The University of Melbourne
School of Computing and Information Systems
COMP90041 Programming and Software Development

Lecturer: Dr Tilman Dingler & Dr Thuang Pham

Semester 2, 2020

Assignment 1

Due: 5pm (AEST), September 17, 2020

1 Background

This project is the first of two, with the objective of designing and implementing a simple variant of the ancient game known as **Nim**. The game will be written using Java and played as a text-based game through the standard output console. As part of the application, you will also create a basic command line input console, that accepts one of four commands; see section 3.

The game Nim is a strategical game, whereby two players take turns in *Nimming* (removing) arbitrary objects from one (or more) distinct pile(s). Each player takes a turn to remove at least one (or often more) object(s) from one pile at a time. The maximum number of objects removable per turn is decided at the beginning of the game. This number is known as an upper limit. Depending on the version being played, the goal of Nim is either to avoid taking the last object or to take the last object. Nim is always a two-player game but can be played with a myriad of varying rules.

The rules for this game variant will be as follows:

- The game begins with n number of objects placed in one pile (e.g. stones placed on a table).
- Each player takes turns removing one or more objects from the pile.
- The maximum number of objects a given player can take is limited by an upper bound; k. For example, if this upper bound is $k = 3$, a player must remove $1, 2$ or 3 objects from the pile on each turn.
- The game ends when there are no more stones remaining; $n = 0$.
- The player who removes the last stone, loses. The other player is, of course, the winner.
- The initial number of stones (n) and the upper bound (k) can be changed from game to game and are chosen before each game commences.

Play-through Rules Example
Here is an example play-through of the game, using 12 initial stones, and an upper bound of 3 (stones removed per turn).

- There are 12 stones on the table.
- Player 1 removes 3 stones. 9 stones remain.
- Player 2 removes 1 stone. 8 stones remain.
- Player 1 removes 1 stone. 7 stones remain.
- Player 2 removes 2 stones. 5 stones remain.
- Player 1 removes 3 stones. 2 stones remain.
- Player 2 removes 1 stone. 1 stone remains.
- Player 1 removes 1 stone. 0 stones remain.
- Player 2 wins. Player 1 loses.

2 Your Task

Go to *https://classroom.GitHub.com/a/FIW3k0j2* and accept the assignment. For details on how to check out the repository, make sure to consult the Lab 3 Materials[1]. Once you have cloned the repository, you will find two classes in it:

```
Nimsys.java
NimPlayer.java
```

`Nimsys.java` will be the main application containing your `main()` method. It will contain all the methods needed for the command console and Nim game to execute. You are free to create all the necessary methods required for your application.

`NimPlayer.java` will be an object class, containing the player `name`, how many `wins` the player has achieved and how many `games` the player has played. It will also provide the appropriate accessor and mutator methods (i.e. `getters and setters`) for effectively utilizing the `NimPlayer` object.

In addition to developing the game (Nim), you will be required to create a command-line input console, that can recognize and action several commands. The game (Nim) will launch from within your created command-line console. You will program the command console to respond to commands when typed by the user; such that the command line console will perform tasks based on the commands entered by the user. The commands are as follows:

Commands: `start, exit, help, commands`

A description of the functionality of each command is described in section 3.

The game will begin by typing in the command `start` into the command console. When first run, the game should ask for the names of both player one and player two, respectively. The user is then prompted for the upper bound and the number of objects (stones) for that particular game round. Each time the game is repeated, the upper bound and number of objects will need to be entered again. The player names should be entered only at the beginning, when launching the game from the command console. While in game (i.e. playing Nim), alternate players are continually prompted to make a selection of how many stones to remove from the pile. Thus, the game asks each players in turn for their selection, in a

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

loop, until one player wins.

Note: Each players selection must be legal, to the game rules; such that the number of stones must be within the upper bounds and less than or equal to the stones left on the pile. Therefore, a player must take one or more stones. They cannot take more stones than is specified in the upper bound limit and they cannot take more than what is left on the pile. If an illegal move is made, the game will show an error message and continue to prompt that particular player until a legal selection is provided. Here are two cases your program should be able to deal with:

- If a player tries to remove more stones than the upper bound, the following error message should be displayed:

  ```
  Upper bound limit exceed, upper bound maximum choice is N
  ```

  where *N* should be the upper bound previously set.

- If a player tries to remove more stones than are remaining (or less than 1), the following error message should be displayed:

  ```
  Invalid attempt, only M stones remaining! Try again:
  ```

  where *M* should be set as the number of remaining stones.

For this assignment, all user inputs will be assumed to be the correct data types; such that if a String is expected then it is assumed a string will be entered, and the same for an integer. No error control for incorrectly entered data types are required... as of yet!

When the game is won, the option to play again is prompted. If the players choose to play again, the upper bound and stones on the table are prompted from the players. The player names are *not* required in this case. If the player chooses **not** to play again, each player game statistics is displayed, showing how many wins and how many games each player has played. The game ends and control is returned to the command console. The game can be restarted by typing `start`, again. If a new game is started from the console, new player names are prompted for once more; and the sequence of playing Nim starts again, until a player wins. To exit the application, the user must enter the command `exit`, from the command console. The command console will continue to respond to commands until existed, or when a game is in progress. For now, if an *unknown* command is entered, the *unknown* command is ignored and the console $ symbol is simply repeated.

3 Commands: Command Console

The command console provides text based functionality to the user. When a user enters a command, it will invoke a method to produce a specific action. After fully executing a command, control is returned to the command console; unless the command `exit` is entered. The commands to be implemented for this assignment are as follows. Please note, the method names are only suggestions.

- `start` - invokes the `startGame()` method.
  This method will trigger the start of the game; Nim. The game will repeat until the players decides not to continue.

- `exit` - invokes the `exit()` method.
  This method will exit the command console environment, thus exiting the application. As is good coding practice, the exit method will terminate the applications operations naturally. This means **not** using Java's forceful `System.exit()` function.

- `help` - invokes the `help()` method.
  This method will display a message to assist the user to play a game and to type `commands` to display a list of all the available commands.
- `commands` - invokes the `commandList()` method.
  This method will display all the commands available to the user. This should be accomplished using a loop; thus enabling more commands to be easily implemented in the future.

To recap, this assignment has two sub-parts. The first part will provide a command console for a user to continually enter single string based commands. The second part will run the game Nim, prompting alternating players to select stones, until there is a winner. When a winner is determined and the game is not continued, control is returned to the command console. The application is terminated with the command `exit`.

4 Application Walk-through

1. Your program will begin by displaying a welcome message.
2. The program will then display a command-prompt, showing the symbol $.
3. This command-prompt will form the base of your game, allowing one of the four commands to be entered (*see section 3*).
4. After selecting one of the four commands, your program will call the corresponding method.
5. After `start` is entered, the method to action the start of the game is invoked, implementing a controlled game loop.
6. The program will then prompt for a string (no space in the string) to be entered via standard input (the keyboard) - this will be the name for player 1.
   You may assume that all inputs to the program will be valid; i.e., an integer inputted for an integer, a string is inputted for a string, etc.
7. The program will then prompt for another string (no space in the string) to be entered - this will be the name for player 2.
8. The program will then prompt for an integer to be entered - this will be the upper bound, i.e. the number of stones that can be removed in a single turn.
9. The program will then prompt for another integer to be entered - this will be the initial number of stones.
10. The program will then print the number of stones and will also display the stones, which will be represented by asterisks '*'.
11. The program will then prompt for another integer to be entered - this time, a number of stones to be removed.
    Again, you may assume this input will be valid but you will need to check if the input exceeds the number of stones remaining or the upper bound on the number of stones that can be removed.
12. The program should then show an updated display of stones.
13. The previous two steps should then be repeated until there are no stones remaining. When this occurs, the program should display '`Game Over`', and the name of the winner.
14. The program should then ask user whether the players wanted a play again. The user is prompted to enter 'Y' for yes or 'N' for no. If the user enters 'Y', that is, the user wants to play another game, repeat steps 8-14. Otherwise, the program should terminate. Note, any input apart from 'Y' should terminate the game.

4.1 Example execution:

```
Welcome to Nim

Please enter a command to continue

$ start

Please enter Player 1's name : Luke
Please enter Player 2's name : Han
Enter upper bound : 3
Enter initial number of stones : 12

12 stones left : * * * * * * * * * * * *
Luke's turn. Enter stones to remove : 4
Upper bound limit exceed, upper bound maximum choice is 3

Luke's turn. Enter stones to remove : 3

9 stones left : * * * * * * * * *
Han's turn. Enter stones to remove : 1

8 stones left : * * * * * * * *
Luke's turn. Enter stones to remove : 1

7 stones left : * * * * * * *
Han's turn. Enter stones to remove : 2

5 stones left : * * * * *
Luke's turn. Enter stones to remove : 3

2 stones left : * *
Han's turn. Enter stones to remove : 1

1 stones left : *
Luke's turn. Enter stones to remove : 1

Game Over
Han wins!

Do you want to play again (Y/N): Y
Enter upper bound : 5
Enter initial number of stones : 15

15 stones left : * * * * * * * * * * * * * * *
Luke's turn. Enter stones to remove : 1

14 stones left : * * * * * * * * * * * * * *
Han's turn. Enter stones to remove : 2

12 stones left : * * * * * * * * * * * *
Luke's turn. Enter stones to remove : 3

9 stones left : * * * * * * * * *
Han's turn. Enter stones to remove : 4
```

```
5 stones left : * * * * *
Luke's turn. Enter stones to remove : 5

Game Over
Han wins!

Do you want to play again (Y/N): n
Luke won 0 game out of 2 played
Han won 2 games out of 2 played
$
$ help
Type 'commands' to list all available commands
Type 'start' to play game
Player to remove the last stone loses!

$ commands

: start
: exit
: help
: commands

$ fakecommand
$
$ exit

Thank you for playing Nim
```

Please note that:

- The class `Nimsys` must contain the `main()` method to manage the above game playing process.
- The application begins with a welcome, followed by the $ symbol; representing a prompt to enter a command.
- Once start is executed and a player's name is entered, a new object of the `NimPlayer` class should be created. You need to create the class `NimPlayer`. Player 1 and Player 2 are two instances of this class. This class should have a `String` typed instance variable representing the player name. This class should also have a `removeStone()` method that returns the number of stones the player wants to remove in his/her turn. You will lose marks if you fail to create two instances of `NimPlayer`.
- Add other variables and methods where appropriate such that the concept of information hiding and encapsulation is reflected by the code written.
- There is **NO** blank line before the first line, i.e., no `println()` before '`Welcome to Nim`'.
- There is a white-space between `stones left :` sentence and `* * *`, and also between all `* * *`, but **NO** withespace at the end of `* * *`.
- The line that prints the winners name should be **a full line**, i.e., '`Han wins!`' is printed out using `println()`.
- And there are **NO** blanks after the last stone in lines displaying asterisks.
- Keep a good coding style.

You do not need to worry about changing your output for singular/plural about stones (it's always *stones*), but you should check your output for singular/plural about displaying > 1 games i.e., you should output '1 stones' and 'Han won 2 games out of 2 played'.

5 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, even if the difference is just having an **extra space or missing a colon**. Therefore, it is crucial 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 Assignment1;
```

at the beginning of the source files before you commit them to GitHub. Otherwise the automatic tests will fail and tell you so.

Also, 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.** Arguments such as "It runs correctly when I do manual test, but fails under automatic test" will not be accepted.

6 Assessment

This project is worth 15% of the total marks for the subject.

Your Java program will be assessed based on correctness of the output as well as quality of code implementation. A detailed marking scheme will be released on Canvas.

7 Testing Before Submission

You will find the sample input file and the expected ouput file in the Projects page on Canvas for you to use on your own. You should use them to test your code on your own first, and then submit your code on GitHub. Note that each commit you make is recorded in your GitHub repository. Details of how the submission works can be found in Section 8.
To test your code by yourself:

1. Check your Java code files named "`Nimsys.java`" and "`NimPlayer.java`", and make sure you have the test input data files ready "`test0.txt`".
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. In your repository you will find test input and output files, such as *testin1*. Run 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 and verify your project inspecting the automated test output (see Section 8).

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.

8 Submission

Your submission should have at least two Java source code files. You must name them `Nimsys.java` and `NimPlayer.java` (if you correctly cloned the assignment repository, these 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.
Once you commit your code, GitHub will run automated tests. This can take a few minutes. You will be shown the results of the tests on GitHub where you can manually inspect any discrepancy between your program's output and what was expected. Passing all tests does not guarantee that your code is correct. For grading purposes we will run a series of tests that go beyond the tests run on GitHub so make sure to thoroughly test your code according to the specifications. You can edit and re-submit your code as many times you want as long as you submit before the submission deadline of the assignment.

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!

"I can't get my code to work on GitHub's server but it worked on my local machine" is <u>not</u> an acceptable excuse for late submissions. Also, submissions via email will not be accepted!

The deadline for the project is **5pm (AEST), September 17, 2020**. The allowed time is more than enough for completing the project.

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

8.1 Late-Submission Penalties

<u>**There is a 20% penalty per day for late submissions.**</u>

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.
There will be **0 marks for your submission if you make changes after the 5pm (AEST), September 21, 2020**.

9 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.