# 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’

TODO: Use ‘function’ and ‘method’ consistently - former when they are called function, latter on objects.

# 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>

~~<div id="content"></div>~~

</body>

</html>

app.ts - empty

app.css – as is

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* What an IDE is, Visual Studio being our choice for this project
* How to create a new project in Visual Studio
* What HTML means
* The basic structure of a web page
* How to add a title, text, headings, and hyperlinks to a web page

## What next?

Now that we can put some simple text on a page, we will now want to add some graphics. The simplest graphics that would fit our problem domain would be to draw the outline of the gameboard.

# 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

TODO: Talk about running, stopping or just refreshing

index.html

<body>

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

<h1>Reversi Game</h1>

app.ts

//Global variables

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 – append to end of file

canvas {

float: left;

}

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* The HTML5 Canvas as the primary way to create graphics under the control of a progam
* The JavaScript progamming language: the only one understood by a web browser
* The TypeScript programming language: move powerful and easier to learn than JavaScript
* How to create a new file to contain some TypeScript
* When and where the TypeScript file is compiled into an equivalent JavaScript file
* How to tell the web-page that it needs to run the JavaScript file
* How to write code that will execute when the web-page is loaded (window.onload)
* The IDE’s intellisense helps you find functions that are available to you in a context
* A function is set of instuctions that has a name and may be invoked by that name
* The set (or ‘block’) of instructions (‘statements’) that make up a function are enclosed in ‘braces’ ( ‘{‘ and ‘{‘).
* Statements within braces should be indented - for readbility only. The IDE should do this automatically for you.
* Some other programming languages don’t require braces, but require the programmer to manage indentation and blank lines correctly to define the boundaries of the block.
* Within a block, the statements are executed in order (until we get on to more complex types of structure, that is)
* Functions may have inputs (‘parameters’) passed into them
* Two of the most common forms of input are numbers and ‘strings’ of characters; there are others
* What ‘global variables’ are and why we want to keep them to a minimum
* How to access the canvas from within code
* The role of the 2D rendering engine (the CanvasRenderingContext2D or ‘renderer’ for short)
* How to instruct the renderer to draw an outline rectangle and a filled rectangle
* How to keep styling separate from the HTML (using the .css file)
* How to get some HTML to be rendered alongside previous HTML instead of undeneath (float)

## What next?

Now that we have the ability to draw an outline square filled with a colour, we want to be able to draw smaller squares within the board.

# 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

TODO: Should we really introduce a namespace to App, just for consistency. It would make sense. Then talk generally about namespaces and importing. Do we really want to do importing, or just qualify the names (as we do with drawing).

model.ts

namespace model {

export class Square {

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

}

}

}

drawing.ts

namespace drawing {

export function drawSquare(sq: model.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);

}

}

app.ts

const squareSide: number = 60;

//Global variables

var canvas: HTMLCanvasElement;

var renderer: CanvasRenderingContext2D;

window.onload = function () {

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

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

var sq = new model.Square(0, 0);

drawing.drawSquare(sq, renderer);

sq = new model.Square(0, 1);

drawing.drawSquare(sq, renderer);

var sq = new model.Square(1, 0);

drawing.drawSquare(sq, renderer);

}

Run, get error, containing this phrase:

JavaScript runtime error: 'model' is undefined

Need to add the **.js** version of these two new files into the Index.html

Note that the order is important (try changing the order and running. Can you figure out why the error is occuring if the order is wrong?

<head>

<meta charset="utf-8" />

<title>My board game</title>

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

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

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

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

</head>

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* Object-oriented programming
* The difference between a class and and instance
* An object’s is a special function for creating an instance of class
* A constructor may have ‘parameters’ that require you to provide the minimum pieces of information needed to create a new instance
* The code for an application may be split across many files
* You can define each class or function in a separate file or have multiple classes/functions in one file
* Namespaces are used to help break up the code into separate chunks.
* You can have two different functions or classes with the same name, and as long they are in different namespaces
* If you reference a class or function from another namespace, you need to ‘qualify’ its name
* How to write functions, with parameters if needed
* The advantage, in TypeScript, of specifying the type of a parameter passed into a function
* How to call a function and pass it parameters.
* Local variables inside a function

## What next?

Now that we can draw individual squares the next step is to draw the whole board. But it would be nice to do that without having to specify the location of each of the 64 squares individually. Fortunately, doing the same thing repeatedly, or a similar thing with predictable differences, is something that program code is very good at.

# Create a whole board

You will need to stop the application running for this next change.

First, install LoDash & . Explain NuGet packages. See the script added.

Explain loDash

As with your own files you will need to add a reference

### Board object – in model.ts

/// <reference path="../Scripts/typings/lodash/lodash.d.ts" />

namespace model {

export class Square {

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

}

}

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);

}

}

}

Note: added new class *inside the namespace*

### initialise board within onload (app.ts)

const squareSide: number = 60;

//Global variables

var canvas: HTMLCanvasElement;

var renderer: CanvasRenderingContext2D;

var board: model.Board;

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);~~

~~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);~~

board = new model.Board();

drawing.drawBoard(board, renderer);

}

### drawBoard function (in drawing.ts)

Note: added new function *inside the namespace*

namespace drawing {

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);

}

export function drawBoard(board: model.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);

}

}

}

}

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* An array holds multiple elements of a specific type, which may then be accessed by their ‘index’ number. It is one of several forms of object ‘collections’.
* An array may be set up with some or all of its values, or you may declare it as an empty array and then add the values
* Programming ‘loops’ repeat the same section of code multiple times, typically varying some parameter(s) each time
* You can create one loop inside another. One use of this is to process a two-dimensional array
* You can push a new element onto the end of an array without having to know the correct index
* Applications are typically made up of custom-written code and re-usable libraries of pre-written code
* An easy way to gain access to such libraries is via the NuGet package manager
* The LoDash library provides powerful functions for manipulating arrays (and other collections) including finding matching objects
* Some functions require you to pass in another function as a parameter. This is known as ‘functional programming’.

## What next?

Now that we can draw the whole board, we would like to be able to draw some pieces on it - black and white. We can start with the first four pieces from which the game proceeds as these are always placed in the same positions each game.

# Set up the starting position

### new constant PieceRadius (in App, underneath the )

const squareSide: number = 60;

const pieceRadius = 25;

### New enum (in models, within namespace (link))

export enum Side { black, white }

### Function to get drawing colour from Side value

In drawing, within namespace,

export function getColourForSide(side: model.Side): string {

if (side == model.Side.black) {

return 'black';

} else {

return 'white';

}

}

### Modify Square class to know if occupied

export class Square {

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

this.occupiedBy = null;

}

public occupiedBy: Side;

}

### new drawPiece function in drawing.ts, within namespace

export function drawPiece(

sq: model.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

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));

}

}

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;

}

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* A constant is a name for a value or object instance that never changes once the program has started
* Global constants don’t have the same liability as global variables
* It is good practice to keep all your global constants together
* An enum is a type that can only have one of a defined set of names values
* Thinking of the best names for things is sometimes one of the hardest parts of programming, and if necessary you should keep changing the name to make it clearer
* Some functions return a value (or object)
* As with parameters (the inputs to a function), TypeScript allows you to specify the type of the return (the output) and this is advantageous
* The conditional operator (‘if’ statement) allows you to execute different instructions based on different inputs or intermediate results.
* Null is a special value indicating that a variable does not have anyting ‘in it’.
* As in ordinary mathematics, multiplication (‘\*’) takes precedence over addition (‘+’), but this may be overridden by adding brackets – ‘(‘ and ‘)’
* ‘Math’ is a library of mathematical functions, but, unlike LoDash, this one comes with the JavaScript language
* ‘Arc’ is for drawing part, or whole, circles (or elipses)

## What next?

We’d like the user to be able to places pieces in new squares, but first we need a mechanism for the user to be able to specify a square of interest. In other words we need a moveable ‘cursor’. We’ll start by drawing a cursor in a specific position.

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

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

In app.ts, a new global variable

//Global variables

var canvas: HTMLCanvasElement;

var renderer: CanvasRenderingContext2D;

var board: Board;

var cursorLocation: Square;

Initilise this in onload

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);

}

### Test at this stage

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

Put in drawing. Recall: where do we insert new functions?

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);

}

}

Test again

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* The DRY principle: Don’t Repeat Yourself in code. When you have two functions, or just blocks of statements that do very similar things, try to turn them into a single function that can be re-used in multiple contexts
* This is not just to save you time - it’s so that you when requirements change you don’t have the risk of updating one and overlooking another implementation

## What next?

Now that we can draw a cursor in a specified location, we want to be able to move the cursor around by using the arrow keys on the keyboard, say.

# Move the cursor around using the arrow keys

## MoveCursorBy in App

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

(don’t need the drawBoard now, because it is included within the moveCursorBy

window.onload = function () {

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

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

board = new Board();

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

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.

JavaScript runtime error: Unable to get property 'col' of undefined or null reference

Explain: debug (we flagged this in an earlier comment on getSquare)

So …

## Key the cursor within bounds

On Board object

public 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 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

## Add user instructions

In Index.html:

<body>

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

<h1>Reversi Game</h1>

<h3>Instructions</h3>

<p>Use the four arrow keys to position the cursor.</p>

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* How to detect and react to keyboard activity when the programming is running (window.onkeydown)
* How to override the default behaviour of the browser in relation to certain keys
* How to use a switch statement as an alternative to multiple if statements, where we are interested in a specific set of values
* The break operator specifies that there is no no need (and it may be harmful) to continue evaluating the various cases.
* A ‘null reference’ error occurs when a program tries to invoke a method on an object that does not exist.
* How to use a function to keep a value within defined limits.

## What next?

Now that we can move the cursor to any square, we’d like to be able to place a piece, black or white, in that square.

# Place a piece on the board

## placePiece function in App

function placePiece(side: model.Side): void {

cursorLocation.occupiedBy = 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(model.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: model.Side): void {

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

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(model.Side.black);

break;

case 87: // 'w'

placePiece(model.Side.white);

break;

}

}

## Add user instructions

In Index.html:

<body>

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

<h1>Reversi Game</h1>

<h3>Instructions</h3>

<p>Use the four arrow keys to position the cursor.</p>

<p>Use the ‘b’ and ‘w’ keys to place pieces.</p>

TODO: Need to capture the turquoise colouring of namespaces/classes

Test it out.

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* void is just a more definite way of saying that a function does not return any value
* It is important to check that the screen is updated *after* all changes have taken place

## What next?

Currently we can chose to place either a black or white piece, but in a real game the turns alternate, so we’d like the program to manage it. Then we can use just one key - ‘Enter’ say – to place the piece for whichever side whose turn it is.

# Alternate turns

## Add a GameManager class

**Tip: to reformat your code (and get the indents right) use ctrl-k ctrl-d (only works properly if your code is compiling)**

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

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 {

//Place new piece

sq.occupiedBy = this.turn;

//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:

//Global variables

var canvas: HTMLCanvasElement;

var renderer: CanvasRenderingContext2D;

var cursorLocation: model.Square;

var board: model.Board;

var game: model.GameManager;

window.onload = function () {

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

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

board = new Board();

game = new model.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);

break;

}

}

Delete the now redundant placePiece function in app

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

~~cursorLocation.occupiedBy = side;~~

~~moveCursorBy(0, 0);~~

~~}~~

## Modify user instructions

In Index.html:

<body>

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

<h1>Reversi Game</h1>

<h3>Instructions</h3>

<p>Use the four arrow keys to position the cursor.</p>

~~<p>Use the ‘b’ and ‘w’ keys to place pieces.</p>~~

<p>Use the ‘Enter’ key to place a piece.<p>

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* Though there may be many instances of a single class, some classes may create just a single instance. (These are sometimes referred to as ‘singletons’)
* An object can correspond to a concrete object in the real world (such as a board, or a piece, or a person)
* An object may also model an imaginary person that might be assigned a specific set of responsibilities. This is known as ‘anthropomorphism’
* As an application development progresses , new stories will very commonly change previous requirements or make them completed redundant
* All code should be thought of as disposable. Getting too precious about deleting code leads to bad code.

## What’s next?

We’d like the system to notify us of who’s turn it is.

# Update the status

TODO: Maybe re-order this. Do the HTML and the updatetext function first, then the content.

## Add method and property within the GameManager class

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 {

//Place new piece

sq.occupiedBy = this.turn;

//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: model.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);

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* How to add specific places on a web-page where text may be inserted from the program, dynamically
* How to access such locations, and update the contents, from program code

## What next?

We can now place black and white pieces on the board, in turn. But no rules are enforced - we could place pieces straight in the corners, for example. Let’s add some rules, starting with the simplest ones first.

# 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 (notice the brace at the end) - in GameManager class:

public placePiece(sq: Square): void {

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

//Place new piece

sq.occupiedBy = this.turn;

//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);

}

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* Rules are implemented as methods and functions, like the rest of programming, but it is good practice to give them names that describe the rule.
* The ‘ternay operator’ (‘?’) is a more succinct way than using an if operator to write certain, simple, conditional statements

## What next

The next simplest rule to add would be to say that for a valid move the piece must be adjacent (orthogonally or diagonally) to at least one piece of the opposite side.

# 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

In the Board class

//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);

}

**Tip:**  line breaks - just to make code more readable

Compile – **Ctlr-Shift-b** (if you are still running, hit Ok in response to Q Do you want to stop debugging)

Note the compile errors in the window and click on each one to see where it is.

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)) {

//Place new piece

sq.occupiedBy = this.turn;

//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) *using new ‘ternary operatory’*. This best goes in the model namespace but not inside any of the classes.

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)) {

//Place new piece

sq.occupiedBy = this.turn;

//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 {

~~var oppositeSide: Side;~~

~~if (side == Side.black) {~~

~~oppositeSide = Side.white~~

~~} else {~~

~~oppositeSide = Side.black;~~

~~}~~

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

}

Test this

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* The LoDash \_.some function finds if there are any members in a collection that match given criteria
* Forgettint he ‘boundary’ or ‘edge’ cases are a common source of programming error
* Simple rules and algorithms are best written straight into code, but more complex ones should be sketched out carefully before starting coding
* Similar logic that appears in two different places should be factored out into a single, generalised function that both can delegate to.

## What next?

We’ve now implemented the simple rules, so we need to get onto the meat of the problem: identifying pieces of the opposite side that are, or would be, captured by a move.

# 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. (new method on Board class):

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

var squares: Square[] = [];

for (var i = 1; i <= 7; i++) { //Can only be maximum of 7 steps 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 drawing the outline of those squares in blue, by adding (in App.ts):

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.squaresFrom(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.ts):

export enum Direction { north, east }

then change squaresFrom function to:

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

var squares: Square[] = [];

for (var i = 1; i <= 7; i++) { //Can only be maximum of 7 steps 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;

}

This will give you a compile error that will need fixing

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.squaresFrom(cursorLocation, model.Direction.north);

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

}

Test again that it still works the same way.

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;

}

modify the moveCursorBy function so that it now calls capturedSquares:

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);

}

Test this sout (you’ll have to move the cursor around until you find a piece to the north)

Then change the penultimate line in the moveCursorBy to specify east instead of north and test that this works:

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

Now a new method (in Board) to cover all of the directions (currently north and east only - we’ll add the others shortly)

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;

}

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));

}

*Remember that it is currently only looking for squares captured to the north and east - you’ll need to position the cursor to find them.*

Now sdd 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:

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

var squares: Square[] = [];

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

var sq: Square;

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;

}

if (sq != undefined) {

squares.push(sq);

}

}

return squares;

}

test this. Should now see all potentially captured squares highlighted in blue.

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* Some stories can’t be broken into small pieces that each add value to the user, but they can and should still be broken into smaller stages that can be designed, implemented, and tested, by the programmer.
* The LoDash \_.forEach function is a very succinct way to write simple loops that apply the same function to each object in a given collection.
* When a function has to handle multiple cases, implement and test the cases one at a time
* Changing the ‘signature’ (name and/or parameters) of a function may break code that already calls it, but in a language like TypeScript the compiler will show you where you need to make changes to match the new signature

## What next?

We can now highlight the squares that would be captured by a new placement. Now we want to use that functionality to add two new pieces of value:

* A placement is no valid unless *at least one* piece would be captured
* When the placement is made, the captured pieces should be flipped automatically

It turns out that having done the hard work in this story, those next two items are so simple to implement that we will combine them into one story.

# 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.

TODO: These three rules could even ven re-written closer to english e.g.

return sq.isEmpty() &&

sq.squareIsAdjacentToPieceOf(oppositeSideTo(side)) &&

sq.wouldCaptureAtLeastOnePieceIfPlacing(side);

As well as some renaming, this involves moving the methods from Board on to Square, which in turn means that Square must be created with a reference to its Board. It can then delegate back.

## Flip the captured pieces as part of the placePiece function in GameManager

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);

//Set the next turn

this.turn = oppositeSideTo(this.turn);

this.updateStatus();

}

}

Test

## Fix bug

note that squares may be highlighted blue even if move is invalid. TODO: Need example

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

if (board.wouldBeValidMove(cursorLocation, game.turn)) {

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

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

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

} else {

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

}

}

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* A good aim is to design functions and methods so that, when called, the syntax reads as close to plain English as possible
* Some bugs don’t show up initially. This highlights the importance of thorough testing.

## What next?

Now that we are flipping pieces correctly, it would be nice to have a continuously updated count of the number of pieces of each side.

# 15: Count pieces

## Add function to board for counting pieces of either side

public countPieces(side: Side): number {

return \_.filter(this.squares, sq => sq.occupiedBy == side).length;

}

## Add placeholders into index.html

<body>

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

<h1>Reversi Game</h1>

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

<p>Black count: <span id="black"></span></p>

<p>White count: <span id="white"></span></p>

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

</body>

## Add to the updateText function in drawing.ts

export function updateText(game: GameManager) {

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

document.getElementById("black").innerHTML = board.countPieces(mode.Side.black).toString();

document.getElementById("white").innerHTML = board.countPieces(model.Side.white).toString();

}

Test

## What have we learned?

Make sure you’ve understood all these points before moving on to the next story.

* The LoDash \_.filter function allows us to filter a larger collection down to a smaller once, containing only objects that match certain criteria

## What next?

With the ability to count pieces we can now check when the board is full, and determine which side has one.

# 16: End game and show result when the board is full

## Add a gameOver flag to the GameManager

public gameOver: boolean;

## Change the updateStatus method on GameManager

Don’t miss the new closing brace at the bottom!

public updateStatus(): void {

if (this.gameOver) {

var black = board.countPieces(Side.black);

var white = board.countPieces(Side.white);

if (white > black) {

this.status = 'GAME OVER. White has won!';

} else if (black > white) {

this.status = 'GAME OVER. Black has won!';

} else {

this.status = 'GAME OVER. A draw!';

}

} else {

switch (this.turn) {

case Side.black:

this.status = 'Black to play';

break;

case Side.white:

this.status = 'White to play';

break;

}

}

}

## Test for board full

On Board

public isFull(): boolean {

return \_.filter(this.squares, sq => sq.occupiedBy).length == 64;

}

Test this after placing new piece

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);

//Set the next turn

this.turn = oppositeSideTo(this.turn);

//test for game over

if (board.isFull()) {

this.gameOver;

}

this.updateStatus();

}

}

Test: should now work

## Use the gameOver flag to prevent further play in App

add new test at the top of the inkeydown function (in app.ts):

window.onkeydown = function (ke: KeyboardEvent) {

ke.preventDefault();

if (game.gameOver) return; //Don’t act on any key

switch (ke.keyCode) {

…

## Update user instructions

<h3>Instructions</h3>

<p>Use the four arrow keys to position the cursor.</p>

<p>Use the Enter key to place your piece.</p>

<p>To start a new game click the Refresh icon (or hit F5) on the browser</p>

How to test this? By setting up a near-complete board?

## What have we learned?

We implemented this story without having to learn any new concepts, just by applying techniques that we learned in previous ones to new situations. This is a good sign!

## What next?

It is actually quite rare that players can simply move alternately until all 64 squares are filled. Usually, towards the end of the game one player will be unable to move. It isn’t the end of the game, though, unless both players are unable to move. So we need to allow one side to ‘skip’ their turn and allow the other side to play again.

# 17: Allow player to skip their turn if they can’t move

## Add SkipTurn function to GameManager

public skipTurn(): void {

this.turn = oppositeSideTo(this.turn);

this.updateStatus();

}

## Allow user to skip turn with the ‘Esc’ key

Add a new case into the onKeyDown function (in App)

case 27: //Esc - Skip turn

game.skipTurn();

drawing.updateText(game);

moveCursorBy(0, 0);

break;

Why do we need the moveCursorBy(0,0) - try without!

## Keep track of whether each side has skipped turn & end game if both skip

In GameManager

lastTurnWasSkipped: boolean;

public skipTurn(): void {

~~this.turn = oppositeSideTo(this.turn);~~

if (this.lastTurnWasSkipped) {

this.gameOver = true;

} else {

this.turn = oppositeSideTo(this.turn);

this.lastTurnWasSkipped = true;

}

this.updateStatus();

}

public placePiece(sq: Square): void {

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

//Reset

this.lastTurnWasSkipped = false;

//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);

//Set the next turn

this.turn = oppositeSideTo(this.turn);

//test for game over

if (board.isFull()) {

this.gameOver;

}

this.updateStatus();

}

}

## Update user instructions in Index.html

<h3>Instructions</h3>

<p>Use the four arrow keys to position the cursor.</p>

<p>Use the Enter key to place your piece.</p>

<p>Use the Esc (Escape) key to skip turn, if you cannot move.</p>

## What have we learned?

* How to write a complete application!]

## What next?

This is the end of the first version of the application, but there are plenty more major improvements and extensions that we could make, including:

* Managing a mini-tournament between multiple players, keeping track of scores, and determining who should play next, based on whether it is a ‘pool’ or ‘knockout’, or ‘best of x’, tournament
* Creating an ‘automated player’ so that one human player can play against the computer
* Creating automated players of increasing sophistication and running them against each other in a fully automated match
* Running a fully automated tournament, possibly over a large number of games, to determine the relative peformance of different automated-players

# 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