The University of Melbourne
School of Computing and Information Systems
COMP90041 Programming and Software Development
Lecturer: Prof. Rui Zhang
Semester 1, 2019

Project A
Due: 4pm, Thursday 4 April, 2019

1 Background

This project is the first in a series of three, with the ultimate objective of designing and implementing (in Java) a simple variant of the game of Nim. It is a two player game, and the rules of the version used here are as follows:

- The game begins with a number of objects (e.g., stones placed on a table).
- Each player takes turns removing stones from the table.
- On each turn, a player must remove at least one stone. In addition, there is an upper bound on the number of stones that can be removed in a single turn. For example, if this upper bound is 3, a player has the choice of removing 1, 2 or 3 stones on each turn.
- The game ends when there are no more stones remaining. The player who removes the last stone, loses. The other player is, of course, the winner.
- Both the initial number of stones, and the upper bound on the number that can be removed, can be varied from game to game, and must be chosen before a game commences.

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

For this first project, the focus will be on creating two players and playing for multiple games:

1. Your program will begin by displaying a welcome message.
2. 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 of Player 1. You may assume that all inputs to the program will be valid.
3. The program will then prompt for another string (no space in the string) to be entered - this will be the name of Player 2.
4. The program will then prompt for an integer to be entered - this will be the upper bound on the number of stones that can be removed in a single turn.
5. The program will then prompt for another integer to be entered - this will be the initial number of stones.
6. The program will then print the number of stones, and will also display the stones, which will be represented by asterisks '*'.
7. 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 and will not exceed the number of stones remaining or the upper bound on the number of stones that can be removed.
8. The program should then show an updated display of stones.
9. 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.
10. 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 4-10. Otherwise, the program should terminate. Note, any input apart from 'Y' should terminate the game.

Here is an example execution:

```
Welcome to Nim

Please enter Player 1's name:
Luke

Please enter Player 2's name:
Han

Please enter upper bound of stone removal:
3

Please enter initial number of stones:
12

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

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

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

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

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

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

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

Game Over
Han wins!

Do you want to play again (Y/N):Y
Please enter upper bound of stone removal:
5

Please enter initial number of stones:
15

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

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

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

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

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

Game Over
```

```
Han wins!

Do you want to play again (Y/N):N
```

Please note that:

- You need to create a class called `Nimsys` with a `main()` method to manage the above game playing process.
- When 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`'.
- 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 entities, i.e., you should output '1 stones', etc.

Important Notes

Computer automatic test will be conducted on your program by automatically 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, please **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 as the IDE, please be aware that the project name may automatically be used as the package name. Please remove the line like

```
package ProjA;
```

at the beginning of the source files before you submit them to the system.

Please use **ONLY ONE** Scanner object throughout your program. Otherwise the automatic test will 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 crucial 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.

3 Assessment

This project is worth 10% 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. See LMS for a detailed marking scheme.

4 Testing Before Submission

You will find the sample input file and the expected ouput file in the Projects page on LMS for you to use on your own. You should use them to test your code on your own first, and then submit your code. Note that you must submit your code through the submit system in order to make a valid submission of the project, details of submit system can be found in Section 5.
To test your code by yourself:

1. Upload to the server your Java code files named "`Nimsys.java`" and "`NimPlayer.java`", and the test input data files "`test0.txt`".
2. Run command: `javac *.java` (this command will compile all your java file in the current folder)
3. Run command: `java Nimsys < test0.txt > output.txt` (this command will run the Nimsys using contents in '`test0.txt`" as input and write the output in `output.txt`)
4. Run command: `more output.txt` (this command will show the your program's output)
5. Compare your result with the provided output file `test0-output.txt`. Fix your code if they are different.
6. When you are satisfied with your project, submit and verify your project using the instructions given in the Section 5.

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.

5 Submission

Your submission should have two Java source code files. You must name them `Nimsys.java` and `NimPlayer.java`, and store them in a directory under your home directory on the Engineering School student server. Then, you can submit your work using the following command:

```
submit COMP90041 projA *.java
```

You should then verify your submission using the following command. This will store the verification information in the file '`feedback.txt`', which you can then view:

```
verify COMP90041 projA > feedback.txt
```

You should issue the above commands from within the same directory as where the file is stored (to get there you may need to use the `cd` 'Change Directory' command). Note that you can submit as many times as you like before the deadline.

How you edit, compile and run your Java program is up to you. You are free to use any editor or development environment. However, **you need to ensure that your program compiles and runs correctly on the Engineering School student server**, using **build 1.8.0** of Oracle's (as Sun Microsystems has been acquired by Oracle in 2010) Java Compiler and Runtime Environment, i.e. `javac` and `java` programs.
Submit your program to the Engineering School student servers **a couple of days before the deadline** to ensure that they work (you can still improve your program and submit it again). **"I can't get my code to work on the student server but it worked on my Windows machine" is not an acceptable excuse for late submissions. Email submissions will not be accepted either.**

The deadline for the project is **4pm, Thursday 4 April, 2019**. The allowed time is more than enough for completing the project. **There is a 20% penalty per day for late submissions. Suppose your project gets a mark of 5 but is submitted within 1 day after the deadline, then you get 20% penalty and the mark will be 4 after penalty. There will be no mark for submissions after 4pm 8 April.**

6 Individual Work

Note well that this project is part of your final assessment, so cheating is not acceptable. Any form of material exchange, whether written, electronic or any other medium is considered cheating, and so 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. 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.

The University of Melbourne
School of Computing and Information Systems
COMP90041 Programming and Software Development
Lecturer: Prof. Rui Zhang
Semester 1, 2019

Project B
Due: 3pm, Monday 6 May, 2019

1 Introduction

This project (Project B), and the next one (Project C), will continue with the same theme introduced in Project A - namely, an implementation of the game of Nim. Please refer to the Project A specification for a description of the game and its rules.

In Project A, a simple Java program was created to handle Nim's core game play mechanics. In the next two projects, the objective is to design and implement a more complete version of Nim, making full use of Java's object-oriented paradigm.

Key Knowledge Points Covered in Project B:

1. Design of the class structure for the project requires the knowledge of UML diagrams (taught in Week 5).
2. Implementation requires understanding of *Classes* (taught in Week 4 - 6) and *Arrays* (taught in Week 7).

You may start working on the project right away except the array part. If you would like to start to work on the array part before week 7, you may learn basics of arrays by yourself.

2 Requirements

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.

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, 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
- 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, although they should be able to reset them.

The system is a text based interactive program that reads and executes commands from standard input (the keyboard) until an 'exit' command is issued, which will 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**. A command prompt (a 'dollar' sign, i.e., `$`) should then be displayed.

In following description, all command line displays are put in a box. This is only for easier understanding the format. **The box should NOT be printed out by your program, only the contents in the box should be printed.** The command prompt is illustrated below:

```
Welcome to Nim

$
```

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. 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 has 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.

Note that in the syntax descriptions below, a term enclosed in square brackets indicates an optional parameter. 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 validness, but you ARE required to check the correctness of the input, as shown in the below examples.**

1. 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:

   (a) add a new user:

   ```
   $addplayer lskywalker,Skywalker,Luke

   $
   ```

   (b) add a user who already exists in the system

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

   $
   ```

2. 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.

   Syntax: `removeplayer [username]`

   Example Execution:

   (a) remove a nonexistent user

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

   $
   ```

   (b) remove a user

   ```
   $removeplayer lskywalker

   $
   ```

   (c) remove all users

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

   $
   ```

3. 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:

   (a) edit a nonexistent user

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

   $
   ```

(b) edit a user

```
$editplayer lskywalker,Skywalker,Laurence

$
```

4. resetstats - Allows player statistics to be reset. The username of the player whose statistics are to be reset is given as an argument to the command. If no username is given, the command should reset all player statistics, but as with the 'removeplayer' command, a confirmation question should be displayed in this case. 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: `resetstats [username]`

   Example Execution:

   (a) reset a nonexistent user

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

   $
   ```

   (b) reset a user

   ```
   $resetstats lskywalker

   $
   ```

   (c) reset all users

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

   $
   ```

5. 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 alphabetically. 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:

   (a) display a nonexistent user

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

   $
   ```

   (b) display a user

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

   $
   ```

(c) 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

$
```

6. rankings - Outputs a list of player rankings. There are three columns displayed. The first column displays percentage wins or 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` or 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. Round the percentages to the nearest integer value. However, you should use the exact values of winning ratios when comparing and sorting two users' winning ratios. Ties should be resolved by sorting on usernames alphabetically. 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:

   (a) rank all users in descending order

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

$
```

   (b) rank all users in descending order

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

$
```

   (c) rank all users in ascending order

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

$
```

7. startgame - Creates and commences a game of Nim. The game's rules, and the usernames of the two players, are provided as arguments. You may assume that the initial stones and upperbound arguments are valid **and correct**. However, if at least one (i.e. one or two) of the

usernames doesn't correspond to an actual player, the system should indicate this by the output "`One of the players does not exist.`", and the game should not commence.

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 Project A, i.e., players should, in an alternating fashion, be asked to enter the number of stones they would like 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 a command prompt should be displayed again.

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

Example Execution:

(a) start game with a non-existent user

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

$
```

(b) 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!

$
```

8. exit - Exits the Nimsys program.

   Syntax: `exit`

   Note that before you call the exit method in Java using `System.exit(0);` , you must print a blank line first.

   (a) exit the system

```
$exit
```

As was described earlier, it is important that your design makes good use of object-oriented design principles. This is particularly relevant when it comes to implementing the actual gameplay. **In a real game, game proceeds with each player performing the action of removing some number of stones from the game. Your design should reflect this structure.**

Don't worry about changing your output for singular/plural entities. Simply always use plural entities, i.e., you should output '1 games', '1 wins', '1 stones', etc.

Checklist For Solution

- **Error handling issues**
  Only the following errors need to be handled (you may choose to handle more if you wish):
  - ☐ Adding a new player with an existing username.
    Error message: The player already exists.
    Follow-up operation: Print '$' and prompt for user input again.
  - ☐ Removing a player with a non-existing username.
    Error message: The player does not exist.
    Follow-up operation: Print '$' and prompt for user input again.
  - ☐ Resetting the statistics of a player with a non-existing username.
    Error message: The player does not exist.
    Follow-up operation: Print '$' and prompt for user input again.
  - ☐ Displaying a player with a non-existing username.
    Error message: The player does not exist.
    Follow-up operation: Print '$' and prompt for user input again.
  - ☐ Starting a game where at least one of the player's username does not exist.
    Error message: One of the players does not exist.
    Follow-up operation: Print '$' and prompt for user input again.
  - ☐ Removing stone outside of the range [1, `stoneRemovalUpperBound`], where `stoneRemovalUpperBound` is the maximum number of stones allowed to be removed in one turn.
    Error message: Invalid move. You must remove between 1 and `stoneRemovalUpperBound` stones (do not need to consider the singular or plural form of "stone").
    Follow-up operation: Prompt for the same player to remove stone again.
- **Blank line and whitespace related issues.**
  - ☐ Make sure that between the output of last command (including indication for nonexistent users, confirmation for all-user operations, display results, and game results) and the next command prompt, there is one blank line.
  - ☐ Make sure that there is a whitespace between `(y/n)` and `Are you sure...` sentence.
  - ☐ Make sure that there is NO whitespace after each comma, e.g., when displaying users, it should be `davader,Darth` instead of `davader, Darth`.
  - ☐ Make sure that in game state, there is one blank line before the `Initial stone count...` and one blank line after the `Player 2:...`.
  - ☐ Make sure that in game state, if a user input an invalid move, there is one blank line between the user-input move and the indication sentence `Invalid move...`.
  - ☐ Make sure that an extra blank line is printed out before calling `System.exit(0);` when command `exit` is entered.
- **Ranking display related issues.**
  - ☐ Make sure that the ranking is in a descending order of winning ratio by default or when "desc" is provided as an argument.

- ☐ Make sure that the ranking is in a ascending order of winning ratio when "asc" is provided as an argument.
- ☐ Make sure that winning ratio is rounded to the nearest integer value when displaying the winning ratio. However, please note that exact values are still used when comparing and sorting two users winning ratios.
- ☐ Make sure that users with the same winning ratio are sorted according to the alphabetical order of the user name irrespective of the argument supplied to the rankings command, e.g., if username ethan and username tom have the same winning ratio, you should rank ethan the higher place.
- ☐ Make sure that the first and the second column strictly have 5 and 10 characters, respectively.

- **Display player related issues.**
  - ☐ Make sure that the displayed list is sorted in an alphabetical order of the username.
- **Game related issues.**
  - ☐ Make sure to correctly update the statistics for players when a game ends.
  - ☐ Make sure to check valid move by comparing the move to the smaller one between the current number of stones and the upper bound for one move.
  - ☐ Make sure that after an invalid move, it is still the turn of the player who made the invalid move.
- **Command line prompt related issues.**
  - ☐ Make sure that the command prompt appears again after each command is issued (except the `exit` command).
  - ☐ Make sure that only one command prompt is displayed after a game is over and the system returns to the idle state.
- **Other issues.**
  - ☐ The maximum number of players can be set as 100.
  - ☐ The branching statement `switch` may not support `Strings` on the submission server.
  - ☐ The boxes enclosing the above example executions are NOT to be printed out, they are just for better illustration.

Important Notes About Submission

Immediately after you make a submission using the "submit" command, computer automatic test will be conducted on your program by automatically 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 comma. Therefore it is crucial that **your output follows exactly the same format shown in the examples above.**

The keyword `import` is available for you to use standard java packages. However, please **DO NOT** use the `package` keyword to organise your source files into packages. The automatic test system cannot deal with packages. If you are using Netbeans as the IDE, please be aware that the project name may automatically be used as the package name. Please remove the line like

```
package ProjB;
```

at the beginning of the source files before you submit them to the system.

Please use **ONLY ONE** Scanner object throughout your program. Otherwise the automatic test will 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 crucial 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.

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.

3 Your Tasks

1. Draw a UML class diagram for Nimsys based on the above information. For each class you need to identify all its instance variables and methods (including public and private modifiers) along with their corresponding data types and list of parameters. You should also identify relationships between classes. You only need to identify the relationships taught in the lecture (i.e., association), if you include other relationships not taught in the lecture, it is your responsibility to make sure your UML relationships are right to avoid any possible deductions. (**5 marks**)
2. Implement Nimsys in Java according to the above specification and in accordance with your UML class diagram. (**15 marks**)

4 Assessment

This project is worth 20% of the total marks for the subject. Project C will build on this project. In order to pass the hurdle (20/40) for all projects, it is very important that you do well in this project first.

The deadline for the project is **3pm, Monday 6 May, 2019**. **There is a 20% penalty per day for late submissions. Suppose your project gets a mark of 10 but is submitted within 1 day after the deadline, then you get 20% penalty and the mark will be 8 after penalty. There will be no mark for submissions after 3pm 10 May, 2019**.

For the UML diagrams, you may submit a preliminary version before **3pm, Monday 15 April, 2019** to get feedback before the final submission. This step is not compulsory and also not marked, but is

highly encouraged. Only the final UML submission will be marked, due at **3pm, Monday 6 May, 2019**. Multiple submissions for the UML diagrams are allowed, yet only the last submission will be assessed. Concerning the submission steps, please refer to Section 5.

Your UML diagram will be marked based on how well it reflects the program structure as described in this project specification. Your Java program will be assessed based on correctness of the output as well as quality of code implementation. See LMS for a detailed marking scheme.

5 Submission

For the UML diagrams, please submit your files in the format of **jpg** or **png** on LMS. The detailed steps are as follows

1. log on to the LMS and click Projects section;
2. below the projects table there is a link named "ProjB UML Diagram Submission", click it;
3. the submission window will pop out, in the "Attach File" section, please choose "Browse My Computer" and select your UML diagram files on computer;
4. click the "submit" button and you shall see that the submission is successful; whenever you click the "ProjB UML Diagram Submission" again, you can browse the submission history and start a new submission

For the implementation, your program should be contained within a number of well structured and documented Java classes. The entry point of your program should be in a class called Nimsys (in a file called Nimsys.java). Thus, your program will be invoked via: `java Nimsys` (you do not need to type this command).

Your Java classes should be stored together in their own directory under your home directory on the student server. Then, you can submit your work using the following command:

```
submit COMP90041 projB *.java
```

Note that you must submit all your Java files, not just `Nimsys.java`. If you submit you code multiple times, the later submission will overwrite the previous one. If you submit all your java source codes and then modify one source code, you need to submit all of your source codes agian, not just the modified one.

You should then verify your submission using the following command. This will store the verification information in the file '`feedback.txt`', which you can then view:

```
verify COMP90041 projB > feedback.txt
```

You should issue the above commands from within the same directory as where your project files are stored (to get there you may need to use the `cd` 'Change Directory' command). Note that you can submit as many times as you like before the deadline.

How you edit, compile and run your Java program is up to you. You are free to use any editor or development environment. However, **you need to ensure that your program compiles and runs correctly on the student servers**. Submit your program to the student servers **a couple of days before the deadline** to ensure that they work (you can still improve your program). **"I can't get my code to work on the student server but it worked on my Windows machine" is not an acceptable excuse for late submissions.**

6 Individual Work

Note well that this project is part of your final assessment, so cheating is not acceptable. Any form of material exchange, whether written, electronic or any other medium is considered cheating, and so is the soliciting of help from electronic newsgroups. 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. 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.

The University of Melbourne
School of Computing and Information Systems
COMP90041 Programming and Software Development
Lecturer: Prof. Rui Zhang
Semester 1, 2019

Project C
Due: 3pm, Friday 31 May, 2019

1 Introduction

The aim of this project is to add some more advanced features to the system developed in Project B. The features to be added are:

- Sort the players with more specific rules
- Handling of invalid input via Exceptions
- Write (read) the game statistics into (from) a file, i.e., one which is stored on the hard disk between executions of Nimsys
- 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
- An advanced Nim game and a victory guaranteed strategy for the AI player in this new game (**for bonus marks**)

The system should still operate as specified in Project B, but with additional functionality, due to the addition of the aforementioned features. Thus, it is advised that you use your Project B solution as a starting point for implementing Project C.

Knowledge Coverage

- Exceptions, covered in Week 9 lecture
- File I/O operations, covered in Week 10 lecture
- Polymorphism, you may use either inheritance or interface to design the AI player in addition to the human player; the corresponding concepts are covered in Week 8 lecture

2 Requirements

In following description, all command line displays are put in a box. This is only for easier understanding the format. **The box should NOT be printed out by your program, only the contents in the box should be printed.** The command prompt is illustrated below:

2.1 Sort the players with more specific rules

In Project B, when resolving the ties of winning ratio, you are required to sort the player by the username in either ascending alphabetical order or descending alphabetical order.

In Project C, **ranking should still consider winning ratio first**. When resolving the ties of winning ratio, you are required to sort the player by the **username** in **ONLY** ascending alphabetical order, for all the commands.

Example Execution:
Suppose we have three players in (username, First Name, Last Name) format as follow: (LS, Luke, Skywalker), (HS, Han, Solo), (DV, Darth, Vader). They all have the same wining ratio.

1. rank all users (default, i.e., descending order)

```
$rankings
0%  | 00 games | Darth Vader
0%  | 00 games | Han Solo
0%  | 00 games | Luke Skywalker

$
```

2. rank all users in descending order

```
$rankings desc
0%  | 00 games | Darth Vader
0%  | 00 games | Han Solo
0%  | 00 games | Luke Skywalker

$
```

3. rank all users in ascending order

```
$rankings asc
0%  | 00 games | Darth Vader
0%  | 00 games | Han Solo
0%  | 00 games | Luke Skywalker

$
```

2.2 Invalid input handling via Exceptions

The system 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. The range of potential invalid inputs you are **required** to address by Exceptions (**not** via if-then statements) are listed below, along with the required behaviour of your program. Note that some of this input checking was also a requirement in Project B. Where this is the case, you need to modify your code so that the invalid input is handled via Exceptions. The rest of the invalid input handling cases described in Project B do **not** need modification.

- 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., `addplayer`, `editplayer`, `removeplayer`, `displayplayer` `resetstats`, `rankings`, `startgame`, and `exit`.
  Example:

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

$
```

- 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.

$
```

- Invalid move (during a game) - 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.

2.3 The player statistics file

In Project B, no program data are stored to disk, so all player data are lost when exiting the program. Here, the task is to store these 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 new/modified players, 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 / family names, and number of games played / won. Note that you do not need to store information about games in progress, since a game should never be in progress when the program exits properly, i.e., via the 'exit' command.

2.4 The AI (Artificial Intelligence) player

Here, a new type of player is 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/won), and they should be stored in the system and appear in player lists/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/attributes common to both Human and AI players. You can modify your original `NimPlayer` class used in Project B 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, so 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' (refer to Project B for details). 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. Note that this file and **Testable.java** are provided to you for auto-testing the task of Section 2.6. You do NOT need to modify the advancedMove() method until you work on the task of Section 2.6.

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 Project B 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:

```
$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 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.

2.5 The 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. Additonally, the commencing condition should satisfy that the rival player first move 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 behavior such that it always wins if **either one of the following holds**:

- initial number of stones is $k(M+1)+1$, where $k \in \{0, 1, 2, ...\}$, and the rival player moves first,
- 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 his victory, the possible game states are ([a, b] here means there are at least a and up to b stones left to be removed by the players):

- Louise's turn, 1 stone left
- Winfred's turn, [2, M+1] stones left
- Louise's turn, (M+1)+1 stones left
- Winfred's turn, [(M+1)+2, 2(M+1)] stones left
- Louise's turn, 2(M+1)+1 stones left
- Winfred's turn, [2(M+1)+2, 3(M+1)] stones left
- Louise's turn, 3(M+1)+1 stones left
- 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 upto M, and hope the other player makes a mistake.

2.6 An advanced Nim game and its victory guaranteed strategy (Bonus)

(**This is a challenging task and it takes time to complete. You might find it worth more spending the time on assignments of other subjects. If you are successful in this task, you can earn back 1.5 marks that you may have lost in this project or previous two projects. The total mark for Project C is up to 11.5. However, the total mark for the three projects cannot exceed 40.**)

In this task, you are required to implement an advanced Nim game which has different rules from the original Nim game. Similar to the implementation of AI players, the polymorphism (inheritence and interface) provided by Java is expected to be demonstrated in your program. That is, both the original Nim game and this new advanced Nim game are specialized *game* with different rules. Either the inheritance or the interface can be chosen to reflect this relationship; the decision is up to you to suit your own design. Since we are going to have two types of games, the `removeStone()` method in the human player and AI player should also have two implementations, i.e., in game instances of different types of game, different `removeStone()` implementations are used, particularly for the AI player. For this advanced game setting, you can use the name `advancedMove()` for the method.

The rules of the advanced Nim game are as follows. The major rule in this advanced Nim game is that a player is **only allowed to remove 1 stone or 2 adjacent stones in each move**. Here, adjacent stones are stones who are neighbors to each other. Hence, the upper bound of stones to be removed is always 2. However, the requirement on the **adjacent stones** makes the position of the stones matters, i.e., removing the same number of stones at different positions may produce different game states, while in the original Nim game stones are conceptually the same, i.e., removing the same number of stones always produces the same game states. For example, given 5 stones represented as $< * * * * * >$,

- in the original Nim game removing 2 stones will always produce the state $< * * * >$
- in the advanced Nim game, depending on which stones are removed, removing 2 stones produces multiple states. In the following example, note that state 2 differs from state 3 because in state 2 the first two remained stones cannot be removed in a single move since they are not adjacent while in state 3 the first two remained stones can be removed in a single move.

1. removing the first and the second, results to $< \times \times * * * >$
2. removing the second the third, results to $< * \times \times * * >$
3. removing the third and the fourth, results to $< * * \times \times * >$
4. removing the fourth and the fifth, results to $< * * * \times \times >$

Similar to the original Nim game, each player takes turns to remove either one stone or two adjacent stones. **A player wins if he / she removed the last stone**.

There should be a new command defined in the `Nimsys`, named `startadvancedgame`, which will commence a new advanced game following the above rules. The syntax of this new command is `startadvancedgame initialstones,username1,username2`.

During the game, each player enters the move in the form of two integers `position number`, indicating the position and the number of stones to be removed. The stones left are displayed on one line, each stone is printed in the form of `<position, *>` if the stone is presented or `<position, x>` if the stone has already been removed.

Following is an example of the expected display of this command:

```
$startadvancedgame 10,lskywalker,artoo

Initial stone count: 10
Stones display: <1,*> <2,*> <3,*> <4,*> <5,*> <6,*> <7,*> <8,*> <9,*> <10,*>
Player 1: Luke Skywalker
Player 2: R2 D2

10 stones left: <1,*> <2,*> <3,*> <4,*> <5,*> <6,*> <7,*> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
3 2

8 stones left: <1,*> <2,*> <3,x> <4,x> <5,*> <6,*> <7,*> <8,*> <9,*> <10,*>
R2's turn - which to remove?

6 stones left: <1,*> <2,*> <3,x> <4,x> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
1 2

4 stones left: <1,x> <2,x> <3,x> <4,x> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
R2's turn - which to remove?

2 stones left: <1,x> <2,x> <3,x> <4,x> <5,*> <6,x> <7,x> <8,x> <9,x> <10,*>
Luke's turn - which to remove?
5 1

1 stones left: <1,x> <2,x> <3,x> <4,x> <5,x> <6,x> <7,x> <8,x> <9,x> <10,*>
R2's turn - which to remove?

Game Over
R2 D2 wins!

$
```

Any invalid input listed below shall be caught by **try / catch syntax**, and a line stating `Invalid move.` should be printed out:

- invalid position; given N stones when the game commences, the position should be $[1, N]$, any other values are invalid;
- invalid number of stones; **the number should be either 1 or 2**, any other values are invalid;
- the stones at the specified positions have already been removed

Here is an example output:

```
6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
12 1

Invalid move.

6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
1 3

Invalid move.

6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
2 2

Invalid move.

6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
1 2

Invalid move.

6 stones left: <1,*> <2,x> <3,x> <4,*> <5,*> <6,x> <7,x> <8,*> <9,*> <10,*>
Luke's turn - which to remove?
```

After implementing the game rules, you are also required to design the strategy for the AI player. Because the game rules change in the advanced Nim game, the victory guaranteed strategy designed for the original Nim game does not work in this new game. Your task here is to implement the AI in this new game. The hints are: if the AI player **is the first one who moves** when the game commences, there is **always a victory guaranteed strategy** for the AI player; if the AI player **is not the first one who moves** when the game commences, it is still **possible** for the AI player to win as long as the rival player does not play exactly following the winning strategy. Hence, the implementation of the strategy for the AI player in this game should enable the AI player to:

- win if it moves first when game commences,
- win if it moves second, given that the rival player does not follow exactly the wining strategy.

The details of the strategy are left for you to design. To simplify the code design, you may assume **a maximum of 11 stones** in each game played. **In order to get bonus marks, you need to implement the strategy to meet all of the below requirements**:

- Your AI player class who can play the advanced game should be named as `NimAIPlayer`, please note this name is case sensitive and mandatory;
- Your AI player class should have a constructor with **no parameters**;
- Your AI player class should implement the interface `Testable` (provided), which is listed as follows:

```
public interface Testable {
        public String advancedMove(boolean[] available, String lastMove);
}
```

Here, the `boolean[] available` represents the stones remained to be removed, `true` as remained and `false` as removed, e.g., $< * \times \times \times * >$ can be represented as [true false false false true]. The `lastMove` represents the last move made by the rival player, e.g., if the rival player removed the second stone in the last turn, then `lastMove` is of value "2 1", if the second and the third stones are removed in the last turn, then `lastMove` is of value "2 2". The `advancedMove()` method returns the move chosen by your AI player this turn, in the form of `position number`. For example, if your AI player chooses to remove the first and the second stones, the returned string should be "1 2".

The definition of this interface should be put into your source directory as a file named `Testable.java` and submitted together with your solution. **The victory guaranteed strategy designed by you should be implemented in this `advancedMove()` method**. This method will be invoked when we test the correctness of your implementation of the victory guaranteed strategy.

Overall, your AI player class will look like (provided):

```
public class NimAIPlayer implements Testable ... {
        // you may further extend a class or implement an interface
        // to accomplish the task in Section 2.6
        public NimAIPlayer() {}
        ...
        public String advancedMove(boolean[] available, String lastMove) {
                // the implementation of the victory
                // guaranteed strategy designed by you
                ...
        }
        ...
}
```

Your solution will be evaluated using test cases in three cases:

1. Your AI player moves first to play against a dummy player, who moves randomly;
2. Your AI player moves first to play against an oracle AI player, who enumerates all the possibilities and try best to win;
3. Your AI player moves second to play against a dummy player, who moves randomly;

The solution will be assessed based on the wining ratio of your solution in the three cases. Figure 1 shows the testing procedure of a submitted solution. Specifically, for three cases, case 1, case 2 and case 3, 0.5 marks will be granted ONLY if your AI winning ratio is 100% in each case, suggesting your AI player passes all the test cases. Otherwise, a non-100% ratio for any case will not get any marks for that particular case.

Checklist For Solution

- **Blank line and whitespace related issues**
  - ☐ Make sure in terms of format, your output matches with the expected output on the submission system.

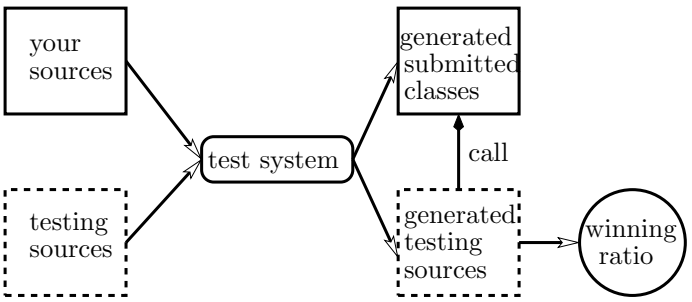

Figure 1: How testing on the advanced Nim game solution works; the testing procedure of the task in Section 2.6, the submitted codes will be complied and run together with the testing sources, while the generated testing classes will generate the winning ratio of the submitted solution.

- **File I/O mechanism related issues**
  - ☐ Make sure your program runs fine no matter whether the `players.dat` file exists.
  - ☐ Make sure every `exit` command triggers data to be written to the `players.dat` file.
- **Polymorphism related issues**
  - ☐ Make sure the AI player is implemented using inheritance mechanism.
  - ☐ Make sure you are leveraging the polymorphism to invoke the methods, i.e., using object declared in parent class to invoke methods overridden in the child class.
  - ☐ If attempting for bonus marks, make sure the advanced game is implemented using either inheritance or interface mechanism
- **Victory guaranteed strategy related issues**
  - ☐ Make sure your AI player can play without errors whether it plays as the first one or second one to move.
  - ☐ Try your best to win as many test cases as possible on the submission system with your AI player.
- **Submission issues**
  - ☐ Make sure there is only **one Scanner** object throughout your program.
- **Other issues**
  - ☐ The wining ratio of players with 0 games is 0.
  - ☐ The maximum number of players can be set as 100.
  - ☐ The maximum number of stones in each game is assumed to be 11 for the task of Section 2.6.
  - ☐ `Collections` such as `ArrayList` are acceptable.
  - ☐ Players are by default sorted according to the lexicographical order of the usernames.
  - ☐ Usernames can be assumed to be all lower-cased.
  - ☐ "*What it means when lastMove is a blank String/null in the advanced Nim game?*"
    It means that there is no last move and your `NimAIPlayer` object is to make the first move.
  - ☐ "*My code works perfectly on my machine but it does not get any winning ratio in the last test.*"
    First, check the FAQ items above and make sure you have handled each of them. Second, make sure that you do not change the Boolean array input parameter in `advancedMove()`. We just need your move returned as a String, and we will update the Boolean array.

Important Notes About Submission

Immediately after you make a submission using the "submit" command, computer automatic test will be conducted on your program by automatically 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 comma. Therefore it is crucial that **your output follows exactly the same format shown in the examples above.**

The keyword `import` is available for you to use standard java packages. However, please **DO NOT** use the `package` keyword to organise your source files into packages. The automatic test system cannot deal with packages. If you are using Netbeans as the IDE, please be aware that the project name may automatically be used as the package name. Please remove the line like

```
package ProjC;
```

at the beginning of the source files before you submit them to the system.

Please use **ONLY ONE** Scanner object throughout your program. Otherwise the automatic test will 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 crucial 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.

3 Your Task

Implement the new version of Nimsys in Java according to the above specification. Specifically, it includes following subtasks:

1. try / catch syntax to detect the listed invalid inputs in Section 2.2;
2. file based mechanism for player data as described in Section 2.3;
3. the AI player implemented using polymorphism mechanism in Section 2.4;
4. the victory guaranteed strategy in the original Nim game for the AI player in Section 2.5;
5. (**bonus**) the advanced Nim game and the corresponding victory guaranteed strategy for AI players in Section 2.6. This is a challenging task. If you cannot work out the strategy, do not get stuck on it. Please complete all other tasks first and leave it as the last one.
6. See LMS for a detailed marking scheme.

4 Assessment

This project is worth 10% of the total marks for the subject. Remember that there is a 50% hurdle requirement (i.e. 20/40) for the three projects.

Please note that if you attempt the bonus task, you can get 1.5 bonus marks. The total mark for Project C is up to 11.5. However, the total mark for the three projects cannot exceed 40. **This is a challenging task and it takes time to complete, please allocate your time wisely.**

The deadline for the project is **3pm, Friday 31 May, 2019**. **There is a 20% penalty per day for late submissions. Suppose your project gets a mark of 10 but is submitted within 1 day**

after the deadline, then you get 20% penalty and the mark will be 8 after penalty. There will be no mark for submissions after 3pm 4 June, 2019 .

Your Java program will be assessed based on correctness of the output as well as quality of code implementation. See LMS for a detailed marking scheme.

5 Submission

The entry point of your program should be in a class called Nimsys (in a file called Nimsys.java). Thus, your program will be invoked via:

```
java Nimsys
```

Your Java classes should be stored together in their own directory under your home directory on the student server. Then, you can submit your work using the following command:

```
submit COMP90041 projC *.java

For late submissions, use the following command:
submit COMP90041 projC.late *.java
```

Note that *.java includes all Java files in current directory, so you must make sure that you don't include irrelevant Java files in the current directory. **Note that you must submit all Java files you have used for your project, not just** `Nimsys.java`. If you submit you code multiple times, the later submission will overwrite the previous one. If you submit all your java source codes and then modify one source code, you need to submit all of your source codes again, not just the modified one.

You should then verify your submission using the following command. This will store the verification information in the file '`feedback.txt`', which you can then view:

```
verify COMP90041 projC > feedback.txt

For late submissions, use the following command:
verify COMP90041 projC.late > feedback.txt
```

You should issue the above commands from within the same directory as where your project files are stored (to get there you may need to use the `cd` 'Change Directory' command). Note that you can submit as many times as you like before the deadline.

How you edit, compile and run your Java program is up to you. You are free to use any editor or development environment. However, **you need to ensure that your program compiles and runs correctly on the student servers**.

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. The tests for section 2.5 are **hidden**.

Submit your program to the student servers **a couple of days before the deadline** to ensure that they work (you can still improve your program). **"I can't get my code to work on the student server but it worked on my Windows machine" is not an acceptable excuse for late submissions.**

6 Individual Work

Note well that this project is part of your final assessment, so cheating is not acceptable. Any form of material exchange, whether written, electronic or any other medium is considered cheating, and so is

the soliciting of help from electronic newsgroups. 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. 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.