ITI 1521. Introduction à l'informatique II
Hiver 2020

Devoir 2
(Modifié le 9 février 2020)

Échéance: 23 février, 2020, 23 h 30

Objectifs d'apprentissage

- **Utiliser** des interfaces
- **Appliquer** le polymorphisme dans la conception d'une application
- **Explorer** le concept de copie profonde
- **Expérimenter** les listes et les énumérations

Introduction

Dans le cadre de ce devoir, nous poursuivons notre travail sur le jeu du Tic-Tac-Toe. Lors du dernier devoir, nous avons mis au point une implémentation de base du jeu, qui peut être joué par deux humains. Cette fois, nous allons d'abord créer un «joueur informatique», qui n'est pas très intelligent mais qui peut au moins jouer selon les règles du jeu. Nous pourrons ainsi jouer humain contre ordinateur. Nous mettrons ensuite cela de côté et nous nous efforcerons d'énumérer tous les jeux possibles. Cette énumération sera utilisée plus tard lorsque nous créerons un joueur informatique qui pourra bien jouer.

Humain contre machine (sans intelligence)

Une façon très simple de faire jouer un programme au Tic-Tac-Toe est de lui faire choisir au hasard une case vide à jouer à chaque tour. Bien sûr, une telle implémentation devrait être facile à battre, mais au moins on peut jouer contre elle.

Afin de concevoir cette solution, nous voulons introduire le concept de joueur (**Player**). Pour l'instant, nous aurons deux catégories de joueurs : le joueur humain, et le joueur informatique sans intelligence. Plus tard, nous pourrons introduire d'autres types de joueurs, par exemple un joueur informatique intelligent, un joueur parfait, etc. Ce sont tous des joueurs (de type **Player**).

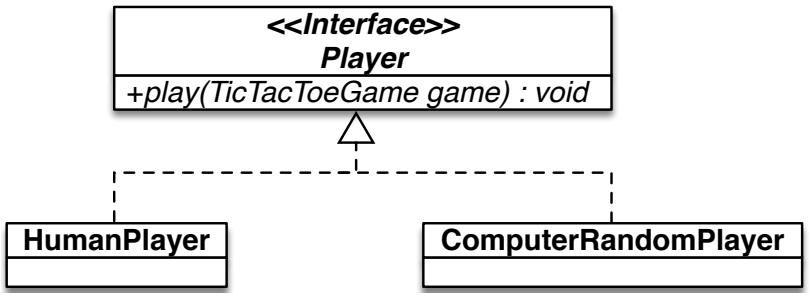


Figure 1 – L'interface **Player** et les deux classes qui la mettent en oeuvre.

Ce que nous obtenons de cette abstraction (**Player**), c'est qu'il est possible d'organiser un match entre deux joueurs, et de faire jouer ces deux joueurs une série de parties, en comptant les points pour le match, etc. Nous pouvons avoir des joueurs humains contre humains, humains contre ordinateurs sans intelligence, intelligents contre

ordinateurs sans intelligence, ou n'importe quelle combinaison de joueurs, cela n'a pas d'impact sur la façon dont le jeu est joué : nous avons deux joueurs, et ils alternent en jouant un coup sur le jeu jusqu'à ce que la partie soit terminée. La condition pour pouvoir faire cela est que tous les **joueurs** possèdente une méthode **play()**, qui peut être appelée lorsque c'est le tour de ce joueur de jouer.

Dans notre implémentation actuelle, nous jouons toujours un humain contre un ordinateur. L'humain est le joueur 1, l'ordinateur est le joueur 2. Le joueur qui joue en premier est choisi au hasard. Dans les parties suivantes, les joueurs alternent comme premier joueur. Comme d'habitude, le premier joueur joue *X* et le deuxième joueur joue *O*, de sorte que chaque joueur alterne entre jouer *X* et jouer *O*.

L'impression suivante montre un jeu typique.

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

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

0 to play:
```

Ici, le joueur 2 (l'ordinateur) a été sélectionné pour commencer la première partie. Comme on peut le voir, le joueur de l'ordinateur n'imprime rien lorsqu'il joue, il fait juste son coup en silence. Ensuite, c'est au tour du joueur 1 (l'humain). Suivant ce que nous avons fait dans le devoir 1, l'objet **HumanPlayer** imprime d'abord la partie (ici, on peut voir que l'ordinateur a joué la case 4) et demande ensuite à l'humain (nous, l'utilisateur) de jouer. Ci-dessous, nous voyons que l'humain a sélectionné la cellule 1. L'ordinateur jouera alors (en silence) et l'humain sera à nouveau invité à jouer. Cela continue jusqu'à la fin de la partie :

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

O to play: 7
Player 2's turn.
Game over
 O | X | X
-----------
 X | O | O
-----------
 O | X | X

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

Ce jeu se termine par un match nul (**DRAW**). La phrase «Game over» est imprimée après le dernier coup (effectué par l'ordinateur dans ce cas), puis le tableau final est imprimé, et le résultat de la partie («Result : DRAW»).

On demande alors à l'utilisateur s'il veut jouer à nouveau.

Ici, nous voulons jouer un autre jeu. Cette fois, c'est l'humain qui fera le premier coup. Ci-dessous, vous pouvez voir l'ensemble de la partie, qui est une victoire pour l'humain. Ensuite, une troisième partie est jouée, également une victoire pour l'humain, et nous arrêtons de jouer après cela.

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

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

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

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

Nous sommes maintenant prêts à programmer notre solution. Nous allons réutiliser l'implémentation de la classe **TicTacToeGame** du devoir 1. Une classe **Utils** a été fournie pour avoir un accès à quelques constantes et variables globales.

Player

Player est une interface. Elle ne définit qu'une seule méthode, la méthode **play**. Le type de sa valeur de retour est **void** et elle a un paramètre d'entrée, une référence à un objet de la classe **TicTacToeGame**.

HumanPlayer

HumanPlayer est une classe qui réalise l'interface **Player**. Dans son implémentation de la méthode **play**, elle vérifie d'abord que le jeu est effectivement jouable (et imprime un message d'erreur si ce n'est pas le cas), puis demande à l'utilisateur une entrée valide, en réutilisant le code qui se trouvait dans la méthode **main** de la classe **TicTacToe** du devoir 1. Une fois qu'une telle entrée a été fournie, il joue le jeu et donne le contrôle à l'autre joueur.

ComputerRandomPlayer

ComputerRandomPlayer est une classe qui réalise également l'interface **Player**. Dans son implémentation de la méthode **play**, elle vérifie d'abord que la partie est effectivement jouable (et imprime un message d'erreur si ce n'est pas le cas), puis choisit au hasard le coup suivant, le joue et passe la main à l'autre joueur. Tous les coups suivants possibles ont une chance égale d'être joués.

TicTacToe

Cette classe implémente le jeu. La partie initiale est très similaire à celle du devoir 1. L'ensemble du jeu est joué dans la méthode principale. Une variable locale **players**, une référence à un tableau de deux joueurs, est utilisée pour stocker le joueur humain et le joueur informatique. Vous **devez** utiliser ce tableau pour stocker vos références **Player**.

Vous devez terminer l'implémentation de la méthode principale afin d'obtenir le comportement spécifié. Vous devez vous assurer que le premier joueur est initialement choisi au hasard et que le premier coup alterne entre les deux joueurs lors des parties suivantes.

Voici un autre exemple de jeu, cette fois sur une grille de 4×4 et 2 cellules alignées pour une victoire. Les joueurs humains font une série d'erreurs de saisie en cours de route.

```
$ 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
 $
```

Énumérations de jeux

Nous nous intéressons maintenant à autre chose : les énumérations (dénombrements) de jeux. Nous souhaitons générer tous les jeux possibles pour une taille de grille et nombre de cellules alignées pour une victoire donnés.

Par exemple, si nous prenons la grille par défaut, 3×3, il y a 1 grille au niveau 0, à savoir :

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

Il y a ensuite 9 grilles au niveau 1, à savoir :

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


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


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


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


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

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


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


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


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

Il y a alors 72 grilles au niveau 2, trop nombreuses pour être imprimées ici. En annexe sectionA, nous fournissons la liste complète des jeux pour une grille 2×2, avec une taille de victoire de 2. Notez qu'aucun jeu de niveau 4 n'apparaît sur cette liste : il est simplement impossible d'atteindre le niveau 3 et de ne pas gagner sur une grille 2×2 et un nombre de cellules à alignées de 2 pour une victoire. Dans notre énumération, nous n'énumérons pas deux fois le même jeu, et nous ne continuons pas après qu'une partie ait été gagnée.

Notre implémentation

Pour cette implémentation, nous allons ajouter quelques nouvelles méthodes à notre classe **TicTacToeGame** et nous allons créer une nouvelle classe, **ListOfGamesGenerator**, pour générer nos jeux. Nous allons stocker nos jeux dans une liste de listes. Nous aurons très bientôt notre propre implémentation du type de données abstrait **List**, mais nous ne l'avons pas encore. Par conséquent, exceptionnellement pour ITI1(1/5)21, nous allons utiliser une solution clé en main. Dans ce cas, nous utiliserons **java.util.linkedList**. La documentation est disponible à l'adresse `https://docs.oracle.com/javase/9/docs/api/java/util/LinkedList.html`.

Le but est de créer une liste de listes : chaque liste contiendra tous les différents jeux pour un niveau donné. Reprenons la grille par défaut, 3×3. Notre liste comportera 10 éléments.

- Le premier élément est la liste des grilles 3×3 au niveau 0. Il y a 1 telle grille, donc cette liste a 1 élément.
- Le deuxième élément est la liste des grilles 3×3 au niveau 1. Il y a 9 grilles de ce type, donc cette liste comporte 9 éléments.
- Le troisième élément est la liste des grilles 3×3 au niveau 2. Il existe 72 grilles de ce type, donc cette liste comporte 72 éléments.
- Le quatrième élément est la liste des grilles 3×3 au niveau 3. Il existe 252 grilles de ce type, donc cette liste comporte 252 éléments.
- Le cinquième élément est la liste des grilles 3×3 au niveau 4. Il y a 756 grilles de ce type, donc cette liste comporte 756 éléments.

etc.

- Le neuvième élément est la liste des grilles 3×3 au niveau 8. Il existe 390 grilles de ce type, donc cette liste comporte 390 éléments.
- Le dixième élément est la liste des grilles 3×3 au niveau 9. Il existe 78 grilles de ce type, donc cette liste comporte 78 éléments.

La classe **TicTacToe.java** vous est fournie. Elle appelle la génération de la liste et imprime quelques informations à son sujet. Voici quelques exécutions typiques :

```
$ 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 O
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

Nous devons ajouter deux nouvelles méthodes publiques à la classe **TicTacToeGame** :

- **public TicTacToeGame(TicTacToeGame base, int next)** : ce nouveau constructeur est utilisé pour créer une nouvelle instance de la classe **TicTacToeGame** basée sur une instance existante, **base**. La prochaine instance sera un jeu dont l'état est le même que celui référencé par **base**, mais dans lequel la position suivante (**next**) a été jouée. Par exemple, imaginez que **base** est une référence au jeu suivant :

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

  L'appel suivant :

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

  renvoie une référence au jeu suivant :

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

  Une considération importante dans la mise en œuvre de ce constructeur est que le jeu référencé par **base ne doit pas être modifié** par l'appel. Consultez l'annexe sectionB pour mieux comprendre ce qui est nécessaire pour y parvenir.

- **public boolean equals(TicTacToeGame other)** compare le jeu actuel avec le jeu référencé par **other**. Cette méthode renvoie **true** si et seulement si les deux jeux sont considérés comme identiques : ils ont les mêmes caractéristiques, et leur grille est dans le même état.

ListOfGamesGenerator

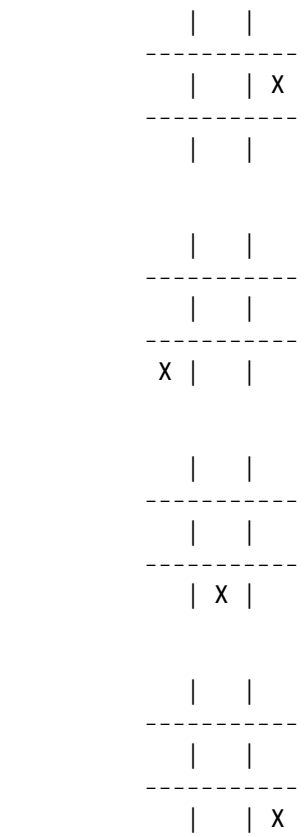
Cette nouvelle classe a une seule méthode, qui calcule la liste des listes qui nous intéressent.

- **public static LinkedList<LinkedList<TicTacToeGame> > generateAllGames(int lines, int columns, int winLength)** : cette méthode retourne la liste (chaînée) des listes (chaînées) de références **TicTacToeGame** que nous cherchons, pour les jeux sur une grille **lines**×**columns** avec un nombre de cellules alignées pour une victoire de **winLength**. Comme précisé, chacune des listes (secondaires) contient les listes de références au jeu du même niveau. Il y a trois facteurs importants à prendre en compte lors de l'élaboration de la liste :
  - Nous ne construisons des jeux que jusqu'à leur point de victoire (ou jusqu'à ce qu'ils atteignent le point d'égalité). Nous ne prolongeons jamais une partie déjà gagnée.
  - Nous ne dupliquons pas les jeux. Il y a plusieurs façons d'atteindre le même état, alors assurez-vous qu'un même jeu n'est pas listé plusieurs fois.
  - Nous n'incluons pas de listes vides. Comme on peut le voir dans sectionA, nous arrêtons notre énumération une fois que tous les jeux sont terminés. Dans le cas 2×2 avec un nombre de cellules alignées pour une victoire de 2, puisque toutes les parties sont terminées après 3 coups, la liste des listes ne comporte que 4 éléments : les parties après 0 coup, les parties après 1 coup, les parties après 2 coups et les parties après 3 coups.

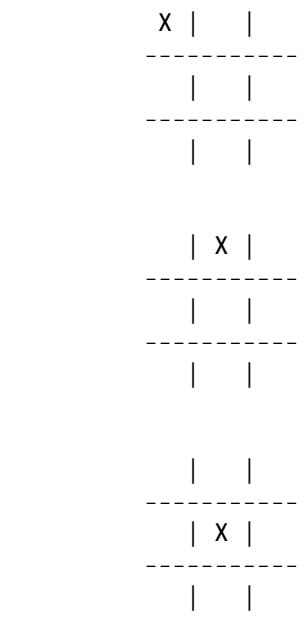
Symétrie

Notre implémentation ne duplique pas les jeux qui sont identiques, mais elle énumère tout de même de nombreux jeux qui sont équivalents. Par exemple, considérons le premier coup sur une grille de 3×3. Il y a 9 premiers coups de ce type :

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


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


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


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


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



Cependant, jouer n'importe quel coin (cases 1, 3, 7 et 9) est équivalent, donc les quatre grilles initiales jouant dans l'un des quatre coins sont les mêmes : c'est en gros le même coup d'ouverture. De même, jouer n'importe lequel des quatre «milieux» (cases 2, 4, 6 et 8) est également équivalent. Donc, en réalité, compte tenu de la symétrie, nous n'avons que 3 premiers coups possibles :



Un coup d'ouverture sur deux équivaut à l'un de ces trois coups. Donc, en substance, nous avons beaucoup de répétitions logiques dans la liste des jeux que nous avons générés précédemment.

Dans notre prochain devoir, nous allons travailler uniquement sur des jeux de 3×3, et notre première étape sera de supprimer ces jeux symétriques et équivalents de notre liste. Nous vous encourageons à y réfléchir, et à essayer de trouver votre propre façon de résoudre ce problème. Nous vous donnerons un bonus de 10 % si vous incluez dans Q3 votre propre solution (correcte) pour générer tous les jeux de 3×3 compte tenu de la symétrie. (veuillez noter que vous devrez tout de même implémenter notre propre solution pour le prochain devoir !).

Intégrité académique

Cette partie du devoir vise à sensibiliser les étudiants au plagiat et à l'intégrité académique. Veuillez lire les documents suivants.

- https://www.uottawa.ca/administration-et-gouvernance/reglement-scolaire-14-autres-informations-impor
- https://www.uottawa.ca/vice-recteur-etudes/integrite-etudes

Les cas de plagiat seront traités conformément au règlement de l'université. En soumettant ce travail, vous reconnaissez :

1. J'ai lu le règlement académique sur la fraude académique.
2. Je comprends les conséquences du plagiat.
3. À l'exception du code source fourni par les instructeurs pour ce cours, tout le code source est le mien.
4. Je n'ai collaboré avec aucune autre personne, à l'exception de mon partenaire dans le cas d'un travail d'équipe.
    - Si vous avez collaboré avec d'autres personnes ou obtenu le code source sur le Web, veuillez alors indiquer le nom de vos collaborateurs ou la source de l'information, ainsi que la nature de la collaboration. Mettez ces informations dans le fichier README.txt soumis. Des points seront déduits proportionnellement au niveau de l'aide fournie (de 0 à 100%).

Directives

- Suivez toutes les directives disponibles sur la page Web Consignes pour la remise de tous vos travaux pratiques.
- Soumettez votre devoir par le biais du système de soumission en ligne campus virtuel.
- Vous devez préférablement réaliser le travail en équipe de deux, mais vous pouvez également le faire individuellement. Toutefois, si vous effectuez le travail en équipe, votre devoir ne doit être soumis qu'une seule fois sur Brightspace. Si les deux partenaires soumettent le devoir, une pénalité de 20 % pour le devoir 2, de 30 % pour le devoir 3 et de 40 % pour le devoir 4 sera appliquée.
- Vous devez utiliser les classes de modèles fournies ci-dessous.
- Si vous ne suivez pas les instructions, votre programme fera échouer les tests automatisés et, par conséquent, votre devoir ne sera pas noté.
- Nous utiliserons un outil automatisé pour comparer toutes les travaux les uns par rapport aux autres (y compris les sections française et anglaise). Les soumissions qui sont signalées par cet outil recevront la note 0.
- Il vous incombe de vous assurer que BrightSpace a bien reçu votre travail. Les soumissions tardives ne seront pas notées.

Fichiers[1]

Vous devez fournir un fichier **zip** (aucun autre format de fichier ne sera accepté). Le nom du répertoire principal doit avoir la forme suivante : **a2_3000000_3000001**, où 3000000 et 3000001 sont les numéros d'étudiant des membres de l'équipe qui soumettent le devoir (il suffit de répéter le même numéro si votre équipe compte un seul membre). Le nom du répertoire commence par la lettre «a» (minuscule), suivie du numéro du devoir, ici 2. Les parties sont séparées par le trait de soulignement (pas le trait d'union). Il n'y a pas d'espace dans le nom du répertoire.

L'archive a2_3000000_3000001.zip contient les fichiers que vous devez utiliser comme point de départ. Votre soumission doit contenir les fichiers suivants.

- README.txt

[1] Une pénalité de 20 % pour le devoir 2, de 30 % pour le devoir 3 et de 40 % pour le devoir 4 vous sera infligée si vous ne suivez pas strictement les instructions.

  - Un fichier texte qui contient les noms des deux partenaires pour le devoir, leurs numéros d'étudiant, la section et une courte description du devoir (une ou deux lignes).
- Un sous-répertoire Q1 qui contient les fichiers suivants :
  - CellValue.java
  - ComputerRandomPlayer.java
  - GameState.java
  - HumanPlayer.java
  - Player.java
  - StudentInfo.java
  - TicTacToe.java
  - TicTacToeGame.java
  - Utils.java
- Un sous-répertoire Q2 qui contient les fichiers suivants :
  - CellValue.java
  - GameState.java
  - ListOfGamesGenerator.java
  - StudentInfo.java
  - TicTacToe.java
  - TicTacToeGame.java
  - Utils.java
- Si vous avez une solution pour la question de symétrie, placez vos fichiers dans un sous-répertoire Q3.

Questions

Pour toutes vos questions, veuillez consulter le site web de la Piazza pour ce cours :

- https://piazza.com/uottawa.ca/winter2020/iti1521/home

A Énumération de tous les jeux d'une grille 2×2

```
======= 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 Copie de surface et copie profonde

Comme vous le savez, les objets ont des variables qui ont soit un type primitif, soit un type référence. Les variables d'un type primitif contiennent une valeur d'un type primitif du langage, tandis que les variables référence contiennent une référence (l'adresse) d'un autre objet (y compris les tableaux, qui sont des objets en Java).

Si vous copiez l'état actuel d'un objet, afin d'obtenir un objet doublon, vous créerez une copie de chacune des variables. Ce faisant, la valeur de chaque variable primitive d'instance sera dupliquée (ainsi, la modification de l'une de ces valeurs dans l'une des copies ne modifiera pas la valeur de l'autre copie). Toutefois, dans le cas des variables référence, ce qui sera copié est la référence elle-même, c'est-à-dire l'adresse de l'objet vers lequel cette variable pointe. Par conséquent, les variables référence de l'objet original et de l'objet dupliqué pointeront vers la même adresse, et les variables référence feront référence aux mêmes objets. C'est ce que l'on appelle une copie de surface (**shallow copy**) : vous avez en effet deux objets, mais ils partagent tous les objets pointés par leurs variables référence d'instance. La figure minipageB fournit un exemple : l'objet référencé par la variable **b** est une copie de surface de l'objet référencé par la variable **a** : il possède ses propres copies des variables d'instance, mais les variables référence **title** et **time** font référence aux mêmes objets.

Souvent, une copie superficielle n'est pas suffisante : ce qu'il faut, c'est une copie dite **profonde**. Une copie profonde diffère d'une copie de surface en ce que les objets référencés par une variable référence doivent également être dupliqués de manière récursive, de telle sorte que lorsque l'objet initial est copié (en profondeur), la copie ne

partage aucune référence avec l'objet initial. La figure minipageB fournit un exemple : cette fois, l'objet référencé par la variable **b** est une copie profonde de l'objet référencé par la variable **a** : maintenant, les variables référence **title** et **time** référencent des objets différents. Notez que, à leur tour, les objets référencés par la variable **time** ont également été copiés en profondeur. L'ensemble des objets accessibles à partir de **a** ont été dupliqués.

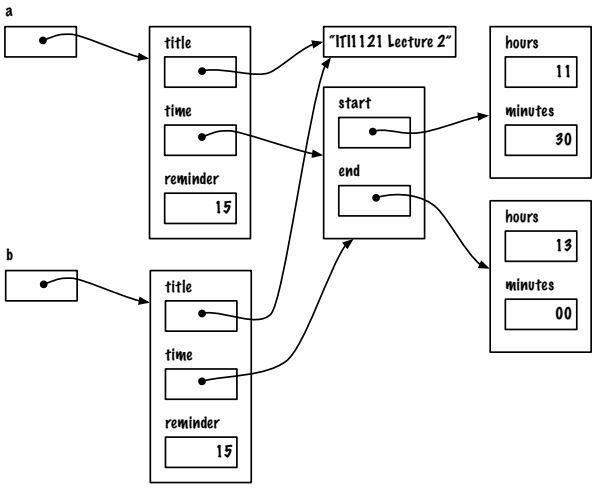


FIGURE 2 – Un exemple de copie de surface d'objets.

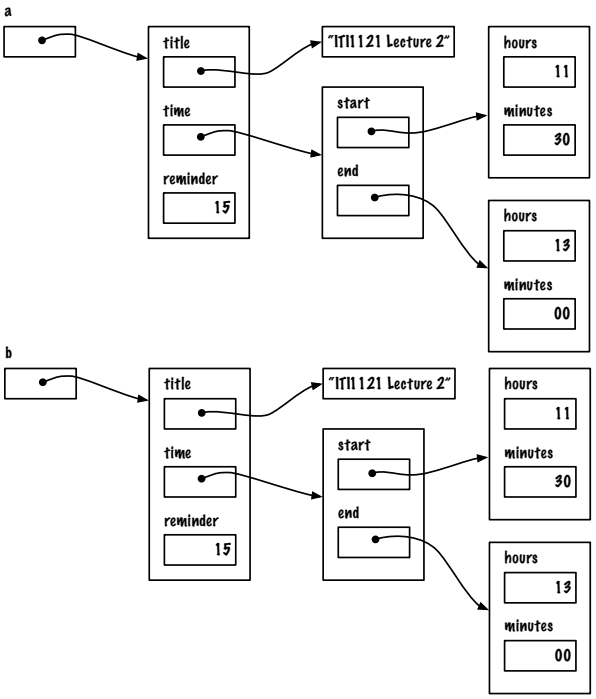


FIGURE 3 – Un exemple de copie profonde d'objets.

Vous pouvez en lire plus sur la différence entre copie de surface et copie profonde sur Wikipedia :

- Copie d'objets

Dernière modification : 9 février 2020