The University of Melbourne
Department of Computing and Information Systems
COMP90041 Programming and Software Development
Semester 1, 2016
Project C
4pm Friday 27th May, 2016

1 Background

Note: If you are unsuccessful in Project B, you can choose to develop your Project C based on your Project A code. However, in this case, the full mark is lower and you are not allowed for the optional bonus task. See Section 5 of the document for the details.

This project is the final of three projects developing the Tic Tac Toe game system. In Project B, we extended our Tic Tac Toe code to develop a player management system, which allows multiple players to be recorded simultaneously, and for any two players to play against each other.

The aim of this project is to continue to extend this system by adding:

- 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 TicTacToe
- A new type of player - an AI (Artificial Intelligence) player, whose moves are automatically determined by the computer rather than a human game user
- (optional) A victory guaranteed strategy for the AI player

The system should still operate as specified in Project B (refer to `projB.pdf` for details), 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.

Key knowledge points covered in this project are:

- Polymorphism, using inheritance (Week 8)
- Exceptions (Week 9)
- File I/O operations (Week 10)

2 Your Task

In this project you will extend your Tic Tac Toe "game system" with improved player management functionalities, and the ability to include AI players in the system. The sections below will walk you through the steps you should follow to complete this project.

2.1 Exception Handling (3 marks)

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` are listed below, along with the required behaviour of your program. You need to implement your own `Exception` class for each of the two types of invalid input listed below. 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 Tic Tac Toe command. Here, *invalid command* suggests the input command is not among the specified commands, i.e., `addplayer`, `editplayer`, `removeplayer`, `displayplayer`, `resetstats`, `rankings`, `playgame`, and `exit`.

  Example:

  ```
  >createplayer jdawson,Dawson,Jack
  'createplayer' is not a valid command.

  >
  ```

  Note: you should type the error message into your code instead of copying from this PDF file because copying from PDF may bring non-English characters into your program which may fail the compilation at submission.

- **Invalid number of arguments**: The user enters a valid Tic Tac Toe command, but does not provide the correct number of arguments. **Note:** You only need to check for *too few* arguments, and simply ignore any extra arguments, i.e., too few arguments will generate an Exception while too many will not. Different commands may have different number of arguments; your program will only need to check the commands that required at least one argument, i.e., `addplayer`, `addaiplayer` (will be introduced below), `editplayer`, and `playgame`.

  Example:

  ```
  >addplayer jdawson
  Incorrect number of arguments supplied to command.

  >
  ```

  After implementing the invalid input checking, your code should continue as specified in Project B. You may assume that, aside from the cases explicitly mentioned above or in Project B, the input to your program will be valid.

2.2 Player Statistics Recording (3 marks)

In Project B, no program data are stored to disk; all played data are lost when exiting the program. In this task you are required to store those 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 TicTacToe`' at the command 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 in this file, i.e., usernames, given and family names, and number of games played, drawn and 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 via the '`exit`' command.

This code should all be contained in the `PlayerManager` class, which should read all the player information from **players.dat** when it is created, and write all the player data when the '`exit`' command is entered. Be sure to use appropriate object-oriented design; you will lose marks if this code is not properly abstracted.

2.3 Artificial Intelligence (AI) Player (4 marks)

The next task is to add an `AIPlayer` class to compete within Tic Tac Toe. There are several steps required to achieve this, as given below.

2.3.1 The Move Class (0.5 marks)

Add a `Move` class to represent a single move made by a player. This class just need to have the following information:

- Row number of move
- Column number of move

You should add proper accessors and mutators to the class as well.

2.3.2 The Abstract "Player" Class (0.5 marks)

Add a new abstract base `Player` class, which will be used to represent the behaviour/attributes common to both Human and AI players. This class should have the following information, plus accessor/mutator methods and other supporting methods to be used in both human and AI players:

- A username
- A family name
- A given name
- Number of games played
- Number of games drawn
- Number of games won
- An abstract method `makeMove(char[][] gameBoard)` that returns an object of the `Move` class

Here, the `makeMove` method will be used by objects of the classes that extend `Player` to make a move (i.e., decide where to place a mark on the game board). Note: in this method, a two-dimensional `char` array named `gameBoard` is used to to represent the game board. If you have used a different representation for the game board in the previous projects, such as a two-dimensional `int` array, or an array of `String`'s, you can still use it for up to Stage 3 (Section 2.3) of this project. However, if you aim to complete Stage 4 (Section 2.4, the bonus stage) as well, you need to define the `makeMove` exactly like the above. This is required so that auto-test can be performed for Stage 4.

2.3.3 The HumanPlayer Class (0.5 mark)

Modify your original `Player` class used in Project B to be the new `HumanPlayer` class, which extends the abstract `Player` class. This class should now implement the `makeMove` method, which uses a `Scanner` object to get input from a human player and returns it as the human player's move.

2.3.4 The AIPlayer Class (0.5 mark)

Add an `AIPlayer` class that extends the abstract `Player` class. This class should implement the `makeMove` method, which uses a dummy approach to make a move and returns it. This dummy approach will scan the game board from top-left to bottom-right row by row until an empty cell is found, and place a mark there.

2.3.5 Modify the Command Interface (1 marks)

Now modify the `TicTacToe` class and the `PlayerManager` class to allow for AI players to be added to the system. You should add a new command to the game system - '`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 player management and game playing commands, e.g., '`removeplayer`' and '`playgame`', should work for both human players and AI players. You should store both human players and AI players in the same player array, *not* in two ararays.
Syntax: `addaiplayer username,family_name,given_name`

```
>addaiplayer alphaTicTacToe,Alpha,TicTacToe

>
```

2.3.6 Modify the GameManager Class (1 mark)

Modify the `GameManager` class to make use of the new player representation. Instead of getting input directly from a human user, it should now make use of the `makeMove` method defined for both `HumanPlayer` and `AIPlayer` classes. All other game-play is identical to that in Project B.

After implementing all of these steps, the '`playgame`' command should allow games to be started with one or both players being AI players. The game should proceed exactly as per Project B, 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, `Jack` is a human player, and `Alpha` is an AI player:

```
>playgame jdawson,alphaTicTacToe
 | |
-----
 | |
-----
 | |
Jack's move:
1 1
 | |
-----
 |O|
-----
 | |
Alpha's move:
X| |
-----
 |O|
-----
 | |
Jack's move:
1 2
X| |
-----
 |O|O
-----
 | |
Alpha's move:
X|X|
-----
 |O|O
-----
 | |
Jack's move:
1 0
X|X|
-----
O|O|O
-----
 | |
Game Over. Jack won!

>
```

2.4 [Optional Bonus Task] A Victory Guaranteed AI Player

For a challenge, implement a game-play strategy for the AI Player that can always make the "correct" move in every round. The correct move will always result in either a tie, or victory for the AI. **Note that there are no extra marks for this task. If you are successful in this task, you can earn back 1 lost mark in other parts of this project if you lose any. However, the total mark of this project cannot exceed 10**.

Game playing algorithms for Tic Tac Toe are readily available online; we will not provide resources for this, but you are free to look them up yourself. Your implemented AI player will play against ours in ten games. In order to receive bonus marks, your AI Player must *defeat ours for at least two games and cannot lose in any game.*

To support auto-testing of this stage, you need to implement your game-play strategy in a class named `AdvancedAIPlayer` by adding code to the following skeleton:

```
public class AdvancedAIPlayer extends Player {

    public AdvancedAIPlayer() {
        /* do not change this method, make sure to add
           public Player() {} to Player class as well
        */
    }

    public Move makeMove(char[][] gameBoard) {
        /* add code here to implement your game-play strategy */
    }

    /* add supporting methods/variables here */
}
```

The auto-test will call `AdvancedAIPlayer()` to construct an object of `AdvancedAIPlayer`, and test your game-play strategy by calling the `makeMove()` method of the object. When your `makeMove()` method is called, a `3 x 3` two-dimensional `char` array `gameBoard` representing the game board will be passed into the method, where `gameBoard[0][0]` represents the top left cell and `gameBoard[2][2]` represents the bottom right cell. Each cell has a value of `'O'`, `'X'`, or `' '`, representing a mark placed by player `O`, player `X`, and an empty cell, respectively. Based on `gameBoard`, you need to figure out whether your `AdvancedAIPlayer` object is player `O` or player `X`, and make a proper move by returning an object of the class `Move`. For our auto-test to understand the move you made, your `Move` class needs to be in the following form:

```
public class Move {

    public int getRow() {
        /* add code here to return the row of a move */
    }

    public int getColumn() {
        /* add code here to return the column of a move */
    }

    /* add supporting methods/variables here */
}
```

2.5 Checklist for the Implementation

- You will be given a sample test input file `test0.txt` and the corresponding sample output file `test0-output.txt`. When you run your program by the following command in a terminal (or Windows command line):

  ```
  java TicTacToe < test0.txt > my-output.txt
  ```

your program should produce a file named `my-output.txt` which should be exactly the same as `test0-output.txt`. In this command, "`< test0.txt`" and "`> my-output.txt`" are called "input redirection" and "output redirection." They use the content in `test0.txt` as the command line input, and print the program output into `my-output.txt`.

- Tests will be conducted on your program by automatically compiling, running, and comparing your outputs for several test cases with generated expected outputs. **These test cases used will be different from the sample test file given**. 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**. Therefore, it is crucial that **you generate your own test files and test your program extensively**.
- The syntax `import` is available for you to use standard Java packages. However, please **DO NOT** use any `package` statement at the top of your code. The automatic test system cannot deal with the `package` statements. 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 any lines of the form

  ```
  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 any remaining Scanner objects with nothing to read, causing a 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.

  You may declare an object `keyboard` of the `Scanner` class as a `public static` variable of the `TicTacToe` class. Then you can use this object throughout your program by "`TicTacToe.keyboard`."

3 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 in total.

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 Submission

Your Java 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 named `TicTacToe` (in a file named `TicTacToe.java`). You should upload all the relevant Java code files to the Engineering School student server. Then, you can submit your work using the following command at once:

```
submit 90041 C *.java
```

Note that you must submit all your Java files, not just `TicTacToe.java`. If you have changed any of your Java source files after a submission, you need to submit **all** of your source files again, not just the modified one.

You should then verify your submission using the following command:

```
verify 90041 C > feedback.txt
```

This will store the verification information in the file `feedback.txt`, which you can view using the following command:

```
more feedback.txt
```

You should issue the above commands from within the same directory as where the 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 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 server 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.**

The deadline for the project is **4pm Friday 27th May, 2016**. The allowed time is more than enough for completing the project. Penalties will apply on late submissions at 2 marks per day. Late submissions after **4pm Sunday 29th May, 2016** will NOT be accepted.

5 For Submission Built Upon Project A

5.1 Exception Handling (3 marks)

If you build upon your code in Project A, your first task is to create three `Exception` classes to handle the following three errors:

- ☐ Placing a mark at an already occupied cell.
  Error message: `Invalid move. The cell has been occupied.`
  Follow-up operation: Prompt for the same player to place a mark again.
- ☐ Placing a mark at a cell outside the valid game board range.
  Error message: `Invalid move. You must place at a cell within {0,1,2} {0,1,2}.`
  Follow-up operation: Prompt for the same player to place a mark again.
- ☐ Player X having the same name as Player O.
  Error message: `The player name has been used already.`
  Follow-up operation: Prompt for Player X to input name again.

5.2 Player Name Recording (2.5 marks)

Your second task is to change your program so that it stores the player names and the number of games they have played in a file named **players.dat**. Every time your program starts (by running '`java TicTacToe`' at the command prompt), it should first check the existence of the file. If the file already exists, then it should read in the player names from the file, and use them to start a game. If the file does not exist, then it should prompt for user input of the two player names, and initialise the number of games played as 0. At the end of the game, it should increase the number of games played by 1, write the player names as well as the updated number of games played into the file players.dat.

5.3 Artificial Intelligence (AI) Player (2.5 marks)

1. If you have not done so, change your code to use an array to represent the game board.
2. Add a `Move` class to represent a single move made by a player. This class just needs to have the following information:
   - Row number of move
   - Column number of move

   You should add proper accessors and mutators to the class as well.
3. Now modify your `TicTacToe` class and add a method:

   `private Move makeMoveHuman()`

   This method will use the `Scanner` object and prompt a user to input two integers to form a move.

4. Add another method to your `TicTacToe` class:

   `private Move makeMoveAI()`

   This method will scan the game board from top-left to bottom-right row by row until an empty cell is found, and return the coordinates of that cell as a move.

5. Modify the `run()` method, such that when human players are playing the game, the `makeMoveHuman()` method is called to get a player's move.

6. Further modify the `run()` method, such that when a player named "`alphaTicTacToe`" is added to the system, and it is its turn to make a move, the `makeMoveAI()` method is called to create a move for it.

Submission: Your Java 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 named `TicTacToe` (in a file named `TicTacToe.java`). You should upload all the relevant Java code files to the Engineering School student server. Then, you can submit your work using the following command at once:

```
submit 90041 CA *.java
```

Note that you must submit all your Java files, not just `TicTacToe.java`. If you have changed any of your Java source files after a submission, you need to submit **all** of your source files again, not just the modified one.

You should then verify your submission using the following command:

```
verify 90041 CA > feedback.txt
```

This will store the verification information in the file `feedback.txt`, which you can view using the following command:

```
more feedback.txt
```

You should issue the above commands from within the same directory as where the 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 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 server 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.**

The deadline for the project is **4pm Friday 27th May, 2016**. The allowed time is more than enough for completing the project. Penalties will apply on late submissions at 2 marks per day. Late submissions after **4pm Sunday 29th May, 2016** will NOT be accepted.

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.