ITI 1121. Introduction to Computing II
Winter 2020

Assignment 1
(Last modified on January 26, 2020)

Deadline: February 2, 2020, 11:30 pm

[PDF]

Learning objectives

- Edit, compile and run Java programs
- Utilize arrays to store information
- Apply basic object-oriented programming concepts
- Understand the university policies for academic integrity

Introduction

This year, we are going to implement the game Tic-Tac-Toe. The game itself is fairly simple and well-known game. You can brush up your Tic-Tac-Toe skills e.g. here: https://en.wikipedia.org/wiki/Tic-tac-toe.

Our ultimate goal is to program a machine-learning algorithm that will learn how to be a good Tic-Tac-Toe player automatically. We will base our approach on a paper published by Donald Michie in 1961 in *Science Survey*, titled *Trial and error*. That paper has been reprinted in the book *On Machine Intelligence* and can be found on page 11 at the following URL: https://www.gwern.net/docs/ai/1986-michie-onmachineintelligence.pdf.

For a more modern take of the same idea, you can also watch https://www.youtube.com/watch?v=R9c-_neaxeU.

But for this assignment, our goal is more modest: we simply want to implement a game of Tic-Tac-Toe, where players are indicating their next move from the command line. In its default configuration, it will look like this: the program first displayed an empty grid and is prompting the first player (*X*) for an input.

```
$ java TicTacToe
   |   |
-----------
   |   |
-----------
   |   |

X to play:
```

The first player played the cell *5*. The program displays the current game, with cell number five taken by *X*, and is prompting the second player (*O*) for an input. The game will keep going following that pattern.

```
X to play:  5
   |   |
-----------
   | X |
-----------
   |   |

O to play:  2
```

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

X to play: 1
 X | 0 |
-----------
   | X |
-----------
   |   |

0 to play: 9
 X | 0 |
-----------
   | X |
-----------
   |   | 0

X to play: 4
 X | 0 |
-----------
 X | X |
-----------
   |   | 0

0 to play: 6
 X | 0 |
-----------
 X | X | 0
-----------
   |   | 0

X to play: 7
 X | 0 |
-----------
 X | X | 0
-----------
 X |   | 0

Result: XWIN
 $
```

As can be seen, at each turn the program prints out the current state of the game and then queries the next user (*X* or *O*) to provide its next move. We simply assume that the cells are numbered line by line, from top left to bottom right, as follows:

```
 1 | 2 | 3
-----------
 4 | 5 | 6
-----------
 7 | 8 | 9
```

So in the game above, the first player's (playing with *X*) initial move is to select the cell 5, which is in the middle of the game. The second player's (playing with *O*) initial move is to select the cell 2, which is in the middle of the first line (and a fatal mistake). After a few more moves, the first player wins.

Our own implementation will be a little bit more general than the usual 3x3 grid game. By default (as shown above), our game will be indeed played on a 3x3 grid, trying to align 3 similar cells horizontally, vertically or diagonally. But our more general implementation will accept 3 parameters n, m and k to play a game on an nxm grid trying to align k similar cells horizontally, vertically or diagonally. Here is an example of a game on a 3x4 grid, trying to align 3 similar cells.

```
$ java TicTacToe 3 4 3
   |   |   |
--------------
   |   |   |
--------------
   |   |   |

X to play: 2
   | X |   |
--------------
   |   |   |
--------------
   |   |   |

O to play: 6
   | X |   |
--------------
   | O |   |
--------------
   |   |   |

X to play: 7
   | X |   |
--------------
   | O | X |
--------------
   |   |   |

O to play: 4
   | X |   | O
--------------
   | O | X |
--------------
   |   |   |

X to play: 12
   | X |   | O
--------------
   | O | X |
--------------
   |   |   | X

Result: XWIN
```

.

Enum

In this application, we have a need to record the "state" of a game: it could be still in play, or one or the other of the players have won, or it could be a draw. Similarly, we need to record the state of a cell on the board: a cell can be empty, or it can contain a *X* or a *O*.

There are several ways to achieve this, but in this assignment we are going to use Java's *Enum* type.

Some programmers use values of type **int** to represent symbolic constants in their programs. For example, to represent the day of the week or the month of the year.

```
1  public class E1 {
2
3      public static final int MONDAY = 1;
4      public static final int TUESDAY = 2;
5      public static final int SUNDAY = 7;
6
7      public static final int JANUARY = 1;
8      public static final int FEBRUARY = 2;
9      public static final int DECEMBER = 12;
10
11     public static void main(String[] args) {
12
13         int day = SUNDAY;
14
15         switch (day) {
16             case MONDAY:
17             System.out.println("sleep");
18             break;
19
20             case SUNDAY:
21             System.out.println("midterm test");
22             break;
23
24             default:
25             System.out.println("study");
26         }
27     }
28 }
```

Using constants, such as MONDAY and JANUARY, improves the readability of the source code. Compare "if (day == MONDAY) { ... }" to "if (day == 1) { ... }". It is one step in the right direction.

However, since all the constants are integer values, there are several kinds of errors that the compiler cannot detect. For example, if the programmer uses the same number for two constants, the compiler would not be able to help, 7 is valid value for both SATURDAY and SUNDAY:

```
1  public static final int SATURDAY = 7;
2  public static final int SUNDAY = 7;
```

But also, assigning a value representing a month to variable representing a day of the week would not be detected by the compiler, both are of type int:

```
1  int day = JANUARY;
```

Enumerated types have the same benefits as the symbolic constants above, making the code more readable, but in a typesafe way.

```
1  public class E2 {
2
3      public enum Day {
4          MONDAY, TUESDAY, SUNDAY
5      }
6
7      public enum Month {
8          JANUARY, FEBRUARY, DECEMBER
9      }
10
11     public static void main( String[] args ) {
12
13         Day day = Day.MONDAY;
14
15         switch (day) {
16             case MONDAY:
17             System.out.println( "sleep" );
18             break;
19
20             case SUNDAY:
```

```
21              System.out.println( "midterm test" );
22              break;
23
24              default:
25              System.out.println( "study" );
26          }
27      }
28  }
```

In the above program, each constant has a unique value. Furthermore, the statement below produces a compile time error, as it should:

```
1  Day day = Month.JANUARY;
```

```
Enum.java:36: incompatible types
found   : E2.Month
required: E2.Day
        Day day = Month.JANUARY;
                          ^
1 error
```

- https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html.

Our Implementation

We are now ready to program our solution. We will only need four classes for this. For the assignment, you need to follow the patterns that we provide. You cannot change any of the signatures of the methods (that is you cannot modify the methods at all). You cannot add new *public* methods or variables. You can, however, add new *private* methods to improve the readability or the organization of your code.

GameState

GameState is an enum type which is used to capture the current state of the game. It has four possible values:

- *PLAYING*: this game is ongoing,
- *DRAW*: this game is a draw,
- *XWIN*: this game as been won by the first player,
- *OWIN*: this game as been won by the second player.

CellValue

CellValue is an enum type which is used to capture the state of a cell. It has three possible values:

- *EMPTY*: the cell is empty,
- *X*: there is a *X* in the cell,
- *O*: there is a *O* in the cell.

TicTacToeGame

Instances of the class *TicTacToeGame* represent a game being played. Each object stores the actual board, which is saved in a single dimension array. There is an instance method that can be used to play the next move. The object figures out the player's turn, so that information is not specified: we simply specify the index to play and the object knows to play either a *X* or a *O*. The object also tracks the state of the game automatically.

The specification for our class *TicTacToeGame* is given in our zip file. You need to fill out all the missing parts, reading carefully all the comments before doing so. You *cannot* modify the methods or the variables that are provided. You can, however, add new *private* methods as required.

The template that you are working with contains the following:

- An instance variable which is a reference to an array of CellValue to record the state of the board.
- Some instance variables to record the game's number of columns and lines, the number of cells to align, the number of turns played ("level") and current state.
- Three constructors: the default one creates the usual game (the 3x3 grid, with a win being 3 similar cells aligned), a second one can be used to specify both lines and columns, and the last one is used to specify lines, columns and number of cells to align. As usual, all the instance variables must be initialized when the object is constructed.
- Getters for lines, columns, sizeWin, level and game's current state
- A method to query the object about the next player (that is, is it *X*'s or *O*'s turn to play?).
- A method **play(int index)** to play at a particular location in the game. This updates the game state and the board.
- We also have a helper method, **private void setGameState(int index)**, which is used to compute and update the game state once a particular move is played in the method **play**.
- Finally, we have a method **toString()**, which returns a string representation of the current state of the board, as shown in the example before. An example of a String returned by toString would be, when printed out:

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

There are a few situations that need our attention. For example, the index selected by the player may be invalid or illegal. We do not have a very good way to handle these situations yet, so for the time being we will simply write an error message. The subsequent behaviour of the method is unspecified, so simply implement something that seems to make sense[1]. One other situation would be that players continue the game after one of them wins. For testing purpose, we actually want that to be possible, however, then game state should reflect the first winner of the game. So if the players keep going after a win, a message is printed out but the game continues as long as the moves are legal. The "first" winner remains.

Note that the method **toString()** returns a reference to a String, it is not actually printing anything. So that one String instance, when printed, should produce the expected output (in other words, that one string instance, when printed, will span several lines).

TicTacToe

This class implements playing the game. You are provided with the initial part, which creates the instance of the class TicTacToeGame based on the parameters submitted by the user. All you need to do here is implementing the remainder of the **main** method, the part that plays the game. It basically loops through each step of the game until the game is over. At each step, it displays the current game and prompts the next player, *X* or a *O*, for a cell to play. It then plays that cell and keeps going until the game is over, at which point it finishes (so although we said that TicTacToeGame is able to play past the winning point, that situation should never occur from the main of TicTacToe).

If the cell provided by the player is invalid or illegal, and a message is displayed to the user, who is asked to play again. Here are a couple of examples of this situation:

```
$ java TicTacToe
   |   |
-----------
   |   |
-----------
   |   |
```

[1] The reason we are not specifying any behaviour here is because once we will have the tools required to deal with these exceptional situations, we will see that we actually will not have to come up with an alternative behaviour at all.

```
X to play: 2
   | X |
-----------
   |   |
-----------
   |   |

O to play: 10
The value should be between 1 and 9
   | X |
-----------
   |   |
-----------
   |   |

O to play: 2
This cell has already been played
   | X |
-----------
   |   |
-----------
   |   |

O to play: 3
   | X | O
-----------
   |   |
-----------
   |   |

X to play:
```

Note that you can assume that the players are only providing integer values as inputs. You do not have to handle the case of other input types such as a character.

JUnit Tests

We are providing a set of JUnit tests for the class *TicTacToeGame*. These tests should of course help making sure that your implementation is correct. They can also help clarifying the expected behaviour of this class, if need be.

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: **a1_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 1. The parts are separated by the underscore (not the hyphen). There are no spaces in the name of the directory. The archive a1_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
- TicTacToe.java
- TicTacToeGame.java
- StudentInfo.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: January 26, 2020