Juston Thibaut
Dupré-la-Tour Grégoire
01/10/2018

Rapport de TP

World of Ecn

-

TP4

-

Gestion d’une multitude de personnages dans notre jeu

Juston Thibaut
Dupré-la-Tour Grégoire
01/10/2018

Introduction :

Le but de ce TP est de réorganiser nos personnages et créatures jusqu'alors indépendants dans des listes afin de faciliter la manipulation de ces objets. Dans l'optique de manipulation des listes il est important de regarder quel type de liste est le plus efficace. Nous comparerons donc les types de listes et implémenterons la meilleure. Nous ferons aussi en sorte de pouvoir générer un nombre X de protagoniste à une certaine distance les uns des autres.

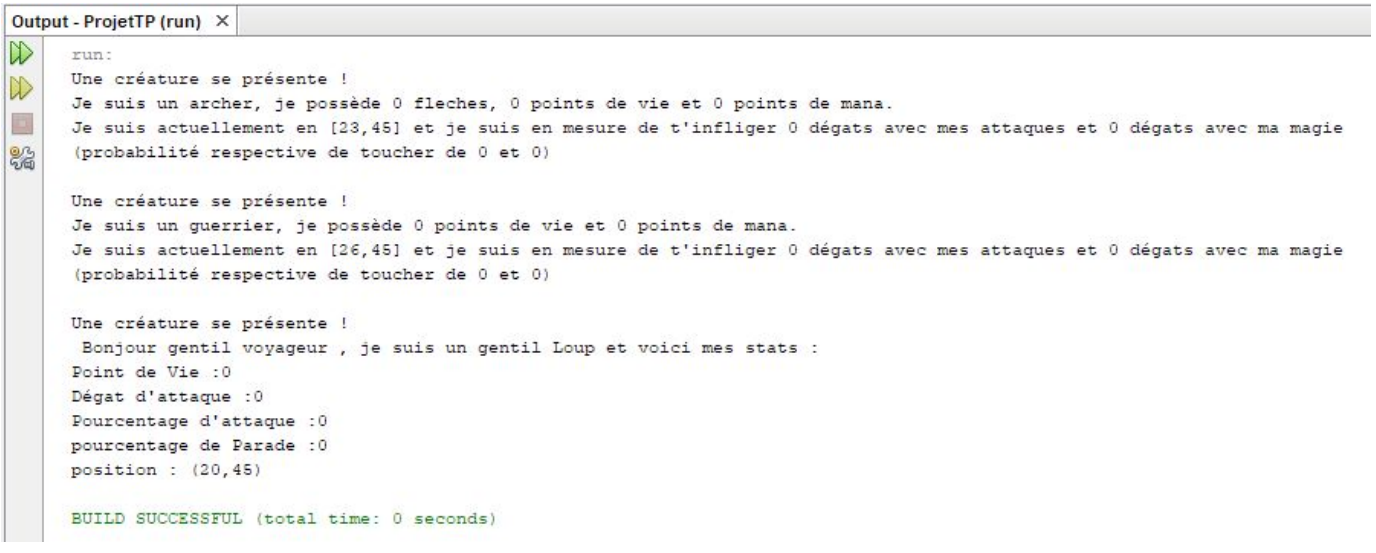
Techniquement la totalité de ce TP repose sur la gestion des listes.

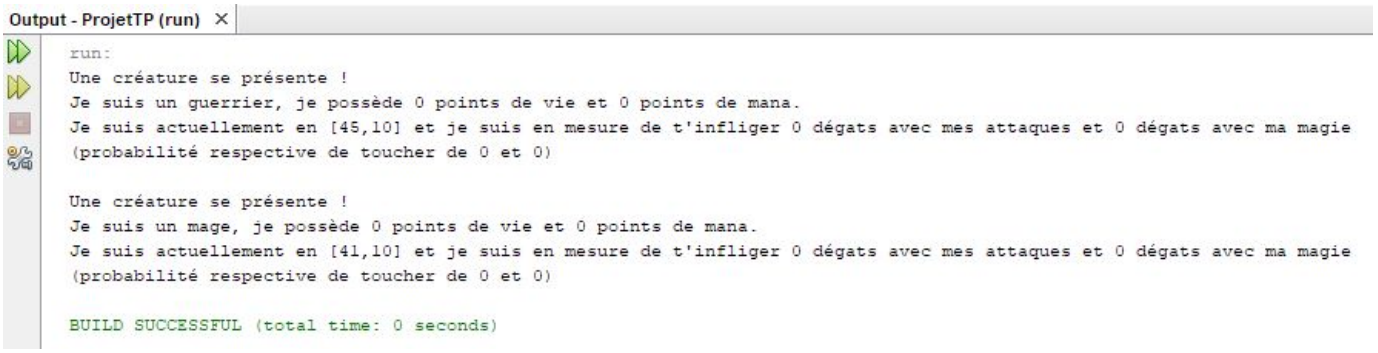
Création d'un nombre aléatoire de créatures différentes :

Pour générer un nombre aléatoire de créatures ou d'objet dans notre monde, on peut assigner à chaque entité qui sera créée un nombre aléatoire, puis boucler pour chaque créature sur ce nombre généré, ce qui en retour va nous créer un nombre aléatoire d'entités voulus.

Pour ce TP, nous avons dans un premier temps décidé de mettre cette solution en place en mettant les créatures dans un **ArrayList** de **Creature** et dans une autre **ArrayList** d'**Objet**.

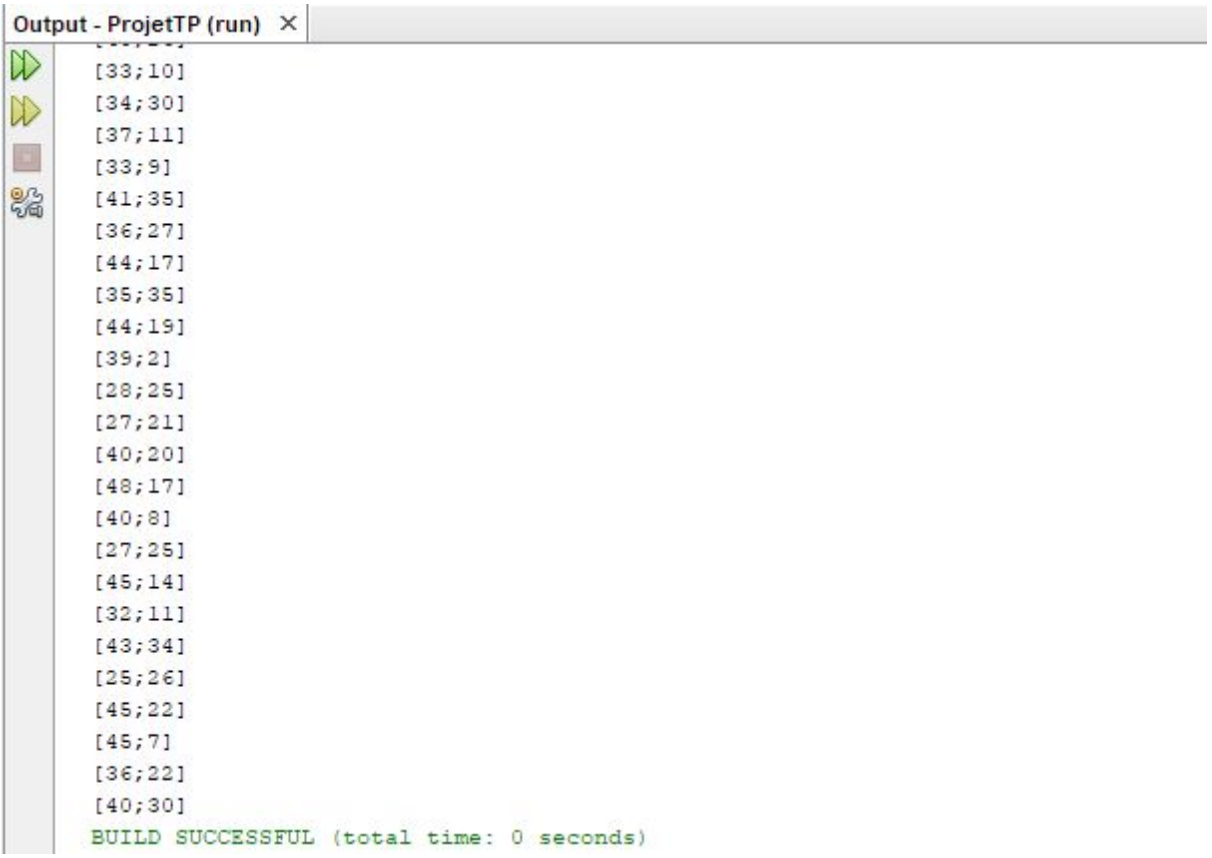
Pour l'exemple ci-dessous, on défini le nombre de créature présente de chaque type comme étant 0 ou 1, et on obtient les résultats suivants.
On peut donc voir qu'un nombre aléatoire d'entité est bien générée, puisque les classes sont présentes ou absentes.





Juston Thibaut
Dupré-la-Tour Grégoire
01/10/2018

On ne se marche pas sur les pieds :
Nous avions déjà implémenté le fait que lors de l'initialisation de leur position, les créatures ne peuvent pas être présents sur une même case. Lorsqu'une position aléatoire est donnée à une créature, on parcourt la liste des positions des autres créatures déjà placées, et si cette position est déjà prise on génère une nouvelle position.
Pour 100 créatures dans une carte de 50x50 positions, voici la sortie :



(il est compliqué ici de représenter tous les points, mais nous avons bien vérifié et aucun des points ne sont au même endroit)

Juston Thibaut
Dupré-la-Tour Grégoire
01/10/2018

On compare maintenant les temps de calculs des barycentre pour une LinkedList avec deux façons de calculer le barycentre, en se basant sur la longueur de la liste et en utilisant les itérateurs.Pour 100 000 Créatures:

```
le barycentre des creatures se trouve en : (499.40684, 499.37997)
Le temps écoulé pour le calcul du barycentre est de : 8522027 nanosecondes
le barycentre des creatures se trouve en : (499.40642593574063, 499.38343616563833)
Le temps écoulé pour le calcul du barycentre est de : 8256546 nanosecondes
BUILD SUCCESSFUL (total time: 1 minute 0 seconds)
```

On fait maintenant la même expérience mais pour une ArrayList de 100 000 créatures.

```
le barycentre des creatures se trouve en : (499.39778, 499.6004)
Le temps écoulé pour le calcul du barycentre est de : 6811350 nanosecondes
le barycentre des creatures se trouve en : (499.39968600313995, 499.5963440365596)
Le temps écoulé pour le calcul du barycentre est de : 2935977 nanosecondes
BUILD SUCCESSFUL (total time: 54 seconds)
```

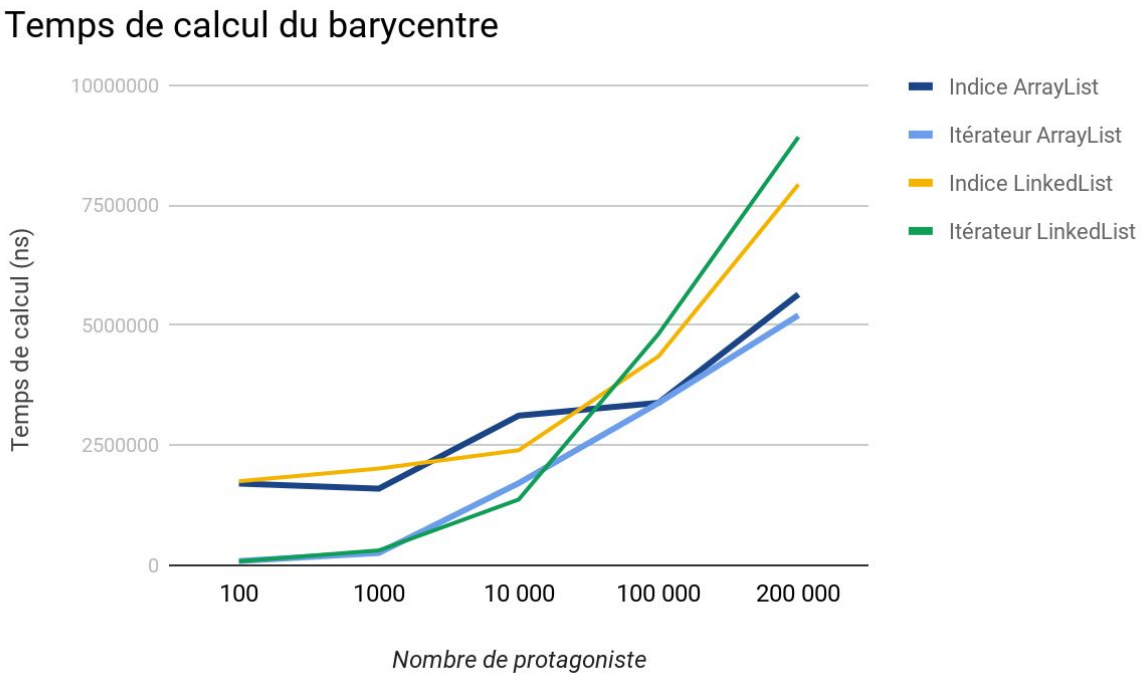
Dans tous les cas le barycentre a été calculé en utilisant la méthode nanoTime qui était la méthode conseillée dans le TP.

	Arraylist		LinkedList	
Nombre de protagonistes	Indice ArrayList	Itérateur ArrayList	Indice LinkedList	Itérateur LinkedList
100	1704478	90439	1752615	84239
1000	1598358	258917	2017367	311794
10 000	3119406	1712501	2398815	1372262
100 000	3386712	3384524	4356376	4823156
200 000	5646221	5209344	7935634	8923896

Pour 1 000 000 nous n’avons pas de résultats car le temps de calcul est trop long :

```
run:
BUILD STOPPED (total time: 19 minutes 55 seconds)
```

Juston Thibaut
Dupré-la-Tour Grégoire
01/10/2018



(attention, l'échelle en abscisse n'est pas linéaire)

Au premier regard on peut déjà remarquer qu'**aucune des droites n'est linéaire**, il n'est pas possible d'approcher facilement les prochains points. De plus les valeurs en elle même ne sont pas si importantes car elles dépendent du **matériel utilisé**.

On observe que de manière général les **itérateurs** sont beaucoup plus efficace que les parcours de listes traditionnelle, sur un **petit nombre d'objets**. L'influence des itérateurs tend à diminuer sur les listes plus vastes.

La différence entre ArrayList et LinkedList quand a elle est assez faible pour un petit nombre d'objets,on observe une différence à partir de **10 000 instances**. En effet on observe que pour un grand nombre d'objets l'utilisation d'une **ArrayList** sera plus efficace qu'une **LinkedList**, et cela indépendamment de l'utilisation des itérateurs.

On peut remarquer qu'il a été très facile de passer **d'ArrayList** aux **LinkedList**, il suffisait de changer dans la création car les méthodes pour accéder aux différents éléments dans java sont compatibles. Ce qui n'est pas le cas pour d'autre types de conteneurs tel que les **TreeSet** et les **HashMap.**

A la suite de ces résultats, on peut modifier notre première approche en parcourant notre ArrayList avec une boucle basée sur des itérateurs et non sur la taille de la liste, car on a vu que cette méthode était beaucoup plus performante. Cette méthode d'autant plus adapté qu'elle fonctionne particulièrement bien pour le nombre d'éléments présents dans les conteneurs (moins de 1000 en général).

Juston Thibaut
Dupré-la-Tour Grégoire
01/10/2018

Pour que tous les protagonistes soient à au moins trois unités les uns des autres, il nous a suffit de modifier un tout petit peu notre fonction de génération des positions, qui vérifiait si la distance entre deux créatures était différente de zéro.
Elle vérifie maintenant que la distance entre deux créatures est supérieur ou égale à 3.
Ci-dessous le fonctionnement de cette méthode pour dix protagonistes sur une carte de taille 10x10.

```
[3;9]
[1;2]
[8;0]
[4;0]
[0;7]
[9;9]
[8;5]
[6;6]
[6;9]
[0;4]
```

Pour qu'aucun de nos protagonistes ne soit au même endroit, il faut qu'à chaque déplacement d'une unité, on regarde si ma case de déplacement est déjà prise, par exemple en parcourant un **ArrayList** contenant toutes les positions de toutes les unités. Deux cas de figures s'offrent alors à nous :

1) La case est vide, dans ce cas la position de l'unité est effectivement mise à jour
2) La case est occupée, il faut alors générer un nouveau déplacement dans la cas d'un déplacement aléatoire, ou avertir l'utilisateur que le déplacement demandé n'est pas valide si il est à l'origine du déplacement.

Dans notre cas il est probable que l'on crée dans la classe **World** une fonction **deplace(Creature c)** qui génère une nouvelle position proche de l'ancienne, vérifie que cette position est bien libre, et met à jour la position de la créature, le tout en utilisant des ArrayList et en bouclant avec des itérateurs pour plus de performances.

Conclusion:

Ce Tp qui reposait sur la création de listes de créatures, n'a pas trop posé de problèmes de prime abord car nous avions déjà créer dans les TPs précédent des listes de créatures ainsi que des méthodes prenant en compte des listes en entrée.

La partie la plus prenante a été dans la comparaison des temps de calcul car les calculs ont été répété de nombreuses fois afin d'obtenir des résultats probants. Le problème étant que même en observant les différents temps de calcul il n'est pas facile d'avoir un avis tranché sur l'efficacité de certains type de liste du fait des comportements non linéaires en fonction du nombre d'instance et que nous ne mesurons pas le temps de création des liste mais juste le temps de calcul du barycentre.