Cosc 1P03 lab 1.

Intro into IntelliJ

IntelliJ will be used as the IDE of choice. It is an industrial level IDE, featuring a solid project environment as well as a powerful debugger. The debugger will be introduced in the Week 3 labs.

Some things to keep in mind. In the labs you have access to your network drive. It is important that all projects be created on your network share and not the desktop.

Start IntelliJ, under File select new project or if it pops up with a get started menu, select New. Pay attention to Location and Name. Replace Some Project name with a descriptive name, and set your location to your network share, “Z drive”. The JDK should default to Java version 11, if not then set it so.

A screenshot of a computer program

Description automatically generated

My example project is called Test, and by default it creates a “Hello World” main.

Most will want to use BasicIO as a get started IO library. Adding the import statement:

A screen shot of a computer

Description automatically generated

Will likely show up in red, indicating it is not in the path. Let us add this to the build path.

Right click the project folder to the left, “Some Project name”, which you called your project. Near the bottom of the menu “Open module settings” F4. Opens into the following screen.

A screenshot of a computer

Description automatically generated

Select Libraries, then the + in the second column to the right. A small box will open, select Java, first option then opening a dialogue box similar to as shown below.

A screenshot of a computer

Description automatically generated

Select Brock.jar from the location it is stored on your computer. Lab computers: it is on the C drive under a folder called Brock. Apply and close out. The import statement should now be white or gray (if it is not in use).

COSC 1P03 – LAB 1 – ARRAYS

This week, we’ll be working with arrays, both single- and two-dimensional.

We’ll primarily look at allocating and indexing (subscripting); next week we can look into how to search for contents.

# 1P03 Labs this term

Note that the purpose of labs is kinesthetic learning: you learn by doing. This means you won’t be given “the answers” at the end; there is no ‘answer’ to a process. It also means you can’t ‘submit’ the following week (nor can you ‘submit’ at the start of lab); there’s no submission for labs. You come in, you work through the task for two hours, you demonstrate that you’ve completed a substantive portion of the lab, and the lab leader records that.

If you have difficulties, your lab leader is there to *answer questions* and *help*. Yes, your lab leader can easily write code, but that code is not ‘the work’.

i.e. be prepared to work.

# Background: Conway’s Game of Life

This is just the beginning for a whole new course, so we’ll start small: **the creation of life itself**.  
Nah, not really. Instead we’re going to look into Conway’s Game of Life: a sort of simulation of the idea of how life could grow. It’s mostly more of a thought experiment, but the kind that can be quickly and easily implemented.

Suppose we have an *infinite grid* of orthogonally-aligned discrete cells. Wait, scratch that. That’d require infinite RAM.  
Suppose we have a small selection from a potentially infinite grid of cells:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 |  |  |  |  |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |

We’ve indexed their locations, simply to make it easier to refer to them.  
I’ll use the convention **RC**: 30 is the bottom-left cell, 33 the bottom-right, 03 the top-right, etc.  
We’ll say a ‘living cell’ is filled with black, and a ‘dead cell’ is filled with white.  
A cell has ‘neighbours’: any of the 8 locations *around* it (a cell is *not* its own neighbour).

In this model, “life” is discretely divided not just into cells, but also by time: all cells start life, die, or continue their previous state each ‘generation’. What’s more, the decision is made **in parallel** for all cells (simultaneously).   
The rules are simple:

* A dead cell with precisely three living neighbours comes alive
  + Any other dead cell stays dead
* A live cell with 2 or 3 living neighbours stays alive
  + Any other living cell dies off (loneliness, overcrowding, whatever)

From the example above, consider cell 11:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | **1** | 2 | 3 |
| 0 |  |  |  |  |
| **1** |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |

* It’s “initially alive”, so we count its neighbours: 01, 02, 20, and 22 are alive, so four
  + This means it “dies”

What about cell 22?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | **2** | 3 |
| 0 |  |  |  |  |
| 1 |  |  |  |  |
| **2** |  |  |  |  |
| 3 |  |  |  |  |

* It’s initially alive, and we count its living neighbours (11 and 33 are alive; 12 13 21 23 31 and 32 are dead): 2
  + We only need “2 or 3” to stay alive, so it stays alive!

But wait, what if we wanted to *implement* this?

Suppose, through some traversal, we’d already applied the rule for cell 11? We’d look at 22 with:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 |  |  |  |  |
| 1 |  | X |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |

It’s “dead” by the time we get to 22, but if we’re supposed to be doing “everything in parallel”, then that means we’re using the wrong data! This idea of parallelism is part of what makes *cellular automata* so interesting!

Well crap, this means there’s only one choice:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 |  |  |  |  |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 |  |  |  |  |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |

We need to use *two* grids: we read from one, and write into the other!  
There’s still one more question though: how do we deal with the *edges*? In the thought experiment, the grid extends into eternity. In an implementation, it’s probably best to not assume infinite RAM. There are different options:

* One can simply ignore indices that would be ‘out of bounds’: the edges/corners simply have fewer opportunities for neighbours
* One can wrap: the left of the left-most is the right-most edge!

Let’s go for the first option. When we get there, we’ll make it simpler by writing a *function* to count neighbours.

# The task: getting a start on the Game of Life

We want to take our time with this, so we don’t need a ‘full application’. We just need to show that we can apply the basic rules for the game, and write it reasonably. This means no ‘static’ nonsense, and also **no instance variables** other than the ASCIIDisplayer/ASCIIPrompter: if you want to access an array in a method, then **pass it as a parameter**.

* Initially assume a 7×7 grid; you can change it to ask the user later if there’s time
* The grid must be populated; we’ll start by hard-coding, but you’ll need to replace that eventually
* For any given location, find out the number of living neighbours (must be safe to call from any legal position)
* Apply the rules of the game of life, to create a **new** identically-sized grid

If there’s time, also extend it to being able to repeat itself.  
Note: being able to use different grid sizes, and being able to repeat **are** very important additions. Do them eventually!

* Different grid sizes (notably where the rows and columns don’t match) are how you ensure it won’t crash
* Repetition is how you confirm you’re managing the memory correctly, and abstracting well

Anyhoo, let’s start with representation, and boilerplate!

## Getting started

Start up IntelliJ and create a project. Note that we’ll be using BasicIO, though that’s not a major deal for this task.

Note: for the most part, you’ll be writing everything yourself. But this is just a “starting point” because there’s little point in working out what’s little more than a template.

Just a couple questions to work out:

* How do we want to *represent* the grid?
* How do we want to *visualize* the grid?

Note that these are separate questions. Each cell is either ‘alive’ or ‘dead’; since it’s binary in nature, 1/0 or true/false would make sense. We might also be tempted to use something like X/O; that’d be more human-friendly, but more tedious to write. We’ll still use X/O when we write a method to display the grid, but booleans or integers?

* Booleans make the most sense for a *single* cell
* Integers of 1/0 make *counting* cells trivial

Neither is “wrong”, but I like 1/0, so integer array it is! (For mine; amend your own as you see fit)  
I’ll hardcode my grid for now. I’ll ***very temporarily*** use a 7 by 7:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

import BasicIO.\*;  
public class Life {  
 *//ASCIIPrompter prompt; //Don't need this yet* ASCIIDisplayer output; *//Could use System.out.println* public Life() {  
 output = new ASCIIDisplayer();  
 int[][] grid={ *//Multiple lines solely for readability* {0,1,0,0,0,0,0},  
 {0,0,0,0,0,1,0},  
 {0,0,1,1,0,1,0},  
 {0,1,1,0,0,0,1},  
 {0,0,0,0,0,0,0},  
 {1,1,0,0,1,0,0},  
 {0,0,1,1,0,1,0}  
 };  
 showGrid(grid); *//Not yet implemented!* }  
  
 public static void main(String[] args) {new Life();}  
}

Even though we should “understand everything before we start coding”, when dealing with data representation, it’s simply more practical to implement some mechanism for user feedback as soon as possible (whether ‘real’ or a placeholder). So let’s focus on showGrid:

private void showGrid( int[][] grid ) {

Let’s talk about this:

* Recall from lecture that we use [][] to show it’s a *two*-dimensional array
  + Technically, it’s an ‘array of arrays’; effectively (int[])[] (but not literally; don’t do that)
* Why did we ***need*** to use a parameter at all?
  + Because we’ll eventually have *two* grids; effectively *many* grids, once we start having multiple iterations through the game of life. We need the ability to display an arbitrary grid
* How big is the grid?
  + It has grid.length rows
  + Each row has a length, which is its number of columns

If all we need is a row-major traversal, there’s not much to this:

private void showGrid( int[][] grid ) {  
 for ( int[] row : grid) { *//For each row...* for (int value : row) *//Each column within the row* output.writeC(value > 0 ? 'X' : 'O'); *//Ternary conditional operator!* output.newLine();  
 }  
}

I’m assuming you remember the ternary conditional operator we used in lecture, right? Yes? Great!  
Okay, so put that in and we actually have something we can compile and run. This is probably your first time using IntelliJ, so your lab leader can absolutely help you with that! Don’t forget to create a new **Run Configuration**, and **select the class**!

## Counting the neighbours

Most of the ‘main algorithm’ will actually be trivial: get some number. Is it 2 or 3? Neat! (barely more than that!)  
But we do need to ‘get that number’. A common early mistake when learning to code is to lose track of the scope of when something happens, ‘how many things it happens to’, etc.  
What we want is to know, precisely, the number of ‘living’ cells around some specific location. No more, no less.

This means we’ll need to accept: a row, a column, and a grid to retrieve the information from.  
Something to watch out for: recall that we don’t include *that location itself* in the tally! e.g. remember:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | **1** | 2 | 3 |
| 0 |  |  |  |  |
| **1** |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |

For cell 11, the total is *four*; not *five*!

private int countNeighbours( int row, int col, int[][] grid ) {

So how do we do this? We have options!

* We could do a loop for the row of three values above the position, another for the three below it, and manually add the left and right
* We could do nested loops for 3×3, adding everything (erroneously including ‘this position’!), then subtracting the ‘this position’, so it all works out
* Since there’s always precisely eight neighbouring positions, we could literally just write it out eight times!

But… we have an issue, right? What if ‘this position’ was 20? Or 32? Or 00? 33? There are two valid approaches to this:

* Our countNeighbours is already going to be messy regardless; that’s part of why we separated it out. Just let it be even messier, and include a bunch of conditional statements!
* We take the abstraction one level further: use a *filter function*!

There are a few meanings of the term ‘filter function’, but most relate to approximately the same idea: take a bunch of ‘stuff’, and give *what matches* or *what’s legal*. So we could write a getCell function, to take the same parameters as countNeighbours, and it returns either a zero or a one. It ***always*** returns a zero or a one.

* Ask for the contents of a valid index? Then get that value!
* Ask for the contents of row -300? I guess it’s a zero!
* Ask for the contents of column 1000000 from a 5×5 grid? Still zero!

This one’s pretty easy to write. So easy I’ll just provide it here:

*//All calls guaranteed to be TREATED as legal:*private int getCell( int row, int col, int[][] grid ) {  
 if ( row<0 || col<0 || row>=grid.length || col>=grid[0].length)  
 return 0;  
 return grid[row][col];  
}

And… that’s it! If the index is out of bounds, just assume there’s no living cell there!

So then, how do we count the neighbours?

Well… *we* don’t. You do. You now have every functional component you need, and three valid approaches listed above.  
Pick one?

After I finished, I added these two lines immediately after the call to show the grid:

output.writeLine("Neighbours of 3|2: "+countNeighbours(3,2,grid));  
output.writeLine("Neighbours of 6|3: "+countNeighbours(6,3,grid));

Of course, I worked out the expected output *before* running it:

* 32 is alive, with 3 living neighbours. It should ignore that it itself is alive, and say 3 (which it does)
* 63 is along the edge, so it’s a good test of the bounds-checking

## Taking stock of progress

Once we get to this point, what tasks are left to complete?

* We need to be able to use arbitrary grids, of arbitrary sizes
* We need to actually *apply the rules* of the game of life, to produce a new grid
* We eventually need repetition

Clearly the second point’s the only one that’s pressing. So what’s involved:

* Creating a new 2D integer array, with dimensions matching the original
* Iterating over the positions, inspecting the old grid, and ‘writing into’ the new one
  + This means we can’t use for-each loops for this, as the subscripts matter!
* Returning the updated grid

You’ll need to work this out for yourself, but this’ll get you started:

private int[][] apply(int[][] original) {  
 int[][] next=new int[original.length][original[0].length];  
 *//Code goes here!*  
 return next;  
}

I swear this is a heck of a lot easier than it initially seems. Your lab leader will hopefully resist the urge to write code for you (it can be *very* tempting when you’ve got a computer hooked up to a projector!), but will perhaps draw a picture or two on the board?

My solution for this part (i.e. the part ‘that does the algorithm’) is all of 13 lines. You got this!