OXO Exercise Briefing

COMSM0086

Dr Simon Lock & Dr Sion Hannuna

Before we begin: Shapes Exercise Recap

A few people asked to see sample solution of Shapes
Simple exercise, but worth spending time reviewing

We won't be looking at every single line of code
A lot of it is fairly straightforward and uninteresting

A few novel features and notable characteristics
Let's focus on just those (in no particular order)...

Overriding and Chaining

ALL 'TwoDimensionalShapes' will have a colour

Specific shapes however have different geometry

Where should we implement 'toString' method ?

```
abstract class TwoDimensionalShape {
  private Colour shapeColour;

  public String toString() {
    return "This shape is " + shapeColour;
  }
}
```

```
class Triangle extends TwoDimensionalShape implements MultiVariantShape {
  public String toString() {
    return "Triangle with sides " + first + "," + second + "," + third + ". " + super.toString();
  }
}
```

Multiple Constructors: With Chaining !

```
class Triangle extends TwoDimensionalShape implements MultiVariantShape {
  private int first;
  private int second;
  private int third;
  private TriangleVariant variant;
  static int population;

  public Triangle(int first, int second, int third) {
    this.first = first;
    this.second = second;
    this.third = third;
    variant = identifyTriangleVariant();
    population++;
  }

  public Triangle(int first, int second, int third, Colour colour) {
    this(first, second, third);
    super.setColour(colour);
  }
```

Triangle Variants

We already have a method to return the longest side
It's sometimes useful to know the shortest two sides
For example, when checking right-angled triangles:

```
square of hypotenuse == sum of squares of other 2 sides
```

You could add 'getShortestSide' and 'getMedianSide'
OR store sides sorted by length (e.g. in an array)
I chose something different - trying to be clever...

Variant Identification: One line each !

```
int longestSide = getLongestSide(first, second, third);
// sumOfAllSides needs to be long to avoid int overflow problems
long sumOfAllSides = ((long)first) + ((long)second) + ((long)third);
// Check for "bad" variants first (otherwise we might classify it as "good" triangle first)
if((first<=0) || (second<=0) || (third<=0)) return TriangleVariant.ILLEGAL;
// Could have been `sumOfAllSides == longestSide*2` but risks overflowing due to large numbers
else if(sumOfAllSides/2.0 == longestSide) return TriangleVariant.FLAT;
// Could have been `sumOfAllSides < longestSide*2` but risks overflowing due to large numbers
else if(sumOfAllSides/2.0 < longestSide) return TriangleVariant.IMPOSSIBLE;
// Relatively straight-forward (and reliable) to check that all sides are equal
else if((first==second) && (second==third)) return TriangleVariant.EQUILATERAL;
// Isosceles check appears late - triangle might have two identical sides, but still be "bad"
else if((first==second) || (second==third) || (third==first)) return TriangleVariant.ISOSCELES;
// Checking for right-angle triangles is a bit complex, so "farm it out" to a separate method
else if(isRightAngle(first,second,third)) return TriangleVariant.RIGHT;
// Scalenes should be near the bottom because the test for them is a bit "loose"
else if((first!=second) && (second!=third) && (third!=first)) return TriangleVariant.SCALENE;
// Otherwise, we don't know what type it is (theoretically, we shouldn't actually get here !)
else return null;
```

Right-Angles: avoiding square-root & overflow

```
private boolean isRightAngle(int first, int second, int third) {
  int longestSide = getLongestSide(first, second, third);
  long firstSquared = ((long)first) * ((long)first);
  long secondSquared = ((long)second) * ((long)second);
  long thirdSquared = ((long)third) * ((long)third);
  long longestSquared = ((long)longestSide) * ((long)longestSide);
  return ( - longestSquared + firstSquared + secondSquared + thirdSquared) == longestSquared;
}

private int getLongestSide(int a, int b, int c) {
  int largest = a;
  if(b>largest) largest = b;
  if(c>largest) largest = c;
  return largest;
}
```

Moving on to this week's exercise...

OXO

Next two workbooks focus on a single application

Aim is to build a noughts-and-crosses (OXO) game

It's a fairly easy activity (easier than later ones ;o)

This is NOT an assessed exercise

Let's take a look at the key features...

Model View Controller

We will often mention 'Design Patterns' in this unit

First pattern we encounter is 'Model-View-Controller'

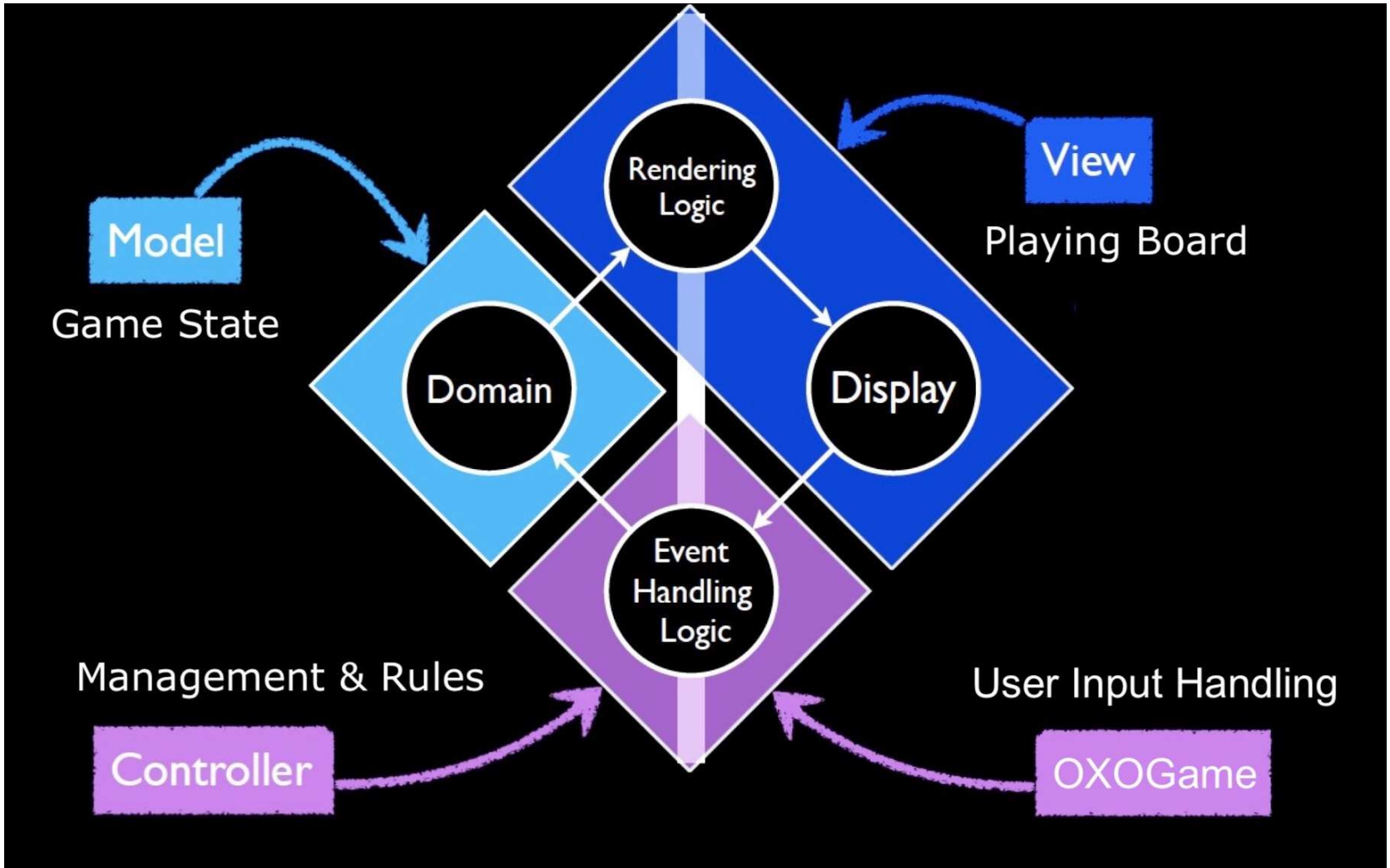
Useful for structuring interactive UI applications

Approach is to split application into 3 'components':

- Model: the core application 'state' data
- View: the bit that the user sees (GUI or text)
- Controller: interprets events and makes decisions

This separation makes change and evolution easier

MVC for OXO


Model
Game State
Rendering Logic
View
Playing Board
Domain
Display
Event Handling Logic
Management & Rules
User Input Handling
Controller
OXOGame

Fill in the gaps

A Maven template project has been provided for you
Already implemented are:

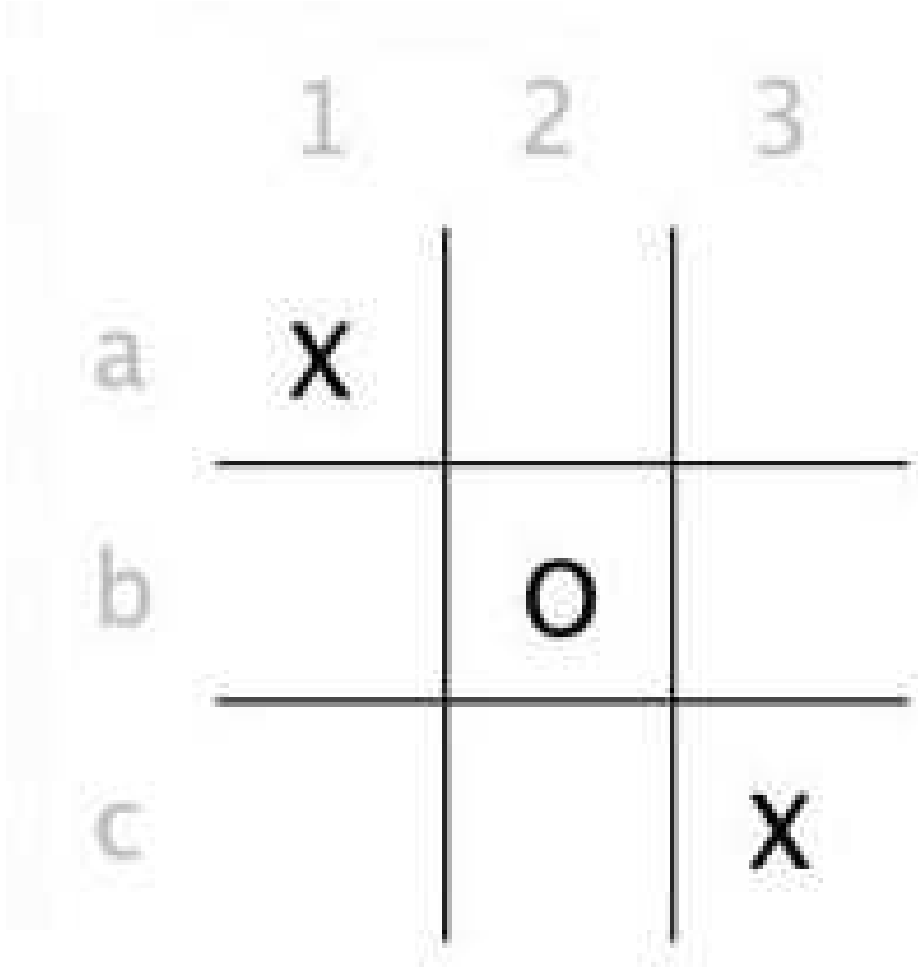
- OXOGame: Main window (that receives user input)
- OXOView: Provides 'Rendering Logic' and 'Display'
- OXOModel: Game State from the OXO domain

OXOController is empty: your task is to complete it !
You must call OXOModel methods to change state
You don't need to interface directly with OXOView
(it monitors OXOModel and updates automatically)

What does Game State 'Model' consist of ?

- The set of players currently playing the game
- The "owner" of each cell in the game grid
- The player whose turn it currently is
- The number in-a-row required to win
- If the game has been drawn
- The winner of the game

OXOView



OXOGame

Key feature of Java: 'Write Once, Run Everywhere'

Main window looks *similar* on different platforms

Playing the Game

Players take it in turns to make their chosen move

Entering cell identifier into OXOGame GUI window

Consists of row letter and column number (e.g. b2)

Inputted cell ID then passed to OXOController via:

```
public void handleIncomingCommand(String command)
```

Your task: interpret identifier & update game state

run-demo

Key Game Features

Win Detection

There is no point playing, if no one actually wins !

It is the Controller's job to detect when a win occurs

It MUST be able to check for wins in ALL directions:

horizontally / vertically / diagonally

Note: win detection MUST work on grids of ANY size

Not just the standard 3x3 board !

The reason for this is...

Dynamic Board Size

It would be nice to alter board size during a game

If it became clear a game was going to be a draw

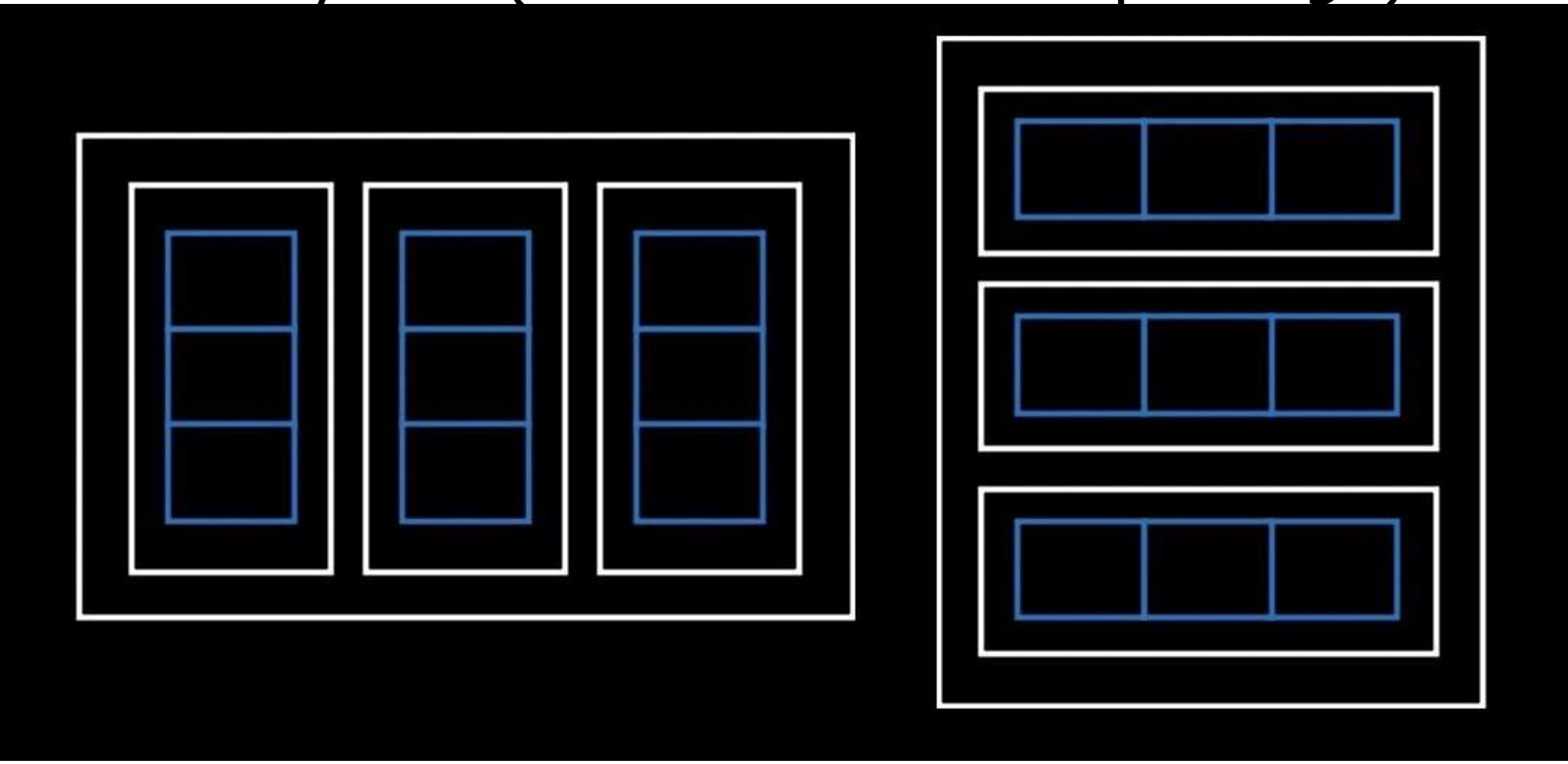
(all blank cells are used but no one has won)

Increasing the board size allows play to continue

This has implications for the storage of game state

You will need to update OXOModel so that it uses...

'ArrayList' (from 'Collections' package)



For a 3x3 game you must instantiate FOUR ArrayLists

Next week

The NEXT workbook will focus on error handling

Namely: what if the user inputs a "bad" cell ID ?

Don't worry about this for the time being

Assume all user inputs are correct/acceptable

We'll also add some additional "fun" extensions !

Automated Testing

Game is complex enough to need automated testing
(Manual testing quickly becomes slow and tedious)

As 'developers' it is YOUR job to write tests scripts
It is not "fun" and it is not particularly "rewarding"
However, it is responsibility of mature professionals

You have been provided with a skeleton test script
Populate this with a comprehensive set of test cases
Focus on testing OXOController (i.e. your code !)

ExampleControllerTests

Warning: There are Lambdas !

Example test script contains some unusual syntax:

```
assert(Exception.class, ()-> handleIncomingCommand());
```

Lambda operator -> is something
we have not actually covered yet
Operates like an inline function
Allows us to pass CODE "into"
an already existing method

More on this NEXT week !

Full Set of Test Cases

We have a (fairly) comprehensive set of test cases
However, we won't let you see these just yet ;o)
It's good to experience writing your own test cases

We'll release OUR test cases near end of exercise
You'll be able to check how well your code is written
(And how comprehensive YOUR tests cases were)

Note regarding 'Serialization'

Some classes we have provided in the project:

- implement the 'Serializable' interface and
- have a private attribute called 'serialVersionUID'

```
public class OXOController implements Serializable {
@Serial private static final long serialVersionUID = 1;
```

GUI applications written in Java require the above so that the application can be stored ('serialized')

We won't be doing any serialization in this exercise
We still need these, or we'll get compiler warnings

To Work !

Random shapes: 'randomInt' class method

```
TwoDimensionalShape[] shapes = new TwoDimensionalShape[100];
for(int i=0; i<shapes.length ;i++) {
    int randomNumber = randomInt(0,3);
    if(randomNumber==0) shapes[i] = new Triangle(randomInt(1,20),randomInt(1,20),randomInt(1,20));
    if(randomNumber==1) shapes[i] = new Circle(randomInt(1,20));
    if(randomNumber==2) shapes[i] = new Rectangle(randomInt(1,20),randomInt(1,20));
}
int loopTriangleCounter = 0;
for(int i=0; i<shapes.length ;i++) {
    if(shapes[i] instanceof Triangle) loopTriangleCounter++;
}
System.out.println("-".repeat(30));
System.out.println("Loop counter population is: " + loopTriangleCounter);
System.out.println("Class variable population is: " + Triangle.getPolulationSize());
```

```
public static int randomInt(int lowerLimit, int upperLimit) {
    return lowerLimit + (int)(Math.random()*(upperLimit-lowerLimit));
}
```