Solutıon explanatıon by Isitan Görkey

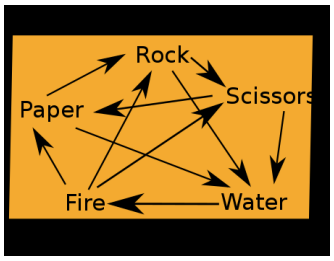
As described in the wiki page of Rock Paper Scissors, the game has at least 3 moves and requires at least 2 players. Therefore I focused on extensibility on my solution, addressing the questions of:

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 scaling up. 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 incrementing the scores of the players against each player they made a winning move against.

Player 1 -> ROCK
Player 2 -> ROCK
Player 3 -> SCISSOR
Player 5 -> PAPER

Player 5 will get 2 points for winning the round against Player 1 and Player 2
Player 1 will get 1 point for winning the round against Player 3
Player 2 will get 1 point for winning the round against Player 3
Player 1 and Player 2 have a tie for having played the same move

Notes of consideration for the solution

- An important decision point is how to represent the results. Unlike the win, tie, and loss results mentioned in the readme, the game gets more complex and turns into a game of ranking when there are more than 2 players. Per round, players win against some and lose against others. Therefore instead of going with an enum representing the result, I choose to go with a score ranking handled by the ScoreBoard class.
- I chose to output the winning and tie situations of combinations of 2 players per Round but didn't separate the scoreboard of each round as only the end result matters to the game. This way we see an overview of the situation per round but also have the end result kept in a shared ScoreBoard.
- In the Round class, I also considered a solution where I kept a hashmap of moves to a list of players that chose that move in the round.

  ROCK -> Player1, Player2
  PAPER -> Player3, Player4
  SCISSORS -> Player5

  Suppose we then adjust the ***Move.winsAgainst*** to use a hashmap of Move to HashSet<Move> in order to find whether a move wins against the other at O(1) complexity. Then instead of having an O(playerCount^2) complexity for scoring by checking each players' move against the others', we would have O(moveCount^2 * playerCount). Because in this solution we would check each move to the other move and

if the first winsAgainst the other move, then we would update scores of each player in its list.

To meet the condition O(moveCount^2 * playerCount) < O(playerCount^2), we can solve the equation:
moveCount^2 < playerCount

So the move hashMap solution only makes sense if we have >= 10 players if we keep the move count at 3 (3^2 < 10). If we go up to 5 moves, then it makes sense if there are >= 26 players (5^2 < 26). For keeping it performant in the simple/realistic case of this study, I chose the O(playerCount^2) solution.