A plan for coursework I: chess video game

Typos/errors? Email markel.vigo@manchester.ac.uk and I'll get them fixed.

☝ This is our recommended plan which aligns with the topics taught each week. Also most of the load is set on week 1 so there is enough time to complete the coursework before the deadline. If you prefer to explore the classes and implement them following some other strategy or order feel free to do it.

Table of Contents

Week 1 onwards plan

1.1 Get familiarised with the development environment

- Make sure you read the lab manual.
- As your read the manual run the scripts and replicate the screenshots of the manual.
- Explore the folder structure and check the java classes (those in the `src > main` folder: **you should modify these**), tests (those under the `src > test` folder: **you must not modify them**), and compiled classes and tests are located (check the `bin` folder). The execution of the scripts moves the binary `.class` files to the `bin` folder.
- Open the running (`run.sh`) and testing (`run-tests.sh`) scripts in your editor and try to understand what each command does.
- Explore the output of testing and try to understand how you can use the output to inform the next steps.

1.2 Read the code and try to understand what it does

Open the source files in your preferred editor but bear in mind that use of IDEs is discouraged and not recommended.

`Game`. The logic of the videogame resides in the `Game` class: this is the class that manages the turns, prompts players to make a move, checks whether the commands are valid and uses remaining classes to make sure that moves are valid, and update the board accordingly.

`Board` controls the logic and visualisation of the chessboard.

1. `initialiseBoard()` creates a matrix of 64 `Square` objects.
2. `setPiece(int, int, Piece)`: puts a piece on a specific `Square` of the board as indicated by the `i` and `j` coordinates.
3. `initialisePieces()` creates instances of classes and places them on the board using the `setPiece` method.
4. `movePiece(int, int, int, int, Piece): boolean`: The first two parameters indicate the `i` and `j` coordinates of the origin of the movement of a `Piece` object, while the next ones are the `i`, `j` coordinates of the destination. Note that moving a piece requires taking a piece from a square and setting it in another one. The return value should only be true if the move has resulted in the king being captured (i.e. the game is over). In all other circumstances (no piece captured, any other piece captured), the return value should be false. This method should be called once the game logic decides that the intended move is valid.
5. `getPiece(int, int): Piece`: returns a piece on a specific coordinate or `Square`.
6. `hasPiece(int, int): boolean`: returns true if there is a piece on a specific coordinate (or `Square`), false otherwise.
7. `printBoard()` displays on the terminal the current state of the board.

`Square` is what the `Board` is made of. It's the container of pieces. The methods in this class will be typically be called by `movePiece`, `setPiece`, `getPiece`, and `hasPiece` of the `Board` class, which are incidentally named in a similar way as those in `Square`.

`PieceColour` is an enumeration class to constrain and define the colour chess pieces have.

`CheckInput` is a class containing one static method that checks whether the commands are correctly formatted. To be valid the input string must be a number 1-8 and a character a-h in this same order. It will be important from week 3 onwards.

`Piece` is an abstract class with an abstract method `isLegitMove(int, int, int, int): boolean` inherited by all the pieces. The boolean value indicates whether the move is valid or not, where the first two parameters indicate the `i` and `j` coordinates of the origin of the movement and the last two are the destination. It will be important from week 4 onwards.

1.3 Implement the movement logic of the pieces

Go to the `src > main` folder. You can start with the implementation of the body of all the `boolean isLegitMove(int, int, int, int)` methods in the `Pawn`, `Rook`, `Bishop`, `Knight`, `King`, `Queen` classes. The first two parameters indicate the `i` and `j` coordinates of the origin of a movement of the aforementioned pieces, while the next ones are the `i`, `j` coordinates of the destination. If the move is valid, the method should return `true`, otherwise `false`. Using these methods from `Board` will be useful:

- `getPiece(int, int): Piece`: returns a piece on a specific coordinate or `Square`.
- `hasPiece(int, int): boolean`: returns true if there is a piece on a specific coordinate (or `Square`), false otherwise.

These two methods are `static` (we will learn what this means in week 2). In practical terms, it means that you have to call them in this way: `Board.getpiece(i,j)` and `Board.hasPiece(i,j)`.

Note that while commands are in the `1` to `8` range, the indices of Java arrays start at `0`. So the commands are internally "translated" in order to be able to run the tests which use a `0` to `7` range. Check lines 36, 37 and 68, 69 of `Game.java`.

This method should implement the following rules:

- The movement rules of the corresponding piece according to the basic moves.
- Some pieces can jump over other pieces (e.g. the knight) while others can't. In these cases `isLegitMove` should check whether there is a piece between the origin and destination. You can use the `boolean hasPiece(int, int)` method from the `Board` class to do so.
- You should also make sure that you are not capturing a piece of your own colour. Use the methods from `Board` combined with those in each piece to know the colour of the captured piece.
- You must work under the assumption that the coordinates inputed will be valid. This is something you will implement later on week 3. Consequently, the body of `isLegitMove` doesn't have to implement this.

Try to run the tests frequently to check whether your updates are making any effect on the number of tests passed/failed.

Week 2 onwards plan

While you should keep working on the `isLegitMove` method so the associated tests pass, from this week on you will be able to address most of the `SpecsTest` tests, i.e. those tests related to meeting the specifications of the UML diagram. In each of the piece class there are 3 tests of this type that fail right now.

You should check the visibility set by the UML diagram to the classes, methods and attributes and make sure all the classes meet this specification. Run the tests to check how the `SpecsTest` tests that failed are passing when you implement your changes.

Week 3 onwards plan

This week's contents are about handling input/output and manipulating strings. Consequently this week's work should be focused on making sure that the coordinates typed by the players in the command line follow the expected format.

Remember what format should be followed: each movement command must contain two characters that convey a specific coordinate of the chess board. The first one is a number from `1` to `8` (corresponding to a `i` coordinate in a matrix), while the second is a character from `a` to `h` (corresponding to a `j` coordinate in a matrix). Note that while commands are in the `1` to `8` range, the indices of Java arrays start at `0`. So the commands are internally "translated" in order to be able to run the tests which use a `0` to `7` range.

Work on the `checkCoordinateValidity` method of the `CheckInput` class to make sure that the above constraints are respected. The method should return `true` if the coordinates follow a valid format and `false` otherwise. The tests in the `CheckInputTest` class will help you to check your progress towards making sure the format is correct.

Week 4 onwards plan

You probably noticed that while pieces extend the `Piece` class, each piece has two attributes that do not follow the specifications of the game: the `PieceColour colour` and `String symbol` attributes.

Remove these attributes in all the subclasses of `Piece` and make use of the inherited methods instead in the constructor of each class.