ITI 1521. Introduction à l'informatique II
Hiver 2020

Devoir 1
(Version du January 30, 2020)

Échéance: 2 février 2020, 23 h 30

Objectifs d'apprentissage

- Éditer, compiler et exécuter des programmes Java
- Utiliser des tableaux pour conserver les informations
- Appliquer les concepts de base de la programmation orientée objet
- UnderstandComprendre les politiques de l'université en matière d'intégrité académique

Introduction

Cette année, nous allons implémenter le jeu Tic-Tac-Toe. Le jeu en lui-même est assez simple et bien connu. Vous pouvez rafraîchir vos connaissances sur le Tic-Tac-Toe, par exemple ici : https://fr.wikipedia.org/wiki/Tic-tac-toe.

Notre but ultime est de programmer un algorithme d'apprentissage machine qui apprendra automatiquement à devenir un bon joueur de Tic-Tac-Toe. Nous baserons notre approche sur un article publié par Donald Michie en 1961 dans *Science Survey*, intitulé *Trial and error*. Cet article a été réimprimé dans le livre *On Machine Intelligence* et se trouve à la page 11 à l'URL suivante : (https://www.gwern.net/docs/ai/1986-michie-onmachineintelligence.pdf).

Pour une interprétation plus moderne de la même idée, vous pouvez aussi regarder https://www.youtube.com/watch?v=R9c-_neaxeU.

Mais pour ce devoir, notre objectif est plus modeste : nous voulons simplement mettre en place un jeu de Tic-Tac-Toe, où les joueurs indiquent leur prochain coup depuis la ligne de commande. Dans sa configuration par défaut, cela ressemblera à ceci : le programme affiche d'abord une grille vide et demande au premier joueur (*X*) une entrée.

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

X to play:
```

Le premier joueur a joué la cellule *5*. Le programme affiche la partie en cours, avec la cellule numéro cinq occupée par *X*, et demande au deuxième joueur (*O*) de faire une entrée. Le jeu continuera à suivre ce modèle.

```
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
 $
```

Comme on peut le voir, à chaque tour le programme imprime l'état actuel de la partie et demande ensuite au prochain utilisateur (*X* ou *O*) de lui fournir son prochain coup. Nous supposons simplement que les cellules sont numérotées ligne par ligne, du haut à gauche au bas à droite, comme suit :

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

Donc dans la partie ci-dessus, le premier coup du joueur (qui joue avec *X*) est de sélectionner la cellule 5, qui est au milieu de la grille. Le coup initial du second joueur (qui joue avec *O*) est de sélectionner la cellule 2, qui est au milieu de la première ligne (et une erreur fatale). Après quelques coups supplémentaires, le premier joueur gagne.

Notre propre implémentation sera un peu plus générale que le jeu habituel sur une grille 3×3. Par défaut (comme montré ci-dessus), notre jeu sera en effet joué sur une grille 3×3, en essayant d'aligner 3 cellules similaires horizontalement, verticalement ou en diagonale. Mais notre implémentation plus générale acceptera 3 paramètres en entrée n, m et k pour jouer un jeu sur une grille $n \times m$ en essayant d'aligner k des cellules similaires horizontalement, verticalement ou en diagonale. Voici un exemple de jeu sur une grille 3×4, essayant d'aligner 3 cellules similaires.

```
$ 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

Dans cette application, nous devons enregistrer l'«état» d'une partie : elle peut être encore en cours, ou l'un ou l'autre des joueurs a gagné, ou encore elle peut être nulle. De même, nous devons enregistrer l'état d'une cellule sur le plateau : une cellule peut être vide, ou elle peut contenir un *X* ou un *O*.

Il y a plusieurs façons d'y parvenir, mais dans ce devoir, nous allons utiliser le type *Enum* de Java.

Certains programmeurs utilisent des valeurs de type **int** pour représenter des constantes symboliques dans leurs programmes. Par exemple, pour représenter le jour de la semaine ou le mois de l'année.

```
public class E1 {

    public static final int MONDAY = 1;
    public static final int TUESDAY = 2;
    public static final int SUNDAY = 7;

    public static final int JANUARY = 1;
    public static final int FEBRUARY = 2;
    public static final int DECEMBER = 12;

    public static void main(String[] args) {

        int day = SUNDAY;

        switch (day) {
            case MONDAY:
            System.out.println("sleep");
            break;

            case SUNDAY:
            System.out.println("midterm test");
            break;

            default:
            System.out.println("study");
        }
    }
}
```

L'utilisation de constantes, telles que **MONDAY** et **JANUARY**, améliore la lisibilité du code source. Comparez "**if (jour == LUNDI) { ... }**" à "**if (jour == 1) { ... }**". C'est un pas dans la bonne direction.

Cependant, comme toutes les constantes sont des valeurs entières, il y a plusieurs types d'erreurs que le compilateur ne peut pas détecter. Par exemple, si le programmeur utilise le même nombre pour deux constantes, le compilateur ne pourra pas l'aider, 7 est une valeur valide pour **SATURDAY** et **SUNDAY** :

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

Mais aussi, assigner une valeur représentant un **mois** à une variable représentant un **jour** de la semaine ne serait pas détecté par le compilateur, les deux sont de type **int** :

```
int day = JANUARY;
```

Les types énumérés ont les mêmes avantages que les constantes symboliques ci-dessus, ce qui rend le code plus lisible, mais d'une manière sûre du point de vue des types.

```
public class E2 {

    public enum Day {
        MONDAY, TUESDAY, SUNDAY
    }

    public enum Month {
        JANUARY, FEBRUARY, DECEMBER
    }

    public static void main(String[] args) {

        Day day = Day.MONDAY;

        switch (day) {
            case MONDAY:
            System.out.println( "sleep" );
            break;

            case SUNDAY:
```

```
            System.out.println( "midterm test" );
            break;

            default:
            System.out.println( "study" );
        }
    }
}
```

Dans le programme ci-dessus, chaque constante a une valeur unique. De plus, l'énoncé ci-dessous provoque une erreur de compilation, comme il se doit :

```
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.

Notre implémentation

Nous sommes maintenant prêts à programmer notre solution. Nous n'aurons besoin que de quatre classes pour cela. Pour le devoir, vous devez suivre les modèles que nous vous fournissons. Vous ne pouvez modifier aucune des signatures des méthodes (c'est-à-dire que vous ne pouvez pas modifier les méthodes du tout). Vous ne pouvez pas ajouter de nouvelles méthodes ou variables de visibilité *public*. Vous pouvez, cependant, ajouter de nouvelles méthodes dont la visibilité est *private* pour améliorer la lisibilité ou l'organisation de votre code.

GameState

GameState est un type enuméré qui est utilisé pour décrire l'état actuel du jeu. Il a quatre valeurs possibles :

- *PLAYING*: ce jeu est en cours,
- *DRAW*: ce jeu est un match nul,
- *XWIN*: ce jeu a été gagné par le premier joueur,
- *OWIN*: ce jeu a été gagné par le premier joueur,

CellValue

CellValue est un type énuméré qui est utilisé pour décrire l'état d'une cellule. Il a trois valeurs possibles :

- *EMPTY*: la cellule est vide,
- *X*: il y a un **X** dans la cellule,
- *O*: il y a un **O** dans la cellule.

TicTacToeGame

Les instances de la classe *TicTacToeGame* représentent un jeu en cours. Chaque objet contient le plateau actuel, qui est sauvegardé dans un tableau à une dimension. Il y a une méthode d'instance qui peut être utilisée pour jouer le prochain coup. L'objet détermine le tour du joueur, de sorte que l'information n'est pas spécifiée : on spécifie simplement l'index à jouer et l'objet sait comment jouer soit un *X* soit un *O*. L'objet enregistre aussi automatiquement l'état de la partie.

La spécification de notre classe *TicTacToeGame* est donnée dans notre fichier zip. Vous devez remplir toutes les parties manquantes, en lisant attentivement tous les commentaires avant de le faire. Vous ne pouvez pas modifier les méthodes ou les variables qui sont fournies. Vous pouvez, cependant, ajouter de nouvelles méthodes de visibilité *private* si nécessaire.

Le modèle que vous utilisez contient les éléments suivants :

- Une variable d'instance qui pointe vers un tableau d'objets de type CellValue afin de représenter l'état de la grille.
- Quelques variables d'instance pour enregistrer le nombre de colonnes et de lignes du jeu, le nombre de cellules à aligner, le nombre de tours joués ("level") et l'état actuel.
- Trois constructeurs : celui par défaut crée le jeu habituel (la grille 3x3, avec un vainqueur si 3 cellules similaires sont alignées), un second peut être utilisé pour spécifier à la fois le nombre de lignes et le nombre de colonnes, et le dernier est utilisé pour spécifier le nombre de lignes, le nombre de colonnes et le nombre de cellules à aligner. Comme d'habitude, toutes les variables d'instance doivent être initialisées lors de la construction de l'objet.
- Méthodes d'accès («getters») pour le nombre de lignes, le nombre de colonnes, **sizeWin**, le niveau et l'état actuel du jeu
- Une méthode pour interroger l'objet sur le prochain joueur (c'est-à-dire, est-ce le tour de *X* ou de *O* de jouer ?).
- Une méthode **play(int index)** pour jouer à un endroit particulier dans le jeu. Ceci met à jour l'état du jeu et la grille.
- Nous avons aussi une méthode auxiliaire, **private void setGameState(int index)**, qui est utilisée pour calculer et mettre à jour l'état du jeu une fois qu'un coup particulier est joué dans la méthode **play**.
- Enfin, nous avons une méthode **toString()**, qui retourne une représentation sous forme de chaîne de caractères de l'état actuel du jeu, comme montré dans l'exemple précédent. Un exemple d'une chaîne de caractères retournée par toString serait, lors de l'impression :

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

Il y a quelques situations qui méritent notre attention. Par exemple, l'index sélectionné par le joueur peut être invalide ou illégal. Nous n'avons pas encore de très bons mécanismes pour gérer ces situations, donc pour le moment nous allons simplement écrire un message d'erreur. Le comportement ultérieur de la méthode n'est pas spécifié, donc il suffit d'implémenter quelque chose qui semble avoir un sens[1]. Une autre situation serait que les joueurs continuent le jeu après que l'un d'eux ait gagné. Pour les besoins des tests, nous voulons que cela soit possible, mais l'état du jeu doit alors refléter le premier gagnant de la partie. Ainsi, si les joueurs continuent après une victoire, un message est imprimé mais le jeu continue tant que les coups sont légaux. Le «premier» gagnant subsiste.

Notez que la méthode **toString()** retourne une référence vers une chaîne de caractères, elle n'imprime rien en réalité. Ainsi, une instance de String, lorsqu'elle est imprimée, devrait produire la sortie attendue (en d'autres termes, cette instance de String, lorsqu'elle est imprimée, occupera plusieurs lignes).

TicTacToe

Cette classe implémente le jeu. On vous fournit une partie initiale du code, qui crée l'instance de la classe TicTacToeGame en fonction des paramètres soumis par l'utilisateur. Tout ce que vous avez à faire ici est d'implémenter le reste de la méthode **main**, la partie qui joue le jeu. Il s'agit en fait de boucler sur chaque étape du jeu jusqu'à ce que le jeu soit terminé. A chaque étape, il affiche le jeu en cours et demande au joueur suivant, *X* ou a *O*, de jouer une cellule. Il joue alors cette cellule et continue jusqu'à la fin de la partie, où il termine (bien que nous ayons dit que

[1] La raison pour laquelle nous ne spécifions aucun comportement ici est qu'une fois que nous aurons les outils nécessaires pour traiter ces situations exceptionnelles, nous verrons que nous n'aurons pas du tout à trouver un comportement alternatif.

TicTacToeGame est capable de jouer au-delà du point gagnant, cette situation ne devrait jamais se produire à partir de la main de TicTacToe).

Si la cellule fournie par le joueur est invalide ou illégale, un message est affiché à l'utilisateur, qui est invité à jouer à nouveau. Voici quelques exemples de cette situation :

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

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:
```

Notez que vous pouvez supposer que les joueurs ne fournissent que des valeurs entières comme entrées. Vous n'avez pas à gérer le cas où d'autres types d'entrée sont fournis, comme par exemple un caractère.

JUnit

Nous fournissons un ensemble de tests JUnit pour la classe *TicTacToeGame*. Ces tests devraient bien sûr permettre de s'assurer que votre implémentation est correcte. Ils peuvent aussi aider à clarifier le comportement attendu de cette classe, si besoin est.

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.
- 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

Vous devez fournir un fichier **zip** (aucun autre format de fichier ne sera accepté). Le nom du répertoire principal doit avoir la forme suivante : **a1_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 1. 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 a1_3000000_3000001.zip contient les fichiers que vous devez utiliser comme point de départ. Votre soumission doit contenir les fichiers suivants.

- README.txt
    - 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).
- CellValue.java
- GameState.java
- TicTacToe.java
- TicTacToeGame.java
- StudentInfo.java

Questions

Pour toutes vos questions, veuillez consulter le site Web Piazza pour ce cours :

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

Version: January 30, 2020