# Once we’ve got started do all the rest as ‘stories’, with a brief description and/or a sketch of what is required.

Within this, have separate boxes for introducing new concepts and examples.

In each case try to get student to figure out all or part of the answer, before revealing code.

Programming principles (coloured box).

Explanation of specific code, where this is given to them.

Coloured boxes for tools & techniques (e.g. debugging hints, testing)

Finish with pointer to a complete version.

# Lesson 1 : Create a new project and create introduction web pages

* Open VS
* Create new project
* Add (or edit) Index.html (the home page)
* Run
* Edit welcome title and add some instructions
* Add some styling, inline and then with .css
* Create a link called play to take you to a second page

# Lesson2: Add a drawing canvas

Add a canvas of specified dimensions  
Add ref to userInterface.js

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8" />

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

</head>

<body>

<h1>Reversi</h1>

<canvas width="1200" height="600"></canvas>

</body>

</html>

userInterface.js

window.onload = function () {

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

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

renderer.strokeStyle = 'blue';

renderer.strokeRect(10, 10, 500, 500);

}

Explain: const - is a holder with a name, but its contents don’t change.

use more of them for boardOriginX, Y & squareSize.

Explain functions/methods

Experiment with adding more rectangles of different positions, sizes, colours and fills. Point out that in procedural programming, statements are executed in order.

Introduce debugging and stepping.

Eventually start to build a board from squares, but:

* Lots of repetition
* What if we wanted to now change the size of the board?

Wrtiting a more general function for drawing a square of a given size, and working out the start coordinates also. So that we can end up writing:

module Reversi {

const squareSize = 60;

window.onload = function () {

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

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

drawBoard(renderer);

drawSquare(0, 0, renderer);

drawSquare(0, 1, renderer);

drawSquare(1, 0, renderer);

drawSquare(1, 1, renderer);

}

function drawBoard(renderer: CanvasRenderingContext2D) {

renderer.fillStyle = 'green'

renderer.fillRect(0, 0, squareSize \* 8, squareSize\*8);

}

function drawSquare(x: number, y: number, renderer: CanvasRenderingContext2D) {

renderer.strokeStyle = 'black';

renderer.strokeRect(x\*squareSize, y\*squareSize, squareSize, squareSize);

}

}

Next we can learn the idea of a repeating loop for x and y. Much of programming consists of loops.

const squareSize = 40;

window.onload = function () {

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

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

drawBoard(renderer);

for (let x: number = 0; x <= 7; x++) {

for (let y: number = 0; y <= 7; y++) {

drawSquare(x, y, renderer);

}

}

}

function drawBoard(renderer: CanvasRenderingContext2D) {

renderer.fillStyle = 'green'

renderer.fillRect(0, 0, squareSize \* 8, squareSize\*8);

}

function drawSquare(x: number, y: number, renderer: CanvasRenderingContext2D) {

renderer.strokeStyle = 'black';

renderer.strokeRect(x\*squareSize, y\*squareSize, squareSize, squareSize);

}

}

Draw a piece on a square.

* Draw a filled black circle
* Give it square coordinates and have it work out the canvas coordinates
* Pass in the required colour

module Reversi {

const squareSide = 60;

const pieceRadius = 25;

window.onload = function () {

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

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

drawBoard(renderer);

drawPiece(3, 3, 'black', renderer);

drawPiece(4, 4, 'black', renderer);

drawPiece(3, 4, 'white', renderer);

drawPiece(4, 3, 'white', renderer);

}

function drawBoard(renderer: CanvasRenderingContext2D) {

renderer.fillStyle = 'green'

renderer.fillRect(0, 0, squareSide \* 8, squareSide \* 8);

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

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

drawSquare(x, y, renderer);

}

}

}

function drawSquare(x: number, y: number, renderer: CanvasRenderingContext2D) {

renderer.strokeStyle = 'black';

renderer.strokeRect(x\*squareSide, y\*squareSide, squareSide, squareSide);

}

function drawPiece(x: number, y: number, colour: string, renderer: CanvasRenderingContext2D) {

renderer.fillStyle = colour;

renderer.beginPath();

renderer.arc(x \* squareSide + squareSide / 2, y \* squareSide + squareSide / 2, pieceRadius,0, 2 \* Math.PI);

renderer.fill();

}

}

Building some user interaction

Start with a simple function to clear and re-draw the board

**Introduce conditional**

**Note:** canvas and renderer moved out into global variables - should do at the beginning.

module Reversi {

var canvas: HTMLCanvasElement;

var renderer: CanvasRenderingContext2D;

const squareSide = 60;

const pieceRadius = 25;

window.onload = function () {

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

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

drawBoard();

}

window.onkeypress = function (ke: KeyboardEvent) {

if (ke.keyCode === 99) {

clearBoard();

}

}

function clearBoard() {

const rect = canvas.getBoundingClientRect();

renderer.clearRect(0, 0, rect.width, rect.height);

drawBoard();

}

function drawBoard() {

renderer.fillStyle = 'green'

renderer.fillRect(0, 0, squareSide \* 8, squareSide \* 8);

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

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

drawSquare(x, y);

}

}

}

function drawSquare(x: number, y: number) {

renderer.strokeStyle = 'black';

renderer.strokeRect(x\*squareSide, y\*squareSide, squareSide, squareSide);

}

function drawPiece(x: number, y: number, colour: string) {

renderer.fillStyle = colour;

renderer.beginPath();

renderer.arc(x \* squareSide + squareSide / 2, y \* squareSide + squareSide / 2, pieceRadius,0, 2 \* Math.PI);

renderer.fill();

}

}

**Move the four statements that draw the starting four pieces into a separate function called drawStartPosition.**

**NO: move this later, after doing cursor**

Add to the window.onKeyPress function so that hitting the ‘s’ key will call this new function.

function drawStartPosition() {

drawPiece(3, 3, 'black');

drawPiece(4, 4, 'black');

drawPiece(3, 4, 'white');

drawPiece(4, 3, 'white');

}

window.onkeypress = function (ke: KeyboardEvent) {

if (ke.keyCode === 99) {

clearBoard();

}

if (ke.keyCode === 115) {

drawStartPosition();

}

}

**Add a cursor that can be moved around the board**

Maybe only add the re-draw in black as a second step

Also, re-factor the colour into drawSquare only as a second step.

Maybe make colour an optional parameter

Introduce refactoring, and correcting compile errors.

function positionCursor(x: number, y: number) {

//First remove the highlight from the current cursor

drawSquare(cursorX, cursorY, 'black');

cursorX = x;

cursorY = y;

drawSquare(cursorX, cursorY, 'red');

}

with global variables:

var cursorX: number;

var cursorY: number;

and set an initial cursor:

positionCursor(0, 0);

* Allow user to point at a square with the mouse (or by touch?)
* Then select black or white to draw a piece there
* Another key to clear the board.
* (Possibly, introduce a Switch statement)

window.onkeypress = function (ke: KeyboardEvent) {

switch (ke.keyCode) {

case 99:

drawSquareBackground(cursorX, cursorY);

drawSquareOutline(cursorX, cursorY, 'black');

break;

case 115:

clearBoard();

drawStartPosition();

break;

case 98:

drawPiece(cursorX, cursorY, 'black');

break;

case 119:

drawPiece(cursorX, cursorY, 'white');

break;

}

}

--------------------------------------- Sprint 2

Two questions:

* How to prevent placing piece onto already occupied square?
* How to do a B/W flip?

We need to find the current status. Could check the colour of the pixel at the centre of the square, but this is ugly.

Need to introduce the idea of a model and separate out the logic of the model from the drawing on the screen. Fundamental principle of ‘separation of concerns’.

Several approaches to modelling, but best and most common (now) is object-oriented design.

We construct software elements that, in the first instance, correspond to ‘things’ in the application domain. Not just physical things, but important concepts. A good starting point is to look for the nouns.

Exercise: write down some nouns from the domain of the board game.

* Board, Square, Corner
* Piece
* Player, Rule, Referree
* Game

There is no simple deterministic process. All are candidates. It takes experience and/or experimentation to find out which are the best objects.

Could do this in JavaScript, but it has a rather unusual approach for objects. So we are going to switch to another language: TypeScript.

Add a TypeScript file, with a module called Model, and a empty class definitions for some of the nouns.

Show how this compiles to a JavaScript file - which is fortunate because that’s the only language we can actually run in the browser. That’s why it is the .js file that we need to add into our Html page.

Classes and instances. So we might have one instance of the class Board (unless we are playing more than one game at a time), 64 instances of Square. Up to 64 instances of Piece. 2 instances of Player and so on. Think of class as the template, or pattern, or cookie cutter even.

Show how in another piece of code we can create new instances and attach them to names.

So far we have some empty objects - but what goes in them?

Know and Know-how-to responsibilities.

# Lesson 2: