Java Chess - Pawn Race

161 (C1/J1) Computing Practical 1

2nd – 6th December 2019

Aims

- To gain further experience in writing loops.
- To practice writing classes and manipulating objects.
- To gain experience in object-oriented design.
- To write a simple game-playing program in Java.

The Game

Gerry and Bobby meet on a plane. They discover that they both like chess, and that one of them is carrying a portable chess board. But alas! Gerry has to sneeze as he opens the board, and a few pieces fall out! After a short but frustrating search, they give up and decide to delay further searching efforts until the plane has landed. Taking a look at the remaining pieces, Bobby has an idea: He challenges Gerry to a pawn race. The rules would be simple – the game consists only of pawns, which are all set in their usual starting positions; the player who first promotes one of his pawns to the last rank wins. After briefly thinking about it, Gerry decides to accept the challenge, but with one slight modification: Each player would only play with seven pawns, thus leaving a gap somewhere in the line of pawns. Since white has the advantage of starting the game, Gerry thought it would only be fair if the black player chooses where the gaps are.

How to play

Board and setup

Pawn races are played on a normal chess board, with 8x8 squares. Rows are commonly referred to as *ranks*, and are labelled 1-8, while columns are referred to as *files*, labelled A-H. From white's perspective, the square in the bottom left corner would thus be referred to as a1, while the bottom right corner is h1. White's pawns are all placed on the second rank initially, while black starts from the seventh rank. Figure 1 shows an example of an initial setup, in which the gaps were chosen on the H and A files, for white and black respectively.

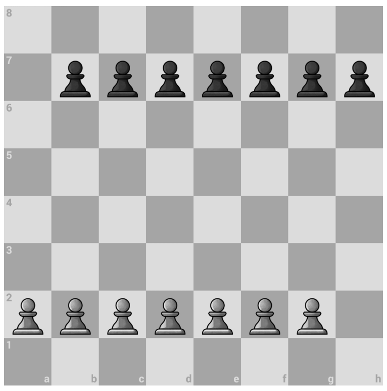

Figure 1: A possible starting position of the pawn race, in which the gaps are set to H (for white) and A (for black).

Pawns and pawn moves

Pawns are considered the simplest pieces on the chess board, yet they often build the back-bone for even very advanced and complex strategies employed by grandmasters. This is despite the fact that unlike other chess pieces, pawns cannot make particularly complex moves, can only move in very limited ways, and only in the forward direction. To illustrate how pawns can move around the chess board, Figure 2 shows a few simple moves in order (note, that white always begins the game). The following rules apply:

- A pawn can move straight forward by 1 square, if the targeted square is empty.
- A pawn can move straight forward by 2 squares, if it is on its starting position, and both the targeted square and the passed-through square are empty.
- A pawn can move diagonally forward by 1 square, iff that square is occupied by an opposite-coloured pawn. This constitutes a capture, and the captured pawn is taken off the board.
- Combining the previous two rules, if a pawn has moved forward by 2 squares in the last move played, it may be captured on the square that it passed through. This special type of capture is a capture *in passing* and commonly referred to as the *En Passant rule.* A pawn can only be captured en passant immediately after it moved forward two squares, but not at any later stage in the game. An example is shown in Figure 3.

Algebraic chess notation

There are many ways of denoting moves in a chess game. The most popular one, however, is certainly the standard algebraic notation, which is also used by FIDE, the Federation Interna-

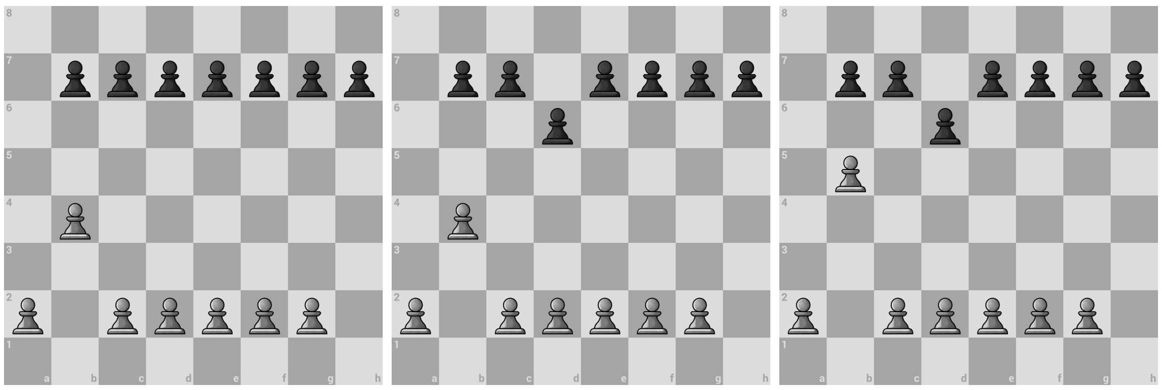

Figure 2: The moves b4 → d6 → b5 in order.

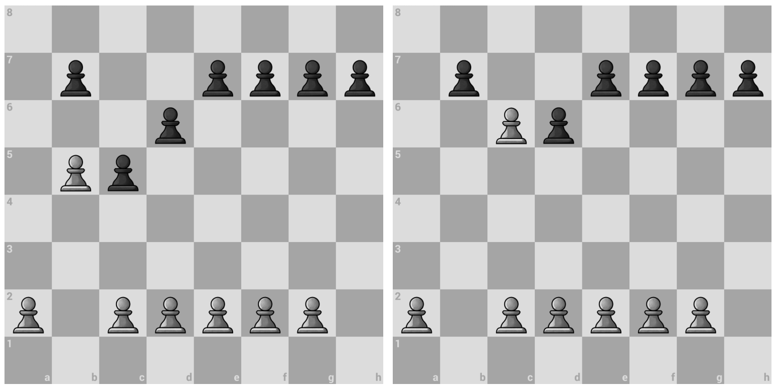

Figure 3: Continuing from Figure 2, if black now advances his C pawn by 2 steps, playing c5, white can capture this pawn en passant and play bxc6.

tionale des Echecs or World Chess Federation, across all competetive matches. Its variant, the long algebraic notation, records both start and target square of any move (sperated by a dash). However, in standard algebraic notation the starting square is omitted, if there is no ambiguity. Figure 2 shows the start of a game in which white opens with b2-b4. Since there is only one pawn which could make that move, the starting coordinate is omitted, and the move is simply denoted as b4. The black player responds by playing the move d7-d6, or simply: d6. White then follows by b4-b5, i.e. b5. You may find that for any straight-forward moving pawn, the starting square can always be omitted, as there can never be ambiguity.

Captures are denoted by an x instead of a dash, and in order to disambiguate the move, the starting file of the capturing pawn is always included. Figure 3 shows an example of an en passant capture, in which black plays c5, followed by white's b5xc6, or in short bxc6, which is read as "b takes on c6".

There is a lot more to algebraic chess notation, but for our pawn race, this subset will suffice.

Gameplay

Traditionally, the white player always starts the game. Both players take turns to make moves. If a player cannot make any valid move because all his pawns are blocked from moving, the game is considered a stale-mate, which is a draw. Whichever player first manages to promote one of his pawns all the way to the last rank, as seen from his own perspective, wins the game. However, the game can also be won by a player capturing all of the opponent's pawns.

Advanced game tactics

This section seeks to cover a few common and important tactics to outline some key concepts of the game.

Pawn chains

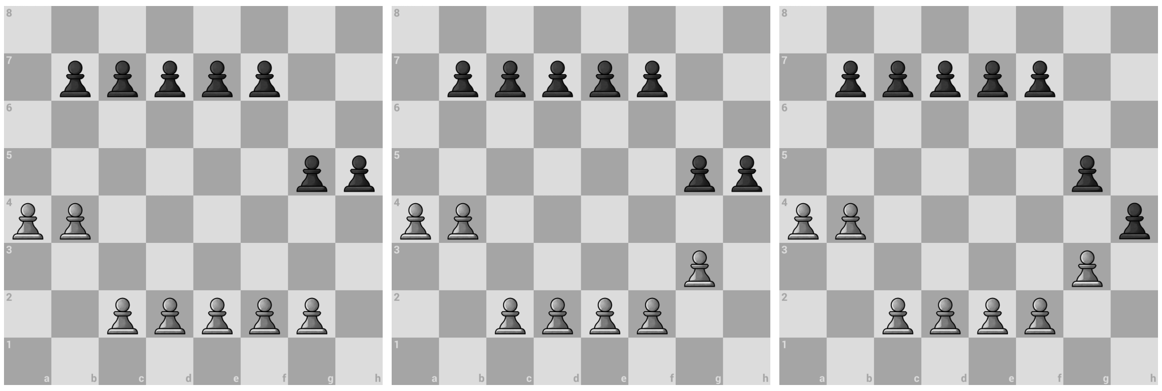

Figure 4: After a4 → h5 → b4 → g5, the above position is reached. White plays a defensive move g3, which turns out to be fatal, as black can establish a small *pawn chain* and easily win the game.

Figure 4 shows the concept of a pawn chain. After white's rather horrible move g3, black responds by playing h4. Black's H pawn is thus backed up by his G pawn, such that if white captures, black will simply recapture (denoted as gxh4 followed by gxh4). Whereas if white does any other move, black can simply move his H pawn forward, and quickly win the game. Black's G and H pawns form a *pawn chain*, in which one pawn is defended by another. The above example shows how powerful pawn chains can be, even if they are short and consist of only two pawns. Once this pawn chain is established, white has little chance to prevent black from winning, since black is now able to pass white's lines, no matter what white decides to do.

(Advanced) A much better defensive move for white instead of g3 would have been f3. Black could not have immediately established a pawn chain, but would have had to move both his pawns further forward. White's pawn on f3 would then guard the g4 square, thus not allowing black to establish a pawn chain on g4 and h3. As you can see, the weakest spot in a pawn chain is usually its back.

(Advanced) A pawn that has no neighbouring pawns of the same colour around itself is commonly referred to as an *isolated pawn*. Isolated pawns are considered weak in many situations, as they cannot be part of any pawn chain. Figure 7 (see below) shows an example of a particularly useless isolated pawn for the black player, which will not be able to participate meaningfully in the game. White also has an isolated pawn on the D file, but this pawn, despite its weakness, will prove useful for defending.

Blocking moves

Figure 5 shows white attempting a poorly executed attack on the left hand side of the board, and an effective defense by the black player. In order to prevent white from playing a5, black decides to play b6, which both blocks whites B pawn, and prevents his *backward pawn* on the A file from advancing to the a5 square. If white does play a5, black will simply capture it and will remain one pawn up, an important material advantage over white. (Advanced: Black's resulting A pawn would then also count as a *passed pawn* - see below - leading to certain victory).

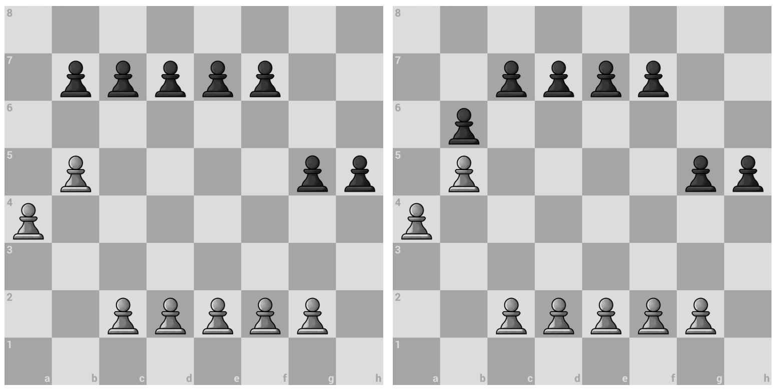

Figure 5: Continuing from Figure 4, white decides to attack instead and plays b5, which he intends to follow up by a5 and then a6. But black can simply counter this with b6, an effective blocking move, blocking the white B pawn, and preventing white's A pawn from moving forward.

Zugzwang

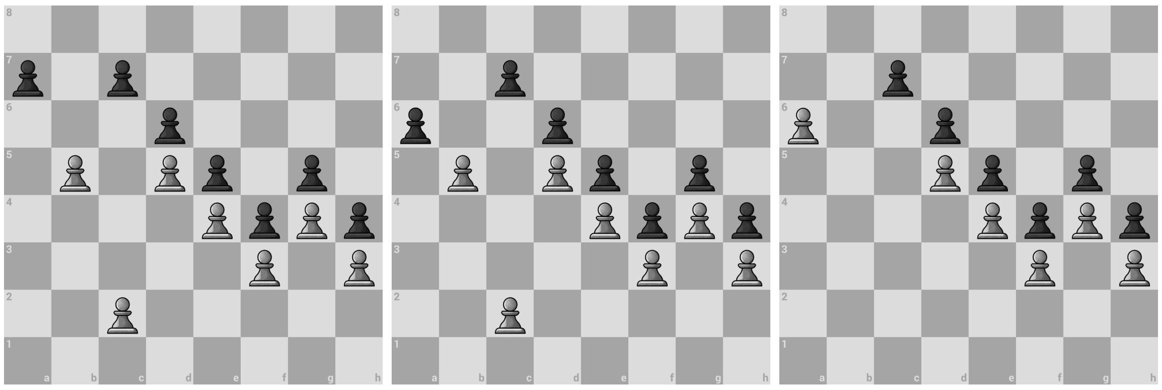

Figure 6: It is black's move – a classic example of *Zugzwang*, in which black is compelled to move, even though he may not want to. With only bad options to choose from, black decides to play a6, which white will capture and then win the game. Black would love to skip a few turns and let white run into his defenses, but that sadly isn't allowed.

The German term *Zugzwang* means that a player is compelled to move, and might thus be compelled to make a move even though he knows it to be bad. He may even wish to skip the move, but that is not allowed according to the rules. Figure 6 shows such a situation, in which there are only bad moves for the black player to make. No matter what he decides to do, he knows that the outcome will be bad.

(Advanced) Note, when your opponent is in Zugzwang, stale-mate is often not far. In the above example, white can go on to win the game. Imagine however, for instance, that in the above example white's C pawn was on the c6 square, thus blocking black's C pawn. If black was then to play a6, white cannot capture the A pawn, or the game would be counted a draw (what a loss!). Instead, white has to let the pawn pass and play b6 to gain the victory. Try it out on a chess board!

Passed pawns

A *passed pawn* is a pawn that has passed the lines of opposite-coloured pawns. It is considered a passed pawn, iff it can neither be blocked nor captured by any of the opponent's pawns. I.e.,

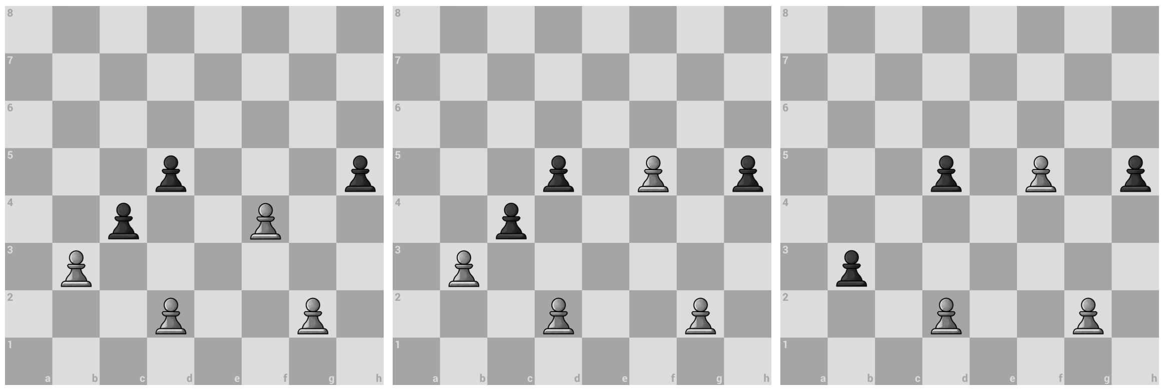

Figure 7: It is white's move. White optimistically decides to push his *passed pawn* on the F file forward, instead of playing a defensive move (bxc4). Yet, forgetting to defend immediately backfires on white, as black will now be able to create a passed pawn himself, and proceed to win the game.

no opposite-coloured pawn may be on either the same or one of the immediately adjacent files. Figure 7 shows a (fictional) example highlighting the importance and pitfalls around passed pawns. White has a passed pawn on f4 which can advance unhindered to the last rank. White might be tempted to think that he has a passed pawn while black does not, and so decides to do advance his passed pawn. Yet, by choosing to promote his passed pawn, he effectively forgets to defend, leaving black the opportunity to create a passed pawn himself. Black thus plays cxb3, capturing white's pawn and creating a passed pawn that is further advanced than white's passed pawn and closer to the final rank. Black will be able to outpace white's pawn, and will proceed to win the game, as simple counting reveals. As the example shows, even in the presence of passed pawns it may still be necessary to defend and play very accurately in order to retain the advantage and secure the win.

What to do

As this exercise is not assessed by your PPTs, and is to be written from scratch, the skeleton files provided will be empty. This section outlines a suggested design.

In order to clone your exercise skeleton repository (remembering to replace *username* with your username), use:

```
> git clone
  https://gitlab.doc.ic.ac.uk/lab1920_autumn/javapawnrace_username.git
```

Enum `Color`

The `Color` enumerated type can be used to describe both players and pawns. We may want to say that a player is either `BLACK` or `WHITE`, and that any given square on the board is occupied by `BLACK`, `WHITE`, or `NONE`. For simplicity, we may choose to combine these three values into one enum type.

Class `Square`

The `Square` class represents one single square on the chess board (see below). It knows about its own coordinates (such that we can use it to describe moves), as well as what `Color` it is occupied by. At this stage, locations are best recorded as integer coordinates on the underlying chess board (0..7, 0..7), and not in the otherwise used string notation (a..h, 1..8). The following methods are suggested:

Constructor `Square(int x, int y)`
Initialises an empty square with the given coordinates.

Method `int getX()`
Returns the x coordinate (0..7).

Method `int getY()`
Returns the y coordinate (0..7).

Method `Color occupiedBy()`
Returns the occupant of this square.

Method `void setOccupier(Color color)`
Sets the occupier of this square.

Class `Move`

The `Move` class captures a single move in the game. We would like to record both the starting and the target locations in terms of their `Square`s (corresponding to long algebraic notation), and whether the move was a capture. This may appear to be more information than strictly needed, but recording all these details will allow us to undo moves later. The move class captures all these variables, which are all set in its constructor:

Constructor `Move(Square from, Square to, boolean isCapture, boolean isEnPassantCapture)`

Creates a new `Move` with all relevant information about a move.

Method `Square getFrom()`
Returns the originating `Square` of the move.

Method `Square getTo()`
Returns the target `Square` of the move.

Method `boolean isCapture()`
Returns whether this move is a capture.

Method `boolean isEnPassantCapture()`
Returns whether this move was an en passant capture.

Method `String getSAN()`
Returns the *short algebraic notation* (see above) for this move.

Class `Board`

The `Board` class keeps track of the current game. It primarily wraps an 8x8 array of `Square`s, which together make up the chess board. The `Board` class is responsible for the initial game setup, and can both apply and un-apply moves. The following methods are suggested:

Constructor `Board(char whiteGap, char blackGap)`
Creates a board with `8x8 Square`s, with all `Square`s initialised, with all pawns setup, and with the pawn gaps in the correct places.

Method `Square getSquare(int x, int y)`
Returns the `Square` at the given integer coordinates.

Method `void applyMove(Move move)`
Applies the given move and updates the board accordingly.

Method `void unapplyMove(Move move)`
Unapplies the last (given) move and updates the board accordingly.

Method `void display()`
Prints the board to the console. For example, at the start of the game shown in Figure 1, your output may look as follows – pawns could simply be denoted as B and W, respectively (or you may wish to make use of the unicode characters, `(char) 9817` and `(char) 9823`), while empty squares can be denoted by a dot:

```
    A B C D E F G H

8   . . . . . . . .   8
7   . B B B B B B B   7
6   . . . . . . . .   6
5   . . . . . . . .   5
4   . . . . . . . .   4
3   . . . . . . . .   3
2   W W W W W W W .   2
1   . . . . . . . .   1

    A B C D E F G H
```

Class `Game`

The `Game` class is responsible for keeping track of the game. To do this, it holds a large array of moves, which is gradually filled up as moves are being played. An index variable can keep track of the position in the array, at which the next move will be placed (it should initially be 0). The `Game` class also holds an instance of the `Board` class, as it will be responsible for managing everything that happens on the board (think: nothing should happen on the `Board` that is not part of the `Game`). The following methods are suggested:

Constructor `Game(Board board)`
The constructor of the `Game` class is given a `Board`, which it will use for the game. The constructor also needs to initialise a `Move` array (ensure it's big enough), in which all moves will be stored. An index variable should be set to 0, indicating the position in the array at which the next move will be stored. The current player should be initialised to `WHITE`.

Method `Color getCurrentPlayer()`
Returns the `Color` of the player whose move it is. This will change after every move.

Method `Move getLastMove()`
Returns the last `Move` that was played, or null if no moves were played yet.

Method `void applyMove(Move move)`
Stores the given `Move` in the array and increments the index variable by 1. The move is applied on the board, and the `Color` of the current player should afterwards be updated accordingly.

Method `void unapplyMove()`
Retrieves the last move from the list of moves, and unapplies it on the board. The `Color` of the current player should be updated accordingly, and the index variable be decremented. Moves cannot be unapplied at the starting position, in which case the method should simply do nothing. After successfully unapplying a move, the output of `getLastMove` will change accordingly, no longer reporting the move that was just unapplied, but instead outputting the preceding one.

Method `boolean isFinished()`
Returns true iff the game is finished. A game is finished, if the last move was a winning move, that is, if a pawn was promoted to the last rank or if all the opponent's pawns were captured. A game is also finished, if the next player to move does not have any valid moves because all his pawns are stuck.

Method `Color getGameResult()`
If the game is finished, this method returns the `Color` of the winning player, or `NONE` in case of a stale-mate. If the game is not finished, the behaviour of this method is undefined (it does not need to return valid output).

Method `Move parseMove(String san)`
Takes a move in standard algebraic notation and returns the corresponding `Move` object. To do this, you will first need to translate the given string into board coordinates, and identify the pawn on the board that is being moved (there is a maximum of two places in which you need to look). This method returns `null` if the move is not valid. You will need to consider the last move to see whether the en passant rule can be played.

Class `Player`

The `Player` class interacts with both the `Board` and the `Game` class. Players may look at the board to evaluate their position and consider their options, but only make moves by using the `Game` class (logically, all moves should be part of the game). A player also knows his own `Color`, and sees his opponent (hint: which may help him to think from his opponents point of view...). The following methods are suggested:

Constructor `Player(Game game, Board board, Color color, boolean isComputerPlayer)`

The player will need to know about these to take part in the game.

Method `void setOpponent(Player opponent)`
Sets the player's opponent, which may or may not be useful to know. This method should only be called once, immediately after initialising both players.

Method `Color getColor()`
Returns the player's `Color`. This may only ever be `BLACK` or `WHITE`.

Method `boolean isComputerPlayer()`
Returns true iff this colour is played by a computer.

Method `Square [] getAllPawns()`
Returns an array of all `Squares` which are occupied by this player's pawns. The array should be of the correct length.

Method `Move [] getAllValidMoves()`
Returns an array of all valid moves that this player can make (assuming it is his turn). From the starting position, there should be exactly 14 moves.

Method `boolean isPassedPawn(Square square)`
Returns true iff this player has a *passed pawn* (see above) on the given square.

Method `void makeMove()`
If this player is played by a computer, this method should identify a move to play, and use the `applyMove()` method of the `Game` class to make that move. Initially, it is suficient to only play random moves (but we will later encourage you to make better-than-random moves). Random moves can be generated as follows: First, you will need to add `import java.util.Random;` to your class. Next, retrieve the array of all valid moves, say of length `n`. To generate a random number from 0..n-1, call `new Random().nextInt(n)`. Pick the move at this index, and apply it to the `Game`.

Class `PawnRace`

The `PawnRace` class is the application's entry point. It therefore exports a `main` method similar to those that you have written in your previous lab exercises. Unlike your previous exercises, however, it will use instances of the classes you have implemented up to this point in order to manage an interactive game of pawn racing. The program should be started with 4 parameters: The first two should record whether a player is played by a human player (P) or by a computer (C), the latter two should record the positions of the gaps, each for white and black, respectively. If the game shown in Figure 1 was played by a human player for white, against a computer for black, it should be called with `PawnRace P C H A`. This class only needs a `main` method:

Method `public static void main(String [] args)`
Starts an interactive pawn race game. Uses the given args to initialise the `Board`, the `Game`, and the two `Players`. You will find, that most of the required functionality is already present in these classes, such that you only need to handle input/output and the sequencing of the gameplay. This can be done as follows:

1. Display the current board
2. If the current player is a human player, prompt the player for a move (in short algebraic notation). If the move is not valid, prompt until a valid move is given, and apply that move to the board.
3. If the current player is a computer, ask him to make a move using the `makeMove()` method, and print which move the computer has played (in short algebraic notation).
4. If the game is now finished, print the result and exit. The result can be that either player has won, or that the game is a stale-mate. If the game is not finished, simply continue the game (you will find that the `Game` class will be indicating that it is now the other player's turn).

Testing and Committing

You should write suitable tests for all classes, other than the `PawnRace` class, that test pieces of functionality in isolation – e.g. whether a `Move` can be correctly applied and unapplied on a board, whether the `Color` of the current player is always updated correctly (for both applying and unapplying), etc. We encourage you to use frequent commits on GitLab with meaningful titles. If you have difficulties, please do ask for help.

Submission & Assessment

You are expected to submit this exercise via LabTS by the deadline. However, this exercise will be **unassessed and unmarked**. We hope you find it both entertaining and rewarding as a learning experience, so we recommend that you attempt to complete it. If you decide to participate in the tournament (see below), **you can continue working on your extension until the day of the competition**.

Extensions

As an extension, we encourage you to write a better-than-random artificial intelligence. You may not actually find this too difficult to get started – for example, in the `makeMove` method of the `Player` class, you could apply any of the valid moves, check if you now have a passed pawn, and unapply the move again if this is not the case. That way, you can identify any particularly good move, and if you are not sure which one to pick, you can still default to executing a random move. You may choose to add further utility functions to the class to give you a fuller analysis and better understanding of the current game – or for more complex strategies such as defensive moves, you may find that you wish to look further ahead. The possibilities are endless! How good can you make it?

One other extension might be to implement a `back` command in the prompt, allowing you to unapply the last move (or the last two moves, if you are playing against a computer), such that you can try out different strategies from the same position, without having to re-start the game from the beginning.

The Tournament

We will be holding a tournament in Week 11, which will be open to all First Year Students. Your “Artificial Intelligence” programs will be competing against each other in a knock-out tournament, and there will be prizes for the Top 4 participants. Whether you’d like to put your AI to the test, or whether you’d just like to try out your luck (by playing random moves), all First Years are encouraged to participate. Watch your emails for more details on this. You can signup to participate here: `https://forms.office.com/Pages/ResponsePage.aspx?id=B3WJK4zudUWDC0-CZ8PTB_jt4TLsB1pBm_F7cxSoxV9UOURJWUlMMDMzM1g4UOY4OFBOVFJNMO5aQy4u`. You are welcome to attend the event, even without participating.

The Tournament Rules

1. Rounds and Scoring - The tournament will consist of several rounds, in which players let their AIs compete with each other. Each round is played as best-of-five games, with only the winner advancing to the next round. In the (up to) five games played, each win counts as 1 point, while a draw/stalemate counts as 0.5 points for both players.

2. The colour of players in the first match of any pairing will be determined by a coin flip. Players swap colours after each game.

3. Players will sit down in their pairings and log in to adjacent lab machines, they will clone their gitlab repo, and they will be verbally communicating moves made by their AIs to each other. Your AI assumes your colour, and moves of the other player need to be entered manually (i.e. if you are playing white, run your program with a AI-plays-white human-plays-black).

4. In each game, the player whose AI plays black may determine where the both of the pawn gaps are (since white has the starting advantage). A player may not choose the same setup (or its mirroring) twice within the same round, although the other player may wish to choose the same setup when it is his/her turn to choose. Each player may choose at most one game per round in which the gaps are directly opposite each other.

5. If a program outputs an invalid move, or refuses to accept a valid move played by the other player, the game in progress will be counted as a loss.
6. All programs should output a move in reasonably less than 5 seconds. Otherwise, at the discretion of the referee, the game in progress may be counted as a loss.
7. **Any code from the standard Java libraries may be used, but any other external code (especially AI or chess libraries) is not allowed to be used**.
8. The code submitted by the Top 4 winners will be inspected after the end of the tournament. The last commit must be before the tournament begins (i.e. you cannot tweak your code as you go).
9. You may use up to 1 MB of pre-computed data.