NELSON MANDELA UNIVERSITY

Department of Computing Sciences

Advanced Programming 3.2 (WRPV302)

Assignment List

July 2023

Please read this document carefully and keep it in a safe place — it contains important information that you will need to refer to throughout the module.

Contents

WRPV302 Assignment List

About

This document contains *all* the assignments that you are required to complete this semester. The due dates are provided for each assignment; be sure to submit your assignment in a zipped file on the Learn site *before* the due date and time. You are also required to *peer assess* a selection of assignments submitted by your peers. The rubrics used for assessment can be found on the Learn site.

Remember to complete *all* the Activities, as *all* of them count towards the Class Mark (and Final Mark) for this Module. In other words, not submitting Assignments or completing Peer Assessments is *throwing* marks away. Assignments are designed to prepare you for the formal assessments, so be sure to complete them yourself.

Class Mark[1]	30%	Practical Assignments	25%	Practical Submissions
			5%	Peer Assessment
	70%	Term Assessments	35%	Term 3 Assessment
			35%	Term 4 Assessment

Schedule

The schedule of assignment submissions and peer assessments is given below:

Week	Dates	WRPV302 L: Mon (10:25-11:35) 350101	WRPV302 P: Wed Th (15:45-18:15)
1	24-28 Jul	Introduction to Android	**Start Assignment 1**
2	31 Jul-4 Aug	Android Views & Activities	
3	7-11 Aug	Android Persistence (files & databases) & Exceptions	**Start Assignment 2** *Tu: Submit Assignment 1* *Fr: Peer Assess Assignment 1*
4	14-18 Aug	Android Multiple Views	
5	21-25 Aug	Android Fragments	**Start Assignment 3** *Tu: Submit Assignment 2* **Fr 25: Term 3 Assessment (2-5pm)**
6	28 Aug-1 Sep	Collections	*Mo: Peer Assess Assignment 2*
7	4-6 Sep	Multi-Threading	**We: last day of lectures**
	RECESS: 7 - 13 Sep		

[1] Remember that a minimum of 40% is required to be allowed to write the Exams.

Week	Dates	WRPV302 L: Mon (10:25-11:35) 350101	WRPV302 P: Wed Th (15:45-18:15)
8	14-15 Sep	**No classes Mo-We**	**Th: lectures start** **Start Assignment 4** *Th: Submit Assignment 3*
9	18-22 Sep	Threads & Animations	*Mo: Peer Assess Assignment 3*
10	25-29 Sep	**No lectures Mo**	**Mo 25: Public Holiday**
11	2-6 Oct	Networking I	**Start Assignment 5** *Tu: Submit Assignment 4* *Fr: Peer Assess Assignment 4*
12	9-13 Oct	Networking II	**Fr 13: Term 4 Assessment (2-5pm)**
13	16-20 Oct	Networking III	
14	23-27 Oct	Multi-media	*Tu: Submit Assignment 5* *Fr: Peer Assess Assignment 5* **Fr 27: Sick Assessment (9am-12pm)**
15	30 Oct-3 Nov	Revision	**We-Fr: Study Period**
END OF YEAR EXAMINATION PERIOD: 4 - 24 Nov			

Assignment 1

Topics

- Introductory Android
- Activities

Task 1: Android "Hello World"

Follow the A103 video in Lesson 1 – ignore the previous tutorial – it was too complex.

Be sure to deploy it to the emulator and test it. **Note that you must use Java for all coding done in this module, not Kotlin!**

Task 2: Android SOS

SOS is a simple game played by children on a piece of paper. The rules of the game are given below:

Game Rules:

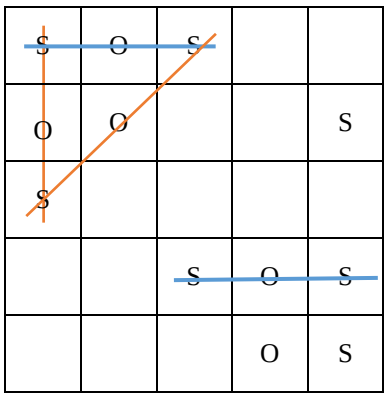
The game of SOS[2] , is played between two players.

The game begins play on an empty square grid of at least 3×3 squares. For this task, play should be on a 5×5 board.

Each player takes a turn. On a player's turn, they choose to place an "S" ***or*** an "O" symbol, and place this symbol in an empty cell[3].

If the newly placed letter completes an "SOS" sequence vertically, diagonally or horizontally, that player scores one point and gets another turn[4] (until placing a symbol does not complete a sequence, in which case it becomes the other player's turn).

Play continues until there are no empty cells left, the winner is the player with the highest score. If the scores are the same, then the game is a draw.



[2] You can read more about it here: https://en.wikipedia.org/wiki/SOS_(game)

[3] Note that a player is allowed to place ***either*** symbol each turn, unlike OXO where each player is allocated a symbol and may *only* place that symbol on their turn.

[4] The player can place ***either*** an "S" ***or*** an "O" on their next turn. They **do not** need to place the *same* symbol that they placed on their previous turn, i.e. this is *not* the Noughts and Crosses (OXO) game.

You are now required to write an Android app[5] that will allow two players to play the game on the same device in a pass-and-play manner, with turns alternating between the two players.

Use whatever Views and interaction methods you think most appropriate to implement the game. Be sure to let players know what their scores are at all times and when they need to pass the device to the other player so that they may take their turn.

When a player completes an SOS sequence, be sure to indicate *visually* that the symbols form part of a sequence and which player the sequence belongs too (in the figure above, there is a coloured line through the symbols, where the colour indicates which player completed the sequence). For example, you could display symbols that are not part of a sequence in lower case, while those that are part of a sequence in capitals and change the colour of the font to indicate who it belongs too. For example, letters belonging to a sequence for player 1 could be in blue, for player 2 in dark yellow, and if a letter belonged to a sequence from both players, it could be green.

Once the game is over, be sure to clearly indicate who the winner is and what the scores were.

You can change the text colour for a button, label etc. using

```
TextView lbl = findViewById(R.id.lblTest);
lbl.setTextColor(Color.RED);

Button btn = findViewById(R.id.btnTest);
btn.setTextColor(Color.GREEN);
```

The Color class has a number of predefined colours, such as RED and GREEN that you can use, or you can specify your own colour.

Make use of the **MVC pattern** to partition functionality into logical groups, i.e. don't mix up you controller and model logic specifically.

Unfortunately, Android does not have property bindings[6], so you will need to manually copy data to/from the model and the view, however, you can make use of the properties classes you wrote in Assignment 2, Task 3 of WRPV301 to create your own version if you want too[7].

Write SOLID code.

---

[5] Someone has made a version that can be found on the Play Store. You can look at it here: https://play.google.com/store/apps/details?id=com.androbros.sos&hl=en_ZA&gl=US

[6] **Not** in the way that you are used too in WRPV301. Properties and bindings are part of the JavaFX library. JavaFX will not work in Android as the UIs and platforms are completely different (desktop vs mobile). There *are*, however, many similarities between the two APIs

[7] We will not be looking at the Data Layer and XML Bindings that Android now supports – it adds too much complexity to the discussions at hand in this module. You may read up about it for interest if you want too though.

Rubric

Below is the rubric, available on Moodle that will be used when assessing this Assignment.

Criteria	Marks	Option
Task 1: Android "Hello World" In this task, you were expected to follow the class demo video and create a running app.	0	Not done
	1	Done
Task 2: Android SOS Game In this task, you were expected to implement the SOS game, specifically: 1. allow the game to be played on a 5x5 game board; 2. allow a player to place an "S" or an "O" in an empty space; 3. check for the completion of an "SOS" sequence. If one is found the current player gets a point and another turn; 4. when a sequence is completed, visually indicate who completed it, i.e. player 1 or player 2. There are many ways of doing this, including using colour lines that strike through a sequence, or a combination of colours and fonts to indicate the same information; 5. once a player's turn is completed, indicate that it is the other player's turn; 6. once the board is full, then the game ends and the winner must be shown.	0	Not done, or does not compile.
	2	1-3 of the functionality mentioned is implemented.
	4	All the functionality was implemented. The UX is average.
	5	All the functionality was implemented. The UX was very good.

Assignment 2

Topics

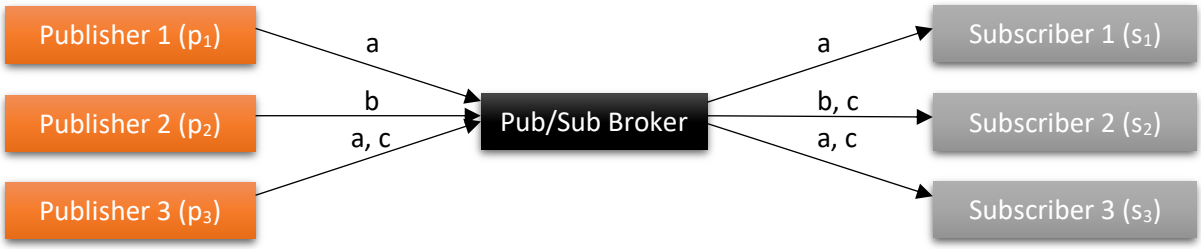
- Multiple Activities
- Views

Task 1: Pub/Sub Broker

Last semester, the SOLID[8] principles for OOP were discussed. One of the major issues related to the *de-coupling* of objects/classes so that they did not depend so much on one another. Highly coupled classes often:

- require complex set up so that communication can occur between them;
- require full knowledge of the other class (e.g. its methods being called); and
- have an issue with *scalability*, specifically when wanting addition classes (potentially unknown) to respond to the same messages.

Topic Name	Subscribers
a	s_1, s_3
b	s_2
c	s_2, s_3



One way of decoupling classes is to use a messaging system (event notification system) where a *publisher* may *publish* a message about a particular *topic* and if *subscribers* are *subscribed* to receive messages about *that* topic, then they do[9]. The important thing, though, is that neither the publishers, **nor** the subscribers know about one another (i.e. they do not have *references* to the publisher or receiver or have full knowledge about one another[10]). Instead, an intermediary, known as the broker[11], handles message passing.

[8] Check out https://en.wikipedia.org/wiki/SOLID if you cannot remember.

[9] If you look *carefully* at this, you should notice the similarity between what is being discussed here and properties and notifications encountered last semester. This approach is just more *generalised*.

[10] One of the advantages of this is that classes can be communicated with even though their code was not available at *design time*, i.e. you can write code to work with classes that haven't even been written yet – and you won't need to alter *your* code.

[11] Note that there is only *one* broker, but there can be many publishers and subscribers. All publishers and subscribers use the *same* broker.

When a subscriber is initialised, it registers itself with the broker, indicating which topics it is interested in receiving messages about. For example, when s_2 is initialised, it registers itself to receive messages about topics *b* and *c*.

When a publisher wishes to send a message about a particular topic, it sends the message to the broker, who then forwards the message to *all* subscribers that registered an interest in the topic. For example, when publisher p_1 sends a message about topic *a* to the broker, *the broker* forwards the message to subscribers s_1 and s_3. Multiple publishers can send a message about a particular topic (p_1 and p_3 can both send messages about topic *a*) and multiple subscribers can register to receive messages about a topic (e.g. s_2 and s_3 can receive messages about topic *c*).

Subscribers can register and deregister themselves at *any* time. The pub/sub broker should be easily accessible to *any* object, so it is often a singleton[12] or a static class.

A message will typically contain the publisher (who published it), the topic and optional data containing extra information about the message.

Using SOLID principles design and implement a simple Publish Subscribe architecture in Java (i.e. do not use Android-only classes) that can be reused in later assignments (including this one). In other words, be sure to use classes, interfaces, groupings of methods, generics, Lambdas etc. in a manner that is decoupled and cohesive.

While designing this architecture, consider how you *could* have used it to implement the demonstration task you did last semester demonstrating your implementation of properties and listeners, i.e. how could you use *this* architecture to implement the Capitals game, or the student records apps?

Hint: the implementations for properties and the pub/sub broker are *very* similar.

Demonstrate that your implementation is working correctly by completing Task 2.

Task 2: Pub/Sub SOS

In the previous assignment, you implemented the SOS game (using standard Android event handling). Now modify this implementation to make use of the Pub/Sub Broker[13].

All the UI cells on the board should now generate *messages* indicating that the user tapped them. Instead of the event being handled directly inside this event, rather have another class (the controller) deal with it.

In addition to this, be sure to include other messages, such as:

- a new game has started or ended;
- a player's turn starts or ends;
- a symbol has been placed in a cell (including *which* symbol); and
- when an SOS sequence has been completed, etc.

[12] Read https://en.wikipedia.org/wiki/Singleton_pattern for more details.

[13] The changes required for *this* task should not be a lot. If you find yourself writing a *lot* of code, it probably means that you need to rethink what you are doing. Either your code was too intertwined in the previous implementation or you are not using the pub/sub broker appropriately. This task is about thinking in an event-driven manner, which might take a bit of practise.

Make use of these messages to update the scores, display the "player turn" and "game over" messages, etc.

In addition to this, create player specific stats calculators that monitor the messages and calculate:

- how many turns each player had;
- how many O's they placed;
- how many S's they placed;
- how many SOS sequences they completed; and
- the highest number of SOS sequences they completed in a single turn.

You *must not* modify the game logic code to calculate these stats. Simply monitor the messages to do so, i.e. it should be possible to easily add new stats calculators *without* needing to modify existing game logic code (*O* of SOLID).

The stats calculated for each player should be displayed during game play and updated *as soon as* a stat changes[14].

Task 3: Contacts

For this task, you are required to create a simple Android app that will allow a user to keep and manage their contacts and dial them. A contact contains an avatar image, the name and contact number for a person or business. The contacts pictures can be found in the "Assignment 2 Files" folder.

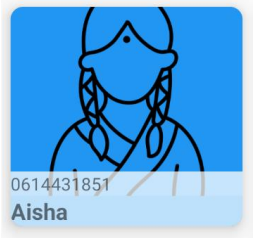


Figure 1: Contact Card

The app must have two Activities, namely the Contacts List and Contact Card Activities.

The Contacts List Activity is the main Activity and displays a vertical staggered grid of contact cards, which can be scrolled up and down. A contact card should look similar to that seen in Figure 1. Tapping on a contact card should display an edit, message and dial button on the card.

Tapping the:

- edit button takes the user to the Contact Activity and displays more details about the selected contact;
- message button will take the user to the SMS messaging Android screen; and
- dial button will dial the number of the contact.

At the bottom of the Contacts List Activity is a floating action button. Clicking on this button adds a new contact to the list and takes the user to the Contact Activity to enter the details;

The Contact Activity displays the details about a specific contact, including the name, number and an avatar image. It must be possible to change the avatar image used, the name and number of the user.

[14] Another message? Hint, hint…

It should be possible to return to the Contacts List Activity once the required changes have been made *and* these changes should be reflected in the Contacts List Activity. There should also be a floating action button at the bottom of the Activity. Tapping it will dial the contact's number.

All the contacts should be persisted to a file.

Populate the initial contact list at start up with at least 10 contacts.

Rubric

Below is the rubric, available on Moodle that will be used when assessing this Assignment.

Criteria	Marks	Option
Task 1: Pub/Sub Broker For this task, you were to design and implement a pub/sub broker and associated interfaces and classes using SOLID design principles. The solution was supposed to be usable in both Android apps and regular Java applications. The following are features that should be found in the solution: • a functional interface (preferably) that subscribers need to implement. This implementation is subscribed to a specific message topic in the broker; • the broker should: ○ be accessible from everywhere easily. One possible solution is making the broker a static class or a singleton ○ have one or more data structure(s) that can be used to easily look up subscribers by message topic ○ have a subscribe method that accepts a message topic and subscriber and adds them to the data structure above. Special care may be need to be taken when subscribing to a message that has not been subscribed too previously; ○ have an unsubscribe method that accepts a message topic and a subscriber and remove them from the data structure. The data structure needs to be cleaned up if there are no more subscribers for a specific topic; ○ have a publish method that accepts a message topic and one or more parameters. Subscribers subscribed to the message topic should be retrieved and each is called, passing the parameter(s). The solution only needs a PubSubBroker class and a Subscriber interface. Additional classes and interfaces may not strictly be necessary.	0	Not implemented, or does not compile.
	1	Some of the required functionality was present, but not all.
	2	All of the required functionality was present, BUT Android of JavaFX specific classes were used in the implementation
	3	All of the required functionality was present, NO Android or JavaFX specific classes used.
Task 2: Pub/Sub SOS (Basic Implementation) The Pub/Sub Broker implemented above was to be used to re-implement the SOS game from Assignment 1 using message passing.	0	Not done, or does not compile.
	2	Partially working.
	4	All the functionality from Assignment 1 was present.

Task 2: Pub/Sub SOS (Stats Calculators) Special stats calculating classes were to be written and instantiated. Each of the calculator classes was to subscribe to one or more message topics (related to what it calculated) and update its stats accordingly. Additionally, these stats were to be displayed on the screen.	0	Not done, or does not compile.
	1	Some of the calculators implemented, but not all.
	2	All of the calculators implemented and their information displayed.
Task 3: Contacts (Basic Functionality) The basic functionality required for the task include: • displaying contacts in a list; • being able to add a new contact (after tapping the + floating button); • being able to edit a contact's details in a separate Activity.	0	Not done, or does not compile.
	1	Most of the functionality present.
	2	All of the functionality present.
Task 3: Contacts (Other Functionality) In addition to the basic functionality, the app was supposed to: • context buttons appear when tapping on a contact in the contact list, when different contact was tapped on, the buttons should disappear (if they were visible); • be able to load and save the contacts to the apps local files; • be able to use the built in SMS and dialling intents; • be able to pass parameters between the two Activities.	0	Not done, or does not compile.
	2	Most of the functionality present.
	4	All of the functionality present.

Assignment 3

Topics

- Views
- Multiple Activities
- Collections

Task 1: Mobile Boop

In WRPV301, you were required to create a JavaFX version of the game Boop. Boop is a two-player game in which players place cats on a bed, trying to knock off their opponent's cats.

A deceptively cute, deceivingly challenging abstract strategy game for two players.

Every time you place a kitten on the bed, it goes "boop." Which is to say that it pushes every other kitten on the board one space away. Line up three kittens in a row to graduate them into cats… and then, get three cats in a row to win.

But that isn't easy with both you AND your opponent constantly "booping" kittens around. It's like… herding cats!
Can you "boop" your cats into position to win?
Or will you just get "booped" right off the bed?

The rules are available on the Learn site, and you can watch a short tutorial video on YouTube here:

https://www.youtube.com/watch?v=-bFHwu2IPdQ

You are now required to implement a mobile version of the game, with the following requirements:

1. the game should be able to be played on a *mobile phone* in portrait *and* landscape mode. The layouts should be appropriately resized/re-organised as necessary;
2. the game's text should be available in English and at least one other language (that you can speak);
3. there should be multiple Activities, at a minimum a Splash Screen, Main Menu, Play Screen and Results Screen;
4. the UX *does not* need to be the same as the JavaFX version from last semester, i.e. it need not employ drag-and-drop for the placement of felines or animations when felines are booped. It is acceptable to use tap-interaction, e.g. tap a feline on the floor, then tap an open spot on the bed to place it; and
5. the state of the game must be clearly displayed at all time.

You are **required** to use the **MVC pattern** when designing the game, in other words, the game's logic, state and operations need to reside in the Model, with the Controller calling operations based on user interaction with the View and updating the View based on the Model's state.

The Model (specifically) will be re-used in Assignment 5, so be sure that it is well separated from the View and Controller. It may be possible to re-use the View and Controller as well, so ensure that it is well written.

Rubric

Below is the rubric, available on Moodle that will be used when assessing this Assignment.

Criteria	Marks	Option
Task 1: Boop Model You were to implement the Boop game, but make use of the MVC pattern. This means that the Model should not have *any* references to View or Controller related code (i.e. only plain Java). The Model should contain only: 1. data (e.g. the board layout, where the felines are, the scores, whose turn it is, etc.); 2. methods to manipulate the game state (e.g. start game, place feline, etc.); 3. methods to query the game state (e.g. is there something on the bed at position <x,y>, get current state of game (not started, player 1's turn, game over), is there a winner, etc?); and 4. optionally callback methods to notify an Observer of state changes (e.g. onGameStarted, onFelinePlaced, onPlayerTurnChanged, etc.).	0	Not implemented, or does not compile.
	1	Contains View/Controller related code, e.g. access to UI controls
	2	Adheres to some of the criteria #1-3
	3	Adheres to criteria #1-3
Task 1: Boop Android Implementation You were to implement the Boop game on Android devices such that the game could be played in pass-and-play mode. The basic functionality required was: 1. Splash screen displays; 2. Main Menu from which game is started; 3. Playing screen showing the bed, felines, whose turn it is, etc. Current player places one of their felines on bed. 4. Results screen displayed after a game is completed, showing who won. The Model created was to be used for managing the data of the game, as well as the game play logic.	0	Not implemented, or does not compile.
	1	At least 50% of expected functionality.
	2	At least 75% of expected functionality.
	3	It worked perfectly.
Task 1: Boop UX Since the device is different from a desktop device, the UX is different. How good was the user experience adjusted to cater for a mobile device, i.t.o. device screen size, interactions, etc.	0	Terrible
	1	It worked somewhat
	2	Good (fairly good overall)
	3	Excellent (utilised touch gestures, swipe, screen orientation changes, etc.)
Task 1: Bonus Points Extra marks for going past the brief. This includes things like sounds, animation, using drag-and-drop gestures, etc.) Note: Moodle does not know how to handle bonus marks (going past 100%), so this Assignment's marks will be adjusted at the end of the module to cater for this.	0	Nothing extra
	1	Some extra things included
	2	Awesome!!

Assignment 4

Topics

- Collections
- Threads

Instructions

You may implement this as a Java console app or an Android app, although a Java app may be easier to debug.

Task 1: Sudoku Generator

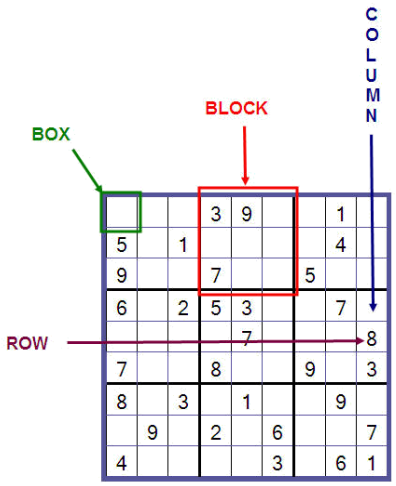
This task was an exam question in previous years.

For this task, you are required to write a Sudoku generator using the *Wave Collapse Function* algorithm. You *must* make use of the streaming API wherever possible.

Hint: consider the data structure(s) you use carefully.

The Rules of Sudoku

The classic Sudoku game involves a 9x9 grid of 81 boxes. The grid is divided into nine blocks, each containing nine boxes.



The rules of the game are simple:

- each of the nine blocks must contain all the numbers 1-9 within its boxes; and
- each number can only appear once in a row, column or block.

One way to randomly generate a *valid* Sudoku grid is using the *Wave Function Collapse* algorithm[15] from Quantum Mechanics. This algorithm can be used to generate many things, including a Sudoku grid.

The algorithm works by initially having each box contain a set of *all* possible valid values, e.g., 1, 2, 3, 4, 5, 6, 7, 8, 9. So initially the grid contains all possible Sudoku grids that can *ever* be generated.

Then using a process of elimination, the possible Sudoku grids are reduced until finally a *single* particular Sudoku board is generated.

The process of reducing the possibilities works as follows:

- pick a box, b, with the least number of possibilities remaining[16], then randomly pick one of the possibilities, p, in that box. This becomes **fact** for box b (i.e. there are no other possibilities for it). Note that a box with one possibility is *not* a fact until it is selected as a fact (with all the subsequent checks taking place as well);

[15] Here is another example of the Wave Collapse Function being used. Also includes a nice description of the general algorithm used for creating Wedding seating charts:
https://robertheaton.com/2018/12/17/wavefunction-collapse-algorithm/

[16] If there are multiple boxes with the same number of possibilities remaining, randomly pick one of them.

- remove p from the possibilities of all boxes in the same row, column and block as b.

Repeat this process until all the boxes contain only facts.

At this stage, a valid Sudoku grid should have been generated, where all the boxes have a single value (fact) and none of the rules have been violated.

The table below shows an example of what a grid could look like after a few reduction steps. Box A4 contains all possibilities (1-9). Boxes B2, C2 and D3 have facts (2, 3 and 6 respectively). Based on these facts, the possibilities of other boxes have been reduced. For example box D2 has possibilities 1, 4, 5, 7, 8 and 9. This is because the possibilities 2 and 3 were removed due to being in the same row as boxes B2 and C2 and the possibility of 6 being removed because it is in the same block (and column) as D3.

	A	B	C	D	E	F
1	1,4,5,6,7,8,9	1,4,5,6,7,8,9	1,4,5,6,7,8,9	1,2,3,4,5,7,8,9	1,2,3,4,5,7,8,9	1,2,3,4,5,7,8,9
2	1,4,5,6,7,8,9	2	3	1,4,5,7,8,9	1,4,5,7,8,9	1,4,5,7,8,9
3	1,4,5,6,7,8,9	1,4,5,6,7,8,9	1,4,5,6,7,8,9	6	1,2,3,4,5,7,8,9	1,2,3,4,5,7,8,9
4	1,2,3,4,5,6,7,8,9	1,3,4,5,6,7,8,9	1,2,4,5,6,7,8,9	1,2,3,4,5,7,8,9	1,2,3,4,5,6,7,8,9	1,2,3,4,5,6,7,8,9

When picking a new box, one of boxes D2, E2 or F2 will be selected as they all have 6 possibilities left. If box E2 was selected, then one of the possibilities 1, 4, 5, 7, 8 or 9 would be randomly chosen. If, for example, 9 was chosen, then E2 would hold the fact 9 and all the 9s would be removed from the possibilities of the boxes in row 2, column E and the block in which E2 resides namely (D1, E1, F1, D2, F2, E3 and F3).

Should there *ever* exist a box with 0 possibilities, then the grid is invalid[17]. Display an error message and try again.

Write two classes named `SudokuGenerator` and `SudokuGrid`.

The `SudokuGrid` class represents a board and the values (the final facts) placed in it. It must provide functionality to set, retrieve and clear box values, as well as a method to determine whether placing a value in a given box is valid, invalid due to row conflict, invalid due to column conflict or invalid due to block conflict. Note that it is possible to be invalid for any combination of the three reasons (e.g. invalid due to row conflict AND invalid due to column conflict).

The `SudokuGenerator` class is responsible for generating a valid complete `SudokuGrid` using the Wave Function Collapse algorithm described above.

You must make appropriate use of the Java Collection Framework (collections, algorithms, streaming API, etc.) when completing this task.

Demonstrate that your implementation works correctly by displaying generated boards after the user presses “enter” and display the number of invalid boxes on the board (using the method you wrote previously).

Task 2: Threaded Generation of Sudoku Boards

In Task 1, you wrote an algorithm to generate Sudoku game boards (full solutions). Now you need to write a program that uses threads to do the following:

[17] Not 100% sure if this case *can* ever occur though, but just in case.

- generate 10,000[18] *unique* Sudoku game boards, allocate each a unique number and save these to a file – as quickly as possible.

Game boards are numbered 0, 1, 2, 3… in the order that they are generated. A game board is saved in a text file named 0.txt 1.txt, 2.txt, etc. which simply contains all the numbers in each box of the game board.

A game board is considered the same regardless of how it is rotated, flipped along the rows, flipped along the columns, rotated 90°, 180° or 270°, or any combination therefore[19]. So a game board is considered the *same* as a mirrored or rotated (or mirrored and rotated) version of itself, etc.

You will need to have threads to generate game boards, some to check for uniqueness and another for saving unique game boards to file.

The correct choices in data structures and algorithms used are *very* important in this task.

Rubric

Below is the rubric, available on Moodle that will be used when assessing this Assignment.

Criteria	Marks	Option
Task 1: Sudoku Grid Class For the grid class, there was supposed to be: 1. a suitable data structure used (from the JCF) 2. methods to clear all boxes, set a box value and query a box value 3. a method to query whether placing a value in a box is valid or not. The method should be able to return whether it is a valid or invalid option, as well as the reason(s)	0	Not implemented, or does not compile.
	1	At least 50% of expected functionality.
	2	At least 75% of expected functionality.
	3	All functionality implemented and correct.
Task 1: Sudoku Generator Class For the generator class, there was supposed to be: 1. an initial set up, in which all possibilities exist; 2. the WCF that gradually determines values for all the boxes (should not be more than 81 iterations needed); 3. stops when a valid board has been generated, or an impossible board is generated. Appropriate use of the JCF was to be made. The streaming API should be used where possible.	0	Not implemented, or does not compile.
	1	At least 50% of expected functionality.
	2	At least 75% of expected functionality.
	3	All functionality implemented and correct. The streaming API was used where possible.
Task 1: Demonstration A simple program to generate and display valid generated boards.	0	Not done, or does not compile.
	1	Done and runs correctly.
Task 2: Threaded Sudoku Board Generation A threaded approach to generating multiple boards was required. Each thread should generate a board and once done, make it available for checking in another thread.	0	Not done, or does not compile.
	1	Generated boards, but all done in a single thread.
	2	Multiple threads generating boards. Boards passed to checking thread.

[18] Or some other large number…

[19] Note that the representation of your data can go a *long* way to how efficiently the comparisons, rotations, mirroring, etc. can be done. Think about this, unless you feel like sitting for months waiting to get all 50,000 game boards.

<table>
<tr><td rowspan="4">Task 2: Threaded Sudoku Board Uniqueness Checking
Once a board has been generated, it needs to be checked for uniqueness. This requires:
• all boards that have been generated to be stored;
• a method (threaded?) to check a new board against all existing boards;
• if no matches are found, store the board.
New boards should be passed to a saving thread.
The matching method should consider the rotation (0, 90, 180, 270 degrees) and mirror row/column variations of the boards.
The data structure chosen is very important for this. Using a simple String or integer array are examples of really simple, efficient examples that could be used.</td><td>0</td><td>Not done, or does not compile.</td></tr>
<tr><td>1</td><td>Generated boards, but all done in a single thread.</td></tr>
<tr><td>2</td><td>Not done, or does not compile.</td></tr>
<tr><td>3</td><td>Generated boards, but all done in a single thread.</td></tr>
<tr><td rowspan="3">Task 2: Threaded Saving
New, unique boards were to be saved.</td><td>0</td><td>Not done, or does not compile.</td></tr>
<tr><td>1</td><td>Done, on Main thread.</td></tr>
<tr><td>2</td><td>Done, on separate thread.</td></tr>
<tr><td rowspan="2">Task 2: Passing boards between threads
An efficient way of passing boards between the different threads was required. One such way is to use two thread-safe queues:
• one for newly created boards - to be checked; and
• one for boards to be saved.
Other mechanisms could be used too, but they need to be thread-safe, i.e. locks might be needed to prevent race conditions.</td><td>0</td><td>Not done, or does not compile.</td></tr>
<tr><td>1</td><td>Suitable mechanism used.</td></tr>
</table>

Assignment 5

Topics

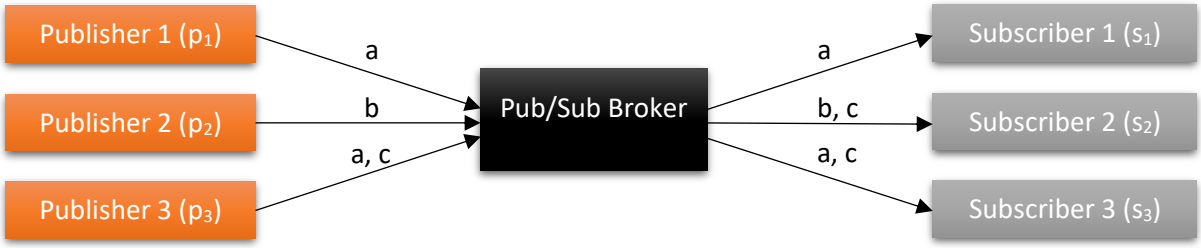
- Networking
- Threads

Task 1: Networked Pub/Sub Broker

In Assignment 2, you were asked to implement a Publish/Subscribe Broker pattern where Subscribers subscribe to receive messages about a specific topic via the Broker. Publishers can publish messages about specific topics, which are then forwarded via the Broker to all subscribers subscribed to those topics. Subscribers can unsubscribe from a topic at any time and will then not receive any further notifications.

A very important aspect of this was that Publishers and Subscribers did not know about each other explicitly. Instead all communication was handled via a single, shared Broker object. In your assignments, the Broker, Publishers and Subscribers were all within the *same* application and on the *same* device.

Topic Name	Subscribers
a	s_1, s_3
b	s_2
c	s_2, s_3



For this task you are to create a *thread-safe networked* version of the Publish/Subscribe Broker. In this scenario, the Broker is hosted on a Server and the Publishers and Subscribers run in different client applications and possibly different devices. The Publishers and Subscribers communicate with each other indirectly via the Broker. Communication with the Broker is handled using a TCP/IP connection[20].

You have been provided with an implementation of a Publish/Subscribe Broker class and associated interface where everything was running within the same application in the "Assignment 4 Files" folder, which will become available from week 6. Modify this implementation, or your own from Assignment 2, to allow for a networked version of the pattern.

Note that your implementation *must not* interfere with the normal execution of an application, e.g. it must not block an Android app's UI thread. Also a client should not "block" while waiting for publications, it should be possible to publish anything at any time (and receive anything at any time).

[20] Networked messages in this task is very similar, in many ways, to the concepts of web services, remote procedure calls, etc. used elsewhere.

For this implementation, when a Client connects to the Broker Server, it is allocated a unique integer ID. This ID is communicated with the Client. The ID is used as the "publisher" in a message.

The Server must be a Java console-based application. The Client side classes must be able to run in either regular Java applications or Android apps, i.e. don't use any Android only or Java only classes.

Make appropriate use of the Java Collection Framework, Lambdas, threads, sockets, etc. in your solution.

Task 2: Multi-platform Networked Boop

In Assignment 3 for WRPV302 and Assignment 4 in WRPV301, you were required to implement an Android and JavaFX version of the game Boop (mobile and desktop).

You were instructed to implement a fully functional version of JavaFX Boop prior to WRPV302 as it would be used in this module again. If you did not do WRPV301 and write your own version of JavaFX Boop this year, you may obtain a copy from one of your peers who did. **Note that this version must be the version from WRPV301 and may not be networked.**

For this task, you will be required to create a networked version of Boop, such that:

- there is a Java console game server that manages networked playing of the game between *multiple* pairs of players, *at the same time*; and
- connecting to the server will be two *types* of clients, one written in JavaFX (desktop) and the other in Android (mobile). For a particular game pairing, clients can be any combination of the two types, e.g. Android & Android, JavaFX & Android or JavaFX & JavaFX.

When a player wishes to play a game, they will click the start button on their app, which will then connect to the Java server. The Java server will pair players and start a game once there are two players (and will go back to creating more pairs of players as well – i.e. use a thread per game pairing). For each game, the server will co-ordinate whose turn it is, the placing of felines, scoring, and game results, etc. Once a game is over, the connection to the server is terminated.

The client apps will respond to messages from the server, updating the user interface. Commands[21] issued by players via the app UI will be sent to the server, who will broadcast the Commands to the playing pairs, who will effect the Command on each player's Boop Model (which will update the UI, etc.). Sufficient information needs to be sent with Commands so that the internal state of the Model of each player is *identical*, e.g. the placements of felines after a boop, etc.

Both the JavaFX and Android versions of the game that has already been written need to be updated to now allow for networked communication.

Note: the final exam practical section will *probably* have a lot of similarity with the tasks in this assignment. It would be short-sighted *not* to complete them.

[21] You may use the networked pub/sub broker created in Task 1 if you wish too.

Rubric

Below is the rubric, available on Moodle that will be used when assessing this Assignment.

Criteria	Marks	Option
Task 1: Pub-Sub Server/Client Implementation Was a thread safe pub-sub broker implemented? This implementation should have addressed issues like - concurrent access (simultaneous publishing and subscribing actions by the server), and - handling of multiple client connections.	0	Not implemented, or does not compile.
	2	Non-thread safe Pub-Sub server implemented and could only communicate with either one publisher and/or one subscribe
	4	Thread-safe pub-sub implemented but could only communicate with one publisher/subscriber at a time.
	6	Thread-safe pub-sub server implemented and could communicate with multiple clients simultaneously.
Task 1: Pub-Sub Server/Client Implementation Dependencies Was the thread-safe pub-sub implementation dependent on any JavaFX / Android-specific libraries?	0	Not implemented, or does not compile.
	1	Some JavaFX / Android specific libraries were used
	2	No specific to Android/JavaFX libraries or classes.
	4	No specific to Android/JavaFX libraries or classes and JCF / lambdas were used.
Task 1: Pub-Sub Server Implementation Did the pub-sub server allow for unique identification of clients via an "ID" and was this ID used to identify publishers/subscribers?	0	Not implemented, or does not compile.
	1	IDs were implemented but did not work correctly
	2	IDs were implemented but not used
	3	Unique IDs were implemented and correctly identified clients
Task 1: Pub-Sub Client A client implementation was required to interact with the thread safe pub-sub broker. The client implementation required: • the client should be able to subscribe to or publish to any specific topic. • the client itself could be a publisher and subscriber of the same topic. • the client pub-sub should also run off of the main (UI) thread running as a non-blocking operation.	0	Not implemented, or does not compile.
	2	Client could either subscribe/publish but not both.
	4	Client could subscribe/publish but to the same topic or was implemented with a blocking style implementation.
	6	Client could publish and subscribe (to the same topic) and was implemented in a non-blocking style ensuring continuous user interaction while waiting for publish messages.
Task 2: Boop Server You were required to write a server that: • was a **Java console** application; • made use of multiple threads to manage the player pairing, and then game play between a pair; and • handles communication with clients on different platforms (in this case desktop & mobile) – by using a common communication protocol. The server should have functionality similar to:	0	Not done, or does not compile.
	2	Pairs players, game play not functional
	4	Pair players, one game at a time
	6	Pairs players, multiple simultaneous games, fully threaded, handles connections being lost

• a single main **pairing** thread, that accepts connections, the waits for **two** connections, before starting a **game play** thread with the two players; • a per pair **game play** thread, that runs for the duration of a game, passing commands between the two players. The server is used as the hub through which communication happens (and for debugging purposes should probably display what is happening at all times). Provision for network connections being lost must be made.		
Task 2: JavaFX Boop Client The JavaFX Boop implementation from last semester (or one obtained from a peer), modified to be networked. This would include: • way to connect to server; • indication that waiting for an opponent; • sending user commands to server; • updating internal state based on network commands received; and • disconnecting from the server once a game has completed. Handles lost connections gracefully.	0	Not done, or does not compile.
	2	Can play the game, but not networked
	4	Networked, but not all of the functionality working correctly
	6	Fully networked, works well, handles lost connections
Task 2: Android Boop Client The Android Boop implementation from Assignment 3, modified to be networked. This would include: • way to connect to server; • indication that waiting for an opponent; • sending user commands to server; • updating internal state based on network commands received; and • disconnecting from the server once a game has completed. Handles lost connections gracefully.	0	Not done, or does not compile.
	2	Can play the game, but not networked
	4	Networked, but not all of the functionality working correctly
	6	Fully networked, works well, handles lost connections
Task 2: Boop Model The Model created in Assignment 3 was supposed to be reused in this assignment, and could have been shared between the Server, JavaFX Client and Android Client – as is. The Model was supposed to be a **Java-only** class (POJO) – and have no references to JavaFX/Android specific classes etc. The Model *could've* been packaged into a library that is used by each of the projects/modules. Each version of the Model would be updated via network messages One way would be the clients send "command intentions" to the server (but do not apply on client side initially) when the user does something. The server applies the intention (if valid) and broadcasts a command to all clients, who then apply these on their models (which then updates the UI).	0	Model not a separate Java-only class.
	1	There is a Model, but usage is inconsistent or inappropriate
	2	Server and both Clients used the Model appropriately