Introduction Scala : seconde partie du cavalier d'Euler.

Notions Scala : List, Tuple, recursivité, heuristique. Modèle MVC.

Vérifiez les conventions à suivre pour l'écriture du code (cf Moodle).

Vous allez réaliser ce TD en Scala.

Avertissement : pour *commencer* à programmer en Scala, vous devez être à jour en développement Java 5 (généricité) a minima + être à l'aise avec les notions de classe abstraite, d'héritage, de polymorphisme, de généricité contrainte. Tout le TD peut tenir en un seul fichier : vous pouvez définir plusieurs classes et objets dans un même fichier scala.

Vous indiquerez EXPLICITEMENT les types de retours des méthodes et des fonctions dans les déclarations systématiquement.

1. Etudiez dans Scaladoc, la classe et l'objet **Array**. En particulier, regardez la méthode **fill** qui permet de créer et d'initialiser un tableau rapidement. Vous utiliserez désormais cette méthode plutôt que **Array.ofDim et un initialiseur**.

   Fabriquez une classe **CavalierEuler** qui servira à la résolution du problème d'Euler.

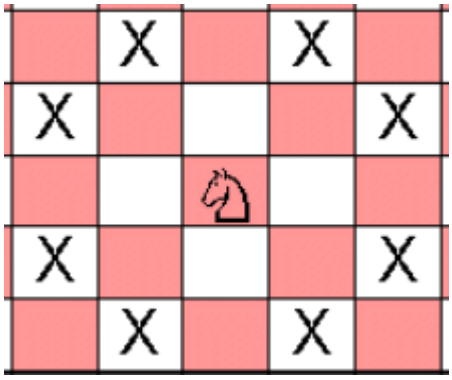
   On appelle modèle **MVC**, une manière de programmer qui fait intervenir un **M**odèle (les données du problème ou de la base de données), une **V**ue (l'affichage du modèle ou sa représentation graphique) et un **C**ontrôleur (ce qui est chargé de modifier le modèle ou de réagir à l'interface).

   Ici, vous utiliserez une instance d'Echiquier comme vue (nommez-le **vue**), et un tableau d'entiers comme Modèle (nommez-le ***modele***). En théorie, on pourrait avoir également un objet ***Controleur*** (qui a le droit de modifier le modèle) mais Scala nous permet de faire l'économie d'une classe supplémentaire : une méthode, ou une fonction peut contenir d'autres fonctions (ou fonctions locales à une fonction). Ici, toutes les fonctions nécessaires à la résolution du problème se trouveront DANS la méthode **controleur** (puisque c'est le seul à les utiliser).

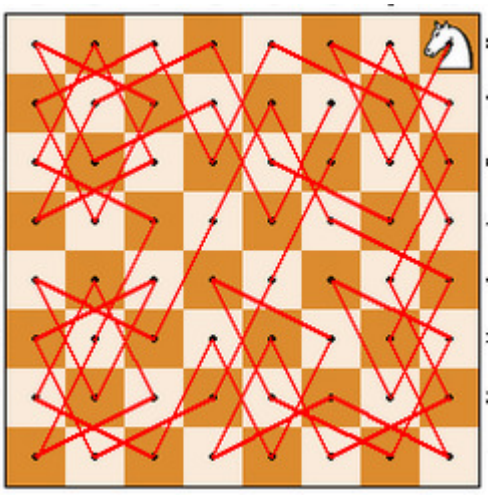
   Fabriquez par conséquent une méthode def ***controleur()*** chargée de transformer le modèle. Pourquoi **controleur()** et pas **controleur** ?

2. Le controleur va se charger de modifier le modèle (et de vérifier que la vue est synchronisée avec le modèle). On cherche à résoudre le problème du cavalier d'Euler.

   Aux échecs, le cavalier peut se déplacer de la manière suivante :



A partir d'une des cases de l'échiquier, on cherche donc à trouver la séquence de déplacements du cavalier qui permet de passer par toutes les cases de l'échiquier une et une seule fois. Par exemple :



Le controleur modifie donc le modèle pour trouver ce déplacement. La position initiale est une valeur de case à 1 (1ère étape), la seconde position 2, etc.. Le modèle sera donc une matrice d'entiers initialisée à zéro. Le controleur prend un paramètre (la position de la case de départ) et modifie le modèle. Modifiez def ***controleur()*** pour rajouter un argument de coordonnées de départ du cavalier. A la fin de l'éxécution du controleur, le modèle contient pour chaque case, l'indice de passage du cavalier.

3. Un point important pour cette recherche, c'est de trouver les positions de déplacement pour le cavalier, étant données ses coordonnées (x,y). Vous allez donc créer une fonction locale à controleur qui renvoie une **liste** des positions possibles de déplacement du cavalier à partir de sa position (x,y). Pensez que vous êtes en Scala et que vous pouvez renvoyer des couples de valeurs de manière simple.

   def trouveDeplacementsCavalier(xy_ : Tuple2[Int, Int]): List[Tuple2[Int, Int]] = …

   Commencez par regarder dans **scaladoc** comment fonctionnent les listes Scala. Vous utiliserez en particulier l'opérateur « :: » . Faites attention à ne retourner que les positions

possibles : utilisez un **filter** pour vérifier les coordonnées qui sont DANS l'échiquier (n'oubliez pas que vous avez accès aux éléments d'un couple par ._1 et par ._2).

Si (et seulement si), on a déjà vu les fonctions partielles en cours, utilisez une fonction partielle avec déconstructeur – sinon utilisez la notation fonctionnelle classique.

4. Fabriquez la fonction locale **def trouvePositions(xy_ : Tuple2[Int, Int], etape_ : Int): Boolean** qui renvoie vrai si le système a trouvé une solution (dans this.modele) au problème. Evidemment, cette fonction est locale au controleur.

5. Rajoutez **def synchroniseVueAuModele()** pour que la vue reflète l'état du modèle (toujours dans **controleur**). Vous indiquerez le numéro de l'étape de recherche à la place du nom des pièces (ne modifiez pas le TD précédent).

6. Enfin, complétez **controleur** pour finaliser la recherche.

7. Rajoutez la redéfinition de **toString** pour **CavalierEuler** afin de pouvoir afficher l'échiquier quand on passe une instance de **CavalierEuler** à **print**.

8. Vous devrez être capable des appels suivants :

```
object Main { // notez que le programme principal ne contient que très peu de choses ☺

 def main(args: Array[String]): Unit ={

  val euler = new CavalierEuler(6)

  euler.controleur(0,0)

  println(euler)

 }

}
```

avec la sortie console :

	0	1	2	3	4	5
0	1	20	27	18	9	36
1	26	11	2	21	28	17
2	3	22	19	10	35	8
3	12	25	4	31	16	29
4	5	32	23	14	7	34
5	24	13	6	33	30	15

Trouvez une solution pour un echiquier de 5x5, notez le temps :

6x6 :

7x7 :

8x8 (si vous y arrivez) :

Dessinez une courbe sur une feuille, à l'échelle, en abscisse le coté du carré (5, 6, 7, 8, 9, 10) et en ordonnée, le temps. Dessinez la courbe et essayez visuellement de trouver combien de temps il faudrait pour trouver une solution pour 8x8, 9x9, 10x10 et 11x11 (donnez ces temps dans le readme). Conclusion ?

9. Bon : on va accélérer la recherche en utilisant une heuristique. Quand le cavalier doit se déplacer, on regarde la liste des nouvelles positions possibles. Cette liste n'est pas ordonnée. On voit bien que plus on avance dans le recherche, plus on remplit les cases, et moins il reste de solutions à envisager (et plus ça va vite). D'un autre côté, si la nouvelle position du cavalier a beaucoup de successeurs possibles, alors le nombre de combinaisons sera important au niveau n+1 de la récursivité. L'idée ici, va être d'ordonner les nouvelles positions du cavalier pour envisager en premier, les nouvelles positions qui ont peu de nouvelles positions au rang suivant – on remplit l'échiquier le plus vite possible en commençant par les positions qui ont moins de successeurs au rang suivant.

   Vous allez donc modifier **def trouvePositions(xy_ : Tuple2[Int, Int], etape_ : Int): Boolean** pour que cette liste soit triée. **Voici comment** : En premier dans la liste, on veut les cases qui auront le moins de possibilités de saut de cavalier.

   On peut calculer *a priori*, pour chaque case de l'échiquier (et indépendamment du problème du cavalier) le nombre de voisins de saut de cavaliers.

   Si le cavalier se trouve sur la case bleu au centre, alors il peut sauter sur 6 cases différentes (notées en bleu également). Vous allez donc créer une **table** qui associe chaque case au nombre de points de sauts possibles.

   val heuristique = Array.fill(cote_, cote_)(0)

Vous allez pouvoir initialiser cette table avec pour la coordonnée (i,j) le nombre de points de sauts possibles, c'est-à-dire la longueur de la liste **trouvePositions((i,j)).**

Maintenant, dans votre algorithme, il reste à trier la liste des cases à explorer pour envisager en premier les cases qui ont le moins de point de saut. Vous pouvez faire de tri de manière simple en utilisant (par exemple après votre **filter** sur les cases occupées) un **sortBy** qui utilise la valeur de la case du tableau heuristique.

Faites les modifications et retestez le temps de calcul de :

5x5 :

6x6 :

7x7 :

8x8 :

9x9 :

Donnez les temps dans le **readme**.

Conclusion ?