University of Sussex
Informatics

Spring 2022

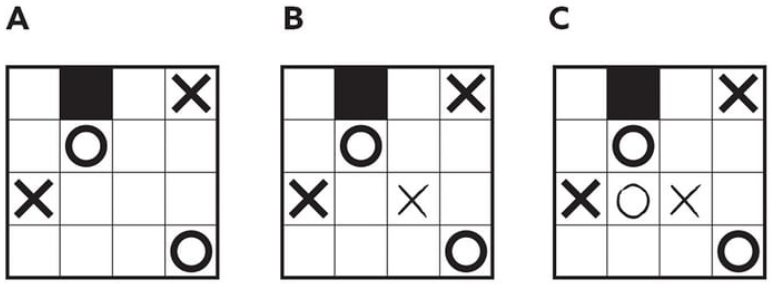
Further Programming
Lab 5

Marupeke Game and Peer Review

(covers Lectures 2-6)

In this lab, you will develop the basics of a textual version of a Marupeke puzzle game. Marupeke is a Japanese puzzle, best described by Alex Bellos in the Guardian[1].

> Marupeke was invented in 2009 by Naoki Inaba, who is Japan's – and the world's –most prolific inventor of grid logic puzzles. "This puzzle is like a simple equation," he says. "In a simple equation two things lead to a third." In other words, when you add, subtract, multiply or divide two numbers, you get a third number. "I wanted to express this idea as simply as possible." It is also like playing noughts and crosses against yourself.
>
> **The rules**: Fill in each empty cell with either an O or an X, so that no more than two *consecutive cells*, either horizontally, vertically or diagonally, contain the same symbol.

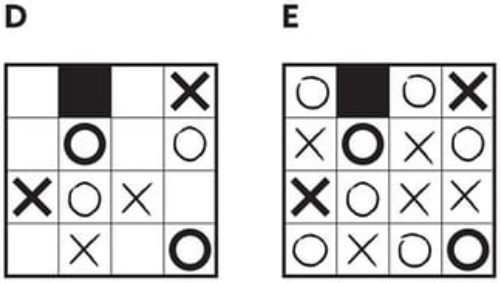


A: A possible starting grid (of size 4×4 in this case).

B: The cell between the two Os must contain an X, otherwise there is a diagonal run of three Os, which is forbidden.

[1] `https://www.theguardian.com/technology/2017/oct/10/puzzle-masters-japan-sudoku-tokyo`

C: The cell between the Xs on the third row must contain an O, otherwise there is a horizontal run of three Xs, which again is forbidden.



D: The cell on the bottom row beneath the Os in the second column must contain an X, otherwise there is a vertical run of three Os. Now you're on a roll. This new X threatens a diagonal run of three Xs, so there must be an O in the final cell of the second row. Continue using these strategies to complete the grid.

E: The completed grid. Note that it is possible for three (or more in larger grids) crosses (or naughts, resp.) to appear in a vetical, horizontal, or diagonal, resp. For example, in (E) you see three Xs in the third horizontal row. The important thing is that these Xs are *not* consecutive.

In this computerised version, you will display a grid of tiles, some of which will be solid. You will then allow the user to mark a tile with an X or a O, and to check whether the current grid position is legal. In this assignment, you will first develop a textual version, and then a full graphical interface.
Note that the grid size in the example above is four. But the rules work for larger grid sizes too. When you develop your game you won't fix the grid size. It will be decided by the player.
The aim of Marupeke is to explore a grid of tiles, applying logic to discover how to mark the grid with noughts and crosses such that the rules of Marupeke are obeyed. We therefore start by designing a simple data structure to hold the solid squares and which squares have been marked with a nought or a cross.
The obvious data structure to use to represent the cells in the grid is a 2-dimensional array of chars, where we use 'O' and 'X' for noughts and crosses, and '#' for solid squares. If we declare:

```
char[][] grid = new char[10][10];
```

then we have a 10 by 10 grid.

Conventionally we think of the first co-ordinate as the row and the second as the column, so we think of grid [0][0] as the top-left square; grid [0][1] as the square just next to it; grid [1][0] as the first square in the second row; and so on, until grid [9][9] is the bottom-right. As well as the location of the naughts and crosses, we need to determine whether a given square is editable – the starting grid will have filled out squares and noughts and crosses which cannot be removed. We will therefore use a 2-dimensional array of booleans, declared as:

```
boolean[][] editable = new boolean[10][10];
```

1 Coding up and printing a Marupeke grid

Create a MarupekeGrid class with fields to store the grid and whether a square is editable, as above. Create a JUnit4 test class to develop the following code using TDD. For each method, first create the tests that would fail against a correctly working method, write method, fail, correct the test, pass. Iterate until the method has the full behaviour intended. Don't forget to test the constructor's correct behaviour. Note that the `set` methods below are intended to be used in two phases – first to initialise the grid with the starting state, so that after setting the value of the square, it should no longer be editable, and then in the user playing phase, where the user can change and remove marked squares as required. Write the following methods for this class:

1. A constructor which takes as parameter the size (width as well as length) of the grid. The constructor should create the initial arrays, up to a maximum length of 10. If the size argument is larger than 10 create a grid of size 10, if the size is smaller than 3, create a grid of size 3.

2. A setSolid method with x and y parameters to set the given grid square solid. This call should also set the square to be uneditable. It should return true if the square was originally editable, and false if the square was uneditable.

3. setX and setO methods, which should have x and y parameters for the square in the grid to set, and a canEdit boolean parameter to change the editable state of the square. It should return true if the square was originally editable, and false if the square was uneditable.

4. Create a static method to generate a random puzzle, with the following signature:

```
public static MarupekeGrid randomPuzzle(int size,
                                        int numFill,
                                        int numX,
                                        int numO);
```

   The squares to be filled (that means set to be solid), X or O in the generated puzzle should be randomly chosen to match the required numbers specified in numFill, numX and numO, respectively. If the sum of squares still to be completed is greater than $size^2$, then you should return null. Note that these puzzles may be illegal, or uncompletable.

5. Create a new toString method, which returns a string representing the grid, with the '#' character for a solid square, a 'O' for a nought and a 'X' for a cross. To identify empty squares, use the '_' (underscore) symbol. For instance, for a 4x4 Marupeke grid as in Figure A above, I would expect output like:

```
_#_X
_O__
X___
___O
```

You may ignore any `ArrayIndexOutOfBoundsException` that might occur when an array (here the MarupekeGrid) is indexed wrongly. We will cover defensive programming later. So your tests do *only have to check for legal access to the grid.*

2 Peer Review

1. Save your IntelliJ project as zip file by using menu item `File` → `Export` → `Project to Zip File`.

2. Exchange this file with another project in your lab.

3. Unzip file given by this other student and then open the directory in IntelliJ using `File` → `Open`.

4. Inspect the tests and code you find. Try to run code and tests. Is all working well? Add some comments how the code could be improved or fixed.

5. How readable and well structured is the code (check the slides for Lecture 7: Code Quality)? Write some comments how the code could be improved. Also check indentation is decent. You may want to use the `Code` → `Reformat code` menu item. Try to remember the key short cut (as given in the menu item).

6. Exchange the version your helpful additional comments with the other student.