THE UNIVERSITY OF SYDNEY

# INFO1113 / COMP9003 Assignment

Due: 14 May 2023, 11:59PM AEST

*This assignment is worth 18% of your final grade.*

## Task Description

In this assignment, you will create a game in the Java programming language using the Processing library for graphics and gradle as a dependency manager. In the game, the player must be able to move chess pieces on a chess board in order to play against an AI, ultimately capturing the opponent's pieces and executing a strategy to checkmate the opponent's king.

You have been given the task of developing a prototype of the game. A full description of gameplay mechanics and entities can be found below. An artist has created a simple demonstration of the game and has posted it on your online forum (Ed). You can also play a similar game here.

You are encouraged to ask questions on Ed under the assignments category if you are unsure of the specification – but staff members will not be able to do any coding or debugging in this assignment for you. As with any assignment, make sure that your work is your own, and do not share your code or solutions with other students.

## Working on your assignment

You have been given a scaffold which will help you get started with this assignment. You can download the scaffold onto your own computer and invoke gradle build to compile and resolve dependencies. You will be using the Processing library within your project to allow you to create a window and draw graphics. You can access the documentation from here.

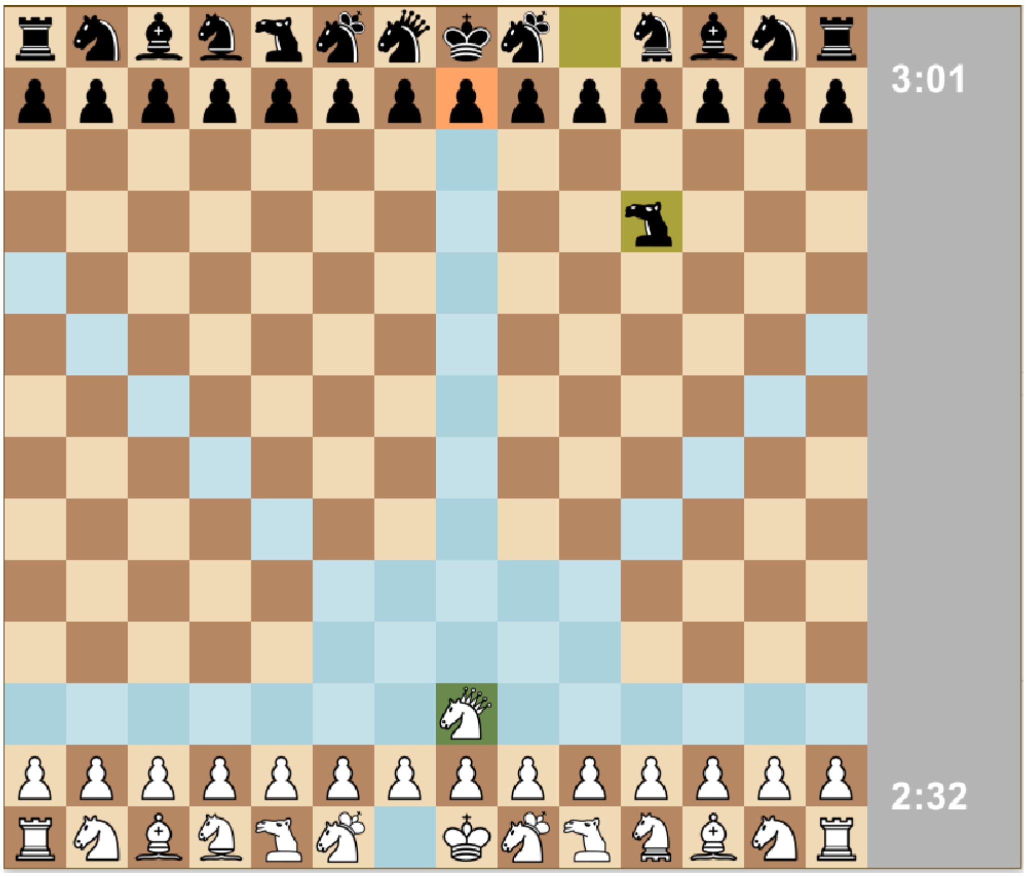
## Gameplay

The game contains a number of entities that will need to be implemented within your application.

### Board

The board consists of a grid of tiles 14x14. Each tile is 48x48 pixels, so the total is 672x672 pixels. However, there are 120 pixels on the right sidebar reserved for information such as timers showing the number of minutes and seconds remaining on each player's clock, and warnings or other messages for the user. The window size is therefore 672x792.

The board is arranged in a checkerboard pattern as below with alternating black and white tiles. These are fixed and do not change. Pieces sit atop these tiles, and the tiles may change colour shade to indicate highlights for particular reasons.

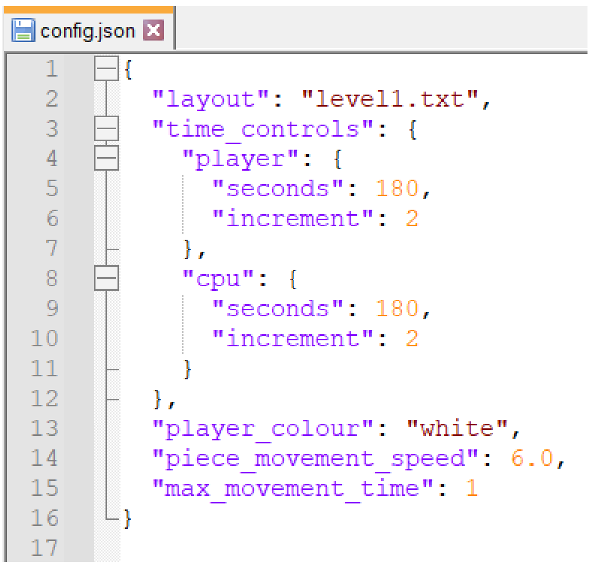


There are 4 main types of highlights:

- Blue – the player clicked on a piece, and it is able to move to this square.
- Light red – the currently selected piece can move to this square, capturing the current piece there
- Green – the player's currently selected piece
- Yellow – the last piece to move, and the square it came from
- Dark red – the king on this square is currently in check, or checkmate has occurred (pieces that contribute to the checkmate are highlighted in light red)

The piece layout is defined in a file named in the "layout" attribute of the JSON configuration file described below.

## Config



```
{
  "layout": "level1.txt",
  "time_controls": {
    "player": {
      "seconds": 180,
      "increment": 2
    },
    "cpu": {
      "seconds": 180,
      "increment": 2
    }
  },
  "player_colour": "white",
  "piece_movement_speed": 6.0,
  "max_movement_time": 1
}
```

The config file is in located in config.json in the root directory of the project. Use the simple json library to read it. Sample config and level files are provided in the scaffold.

The config sample as shown to the left, contains the name of the layout file. This is also located in the root directory of the project. The layout file will contain a grid of text characters, where each character represents the piece that should be in that cell. Uppercase characters are for black, and lowercase are for white. See the table below.

| Layout file | | | | | See page 5 for images of movement |
|---|---|---|---|---|---|
| Black | White | Chess piece | Value | Sprites | Movement |
| P | p | Pawn | 1 | ♟ ♙ | One space forward. Captures diagonally only. If blocked, cannot move. |
| R | r | Rook | 5.25 | ♜ ♖ | Horizontally and vertically. |
| N | n | Knight | 2 | ♞ ♘ | 2 squares vertical, 1 horizontal, or vice versa |
| B | b | Bishop | 3.625 | ♝ ♗ | Diagonally in any direction. |
| H | h | Archbishop | 7.5 | ♞ ♘ | Like Knight + Bishop |
| C | c | Camel | 2 | ♞ ♘ | 3 squares vertical, 1 horizontal, or vice versa |
| G | g | General/Guard | 5 | ♞ ♘ | Like Knight + King |
| A | a | Amazon | 12 | ♞ ♘ | Like Knight + Bishop + Rook |
| K | k | King | ∞ | ♚ ♔ | 1 space in any direction. Cannot move into danger. |
| E | e | Chancellor | 8.5 | ♞ ♘ | Like Knight + Rook |
| Q | q | Queen | 9.5 | ♛ ♕ | Like Bishop + Rook |

Empty spaces are empty tiles. All maps used for marking will be valid, but you should write your own tests for invalid maps and handle them as you see fit.

The "time_controls" section contains the amount of time to be given to player 1 and player 2 (player 2 is the computer - cpu). Seconds it the total time they start with, which is consumed while they are thinking about a move. The "increment" is a number in seconds added to their remaining time once they make a move.

The "player_colour" property denotes the colour of the pieces of player 1 (the human player). It should either have the value "white" or "black". If player 1 is white, then player 2 is black. If player 1 is black, then player 2 is white. Whoever is white has the first move, as in regular chess.

## Movement

The “piece_movement_speed” property in the config denotes how fast in pixels per frame a piece move should occur. This is limited by the “max_movement_time”, a number in seconds that the movement time should not exceed. If the movement would exceed this amount of time, the speed is increased to ensure it doesn’t take longer. Moves occur at a constant speed, with the chess piece smoothly transitioning in a straight line from its original position to its new position.

To trigger a move, the player must first select a piece by clicking on the cell it’s located in. Then, click to the cell the piece should move to. If the player instead selects one of their other pieces, then that piece becomes selected instead. If the player clicks on the selected piece again, or an invalid move, it becomes unselected.

Normal movement of pieces is described in the table above. The king, queen, bishop, knight, rook and pawn all have the same movement as in regular 8x8 chess. For the purposes of pawn movement, “forward” is considered going up the board for the human player, and going down the board for the computer player.

In addition, be mindful of the following special moves.

Special moves:

- A pawn can move two squares forward if it is located on 2nd row from the top or bottom of the board (rank 2 and rank 13), and has not moved before.
- A king may perform a ‘castling’ move if it has not moved before, which allows it to move two squares horizontally in either direction so long as there is also a rook towards the direction it will move (on the same rank), and that rook hasn’t moved. When this move is performed, the rook is placed on the other side of the king, adjacent to it.
- Pawn promotion: When a pawn reaches the 8th rank (ie. when it crosses the halfway point on the 14x14 board), then it is promoted to a queen. It immediately turns into a queen and can be used as such in all subsequent moves

Only a camel or knight move may jump over pieces (or the rook when castling), and a player may not move a piece onto a cell already containing one of their own pieces. If a move causes the piece to enter a tile containing one of the opponent’s pieces, the opponent’s piece is ‘captured’ and removed from the board. All pieces capture on the same tiles as their regular movement, with the only exception being pawns which capture diagonally forwards instead, if there is a piece there. This is the only time they are allowed to move diagonally. If there is a piece directly in front of a pawn, it is blocked and cannot move to the cell occupied by that piece.

Examples of possible moves for each piece are shown below.

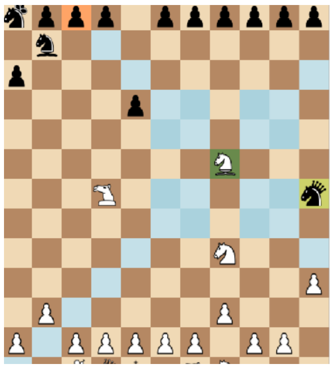


*Figure 1: Archbishop movement*

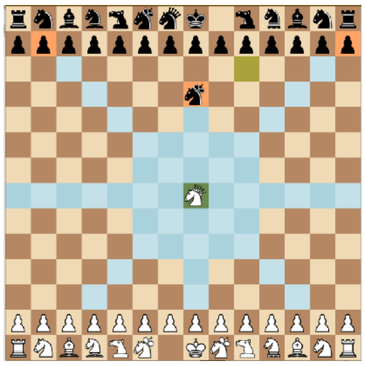


*Figure 2: Amazon movement*

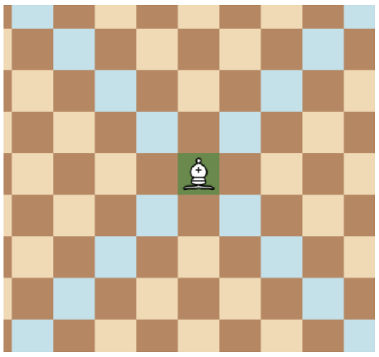


*Figure 3: Bishop movement*

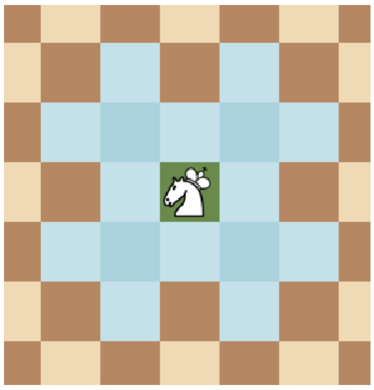


*Figure 4: General/Guard movement*

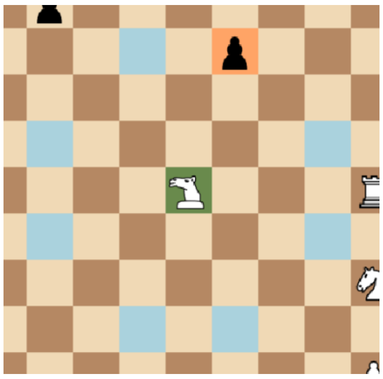


*Figure 5: Camel movement*

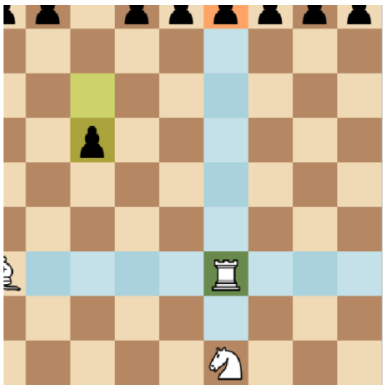


*Figure 6: Rook movement*

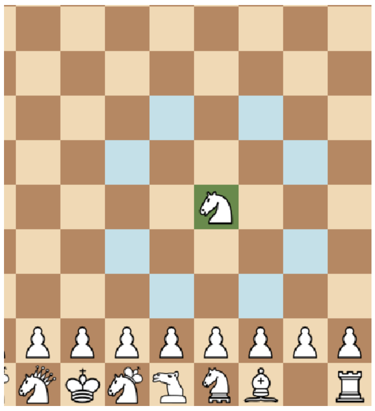


*Figure 7: Knight movement*

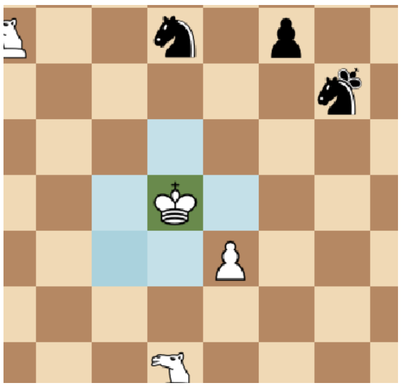


*Figure 8: King movement. The king cannot move to squares controlled by an opponent's piece.*

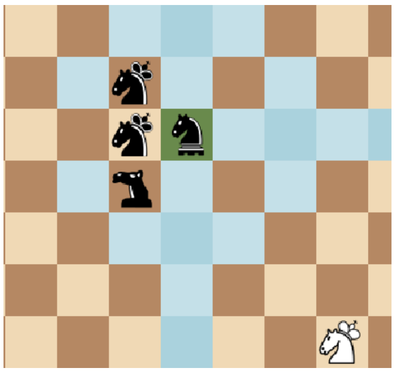


*Figure 9: Chancellor movement. The camel and generals block movement in the left direction, but knight moves can still jump over (same for other knight combinations).*

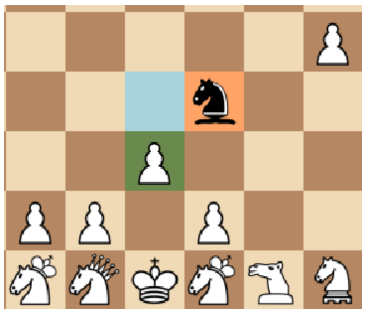


*Figure 10: Pawn movement. The pawn can only move diagonally when capturing and cannot move directly forward if blocked.*

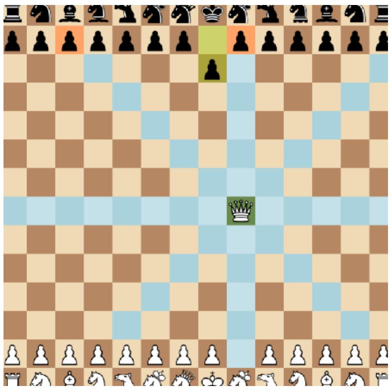
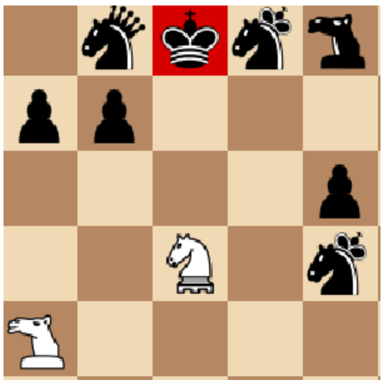


*Figure 11: Queen movement*

## Check and Checkmate



If after a move, a king is under attack, the king is said to be in 'check'. Under this circumstance, the player whose king is in check must do one of the following (all must already be legal moves):

- Move their king to a safe square
- Move a piece to block the attack
- Capture the attacking piece

*Figure 12: The black king is in check due to the white chancellor. Black cannot move their king, their only option is to move their General or Amazon to block (assuming there is no piece off-screen that can capture the white chancellor).*

This is because otherwise, the player would lose their king on the next turn, and therefore lose the game. If none of these possibilities are available, then the player has been checkmated – there is no move available to them that would save their king, and they have lost.

When check occurs, the king's square is highlighted in dark red, and the message "Check!" appears in the right sidebar. If a player attempts to make an otherwise legal move that doesn't protect their king, display a message on the right sidebar: "You must defend your king!", and the highlighted cell the king is on will flash 3 times with a duration of 0.5 seconds each.

A player cannot make a move that would result in their king coming under attack. This could be any of either:

- Moving the king to a square which is under attack by the opponent
- Moving a piece that is blocking an attack on their king by the opponent (this piece is said to be 'pinned')

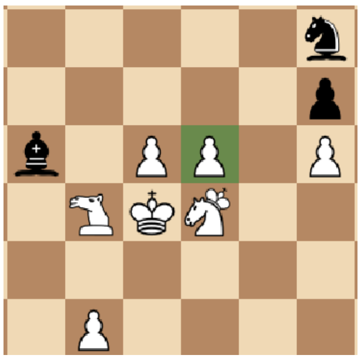


*Figure 13: The selected pawn is pinned by the archbishop, so cannot move. However if the archbishop was one space closer to it diagonally, then the pawn would be able to capture the archbishop but not move straight forward. If the king was on the square one space below where it currently is, then the camel would be pinned by the bishop.*

Illegal moves due to check or pins should not be highlighted as blue tiles when selecting a piece to potentially move it – only legal moves should be highlighted.

## Computer AI movement

The way the computer player determines moves is up to you. It can be as simple as choosing a move randomly out of all available legal moves. However, you should try and make it at least a bit intelligent. For example, here is guidance on some basic rules you might want to have:

- Capture a piece if the piece's value is higher than the capturing piece, and if multiple such options exist, choose the one with the highest difference

- If a piece is threatened to be captured by a piece of lower value (or any piece if the threatened one is undefended), move it
- Prefer to only move to squares that are not under attack by a piece of lower value (or even a piece of higher or equal value if the square is undefended) – let's call these 'safe squares'
- If possible, attack the opponent's king or a square adjacent to it if it's not already under attack
- If possible, checkmate the opponent's king
- If all else fails, choose a random move (prefer safe squares, but it's possible none may be available)

## Win and lose conditions

The game ends when either one player runs out of time, or their king is checkmated. Then other player wins.

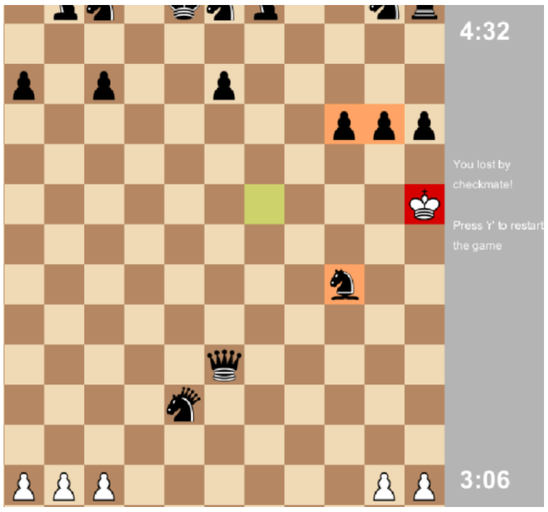
If the human player wins by checkmate, display a message saying "You won by checkmate" in the right sidebar. If the human player wins due to the timer, display instead "You won on time".

If the human player loses by checkmate, display a message saying "You lost by checkmate" in the right sidebar. If instead the cause was the timer, display "You lost on time".

The player can also resign the game by pressing 'escape' on the keyboard. The game ends and the message "You resigned" is displayed in the right sidebar.

When the game ends, the board remains intact and frozen so that the player cannot make any moves (but may restart the game with the key press 'r'). If checkmate occurred, the board should highlight the king of the checkmated player in red and the pieces contributing to checkmate in orange. Pieces contributing to checkmate are defined as a piece that is either attacking the king or one of the empty squares adjacent to it, or defending a piece that the king could otherwise capture. For each such square, there should only be one piece highlighted.

If there are no legal moves for a player, then the game is considered a draw and enters the end state. Display the message "Stalemate – draw".


4:32

You lost by checkmate!

Press 'r' to restart the game

3:06

*Figure 14: The white king is checkmated. The contributing pieces are the archbishop and two pawns.*

# Application

Your application will need to adhere to the following specifications:

- The window must have dimensions 672x792
- The game must maintain a frame rate of 60 frames per second.
- Your application must be able to compile and run on any the university lab machines (or Ubuntu VM) using gradle build & gradle run. Failure to do so, will result in 0% for Final Code Submission.
- Your program must not exhibit any memory leak.

- You must use the processing library (specifically processing.core and processing.data), you cannot use any other framework such as javafx, awt or jogl

You have been provided a /resources folder which your code can access directly (please use a relative path). These assets are loadable using the *loadImage* method attached to the PApplet type. Please refer to the processing documentation when loading and drawing an image. You may decide to modify these sprites if you wish to customise your game. You will be required to create your own sprites for any extensions you want to implement.

## Extension

The extension is worth 2 marks maximum. For an extension, you can choose to implement one of the following:

- New piece type with different movement and/or special behaviour
- Start menu to select options without having to change the config file
- More complex computer AI (specify difficulty in config?)
- Option for 2-player in config (two human players)
- Option for fog of war chess in config (Tiles not able to be moved to are greyed out. The only pieces of the opponent's that are visible are those which are present on tiles available to be moved to. Game ends upon king capture.)
- Sound effects
- Expandable resizable board (drag corners to change game resolution)
- More pawn promotion options rather than just promote to queen – user can choose while in-game
- Allow pre-moves (and/or drag and drop pieces rather than just click to move)
- Allow different types of new configurable pieces (combinations) within the json config file, and what their symbol would be in the layout
- Multiple levels specified in the config for the purpose of solving chess puzzles / problems.

OR, a feature you come up with which is of a similar or higher level of complexity (ask your tutor)

Please ensure you submit a config and level layout file with the features of your extension, and ensure the extension doesn't break any of the default behaviour. Also, describe your extension functionality in the report.

## Marking Criteria (18%)

Your final submission is due on Sunday 14 May at 11:59PM. To submit, you must upload your build.gradle file and src folder to Ed. Please also include sample config and layout files that you have tested with your program to ensure it works. Do NOT submit the build folder (unless you only include the build/reports/ folder which contains the results of your testing and code coverage). Ensure src is in the root directory with the other files, and not part of a zip, then press MARK. Submit your report and UML to canvas.

A demo of your assignment will be conducted during labs in week 12 where you will demonstrate the features you have created to your tutor.

### Final Code Submission and Demo (10%)

You will need to have implemented and satisfied requirements listed in this assignment. Make sure you have addressed the following and any other requirements outlined previously.

- Window launches and shows checkerboard pattern.
- Configuration file is correctly read in – timers display correctly
- Map loads and pieces are displayed in correct positions
- Piece colour for player and computer is correct
- Pieces are controlled by mouse clicks
    - A piece can be selected – green highlight appears
    - Potential moves show in blue highlights, or red for potential captures
    - Clicking on a highlighted blue/red cell cause the selected piece to move to that location
    - Captured pieces are deleted properly
- Pieces move correctly according to the table of their movement
- Piece movement – is smooth and correct speed
- If a move causes the opponent's king to come under attack, display "Check!" and highlight the king's tile in dark red.
- While the king is in check, only moves to save the king are allowed:
    - Capture the attacking piece
    - Move a piece to block
    - Move the king to a safe square
- A move that causes your own king to be in check is not allowed:
    - Moving the king to a square that is under attack
    - Moving a pinned piece that was blocking an attack on the king
- Special moves: Pawn can move two squares initially, Castling, Pawn promotion to queen on the 8th rank (ie. when passing the halfway point)
- Computer AI moves automatically after the player makes their move
- Computer AI only makes legal moves
- Game ends if either player runs out of time – message is displayed correctly
- Game ends if either player is checkmated – message is displayed correctly
- Pieces contributing to checkmate have their tiles highlighted in red
- Player can resign if they want to by pressing 'escape'
- Timer counts down and increments according to config after a move is made
- Ensure that your application does not repeat large sections of logic
- Ensure that your application is bug-free

## Testcases (3%)

During development of your code, add testcases to your project and test as much functionality as possible. You will need to construct unit test cases within the src/test folder using JUnit. To test the state of your entities without drawing, implement a simple loop that will update the state of each object but not draw the entity.

Ensure your test cases cover over 90% of execution paths (Use jacoco in your gradle build) Ensure your test cases cover common cases. Ensure your test cases cover edge cases. Each test case must contain a brief comment explaining what it is testing. To generate the testing code coverage report with gradle using jacoco, run "gradle test jacocoTestReport".

## Design, Report, UML and Javadoc (3%)

You will need to submit a report that elaborates on your design. This will include an explanation of any object-oriented design decisions made (such as reasons for interfaces, class hierarchy, etc) and an explanation of how the extension has been implemented. This should be no longer than 500 words. This report will be submitted through Canvas.

You will need to submit a UML diagram in PDF form to Canvas to provide a brief graphical overview of your code design and use of Object Oriented Principles such as inheritance and interfaces. Markers will use this to determine whether you have appropriately used those principles to aid you in your design, as well as figure out whether more should have been done. A general guideline is that markers will be looking for some use of inheritance or interfaces, how extensible the code is, and penalising repeated code. Note that you should not simply use a UML generator from an IDE such as Eclipse, as they typically do not produce diagrams that conform to the format required. We suggest using software such as LucidChart or draw.io for making your diagrams.

Your code should be clear, well commented and concise. Try to utilise OOP constructs within your application and limit repetitive code. The code should follow the conventions set out by the Google Java Style Guide. As part of your comments, you will need to create a Javadoc for your program. This will be properly covered in week 11 but the relevant Oracle documentation can be found here.

| | |
|---|---|
| Report, UML and OO design: | 2% |
| Javadoc, comments, style and readability: | 1% |

## Extension (2%)

Implement an extension as described above. Partial marks may be awarded if you choose a more limited extension or it is partially completed. Please specify what extension you decided to implement within your report.

## Suggested Timeline

Here is a suggested timeline for developing the project. Note that it is released on April 3 (start of week 7) and due May 14 (end of week 11).

**Week 7:** Familiarise yourself with gradle and processing, utilising the processing Javadoc and week 8 supplementary lecture. Identify opportunities to utilise Object Oriented Design principles such as inheritance and interfaces and begin to plan a design for the codebase with regards to the classes that you will need to make. Make a rough UML diagram for your design that you can base your codebase from.

**Week 8:** Begin writing the actual code for the program. Start small, for example by initially creating the board layout and pieces, then gradually add more elements. At the end of the week, you should have loading in the board and piece movement finished, as well as some sprite management. If confident, use Test Driven Development (writing test cases at same time as writing the code). Conduct a large amount of user testing to ensure the initial mechanics work as expected.

**Weeks 9-10:** Develop more gameplay features, such as check, checkmate, special moves, and the computer AI. Sprite management should be streamlined at this point. You should have a fairly high code coverage for your test cases at this stage. If you are noticing any questionable design decisions, such as God classes or classes that are doing things they logically should not be doing, this is the time to refactor your code. Think about what extension you want to make and start to implement it.

**Week 11:** Finish developing the remaining features for your program, notably the configuration file, and timers. Additionally, finish writing your testing suite. Create the UML and Javadoc for the program. Fix any remaining bugs that your code exhibits. Submit your code to Ed (by uploading the entire project and pressing MARK) and submit your UML to Canvas in PDF form.

**Week 12:** Demonstrate the completed program to your tutor during the week 12 lab. They will check each criteria item has successfully been completed, and may ask you questions about how you implemented it to test your understanding.

## Academic Declaration

By submitting this assignment you declare the following:

*I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.*

*I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.*

*I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.*

*I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.*