

## 1 Jeu de la vie

## 2 Introduction

Le 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 qu'on appelle des cellules peuvent prendre deux états distincts : vivantes ou mortes.

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 par le dépôt afin d'obtenir un simulateur de jeu de la vie.

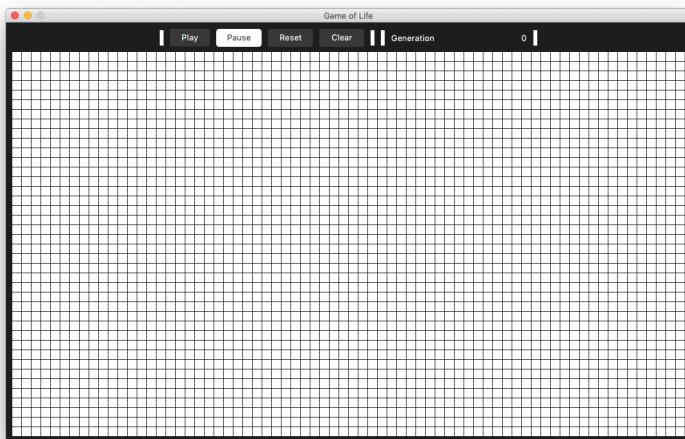
### 2.1 Consignes

Comme pour le TP précédent et celui d'avant, on va utiliser git pour la gestion de versions. Il vous faut donc vous reporter aux consignes du premier TP.

Lien vers la classroom du TP : [lien](#).

## 3 Projet

Une fois le dépôt téléchargé, vous pouvez compiler et exécuter le code cliquer deux fois sur `gameoflife-***` -> `application` -> `run` (en oubliant pas d'ajouter votre projet en tant que projet *Gradle*). Vous devriez obtenir l'affichage suivant.



Malheureusement, l'application ne fonctionne pas correctement. Vous pouvez changer l'état de chaque cellule en cliquant dessus (Une cellule morte est blanche alors qu'une cellule vivante est rouge). Par contre le lancement de la simulation via le bouton *play* ne crée aucun changement dans la grille.

Pour que l'application fonctionne, vous devez compléter le code de la classe `model.Grid`. Cette classe sert à modéliser la grille des cellules. Elle est composé d'une matrice de cellules (objet de type `model.Cell`). Chaque

cellule contient elle-même un état qui est représenté par une énumération de type `CellState` pouvant être égal à `CellState.DEAD` (état d'une cellule morte) ou `CellState.ALIVE` (état d'une cellule vivante).

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.
- `setState(CellState cellState)` : fixe l'état de la cellule.
- `public CellState getState()` : retourne l'état de la 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 supérieure 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 int getNumberOfColumns()` : retourne le nombre de colonnes de la grille.

### 3.1 Méthode `getNeighbours`

La première méthode que vous devez coder est `List<Cell> getNeighbours(int rowIndex, int columnIndex)` qui 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 et aussi que les bord à droite et à gauche sont aussi voisins.

### 3.2 Méthode `countAliveNeighbours`

La deuxième méthode que vous devez coder est `private 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.

### 3.3 Méthode `calculateNextState`

La troisième méthode que vous devez coder est `private CellState calculateNextState(int rowIndex, int columnIndex)` qui renvoie l'état que doit prendre la cellule en position (`rowIndex`, `columnIndex`) à la prochaine étape. On rappelle 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.

### 3.4 Méthode `calculateNextStates`

La quatrième méthode que vous devez coder est `private CellState[][] calculateNextStates()` qui renvoie une matrice correspondant aux états que doivent prendre les cellules de la grille à la prochaine étape.

### 3.5 Méthode `updateStates`

La cinquième méthode que vous devez coder est `private void updateStates(CellState[][] nextState)` qui met à jour tous les états des cellules de la grille en utilisant les valeurs de la matrice donnée en argument.

### 3.6 Méthode `updateToNextGeneration`

La sixième méthode que vous devez coder est `void updateToNextGeneration()` qui 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.

### 3.7 Méthode `clear`

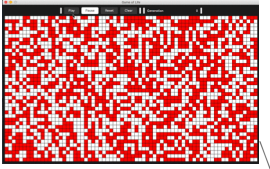
La septième méthode que vous devez coder est `void clear()` qui met à jour tous les états des cellules de la grille à `CellState.DEAD`.

### 3.8 Méthode `randomGeneration`

La huitième méthode que vous devez coder est `void randomGeneration(Random random)` qui met à jour tous les états des cellules de la grille de manière aléatoire à `CellState.DEAD` ou `CellState.ALIVE`. On utilisera pour cela la méthode `nextBoolean` avec l'objet `random` passé en argument.

### 3.9 Test de vos méthodes

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



## 4 Variante colorée

On souhaite maintenant avoir deux états possibles pour une cellule vivante : `BLUE` ou `RED` en plus de l'état `DEAD` pour une cellule morte. Une cellule `BLUE` devra être affiché en bleue alors qu'une cellule `RED`

- 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  $1/4$ .

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

