Aix*Marseille université
Initiative d'excellence
Année 2020-21

Domaine Sciences et Technologies
Licence informatique

Programmation 2 : TP 3 (informatique)
Code UE : SIN3U02

Jeu de la vie

1 Introduction

Le jeu de la vie (`https://fr.wikipedia.org/wiki/Jeu_de_la_vie`) n'est pas vraiment un jeu, puisqu'il ne nécessite aucun joueur.

Le jeu se déroule sur une grille à deux dimensions dont les cases, que l'on appelle des cellules, peuvent prendre deux états distincts : vivant ou mort.

À chaque étape, l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines de la façon suivante :

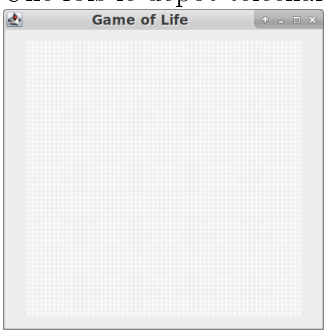
— Une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît).

— Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt.

Le but de ce TP est de compléter le code fourni dans le dépôt `https://etulab.univ-amu.fr/tp3` afin d'obtenir un simulateur de jeu de la vie.

2 Projet

Une fois le dépôt téléchargé, si vous exécutez le code fourni, vous devriez obtenir l'affichage suivant.



Malheureusement, l'application ne fonctionne pas correctement, l'état des cellules n'évolue pas.

Pour que l'application fonctionne, vous devez compléter le code de la classe `Grid`. Cette classe sert à modéliser la grille des cellules. Elle est composé d'une matrice de cellules (objets de type `Cell`). Chaque cellule possède la variable booléenne `isAlive` qui permet de savoir si elle est vivante ou morte.

Pour compléter la classe `Grid`, vous aurez besoin d'utiliser les trois méthodes suivantes de la classe `Cell` :

— `public boolean isAlive()` : retourne `true` si la cellule est vivante et `false` sinon.

— `public void setAlive()` : rend une cellule vivante.

— `public void setDead()` : tue une cellule.

La classe `Grid` contient déjà les méthodes suivantes (que vous n'avez pas à modifier) :

— `public Iterator<Cell> iterator()` : permet de parcourir les cellules d'une grille `grid` à l'aide d'une boucle `for(Cell cell : grid)`.

— `public Cell getCell(int rowIndex, int columnIndex)` : retourne la cellule à la position (`rowIndex`, `columnIndex`). Afin de vous simplifier le travail, la méthode `public Cell getCell(int rowIndex, int columnIndex)` autorise des indices en dehors des bornes habituelles (avec des valeurs négative ou supérieur au nombre de ligne/colonnes) en considérant que la grille est enroulée sur elle-même. Elle renvoie donc la cellule en position (0,0) si on lui donne en paramètre (`numberOfRows`,`numberOfColumns`) et la cellule en position (`numberOfRows-1`,`numberOfColumns-1`) si on lui donne en paramètre (-1,-1).

— `public int getNumberOfRows()` : retourne le nombre de lignes de la grille.

— `public getNumberOfColumns()` : retourne le nombre de colonnes de la grille.
— `private Cell[][] createCells()` : alloue un tableau bi-dimensionnel de cellules, de taille `numberOfRows` × `numberOfColumns` et le retourne.

Note : pensez à utiliser l'itérateur de cellules pour rendre votre code plus lisible.

Les méthodes que vous devez implémenter sont les suivantes

2.1 `getNeighbours`

La méthode `List<Cell> getNeighbours(int rowIndex, int columnIndex)` doit retourner la liste des 8 cellules voisines à la cellule (cellules partageant un coin ou un coté avec la cellule) à la position (`rowIndex`, `columnIndex`). Pour que toutes les cellules aient 8 voisines, on considère que les bords en haut et en bas sont voisins ainsi que les bord à droite et à gauche.

2.2 `countAliveNeighbours`

La méthode `int countAliveNeighbours(int rowIndex, int columnIndex)`. Elle doit retourner le nombre de cellules vivantes parmi les 8 voisines de la cellule. Vous devez appeler la méthode `getNeighbours` pour le code de cette méthode.

2.3 `calculateNextState`

La méthode `boolean calculateNextState(int rowIndex, int columnIndex)` renvoie `true` si la cellule en position (`rowIndex`, `columnIndex`) est vivante à la prochaine étape, et `false` sinon. On rapelle que cet état est déterminé par les règles suivantes :

— Une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît).
— Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt.

2.4 `calculateNextStates`

La méthode `boolean[][] calculateNextStates()` renvoie une matrice correspondant aux états que doivent prendre les cellules de la grille à la prochaine étape.

2.5 `goToNextState`

La méthode `void goToNextState(boolean[][] nextState)` met à jour tous les états des cellules de la grille en utilisant les valeurs de la matrice donnée en argument.

2.6 `nextGeneration`

La méthode `void nextGeneration()` fait avancer la grille d'une étape et donc met à jour tous les états des cellules de la grille en fonction des règles du jeu de la vie.

2.7 `clear`

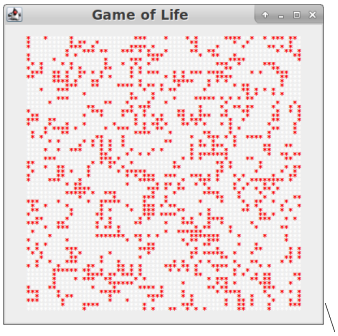
La méthode `void clear()` tue toutes les cellules de la grille.

2.8 `randomGeneration`

La méthode `void randomGeneration(Random random)` décide de manière aléatoire quelles cellules sont vivantes et quelles cellules sont mortes. On utilisera pour cela la méthode `nextBoolean` avec l'objet `random` passé en argument.

2.9 Test de vos méthodes

Si vous avez tout codé correctement, vous allez obtenir un affichage comme suivant :



3 Variante colorée

On souhaite maintenant avoir deux couleurs possibles pour une cellule vivante : `BLUE` ou `RED`. Une cellule `BLUE` devra être affiché en bleue alors qu'une cellule `RED` sera affichée en rouge.

Les règles déterminant la couleur d'une cellule sont les suivantes :

— Une cellule morte possédant exactement trois voisines vivantes devient vivante et naît avec la couleur majoritaire de ses voisines.

— Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt. Une cellule survivante garde sa couleur.

Pour la génération aléatoire, on gardera la probabilité d'avoir un cellule morte à 1/2. La probabilité d'avoir une cellule `RED` devra être de 1/4 et la probabilité d'avoir une cellule `BLUE` devra aussi être de 1/4.