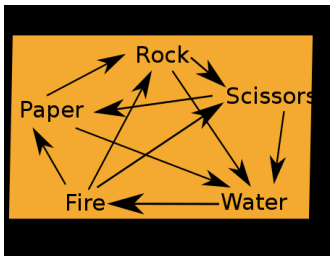
As described in the wiki page of Rock Paper Scissors, the game has at least 3 moves and requires at least 2 players. Therefore I focused on extensibility on my solution, addressing the questions of:

1. What if there are more than 2 players?
2. What if there are more than 2 moves?

Underneath an explanation of the classes can be found:

Move

The moves are pre-set actions that players can take in the game. Thus they are represented as an enum. Currently there are only 3 in my implementation, however, the wiki page explains there are other versions we can extend to, for instance, with the elements:



Player

The requirements markdown document explains that we have a human and computer player. They share the implementation of many methods. Therefore I created an abstract class called **Player** and extended it with **ComputerPlayer** and **HumanPlayer** classes. The only difference between them is ComputerPlayer having a ***setRandomMove*** method.

Score

Score is simply a java record keeping player - score binding as a pair. It is used in ScoreBoard class for ordering purposes.

ScoreBoard

Scoreboard keeps a player to integer score mapping of the game. Despite the requirement being just for 2 players of user vs computer, here I also kept extensibility in mind. I wanted a generic solution that can handle it all. What if there are 1000 people playing rock paper scissors online at once?

By keeping the player to score binding in a hashmap we get O(1) time complexity when we want to increment the score of a user instead of doing a linear search in an array/list.

GameManager
This is where the game is set up and rounds are managed. The class is initiated with the amounts of human and computer players. There is also a maximum rounds field which comes in handy when only computers are playing against each other, as then there is no human to initiate an exit. GameManager will initiate a Round object per round where a round will be played and scores are updated.

Round
This is where the actual rounds of games are played! The HumanPlayer inputs are asked and ComputerPlayer inputs are chosen by random. Then we are checking the winners.

The interesting part in this class is the ***playersByMove*** map. This is a mapping done per round of a move to the players who chose that move.
The mapping comes in handy in situations where the game is scaled up with many players. Instead of doing an O(amountOfPlayers^2) loop of checking the winning situation of each player to others, we instead just compare moves that are winning against each other and incrementing the scores of all the winning players.

“Naive” implementation:
PlayerList: *Player 1, Player 2, Player 3, Player 4, Player 5, Player 6, Player 7, Player 8*

player1.winsAgainst(player2) ? incrementScore(player1)
player1.winsAgainst(player3) ? incrementScore(player1)
…
*Complexity O(amountOfPlayers^2 * amountOfMoves)*

My implementation:
Rock *-> Player 1, Player 2, Player 3*
Paper *-> Player 4, Player 5, Player 6*
Scissors *-> Player 7, Player 8*

Rock.winstAgainst(Paper) ? Rock.getPlayers.forEach(incrementScore)
Rock.winsAgainst(Scissor) ? Rock.getPlayers.forEach(incrementScore)
…
*Complexity O(amountOfMoves^2 * amountOfPlayers)*

The hashmap implementation that I wrote would perform better in the cases where there are some more players than the moves. To be precise it makes sense for instance if there are 4 players whilst there are still 3 moves.
Naive implementation -> 4^2 * 3 = 48
Hashmap implementation -> 3^2 * 4 = 36