# Introduction

## Goals of project

Learn programming

Deliver a real application

## Story-based development

Based on agile development

Each story adds some new value to user

Each story kept to the minimum complexity that adds some value

## Learning approach

Programming patterns, tools & techniques introduced only as they are used

Where possible, each is introduced in a context outside the app

The student is then invited to apply this learning, with some hints and guidelines

Model code available at the end of each module

## Technical approach

Web-based

Microsoft tooling platform

TypeScript rather than JavaScript

## Professional perspective

What’s realistic and what’s not

Stories have been nicely ordered

While there is some change to code, this is reduced because we know where we are going.

Watch out for ‘Professional Perspective’

# Create a web page with a welcome screen

## Learnings: patterns & techniques

Use Visual Studio

Create a web-based project

The index page

The run-edit cycle

Some simple Html

## Exercises

Create new typescript project

Run it unmodified.

Explain parts, including: TypeScript vs. JavaScript

Empty App.ts for now

Modify Index

Modify CSS

Show that if app is still running, changes can be picked up by refreshing the browser

## Code written

Index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8" />

<title>My board game</title>

<link rel="stylesheet" href="app.css" type="text/css" />

<script src="app.js"></script>

</head>

<body>

<h1>Reversi Game</h1>

<p>Look up the <a href="http://www.yourturnmyturn.com/rules/reversi.php" target="\_blank">rules</a></p>

</body>

</html>

app.ts - empty

app.css – as is

# Draw the outline of a board

Large green square with an outline. Text to the side

## Learnings: patterns & techniques

Canvas

Script files

TypeScript (& JavaScript)

Renderer

Rectangle

Fill & stroke

Arc

Procedural programming - executed in order

Global variables (which we want to avoid)

## Exercises

Add code

Explain code

What happens if:

* You change numbers in the rectangle method
* You change the order of the statements
* Draw another rectangle of different size, colour & location
* You spell canvas wrong in various places.
* Step through code

## Code written

index.html

…

<body>

<canvas width="500" height="500"></canvas>

<h1>Reversi Game</h1>

…

app.ts

var canvas: HTMLCanvasElement;

var renderer: CanvasRenderingContext2D;

window.onload = function () {

canvas = document.getElementsByTagName("canvas")[0];

renderer = canvas.getContext("2d");

//Draw background

renderer.fillStyle = 'green';

renderer.fillRect(0, 0, 480, 480);

//Draw outline

renderer.strokeStyle = 'black';

renderer.strokeRect(0, 0, 480, 480);

}

app.css

canvas {

float: left;

}

# Create squares in specific positions

## Learnings: patterns & techniques

Separation of concerns - domain logic from presentation

Beginnings of object modelling

Classes and instances

Constructor

Writing a generalised function

Passing params

## Code written

app.ts

const squareSide: number = 60;

var canvas: HTMLCanvasElement;

var renderer: CanvasRenderingContext2D;

import Square = model.Square;

window.onload = function () {

canvas = document.getElementsByTagName("canvas")[0];

renderer = canvas.getContext("2d");

var sq = new Square(0, 0);

drawing.drawSquare(sq, renderer);

sq = new Square(0, 1);

drawing.drawSquare(sq, renderer);

var sq = new Square(1, 0);

drawing.drawSquare(sq, renderer);

}

drawing.ts

module drawing {

import Square = model.Square;

export function drawSquare(sq: Square, renderer: CanvasRenderingContext2D) {

//Draw background

renderer.fillStyle = 'green';

renderer.fillRect(sq.col \* squareSide, sq.row \* squareSide, squareSide, squareSide);

//Draw outline

renderer.strokeStyle = 'black';

renderer.strokeRect(sq.col \* squareSide, sq.row \* squareSide, squareSide, squareSide);

}

}

model.ts

module model {

export class Square {

constructor(public col: number, public row: number) {

}

}

}

# Create a whole board

## Learnings: patterns & techniques

Loops

Instance method (function on an object)

Using external libraries, installed via NuGet (lodash & definitely typed - but delete duplication)

## Code written

### Board object

export class Board {

constructor() {

this.squares = [];

for (var col = 0; col < 8; col++) {

for (var row = 0; row < 8; row++) {

this.squares.push(new Square(col, row));

}

}

}

private squares: Square[];

//If the coordinates lie outside of the board boundaries, returns 'undefined'

public getSquare(col: number, row: number): Square {

return \_.find(this.squares, sq => sq.col === col && sq.row == row);

}

}

### initialise board within onload

var board: Board;

window.onload = function () {

canvas = document.getElementsByTagName("canvas")[0];

renderer = canvas.getContext("2d");

board = new Board();

}

### drawBoard function

export function drawBoard(board: Board, renderer: CanvasRenderingContext2D) {

for (var col: number = 0; col <= 7; col++) {

for (var row: number = 0; row <= 7; row++) {

const sq = board.getSquare(col, row);

drawSquare(sq, renderer);

}

}

}

# Set up the starting position

## Learnings: patterns & techniques

Enum

Importance of naming & renaming

## Code written

### new constant PieceRadius (in App)

const pieceRadius = 25;

### New enum (in models)

export enum Side { black, white }

### Function to get drawing colour from Side value

import Side = model.Side;

export function getColourForSide(side: Side): string {

return side === Side.black? 'black' : 'white';

}

### Modify square to know if occupied

export class Square {

constructor(public col: number, public row: number) {

this.occupiedBy = null;

}

public occupiedBy: Side;

}

### new drawPiece function

export function drawPiece(

sq: Square,

colour: string,

renderer: CanvasRenderingContext2D) {

var centreX: number = sq.col \* squareSide + squareSide / 2;

var centerY: number = sq.row \* squareSide + squareSide / 2;

renderer.fillStyle = colour;

renderer.beginPath();

renderer.arc(centreX, centerY, pieceRadius, 0, 2 \* Math.PI);

renderer.fill();

}

### Modify drawSquare to draw piece if occupied

export function drawSquare(sq: Square, renderer: CanvasRenderingContext2D) {

//Draw background

renderer.fillStyle = 'green';

renderer.fillRect(sq.col \* squareSide, sq.row \* squareSide, squareSide, squareSide);

//Draw outline

renderer.strokeStyle = 'black';

renderer.strokeRect(sq.col \* squareSide, sq.row \* squareSide, squareSide, squareSide);

if (sq.occupiedBy != null) {

var colour: string = getColourForSide(sq.occupiedBy);

drawPiece(sq, colour, renderer);

}

}

### Set up starting pieces in the Board’s constructor

this.getSquare(3, 3).occupiedBy = Side.white;

this.getSquare(4, 4).occupiedBy = Side.white;

this.getSquare(4, 3).occupiedBy = Side.black;

this.getSquare(3, 4).occupiedBy = Side.black;

# Add a cursor, starting in the top-left corner

## Learnings: patterns & techniques

## Code written

### Create a cursor location global variable on App & initialise in the OnLoad & draw it

var cursorLocation: Square;

window.onload = function () {

canvas = document.getElementsByTagName("canvas")[0];

renderer = canvas.getContext("2d");

var board = new Board();

drawing.drawBoard(board, renderer);

cursorLocation = board.getSquare(0, 0);

renderer.strokeStyle = 'yellow';

renderer.strokeRect(cursorLocation.col \* squareSide, cursorLocation.row \* squareSide, squareSide, squareSide);

}

### Refactor by extracting a general function to draw square outline in any colour

export function drawSquareOutline(square: Square, colour: string, renderer: CanvasRenderingContext2D) {

renderer.strokeStyle = colour;

renderer.strokeRect(square.col \* squareSide, square.row \* squareSide, squareSide, squareSide);

}

Simplify the onLoad function to:

window.onload = function () {

canvas = document.getElementsByTagName("canvas")[0];

renderer = canvas.getContext("2d");

var board = new Board();

drawing.drawBoard(board, renderer);

cursorLocation = board.getSquare(0, 0);

~~renderer.strokeStyle = 'yellow';~~

~~renderer.strokeRect(cursorLocation.col \* squareSide, cursorLocation.row \* squareSide, squareSide, squareSide);~~

drawing.drawSquareOutline(cursorLocation, 'yellow', renderer);

}

And also re-use it to simplify the drawSquare function

export function drawSquare(sq: Square, renderer: CanvasRenderingContext2D) {

//Draw background

renderer.fillStyle = 'green';

renderer.fillRect(sq.col \* squareSide, sq.row \* squareSide, squareSide, squareSide);

//Draw outline

~~renderer.strokeStyle = 'black';~~

~~renderer.strokeRect(sq.col \* squareSide, sq.row \* squareSide, squareSide, squareSide);~~

drawSquareOutline(sq, 'black', renderer);

if (sq.occupiedBy != null) {

var colour: string = getColourForSide(sq.occupiedBy);

drawPiece(sq, colour, renderer);

}

}

# Move the cursor around using the arrow keys

## Code written

### MoveCursorBy

function moveCursorBy(cols: number, rows: number) {

var col = cursorLocation.col + cols;

var row = cursorLocation.row + rows;

cursorLocation = board.getSquare(col, row);

drawing.drawBoard(board, renderer); //Re-drawing board clears the current cursor

drawing.drawSquareOutline(cursorLocation, 'yellow', renderer);

}

Test this by moving to a possible start location on loading

window.onload = function () {

canvas = document.getElementsByTagName("canvas")[0];

renderer = canvas.getContext("2d");

board = new Board();

cursorLocation = board.getSquare(0, 0);

moveCursorBy(3, 2);

~~drawing.drawSquareOutline(cursorLocation, 'yellow', renderer);~~

}

### OnKeyDown (in App)

window.onkeydown = function (ke: KeyboardEvent) {

ke.preventDefault();

switch (ke.keyCode) {

case 37: // left arrow

moveCursorBy(-1, 0);

break;

case 38: // up arrow

moveCursorBy(0, -1);

break;

case 39: // right arrow

moveCursorBy(1, 0);

break;

case 40: // down arrow

moveCursorBy(0, 1);

break;

}

}

Test this. But see what happens when you move to the edge of the board. So …

### Key the cursor within bounds

On Board object

function keepWithinBounds(value: number): number {

if (value < 0) return 0;

if (value > 7) return 7;

return value;

}

and use this within the moveCursorBy function:

function moveCursorBy(cols: number, rows: number) {

~~var row = cursorLocation.row;~~

~~var col = cursorLocation.col;~~

var col = board.keepWithinBounds(cursorLocation.col + cols);

var row = board.keepWithinBounds(cursorLocation.row + rows);

cursorLocation = board.getSquare(col, row);

drawing.drawBoard(board, renderer); //Re-drawing board clears the current cursor

drawing.drawSquareOutline(cursorLocation, 'yellow', renderer);

}

Test

# Place a piece on the board

## Code written

### placePiece function

function placePiece(side: Side): void {

cursorLocation.occupiedBy = side;

}

you’ll also need (up with the other imports)

import Side = model.Side;

test this by *temporarily* calling it within onload:

window.onload = function () {

canvas = document.getElementsByTagName("canvas")[0];

renderer = canvas.getContext("2d");

board = new Board();

cursorLocation = board.getSquare(0, 0);

moveCursorBy(3, 2);

placePiece(Side.black);

}

It doesn’t work!

Trace through to see what is happening. All is working but the board isn’t being re-drawn. So add:

function placePiece(side: Side): void {

//Place new piece

cursorLocation.occupiedBy = side;

drawing.drawBoard(board, renderer);

}

this should now draw, but note that we have lost the cursor. We could add functionality to draw the cursor again, but a simpler option would be to replace that line with a call to move the cursor, but by zero, which causes both the board and cursor to be re-drawn.

function placePiece(side: Side): void {

cursorLocation.occupiedBy = side;

~~drawing.drawBoard(board, renderer);~~

moveCursorBy(0,0);

}

Now *delete the* placePiece(Side.black); linefrom onload as soon as you can see it is working

### Modify the onkeydown function to recognise ‘b’ and ‘w’ keys

window.onkeydown = function (ke: KeyboardEvent) {

ke.preventDefault();

switch (ke.keyCode) {

case 37: // left arrow

moveCursorBy(-1, 0);

break;

case 38: // up arrow

moveCursorBy(0, -1);

break;

case 39: // right arrow

moveCursorBy(1, 0);

break;

case 40: // down arrow

moveCursorBy(0, 1);

break;

case 66: // 'b'

placePiece(Side.black);

break;

case 87: // 'w'

placePiece(Side.white);

break;

}

}

Test it out.

# Alternate turns

## Code written

## Add a GameManager class

Think of this as the referee that has responsibility to manage the turns, enforce the rules, and decide the outcome of the game.

export class GameManager {

public constructor(public board: Board) {

this.turn = Side.black; //The rules state that Black always moves first

}

public turn: Side;

public placePiece(sq: Square): void {

sq.occupiedBy = this.turn;

//Now set the next turn

if (this.turn == Side.black) {

this.turn = Side.white

} else {

this.turn = Side.black;

}

}

}

Initialise the GameManager in the onload function:

import GameManager = model.GameManager;

const squareSide: number = 60;

const pieceRadius: number = 25;

var canvas: HTMLCanvasElement;

var renderer: CanvasRenderingContext2D;

var cursorLocation: Square;

var board: Board;

var game: GameManager;

window.onload = function () {

canvas = document.getElementsByTagName("canvas")[0];

renderer = canvas.getContext("2d");

board = new Board();

game = new GameManager(board);

cursorLocation = board.getSquare(0, 0);

moveCursorBy(3, 2);

}

Change the onKeyDown function using ‘Enter’ to place next piece instead of

window.onkeydown = function (ke: KeyboardEvent) {

ke.preventDefault();

switch (ke.keyCode) {

case 37: // left arrow

moveCursorBy(-1, 0);

break;

case 38: // up arrow

moveCursorBy(0, -1);

break;

case 39: // right arrow

moveCursorBy(1, 0);

break;

case 40: // down arrow

moveCursorBy(0, 1);

break;

~~case 66: // 'b'~~

~~placePiece(Side.black);~~

~~break;~~

~~case 87: // 'w'~~

~~placePiece(Side.white);~~

~~break;~~

case 13: // Enter

game.placePiece(cursorLocation);

moveCursorBy(0,0);

}

}

Delete the now redundant placePiece function in app

~~function placePiece(side: Side) : void {~~

~~cursorLocation.occupiedBy = side;~~

~~moveCursorBy(0, 0);~~

~~}~~

# Update status

## Add method and property to GameManager

public status: string;

public updateStatus(): void {

switch (this.turn) {

case Side.black:

this.status = 'Black to play';

break;

case Side.white:

this.status = 'White to play';

break;

}

}

and call it in the GameManager’s constructor,

public constructor(board: Board) {

this.turn = Side.black; //The rules state that Black always moves first

this.updateStatus();

}

and again after a piece has been placed:

public placePiece(sq: Square): void {

sq.occupiedBy = this.turn;

//Now set the next turn

if (this.turn == Side.black) {

this.turn = Side.white

} else {

this.turn = Side.black;

}

this.updateStatus();

}

## Add a placeholder into the Index page for the status

<body>

<canvas width="500" height="500"></canvas>

<h1>Reversi Game</h1>

<p>Status: <span id="status"></span></p>

<p>Look up the <a href="http://www.yourturnmyturn.com/rules/reversi.php" target="\_blank">rules</a></p>

</body>

## Add a function to copy the status from the GameManager to the page

In drawing.ts:

export function updateText(game: GameManager) {

document.getElementById("status").innerHTML = game.status;

}

Call this from within onload…

window.onload = function () {

canvas = document.getElementsByTagName("canvas")[0];

renderer = canvas.getContext("2d");

board = new Board();

game = new GameManager(board);

cursorLocation = board.getSquare(0, 0);

drawing.updateText(game);

moveCursorBy(3, 2);

}

and when after the user places a piece …

case 13: // Enter

game.placePiece(cursorLocation);

drawing.updateText(game);

moveCursorBy(0, 0);

# Can only place piece onto an empty square

## Add function to test if a placement would be valid

Add to Board class:

public wouldBeValidMove(sq: Square): boolean {

return sq.occupiedBy == null;

}

Only place a piece if this function passes:

public placePiece(sq: Square): void {

if (this.board.wouldBeValidMove(sq)) {

sq.occupiedBy = this.turn;

//Now set the next turn

if (this.turn == Side.black) {

this.turn = Side.white

} else {

this.turn = Side.black;

}

this.updateStatus();

}

}

Test that you can’t place over another piece (or at least not of the opposite colour!)

## Change the cursor colour to show that a move

On App

function moveCursorBy(cols: number, rows: number) {

var col = board.keepWithinBounds(cursorLocation.col + cols);

var row = board.keepWithinBounds(cursorLocation.row + rows);

cursorLocation = board.getSquare(col, row);

drawing.drawBoard(board, renderer); //Re-drawing board clears the current cursor

var cursorColour = board.wouldBeValidMove(cursorLocation) ? 'yellow' : 'red';

drawing.drawSquareOutline(cursorLocation, cursorColour, renderer);

}

# New piece must be adjacent to a piece of opposite colour

## Write a function to find the adjacent squares (neighbours)

TODO: Draw a sketch. This *must* be built up gradually and tested at each stage, dealing progressively with:

* Exclude the square itself
* Exclude squares that would be off the edge of the board

Explain: why private

//Returns all squares (on the board) that are immediate neighbours

//of the given square - between 3 and 8 of them.

private getAdjacentSquares(sq: Square): Square[] {

var neighbours: Square[] = [];

for (var col: number = sq.col - 1; col <= sq.col + 1; col++) {

for (var row: number = sq.row - 1; row <= sq.row + 1; row++) {

var neighbour = this.getSquare(col, row);

if (neighbour != undefined && neighbour != sq) {

neighbours.push(neighbour);

}

}

}

return neighbours;

}

## Write a function to test if a square has at least one neighbour occupied by specified side

private isAdjacentToPiece(sq: Square, piece: Side): boolean {

var neighbours = this.getAdjacentSquares(sq);

return \_.some(neighbours, sq => sq.occupiedBy == piece);

}

Explain: second line

## Call new function from within the test for a valid move

public wouldBeValidMove(sq: Square, side: Side): boolean {

return sq.occupiedBy == null && this.isAdjacentToPiece(sq, side);

}

Adding new parameter will break other code, so we need to add it within the calls

function moveCursorBy(cols: number, rows: number) {

var col = board.keepWithinBounds(cursorLocation.col + cols);

var row = board.keepWithinBounds(cursorLocation.row + rows);

cursorLocation = board.getSquare(col, row);

drawing.drawBoard(board, renderer); //Re-drawing board clears the current cursor

var cursorColour = board.wouldBeValidMove(cursorLocation, game.turn) ? 'yellow' : 'red';

drawing.drawSquareOutline(cursorLocation, cursorColour, renderer);

}

and

public placePiece(sq: Square): void {

if (this.board.wouldBeValidMove(sq, this.turn)) {

sq.occupiedBy = this.turn;

//Now set the next turn

if (this.turn == Side.black) {

this.turn = Side.white

} else {

this.turn = Side.black;

}

this.updateStatus();

}

}

Test. IT DOESN’T WORK as planned! Error in logic.

We need to be testing for adjacency to piece of the *opposite*  side.

We *could* write:

public wouldBeValidMove(sq: Square, side: Side): boolean {

var oppositeSide: Side;

if (side == Side.black) {

oppositeSide = Side.white

} else {

oppositeSide = Side.black;

}

return sq.occupiedBy == null && this.isAdjacentToPiece(sq, oppositeSide);

}

But notice that this is very similar to some code above. So much better to ‘factor out’ a function for getting the opposite side (in model.ts):

function oppositeSideTo(side: Side): Side {

return side === Side.black ? Side.white : Side.black;

}

then call this from both here (in Game class):

public placePiece(sq: Square): void {

if (this.board.wouldBeValidMove(sq, this.turn)) {

sq.occupiedBy = this.turn;

~~//Now set the next turn~~

~~if (this.turn == Side.black) {~~

~~this.turn = Side.white~~

~~} else {~~

~~this.turn = Side.black;~~

~~}~~

this.turn = oppositeSideTo(this.turn);

this.updateStatus();

}

}

and (in Board):

public wouldBeValidMove(sq: Square, side: Side): boolean {

return sq.occupiedBy == null &&

this.isAdjacentToPiece(sq, oppositeSideTo(side));

}

Test this

# Identify captured pieces

This is a big story, which we will attempt to break up into as small chunks as we can

First, let’s identify all the squares from the cursor to the edge of the board in all eight directions. Start by trying one direction, North:

public squaresNorthFrom(location: Square): Square[] {

var squares: Square[] = [];

for (var i = 1; i <= 7; i++) { //Can only be maximum of 7 steps in any direction to edge

var sq: Square = this.getSquare(location.col, location.row - i);

if (sq != undefined) {

squares.push(sq);

}

}

return squares;

}

We can test this by *temporarily* drawing the outline of those squares in blue. In App:

function moveCursorBy(cols: number, rows: number) {

var col = board.keepWithinBounds(cursorLocation.col + cols);

var row = board.keepWithinBounds(cursorLocation.row + rows);

cursorLocation = board.getSquare(col, row);

drawing.drawBoard(board, renderer); //Re-drawing board clears the current cursor

var cursorColour = board.wouldBeValidMove(cursorLocation, game.turn) ? 'yellow' : 'red';

drawing.drawSquareOutline(cursorLocation, cursorColour, renderer);

var captured = board.squaresNorthFrom(cursorLocation);

\_.forEach(captured, n => drawing.drawSquareOutline(n, 'blue', renderer);

}

Test this, and when you are happy it works, delete those last two lines again.

Now we’ll do East, but instead of writing a new function, we’ll rename and generalise this one, first adding an Enum for the different directions (in model):

export enum Direction { north, east }

then

public squaresFrom(location: Square, dir: Direction): Square[] {

var squares: Square[] = [];

for (var i = 1; i <= 7; i++) { //Can only be maximum of 7 steps in any direction to edge

~~var sq: Square = this.getSquare(location.col, location.row - i);~~

var sq: Square;

switch (dir) {

case Direction.north:

sq = this.getSquare(location.col, location.row - i);

break;

case Direction.east:

sq = this.getSquare(location.col + i, location.row);

break;

}

if (sq != undefined) {

squares.push(sq);

}

}

return squares;

}

Now a new function to identify captured squares in that direction:

public capturedSquares(placement: Square, side: Side, dir: Direction): Square[] {

var coveredSquares = [];

var squares = this.squaresFrom(placement, dir);

for (var i: number = 0; i < squares.length; i++) {

var sq: Square = squares[i];

if (sq.occupiedBy == side) {

return coveredSquares;; //Terminate loop

}

if (sq.occupiedBy == undefined) {

return []; //no squares are bookended

}

coveredSquares.push(sq);

}

return []; //Didn't find a bookend so return no squares;

}

test this, using one direction only (you’ll have to move the cursor around until you find a captured piece to the north)

function moveCursorBy(cols: number, rows: number) {

var col = board.keepWithinBounds(cursorLocation.col + cols);

var row = board.keepWithinBounds(cursorLocation.row + rows);

cursorLocation = board.getSquare(col, row);

drawing.drawBoard(board, renderer); //Re-drawing board clears the current cursor

var cursorColour = board.wouldBeValidMove(cursorLocation, game.turn) ? 'yellow' : 'red';

drawing.drawSquareOutline(cursorLocation, cursorColour, renderer);

var captured = board.capturedSquares(cursorLocation, game.turn, model.Direction.north);

\_.forEach(captured, n => drawing.drawSquareOutline(n, 'blue', renderer);

}

Then change last two lines to test that it works for East.

Now a new function to cover all of the directions (currently North and East only - we’ll add the others shortly)

//Returns a list of all squares that would be captured (ultimately to be flipped)

//by a given placement on a square

public allCapturedSquares(placement: Square, side: Side): Square[] {

var results = [];

\_.forEach(Direction, d => {

var toAdd = this.capturedSquares(placement, side, d);

\_.forEach(toAdd, sq => results.push(sq));

});

return results;

}

Test this:

function moveCursorBy(cols: number, rows: number) {

var col = board.keepWithinBounds(cursorLocation.col + cols);

var row = board.keepWithinBounds(cursorLocation.row + rows);

cursorLocation = board.getSquare(col, row);

drawing.drawBoard(board, renderer); //Re-drawing board clears the current cursor

var cursorColour = board.wouldBeValidMove(cursorLocation, game.turn) ? 'yellow' : 'red';

drawing.drawSquareOutline(cursorLocation, cursorColour, renderer);

var captured = board.allCapturedSquares(cursorLocation, game.turn);

\_.forEach(captured, n => drawing.drawSquareOutline(n, 'blue', renderer));

}

Add the remaining six directions to Direction:

export enum Direction { north, northEast, east, southEast, south, southWest, west, northWest }

the order doesn’t matter, but it is best to be neat and logical.

Modify logic in squaresFrom to work with all 8 directions:

switch (dir) {

case Direction.north:

sq = this.getSquare(location.col, location.row - i);

break;

case Direction.northEast:

sq = this.getSquare(location.col + i, location.row - i);

break;

case Direction.east:

sq = this.getSquare(location.col + i, location.row);

break;

case Direction.southEast:

sq = this.getSquare(location.col + i, location.row + i);

break;

case Direction.south:

sq = this.getSquare(location.col, location.row + i);

break;

case Direction.southWest:

sq = this.getSquare(location.col - i, location.row + i);

break;

case Direction.west:

sq = this.getSquare(location.col - i, location.row);

break;

case Direction.northWest:

sq = this.getSquare(location.col - i, location.row - i);

break;

}

test this

# 14: Captured pieces flipped automatically

## Add the rule that a move is only valid if at least one piece is going to be captured

In Board, change:

public wouldBeValidMove(sq: Square, side: Side): boolean {

return sq.occupiedBy == null &&

this.isAdjacentToPiece(sq, oppositeSideTo(side)) &&

this.allCapturedSquares(sq, side).length > 0;

}

Test this (checking that cursor shows red otherwise.

## Flip the captured pieces as part of the place piece function

public placePiece(sq: Square): void {

if (this.board.wouldBeValidMove(sq, this.turn)) {

//Place new piece

sq.occupiedBy = this.turn;

//Flip captured pieces

var flips: Square[] = this.board.allCapturedSquares(sq, this.turn);

\_.forEach(flips, sq => sq.occupiedBy = this.turn);

//Now set the next turn

this.turn = oppositeSideTo(this.turn);

this.updateStatus();

}

}

Test

# Keep track of piece counts

# Allow move to be skipped

# Identify the end of the game

# Automated player

## Learnings: patterns & techniques

Sub-classing: automated player and manual player

## Notes

Get list of valid moves.

First, pick one at random

Next option, go for move with highest number of pieces gained, otherwise random

# TO THINK ABOUT

When to introduce the idea of a Player object

Game master need to ask Players for next move

Introduce idea of an automated player

AutomatedPlayer1

# Match

Set up two players at the beginning

Specify number of games

Execute games, keeping cumulative score

Output results.

## Learnings: patterns & techniques

Executing model without a UI