

# Coursework I manual: chess video game

---

**Typos/errors?** Email [markel.vigo@manchester.ac.uk](mailto:markel.vigo@manchester.ac.uk) and I'll get them fixed.


---

## Table of Contents

- [1. Instructions](#)
- [2. Rules to implement](#)
- [3. Running and testing the game](#)
- [4. Marking](#)

## Submission

The submission deadline is March 12 (Friday of week 5) at 6PM. The `tag` you have to use is `comp16412-coursework1`. Check the [coursework processes](#) established by the department to do so.

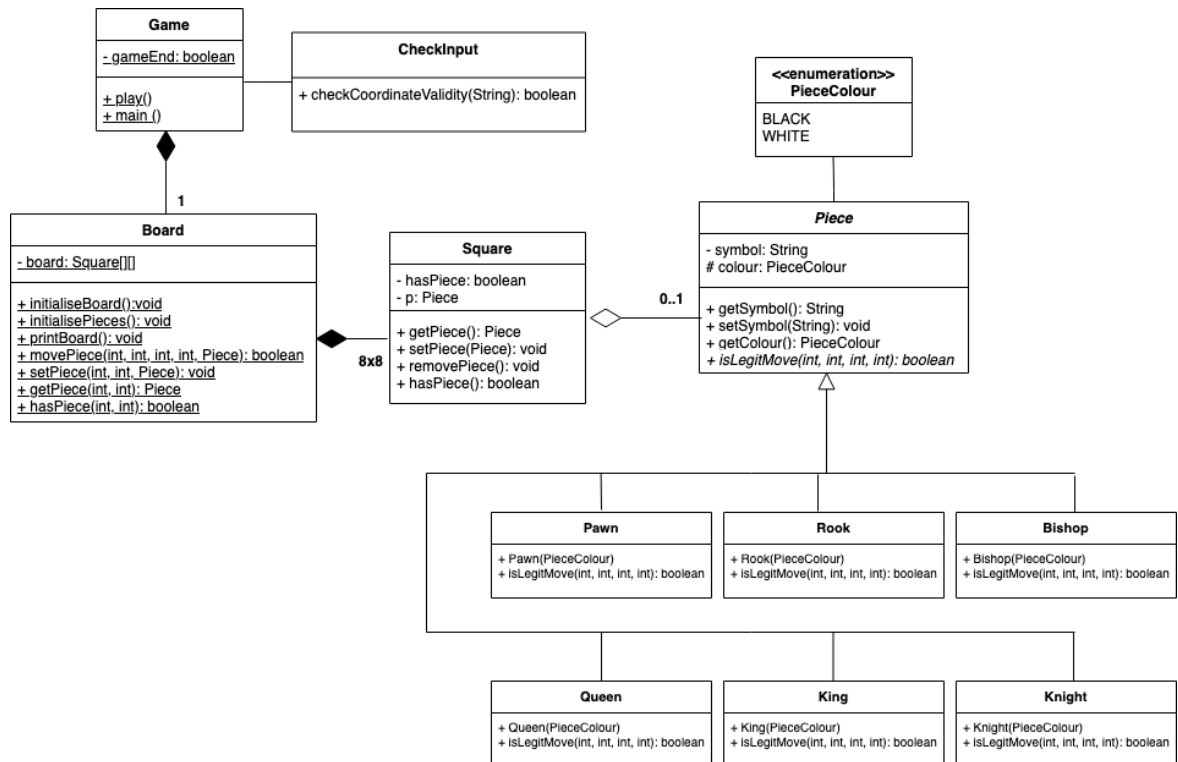
 Failure to use the right tag will delay your marking and will incur in penalisations.

---

## 1. Instructions

In this coursework we'll be asking you to implement a console-based chess game. The game **must** meet the provided specification, so you should take time to read all instructions carefully.

1. The structure of the following UML class diagram informs the design of the application. Your implementation must meet the structure provided in the UML diagram.



Check [the UML reference sheet](#) to help you understand the syntax of the above class diagramme.

- Most classes and methods are self-explanatory but to remove any ambiguity we define the behaviour of following ones:

**Game** is the driver class controlling the game logic including:

- Initialising the board, the pieces and printing it using the **Board** class in the `main` method. The logic of the game is handled in the `play()` method.
- Input handling and making sure that input values are valid (through the **CheckInput** class);
- Managing turns (ie which pieces move when);
- Making sure there is a piece on the origin coordinates;
- If the coordinates are valid as indicated by (2) and (4), making sure that the intended movement is legitimate (through **Piece**).
- Making movements and printing the board after every move (through **Board**);
- Managing the end of the game and who the winner is.

**Board** controls the logic and visualisation of the chessboard where `initialiseBoard()` creates a matrix of 64 **Square** objects. **This class is partially implemented so the game can be run.** Some methods need their body written, some others are fine as they are and others may need to be extended. The methods are:

- `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.

2. `setPiece(int, int, Piece): void` : puts a piece on a specific `Square` of the board as indicated by the `i` and `j` coordinates.
3. `getPiece(int, int): Piece` : returns a piece on a specific coordinate or `Square`.
4. `hasPiece(int, int): boolean` : returns true if there is a piece on a specific coordinate (or `Square`), false otherwise.

`Square` is what the `Board` is made of. It's the container of pieces. **This class is partially implemented so the game can be run and give you some guidance.** 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`.

`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.

The `CheckInputTest` class provides a method `checkCoordinateValidity` that checks whether the input coordinate is valid. To be valid the input string must be a number 1-8 and a character a-h in this same order.

## 2. Rules of the game to be implemented

### 2.1 The rules

This chess game, will implement a subset of the rules of chess including:

- The initial setup as defined in the [Rules of Chess wikipedia page](#).
- All the [basic moves](#).
- The game ends when a king is captured or one player resigns.
- Note that the notation for the coordinates is different in our game. The original game is *a* to *h* on the X axis (left to right) and 8 to 1 on the Y axis (top to bottom), while in our case the latter should be 1 to 8 from the top to the bottom.

## 2.2 What not to do

Do **NOT** implement:

- *En passant* (pawn)
- Pawn promotion (pawn)
- Castling (king and rook)
- Check detection
- Draw detection
- Time control

## 2.3 Other considerations:

- Squares do not have to be coloured.
- Pieces are coloured using the black and white [unicode chess symbols](#). See for instance: ♔ and ♚.
- Make sure to configure your terminal to use a monospaced font. This gives the same width to whitespace and pieces. Otherwise, the board and the pieces may look odd. This is what the console output should look like. Oddly, in a terminal with a black background the black pieces look white (don't get confused with this).

```
bin — java chess/Game — 92x37
[cspc375:bin markel$ java chess/Game

  a b c d e f g h
  -----
1 ♔|♚|♙|♜|♞|♛|♡|♠| 1
2 ♜|♚|♙|♜|♞|♛|♡|♠| 2
3 | | | | | | | | 3
4 | | | | | | | | 4
5 | | | | | | | | 5
6 | | | | | | | | 6
7 ♜|♚|♙|♜|♞|♛|♡|♠| 7
8 ♔|♚|♙|♜|♞|♛|♡|♠| 8
  -----
  a b c d e f g h

----- Whites move -----
> Enter origin:
7a
> Enter destination:
5a

  a b c d e f g h
  -----
1 ♔|♚|♙|♜|♞|♛|♡|♠| 1
2 ♜|♚|♙|♜|♞|♛|♡|♠| 2
3 | | | | | | | | 3
4 | | | | | | | | 4
5 ♜| | | | | | | | 5
6 | | | | | | | | 6
7 | ♜|♚|♙|♜|♞|♛|♡|♠| 7
8 ♔|♚|♙|♜|♞|♛|♡|♠| 8
  -----
  a b c d e f g h

----- Blacks move -----
> Enter origin:

```

## 2.4 Commands:

- Movement command: As you see in the above screenshot, 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 should be "translated" in order to be able to run the tests which use a 0 to 7 range.
- Ending the game: type `END`.

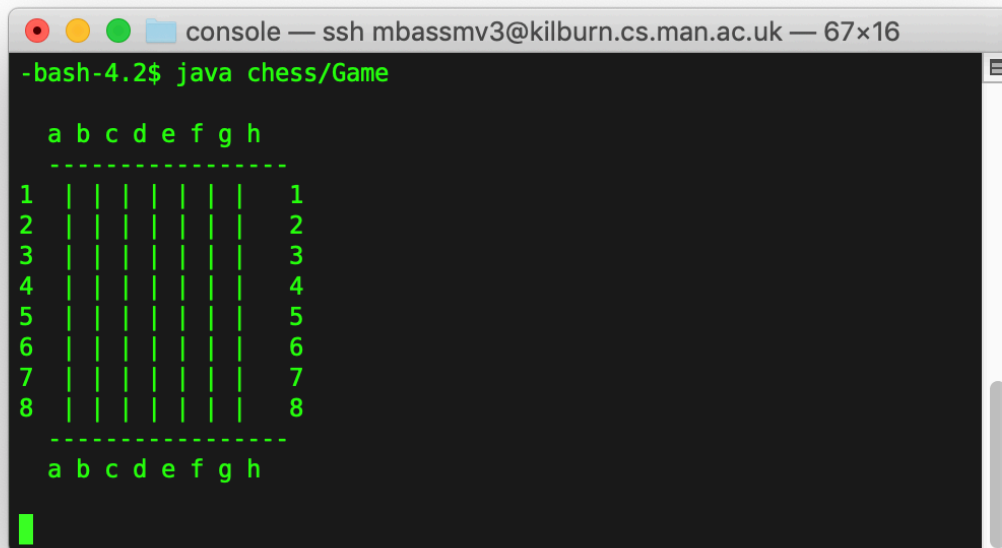
## 3. Running and testing the game

### 3.1 Running

You should be able to run the current codebase by

1. In the `coursework1` folder compile the given classes: `javac chess/*.java`.
2. Run the game: `java chess/Game`

You should see an empty chessboard as a result:



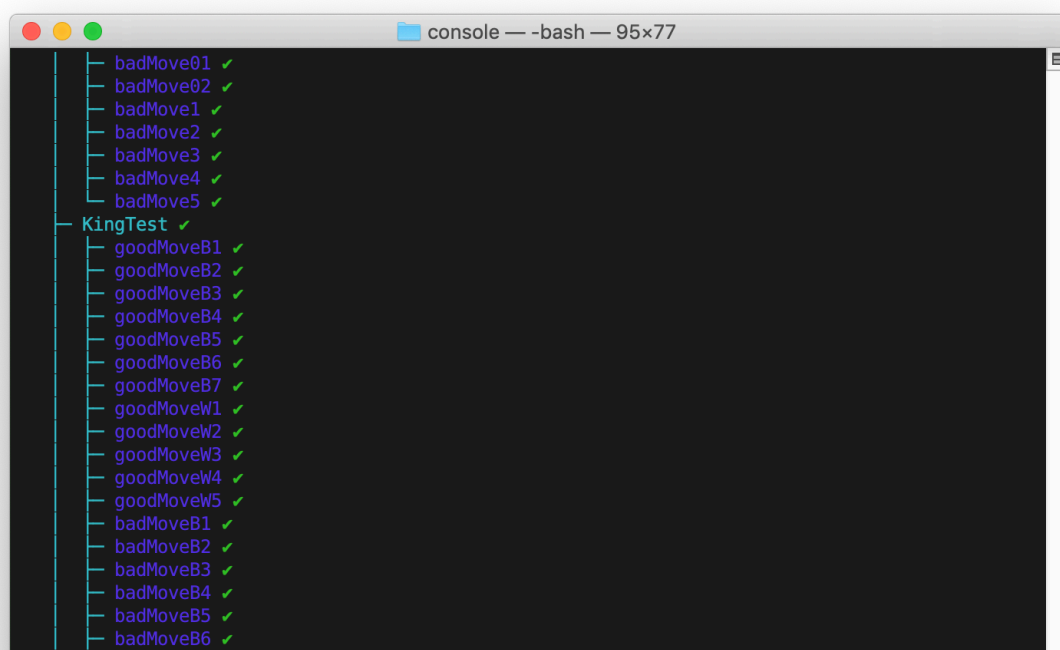
```
console — ssh mbassmv3@kilburn.cs.man.ac.uk — 67x16
-bash-4.2$ java chess/Game

  a b c d e f g h
-----
1 | | | | | | | 1
2 | | | | | | | 2
3 | | | | | | | 3
4 | | | | | | | 4
5 | | | | | | | 5
6 | | | | | | | 6
7 | | | | | | | 7
8 | | | | | | | 8
-----
  a b c d e f g h
```

### 3.2 Testing

With the given codebase the tests cannot be compiled. As you make progress you should be able to compile and run the tests. We provide a script that compiles the tests and runs them:

`./run-tests-debug-dev.sh` which shows a long list of tests with a green tick next to them if they pass and a red cross otherwise:



```
console — -bash — 95x77
- badMove01 ✓
- badMove02 ✓
- badMove1 ✓
- badMove2 ✓
- badMove3 ✓
- badMove4 ✓
- badMove5 ✓
- KingTest ✓
  - goodMoveB1 ✓
  - goodMoveB2 ✓
  - goodMoveB3 ✓
  - goodMoveB4 ✓
  - goodMoveB5 ✓
  - goodMoveB6 ✓
  - goodMoveB7 ✓
  - goodMoveW1 ✓
  - goodMoveW2 ✓
  - goodMoveW3 ✓
  - goodMoveW4 ✓
  - goodMoveW5 ✓
  - badMoveB1 ✓
  - badMoveB2 ✓
  - badMoveB3 ✓
  - badMoveB4 ✓
  - badMoveB5 ✓
  - badMoveB6 ✓
  - badMoveB7 ✓
```

```

- badMoveW1 ✓
- badMoveW2 ✓
- badMoveW3 ✓
- badMoveW4 ✓
- badMoveW5 ✓
- badMoveW6 ✓
- badMoveW7 ✓
- BishopTest ✓
  - goodMoveNE ✓
  - goodMoveNW ✓
  - goodMoveSE ✓
  - goodMoveSW ✓
  - goodMove1 ✓
  - goodMove2 ✓
  - goodMove3 ✓
  - goodMove4 ✓
  - badMove10 ✓
  - badMove11 ✓
  - badMove12 ✓
  - badMove13 ✓
  - badMove14 ✓
  - badMoveNE ✓
  - badMoveSE ✓
  - badMoveSW ✓
  - badMoveW0 ✓
  - badMoveW1 ✓
  - badMove2 ✓
  - badMove3 ✓
  - badMove4 ✓
  - badMove5 ✓
  - badMove6 ✓
  - badMove7 ✓
  - badMove8 ✓
  - badMove9 ✓
  - badMoveB ✓

Test run finished after 424 ms
[ 9 containers found ]
[ 0 containers skipped ]
[ 9 containers started ]
[ 0 containers aborted ]
[ 9 containers successful ]
[ 0 containers failed ]
[ 215 tests found ]
[ 0 tests skipped ]
[ 215 tests started ]
[ 0 tests aborted ]
[ 215 tests successful ]
[ 0 tests failed ]

cspc375:console markel$

```

## 4. Marking

The marks of this piece of coursework account for 40% of the coursework marks. Marking will have conducted in two stages: (1) offline: TAs will run automated tests face-to-face in the labs (40% of the marks). (2) online: TAs will ask a set of questions about your codebase (60% of the marks).

### 4.1 Automated testing

A test suite will be run against your code. The outcome of the testing will account for 40% of the marks (40 marks). They mostly check whether movements of pieces are legal or not. 122 of these tests have been provided to you, others will be new (expect around 240 tests more or less) but will match the provided specification. The coverage of the number of tests

passed will be your mark for automated testing. If 200 tests pass out of, say 215, you will get 37 (200/215) of the 40 marks. If tests cannot be run, you will get 0/40 marks.

To ensure that the tests are successful, **you must follow the provided UML diagram and specification.**

## 4.2 Non-automated assessment

This accounts for 60% of the marks (60 marks). The questions will be about:

- Your understanding of the UML representation.
- The use of `static` methods.
- How you employed object-oriented constructs such as access level modifiers, encapsulation, inheritance, abstraction.
- How you programmed the game logic including turns management and determining whether the resulting commands are legitimate.
- How pieces follow the rules and move accordingly.

## Marking Disputes / Queries

We appreciate that some students may have questions about their marks. Any questions about your marks or feedback **must** be made within one week of your marking session. Queries received after this time will not be accommodated.