ITI 1121. Introduction to Computing II
Winter 2020

Assignment 3
(Last modified on March 8, 2020)

Deadline: March 22, 2020, 11:30 pm

Learning objectives

- Iterating through different states
- Using indirection

Introduction

In the previous assignment, we had an optional question, Q3, to remove symmetries in our list of games, when dealing with 3x3 games. In this assignment, we will solve that problem in the general case, and use iteration to do so.

Symmetries and iterators

When we created our list of games in Q2 of assignment 2, we added a lot of solutions that were essentially identical, simply a symmetry of other games already listed. For example, as we discussed in Q3 of assignment 2, there are only 3 different ways to play the first move on a 3x3 Tic-Tac-Toe game. The other 6 opening moves are equivalents by symmetry.

Let's looks at symmetries in a *n*x*m* grid. Let's first assume that $n \neq m$, that is, the grid is not square. In the case of a non-square grid, we have essentially two symmetries: vertical flip, and horizontal flip (Figure 1).

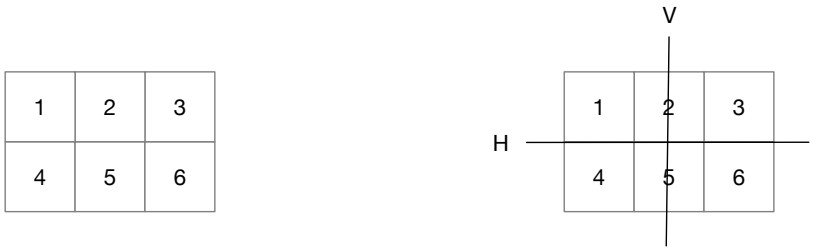

Figure 1: Non-square boards have 2 axes of symmetry.

For each such non-square *n*x*m* grid, there are up to 3 different but symmetrical grids: the one obtain with a vertical symmetry, the one obtained with a horizontal symmetry, and the one obtain with a combination of both symmetries (Figure 2).

Something that will come handy is that it is possible to iterate through all of these symmetries by repeatedly applying transformations, for example the series horizontal, then vertical, then horizontal symmetry will give you all four boards, as shown Figure 3.

Things are a little bit more complicated when the grid is square. In addition to horizontal and vertical symmetries, we have both diagonals, and well as rotation (Figure 4). Each grid now gives us up to 7 other different but symmetrical grids, shown Figure 5.

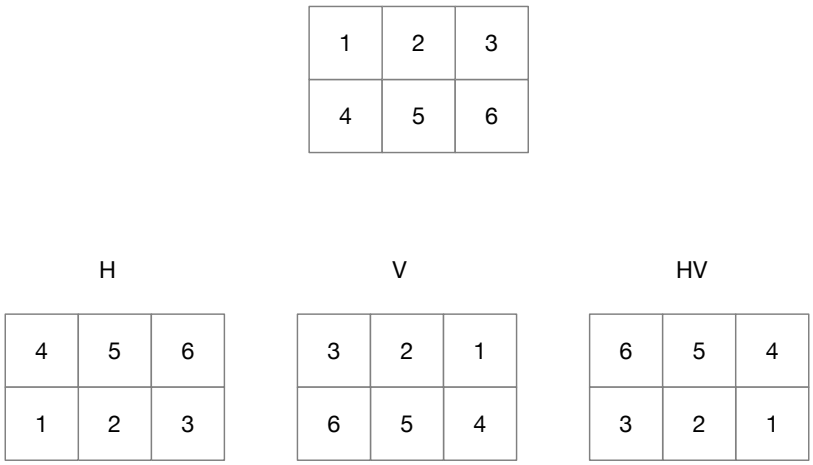

Figure 2: Non-square boards have up to 3 symmetrical equivalent grids.

Here again, it is possible to iterate through all 8 different but symmetrical equivalent (square) grids, for example with the sequence rotation-rotation-rotation-horizontal symmetry-rotation-rotation-rotation, as illustrated Figure 6.

1 Step 1: Modifying Utils.java

From the discussion above, we can deduce that implementing the horizontal symmetry, the vertical symmetry and the 90 degree (clockwise) rotation is enough to get every possibly symmetrical grid, for both square and non square grids. So we are going to add three methods in **Utils.java** to do this. In keeping with our previous approach, the grids are going to be stored using a one dimensional array. For reasons that will become clear very soon, we will use an array of integers for our grid. Each of the three methods will have three parameters: the number of lines and the number of columns of the grid, and a reference to the array of integers representing the grid. You need to implement the three class methods in the class Utils.java, that is:

- **public static void horizontalFlip(int lines, int columns, int[] transformedBoard)**: performs a horizontal symmetry on the elements in the columnsxlines grid stored in the array reference by *transformedBoard*. The elements in the array referenced by *transformedBoard* are modified accordingly (see example below).
- **public static void verticalFlip(int lines, int columns, int[] transformedBoard)**: performs a vertical symmetry on the elements in the columnsxlines grid stored in the array reference by *transformedBoard*. The elements in the array referenced by *transformedBoard* are modified accordingly (see example below).
- **public static void rotate(int lines, int columns, int[] transformedBoard)**: rotates clockwise by 90 degrees the elements in the columns×lines grid stored in the array reference by *transformedBoard*. The elements in the array referenced by *transformedBoard* are modified accordingly (see example below).

All three methods must check the provided inputs and handle all possible cases as required.
The class **Utils.java** comes with the following tests:

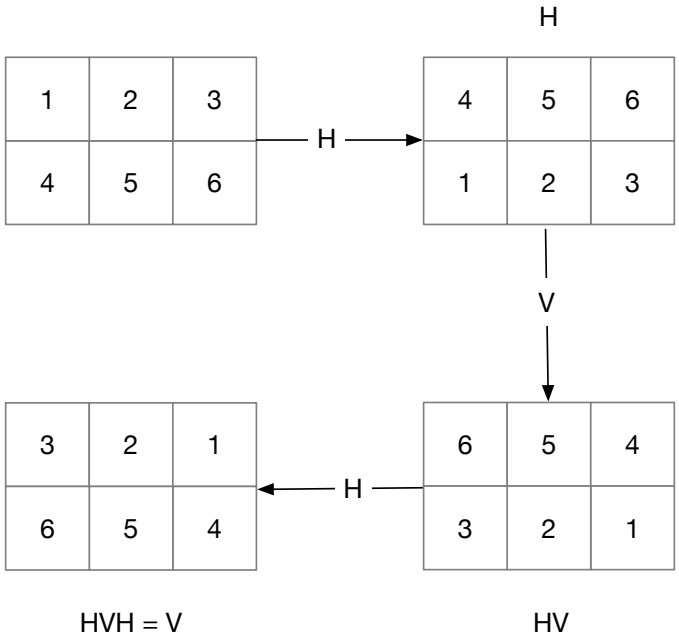

Figure 3: Enumerating all non-square symmetrical boards.

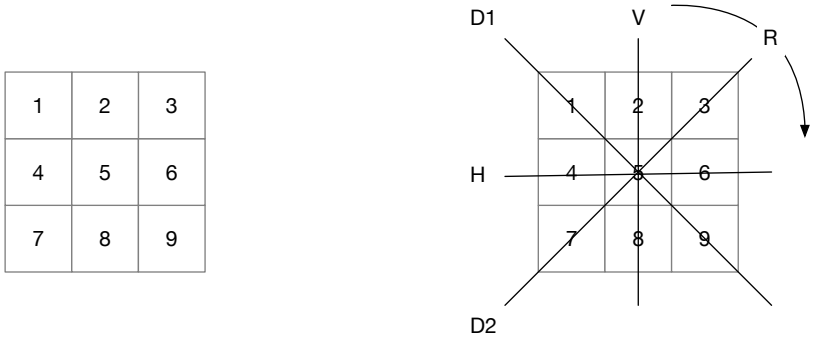

Figure 4: Square board have 4 axes of symmetry, and can be rotated 90 degrees as well

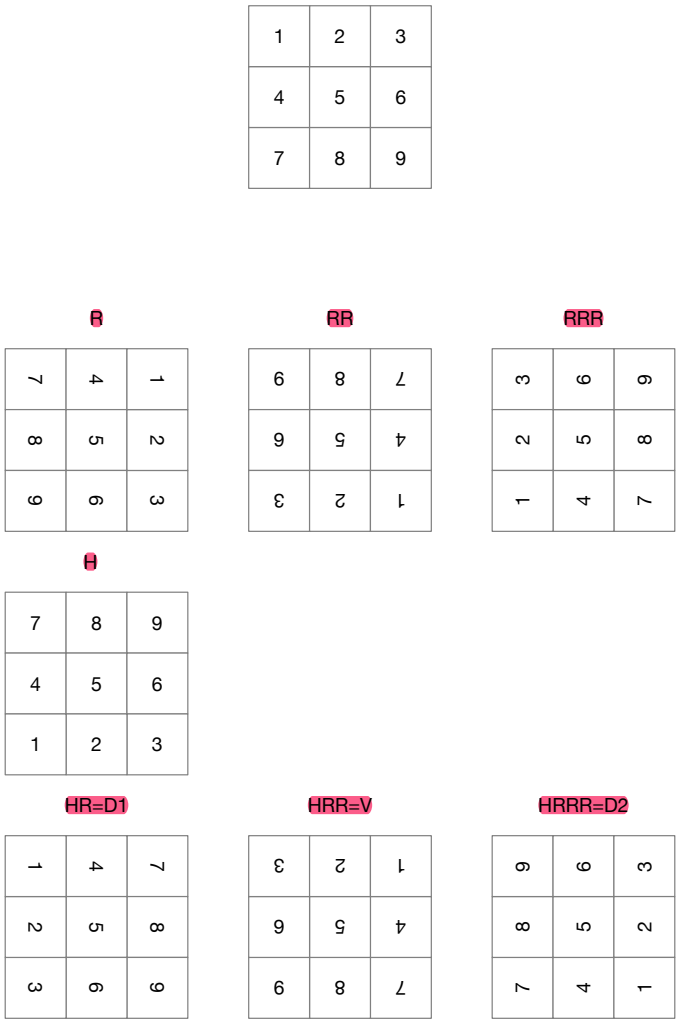

Figure 5: Square boards have 7 symmetrical equivalent boards.

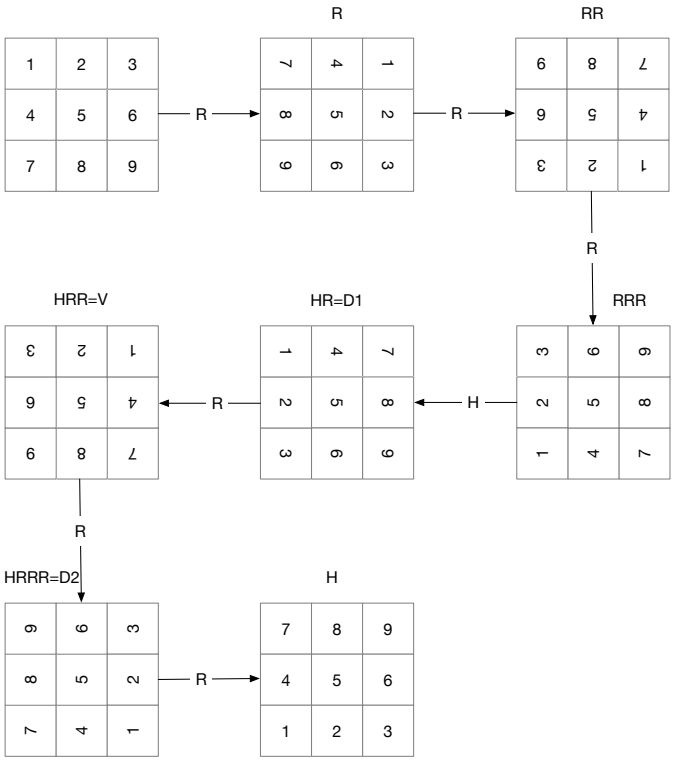

Figure 6: Enumerating all symmetrical square boards.

```
1  public class Utils {
2  // code omitted
3      private static void test(int lines, int columns){
4          int[] test;
5          test = new int[lines*columns];
6          for(int i = 0 ; i < test.length; i++){
7                  test[i] = i;
8          }
9          System.out.println("testing " + lines + " lines and " + columns + " columns.");
10         System.out.println(java.util.Arrays.toString(test));
11         horizontalFlip(lines,columns,test);
12         System.out.println("HF => " + java.util.Arrays.toString(test));
13         horizontalFlip(lines,columns,test);
14         System.out.println("HF => " + java.util.Arrays.toString(test));
15         verticalFlip(lines,columns,test);
16         System.out.println("VF => " + java.util.Arrays.toString(test));
17         verticalFlip(lines,columns,test);
18         System.out.println("VF => " + java.util.Arrays.toString(test));
19         if(lines == columns){
20                 for(int i = 0; i < 4; i++) {
21                         rotate(lines,columns,test);
22                         System.out.println("ROT => " + java.util.Arrays.toString(test));
23                 }
24         }
25     }
26
27     public static void main(String[] args){
28         int[] test;
29         int lines, columns;
30
31         test(2,2);
32         test(2,3);
33         test(3,3);
34         test(4,3);
35         test(4,4);
36     }
37 }
```

Running the test above should produce the following output:

```
% javac Utils.java
% java Utils
testing 2 lines and 2 columns.
[0, 1, 2, 3]
HF => [2, 3, 0, 1]
HF => [0, 1, 2, 3]
VF => [1, 0, 3, 2]
VF => [0, 1, 2, 3]
ROT => [2, 0, 3, 1]
ROT => [3, 2, 1, 0]
ROT => [1, 3, 0, 2]
ROT => [0, 1, 2, 3]
testing 2 lines and 3 columns.
[0, 1, 2, 3, 4, 5]
HF => [3, 4, 5, 0, 1, 2]
HF => [0, 1, 2, 3, 4, 5]
VF => [2, 1, 0, 5, 4, 3]
VF => [0, 1, 2, 3, 4, 5]
testing 3 lines and 3 columns.
[0, 1, 2, 3, 4, 5, 6, 7, 8]
HF => [6, 7, 8, 3, 4, 5, 0, 1, 2]
HF => [0, 1, 2, 3, 4, 5, 6, 7, 8]
VF => [2, 1, 0, 5, 4, 3, 8, 7, 6]
VF => [0, 1, 2, 3, 4, 5, 6, 7, 8]
ROT => [6, 3, 0, 7, 4, 1, 8, 5, 2]
```

```
ROT => [8, 7, 6, 5, 4, 3, 2, 1, 0]
ROT => [2, 5, 8, 1, 4, 7, 0, 3, 6]
ROT => [0, 1, 2, 3, 4, 5, 6, 7, 8]
testing 4 lines and 3 columns.
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
HF => [9, 10, 11, 6, 7, 8, 3, 4, 5, 0, 1, 2]
HF => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
VF => [2, 1, 0, 5, 4, 3, 8, 7, 6, 11, 10, 9]
VF => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
testing 4 lines and 4 columns.
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
HF => [12, 13, 14, 15, 8, 9, 10, 11, 4, 5, 6, 7, 0, 1, 2, 3]
HF => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
VF => [3, 2, 1, 0, 7, 6, 5, 4, 11, 10, 9, 8, 15, 14, 13, 12]
VF => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
ROT => [12, 8, 4, 0, 13, 9, 5, 1, 14, 10, 6, 2, 15, 11, 7, 3]
ROT => [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
ROT => [3, 7, 11, 15, 2, 6, 10, 14, 1, 5, 9, 13, 0, 4, 8, 12]
ROT => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
%
```

2 Step 2: Generating every non-symmetrical *n*x*m* games

In assignment 2, we have already created a method that generates all the possible games for a given grid size. That method would only add a game to the list if the game was not **equal** to a game that was already there. Therefore, from that viewpoint, all we need to do is slightly update that method to only add a game if it is not *equal or symmetrical* to a game that is already there. The new instance method **equalsWithSymmetry** of **TicTacToeGame** provides that information. We just need to implement it, and the method **generateAllUniqueGames** which is provided in the class **ListOfGamesGenerator** will generate the list of lists we are looking for.

Indirection

In order to implement **equalsWithSymmetry** in **TicTacToeGame**, we will have to loop through all the possible symmetrical games to see if we have a match. Of course, we could simply apply the symmetries on the board itself. We would thus apply transformations to the board until it either matches the board of the game we are comparing it with (in which case it is symmetrical) or we run out of symmetrical games (in which case it is not symmetrical).

However, changing the board itself might have unwanted side effects. For example, imagine that we are printing the game to the user. What would happen is that since the board is flipped through symmetrical equivalent games, the game presented to the user might be a different but symmetrical game every time, which would clearly not be good.

Therefore, we are going to introduce a level of **indirection** to compute our symmetries. The board itself will remain unchanged, but we will use another array that will map the indices of the board to their current symmetrical locations. We will use an instance variable, the array of integer **transformedBoard** to store the indirection. Initially, since there is no transformation, we always have **board[i]==board[transformedBoard[i]]**. But after some symmetries have been applied to the game, **transformedBoard[i]** records where the index i of the board is mapped to in the symmetry.

For example, imagine a $2x2$ board. Before any symmetry, we have **transformedBoard[i]==i** for $0 \leq i < 4$, and thus **board[i]==board[transformedBoard[i]]**. If we apply a vertical symmetry for example, the index 0 is mapped to 1, 1 is mapped to 0, 2 is mapped to 3 and 3 is mapped to 2. **board** remains unchanged, and **transformedBoard** $= \{1, 0, 3, 2\}$. So if one want to know what is at index 1 of the board after symmetry, they have to access $board[transformedBoard[1]]$, which is thus $board[0]$ before symmetry.

Iterating though symmetrical board

The second thing to do is to decide how we are going to systematically go through all the symmetries of a given board. Each board has either four or eight symmetrical positions depending if it is square or not[1]. We propose a convenient mechanism to iterate through all possible positions, using new instance methods in **TicTacToeGame**:**hasNext**, **next**, and **reset**.

- **hasNext()** returns **true** if and only if a call to the method **next** would succeed, and **false** otherwise.
- Each call to the method **next** finds the next symmetrical equivalent board in transformedBoard. It throws an exception **IllegalStateException** if it is called **next** more times than there are symmetries between two calls to the method **reset**.
- The method **reset** puts transformedBoard back in its original value, correponding the the unchanged board.

The following Java program illustrates the intended use for **hasNext**, **next**, and **reset** (the method **toStringTransformed** of **TicTacToeGame** returns a String representation of the tranformed game).

```
1  public class Test {
2
3          public static void printTest(TicTacToeGame g){
4                  g.reset();
5                  while (g.hasNext()) {
6                      g.next();
7                      System.out.println(g.toStringTransformed());
8                  }
9                   System.out.println("reset:");
10                 g.reset();
11                 while (g.hasNext()) {
12                     g.next();
13                     System.out.println(g.toStringTransformed());
14                 }
15         }
16         public static void main(String[] args) {
17
18                 TicTacToeGame g;
19
20                 System.out.println("Test on a 3x3 game");
21                 g = new TicTacToeGame();
22                 g.play(0);
23                 g.play(2);
24                 g.play(3);
25                 printTest(g);
26                 System.out.println("Test on a 5x4 game");
27                 g = new TicTacToeGame(4,5);
28                 g.play(0);
29                 g.play(2);
30                 g.play(3);
31                 printTest(g);
32          }
33 }
```

The execution produces the following output:

```
% java Test
Test on a 3x3 game
 X |   | O
-----------
 X |   |
-----------
   |   |

   | X | X
-----------
```

[1] Strictly speaking, sometimes there are fewer different choices as some of the symmetries are actually identical, but we won't worry about this here.

```
   |   |
-----------
   |   | 0

   |   |
-----------
   |   | X
-----------
 0 |   | X

 0 |   |
-----------
   |   |
-----------
 X | X |

 X | X |
-----------
   |   |
-----------
 0 |   |

 0 |   | X
-----------
   |   | X
-----------
   |   |

   |   | 0
-----------
   |   |
-----------
   | X | X

   |   |
-----------
 X |   |
-----------
 X |   | 0

reset:
 X |   | 0
-----------
 X |   |
-----------
   |   |

   | X | X
-----------
   |   |
-----------
   |   | 0

   |   |
-----------
   |   | X
-----------
```

```
 O |   | X

 O |   |
-----------
   |   |
-----------
 X | X |

 X | X |
-----------
   |   |
-----------
 O |   |

 O |   | X
-----------
   |   | X
-----------
   |   |

   |   | O
-----------
   |   |
-----------
   | X | X

   |   |
-----------
 X |   |
-----------
 X |   | O
```

Test on a 5x4 game

```
 X |   | O | X |
-------------------
   |   |   |   |
-------------------
   |   |   |   |
-------------------
   |   |   |   |

   |   |   |   |
-------------------
   |   |   |   |
-------------------
   |   |   |   |
-------------------
 X |   | O | X |

   |   |   |   |
-------------------
   |   |   |   |
-------------------
   |   |   |   |
-------------------
   | X | O |   | X
```

```
   | X | O |   | X
------------------
   |   |   |   |
------------------
   |   |   |   |
------------------
   |   |   |   |

reset:
 X |   | O | X |
------------------
   |   |   |   |
------------------
   |   |   |   |
------------------
   |   |   |   |

   |   |   |   |
------------------
   |   |   |   |
------------------
   |   |   |   |
------------------
 X |   | O | X |

   |   |   |   |
------------------
   |   |   |   |
------------------
   |   |   |   |
------------------
   | X | O |   | X

   | X | O |   | X
------------------
   |   |   |   |
------------------
   |   |   |   |
------------------
   |   |   |   |
%
```

As pointed out ealier, we only need three transformations to iterate through every symmetrical games: vertical symmetry (**VSYM**), horizontal symmetry (**HSYM**) and rotation (**ROT**). We also need the ability to reset the game to its original state, the identity transformation (**ID**).

Following a call to the method **reset**, each call to the method **next** changes the orientation of the game according to the following list of operations:

- for a non-square board: **ID**, **HSYM**,**VSYM**, **HSYM**
- for a square board: **ID**, **ROT**, **ROT**, **ROT**, **HSYM** ,**ROT**, **ROT**, **ROT**

You must add all the necessary instance variables to implement the methods: **hasNext**, **next** and **reset**.

Finally, you need to implement the instance method **boolean equalsWithSymmetry(TicTacToeGame other)**, which returns **true** if and only if **this** instance of **TicTacToeGame** and **other** are identical up to symmetry.

The class **ListOfGamesGenerator** has a method **generateAllUniqueGames** (already implemented) which relies on **equalsWithSymmetry** to generate the list of games. Here are a few sample runs:

```
%java TicTacToe
```

```
************************************************************
*                                                          *
*                                                          *
*                                                          *
*                                                          *
************************************************************

======= level 0 =======: 1 element(s) (1 still playing)
======= level 1 =======: 3 element(s) (3 still playing)
======= level 2 =======: 12 element(s) (12 still playing)
======= level 3 =======: 38 element(s) (38 still playing)
======= level 4 =======: 108 element(s) (108 still playing)
======= level 5 =======: 174 element(s) (153 still playing)
======= level 6 =======: 204 element(s) (183 still playing)
======= level 7 =======: 153 element(s) (95 still playing)
======= level 8 =======: 57 element(s) (34 still playing)
======= level 9 =======: 15 element(s) (0 still playing)
that's 765 games
91  won by X
44  won by O
3 draw
It took 116ms.
% java TicTacToe 2 2 2
************************************************************
*                                                          *
*                                                          *
*                                                          *
*                                                          *
************************************************************

======= level 0 =======: 1 element(s) (1 still playing)
======= level 1 =======: 1 element(s) (1 still playing)
======= level 2 =======: 2 element(s) (2 still playing)
======= level 3 =======: 2 element(s) (0 still playing)
that's 6 games
2  won by X
0  won by O
0 draw
It took 38ms.
% java TicTacToe 2 2 3
************************************************************
*                                                          *
*                                                          *
*                                                          *
*                                                          *
************************************************************

======= level 0 =======: 1 element(s) (1 still playing)
======= level 1 =======: 1 element(s) (1 still playing)
======= level 2 =======: 2 element(s) (2 still playing)
======= level 3 =======: 2 element(s) (2 still playing)
======= level 4 =======: 2 element(s) (0 still playing)
that's 8 games
0  won by X
0  won by O
2 draw
It took 40ms.
```

```
% java TicTacToe 4 3 2
************************************************************
*                                                          *
*                                                          *
*                                                          *
*                                                          *
************************************************************

======= level 0 =======: 1 element(s) (1 still playing)
======= level 1 =======: 4 element(s) (4 still playing)
======= level 2 =======: 36 element(s) (36 still playing)
======= level 3 =======: 172 element(s) (98 still playing)
======= level 4 =======: 439 element(s) (245 still playing)
======= level 5 =======: 1006 element(s) (167 still playing)
======= level 6 =======: 639 element(s) (91 still playing)
======= level 7 =======: 379 element(s) (13 still playing)
======= level 8 =======: 50 element(s) (2 still playing)
======= level 9 =======: 6 element(s) (0 still playing)
that's 2732 games
1285  won by X
790  won by O
0 draw
It took 235ms.
% java TicTacToe 4 4 2
************************************************************
*                                                          *
*                                                          *
*                                                          *
*                                                          *
************************************************************

======= level 0 =======: 1 element(s) (1 still playing)
======= level 1 =======: 3 element(s) (3 still playing)
======= level 2 =======: 33 element(s) (33 still playing)
======= level 3 =======: 219 element(s) (141 still playing)
======= level 4 =======: 913 element(s) (587 still playing)
======= level 5 =======: 3338 element(s) (883 still playing)
======= level 6 =======: 4702 element(s) (1217 still playing)
======= level 7 =======: 7048 element(s) (511 still playing)
======= level 8 =======: 2724 element(s) (194 still playing)
======= level 9 =======: 1119 element(s) (0 still playing)
that's 20100 games
10189  won by X
6341  won by O
0 draw
It took 23865ms.
%
```

Academic Integrity

This part of the assignment is meant to raise awareness concerning plagiarism and academic integrity. Please read the following documents.

- https://www.uottawa.ca/administration-and-governance/academic-regulation-14-other-important-informat
- https://www.uottawa.ca/vice-president-academic/academic-integrity

Cases of plagiarism will be dealt with according to the university regulations. By submitting this assignment, you acknowledge:

1. I have read the academic regulations regarding academic fraud.
2. I understand the consequences of plagiarism.
3. With the exception of the source code provided by the instructors for this course, all the source code is mine.
4. I did not collaborate with any other person, with the exception of my partner in the case of team work.
    - If you did collaborate with others or obtained source code from the Web, then please list the names of your collaborators or the source of the information, as well as the nature of the collaboration. Put this information in the submitted README.txt file. Marks will be deducted proportional to the level of help provided (from 0 to 100%).

Rules and regulation

- Follow all the directives available on the assignment directives web page.
- Submit your assignment through the on-line submission system virtual campus.
- You must preferably do the assignment in teams of two, but you can also do the assignment individually.
- You must use the provided template classes below.
- If you do not follow the instructions, your program will make the automated tests fail and consequently your assignment will not be graded.
- We will be using an automated tool to compare all the assignments against each other (this includes both, the French and English sections). Submissions that are flagged by this tool will receive the grade of 0.
- It is your responsibility to make sure that BrightSpace has received your assignment. Late submissions will not be graded.

Files

You must hand in a **zip** file (no other file format will be accepted). The name of the top directory has to have the following form: **a3_3000000_3000001**, where 3000000 and 3000001 are the student numbers of the team members submitting the assignment (simply repeat the same number if your team has one member). The name of the folder starts with the letter "a" (lowercase), followed by the number of the assignment, here 3. The parts are separated by the underscore (not the hyphen). There are no spaces in the name of the directory. The archive a3_3000000_3000001.zip contains the files that you can use as a starting point. Your submission must contain the following files.

- README.txt
    - A text file that contains the names of the two partners for the assignments, their student ids, section, and a short description of the assignment (one or two lines).
- CellValue.java
- GameState.java
- ListOfGamesGenerator.java
- StudentInfo.java
- Test.java
- TicTacToe.java
- TicTacToeGame.java
- Transformation.java
- Utils.java

Questions

For all your questions, please visit the Piazza Web site for this course:

- https://piazza.com/uottawa.ca/winter2020/iti1121/home

Last modified: March 8, 2020