The University of Melbourne
School of Computing and Information Systems
COMP90041 Programming and Software Development
Lecturer: Dr Tilman Dingler & Dr Thuan Pham
Semester 2, 2020

Assignment 2
5pm (AEST), October 15, 2020

1 Introduction

This project will continue the work you did on assignment 1. Please refer to the assignment 1 specification for a detailed description of the game and its general rules.

The aim of this project is to add some more advanced features to the system you previously developed. In the last assignment, you created a simple Java program to handle Nim's core game play mechanics, including a basic command prompt environment. In this assignment, the objective is to design and implement a more sophisticated version of Nim, making full use of Java's object-oriented paradigm.
In addition, you will develop more advanced interactions via the command prompt. As a bonus, we added an optional challenge for those of you who would like to dive into some more advanced topics. This bonus task can be found in the supplemental material for this assignment.

The following new features need to be added:

- Additional commands to allow users to perform further actions on the system
- Command prompt parameters; allowing command specific values to be provided together with the command, on the same line
- Manage the players on the system, including the displaying of players on the system in specified sort orders
- Handling of invalid input via Exceptions
- Read and write the game-state variables to and from a file, persistently stored on the hard disk between executions
- Implement a new type of player - an AI (Artificial Intelligence) player, whose moves are automatically determined by the computer rather than a game user
- A victory guaranteed strategy for the AI player
- (**optional**) An advanced Nim game variant (for bonus marks see supplemental materials)

The system should still operate as specified in assignment 1 but accommodate these additional features. It is advised that you use your assignment 1 solution as a starting point for further implementation. Needless to say, you will need to fix any errors that may be present in your previous project.

This assignment will cover the following contents from lectures and tutorials:

- Implementation of Classes and Arrays
- Polymorphism, you may use either inheritance or interfaces to design the AI player in addition to the human player;
- Exceptions
- File I/O operations

2 Requirements

Go to *https://classroom.github.com/a/3djzcZB3* and accept the assignment. For details on how to check out the repository, make sure to consult the Lab 3 Materials[1].

In this project, we introduce a third class *NimGame*. The game playing process is delegated from Nimsys to NimGame. Since only one game will be active at any given time, only a single NimGame instance is required at any time by Nimsys. Nimsys should also maintain a collection of players. Initially, this collection will be empty - players will need to be added in order to play a game. Your design should be able to handle up to 100 players.
A NimGame instance needs to have the following information associated with it:

- The current stone count
- An upper bound on stone removal
- Two players

The system should allow for games of Nim to be played, with the rules of the game, and the players as specified by the user.
A player, as described by the NimPlayer class, needs to have the following information associated with it:

- A username
- A given name (firstname)
- A family name
- Number of games played
- Number of games won

The system should allow players to be added. It should also allow for players to be deleted, or for their details to be edited. Players should not be able to directly edit their game statistics but they should be able to reset them. This system will extend the command prompt from assignment 1, which will continue until an 'exit' command is issued, which will naturally terminate the program. If a command produces output, it should be printed to standard output (the terminal). When Nimsys is first executed, it should display a welcome message, followed by a blank line. As before, a command prompt (a 'dollar' sign, i.e., $) should then be displayed.

In the following description, all command line displays are put in a box for illustration purposes. **The box should NOT be printed out by your program, only the contents in the box should be printed.** The command prompt is illustrated as follows:

[1] https://canvas.lms.unimelb.edu.au/courses/1507/assignments/138029

```
Welcome to Nim

Please enter a command to continue or type 'help' for more information

$
```

At any given time, the system can be in one of two states - either a game is in progress, or no game is in progress. Hereafter, these will be referred to as the 'game' and 'idle' states, respectively. (Note: the states are just used to explain the mechanism of Nimsys. You don't need to create a variable called 'state' in your code).

When in the idle state, the system should accept the following commands:

`exit, addplayer, addaiplayer, removeplayer, editplayer, resetstats, displayplayer, rankings, startgame, startadvancedgame, commands, help`

More details of each command is provided in their corresponding sections below. These commands are entered at the Nimsys command prompt. If a command produces output, it should be printed immediately below the line where the command was issued. After the command is executed, a new command prompt should be displayed. This new command prompt should be separated from the previous command (and its output, if any) by a single blank line.

Some commands are able to accept zero or more parameters. This feature is describe below.

Note that in the syntax descriptions below, a term enclosed in square brackets indicates an optional parameter (e.g., *rankings [asc—desc]*. The input is assumed to be always valid, but not always correct. Valid input suggests that entered data have the same type of the corresponding variables, e.g., String data are entered for String variables, Integer data are entered for int variables. Correct input suggests that the entered data can be correctly processed by the corresponding command, e.g., adding an existing user and removing a nonexistent user are incorrect input. Unless otherwise stated, you are not required to check validity, but you need to check the correctness of the input, as shown in the below examples.

3 Commands

This section describes the functionality and expected output of each command your program needs to accommodate.

3.1 help

help adds a basic starting prompt to assist the user. Syntax: help

Example Execution:

```
$ help
Type 'commands' to list all available commands
Type 'startgame' to play game
The player that removes the last stone loses!
```

3.2 commands

commands displays a list of all available commands, including itself. You are free to develop your own implementation of this feature. One method might be to create an object (class) of type *Command*. An array containing all the objects of type Command might hold each command, the commands syntax use, and a method to display itself. Note the following output characteristics: The command number

has a space to the left, if the integer is a single digit. The required parameters, including optional parameters are displayed in line. Where no parameters are needed, this is specified. Commas separate each parameter, only. The parameters are surrounded by parentheses for illustration purposes. When the command is invoked, there are no parentheses around parameters.
Syntax: commands

Example Execution:

```
$ commands
 1: exit              (no parameters)
 2: addplayer         (username, secondname, firstname)
 3: addaiplayer       (username, secondname, firstname)
 4: removeplayer      (optional username)
 5: editplayer        (username, secondname, firstname)
 6: resetstats        (optional username)
 7: displayplayer     (optional username)
 8: rankings          (optional asc)
 9: startgame         (initialstones, upperbound, username1, username2)
10: startadvancedgame (initialstones, upperbound, username1, username2)
11: commands          (no parameters)
12: help              (no parameters)
```

3.3 addplayer

addplayer allows new players to be added to the game. If a player with the given username already exists, the system should indicate this, as shown in the example execution. Syntax: addplayer username, family_name, given_name

Example Execution:

1. add a new user:

```
$ addplayer lskywalker, Skywalker, Luke

$
```

2. add a user who already exists in the system

```
$ addplayer lskywalker, Skywalker, Luke
The player already exists.

$
```

3.4 removeplayer

removeplayer allows players to be removed from the game. The username of the player to be removed is given as an argument to the command. If no username is given, the command should remove all players, but in this case, it should display a confirmation question first. If a username for a non-existent player is given, the system should indicate that the player does not exist. The format of these messages is illustrated in the example execution below. Handling small caps 'y' and 'n' is sufficient.

Syntax: `removeplayer [username]`

Example Execution:

1. remove a nonexistent user

```
$ removeplayer lskyrunner
The player does not exist.

$
```

2. remove a user

```
$ removeplayer lskywalker

$
```

3. remove all users

```
$ removeplayer
Are you sure you want to remove all players? (y/n)
y

$
```

3.5 editplayer

editplayer allows player details to be edited. Note that the player's username cannot be changed after the player is created. If a username for a non-existent player is given, the system should indicate that the player does not exist, as illustrated in the example execution.
Syntax: `editplayer username, new_family_name, new_given_name`

Example Execution:

1. edit a nonexistent user

```
$ editplayer lskyrunner, Skywalker, Laurence
The player does not exist.

$
```

2. edit a user

```
$ editplayer lskywalker, Skywalker, Laurence

$
```

3.6 resetstats

resetstats erases statistics for a single or all players. If no username is given, the command should reset all players' statistics, but in this case, it should display a confirmation question first. If a username for a non-existent player is given, the system should indicate that the player does not exist.
Syntax: `resetstats [username]`

Example Execution:

1. reset a nonexistent user

```
$ resetstats lskyrunner
The player does not exist.

$
```

2. reset a user

```
$ resetstats lskywalker

$
```

3. reset all users

```
$ resetstats
Are you sure you want to reset all player statistics? (y/n)
y

$
```

3.7 displayplayer

displayplayer displays player information. The username of the player, whose information is to be displayed, is given as an argument to the command. If no username is given, the command should display information for all players, ordered by username in alphabetical order (from A to Z). If a username for a non-existent player is given, the system should indicate that the player does not exist, as illustrated in the example execution. **Please note when displaying player, the sequence of syntax is**

`username`, `givenname`, `familyname`, `number of games played`, `number of games won`

Syntax: `displayplayer [username]`

Example Execution:

1. display a nonexistent user

```
$ displayplayer lskyrunner
The player does not exist.

$
```

2. display a user

```
$ displayplayer lskywalker
lskywalker, Luke, Skywalker, 3 games, 3 wins

$
```

3. display all users

```
$ displayplayer
dvader, Darth, Vader, 7 games, 1 wins
hsolo, Han, Solo, 4 games, 3 wins
lskywalker, Luke, Skywalker, 3 games, 3 wins

$
```

3.8 rankings

rankings outputs a list of player rankings. Three columns should be displayed. The first column displays percentage of wins (winning ratio), the second column displays the number of games played, and the final column shows the player's full name, that is, first name followed by last name. This command takes the sort order as an argument. The sort order is `desc` (descending) by default. That is, if no argument or `desc` is provided, the program should rank the players by the percentage of games they have won in descending order, i.e., players with highest percentage wins should be displayed first. If the user provides `asc` as an argument, the players should be ranked by the percentage of games they have won in ascending order.

You need to round the percentages to the nearest integer value (no decimals). However, you should use the exact values of winning ratios when comparing and sorting two users' winning ratios. If two users have the same winning ratios (tie), sort them by usernames in alphabetical order. Only the first 10 players should be displayed, if there are more than 10. The output should be formatted according to the example below. For the purposes of formatting the output, you may assume that no player has played more than 99 games. Note that the vertical lines need to be aligned, with a single space appearing on either side. This means that in the first column you **must** have 5 characters consisting of a number, '%', and spaces. The first column must be left-justified.

Syntax: `rankings [asc|desc]`

Example Execution:

1. rank all users in descending order

```
$ rankings
100% | 03 games | Luke Skywalker
75%  | 04 games | Han Solo
14%  | 07 games | Darth Vader

$
```

2. rank all users in descending order

```
$ rankings desc
100% | 03 games | Luke Skywalker
75%  | 04 games | Han Solo
14%  | 07 games | Darth Vader

$
```

3. rank all users in ascending order

```
$ rankings asc
14%  | 07 games | Darth Vader
75%  | 04 games | Han Solo
100% | 03 games | Luke Skywalker

$
```

3.9 startgame

startgame creates and commences a game of Nim. In order to use this command, two players must first have been created using the *addplayer* command (see Section 3.3). You may assume that the initial stones and upper-bound arguments are valid **and correct**. However, if at least one (i.e., one or two) of

the usernames does not correspond to an actual player, the system should indicate this by prompting "`One of the players does not exist.`", in which case the game should not commence but return to the command prompt.

Otherwise, the 'startgame' command will commence a game, i.e., after executing it, the system is in the game state. When a game is in progress, the system should proceed according to the game play mechanics discussed in assignment 1, i.e., players should, in an alternating fashion, be asked to enter the number of stones they want to remove, with the game state being updated accordingly. In this project, bounds on stone removal should be enforced. That is, players should only be allowed to remove between 1 and N stones inclusive, where N is the upper bound or the number of stones remaining, whichever is smaller. Once all the stones are gone, a winner should be announced, and the statistics for the two players should be updated accordingly. The system should then return to the idle state, and the command prompt should be displayed again.

Syntax: `startgame (initialstones, upperbound, username1, username2)`

Example Execution:

1. start game with a non-existent user

```
$ startgame 10, 3, lskyrunner, hsolo
One of the players does not exist.

$
```

2. start a game

```
$ startgame 10, 3, lskywalker, hsolo

Initial stone count: 10
Maximum stone removal: 3
Player 1: Luke Skywalker
Player 2: Han Solo

10 stones left: * * * * * * * * * *
Luke's turn - remove how many?
3

7 stones left: * * * * * * *
Han's turn - remove how many?
4

Invalid move. You must remove between 1 and 3 stones.

7 stones left: * * * * * * *
Han's turn - remove how many?
3

4 stones left: * * * *
Luke's turn - remove how many?
3

1 stones left: *
Han's turn - remove how many?
0
```

```
Invalid move. You must remove between 1 and 1 stones.

1 stones left: *
Han's turn - remove how many?
1

Game Over
Luke Skywalker wins!

$
```

3.10 exit

exit terminates the Nimsys program.

Note: The use of `System.exit()` is often considered bad practice, as it can interfere with testing systems and can prevent program states from being saved, among other reasons. You should exit the program in a controlled manner, naturally; such that the game loop is exited and the end of your code is reached.
Syntax: `exit`

Example Execution:

1. exit the system

```
$ exit
```

If using an IDE, a typical program exit message will likely be produced, such as:

```
BUILD SUCCESSFUL (total time: 000 minutes 000 second)
```
(Although this is not important, nor created by you.)

4 Exception Handling

Your program should check inputs for validity. For this task, you will not be required to implement exception handling for all possible invalid inputs - just a subset of them as described in this specification document. The range of potential invalid inputs you are **required** to address using Exceptions (**not** via if-then statements) are listed below, along with the required behaviour of your program. You may need to refactor your assignment 1 code so that invalid input is handled via Exceptions.

4.1 Invalid Command

The user enters a command which is not a valid Nimsys command. Here, invalid command suggests the input command is not among the specified commands:
i.e.,

```
exit, addplayer, addaiplayer, removeplayer, editplayer, resetstats,
displayplayer, rankings, startgame, startadvancedgame, commands, help
```

Example:

```
$ createplayer lskywalker, Skywalker, Luke
'createplayer' is not a valid command.

$
```

4.2 Invalid Number Of Arguments

The user enters a valid Nimsys command, but does not provide the correct number of arguments. **Note:** You only need to check for insufficient number of arguments, and simply ignore any extra arguments, i.e., an insufficient argument count will generate an Exception while an excessive count will not. Different commands may have different number of arguments; your program should be able to check invalid number of arguments for **all** commands.

Example:

```
$ addplayer lskywalker
Incorrect number of arguments supplied to command.

$
```

4.3 Invalid Move

The player tries to remove an invalid number of stones from the game. For the move to be valid, it must be an integer between 1 and N inclusive, where N is the minimum of the upper bound and the number of stones remaining. Any other inputs (e.g. fractions, decimals, non-numeric entries) should be detected as invalid.

Example: *(Upper bound is 3 stones here)*

```
7 stones left: * * * * * * *
Han's turn - remove how many?
4

Invalid move. You must remove between 1 and 3 stones.

7 stones left: * * * * * * *
Han's turn - remove how many?
```

After implementing the invalid input checking, the scenarios detailed above should **not** cause your program to crash - rather, your program should display the appropriate error message, and continue execution, as illustrated in the examples. **You may assume that, aside from the cases explicitly mentioned above, the input to your program will be valid.**

5 The Player Statistics File

In previous labs and assignments, no program data were stored to disk. Therefore, all player data were lost when exiting the program. Understanding and utilizing persistent data (saved to and accessed from disk) is an invaluable and important process.

Here, the task is to store your game-state data upon exiting the program, and to restore them on subsequent executions. Thus, if one was to exit your program (using the 'exit' command), and then start it again (by running 'java Nimsys' at the shell prompt), your program should be restored to the state it was in immediately before exiting. That is, it should be as if the program never exited at all.

This can be achieved by storing your player data in a file. At the beginning of the execution of your program, if the file exists, it is opened and its contents loaded into the system. When your program exits, this file will be updated with the new modified players data, and then closed. If the file does not already exist, it will need to be created. It is up to you to decide the most appropriate format, e.g., text

or binary, of this file. The name of the file should be **players.dat**, and it should be stored in the same directory as your program.

All player information should be stored, i.e., usernames, given names, family names, and number of games played and won. Note that you do not need to store information about games in progress, since a game could/should never be in progress when the program exits properly, i.e., via the 'exit' command.

6 The AI (Artificial Intelligence) player

For this functionality, a new type of player class needs to be added: an AI player.

This player type should be controlled by the program, not by a human player. Aside from this, an AI player should be the same as a human player. That is, they should have all the same information associated with them (i.e., username, given/family names, and number of games played and won). They should be stored in the system and appear in player lists and rankings, just as human players are. They should also be manipulated via all the same commands (with the exception of `'addplayer'`, since we now need to indicate whether we are adding a human player or an AI player to the system - see below).

The only difference between a human player and an AI player is in the way that they make a move. Instead of prompting for a move to be entered via standard input, the AI player should choose their own move, based on the state of the game. Thus, the only difference between a human player and an AI player should be in the method used to make a move. This suggests that the object-oriented principle of *polymorphism* should be applied here. Java offers polymorphism via two main avenues - inheritance and interfaces. In this case, inheritance is the more appropriate choice. Conceptually speaking, we can think of human players and AI players as specialized players. i.e., a human player *is a* player, and an AI player *is a* player. They are identical in almost every way, and so most of their attributes and methods can be inherited. The only exception to this is the method used to make a move, which will need to be rewritten to act autonomously.

You can add an abstract `NimPlayer` class, which will be used to represent the behaviour and attributes common to both Human and AI players. You can modify your original `NimPlayer` class used from assignment 1 to be the new abstract `NimPlayer` class, and the human player class (`NimHumanPlayer`) and AI player class (`NimAIPlayer`) can extend the abstract player class.

Part of your mark for this project will be based on how well you apply polymorphism in your implementation of the human and the AI player. Therefore, it is important that you *do* use the principle of polymorphism in your design.

To allow for AI players to be added to the system, you should create a new command - `'addaiplayer'`. This command should operate in exactly the same way as `'addplayer'`. The only difference is that the resulting player is an AI player. Note that all other commands, e.g., `'removeplayer'` and `'editplayer'` should work for both human players and AI players. Provided below is an example of the use of the `'addaiplayer'` command:

```
$ addaiplayer artoo, D2, R2

$
```

In this task, you need to modify the provided **NimAIPlayer.java** to implement the AI player functionality.

After implementing the AI player, the `'startgame'` command should allow games to be started with one or both players being AI players. The game should proceed exactly as per the assignment 1 spec. Except

that when it comes to an AI player's turn, there should be no reading of input from standard input. Instead, the move should be immediately made by the AI. Provided below is an example execution. Here, Luke is a human player, and R2 D2 is an AI player and have already been created using the `'addplayer'` and `addaiplayer` respectively.

```
$ startgame 10, 3, lskywalker, artoo

Initial stone count: 10
Maximum stone removal: 3
Player 1: Luke Skywalker
Player 2: R2 D2

10 stones left: * * * * * * * * * *
Luke's turn - remove how many?
3

7 stones left: * * * * * * *
R2's turn - remove how many?

5 stones left: * * * * *
Luke's turn - remove how many?
3

2 stones left: * *
R2's turn - remove how many?

1 stones left: *
Luke's turn - remove how many?
1

Game Over
R2 D2 wins!

$
```

The move an AI player makes given a specific situation shall follow a certain strategy, such that the victory is guaranteed for the AI player if it holds the ability to win when the game commences. For details, please see the following section.

7 Victory Guaranteed Strategy for AI Players

When the number of remaining stones satisfy certain conditions, the player about to make a move may have a strategy to guarantee the victory. In this section, your task is to implement this strategy for the AI player as its `removeStone()` method. Please note that the `removeStone()` method will determine and return the number of stones to be removed. We describe such strategy in the following paragraph:

Simply, for a player to guarantee the victory no matter how the rival player moves in the future, it needs to ensure that the rival player is always left with $k(M+1)+1$ stones. Where $k \in \{0, 1, 2, ...\}$ and M is the maximum number of stones can be removed at a time. Additionally, the commencing condition should satisfy that the rival player first moves and there are $k(M+1)+1$ stones. Or alternatively, in every winning player's turn there are some number of stones, which cannot be expressed as $k(M+1)+1$.

The task is to implement the AI player's behaviour such that it always wins if **either one of the following holds**:

(a) initial number of stones is $k(M+1)+1$, where $k \in \{0, 1, 2, ...\}$, and the rival player moves first or

(b) initial number of stones is **not** $k(M+1)+1$, where $k \in \{0, 1, 2, ...\}$, and the rival player moves second

For example, suppose Winfred and Louise are playing a game where a maximum of M stones can be removed each turn. Here, the notation [a, b] will be used to indicate all integers from a to b inclusive. So the number of stones a player must remove is in the range [1, M]. For Winfred to ensure a victory, the possible game states are ([a, b] means there are at least a and up to b stones left to be removed by the players):

(i) Louise's turn, 1 stone left

(ii) Winfred's turn, [2, M+1] stones left

(iii) Louise's turn, (M+1)+1 stones left

(iv) Winfred's turn, [(M+1)+2, 2(M+1)] stones left

(v) Louise's turn, 2(M+1)+1 stones left

(vi) Winfred's turn, [2(M+1)+2, 3(M+1)] stones left

(vii) Louise's turn, 3(M+1)+1 stones left

(viii) and so on...

Notice that Winfred always has a way of moving to the next state, while Louise is always forced to move to the next state. i.e., the numbers of stones left for Louise are fixed in each of her turns.

If the winning condition does not hold, the AI player can remove any number of stones up to M, and hope the other player makes a mistake.
Your task is to implement this strategy for the AI Player to put the AI on a path to win.

8 Important Notes

Automatic tests will be conducted on your program by compiling, running, and comparing your outputs for several test cases with generated expected outputs. The automatic test will deem your output wrong if your output does not match the expected output. Therefore, make sure that **your output follows exactly the same format shown in the examples above.**

The syntax `import` is available for you to use standard java packages. However, **DO NOT** use the `package` syntax to customize your source files. The automatic test system cannot deal with customized packages. If you are using Netbeans, IntelliJ or Eclipse for your development, be aware that the project name may automatically be used as the package name. You must remove lines like

```
package Assignment2;
```

at the beginning of the source files before you commit them to GitHub. Unlike in assignment 1, GitHub will not run tests on your code. **You are responsible for running your tests locally!**

Use **ONLY ONE** Scanner object throughout your program. Otherwise the automatic tests may cause your program to generate exceptions and terminate. The reason is that in the automatic test, multiple lines of test inputs are sent all together to the program. As the program receives the inputs, it will pass them all to the currently active Scanner object, leaving the rest Scanner objects nothing to read and hence cause run-time exception. Therefore, it is important that **your program has only one Scanner object.**

9 Assessment

This project is worth 15% of the total marks for the subject. Remember that there is a 50% hurdle requirement (i.e. 20/40) for the combined scores from the quiz and the two assignments.

Your Java program will be assessed based on correctness of the output as well as quality of code implementation.

Note that if you attempt the bonus task, you can get 1.5 bonus marks. The total mark for assignment 2 is up to 16.5. However, the total mark for the two assignments cannot exceed 30. The bonus task is found in the supplemental material for this assignment.

This is a challenging assignment and it takes time to complete, make sure to allocate your time wisely and start early.

10 Testing Before Submission

You will find sample input files and the expected ouput files in the template repository to use on your own. You should use them to test your code thoroughly before submitting your code to GitHub. Note that each commit you make is recorded in your GitHub repository. Details of how the submission works can be found in Section 11.
To test your code follow these steps:

1. Check your Java code files and make sure you have the test input data files ready "`testin1`".
2. Open a console, navigate to your project directory (where your .java classes reside), and run compile your program: `javac *.java` (this command will compile all your java file in the current folder)
3. Run the command: `java Nimsys < testin1 > output` (this command will run the Nimsys using contents in '`testin1`" as input and write the output in `output`)
4. Inspect the file `output` as it contains any errors your program execution may have encountered.
5. Compare your result with the provided output file `testout1`. Fix your code if they are different.
6. When you are satisfied with your project, commit your changes to GitHub. No further tests will be done through GitHub so it is up to you to check your program's syntax and semantics.

NOTE: The test cases used to mark your submissions will be different from the sample tests given. You should test your program extensively to **ensure it is correct for other input values** with the same format as the sample tests.

11 Submission

Your submission must have all required files from assignment 1 plus all the source files necessary to run your program. This includes any class file you may have created locally. If you correctly cloned the assignment repository, some mandatory skeleton files should already be in your working directory.

Note that *.java includes all Java files in the current directory, so you must make sure that you don't include irrelevant Java files in the current directory. You should verify your submission locally as described above before submitting your code to GitHub.

Make sure to fill in your information into the *authorship.txt* declaration, which is included in the skeleton code. If you fail to do so, there is a chance we cannot allocate your submission and you run the risk of scoring 0 on the assignment.

You can edit and re-submit your code to GitHub as many times you want as long as you submit before the submission deadline of the assignment.

A good practice to test your code is to clone your repository into a fresh directory and try and run your program there.

The deadline for the project is **5pm (AEST), October 15, 2020**.

What will be graded? The **last** version of your program committed before the submission deadline.

11.1 Late-Submission Penalties

There is a 20% penalty per day for late submissions.

For example, suppose your project gets a mark of 5 but changes are submitted within 1 day after the deadline, then you get **20% penalty** and the mark will be 4 after the penalty.

Any updates to your code on GitHub made after the submission deadline will incur a late-submission penalty. This means that even if you make a minor edit and re-submit a file after the deadline your entire submission will be subject to a late-submission penalty! Submissions via email will not be accepted.

There will be **0 marks for your submission if you make changes after the 5pm (AEST), October 19, 2020**.

12 Individual Work

Note well that this project is part of your final assessment, so copying, working together, sharing work (i.e. cheating) is not acceptable! Any form of material exchange, whether written, electronic or any other medium is considered cheating, as is copying from any online sources in case anyone shares it. Providing undue assistance is considered as serious as receiving it, and in the case of similarities that indicate exchange of more than basic ideas, **formal disciplinary action will be taken for all involved parties without exceptions!** A sophisticated program that undertakes deep structural analysis of Java code identifying regions of similarity will be run over all submissions in "compare every pair" mode.