Cours MOOC EPFL d'initiation à la programmation illustré en Java

Cinquième devoir noté
Fonctions/Méthodes

J. Sam & J.-C. Chappelier

du 15 octobre au 9 novembre 2015

1 Exercice 1 — Conjecture de Legendre

1.1 Introduction

Le but de cet exercice est d'écrire un programme pour vérifier une propriété des nombres entiers, la « conjecture de Legendre ».

Télécharger le programme fourni sur le site du cours [1] et le compléter.

ATTENTION : vous ne devez modifier ni le début ni la fin du programme, juste ajouter vos propres lignes à l'endroit indiqué. Il est donc primordial de respecter la procédure suivante (les points 1 et 3 concernent spécifiquement les utilisateurs d'Eclipse) :

1. désactiver le formatage automatique dans Eclipse :
   `Window > Preferences > Java > Editor > Save Actions`
   (et décocher l'option de reformatage si elle est cochée)
2. sauvegarder le fichier téléchargé sous le nom `Legendre.java` (avec une majuscule, notamment). Si vous travaillez avec Eclipse vous ferez cette sauvegarde à l'emplacement `[dossierDuProjetPourCetExercice]/src/` ;
3. rafraîchir le projet Eclipse où est stocké le fichier (clic droit sur le projet > refresh) pour qu'il le prenne en compte ;
4. écrire le code à fournir entre ces deux commentaires :

1. `https://d396qusza40orc.cloudfront.net/initprogjava/assignements-data/Legendre.java`

```
/*****************************************
 * Completez le programme a partir d'ici.
 *****************************************/

/*****************************************
 * Ne rien modifier apres cette ligne.
 *****************************************/
```

5. sauvegarder et tester son programme pour être sûr(e) qu'il fonctionne correctement, par exemple avec les valeurs données plus bas ;
6. rendre le fichier modifié (toujours `Legendre.java`) dans « OUTPUT submission » (et non pas dans « Additional ! »).

1.2 Code à produire

Les mathématiciens supposent [2] que pour tout nombre entier n, il existe au moins un nombre premier compris entre n^2 et $(n+1)^2$. C'est ce que nous voulons vérifier à l'aide d'un programme.

Nous allons pour cela utiliser plusieurs sous-tâches [3] :

— savoir si un nombre donné est un nombre premier ;
— tester si *un* nombre entier donné n vérifie la propriété ci-dessus ;
— tester tous les nombres entiers compris entre deux bornes.

Pour la première sous-tâche, nous vous demandons d'écrire une fonction `estPremier` prenant en paramètre un `int` et retournant un booléen, « vrai » si le nombre fourni en paramètre est un nombre premier et « faux » sinon.

Pour tester si un nombre `x` est premier, on peut procéder comme suit :

— tout nombre inférieur ou égal à 1 n'est pas premier ;
— 2 est premier ;
— tout *autre* nombre pair n'est pas premier ;
— pour tout autre nombre (forcément impair donc), on recherche un diviseur : faire une boucle de 3 à la racine carrée de ce nombre `x` [4] ; si le reste de la division entière de `x` par le nombre de la boucle est nul, alors `x` **n'**est **pas** premier ;

2. Mais cela n'a pas encore été démontré à ce jour ; c'est ce qu'on appelle la « conjecture de Legendre », d'après le mathématicien français du 18e–19e siècle Adrien-Marie Legendre.

3. Il existe bien sûr plusieurs autres solutions (d'autres *algorithmes*), comme par exemple stocker en mémoire les résultats déjà acquis. Nous avons ici pris le parti de ne rien stocker en mémoire, mais de refaire les calculs à chaque fois. Nous vous demandons de respecter cette marche à suivre.

4. La racine carrée de `x` s'écrit « `Math.sqrt(x)` » en Java.

si à la fin de la boucle, tous les restes étaient non nuls, alors `x` est premier ; pour rappel, le reste de la division entière de `x` par `y` s'écrit « `x % y` » en Java.

Afin de tester si votre procédure est correcte, nous vous demandons de plus d'écrire une fonction `testPremiers` prenant deux entiers en paramètres et testant chaque entier compris entre ces deux bornes pour voir s'il est premier. Cette fonction devra écrire à l'écran la liste des nombres premiers trouvés en respectant strictement le format donné dans les exemples de déroulement fournis plus loin.

Pour tester si *un* nombre entier donné n vérifie la propriété de la conjecture de Legendre, nous vous demandons d'écrire une fonction `legendre` qui prend en paramètre un `int n` et retourne un `int`. Cette fonction testera tous les nombres compris entre `n*n+1` et `(n+1)*(n+1)-1` et s'arrêtera au *premier* nombre premier rencontré, qu'elle retournera. Si, par contre, aucun nombre premier n'a été trouvé entre ces deux bornes, la fonction retournera 0.

Enfin, la dernière fonction à fournir s'appelle `testLegendre`. Elle ne prend aucun paramètre et ne retourne rien. Elle commence par demander à l'utilisateur d'entrer un nombre en respectant strictement le format donné dans les exemples de déroulement fournis plus loin. Tant que le nombre entré par l'utilisateur est inférieur ou égal à 0, la fonction redemande un nombre. Une fois ce premier nombre entré, elle demande un second nombre, qui doit être supérieur ou égal au premier nombre entré (voir exemples de déroulement plus bas).

Une fois que deux valeurs valides ont été fournies, la fonction effectue enfin le test de la conjecture de Legendre sur tous les nombres compris entre les deux nombres saisis. Si la conjecture est vérifiée, cette procédure affiche le nombre testé suivi de « **:** » suivi du premier nombre premier trouvé (voir exemples de déroulement plus bas). Sinon, si jamais le nombre testé ne vérifie pas la conjecture de Legendre, alors le programme doit afficher « `PAS TROUVÉ !` »[5].

Exemples de déroulement

1) Exemple de test entre 4 et 9 (Note : le `main()` fourni commence par appeler `testPremiers` et donc affiche les nombres premiers entre 0 et 100) :

```
Premiers entre 0 et 100 :
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,

Tester la conjecture de Legendre entre : 4
et : 9
4 : 17
5 : 29
```

5. Et à vous la célébrité !

```
6 : 37
7 : 53
8 : 67
9 : 83
```

2) Exemple de test entre 1000 et 1010 :

```
Premiers entre 0 et 100 :
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,

Tester la conjecture de Legendre entre : 1000
et : 1010
1000 : 1000003
1001 : 1002017
1002 : 1004027
1003 : 1006021
1004 : 1008017
1005 : 1010033
1006 : 1012043
1007 : 1014061
1008 : 1016069
1009 : 1018091
1010 : 1020101
```

3) Exemple d'entrées incorrectes :

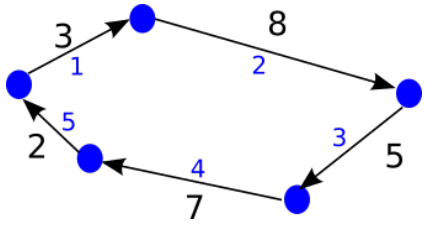
```
Premiers entre 0 et 100 :
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97,

Tester la conjecture de Legendre entre : -1
Tester la conjecture de Legendre entre : 0
Tester la conjecture de Legendre entre : 1
et : 0
et : 1
1 : 2
```

2 Exercice 2 — Courses pédestres

2.1 Introduction

Un comité d'organisation de courses pédestres souhaite que vous l'aidiez à sélectionner des parcours.

Dans votre programme, un parcours tel que (en bleu le numéro de l'étape et en noir son kilométrage) :



sera représenté par le tableau : $\{3, 8, 5, 7, 2\}$

Les valeurs du tableau sont les distances séparant les étapes :

- la distance de l'étape 1 est de 3 km ;
- celle de l'étape 2 est de 8 km ;
- etc.

Notez que les indices sont numérotés depuis 0 (numérotation classique des indices de tableaux en Java). Les étapes le sont depuis 1 (l'étape 1 a pour indice 0 dans le tableau par exemple).

On suppose ici que tous les parcours sont des circuits qui seront parcourus en boucle (parfois plusieurs fois) ; l'étape suivant la dernière étape est donc la première. Télécharger le programme fourni sur le site du cours [6] et le compléter.

ATTENTION : vous ne devez modifier ni le début ni la fin du programme, juste ajouter vos propres lignes à l'endroit indiqué. Il est donc primordial de respecter la procédure suivante (les points 1 et 3 concernent spécifiquement les utilisateurs d'Eclipse) :

1. désactiver le formatage automatique dans Eclipse :
   `Window > Preferences > Java > Editor > Save Actions`
   (et décocher l'option de reformatage si elle est cochée)
2. sauvegarder le fichier téléchargé sous le nom `Running.java` (avec une majuscule, notamment). Si vous travaillez avec Eclipse vous ferez cette sauvegarde à l'emplacement `[dossierDuProjetPourCetExercice]/src/` ;
3. rafraîchir le projet Eclipse où est stocké le fichier (clic droit sur le projet > refresh) pour qu'il le prenne en compte ;
4. écrire le code à fournir entre ces deux commentaires :

```
/*****************************************
 * Completez le programme a partir d'ici.
 *****************************************/

/*****************************************
```

6. `https://d396qusza40orc.cloudfront.net/initprogjava/assignements-data/Running.java`

```
 * Ne rien modifier apres cette ligne.
 ****************************************/
```

5. sauvegarder et tester son programme pour être sûr(e) qu'il fonctionne correctement, par exemple avec les valeurs données plus bas ;
6. rendre le fichier modifié (toujours `Running.java`) dans « OUTPUT submission » (et non pas dans « Additional ! »).

Il vous est demandé de compléter le programme fourni au moyen d'un certain nombre de méthodes décrites ci-dessous.

Toutes les méthodes que vous définirez dans votre programme devront être `public` et `static`. Il n'est pas permis de déclarer en dehors des méthodes des variables qui ne seraient pas `final` et `static` (seules les constantes sont autorisées).

Le programme fourni comporte déjà quelques lignes d'affichage utiles, que vous devez utiliser telles qu'elles.

2.2 Quelques méthodes de base

Dans le fichier fourni, commencez par définir une méthode :

```
int totalLength(int[] circuit)
```

qui calcule et retourne la longueur d'un circuit passé en paramètre comme étant la somme de la longueur de ses étapes.

Sur l'exemple du début, cette méthode retournerait donc 25 ($= 3 + 8 + 5 + 7 + 2$).

Programmez ensuite une méthode :

```
int stepMaxDiff(int[] circuit)
```

qui prend en argument un circuit et retourne l'indice de l'étape ayant le plus grand écart de distance (en valeur absolue) avec la suivante. La méthode `Math.abs` peut-être utilisée.

S'il y a plusieurs étapes candidates la méthode retournera celle d'indice le plus élevé.

Par exemple pour le circuit $\{7, 8, 5, 7, 2\}$ cette étape est l'étape d'indice 4. En effet :

— l'écart de distance entre l'étape 1 et 2 est de 1 ($= 8 - 7$) ;
— l'écart de distance entre l'étape 2 et 3 est de 3 ($= 8 - 5$) ;

— l'écart de distance entre l'étape 3 et 4 est de 2 ($= 7 - 5$) ;
— l'écart de distance entre l'étape 4 et 5 est de 5 ($= 7 - 2$) ;
— l'écart de distance entre l'étape 5 et 1 est de 5 ($= 7 - 2$).

Il y a deux étapes qui ont le plus grand écart de distance avec la suivante, la 4 et la 5. On retient celle d'indice le plus élevé, étape 5 (d'indice 4) ici.

Si le circuit est vide la méthode doit retourner -1.

Voir l'exemple de déroulement plus bas pour des valeurs de test.

2.3 Affichage et sélection de circuits

Codez la méthode :
`void displayAllCircuits(int[][] circuits)`

prenant en argument un ensemble de circuits et les affichant en respectant strictement le format des exemples d'exécution donnés plus bas.

Si l'ensemble des circuits est vide, le message suivant sera affiché :
`Liste de circuits vide !`

Codez finalement la méthode :

`void selectCircuit(int[][] circuits, int minTotalDist)`

qui prend en argument un ensemble de circuits.

Cette méthode affiche le circuit qui parmi tout ceux de longueur au moins égale à `minTotalDist` a l'étape la plus longue la plus proche (en terme d'indice) de la dernière étape. On entend par plus longue étape, celle ayant le plus grand écart de distance avec sa suivante.

S'il y a plusieurs circuits candidats, le dernier sera retenu. Par exemple, dans la situation suivante et pour un `minTotalDist` de 48km :

```
Circuit no 1 (21 km) : 1 6 14 (étape ayant le plus grand écart avec la suivante = 3) -> trop court
Circuit no 2 (75 km) : 3 4 15 4 19 3 11 1 12 3 (étape ayant le plus grand écart avec la suivante = 5)
 -> difference  avec  la dernière étape = 5
Circuit no 3 (21 km) : 1 10 1 1 8 (étape ayant le plus grand écart avec la suivante = 2) -> trop court
Circuit no 4 (32 km) : 1 11 18 2 (étape ayant le plus grand écart avec la suivante = 3) -> trop court
Circuit no 5 (107 km) : 15 12 14 15 4 6 6 17 10 8 (étape ayant le plus grand écart avec la suivante = 7
-> difference avec la dernière étape = 3
Circuit no 6 (104 km) : 16 6 10 14 14 17 10 17 (étape ayant le plus grand écart avec la suivante = 1)
-> différence avec la dernière étape = 6
Circuit no 7 (86 km) : 8 8 13 6 18 14 13 6 (étape ayant le plus grand écart avec la suivante = 4)
-> différence avec la dernière étape = 4
Circuit no 8 (15 km) : 2 3 10 (étape ayant le plus grand écart avec la suivante = 3) -> trop court
```

Le circuit affiché sera le 5.

Les affichages produits doivent être *strictement conformes* aux modèles donnés dans les exemples d'exécution (voir plus bas).

En particulier, si `minTotalDist` a une valeur négative ou nulle, le programme affichera le message :
`votre circuit doit avoir au moins 1 km !`
suivi d'un saut de ligne.

Si aucun circuit ne répondant aux critères souhaités n'a pu être trouvé, le message suivant s'affichera :
`aucun circuit possible avec ces critères !`
suivi d'un saut de ligne.

Exemple de déroulement Cet exemple correspond au programme principal fourni :

```
Circuit no 1 (28 km) : 3 1 2 7 2 5 6 2 (étape ayant le plus grand écart avec la suivante = 4)
Circuit no 2 (57 km) : 2 10 20 12 13 (étape ayant le plus grand écart avec la suivante = 5)
Circuit no 3 (59 km) : 3 7 12 15 18 4 (étape ayant le plus grand écart avec la suivante = 5)
Circuit no 4 (59 km) : 2 11 21 12 13 (étape ayant le plus grand écart avec la suivante = 5)
Sélection d'un circuit de 22 km au minimum : le circuit choisi est le 4

Liste de circuits vide !
Sélection d'un circuit de 10 km au minimum : aucun circuit possible avec ces critères !

Circuit no 1 (25 km) : 6 1 2 2 14 (étape ayant le plus grand écart avec la suivante = 4)
Circuit no 2 (22 km) : 3 1 2 7 9 (étape ayant le plus grand écart avec la suivante = 5)
Sélection d'un circuit de 20 km au minimum : le circuit choisi est le 2
Sélection d'un circuit de -2 km au minimum : votre circuit doit avoir au moins 1 km !
Sélection d'un circuit de 25 km au minimum : le circuit choisi est le 1
```

3 Exercice 3 — Proies et prédateurs

3.1 Introduction

Plus élaboré que le modèle de croissance exponentielle présenté dans le devoir précédent, le modèle prédateurs–proies (aussi appelé « modèle Lotka-Volterra ») permet de simuler la croissance de deux espèces en relation de proies et prédateurs, comme par exemple des lapins et des renards.

Dans le fichier fourni, est défini un programme principal qui a besoin de plusieurs méthodes pour fonctionner. Il vous est demandé de les compléter conformément aux instructions qui suivent.

Ce programme permettra de simuler la co-évolution de populations de lapins et de renards au cours du temps.

Toute méthode programmée dans cet exercice devra obligatoirement être `public` et `static`. Il n'est pas permis de déclarer en dehors des méthodes des variables qui ne seraient pas `final` et `static` (seules les constantes sont autorisées).

Télécharger le programme fourni sur le site du cours [7] et le compléter.

ATTENTION : vous ne devez modifier ni le début ni la fin du programme, juste ajouter vos propres lignes à l'endroit indiqué. Il est donc primordial de respecter la procédure suivante (les points 1 et 3 concernent spécifiquement les utilisateurs d'Eclipse) :

1. désactiver le formatage automatique dans Eclipse :
   `Window > Preferences > Java > Editor > Save Actions`
   (et décocher l'option de reformatage si elle est cochée)
2. sauvegarder le fichier téléchargé sous le nom `PredProie.java` (avec une majuscule, notamment). Si vous travaillez avec Eclipse vous ferez cette sauvegarde à l'emplacement `[dossierDuProjetPourCetExercice]/src/` ;
3. rafraîchir le projet Eclipse où est stocké le fichier (clic droit sur le projet > refresh) pour qu'il le prenne en compte ;
4. écrire le code à fournir entre ces deux commentaires :

```
/*****************************************
 * Completez le programme a partir d'ici.
 *****************************************/

/*****************************************
 * Ne rien modifier apres cette ligne.
 *****************************************/
```

5. sauvegarder et tester son programme pour être sûr(e) qu'il fonctionne correctement, par exemple avec les valeurs données plus bas ;
6. rendre le fichier modifié (toujours `PredProie.java`) dans « OUTPUT submission » (et non pas dans « Additional ! »).

3.2 Saisie des populations initiales

Commencez par compléter la méthode `entrerPopulation` de sorte à ce qu'elle demande à l'utilisateur d'introduire le nombre initial d'animaux (`animal`), supérieur ou égal à `nombreMin` ; ce nombre sera redemandé à l'utilisateur tant qu'il n'est pas supérieur ou égal à `nombreMin` .

7. `https://d396qusza40orc.cloudfront.net/initprogjava/assignements-data/PredProie.java`

Pour faciliter la correction, nous avons déjà écrit le texte de la question à poser à l'utilisateur. Veuillez ne pas le modifier mais simplement l'utiliser.

La méthode retournera le nombre saisi par l'utilisateur lorsqu'il est saisi correctement.

Les lectures de données devront se faire au moyen de l'argument de type `Scanner`.

Cette méthode est appelée au début du programme principal fourni.

3.3 Evolution des populations

On souhaite ensuite simuler l'évolution des populations de lapins et de renards pendant `DUREE` mois (constante fournie dans le code) et en incrémentant à chaque fois le mois d'une unité.

Complétez pour cela les méthodes `calculerLapins` et `calculerRenards` qui serviront à mettre à jour les populations de ces animaux chaque mois.

La méthode `calculerLapins` calcule et retourne la croissance des lapins, c'est à dire leur nouveau nombre après l'écoulement d'un mois. Le calcul se fait en multipliant le nombre courant de lapins par la formule suivante :
`(1.0 + TAUX_CROISSANCE_LAPINS - tauxAttaque * nbRenards )`
où `nbRenards` est le nombre de renards du mois *précédent* le calcul.

Cela représente le fait que le nouveau nombre de lapins équivaut au nombre de lapins qu'il y avait précédemment plus le nombre de lapins qui sont nés (`TAUX_CROISSANCE_LAPINS`) moins le nombre de lapins tués par les renards. Le nombre de lapins tués dépend du nombre de renards puisque plus il y a de renards, plus ils mangent de lapins.

De son coté, la croissance des renards, retournée par la méthode `calculerRenards`, est calculée en multipliant le nombre courant de renards par :

```
(1.0 + tauxAttaque * nbLapins * TAUX_CROISSANCE_RENARDS
     - TAUX_MORTALITE)
```

où `nbLapins` est le nombre de lapins du mois *précédent*.

Cela représente le fait que le nouveau nombre de renards augmente du nombre de renards nés moins le nombre de renards morts. On voit que dans le cas des renards, c'est le nombre de naissances qui dépend du nombre de lapins, car les renards doivent attraper des lapins afin de pouvoir se nourrir, se reproduire et nourrir leurs petits.

Les formules ci-dessus n'empêchent pas les populations de lapins et de renards d'être négatives. Il vous faut donc faire en sorte que la valeur retournée soit plafonnée à 0 si la formule calcule un nombre négatif.

Vous écrirez et utiliserez pour cela une méthode
`double plafonner(double value)`
qui retourne 0 si `value` est négative et `value` sinon.

Note : Nous avons représenté le nombre de lapins et de renards retournés par les méthodes `calculerRenards` et `calculerLapins` par des `double` afin de garder de la précision sur le calcul. On pourrait interpréter cela comme une incertitude sur le nombre exact d'animaux existants.

3.3.1 Simulation

Complétez enfin la méthode `simule` qui fait évoluer les populations de lapins et de renards sur `DUREE` mois en mettant à jour, à chaque itération, les populations de renards et de lapins au moyen des méthodes `calculerRenards` et `calculerLapins`.

La méthode `simule` devra :

- Arrêter d'itérer si les lapins et les renards ont disparus. On considère que les lapins/renards ont disparu si leur nombre devient inférieur à `2`. Si la population de renards ou de lapins passe en dessous de 2, nous supposons que l'espèce ne peut plus se reproduire. Dans ce cas de figure, le nombre de renards ou de lapins doit alors être mis à zéro.
- Afficher après la fin des itérations le nombre de mois écoulés et le nombre de lapins et de renards à ce moment. Cet affichage se fera au moyen de la méthode fournie `afficheStatus` (voir l'exemple de déroulement plus bas).
- Répéter ces traitements pour tous les taux d'attaque à partir de `tauxInit`, en incrémentant le taux de 1% (`0.01`) à chaque fois, tant que le taux est inférieur ou égal à `tauxFin`. La comparaison se fera au moyen de l'opérateur <= et strictement par rapport à `tauxFin`. En raison de l'imprécision liée à la représentation des doubles, il est possible que la borne tauxFin ne soit pas atteinte. Ceci est considéré comme un comportement normal du programme.
  Pour chaque itération de cette boucle, il faudra afficher, en début d'itération, le taux d'attaque utilisé au moyen de la méthode fournie `afficheTauxDAttaque`.
  A chaque itération, la simulation démarre des populations initiale fournies en argument de `simule` (voir l'exemple de déroulement plus bas).

De plus, la méthode `simule` devra mémoriser dans des booléens si au cours de la simulation pour un taux donné :

- les renards et/ou les lapins ont été en voie d'extinction (leur nombre descend en dessous de 5 strictement) ;

— leur population a réussi a remonter après avoir été menacée d'extinction ;
— les renards et/ou lapins ont disparu.

Ces booléens seront utilisés pour afficher des messages d'alertes après l'affichage fait par `afficheStatus` (voir l'exemple de déroulement ci-dessous).

Si la population d'un animal passe brutalement d'un nombre supérieur ou égal à 5 à zéro dans un cycle on considérera que la population a été en voie d'extinction et a disparu.

Pour simplifier nous ne tiendrons pas compte des oscillations entre les différents états : il suffit qu'ils se soient produits une seule fois pour mettre les booléens concernés à true. Par exemple si la population a disparu après avoir pu remonter (après une menace d'extinction), le message sur la remontée de la population s'affichera quand même.

Affichage des messages d'alerte Ces derniers seront faits par la méthode `afficheResultat`, à compléter, qui affichera :

— `Les ... ont été en voie d'extinction`
(avec `...` valant soit `renards` soit `lapins`), si les animaux ont été menacés d'extinction
— `et les ... ont disparus :-(`
si les animaux ont disparus
— `mais la population est remontée ! Ouf !`
si la population a observé une remontée en cours de simulation.

Si durant la simulation (pour un taux d'attaque donné), **aucun** des événements décrits ci-dessous n'est arrivé, la méthode `simule` affichera :

```
Les lapins et les renards ont des populations stables.
```

pour indiquer que pour la simulation avec un taux d'attaque donné, les populations de lapins et de renards ont réussi à demeurer stables

Indication : le booléen sur la remontée de la population ne sera exploité que si celui sur la menace d'extinction est vrai. Celui sur la disparition dans tous les cas.

Le programme principal fourni permet de tester la méthode `simule` pour une seul taux d'attaque puis pour des taux d'attaque initiaux et finaux saisis par l'utilisateur.

Exemple de déroulement (20 renards et 500 lapins au départ)

```
Combien de renards au départ (>=2.0) ? 20
Combien de lapins au départ (>=5.0) ? 500

***** Le taux d'attaque vaut 1.00%
Après 50 mois, il y a 950.302 lapins et 7.648 renards
Les lapins et les renards ont des populations stables.

taux d'attaque au départ en % (entre 0.5 et 6) ? 2
taux d'attaque à la fin  en % (entre 2.0 et 6) ? 3

***** Le taux d'attaque vaut 2.00%
Après 50 mois, il y a 204.956 lapins et 10.101 renards
Les lapins et les renards ont des populations stables.

***** Le taux d'attaque vaut 3.00%
Après 50 mois, il y a 0.000 lapins et 16.898 renards
Les renards ont été en voie d'extinction
mais la population est remontée ! Ouf !
Les lapins ont été en voie d'extinction
et les lapins ont disparus :-(
```

Note : Si vous voulez observer une évolution intéressante des populations, le nombre de lapins doit être bien supérieur à celui des renards, sinon les renards tuent tous les lapins avant qu'ils aient eu le temps de suffisamment se reproduire. De plus, évitez d'avoir des populations trop petites au début. Dix renards et 1000 lapins ou 20 renards et 500 lapins sont des exemples intéressants et stable de dynamiques des populations.

Notez également, que si vous voulez vérifier que vos messages d'erreurs sont cohérents, vous pouvez ajouter un appel à `afficheStatus` à votre programme de sorte à ce qu'elle affiche l'évolution chaque mois. Par exemple, si l'on fait cet ajout et que l'on teste avec le jeu de données précédent, on devrait obtenir l'affichage suivant pour le taux 3% :

```
***** Le taux d'attaque vaut 3.00%
Après 0 mois, il y a 500.000 lapins et 20.000 renards
Après 1 mois, il y a 350.000 lapins et 20.400 renards
Après 2 mois, il y a 240.800 lapins et 20.074 renards
Après 3 mois, il y a 168.028 lapins et 19.226 renards
Après 4 mois, il y a 121.520 lapins et 18.079 renards
Après 5 mois, il y a 92.067 lapins et 16.798 renards
Après 6 mois, il y a 73.290 lapins et 15.490 renards
Après 7 mois, il y a 61.219 lapins et 14.213 renards
```

```
Après 8 mois, il y a 53.481 lapins et 13.001 renards
Après 9 mois, il y a 48.667 lapins et 11.868 renards
Après 10 mois, il y a 45.940 lapins et 10.819 renards
Après 11 mois, il y a 44.811 lapins et 9.857 renards
Après 12 mois, il y a 45.004 lapins et 8.977 renards
Après 13 mois, il y a 46.385 lapins et 8.176 renards
Après 14 mois, il y a 48.922 lapins et 7.450 renards
Après 15 mois, il y a 52.665 lapins et 6.792 renards
Après 16 mois, il y a 57.734 lapins et 6.199 renards
Après 17 mois, il y a 64.317 lapins et 5.665 renards
Après 18 mois, il y a 72.682 lapins et 5.186 renards
Après 19 mois, il y a 83.179 lapins et 4.758 renards
Après 20 mois, il y a 96.261 lapins et 4.377 renards
Après 21 mois, il y a 112.499 lapins et 4.040 renards
Après 22 mois, il y a 132.613 lapins et 3.745 renards
Après 23 mois, il y a 157.496 lapins et 3.490 renards
Après 24 mois, il y a 188.255 lapins et 3.273 renards
Après 25 mois, il y a 226.247 lapins et 3.094 renards
Après 26 mois, il y a 273.124 lapins et 2.952 renards
Après 27 mois, il y a 330.872 lapins et 2.850 renards
Après 28 mois, il y a 401.839 lapins et 2.792 renards
Après 29 mois, il y a 488.736 lapins et 2.782 renards
Après 30 mois, il y a 594.569 lapins et 2.830 renards
Après 31 mois, il y a 722.462 lapins et 2.951 renards
Après 32 mois, il y a 875.245 lapins et 3.167 renards
Après 33 mois, il y a 1054.653 lapins et 3.516 renards
Après 34 mois, il y a 1259.806 lapins et 4.054 renards
Après 35 mois, il y a 1484.519 lapins et 4.875 renards
Après 36 mois, il y a 1712.779 lapins et 6.124 renards
Après 37 mois, il y a 1911.941 lapins et 8.029 renards
Après 38 mois, il y a 2024.997 lapins et 10.910 renards
Après 39 mois, il y a 1969.698 lapins et 15.122 renards
Après 40 mois, il y a 1667.056 lapins et 20.758 renards
Après 41 mois, il y a 1129.037 lapins et 26.987 renards
Après 42 mois, il y a 553.662 lapins et 31.601 renards
Après 43 mois, il y a 194.870 lapins et 32.640 renards
Après 44 mois, il y a 62.513 lapins et 30.903 renards
Après 45 mois, il y a 23.312 lapins et 28.276 renards
Après 46 mois, il y a 10.531 lapins et 25.607 renards
Après 47 mois, il y a 5.600 lapins et 23.111 renards
Après 48 mois, il y a 3.398 lapins et 20.831 renards
Après 49 mois, il y a 2.294 lapins et 18.765 renards
Après 50 mois, il y a 0.000 lapins et 16.898 renards
```

```
Les renards ont été en voie d'extinction
mais la population est remontée ! Ouf !
Les lapins ont été en voie d'extinction
et les lapins ont disparus :-(
```

N'oubliez pas cependant de supprimer cet affichage avant de soumettre.