a file efficiently.
    - **IOException** (missing files, but more so on general input output error) and **FileNotFoundException** (subclass of IOException when a file is missing or inaccessible) are exception errors, in this context, they are used for handling errors that correspond to them.
- Java.util package
    - **Scanner** is an instance class that takes in input streams. In this context it is used for scanning text files.
    - **List** is an interface (a specifier as to what methods a class must implement (a class blueprint)), it defines the behavior of list-like structures, one of it being:
    - **ArrayList** is a class implementing List, it allows the creation of a dynamically sized array.
- Java.lang (default package)
    - **System** is a final class that provides utility methods for interacting with the runtime environment (software and system components a program needs to execute). It cannot be instantiated because all its members are static. It handles standard i/o, system properties, and memory management.
        - **getProperty** is a method that takes in the value of a system property. In the case of this program, “user.home” is used as an argument, returning the value of the user’s home directory (C:/Users/this)
        - **err** is used in a similar manner to System.out.print. It also prints but mostly for error messages.
    - **Exception** Classes are less broad functionalities and more on a signal for an error then handling them.
        - **SecurityException** is an error for when a file or directory has restricted access
        - **RuntimeException** is a broad error that represents an error upon execution. Mostly for mistakes in logic or invalid inputs.
            - **NullPointerException** occurs when you try to use a variable or an object that has the value of “null” as if it did have value.

- **NumberFormatException** occurs when a String is attempted to be converted into a numerical data type despite not being a numerical value in the first place.

Method:

public static void ensureDataDirectoriesExist()

- **The ensureDataDirectoriesExist method checks if two directories (DATA_DIR and USERS_DIR) exist. If they don't, it attempts to create them. It handles any errors, such as permission issues, and prints appropriate messages if the directories cannot be created.**

public static void updateAccountsList(List<String> usernames)

- **The updateAccountsList method updates the accounts.txt file with a list of usernames. It writes each username to the file, overwriting its contents. If an error occurs during the file writing process, it catches and prints the error message.**

public static void writeFile(String path, String content)

- **The writeFile method writes the specified content to a file at the given path. If an error occurs, it catches the exception and prints an error message.**

public static void saveUserInfo(Account account)

- **The saveUserInfo method saves the details of an Account into separate text files for that user. It ensures the necessary directories and files exist, then writes the account's information, transaction history, and game history.**

public static void saveGameHistory(Account account, String gameLog)

- **The saveGameHistory method appends a game log entry to the user's game history file. It ensures that the log is added to the end of the file, preserving any existing game history**

public static void saveTransaction(Account account, String transaction)

- **The saveTransaction method appends a transaction entry to the user's transactions file. It ensures that the transaction is added to the file without overwriting any existing data**.

public static boolean usernameExists(String username)

- **The usernameExists method checks if a given username already exists in the accounts.txt file.**

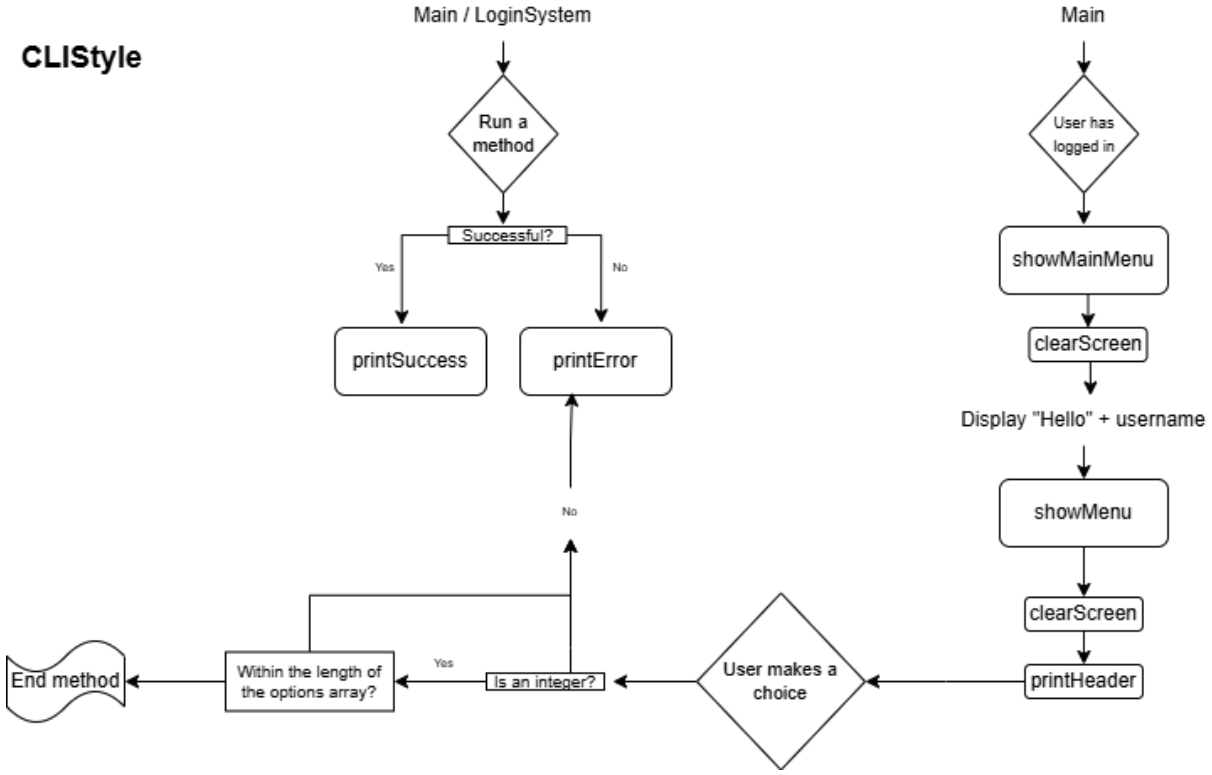
public static List<String> getAllUsernames()

- **The getAllUsernames method reads all the usernames from the accounts.txt file and returns them as a list of strings.**

public static Account readUserFile(String username)

- **The readUserFile method reads user data from a file named username_info.txt in the USERS_DIR directory and returns an Account (specifically a TopUpSystem object) with the extracted information.**

CLIStyle



Definition and Purpose:

- A utility class mostly used in the main class. It is essentially the “front-end” of the program, designing and formatting what is to be displayed on the terminal.

Classes and Imports:

- java.lang
  - **System.out.flush** is a method that forces all buffered data (data waiting to be written) to be written immediately. This method in this context is used alongside the printing of a sequence of escape characters (“**\033[H\033[2J**”) which emulates a “clearing” of the terminal by moving the cursor and refreshing the screen.
- Exceptions
  - **InterruptedException** occurs when a thread is interrupted by another thread before its specified time to sleep. A thread, or simply, lines of codes to be executed, can be paused using the method thread.sleep for a certain amount of time (in milliseconds).
  - **InputMismatchException** is an error thrown by a Scanner, it is an error that indicates the input doesn’t match the scanner’s type (e.g a string on a nextInt scanner method)

Method:

`private static void clearScreen()`

- **A method that effectively emulates a terminal clear by moving the cursor away and refreshing the terminal.**

`public static void printHeader(String title)`

- **The printHeader method displays a formatted title with borders, centered within a fixed width (50 characters), to make section headers stand out.**

`public static void printSuccess(String message)`

- **The printSuccess method displays a message in green text, typically used to indicate successful actions.**

`public static void printError(String message)`

- **The printError method prints an error message in red text to the console for better visibility.**

`public static int showMenu(String title, String[] options, Scanner scanner)`

- **The showMenu method displays a menu with a title and a list of options, then repeatedly prompts the user until they enter a valid number corresponding to one of the options. It uses a Scanner to read input and returns the user's selection as an integer.**

`public static int showMainMenu(Account account, Scanner scanner)`

- **The showMainMenu method displays a personalized main menu to the user using their Account info and returns their selected option as an integer.**

Expected Program Flow

1.) The program checks if the directory of the program's file outputs are accessible, can exist, or does not exist.
2.) The user is asked to either Log In a pre-existing account or to register a new account
    - This account is given a set 10 tokens upon creation
    - The user is always linked and accessed via its username and password
3.) Main Menu. The program always loops back here (with the above criteria being met) unless specifically stated by the user to Log out.
    a.) Choice 1: User selects the Top Up choice. Allowing them to convert tickets (game victory currency) into tokens or buy it directly with their money
        i.) Interacting with the top up system modifies the unique user transaction files while also increasing their account currency values (specifically balance and tokens).
    b.) Choice 2: User selects the Game choice. Allowing them to select games to gain tickets.
        i.) Interacting with the games modifies the unique user game history files while increasing their tickets.
    c.) Choice 3: User selects to Log Out, effectively ending the program

I didn't specify what classes are being used during that flow, I don't know if I should.
