LaSalle College
420CT2AS – Object Oriented Programming Concepts (in C#)

Michelle M. Khalifé
Project Description

A Simplified and Partial Implementation of Chess

// 2D arrays, abstract classes, inheritance, polymorphism, interfaces, static classes, namespaces, using directives

Chess is a two-player strategy game played on an 8 x 8 board, with alternating white & black squares. It's known as the chessboard. Each player starts off with 16 pieces, either white or black. White makes the first move (moving upwards), black goes next (moving downwards), and the goal of both players is to breach their opponent's territory and checkmate their king. This happens when any valid move the king attempts to make results in his capture. In other words, the king is out of moves.

The image below shows the board's configuration at the beginning of the game. Rows 1 to 8 are called ranks and columns A to H are called files.

https://www.houseofstaunton.com/hos-setting-up-the-chess-board

Pieces	Points	Subset of Moves
The **Pawn** (in rows 2 and 7)	1	_ If the coast is clear, move one square forward _ If the coast is clear & it's the first move, move one or two squares forward _ If on either adjacent diagonal square, there is an opponent's piece, the pawn can capture it and take its place
The **Rook** (in A1, A8, H1, H8)	5	_ Move horizontally along the same rank _ Move vertically along the same file _ If the coast is clear, you can move as far as the boundary _ If there is an opponent's piece, the rook can capture it and take its place
The **Knight** (in B1, B8, G1, G8)	3	_ Move in L, two squares along the same file, up or down, and one square to the left or to the right _ Move in L, two squares along the same rank, left or right, and one square up or down _ If the landing square is occupied by an opponent's piece, the knight can capture it and take its place _ If the landing square is occupied by the players' own piece, the move is not allowed.
The **Bishop** (in C1, C8, F1, F8)	3	_ Moves in diagonal, on its own colour, left or right, up or down _ If the coast is clear, the bishop can move as far as the boundary _ If there is an opponent's piece along the way, the bishop can capture it and take its place _ If a square along the way is occupied by the players' own piece, the bishop cannot go through
The **Queen** (in D1, D8), on her colour	9	_ Rook & Bishop logic
The **King** (in E1, E8), opposite colour	Int32.MaxValue	_ Moves only one square at a time in any direction. _ If the adjacent square is occupied by an opponent's piece, the knight can capture it and take its place _ If the landing square is occupied by the players' own piece, the move is not allowed.

1. **Position Class**

➔ Implement a **class Position** characterized by an *int rank* (to represent the row) and a *char file* (to represent the column).
➔ Implement validation logic for Rank (0-8) and Files (A-H) properties.
➔ You will need a fully parametrized constructor for the class.
➔ Override *ToString()* to print the position in the following format: A1, B2, C3, etc.
➔ Override *Equals()* to return true if two positions have the same rank and file, false otherwise.

2. **Chess Pieces – Base & Derived Classes**

Base Class – When we think of a chess piece, the following comes to mind:

- the colour,
- the name,
- the number of points the opponent gains if the piece is captured,
- the starting or current position on the board, and
- the allowed moves across the board.

➔ Implement an **abstract class ChessPiece** with the first two characteristics only
➔ Implement validation logic for Name (see below) and Colour (white, black).
➔ Implement a fully parametrized constructor.
➔ Override ToString() to print the first character in the piece's colour, followed by the first character in the piece's name. E.g., for the black queen: "BQ", and for the white bishop: "WB"

Derived Classes – Pawn, Rook, Knight, Bishop, King, Queen

➔ Implement, for each type, a **class that derives from Piece**
➔ Add a *const int Points* to the class and set it directly to its value, given in the table above. The constant will behave like a static variable and will belong to the class. Access Syntax: *ClassName.Points*
➔ Implement a fully parameterized constructor for each class that calls the base constructor.
➔ Override ToString() for the Knight and **instead** of printing K, print N.

3. **Board Utils Static Class**

Implement a static class BoardUtils.

➔ Implement a static method **FileCharToIdx()** that takes a *char fileChar* ('A', 'B', 'C', 'D', …'H') and returns an *int* (0, 1, 2, …, 7). This helps translate the board's files to the column index in the 2D array.

➔ Implement a static method **FileIntToChar()** that take a 2D array *int fileIdx* and returns the correspondent board *char*. For e.g., 0 would return 'A', and 3 would return 'D', etc.

➔ Implement a static method **RankIntToIdx()** that takes a board's *int rank* and returns the correspondent 2D row index. For e.g., rank 8 would return 0 and rank 1 would return 7.

➔ Implement a static method **RankIdxToInt()** that takes a 2D *int idx* and returns the correspondent board rank. For e.g., idx 7 would return 1 and idx 0 would return 8.

➔ Implement a static method **PositionToPiece()** that takes a *Board board* and a *string position (A1, B2, H0, etc.)* and returns the piece at that position. In the image above, *PositionToPiece(A1)* would return an object: Name/Rook and Colour/white.

Optional Implementation

Feel free to optimize the methods above, using enums, char for ranks, the position class, or other constructs/approaches.

4. ChessBoard Class

Think of the chessboard as a (2D array) container of chess pieces. *Syntax: Piece [,] = new Piece [8,8];*

➔ Implement a **class ChessBoard** with an 8 x 8 2D array of Piece
➔ Implement a parameterless constructor that instantiates a new 8x8 board
➔ Implement a **SetUpBoard()** method that initializes the board with the configuration in the image above. Below is how you would set the black queen & the white king. You would have to do this for all the pieces. Do not hard-code 32 pieces… You have to think of patterns.

```
// set queens
Board[0, 3] = new Queen("Queen", rank == 0 ? "Black" : "White");

// set kings
Board[7, 4] = new King("King", rank == 0 ? "Black" : "White");
```

➔ Implement a **DisplayBoard()** method that iterates over the 2D array's rows & cols and prints its content. If the cell contains a piece, print the piece, else print an underscore.

Optional Display Implementation

Alternatively, you can: Console.BackgroundColor = ConsoleColour.DarkGreen; and

Instead of printing B and W, set the text colour to black or white depending on the piece
Console.ForegroundColor = ConsoleColour.White; or
Console.ForegroundColor = ConsoleColour.Black; and

Instead of printing all underscores in the same color, follow the board's square colours:
Console.ForegroundColor = (row + col) % 2 == 0 ? ConsoleColor.White : ConsoleColor.Black;

This would also require dropping the colour char in the ToString() impl. of the derived pieces

5. IMovable Interface

To implement an interface, first select Add → New Item → Interface.

Name your interface IMovable. Interfaces often start with the letter I to let the programmers know this is an Interface, not a class. The interface will offer behaviour/methods related to movement. The methods are public and abstract by default, but you can be explicit if you want to. They don't have a body or curly braces. Only the signature, i.e., *returnType methodName (param*);*

➔ Declare the following abstract methods in IMovable:

```
abstract bool IsValidMove(Board board, String fromPosition, String toPosition);
abstract ChessPiece MovePiece(Board board, String fromPosition, String toPosition);
```

➔ Let the ChessPiece class implement the interface IMovable like so:

```
abstract class Piece : IMovable
```

➔ Re-declare the interface methods in the ChessPiece class. No implementation, signature only.

➔ Override the interface methods for each derived class. Follow the compiler's prompt: in the absence of a body implementation, *throw new NotImplementedException();*

➔ Implement **IsValidMove()** concretely for the Pawn class, **as a team.** You want to answer the question: can a pawn move *fromPos* to *toPos*?

- Extract fromRowIdx and fromColIdx from fromPos
- Extract toRowIdx and toColIdx from toPos
- Based on the pawn's current position, generate the possible positions it can move into. For example, the pawn in A2 is at [6,0], and it can move into [5,0] or [4,0], because it's the first move and because the coast is clear. These indexes translate to positions A3 and A4.
- Save the positions in a local List<string> allowedPositions = new List<string>().
- If the *toPos* happens to be A3 or A4, the method should return true;

➔ Implement **MovePiece()** concretely for the Pawn class, **as a team.**

- This method is only called if IsValidMove() returned true.
- It assigns the pawn to the *toPos* position, which has to be translate to 2D indexes
- If there's a piece in the *toPos,* the method returns that piece

➔ Choose another piece and implement its methods, **individually.**

6. Player Class

A player is characterized by the following:

- a name,
- a colour,
- a score, and
- a List of captured pieces. *// Property Syntax: List<Piece> CapturedPieces {get; set;}*

➔ Implement a class Player with the above properties.
➔ Implement a constructor that accepts a name and a colour. In the constructor's body, instantiate a new List<Piece>(); You may explicitly set the Score to 0.
➔ Implement a *DisplayStatus()* method that prints all the player's info, including all the opponent pieces the player captured.

6. Namespaces

Typically, you want your files to be organized. Add the following folders to your project folder and move the classes into them.

- **Board**: Position, ChessBoard
- **Utils**: BoardUtils
- **Interface**: IMovable
- **GamePieces**: ChessPiece, Pawn, Rook, Knight, Bishop, King, Queen

Follow the compiler's prompt. Use a using directive followed by the name of the folder/namespace if you're in class A and making a ref to a class B that is within the project but outside your folder/namespace.

For example, to setup a board in the ChessBoard class, you need access to the Piece class. So, at the top of the ChessBoard class, you will add: `using` **ProjectName.GamePieces**

7. Main

Do not program the game, simply write a series of commands that tests what you're implementing.