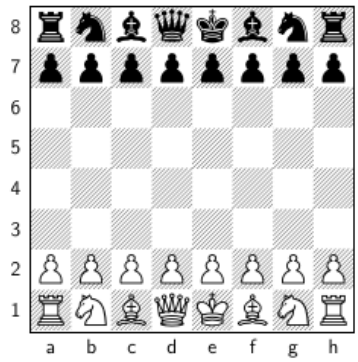
Chess

Finally, we go on to the greatest game!

This chapter covers how to represent the board, how to share common code between pieces, how to program the basic moves of individual pieces, how to account for the special piece moves (like castling, en passant, and promotion), and how to detect checks, stalemates and checkmates.

This chapter also briefly surveys how to implement poorly performing chess AIs inspired by the imagination of "tom7" DR. Tom Murphy VII PH.D. Wait, why would we implement poorly performing chess AIs? Well, they're a lot easier to build than highly performing chess AIs!

The Board



The chess board is a 8x8 grid. Pieces always begin set up in one fixed arrangement. We will hard-code this arrangement.

Chess players refer to squares on the board using `a1` to describe the bottom-left corner and `e4` for the space two squares ahead of white's king pawn. To make our programming easier we will refer to squares as array indices.

Representing the board with a 2D array the top left corner is at `[0][0]` or `a8`. `a1` is at array position `[7][0]`. Black's queen pawn at `d7` is at array position `[1][3]`. `e4` is at array position `[4][4]`.

We can write functions to convert both ways back and forth between chess notation and array indices. Creating these two conversions is useful because now we can translate either way between both systems.

```
// 'a8' returns {row: 0, col: 0}
// 'a1' returns {row: 0, col: 7}
// 'd7' returns {row: 1, col: 3}
// 'e4' returns {row: 4, col: 4}
function positionToIndex(position) {
    let rank = position[1];
    let file = position[0];

    let rowIndex = '87654321'.indexOf(rank);
    let colIndex = 'abcdefgh'.indexOf(file);

    return {row: rowIndex, col: colIndex};
}

function indexToPosition(row, col) {
    let rank = '87654321'[row];
    let file = 'abcdefgh'[col];
```

```
    let position = file + rank;
}
```

We can hard-code the creation of an initial board by writing out an 8x8 two-dimensional array. And we can hard-code the initial arrangement of the pieces and their colors too. Hard-coding in this instance doesn't take too long to write out by hand so we don't need to worry about over-optimizing or eliminating each piece of redundancy.

```
var board = [
    [null, null, null, null null, null, null, null],
    [null, null, null, null null, null, null, null],
    [null, null, null, null null, null, null, null],
    [null, null, null, null null, null, null, null],
    [null, null, null, null null, null, null, null],
    [null, null, null, null null, null, null, null],
    [null, null, null, null null, null, null, null],
    [null, null, null, null null, null, null, null],
]


var board = [
    [new Rook(0), new Knight(0), new Bishop(0), new Queen(0), new King(0), new Bishop(0), new Knight(0), new Rook
    [new Pawn(0), new Pawn(0), new Pawn(0), new Pawn(0), new Pawn(0), new Pawn(0), new Pawn(0), new Pawn(0)],
    [null, null, null, null null, null, null, null],
    [null, null, null, null null, null, null, null],
    [null, null, null, null null, null, null, null],
    [null, null, null, null null, null, null, null],
    [new Pawn(1), new Pawn(1), new Pawn(1), new Pawn(1), new Pawn(1), new Pawn(1), new Pawn(1), new Pawn(1)],
    [new Rook(1), new Knight(1), new Bishop(1), new Queen(1), new King(1), new Bishop(1), new Knight(1), new Rook
]
```

If we care about reducing redundancy we can use for loops to eliminate the repetitiveness of creating pawns and writing out the four empty rows in the middle.

```
let board = [];

// Set up black's pieces
let blackBackRow = [new Rook(0), new Knight(0), new Bishop(0), new Queen(0), new King(0), new Bishop(0), new Knig
let blackFrontRow = new Array(8);
for (let i = 0; i < blackFrontRow.length; i++) {
    blackFrontRow[i] = new Pawn(0);
}

// Add four empty rows for the middle of the board
for (let i = 0; i < 4; i++) {
    board.push(new Array(8))
}

// Set up white's pieces
let backRow = [new Rook(0), new Knight(0), new Bishop(0), new Queen(0), new King(0), new Bishop(0), new Knight(0)
let frontRow = new Array(8);
for (let i = 0; i < frontRow.length; i++) {
    frontRow[i] = new Pawn(0);
}
```

We can go even further and eliminate the redundancy of writing out the order the back row for black and white in two different spots. Create a function called `createPieces` that accepts `color` as a parameter. The function returns an object with properties `frontRow` (pawns) and `backRow` so we can pluck these two rows off and arrange them on the overall board correctly.

```
function createPieces(color) {
    let frontRow = (new Array(8)).map(() => new Pawn());
    let backRow = [new Rook(), new Knight(), new Bishop(), new Queen(), new King(), new Bishop(), new Knight(), r

    // add the color for each piece
    frontRow.forEach(piece => piece.color = color);
    backRow.forEach(piece => piece.color = color);

    return {frontRow, backRow};
}

function createBoard() {
    let board = new Array(8);
    let blackPieces = createPieces('black');
    let whitePieces = createPieces('white');

    board[0] = blackPieces.backRow;
    board[1] = blackPieces.frontRow;

    for (let i = 2; i < board.length - 2; i++) {
        board[i] = new Array(8);
    }

    board[board.length - 2] = whitePieces.frontRow;
    board[board.length - 1] = whitePieces.backRow;
}
```

Moving Pieces

A classic chess opening is moving the white king pawn up to spaces to control the center. Let's write a function that manipulates our board to make that move.

In chess players write this move down as `e4`. Chess has a whole notation with rules about how to write down moves. The intricacies of chess notation is fascinating. We would be putting the cart in front of the horse if we rabbit-holed into creating a program that understands and manipulates the board by receiving commands in chess notation. We need to use our own simpler chess notation.

We will initially manipulate our chess board by specifying the exact starting square and the exact ending square of the piece being moved. We can build more sophisticated ways to interface with the board on top of this basic functionality.

```
e2 to e4
6,4 to 4,4
```

Writing down either format of these is fine because we can already translate between the two ways to refer to those squares. Inside the program we will standardize on manipulating array index coordinates. This means we can refer to position like `e2` and `e4` but inside the program we will immediately convert those formats to the array index format.

Notice how `makeMoveByPosition` does nothing else but perform a conversion and immediately feed into `makeMoveByIndex`.

```
function makeMoveByPosition(board, startPosition, endPosition) {
    let startIndex = positionToIndex(startPosition);
    let endIndex = positionToIndex(endPosition);
    makeMoveByIndex(board, startIndex.row, statIndex.col, endRow.row, endCol.col);
}

function makeMoveByIndex(board, startRow, startCol, endRow, endCol) {
    let startPiece = board[startRow][startCol];
```

```
        let endPiece = board[endRow][endCol];

        board[startRow][startCol] = null;
        board[endRow][endCol] = startPiece;
    }
```