ITI 1121. Introduction to Computing II
Winter 2020

Assignment 2
(Last modified on February 6, 2020)

Deadline: February 23, 2020, 11:30 pm

Learning objectives

- Using Interfaces
- Polymorphism
- Experiment with Deep-Copy
- Experiment with lists and enumerations

Introduction

In this assignment, we are continuing our work on the Tic-Tac-Toe game. In the previous assignment, we came up with a basic implementation of the game, that can be played by two humans. This time, we will first create a "computer player", which isn't very smart at all but can at least play the game according to the rules. We will thus be able to play human against computer. We will then put this aside and work on enumerating all the possible games. That enumeration will be used later when we create a computer player which can play well.

Human vs (Dumb) Machine

A very simple way to have a program play Tic-Tac-Toe is to simply have the program pick randomly an empty cell to play at each turn. Of course, such an implementation should be easy to beat, but at least it can be played against.

In order to design this solution, we want to introduce the concept of a **Player**. For now, we will have two kinds of players: the human player, and the dumb computer player. Later, we can introduce more types of players, e.g. a smart computer player, a perfect player etc. All of these are **Players**.

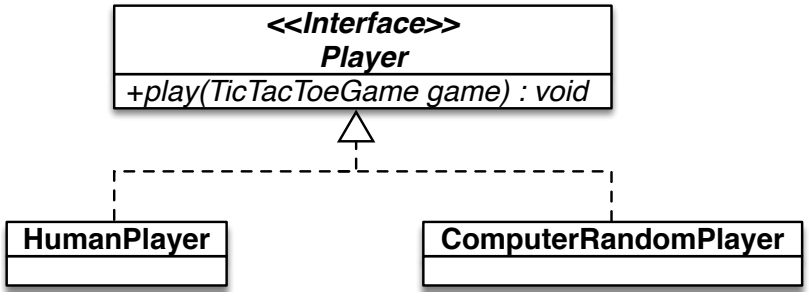


Figure 1: The interface Player and the two classes implementing it.

What we gain from this **Players** abstraction is that it is possible to organize a match between two players, and have these two players play a series of games, keeping score for the match etc., without having to worry about the type of players involved. We can have human vs human, human vs dumb computer, smart vs dumb computer players, or any combination of players, this does not impact the way the game is played: we have two players, and they alternate playing a move on the game until the game is over. The requirement to be able to do this is that all **Player** implement the same method, say **play()**, which can be called when it is that player's turn to play.

In our current implementation, we always play a human against a computer. Human is player 1, computer is player 2. The player who plays first initially is chosen randomly. In subsequent games, the players alternate as first player. As usual, the first player plays *X* and the second player plays *O* so each player will alternate between playing playing *X* and playing *O*.

The following printout shows a typical game.

```
$ java TicTacToe
************************************************************
*                                                          *
*                                                          *
*                                                          *
*                                                          *
************************************************************

Player 2's turn.
Player 1's turn.
   |   |
-----------
 X |   |
-----------
   |   |

0 to play:
```

Here, player 2 (the computer) was selected to start for the first game. As can be seen, the computer player doesn't print out anything when it plays, it just makes its move silently. Then, it is player 1's turn (human). Following what we did in assignment 1, the HumanPlayer object first prints the game (here, we can see that the computer played cell 4) and then prompts the actual human (us, the user) for a move. Below, we see that the human has selected cell 1. The computer will then play (silently) and the human will be prompt again. It continues until the game finishes:

```
0 to play: 1
Player 2's turn.
Player 1's turn.
 0 |   | X
-----------
 X |   |
-----------
   |   |

0 to play: 5
Player 2's turn.
Player 1's turn.
 0 |   | X
-----------
 X | 0 |
-----------
   |   | X

0 to play: 6
Player 2's turn.
Player 1's turn.
 0 | X | X
-----------
 X | 0 | 0
-----------
   |   | X

0 to play: 7
```

```
Player 2's turn.
Game over
 O | X | X
-----------
 X | O | O
-----------
 O | X | X

Result: DRAW
Play again (Y)?:
```

This game finishes with a DRAW. The sentence "Game over" is printed after the last move (made by the computer in this case), then the final board is printed, and the outcome of the game ("Result: DRAW").

The user is then asked if they want to play again.

Here, we want to play another game. This time, the human will make the first move. Below, you can see the entire game, which is a human win. Then a third game is played, also a human win, and we stop playing after this.

```
Play again (Y)?:y
Player 1's turn.
   |   |
-----------
   |   |
-----------
   |   |

X to play: 5
Player 2's turn.
Player 1's turn.
   |   |
-----------
   | X |
-----------
 O |   |

X to play: 1
Player 2's turn.
Player 1's turn.
 X |   |
-----------
   | X | O
-----------
 O |   |

X to play: 9
Game over
 X |   |
-----------
   | X | O
-----------
 O |   | X

Result: XWIN
Play again (Y)?:y
Player 2's turn.
Player 1's turn.
   |   |
-----------
   |   |
```

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

0 to play: 1
Player 2's turn.
Player 1's turn.
 O |   |
-----------
   |   |
-----------
 X | X |

0 to play: 9
Player 2's turn.
Player 1's turn.
 O |   |
-----------
   |   | X
-----------
 X | X | O

0 to play: 5
Game over
 O |   |
-----------
   | O | X
-----------
 X | X | O

Result: OWIN
Play again (Y)?:n
$
```

We are now ready to program our solution. We will reuse the implementation of the class *TicTacToeGame* from assignment 1. A class *Utils* has been provided to get a simple access to a few constants and global variables.

Player

Player is an interface. It defines only one method, the method **play**. Play is **void** and has one input parameter, a reference to a **TicTacToeGame**.

HumanPlayer

HumanPlayer is a class which implements the interface *Player*. In its implementation of the method **play**, it first checks that the game is indeed playable (and prints out an error message is that is not the case), and then queries the user for a valid input, reusing the code that was in the **main** of the class **TicTacToe** of assignment 1. Once such an input has been provided, it plays in on the game and returns.

ComputerRandomPlayer

ComputerRandomPlayer is a class which also implements the interface *Player*. In its implementation of the method **play**, it first checks that the game is indeed playable (and prints out an error message is that is not the case), and then chose randomly the next move and plays it on the game and returns. All the possible next moves have an equal chance of being played.

TicTacToe

This class implements playing the game. You are provided with the initial part very similar to the one from assignment 1. The entire game is played in the main method. A local variable **players**, a reference to an array of two players, is used to store the human and the computer player. You **must** use that array to store your **Player** references.

You need to finish the implementation of the main to obtain the specified behaviour. You need to ensure that the first player is initially chosen randomly, and that the first move alternate between both players in subsequent games.

Below is another sample run, this time on a 4x4 grid with a win length of 2. The human players makes a series of input mistakes along the way.

```
$ java TicTacToe 4 4 2
************************************************************
*                                                          *
*                                                          *
*                                                          *
*                                                          *
************************************************************

Player 1's turn.
   |   |   |
---------------
   |   |   |
---------------
   |   |   |
---------------
   |   |   |

X to play: 2
Player 2's turn.
Player 1's turn.
   | X |   |
---------------
   |   | O |
---------------
   |   |   |
---------------
   |   |   |

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

X to play: -1
The value should be between 1 and 16
   | X |   |
---------------
   |   | O |
---------------
   |   |   |
---------------
```

```
   |   |   |

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

Result: XWIN
Play again (Y)?:y
Player 2's turn.
Player 1's turn.
   |   |   |
---------------
   |   |   |
---------------
   |   | X |
---------------
   |   |   |

O to play: 11
This cell has already been played
   |   |   |
---------------
   |   |   |
---------------
   |   | X |
---------------
   |   |   |

O to play: 12
Player 2's turn.
Player 1's turn.
   |   | X |
---------------
   |   |   |
---------------
   |   | X | O
---------------
   |   |   |

O to play: 13
Player 2's turn.
Game over
   |   | X |
---------------
   |   |   |
---------------
   |   | X | O
---------------
 O | X |   |

Result: XWIN
```

```
Play again (Y)?:n
 $
```

Games enumeration

We are now looking at something else: game enumerations. We would like to generate all the possible games for a given size of grid and win size.

For example, if we take the default, 3x3 grid, there is 1 grid at level 0, namely:

```
   |   |
-----------
   |   |
-----------
   |   |
```

There are then 9 grids at level 1, namely:

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


   | X |
-----------
   |   |
-----------
   |   |


   |   | X
-----------
   |   |
-----------
   |   |


   |   |
-----------
 X |   |
-----------
   |   |


   |   |
-----------
   | X |
-----------
   |   |


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

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


   |   |
-----------
   |   |
-----------
   | X |


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

There are then 72 grids at level 2, too many to print here. In Appendix A, we provide the complete list of games for a 2x2 grid, with a win size of 2. Note that no game of level 4 appears on that list: it is simply impossible to reach level 3 and not win on a 2x2 grid with a win size of 2. In our enumeration, we do not list the same game twice, and we do not continue after a game has been won.

Our Implementation

For this implementation, we are going to add a couple of new methods to our class **TicTacToeGame** and we will create a new class, **ListOfGamesGenerator**, to generate our games. We will store our games in a list of lists. We will have our own implementation of the abstract data type List very soon, but we do not have it yet. Therefore, exceptionally for ITI1(1/5)21, we are going to use a ready-to-use solution. In this case, we will use **java.util.LinkedList**. The documentation is available at https://docs.oracle.com/javase/9/docs/api/java/util/LinkedList.html.

The goal is to create a list of lists: each list will have all the different games for a given level. Consider again the default, 3x3 grid. Our list will have 10 elements.

- The first element is the list of 3x3 grid at level 0. There is 1 such grid, so this list has 1 element.
- The second element is the list of 3x3 grid at level 1. There are 9 such grids, so this list has 9 elements.
- The third element is the list of 3x3 grid at level 2. There are 72 such grids, so this list has 72 elements.
- The fourth element is the list of 3x3 grid at level 3. There are 252 such grids, so this list has 252 elements.
- The fifth element is the list of 3x3 grid at level 4. There are 756 such grids, so this list has 756 elements.

etc.

- The ninth element is the list of 3x3 grid at level 8. There are 390 such grids, so this list has 390 elements.
- The tenth element is the list of 3x3 grid at level 9. There are 78 such grids, so this list has 78 elements.

The class **TicTacToe.java** is provided to you. It calls the generation of the list and prints out some information about it. Here are a few typical runs:

```
$ java TicTacToe
************************************************************
*                                                          *
*                                                          *
```

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

======= level 0 =======: 1 element(s) (1 still playing)
======= level 1 =======: 9 element(s) (9 still playing)
======= level 2 =======: 72 element(s) (72 still playing)
======= level 3 =======: 252 element(s) (252 still playing)
======= level 4 =======: 756 element(s) (756 still playing)
======= level 5 =======: 1260 element(s) (1140 still playing)
======= level 6 =======: 1520 element(s) (1372 still playing)
======= level 7 =======: 1140 element(s) (696 still playing)
======= level 8 =======: 390 element(s) (222 still playing)
======= level 9 =======: 78 element(s) (0 still playing)
that's 5478 games
626  won by X
316  won by 0
16 draw
 $ java TicTacToe 3 3 2
************************************************************
*                                                          *
*                                                          *
*                                                          *
*                                                          *
************************************************************

======= level 0 =======: 1 element(s) (1 still playing)
======= level 1 =======: 9 element(s) (9 still playing)
======= level 2 =======: 72 element(s) (72 still playing)
======= level 3 =======: 252 element(s) (112 still playing)
======= level 4 =======: 336 element(s) (136 still playing)
======= level 5 =======: 436 element(s) (40 still playing)
======= level 6 =======: 116 element(s) (4 still playing)
======= level 7 =======: 12 element(s) (0 still playing)
that's 1234 games
548  won by X
312  won by 0
0 draw
 $ java TicTacToe 2 2 2
************************************************************
*                                                          *
*                                                          *
*                                                          *
*                                                          *
************************************************************

======= level 0 =======: 1 element(s) (1 still playing)
======= level 1 =======: 4 element(s) (4 still playing)
======= level 2 =======: 12 element(s) (12 still playing)
======= level 3 =======: 12 element(s) (0 still playing)
that's 29 games
12  won by X
0  won by 0
0 draw
 $ java TicTacToe 2 2 3
************************************************************
*                                                          *
```

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

======= level 0 =======: 1 element(s) (1 still playing)
======= level 1 =======: 4 element(s) (4 still playing)
======= level 2 =======: 12 element(s) (12 still playing)
======= level 3 =======: 12 element(s) (12 still playing)
======= level 4 =======: 6 element(s) (0 still playing)
that's 35 games
0  won by X
0  won by O
6 draw
$ java TicTacToe 5 2 3
************************************************************
*                                                          *
*                                                          *
*                                                          *
*                                                          *
************************************************************

======= level 0 =======: 1 element(s) (1 still playing)
======= level 1 =======: 10 element(s) (10 still playing)
======= level 2 =======: 90 element(s) (90 still playing)
======= level 3 =======: 360 element(s) (360 still playing)
======= level 4 =======: 1260 element(s) (1260 still playing)
======= level 5 =======: 2520 element(s) (2394 still playing)
======= level 6 =======: 3990 element(s) (3798 still playing)
======= level 7 =======: 3990 element(s) (3290 still playing)
======= level 8 =======: 2580 element(s) (2162 still playing)
======= level 9 =======: 1032 element(s) (646 still playing)
======= level 10 =======: 150 element(s) (0 still playing)
that's 15983 games
1212  won by X
660  won by O
100 draw
 $ java TicTacToe 2 5 3
************************************************************
*                                                          *
*                                                          *
*                                                          *
*                                                          *
************************************************************

======= level 0 =======: 1 element(s) (1 still playing)
======= level 1 =======: 10 element(s) (10 still playing)
======= level 2 =======: 90 element(s) (90 still playing)
======= level 3 =======: 360 element(s) (360 still playing)
======= level 4 =======: 1260 element(s) (1260 still playing)
======= level 5 =======: 2520 element(s) (2394 still playing)
======= level 6 =======: 3990 element(s) (3798 still playing)
======= level 7 =======: 3990 element(s) (3290 still playing)
======= level 8 =======: 2580 element(s) (2162 still playing)
======= level 9 =======: 1032 element(s) (646 still playing)
======= level 10 =======: 150 element(s) (0 still playing)
that's 15983 games
```

```
1212  won by X
660  won by 0
100 draw
 $
```

TicTacToeGame

We need to add two new public methods to the class **TicTacToeGame**:

- **public TicTacToeGame(TicTacToeGame base, int next)**: this new constructor is used to create a new instance of the class TicTacToeGame, based on an existing instance *base*. The next instance will be a game in the state of the game referenced by *base*, in which the position *next* has been played. For example, imagine that *base* is a reference to the following game:

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

  A call to

  ```
  new TicTacToeGame(base,7)
  ```

  returns a reference to the following game:

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

  One important consideration in implementing this constructor is that the game referenced by *base* **should not be modified** by the call. Have a look at Appendix B to better understand what is required to achieve this.

- **public boolean equals(TicTacToeGame other)** compares the current game with the game referenced by *other*. This method returns **true** if and only if both games are considered the same: they have the same characteristics, and their board is in the same state.

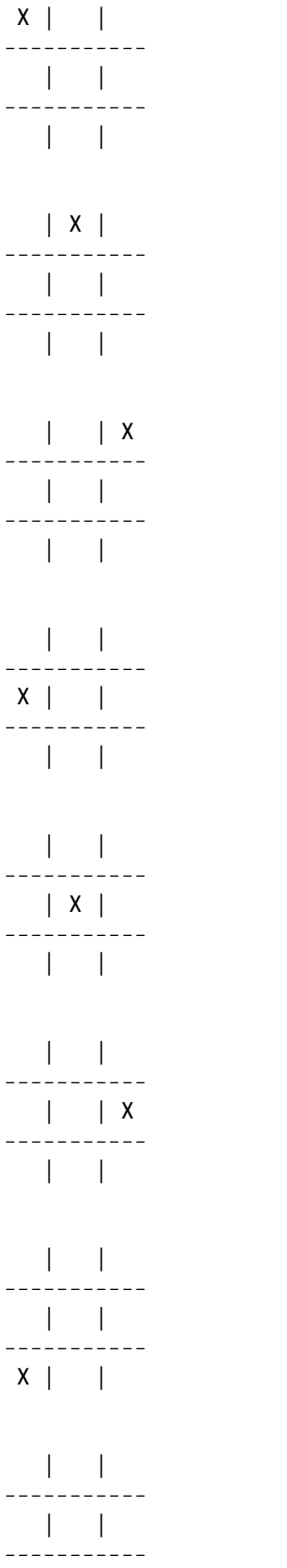
ListOfGamesGenerator

This new class has a single method, which computes the list of lists that we are interested in.

- **public static LinkedList<LinkedList<TicTacToeGame> > generateAllGames(int lines, int columns, int winLength)**: this method returns the (Linked) list of (Linked) lists of TicTacToeGame references that we are looking for, for the games on a grid *lines*x*columns* with a win size of *winLength*. As explained, each of the (secondary) lists contains the lists of references to the game of the same level. There are three important factors to take into account when building the list:
  - We only build games up to their winning point (or until they reach a draw). We never extend a game that is already won.
  - We do not duplicate games. There are several ways to reach the same state, so make sure that the same game is not listed several times.
  - We do not include empty lists. As can be seen in A, we stop our enumeration once all the games are finished. In the 2x2 case with a win size of 2, since all the games are finished after 3 moves, the list of lists has only 4 elements: games after 0 move, games after 1 move, games after 2 moves and games after 3 moves.

Symmetry

Our implementation does not duplicate games that are identical, but it still lists many games that are equivalent. For example, consider the first move on a 3x3 grid. There are 9 such first move:



```
   | X |


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

However, playing any corner (cells 1, 3, 7 and 9) is equivalent, so the four initial grids playing into one of the four corners are the same: that is basically the same opening move. Similarly, playing any of the four "middles" (cells 2, 4, 6 and 8) is also equivalent. So in reality, up to symmetry, we only have 3 possible first move:

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


   | X |
-----------
   |   |
-----------
   |   |


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

Every other opening move is equivalent to one of these three ones. So in essence, we do have a lot of logical repetition in the list of games that we have generated previously.

In the next assignment, our first step is going to remove these symmetrical, equivalent games from our list. We encourage you to think about this, and try to come up with your own way of addressing this problem. We will give a 10% bonus mark to you if you include in Q3 your own (correct) solution for generating all the games up to symmetry. (please note that you will still be asked to implement our own solution for the next assignment!)

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. However, if you do the assignment in teams, your assignment must only be submitted once to brightspace. If both partners submit the assignment, a penalty of 20% for assignment 2, of 30% for assignment 3 and of 40% for assignment 4 will be given.
- 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[1]

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: **a2_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 2. The parts are separated by the underscore (not the hyphen). There are no spaces in the name of the directory. The archive a2_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).
- A subdirectory Q1 which contains the following files:
  - CellValue.java
  - ComputerRandomPlayer.java
  - GameState.java
  - HumanPlayer.java
  - Player.java
  - StudentInfo.java
  - TicTacToe.java
  - TicTacToeGame.java
  - Utils.java
- A subdirectory Q2 which contains the following files:
  - CellValue.java

[1] A penalty of 20% for assignment 2, of 30% for assignment 3 and of 40% for assignment 4 will be given if you do not strictly follow the instructions.

  - GameState.java
  - ListOfGamesGenerator.java
  - StudentInfo.java
  - TicTacToe.java
  - TicTacToeGame.java
  - Utils.java
- If you have a solution for the symmetry question, put your files in a subdirectory Q3.

Questions

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

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

A Enumerating all games of a 2x2 grid

```
======= level 0 =======: 1 element(s)
   |
-------
   |


======= level 1 =======: 4 element(s)
 X |
-------
   |


   | X
-------
   |


   |
-------
 X |


   |
-------
   | X


======= level 2 =======: 12 element(s)
 X | O
-------
   |


 X |
-------
 O |


 X |
-------
   | O


 O | X
-------
   |


   | X
-------
 O |


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

```
   | O


 O |
-------
 X |


   | O
-------
 X |


   |
-------
 X | O


 O |
-------
   | X


   | O
-------
   | X


   |
-------
 O | X


======= level 3 =======: 12 element(s)
 X | O
-------
 X |


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


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


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


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

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


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


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


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


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


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


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

B Shallow copy versus Deep copy

As you know, objects have variables which are either a primitive type, or a reference type. Primitive variables hold a value from one of the language primitive type, while reference variables hold a reference (the address) of another object (including arrays, which are objects in Java).

If you are copying the current state of an object, in order to obtain a duplicate object, you will create a copy of each of the variables. By doing so, the value of each instance primitive variable will be duplicated (thus, modifying one of these values in one of the copy will not modify the value on the other copy). However, with reference variables, what will be copied is the actual reference, the address of the object that this variable is pointing at. Consequently, the reference variables in both the original object and the duplicated object will point at the same address, and the reference variables will refer to the same objects. This is known as a **shallow** copy: you indeed have two objects, but they share all the objects pointed at by their instance reference variables. The Figure B provides an example: the object referenced by variable **b** is a shallow copy of the object referenced by variable **a**: it has its own copies of the instances variables, but the references variables **title** and **time** are referencing the same objects.

Often, a shallow copy is not adequate: what is required is a so-called **deep** copy. A deep copy differs from a shallow copy in that objects referenced by reference variable must also be recursively duplicated, in such a way that when the initial object is (deep) copied, the copy does not share any reference with the initial object. The Figure B provides an example: this time, the object referenced by variable **b** is a deep copy of the object referenced

by variable **a**: now, the references variables **title** and **time** are referencing different objects. Note that, in turn, the objects referenced by the variable **time** have also been deep-copied. The entire set of objects reachable from **a** have been duplicated.

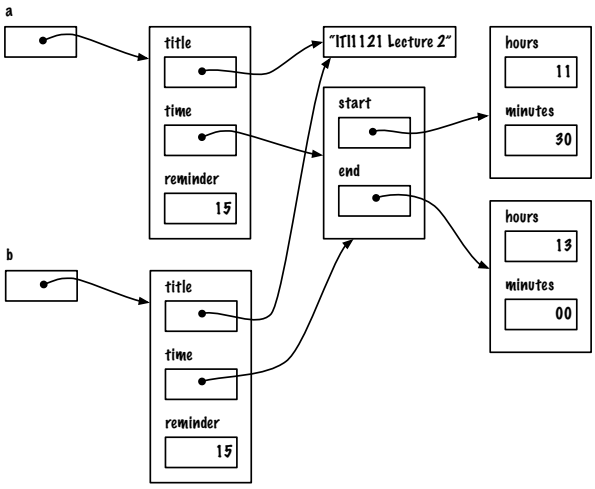


Figure 2: A example of a shallow copy of objects.

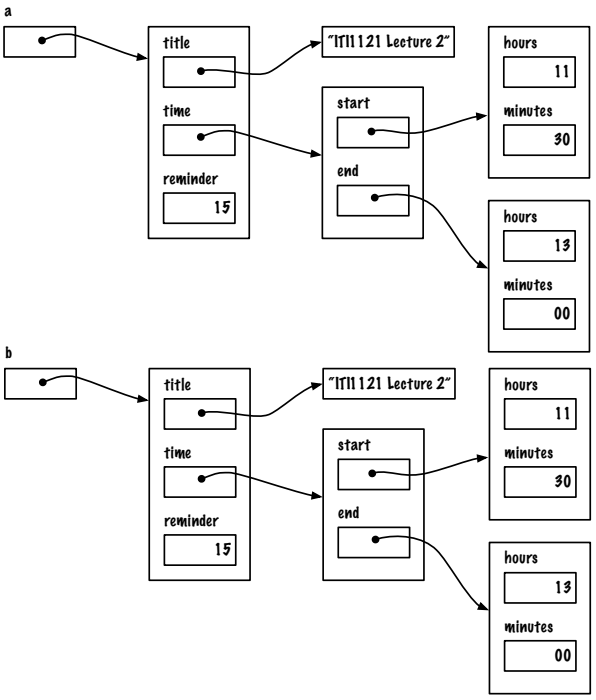


Figure 3: A example of a deep copy of objects.

You can read more about shallow versus deep copy on Wikipedia:

- Object copying

Last modified: February 6, 2020