Kotlin Chess - Pawn Race

COMP40009 - Computing Practical 1

4th – 8th December 2023

Aims

- To gain further experience in using appropriate control and data-structures including pattern matching, recursion, loops and HoFs.
- To practice writing classes and manipulating objects.
- To gain experience in object-oriented design.
- To write a simple game-playing program in Kotlin.

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.

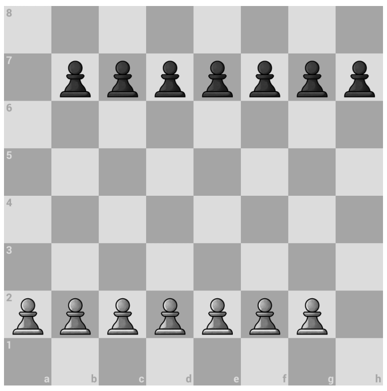

Figure 1: A possible starting position of the pawn race, in which the gaps are set to H (for white) and A (for black).

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-

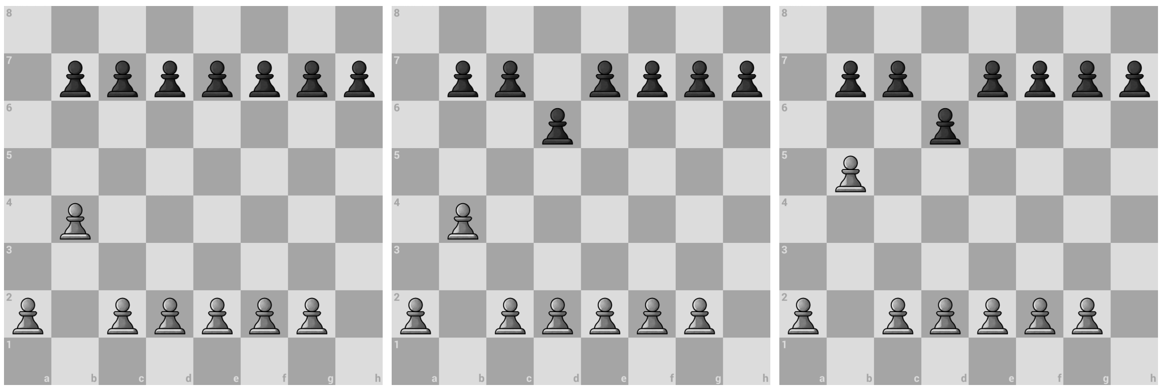

Figure 2: The moves b4 → d6 → b5 in order.

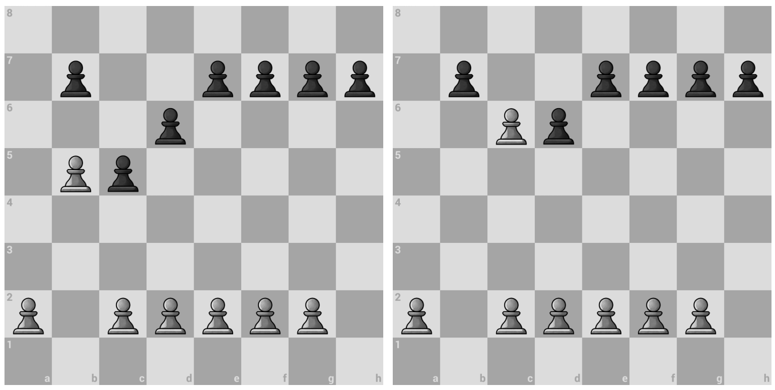

Figure 3: Continuing from Figure 2, if black now advances his C pawn by 2 steps, playing c5, white can capture this pawn en passant and play bxc6.

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

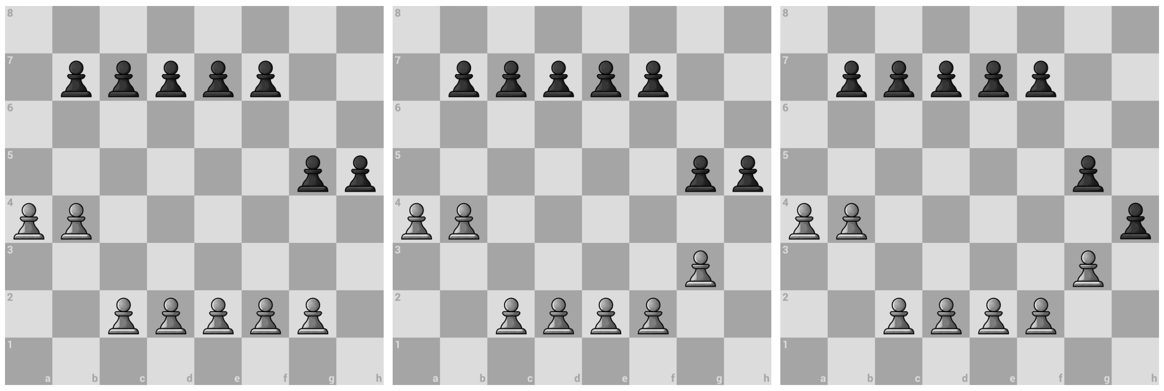

Figure 4: After a4 → h5 → b4 → g5, the above position is reached. White plays a defensive move g3, which turns out to be fatal, as black can establish a small *pawn chain* and easily win the game.

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).

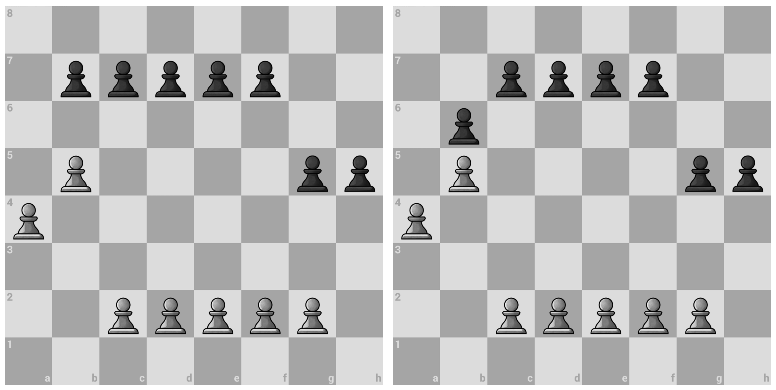

Figure 5: Continuing from Figure 4, white decides to attack instead and plays b5, which he intends to follow up by a5 and then a6. But black can simply counter this with b6, an effective blocking move, blocking the white B pawn, and preventing white's A pawn from moving forward.

Zugzwang

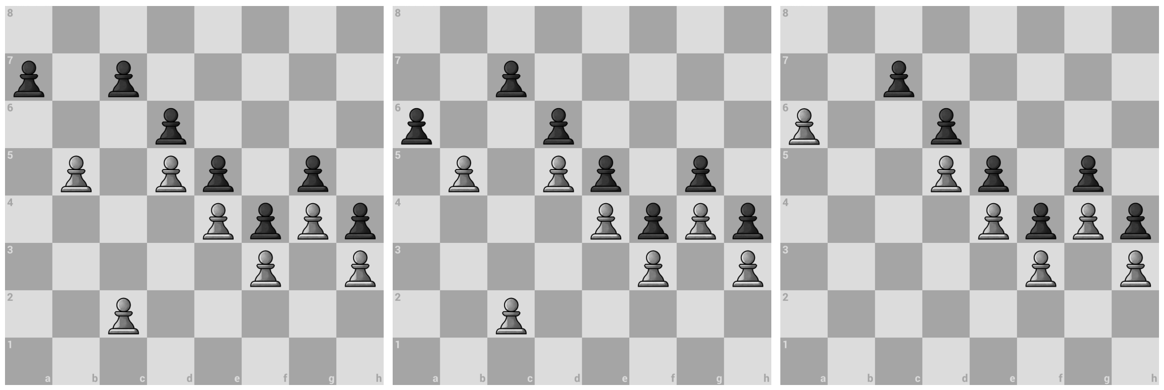

Figure 6: It is black's move – a classic example of *Zugzwang*, in which black is compelled to move, even though he may not want to. With only bad options to choose from, black decides to play a6, which white will capture and then win the game. Black would love to skip a few turns and let white run into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

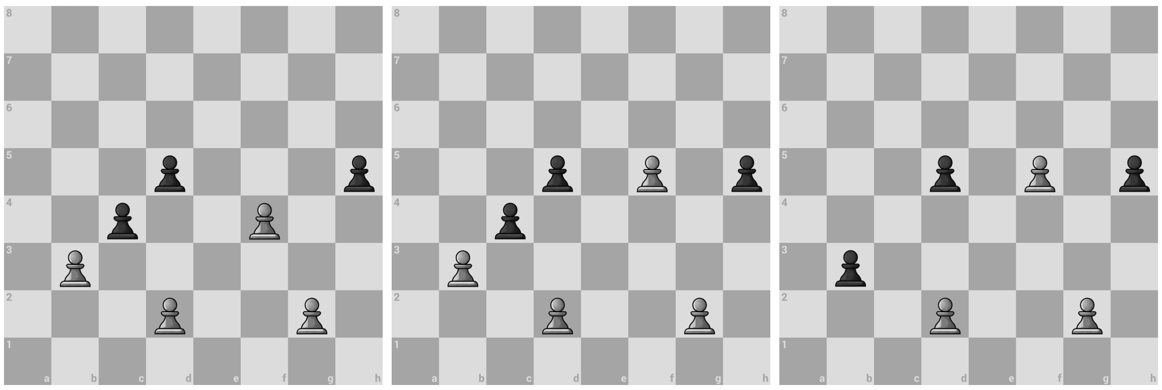

Figure 7: It is white's move. White optimistically decides to push his *passed pawn* on the F file forward, instead of playing a defensive move (bxc4). Yet, forgetting to defend immediately backfires on white, as black will now be able to create a passed pawn himself, and proceed to win the game.

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., 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 pretty much empty; it will provide a class with a single method which will be required for the autorunner to run correctly along with a main method for your convenience. This section outlines a *suggested* design: feel free to design it in any other way you wish, but you **must** ensure that the interactions in the `PawnRace` are exactly as specified in Class `PawnRace` and Correct IO part.

To make it easier for you to use the autorunner, please first clone the autorunner repo, then `cd` into that folder and clone your exercise skeleton repo inside it. This can be done like so (remembering to replace *`username`* with your username):

```
> git clone https://gitlab.doc.ic.ac.uk/kgk/kotlinpawnrace_autorunner.git
> cd kotlinpawnrace_autorunner
> git clone
  https://gitlab.doc.ic.ac.uk/lab2324_autumn/kotlinpawnrace_username.git
```

Enum `Piece`

The `Piece` enumerated type can be used to describe the colour of a player's pawns: a player's pawns are either `BLACK` or `WHITE`. The string representation of `BLACK` or `WHITE` can be set as their Unicode characters (or simply `B` or `W` if you prefer). Given a colour, it should be possible to get the opposite colour as well.

Class `File` and Class `Rank`

The coordinates of a `Piece` location can be recorded either as integer or character coordinates on the underlying chess board (see below); that is, for `File` and `Rank` either 0..7, 0..7 or a..h, 1..8. However, you must handle the reading and writing of positions in their letter-digit form: for instance, the strings A4 or a4 (file and rank).

Class `Position`

The `Position` class represents the a single location on the chess board (see below); the locations are recorded as `File` and `Rank`.

The following methods are suggested:

Constructor `(pos: String)`
The constructor takes this parameter and initialises a position with the given coordinates in the *short algebraic notation* (see above), for example A4 or a4.

Method `override fun toString(): String`
Returns the *short algebraic notation* of this position.

Enum `MoveType`

Enumerated type can be used to describe the type of a move (see below), which can be with the three values:

- `PEACEFUL` when no capture is involved
- `CAPTURE` when a regular capture happens
- `EN_PASSANT` when there is a capture following the En Passant rule

Class `Move`

The `Move` class captures a single move in the game and should be an immutable class. It should record both the starting and the target positions of the move in terms of their `Positions` (corresponding to long algebraic notation), the colour of the piece that made the move, and whether or not the move was a capture. This may appear to be more information than strictly needed, but this makes the application of the move to the board much more straightforward (and allows for undoing moves in a mutable, *imperative* approach). The `Move` class captures all these variables, which are all set in its constructor:

Constructor `(val piece: Piece, val from: Position, val to: Position, val type: MoveType)`
The constructor takes these parameters and creates a new `Move` with all relevant information about a move. You can also define this class as a data class instead.

Method `override fun toString(): String`
Returns the *short algebraic notation* for this move.

Class `Board`

The `Board` class keeps track of the current game. It primarily wraps an 8x8 grid which can be occupied by `Pieces`, that together make up the chess board. The `Board` class is responsible for the initial game setup, is immutable in the functional version[1], is mutable in the imperative version, and can apply moves. Note that, the students who wish to take the imperative path, need to consider how to undo `Move` possibly by keeping a list of moves to track that (in (`class Game`)). However, the functional path won't likely require such a mechanism if using a sequence of immutable boards to describe the flow of the game.

The following methods are suggested:

Constructor `(whiteGap: Rank, blackGap: Rank)`
The constructor takes these parameters and creates a board with an `8x8` grid (e.g., a list of lists of items), with all pawns setup, and with the pawn gaps in the correct places.

Method `fun pieceAt(pos: Position): Piece?`
Returns the *possible* `Piece` at the given position.

Method `fun positionsOf(piece: Piece): List<Position>`
Returns the `Positions` of a given `Piece` on the board - that is, all white or black pieces' positions.

Method `fun isValidMove(move: Move, lastMove: Move? = null): Boolean`
Checks whether the given `Move` is valid.You need to take into account: the move's type, the board's state and the position (e.g., other pieces' position or the piece's position on the board).

Method `fun move(m: Move): Board`
Applies the given move (according to its `MoveType`) and creates an updated board[2] accordingly.

Method `override fun toString(): String`
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
```

[1]During the game in the functional version, we create a sequence of immutable boards, that is, board, board', board'', etc.

[2]In the imperative path, we update a mutable board instead of creating an updated board and return `this`.

Class `Game`

The `Game` class holds an instance of the `Board` class, as it will be responsible for managing everything that happens on its board (think: nothing should happen on a `Board` that is not part of a `Game`). It also keeps track of the current player and the current and last move.

Note: in the *imperative version*, the `Game` class is also responsible for keeping track of all the moves. To do this, it holds a mutable stack of moves, whose size increases as moves are being played. The next move will be placed at the top of the stack, and it should initially be empty. However, this tracking of states is *not necessary* when writing a more *functional* solution. Accordingly, we propose having a different constructor for the imperative version and the functional version (not both of them); the `unapplyMove` method is only required for the imperative version.

The following methods are suggested:

Constructor `(val board: Board, val player: Player, val lastMove: Move? = null)`
The **functional version**'s constructor of the `Game` class is given a `Board`, which it will use for the game; it also gets the current player which should be `WHITE` at the beginning of the game. If the game is not at its start, the game should also keep its last `Move`[3].

Constructor `(val board: Board, val player: Player, val moves: MutableStack<Move> = listOf())`
The **imperative version**'s constructor of the `Game` class is given a `Board`, which it will use for the game. It also gets the current player which should be `WHITE` at the beginning of the game and initialise a mutable stack of `Move`s this stack should contain all moves that have been played throughout the game.

Method `fun applyMove(move: Move): Game`
Stores the given `Move` in the mutable stack for the imperative version, or stores the given `Move` in `lastMove` for the functional version. The move is applied on the board, and the `Player` should afterwards be updated to its opponent accordingly.

Method `fun unapplyMove()`
Only the imperative version requires this method (for the AI).

This method retrieves the last move from the list of moves (top of the mutable stack), unapplies it on the board, and remove it from the top of the stack. The `Player` should be updated accordingly. Moves cannot be unapplied at the starting position, in which case the method should simply do nothing.

Method `fun moves(piece: Piece): List<Move>`
Finds all possible valid moves. This can use two helpers: `moveForwardBy` and `moveDiagonalBy`, since a pawn can move either forward or diagonal.

Method `private fun moveForwardBy(pos: Position, step: Int, piece: Piece): Move?`
Helper function that creates a forward move if possible (target move is valid) otherwise returns `null`.

Method `private fun moveDiagonalBy(pos: Position, isLeft: Boolean, piece: Piece, type: MoveType): Move?`
Helper function that creates a diagonal move if possible (target move is valid) otherwise returns `null`.

[3]In fact, it only suffices to store the `lastMove` when that move was a double move: this is only needed to track whether en-passant was possible

Method `fun over(): Boolean`
Returns true when the game has finished. A game is over according to the rules laid out in Gameplay Section.

Method `fun winner(): Player?`
If the game is finished, this method returns the winning `Player` or `null` 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 `fun parseMove(san: String): Move?`
Takes a move in standard algebraic notation and returns the corresponding `Move` object. This method should first parse the 'san' move and identify the relevant piece on the board. It should return `null` if the move is not valid. The last move can be used 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 its own `Pieces` and hence its colour, and sees its opponent (this may help them to think from their opponents point of view...).

The following methods are suggested:

Constructor `(val piece: Piece, var opponent: Player? = null)`
The player will need to know about these parameters to take part in the game.

Method `fun getAllPawns(): List<Position>`
Returns a list of all `Positions` which are occupied by this player's pawns.

Method `fun getAllValidMoves() List<Move>`
Returns a list of all valid moves that this player can make (assuming it is its turn). From the starting position, there should be exactly 14 moves.

Method `fun isPassedPawn(pos: Position): Boolean`
Returns true iff this player has a *passed pawn* (see above) on the given position.

Method `fun makeMove(game: Game): Move?`
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 sufficient to only play random moves (but we will later encourage you to make better-than-random moves). To generate random numbers to select a move, you can use of `kotlin.random.Random` and get a random number in range $0, n-1$ inclusive with `(0 until n).random()`.

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 to manage an interactive game of pawn racing.

The `main` method will be provided with a single argument on the command line, which is the colour that the AI kPlayer should assume. For example: `java -jar pawnrace.jar W` will start a game with you controlling the Black pieces from the terminal, and your AI controlling the White pieces. This argument is forwarded directly to the `playGame` method, where the logic of the flow of the game should be implemented (see Correct IO section for an explanation of the exact order of the input/output interactions).

Method `public fun main(args: Array<String>)`
Starts an interactive pawn race game. You should avoid editing this method.

Method `fun playGame(colour: Char, output: PrintWriter, input: BufferedReader)`
You should edit this part of the skeleton file `PawnRace.kt`. Uses colour to initialise the two `Players`: a player and its opponent, the `Board`, and the `Game`. The argument `colour` indicates your AI's colour, which is the random or AI player you wrote. The opponent can be a human player (if using `main`) or another computer player (if using the autorunner). Use `output` and `input` to print results and read input for the opponent player, respectively.

You will find, that most of the required functionality is already present in the classes we have suggested above, such that you only need to handle input/output and the sequencing of the gameplay via the input and output objects of `playGame`.

Correct IO

The presence of an autorunner means that you must be very precise with how you communicate your moves to your opponent. This is always done using the `output` and `input` streams provided to the `playGame` method. If you use the regular Kotlin `println` this will not be sent to the autorunner: you may use this to print out prompts to a human, or debug information.

The streams we provide are used with `output.println(...)` and `input.readLine()`. The correct sequence of events in the `playGame` method is as follows.

1. If you are Black (that is to say, `colour == 'B'`), your first step is to print your AI's chosen gaps: this should be done in a string form, e.g. `output.println("aa")`, with the first character being the white's gap and the second the black's gap. Note: if you don't want to choose your gaps, you may always send the same gaps, the autorunner will automatically pick new gaps if the ones you send have already been used.

2. Regardless of your colour, the autorunner (or you on the terminal) will then send back gaps to you **these are the ones you should use, regardless of what you picked before even if you are Black**. These will be sent as a single string with white's gap first and then black's (you may collect them with `input.readLine()`). The reason they are echo'd back to you is if the autorunner determined you'd already used them before and is handing back new ones.

3. With the gaps negotiated, the game can commence. You may wish to print the board to the terminal for the human's sake, but the autorunner will also do this for you if playing against another player or the autorunner's own random AI. If you do wish to print the board, do so using `println()` and *not* `output.println()`.

4. White moves first and then black. If you are Black, then you will need to wait and receive a move from White with `input.readLine()` which will be sent in the algebraic notation presented previously in the spec. You may then update your own board and make your own move, which you will send using `output.println(move)`. You should ensure the game isn't over after each move. This step will be performed in a loop until one player wins the game. You may wish to report which player won the game at the end. The autorunner will also display this information.

Compiling your Code

To create an executable Kotlin JARs via terminal, you can compile your code that way:

```
kotlinc . -include-runtime -d ../pawnrace.jar
```

assuming you compile your code from `src` folder. You can also compile via intellij artifacts if you wish.

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 `Piece` 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.

If you wish to test your `playGame` functionality, you may do so by running the autorunner, or by calling the main method and playing it yourself. To use the autorunner, you must compile your code into a **runnable** jar file called `pawnrace.jar`. This should live in the root of your repository. Just outside of your repository is where the `pawnrace-autorunner.jar` lives[4]. Then you may execute it with one of:

```
java -jar -ea pawnrace-autorunner.jar 0 kotlinpawnrace_YOURUSERNAME
java -jar -ea pawnrace-autorunner.jar 1 kotlinpawnrace_YOURUSERNAME
java -jar -ea pawnrace-autorunner.jar 5 kotlinpawnrace_YOU kotlinpawnrace_OPPONENT
```

Each of these does something slightly different: the first will run your solution for a single game against the autorunner's own random AI. This is done by calling with `0`. The second plays your AI against a copy of itself for `1` round, but this could be changed to `10` rounds etc. The third will play your AI against another opponents AI (which should be a folder with their username with a `pawnrace.jar`) inside, in this case for `5` rounds. In all cases the autorunner will decide what colour you will play. If you are curious about how this works, ask a TA to point you to the relevant person (Jamie).

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!

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 (or indeed, revert to the previous board) 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?

[4]assuming you followed the steps to clone *both* repositories. If you haven't, then just ensure that the `pawnrace-autorunner.jar` can be found just outside your exercise repo!

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. If you would like to put your AI to the test, you can signup to participate here: `https://forms.office.com/e/wcD3qJG8T4` (*registration closes on 08/12/2023 19:00*). You are welcome to attend the event, even without participating, to cheer for your classmates and their AIs!

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 2 points, while a draw/stalemate counts as 1 point for both players.
2. The colour of players in the first match of any pairing will be determined randomly. Players swap colours after each game.
3. The autorunner will be used to play off two students AIs against each other. and will be done by us: the specific machine we use to play them off against each other will likely be a fast machine with many CPU cores (possibly up to 20).
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 less than 5 seconds. If it takes longer to return a move, the automarker will forfeit the game on that player’s behalf.
7. The top-ranked 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).
8. **Any code from the Kotlin standard library (including the standard Java libraries) may be used, but any other external code (especially AI or chess libraries) is not allowed to be used**.
9. You will participate with the GitLab commit that you have submitted via LabTS to CATe before the deadline (i.e., you cannot tweak your code after the exercise deadline). Any late submissions **cannot** participate in this tournament.
10. The code submitted by the Top 4 winners will be inspected after the end of the tournament and you may be invited to explain how it works.
11. You may use up to 1 MB of pre-computed data.

Enforcing the Rules

The previous section outlined the tournament rules. Well, in fact, the autorunner has the power to enforce these rules and more. Specifically:

1. If you make use of multi-threading in your advanced AI, then you are limited to at most `Runtime.getRuntime().availableProcessors()/2 - 1` threads. Any attempt to make more than this number of *simultaneously running* threads will result in a forfeit.

   The reason for this restriction is to keep the balance of the computational resources fair: we don't want you to starve out your competition for control of the CPU!

2. You may use 1MB of pre-computed data: this should come in the form of files. The autorunner will track every file you open, and you are allowed to open as many files as you wish, so long as they sum to less than 1MB. You may read from the same file many times without increasing your quota, but a file of size 2MB that you read 10 bytes from will cause you to exceed your quota and forfeit the game. You should keep those files alongside your jar (or even inside them).

3. You must respond to *any and all* interactions from the autorunner within 5 seconds.

4. You may not open any network connections or sockets

5. You may not exit the program from within your code

6. You may not execute command line programs from within your code

7. You may not write to any files, or attempt to delete any files

8. You may not attempt to meddle with the other player's thread pool

9. You may not try and change the security manager that has been imposed by the autorunner to make all of this magic possible.