University of Manitoba

Assignment 3

Introductory Computer Science 1

Pokémon Game

Deadline: **March 7**, 2025 11:59 PM

Material Covered: Weeks 1-5

- Objects
- Strings
- Exceptions
- File I/O
- Arraylists

Assignment Guidelines

- All students in this course must read and meet the expectations described in the Expectations for Individual Work in Computer Science. Follow the link and read all the information provided.
- These assignments are your chance to learn the material for the exams. Code your assignments independently. We use software to compare all submitted assignments to each other, and pursue academic dishonesty vigorously. **You must complete the *Honesty Declaration (under Assessments then Quizzes) before you will be able to submit your assignment.***
- Assignments must be completed using the course material. Do not use any advanced material we have not yet covered in the course, even if you have prior programming experience. Assignment 3 should be completed using concepts up to and including Week 5.
- You must adhere to the general assignment instructions **and** the programming standards document published on the course website on UM Learn.
- Submit **.java** files mentioned in the end of this assignmnet instructions.
- Do **NOT** zip the files that you submit.
- You may submit the assignment multiple times, but only the most recent version will be marked.

Pokémon Game

Objective:
The goal of this assignment is to develop a Pokémon Game simulation that reinforces key programming concepts such as ArrayLists, file reading and writing, string manipulation, objects, encapsulation, static variables, loops, and conditionals. You will implement various classes to model Pokémons, their types, stats, and moves while integrating battle mechanics based on turn-based logic. You will implement a Menu and a Game class to simulate a menu based game with statistics tracking. The game will load Moves and Pokémon from input files, then shows a menu to the user to either play a game or get statistics. The game itself simulates a battle between two Pokémon teams of max 3 Pokémon. They will keep battling based on how the players use them and in the end the team who still has Pokémons alive, wins. This assignment will challenge you to manage structured data, apply object-oriented programming principles, and implement a dynamic combat system using conditionals and loops. Make sure you fully read the assignment instructions, watch the video uploaded and listen to the podcast that has been provided.

Overview of What the Code Should Do:

1. **Load Pokémons and Moves from Files:** At the start of the game, Moves and Pokémons will be loaded from two separate text files. Each Pokémon will have a predefined set of moves.

    - The input files are ALWAYS named moves-data.txt and Pokémon-data.txt
    - There could be any number of moves. The number of Pokémon will be positive and an even number (2, 4, 6, etc.).
    - You may assume the the lines in the files are in correct format, meaning input with the correct number of parameters on every line of the file, always separated with a comma and a space, and correct data type.
    - You should still handle Exceptions such as FileNotFoundException when reading the file.

    Example input files: Pokemon-data.txt:

```
Name, Type1, Type2 (optional), Level, HP, Attack, Defense, Speed, Move1, Move2, Move3, Move4
Charmander, Fire, , 10, 39, 52, 43, 65, Scratch, Ember, Flamethrower, Tackle
Squirtle, Water, , 10, 44, 48, 65, 43, Tackle, Water Gun, Hydro Pump, Scratch
Bulbasaur, Grass, Poison, 10, 45, 49, 49, 45, Vine Whip, Tackle, Razor Leaf, Scratch
```

    moves-data.txt:

```
MoveName, MoveType, Category(Physical/Special), Power, Accuracy
Scratch, Normal, Physical, 40, 95
Ember, Fire, Special, 40, 85
Flamethrower, Fire, Special, 90, 75
```

```
Tackle, Normal, Physical, 40, 90
Water Gun, Water, Special, 40, 80
Hydro Pump, Water, Special, 110, 65
```

2. **Menu Flow:** When the game starts, the menu would be shown to the user to select from.

   Step 1: Display Menu

```
Welcome to the Pokémon Game!
1. Play Game
2. Get Pokémon Stats
3. Get Player Stats
4. Get Type Stats
5. Get Move Stats
6. Exit
```

   Step 2: User Select an Option

   - **1. Play Game**
     This option starts the game simulation, where players can battle each other using their Pokémon teams. The game flow involves getting player names, picking Pokémon teams, and managing battles between the players' Pokémon.

```
Play Game:
1. Player 1 Enter your name:
2. Player 2 Enter your name:
3. Players pick their Pokémon teams from a draft.
4. Players send their Pokémon for battle.
5. Players choose moves for their Pokémon.
6. The battle is simulated, and the results are displayed.
7. The game continues until one player's team is completely defeated.
8. The winner is announced, and the game ends.
```

     For details, please refer to the Game flow Section. Watch the videos provided, and listen to the podcast that has been uploaded.

   - **2.Get Pokémon Stats** This provides a detailed view of each Pokémon's attributes, including level, HP, wins, losses, attack, defense, speed, types, and moves. This provides a detailed view of each Pokémon's attributes, including level, HP, wins, losses, attack, defense, speed, types, and moves.

```
Pokémon Stats:
1. Charmander
        -Level: 10
        -HP: 39
        -Wins: 0
        -Losses: 0
```

```
         -Attack: 52
         -Defense: 43
         -Speed: 65
         -Types: [Fire]
         -Moves: [Scratch, Ember, Flamethrower, Tackle]
2. Squirtle
        -Level: 10
        ...
```

- **3.Get Player Stats** This provides a detailed view of each player's statistics, including their name, wins, and losses.

```
Player Stats:
    1. alice (Wins: 0, Losses: 1)
    2. bob (Wins: 1, Losses: 0)
        ...
```

- **4. Get Type Stats**
  This provides a detailed view of each type's attributes, including the name of the type and the list of Pokémon that have this type.

```
Types:
1. Fire Pokémons: [ Charmander ]
2. Water Pokémons: [ Squirtle ]
3. Grass Pokémons: [ Bulbasaur ]
4. Poison Pokémons: [ Bulbasaur, Gengar ]
...
```

- **5. Get Move Stats**
  This provides a detailed view of each move's attributes, including the name, type, category, power, accuracy, and the list of Pokémon that can learn this move.

```
Moves:
1. Scratch (MoveType: Normal, Category: Physical, Power: 40, Accuracy: 95)
Pokémons: [ Charmander, Squirtle, Bulbasaur ]

2. Ember (MoveType: Fire, Category: Special, Power: 40, Accuracy: 85)
Pokémons: [ Charmander ]

3. Flamethrower (MoveType: Fire, Category: Special, Power: 90, Accuracy: 75)
Pokémons: [ Charmander ]
...
```

- **6. Exit**
  This option exits the game. The user is prompted with a goodbye message, and the game terminates.

```
Exiting the game. Goodbye!
```

3. **Game Flow:** When the game starts, the menu would be shown to the user to select from.

   - **Step 1: Get Player Names**

     ```
     Player 1 Enter your name:
     alice
     Player 2, Enter your name:
     bob
     ```

   - **Step 2: Players Pick Pokémons for Their Teams**

     ```
     alice, choose a Pokémon for your team:
     1. Charmander (HP: 39/39, Speed: 65, Level: 10, Types: [Fire])
     2. Squirtle (HP: 44/44, Speed: 43, Level: 10, Types: [Water])
     3. Bulbasaur (HP: 45/45, Speed: 45, Level: 10, Types: [Grass, Poison])
     ...
     1
     bob, choose a Pokémon for your team:
     1. Squirtle (HP: 44/44, Speed: 43, Level: 10, Types: [Water])
     2. Bulbasaur (HP: 45/45, Speed: 45, Level: 10, Types: [Grass, Poison])
     3. Pikachu (HP: 35/35, Speed: 90, Level: 12, Types:
     ...
     1
     alice Team:
     1. Charmander (HP: 39/39, Speed: 65, Level: 10, Types: [Fire])

     bob Team:
     1. Squirtle (HP: 44/44, Speed: 43, Level: 10, Types: [Water])
     ```

   - **Step 3: Send Pokémons for a Round**

     ```
     alice, send a Pokémon for this round:
     1. Charmander (HP: 39/39, Speed: 65, Level: 10, Types: [Fire])
     2. Bulbasaur (HP: 45/45, Speed: 45, Level: 10, Types: [Grass, Poison])
     3. Jigglypuff (HP: 115/115, Speed: 20, Level: 8, Types: [Normal, Fairy])
     1
     bob, send a Pokémon for this round:
     1. Squirtle (HP: 44/44, Speed: 43, Level: 10, Types: [Water])
     2. Pikachu (HP: 35/35, Speed: 90, Level: 12, Types: [Electric])
     3. Gengar (HP: 60/60, Speed: 110, Level: 15, Types: [Ghost, Poison])
     1
     ```

   - **Step 4: Battle Begins** When a move is executed, its hit or miss is calculated using accuracy of the move:

```
hitRate = random number between [0-100] inclusive both 0 and 100.
hitTarget = hitRate < accuracy;
```

If a move has 85% accuracy, it is very likely to hit, compared to 70%.

```
###### BATTLE STARTS #######
######   Charmander   VS.   Squirtle   #######

Charmander is faster than Squirtle! So it attacks first!
###### Pokémon STATS #######
Charmander (HP: 39/39, Speed: 65, Level: 10, Types: [Fire])
Squirtle (HP: 44/44, Speed: 43, Level: 10, Types: [Water])
alice, choose the move for Charmander:
1. Scratch (MoveType: Normal, Category: Physical, Power: 40, Accuracy: 95)
2. Ember (MoveType: Fire, Category: Special, Power: 40, Accuracy: 85)
3. Flamethrower (MoveType: Fire, Category: Special, Power: 90, Accuracy: 75)
4. Tackle (MoveType: Normal, Category: Physical, Power: 40, Accuracy: 90)
1
Charmander used Scratch! It dealt 40 damage!
bob, choose the move for Squirtle:
1. Tackle (MoveType: Normal, Category: Physical, Power: 40, Accuracy: 90)
2. Water Gun (MoveType: Water, Category: Special, Power: 40, Accuracy: 80)
3. Hydro Pump (MoveType: Water, Category: Special, Power: 110, Accuracy: 65)
4. Scratch (MoveType: Normal, Category: Physical, Power: 40, Accuracy: 95)
1
Squirtle used Tackle! It dealt 40 damage!
Charmander fainted!
```

Or (if move misses)

```
Pikachu used Iron Tail and missed.
```

- **Step 5: Conclude Battle** If one team runs out of Pokémon, they player loses:

```
Gengar used Shadow Ball! It dealt 80 damage!
Jigglypuff fainted!
Game Over!
Jane Team Wins!
```

4. **Classes:**

- **Menu Class:**
  The Menu class is designed to handle user interactions in a Pokémon game. It follows the Singleton pattern to ensure only one instance of the menu exists throughout the program. Below are the steps to create the Menu class and its components:

- **Variables**
  - Static private instance of type Menu called menu.
  - An Instance variable of type Scanner to read input.
- Private constructor to prevent instantiation (This will be an empty constructor)
- **Static Methods**
  - public method named **getInstance** that provides access to the single instance of Menu
- **Instance Methods**
  - public **setScanner** method. The Menu class is the only place that will have a scanner object reference. This scanner object is assigned to be the scanner object that was created in Main.java. This scanner address is taken from main where we called the setScanner() function. In no other place in your code you should create or pass any Scanners. For reading input from files, use bufferedReader.
  - **displayMenu** method to just display the 6 menu options.
  - **runMenu** method: This is the core of the Menu class and runs the Menu until the user decides to exit. The runMenu function directs the program to do different tasks by calling functions the are both in Menu class and in other classes.
  - **public int getValidIntegerChoice(int max)**
    - Method to get a valid integer choice from the user. This utility function is called to make sure the input is an integer between 1 and max.
    - You can either hardcode 1 or store it as a constant.
    - This function takes on parameter only representing the max, the min is always 1 because menu choices always start from 1.
    - In this function you will use a loop to validate int but be careful about using nextInt() and next() or nextLine() after that. Make sure that the function handles the input leakage that happens with nextInt(). (Why? remember from COMP1010; When you read integers using nextInt(), the $\backslash n$ character would be passed as the input when you want to read next time with next() or nextLine().)
    - An input such as "2 12 hello" should be accepted if 2 is a valid input. Basically any input after the whitespace would be ignored and only the first term would be evaluated.
    - This function should not use BufferedReader, it is reading input from the stream and should use the scanner instance in the Menu class
  - **public getValidStringInput** mehod to get a valid string input from the user. This utility function is called to make sure the input is only letters. Only letters means that any whitespace in between letters would make it invalid (e.g *wild card* is not valid) it will return a string.

This function should not use BufferedReader, it is reading input from the stream and should use the scanner instance in the Menu class.

- **Move Class:**
  The Move class is designed to represent a move that a Pokémon can learn and use in battles. It manages the move's information, including its name, type, category, power, and accuracy. The class also provides methods to interact with and manipulate move data, such as adding Pokémon that can learn the move, finding moves by name, and displaying move statistics. Additionally, it maintains a static list of all moves.
  - **Instance Variables:**
    - **moveName:** Stores the name of the move (String).
    - **moveType:** Stores the type of the move (Type). Type is not used in any way during the gameplay but you need to store it properly.
    - **category:** Stores the category of the move (physical, special, or status) (String). Category is not used in any way during the gameplay but you need to store it properly.
    - **power:** Stores the power of the move (int).
    - **accuracy:** Stores the accuracy of the move (int).
    - **pokemons:** Stores the list of Pokémon that can use this move (ArrayList<Pokemon>
  - **Static Variables:**
    - **moves:** A static ArrayList to store all moves (ArrayList<Move>). When you load the moves-data.txt, it should be stored into this array list.
  - **Constructor:**
    - **Move(String moveName, Type moveType, String category, int power, int accuracy):** Initializes the move's name, type, category, power, and accuracy.
  - **Instance Methods:**
    - **getMoveName():** Gets the name of the move.
    - **getPower():** Gets the power of the move.
    - **getAccuracy():** Gets the accuracy of the move.
    - **getMoveType():** Gets the type of the move.
    - **addPokemon(Pokmon p):** Adds a Pokémon to the list of Pokémon that can learn this move.
    - **getPokemonsToString():** Returns a string representation of all the Pokémon that can use this move. Example:

      $$Pokemons : [Pikachu, Charmander, Bulbasaur]$$

    - **toString():** Returns a string representation of the move, including moveName, moveType, category, power and accuracy. Example:

      $$ThunderShock(MoveType : Electric, Category : Special, Power : 40, Accuracy : 75)$$

  - **Static Methods:**

    - **getMoves():** Gets the list of all moves.
    - **loadMoves():** Loads moves from the file moves-data.txt. This function would be reading the file, handling Exceptions when opening or reading the file and parse the data and store it in the move arraylist. You may assume the file is valid and they are not empty. The lines are also properly formatted.
    - **findMove(String moveName):** Finds and returns a move by name in the list of moves. if the move isn't there, it should return null.
    - **displayMoveStats():** Displays the list of moves with the Pokémon that can use them. This is used when the user selects to see the Move stats in the Menu. Example: See Step 2.5

- **Type Class:**
  The Type class is designed to represent a type that a Pokémon can have (e.g., Water, Fire). It manages the type's information, including its name and the list of Pokémon that have this type. The class also provides methods to interact with and manipulate type data, such as adding Pokémon to the type, finding types by name, and displaying type statistics. Additionally, it maintains a static list of all types. Each type is created only once and should be referenced and used if it already has been created.
  - **Instance Variables:**
    - **typeName:** Stores the name of the type (String).
    - **Pokemons:** Stores the list of Pokémon that have this type (ArrayList<Pokemon>).
  - **Static Variables:**
    - **types:** A static ArrayList to store all types (ArrayList<Type>).
  - **Constructor:**
    - **Type(String typeName):** Initializes the type's name and the list of Pokémon.
  - **Instance Methods:**
    - **getTypeName():** Gets the name of the type.
    - **addPokemon(Pokemon p):** Adds a Pokémon to the list of Pokémon that have this type.
    - **getPokemons():** Gets the list of Pokémon that have this type.
    - **toString():** Returns the name of the type. Just like getTypeName.
    - **getPokemonsToString():** Returns a string representation of the Pokémon that have this type. Example:

      $$Pokemons : [Pikachu, Charmander, Bulbasaur]$$

  - **Static Methods:**
    - **getTypes():** Gets the list of all types.
    - **displayTypeStats():** Displays the list of types with the Pokémon that have them. Example: Step 2.4

    - **findType(String typeName):** Finds a type by name in the list of types. If the type is not found, you should create the type, add it to the list and return it.

- **Stat Class:**
  The Stat class is designed to represent the statistics of a Pokémon. It manages the Pokémon's HP, attack, defense, speed, and win/loss record. The class provides methods to get and set these statistics, as well as to record wins and losses. Additionally, it includes a method to return a string representation of the statistics.
  - **Instance Variables:**
    - **HP:** Stores the HP of the Pokémon (int).
    - **attack:** Stores the attack stat of the Pokémon (int).
    - **defense:** Stores the defense stat of the Pokémon (int).
    - **speed:** Stores the speed stat of the Pokémon (int).
    - **wins:** Stores the number of wins the Pokémon has (int).
    - **losses:** Stores the number of losses the Pokémon has (int).
  - **Constructor:**
    - **Stat(int HP, int attack, int defense, int speed):** Initializes the HP, attack, defense, speed, wins, and losses of the Pokémon. The wins and losses of a newly created Pokemon is zero.
  - **Instance Methods:**
    - **getHP():** Gets the HP of the Pokémon as int.
    - **getAttack():** Gets the attack stat of the Pokémon.
    - **getDefense():** Gets the defense stat of the Pokémon.
    - **getSpeed():** Gets the speed stat of the Pokémon.
    - **setHP(int HP):** Sets the HP of the Pokémon.
    - **setAttack(int attack):** Sets the attack stat of the Pokémon.
    - **setDefense(int defense):** Sets the defense stat of the Pokémon.
    - **setSpeed(int speed):** Sets the speed stat of the Pokémon.
    - **recordWin():** Increments the win count of the Pokémon.
    - **recordLoss():** Increments the loss count of the Pokémon.
    - **getWins():** Gets the number of wins the Pokémon has.
    - **getLosses():** Gets the number of losses the Pokémon has.
    - **toString():** Returns a string representation of the statistics. Example:
      $$HP : 100, Attack : 50, Defense : 75, Speed : 80$$

- **Pokémon Class:**
  The Pokémon class is designed to represent a Pokémon in the game. It manages the Pokémon's information, including its name, level, statistics, current HP, types that the Pokémon is, and its moves. The class also provides methods to interact with and manipulate Pokémon data, such as checking if a Pokémon

has fainted, taking damage, attacking an opponent, and displaying Pokémon statistics. Additionally, it maintains a static list of all Pokémon and keeps track of the total number of Pokémon.

- **Instance Variables:**
  - **name:** Stores the Pokémon's name (String).
  - **level:** Stores the Pokémon's level (int).
  - **statistics:** Stores the Pokémon's statistics (Stat).
  - **currentHP:** Stores the Pokémon's current HP (int).
  - **types:** Stores the Pokémon's types ($ArrayList < Type >$).
  - **moves:** Stores the Pokémon's moves ($ArrayList < Move >$).
- **Static Variables:**
  - **Pokemons:** A static ArrayList to store all Pokémon ($ArrayList < Pokemon >$).
- **Constructor:**
  - **Pokemon(String name, int level, Stat statistics):** Initializes the Pokémon's name, level, statistics, current HP, types, and moves.
- **Instance Methods:**
  - **isFainted():** Checks if the Pokémon has fainted.
  - **boolean takeDamage(int damage):** Takes damage and updates the Pokémon's current HP. Records a loss if the Pokémon faints. If the Pokémon faints it should only update the Pokémons loss stats. The win stats will be updated in the attack function. It will return true if the Pokemon faints and false otherwise.
  - **attack(Pokemon opponent, Move move):** Attacks an opponent Pokémon with a specified move. Calculates the damage dealt based on hit or miss and updates the opponent's HP. It also records a win if the opponent faints. Considering the move's accuracy, you should determine if the attack lands or not. To do this, generate a random number from 0-100 If landed, the opponent would take that damage, and if not damage would be zero. If the attack faints the opponent, you should record the win for the pokemon and the loss for the opponent. This function would return the damage as int. This function should also print messages accordingly:
    - *Charmander used Scratch! It dealt 40 damage!*
    - *Pikachu used Iron Tail and missed.*
  - **getName():** Gets the name of the Pokémon.
  - **getHP():** Gets the current and maximum HP of the Pokémon as a string. *Example: 10/100 or 40/40*
  - **getSpeed():** Gets the speed of the Pokémon.
  - **getMoves():** Gets the moves of the Pokémon as an ArrayList.
  - **isFasterThan(Pokemon opponent):** Determines if the Pokémon is faster than the opponent. If they have the same speed, choose one randomly with a 50-50 chance. It returns a boolean.

* **toString():** Returns a string representation of the Pokémon, including the name, HP, speed, level, and types. Example: *Pikachu (HP: 40/40, Speed: 50, Level: 10, Types: [Fire, Water])*
* **addType(Type type):** Adds a type to the Pokémon.
* **addMove(Move move):** Adds a move to the Pokémon.
* **toStringMoves():** Returns a string representation of the moves of the Pokémon. Example:

$$\{Tackle, Growl, Ember, Scratch\}$$

* **toStringTypes():** Returns a string representation of the types of the Pokémon. Example:

$$\{Fire, Flying\}$$

* **restoreHP():** Restores the HP of the Pokémon to its maximum. The maximum is the value in the statistics of the pokemon.
* **getStat():** Gets a string representation of the Pokémon's info and statistics. This is way more comprehensive than toString and will be used to show the second option of the menu, get Pokémon stats. Example: Each pokemon in Step 2.2

– **Static Methods:**

* **getPokemons():** Gets the list of all Pokémon.
* **displayPokemonStats():** Displays the stats of all Pokémon. This function is called by the Menu singleton instance. Example: This function prints all Pokemons in Step 2.2 but getStat() is for one of the pokemons.
* **restoreAllHP():** Restores the HP of all Pokémon to their maximum.
* **loadPokemons():** Loads Pokémon from the file Pokémon-data.txt. Reads the Pokémon data from the file and loads them into the static list of Pokémon. This function must handle the potential Exceptions such as FileNotFoundException. This function should properly read the file, ignore the first line, and parse the lines based on comma. You may assume the file is properly formatted and will always have more than 1 Pokémons and the number of Pokémons will be even so each team will have equal number of Pokémon for battle. You should be loading the Moves data first and then attempt to load the Pokémon files. If you get an exception, print the exception message and let the program halt.

- **PokemonTeam Class:**
  The PokémonTeam class is designed to manage a team of Pokémon. It provides methods to add, remove, and retrieve Pokémon from the team, as well as to update the team by removing fainted Pokémon. The class also includes methods to get the size of the team and to return a string representation of the team.

  – **Instance Variables:**

    - **team:** Stores the list of Pokémon in the team (ArrayList<Pokemon>).
  - **Constructor:**
    - **PokémonTeam():** Initializes the list of Pokémon.
  - **Instance Methods:**
    - **addPokemon(Pokemon pokemon):** Adds a Pokémon to the team.
    - **getPokemon(int index):** Retrieves a Pokémon from the team by index.
    - **removePokemon(int index):** Removes a Pokémon from the team by index.
    - **size():** Returns the number of Pokémon in the team.
    - **updateTeam():** Removes fainted Pokémon from the team.
    - **getTeam():** Returns the list of Pokémon in the team.
    - **toString():** Returns a string representation of the team. The output should be like the gameplay sample so the player could choose from the list of Pokémon they have in their team. Example: Step 3.3
    - **clearTeam** function clears the team.

- **Player Class:**
  The Player class is designed to represent a player in a Pokémon game. It manages the player's information, including their name, team of Pokémon, and their win/loss record. The class also provides methods to interact with and manipulate player data, such as recording wins, finding players, and displaying player statistics. It also stores and ArrayList of Players so all the players who ever play will be stored and remembered later, including their stats. This ArrayList will obviously need to be static.
  - **Instance Variables and Methods**
    - Stores the player's name (String), team(PokemonTeam), wins(int), and losses(int)
    - **getName** and **getTeam** functions to get those variables
    - **wonAgainst** function that updated the win stat of the current player and the loss stat of the other player passed to it as a parameter.
    - **toString** function that reports the player name and their wins and losses. (Example: *Alice (Wins: 2, Losses: 1)*)
  - **Constructor** that only takes the player name. Other variables should be initializes to 0 created as an empty object.
  - **Static Variables and Methods**
    - Stores players (ArrayList<Player>) to keep track of all the players who ever played the game.
    - **findPlayer** function that takes a player name, searches through the arraylist of players and returns that player. If the player is playing for the first time and is not found, create that player and add it to the list, then return it.

    - **displayPlayerStats** function should do what the 3rd option in the menu is doing; showing the stat of all the players in a formatted way like the example.
    - **clearPlayers** function should clear the list of all Players.

- **Game Class:**
  The Game class is designed to manage the flow of the Pokémon game. It handles player interactions, Pokémon selection, and battle simulation. The class provides methods to get player names, pick Pokémon, send Pokémon for battle, choose moves, and simulate fights. Additionally, it maintains references to the players and the menu.
  - **Instance Variables:**
    - **player1:** Stores the first player (Player).
    - **player2:** Stores the second player (Player).
    - **menu:** Stores the menu instance (Menu). Through menu instance, you can access the functions in the Menu class. That is basically how singleton works and is used.
  - **Constructor:**
    - **Game():** Initializes the menu instance by calling static function getInstance() from the Menu class.
  - **Instance Methods:**
    - **setPlayer1** method that sets the player1.
    - **setPlayer2** method that sets the player2.
    - **getPlayer1** method that returns player1.
    - **getPlayer2** method that returns player2.
    - **simulateGame():** Starts the game simulation by getting player names and starting the battle. This will handle the game from beginning to the end. In this function you can call other functions for modularization. After the simulateGame is finished, you need to still keep the teams from the last time the players played the game. Each time a new simulateGame() is called, clear the previous teams of player 1 and 2.
    - **getPlayerNames():** Prompts the players to enter their names and finds or creates the players. Once they enter their names, the findPlayer function should see if they have played before or not and return the player. It should handle the scenarios where the player has played before and not played before.
    - **pickPokemons():** Allows players to pick their Pokémon teams from a draft. Any changes here shouldn't affect the Pokémons list in the Pokémon class. If any Pokémon it picked or fainted, that shouldn't affect the output of the statistics in the other menu options. The players will take turns until they pick three Pokémons each or until the list of Pokémons runs out. On each turn, you should show the Pokémon in each team.

    - **sendPokemon(Player player):** Prompts a player to send a Pokémon for battle. The player will get a change to choose from the remaining not fainted Pokémon they have to send for this round of the battle.
    - **choosePokemonMove(Player player, Pokemon pokemon):** Prompts a player to choose a move for their Pokémon. This function will know the player and the Pokémon and will ask them what move they are choosing from the list of available moves for that Pokemon and return the move they selected. A player can choose one move many times so the same list of moves that a pokemon has would be available to choose from every time.
    - **startBattle():** Manages the battle between the players' two Pokémon from beginning to the end. It should let players pick their Pokémon from the draft which is the list of Pokémon. Then ask the players to choose and send a Pokémon from their team. Then simulate the fight, update the teams to remove any fainted Pokémon, and either conclude the battle or keep continuing the battle based on each teams status. A player loses if all their Pokémon faint. After the battle is done, report accordingly (Just like the examples) and make sure the HP of the Pokémons are back to normal and no Pokémon has been deleted or changed. You should update the Player win/loss stats as well.
    - **simulateFight(Pokemon player1Pokemon, Pokemon player2Pokemon):** Simulates a fight between two Pokémon. Determines which Pokémon goes first, asks the user to choose the move for the Pokémon. A move could be used many times. If the other Pokémon didn't faint, they get to choose a move and retaliate. You should also update the Pokémon win/loss stats as well.

- **Colors Class:**
  The Colors class is designed to provide ANSI escape codes for coloring text output in the console. These color codes can be used to enhance the visual presentation of text in the console, making it easier to distinguish different types of messages (e.g., errors, success messages, warnings). The class defines a set of constants for various colors and their bright variants. Note that using these colors is optional and not necessary for the functionality of the program. **They don't have any points.**

Submission Instructions

File Requirements: Submit all necessary Java files. Your submission should include:

- `Main.java`(Main file that was provided and shouldn't have been changed)
- `Pokemon.java`
- `PokemonTeam.java`
- `Type.java`
- `Move.java`

- `Stat.java`
- `Game.java`
- `Menu.java`
- `Player.java`
- `Colors.java` (Colors file that was provided and shouldn't have been changed and just upload to prevent unexpected error)

Ensure that your files follow the exact naming convention.

Programming Standards: Your code should follow standard Java programming practices as outlined in UMLearn, including:
- Proper indentation and formatting
- Meaningful variable and method names
- Encapsulation and appropriate use of access modifiers
- Proper use of comments to explain logic

Unfinished Sections/Functions: If any parts of your assignment are incomplete, include comments explaining:
- What is missing
- How you intended to implement it
- Any challenges you faced

Tips

- Read the instruction several times before attempting to plan your code.
- Do not attepmt to write the code unless you have the code plan mapped out on paper.
- Check Piazza questions to find potential answers to your question.
- Plan your code before implementation by sketching out class relationships and game logic.
- Write additional helper methods where necessary to improve code organization and readability.
- Follow the object-oriented principles of encapsulation and modularity.
- Test individual components before integrating them into the full program.
- Think through the battle logic, including hit rate and accuracy and damage deductions, before implementation. Most of the logic goes into the Game and Menu classes.