**Chutes and Ladders (Desktop version)**

This is the simulation of the classic children’s game “*Chutes And Ladders.”*

**Assumptions:**

1. Player names need to be provided on the command line.
2. At least 2 and at the max 4 players can start the game.
3. If only 1 player name is provided, then it will throw an exception and the game won’t start.
4. If player names are not provided at all, then this game assumes 4 players with their names as Player 1, Player 2, Player 3, Player 4
5. Considering the requirements mentioned in the document, this code does not support the GUI version.
6. To decide the first player, everyone spins the spinner. Whoever gets the highest score goes first. To comply with this requirement, random numbers within 1 to 6 are generated, and those are allocated to each player. Now the players will be sorted in the descending order of these values. The player having the max value will start the game. It is assumed that rest of the players will have their turn based on the order of the random number generated (the player having least random number will go last). It is possible that duplicate random numbers are generated. To avoid this scenario, every random number is checked with the previously generated numbers, and duplicates are avoided.
7. Any player may reach to a number greater than 100 (home position). While he gets to the position which less than or equal to 100, he has to wait and keep on playing.
8. Among the participant players, if any one of them reaches the home position, the game stops.

**Design decisions:**

*Class model:* To play the games, we need three things:

1. Players
2. Spinner
3. Game board

To represent each of them, 3 different POJOs are created.

*Design patterns:* This code utilizes two design patterns- Factory method pattern and Singleton pattern.

So far, it is assumed that to play any game; we need at least two methods- arranging players/deciding the sequence of players & actual game play logic. Hence, these methods are declared in an Interface named Game.

Interface name: Game

Methods supported:

**public** List<Player> setUpPlayers(String[] playerNames);

**public** String playGame(List<Player> players);

Implementer class: ChutesAndLaddersGame

So, in the future if it is needed to add another game, it just needs to implement this interface.

Considering there can be multiple games following the contract specified in the interface, there is a factory class which can provide the instance of any of the game. This instantiation is based on the singleton design pattern. The simulator (main method in this case) provides the choice of the game to be played to the factory method.

An attempt is made to make this application more generic. The code is structured in such a way that if a new game logic needs to be added then it is not necessary to modify the existing structure. For example, if this application is supposed to incorporate Tic-tac-Toe game, then expected code changes are:

1. Add a separate class having main method. This class can be created in a separate package inside existing *com.candidate*

2. Add a new switch case in the GameFactory class

3. Add a new service class in the package *com.simulator.game.service* which has to extend Game interface

4. Existing player model can be used as it is. Otherwise new fields (and their corresponding setter/getter) need to be added.

5. Utility methods, constants can be added in their corresponding packages.

In short, the project structure will remain as it is; only the newly required classes need to be added in the appropriate package structure.

**How to run the code:**

All of the Java classes are in the src folder. Extract the zip file. First, it is necessary to compile the code. The java version required to be at least 1.8

After the successful compilation go to the bin directory and execute below command (for command line execution).

*java com.candidate.chutesladders.ChutesAndLadders*

This command will assume 4 default players (names as Player 1, Player 2, Player 3, Player 4). If the name of the players is to be provided, then execute the below command:

*java com.candidate.chutesladders.ChutesAndLadders Eric Paul*

Here, the names of two players are provided are Eric and Paul

**To-do/Future enhancements:**

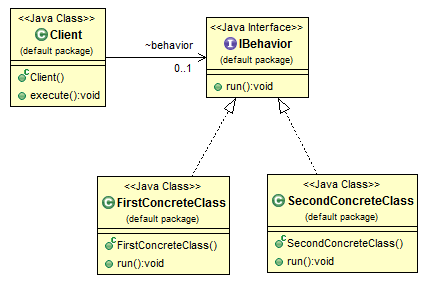
This section will list the possible enhancements to be made in this application if it is a "real" project.

1) Strategy pattern: This pattern will increase the reusability of the code. As per this pattern, setting up the players & game play logic can be considered as 2 different behaviors. For this, two different interfaces having these two behaviors need to be created. For each behavior, there will be an interface.

In addition to this, there will be a single class, which will hold the references for these two interfaces. In short, this class will have a dependency on these two interfaces. Any class which wants to instantiate this class has to provide the dependencies.

Since the behaviors are injected from outside the class, any two different games sharing the logic for any of the behaviors can reuse the code.

General class diagram for a strategy pattern:



2) MVC architecture: The “Control – Model – View” architecture works to divide the work within an application. The “Control” talks in user, gets the data and passes it to the “Model.” The “Model” uses data structures, Classes, instances, algorithms, and functions to calculate the Game state at specific intervals. The “Model” then passes this data to the “View” which draws our outputs Game data to the screen for the user/players. The flow of the application will be like this. Broadly it can be separated into 4 different pages/steps.

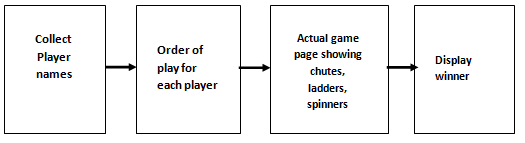
i. Ask the names of the players. Proper validations (number of players) can be placed using JavaScript. Once validated correctly redirect to the second page

ii. Generate a random number for each player to decide who will play first. This page can be made more interactive by placing visually moving spinner.

Decide the first player and indicate it accordingly on the screen.

iii. Actual game page having display board, spinner, the position of each player, etc.

iv. Winner page displaying the winner’s name



3) Service-oriented architecture: The business logic, i.e., which player will go first, the next position of the player based on the spinner result, chutes & ladders, the winner of the game, etc. can be separated to the service. Based on the service response, the client application will decide the page or the content of the page to be rendered, next position of the player on the board etc.

4) TDD approach: Test cases act as documentation for the code. It helps to check whether recent code changes have not broken the previously working code.

5) Data persistence: Persisting the data can help to add the new functionality like “Leaderboard” to the application. The data that can be persisted:

1. Participant players

2. Winner

3. Number of moves to find the winner

4. Time at which winner was declared

5. The location (country/state/county) of the player.

This will help to create the leader board among players from a specific country, state, or a county. The factor to consider for a “Leaderboard” is to use the number of moves to reach the home position. Whoever has the minimum number of moves, will be the top in the list.