TP de Java bis
L'armée de monstres

MEIM

2024-2025

Pour ceux qui ont terminé le TP1, vous pouvez faire ce TP supplémentaire qui a trois buts distincts. Le premier but est d'apprendre à manipuler des structures de données. Le second but est de concevoir des tableaux extensibles et des listes chaînées. Ces structures de données classiques permettent de gérer des ensembles homogènes non bornés. Le troisième but est de vous amener pas à pas vers une conception modulaire, qui nous servira à introduire les principes de programmation objet des prochains cours.

Durée indicative des exercices

- Exercice 1 : facile, 40mn
- Exercice 2 : moyen, 1h
- Exercice 3 : facile, 15mn
- Exercice 4 : difficile
- Exercice 5 : difficile

Contenu du TP

Le but de ce TP est de coder un petit jeu. Dans celui-ci, des joueurs se promènent dans le monde réel avec leur téléphone portable pour capturer des monstres virtuels géolocalisés. Ces monstres servent ensuite aux joueurs à conquérir des lieux spéciaux, ce qui permet aux joueurs d'interagir. Toute ressemblance avec un jeu lancé en grande pompe en 2016 est totalement non-fortuite.

Le but de l'exercice n'est pas de mettre entièrement le jeu en œuvre, mais uniquement la partie du jeu qui gère les armées de monstres. Cette mise en œuvre est rendue difficile, car les scénaristes fantaisistes du jeu ont prévu que chaque joueur puisse acquérir un nombre non borné de monstres. Pour cette raison, utiliser un tableau de taille fixe pour stocker les monstres est impossible. Nous allons donc étudier des structures de données permettant de traiter un nombre non borné d'éléments.

L'exercice est séparé en quatre parties :
— Dans une première partie, nous concevons la structure de données qui représente un monstre.
— Dans une seconde partie, nous utilisons des tableaux extensibles pour stocker nos monstres. Un tableau extensible est une structure de données reposant sur un tableau, mais permettant de réallouer dynamiquement le tableau de façon à l'étendre. Dans cette partie, ainsi que dans la précédente, nous concevons notre application de manière non modulaire, c'est-à-dire que nous mettons en vrac l'ensemble des méthodes de notre programme dans la même classe.
— Dans une troisième partie, nous constatons que cette conception non modulaire ne permet pas de réutiliser le code. Pour cette raison, nous modifions le design de notre application.
— Dans une quatrième partie, nous observons qu'un tableau extensible n'est pas la structure de données la plus efficace en termes de performance si le nombre de monstres devient grand. Pour cette raison, nous modifions notre application de façon à utiliser des listes chaînées pour stocker nos monstres.

Remarque

Pour les étudiants qui ont déjà de bonnes habitudes de la programmation, certains choix de conception proposés dans cet exercice pourront vous paraître inadéquats (notamment coder des fonctions en dehors d'une classe au lieu de méthodes à l'intérieur de celle-ci). Ces choix sont volontairement inadéquats, mais visent à montrer aux étudiants qui n'ont pas encore l'habitude de programmer qu'il faut concevoir une application de façon modulaire. Vous pouvez implanter les questions avec vos propres choix, qui seront sûrement validés (ou pas ?!) dans les deux dernières parties du TP.
A la fin des deux premiers TPs, vous devriez être capables de mettre en œuvre une application modulaire et appliquer l'encapsulation de la programmation objet pour un projet donné.

Lisez attentivement le sujet une fois en entier avant de vous lancer dans la programmation.

Exercice 1 : Les monstres, nos premiers objets

Question 1.a Pour commencer, créez un projet nommé **monsters** contenant une unique classe nommée **Pokedex** avec une méthode **main**.

Question 1.b Dans le même package, créez une nouvelle classe nommée **Monster** et contenant les champs suivant :

- **name** : une chaîne de caractères donnant le nom du monstre.
- **health** : un entier donnant le nombre de points de vie du monstre. Lorsque ce nombre tombe à zéro, le monstre meurt.

Vous pourriez bien sûr imaginer ajouter d'autres champs à cette classe, par exemple une image représentant le monstre, mais nous nous contenterons dans cet exercice d'une version minimaliste de nos monstres.

Question 1.c Dans la méthode **main**, allouez un premier monstre que vous stockerez dans une variable nommée **aMonster**. Affectez le nom du monstre à "Pikachu", son nombre de points de vie à 0, puis affichez le nom du monstre sur le terminal. Vérifiez que votre programme affiche bien "Pikachu".

Question 1.d Dans la classe Pokedex, écrivez une méthode de classe **createMonster** prenant en argument un nom et un nombre de points de vie. Cette méthode doit renvoyer une référence vers une nouvelle instance de **Monster** initialisée de façon adéquate. Supprimez le code qui alloue et initialise un monstre dans la méthode **main**. Remplacez ce code par un appel à **createMonster** pour créer le monstre "Pikachu". Vérifiez que votre programme affiche toujours "Pikachu".

Question 1.e Dans la classe Pokedex, écrivez une méthode **displayMonster** prenant en paramètre un monstre. Cette méthode doit afficher **Monster<name>**, où **name** est le nom du monstre passé en paramètre. Au lieu de directement afficher le champ **name** du monstre dans la méthode main, utilisez **displayMonster** pour afficher votre monstre. Vérifiez que **Monster<Pikachu>** est bien affiché sur le terminal.

Question 1.f Avant de mettre en œuvre la structure de données stockant une armée de monstres, nous allons créer plusieurs monstres. Remplacez la création et l'affichage de **Pikachu** par une boucle créant et affichant 8 monstres dont les noms vont de **Pikachu0** à **Pikachu7**. Vérifiez que votre programme affiche bien vos 8 Pikachus dans la console et que leurs noms sont corrects.

> Attention
>
> Conservez bien le code qui crée les 8 Pikachus jusqu'à la fin de l'exercice, car vous en aurez besoin dans les questions suivantes.

Exercice 2 : Le tableau extensible

Dans cette partie nous utilisons des tableaux extensibles pour stocker nos monstres. Dans un premier temps, avant de nous occuper de rendre nos tableaux extensibles, nous commençons par utiliser des tableaux à taille fixe.

Le principe, à cette étape, est d'allouer un tableau suffisamment grand pour accueillir nos 8 monstres, et d'utiliser un indice indiquant le niveau de remplissage du tableau. Initialement cet indice d'insertion vaut 0. L'ajout du premier monstre se fait donc à cet indice dans le tableau et l'indice d'insertion est augmenté de 1. De façon similaire, si l'indice d'insertion vaut N, l'ajout d'un nouveau monstre se fait dans la case d'indice N du tableau et l'indice d'insertion est ensuite augmenté de 1.

Au début de l'exercice, nous considérons que le tableau a toujours une taille suffisante pour accueillir un nouveau monstre, et nous ne nous préoccuperons de gérer le problème du débordement du tableau qu'à la fin de l'exercice.

Si vous avez déjà fait ces tableaux extensibles dans le TP n°1, réutilisez votre implantation ici.

Question 2.a Créez une classe nommée **Army** contenant deux champs : un tableau de monstres nommé **monsters** et un entier nommé **top** indiquant jusqu'à quel indice le tableau est occupé. Écrivez aussi une méthode **Army createArmy()** dans la classe **Pokedex**. Cette méthode doit (i) créer une instance de la classe **Army**, (ii) allouer un tableau de taille 100 et l'affecter au champ monsters de l'instance nouvellement créée, (iii) fixer la valeur initiale du champ **top** de l'instance à 0 puisque l'armée est initialement vide, et (iv) renvoyer l'instance de la classe **Army**.

Question 2.b Écrivez, dans la classe **Pokedex**, une méthode nommée **addMonster** prenant en argument une armée et un monstre et permettant d'ajouter le monstre à l'armée. Dans cette version préliminaire de **addMonster**, nous considérons que le tableau monsters est suffisamment grand pour accueillir un nouveau monstre. Pour tester **addMonster**, modifiez la boucle de la méthode **main** qui crée les 8 Pikachus de façon à les ajouter à une armée au lieu de les afficher.

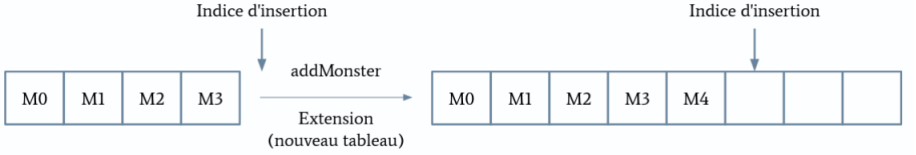
Question 2.c Écrivez, dans la classe **Pokedex**, une méthode nommée **void displayArmy(Army army)**. Cette méthode doit utiliser **displayMonster** pour afficher les monstres actuellement enregistrés dans l'armée. Vérifiez que votre programme est correct en appelant **displayArmy** à la fin de la méthode **main**. Cet appel devrait afficher les 8 Pikachus qui se trouvent dans l'armée.

Aide

Techniquement, **displayArmy** doit afficher les monstres se trouvant aux indices allant de 0 (inclus) à **top** (non inclus) du tableau monsters.

Question 2.d Nous pouvons maintenant rendre notre tableau de monstres extensible. Dans **createArmy**, au lieu de créer un tableau à 100 entrées lorsque vous créez une armée, créez un tableau à 4 entrées. Lancez votre programme, que constatez-vous ?

Question 2.e Modifiez votre méthode **addMonster** de façon à traiter le cas où **top** est plus grand ou égal que le taille du tableau de monstres. Dans ce cas, vous devez allouer un tableau plus grand avant de recopier l'ancien contenu.



Aide

Vous devez :
- Allouer un nouveau tableau nommé **tmp**. Ce tableau doit avoir une taille plus grande que le tableau de monstres original (par exemple, en doublant la taille du tableau original).
- Recopier le tableau original de monstres dans le tableau **tmp**.
- Affecter **tmp** au champ **monsters** de l'armée.

Remarquez que vous n'avez jamais libéré la mémoire occupée par l'ancien tableau de monstres. Pour rappel, en Java, le développeur n'a pas à se soucier de libérer la mémoire, car l'environnement d'exécution s'en occupe pour le développeur. Le mécanisme qui libère la mémoire s'appelle un ramasse-miette (*garbage collector* en anglais). Ce mécanisme de libération automatique de la mémoire est présent dans la plupart des langages modernes (Java, C#, Javascript, Python, Bash...), ce qui est une des raisons qui explique le succès de ces langages.

Question 2.f Quelle est la complexité de **addMonster** dans le pire des cas ? Dans le meilleur des cas ?

Question 2.g Avec cet exemple, nous nous rendons compte que la complexité dans le pire des cas n'est pas la plus adaptée pour décrire la complexité en pratique du tableau extensible. On a l'intuition que dans la majorité des cas, on aura une complexité constante. Pour mieux traduire la véritable complexité, nous pouvons utiliser la complexité amortie. Il s'agit de la complexité moyenne quand on appelle un grand nombre de fois une fonction. Quelle est la complexité amortie de **addMonster** ?

Exercice 3 : Conception Modulaire

Face au succès phénoménal de votre jeu, votre PDG, toujours aussi visionnaire, souhaite lancer une nouvelle franchise basée sur le film "Black Sheep", relatant les passionnantes aventures de moutons zombies. Dans ce nouveau jeu, un joueur se promène dans le monde réel et rencontre des moutons zombies géolocalisés qui se mettent à l'attaquer. Le joueur doit collectionner des pieux en argent pour vaincre les moutons zombies, ce qui lui rapporte des pièces lui permettant d'acheter des vêtements de luxe, le but du jeu étant d'avoir le personnage le plus chic du monde.

Question 3.a Souhaitant limiter le temps de développement, votre géniale PDG vous suggère de réutiliser la structure de données représentant les monstres du précédent jeu de façon à modéliser un mouton zombie. Pour quelle raison la structure non modulaire que nous avons choisi se prête mal à cette opération ?

Question 3.b Comme vous l'avez probablement compris, la structure que nous avons utilisé limite la réutilisabilité du code. Ce manque de réutilisabilité oblige à re-développer inutilement du code, ce qui a un coût financier non négligeable. Pour cette raison, avant de passer à la suite, nous redesignons pas à pas notre application de façon à pouvoir en extraire facilement des parties pour d'autres projets.

Le problème fondamental du design que nous avons utilisé est que les méthodes qui manipulent les monstres ou l'armée sont difficiles à identifier puisque ces dernières sont mises en vrac dans la classe **Pokedex**. Pour cette raison, nous allons recréer notre application de façon modulaire : nous allons mettre les méthodes qui manipulent les monstres dans la classe Monster et les méthodes qui manipulent l'armée dans la classe Army.

Dans la suite de l'exercice, vous devrez chercher comment faire les différentes *refactoring* avec votre environnement de développement qui, s'il est moderne et bien fait, devrait vous offrir sensiblement les mêmes fonctionnalités.

Déplacer les méthodes createMonster et displayMonster dans la classe Monster. Sous Eclipse, vous pouvez cliquer avec le bouton de droite de la souris sur la méthode, aller dans le sous-menu Refactor et sélectionner Move. Sous IntelliJ, la procédure est la même, mais il vous faut choisir Move Members.. à la fin. De la même façon, déplacez createArmy, addMonster et displayArmy dans la classe Army. Relisez le code de votre main : vous devriez constater que maintenant, par exemple, vous appelez Army.createArmy pour créer une armée puisque la méthode createArmy se trouve dans la classe Army.

Question 3.c Maintenant que nos méthodes sont rangées à leur place, nous pouvons les renommer : il est inutile que la méthode de création de monstre de la classe **Monster** s'appelle**createMonster**, puisque le suffixe **Monste**r devient maintenant implicite. Dans la classe **Monster**, renommez donc la méthode **createMonster** en **create** et la méthode **displayMonster** en **display**. De façon similaire, dans la classe **Army**, renommez la méthode **createArmy** en**create** et la méthode **displayArmy** en **display**.

Le design de votre application est maintenant correct. Vous pouvez facilement réutiliser la classe **Monster** pour concevoir un jeu de moutons zombies, ce qui minimise l'effort de développement et donc le coût financier. De façon générale, comme mentionné en cours, vous constaterez au fur et à mesure que vous développerez qu'il est beaucoup plus judicieux de mettre les méthodes qui manipulent une structure de données dans la classe qui définit cette structure de données. C'est tout le principe de la programmation par objet et de l'encapsulation.

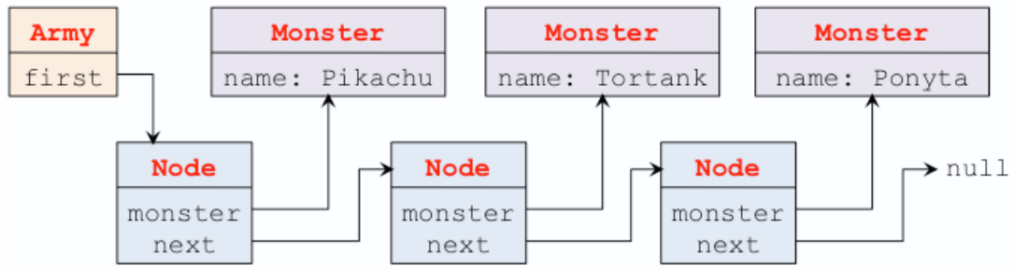
Exercice 4 : La liste chaînée

Avant de commencer, de façon à conserver le code que vous avez écrit jusqu'à présent, nous vous conseillons de dupliquer votre projet **monster**. Vous pouvez chercher la fonctionnalité qui fait ça sur Visual Studio ou Eclipse (ou votre IDE préféré). Mettez le nouveau nom (**monster-array** par exemple).

Suite à l'extraordinaire succès de votre jeu, vous vous mettez à recevoir des centaines de milliers de mails d'utilisateurs mécontents, car les performances de votre jeu sont désastreuses. Après une analyse de performance poussée, vous vous apercevez que des joueurs monomaniaques collectionnent des centaines de milliers de monstres. Lorsque le tableau extensible est étendu pour accueillir les nouveaux monstres de ces joueurs, des centaines de milliers de cases sont copiées de l'ancien tableau vers le nouveau, ce qui écroule les performances de vos serveurs et impacte l'ensemble des joueurs connectés au même serveur (cette anecdote est bien sûr imaginaire, mais ressemble aux problèmes de performances que vous pouvez rencontrer tous les jours dans une entreprise).

De façon à éviter ces recopies de tableaux, nous mettons en œuvre l'armée de monstres avec des listes chaînées. Une liste chaînée est une structure de données classique évitant cette recopie au prix d'un temps de parcours des éléments plus long. Le principe est de construire une chaîne d'éléments homogènes, chacun référençant le suivant de la liste. Pour cela, nous introduisons une nouvelle classe nommée **Node** permettant de référencer (i) une instance de **Monster** et (ii) l'instance de **Node** suivante dans la liste.

La figure ci-après présente la structure sous une forme graphique. Chaque boite représente une instance d'une classe. Le nom en haut de la boite donne le nom de la classe instanciée et les noms en dessous sont les champs les plus importants de l'instance. Les flèches quant à elles représentent les références. Dans cette illustration, l'armée (l'instance de **Army**) référence le premier nœud de la liste chaînée. Le premier nœud référence un premier monstre (**Pikachu**) et le deuxième nœud. Le deuxième nœud référence le deuxième monstre (**Tortank**) et le nœud suivant de la liste, jusqu'à la fin de la liste qui est indiquée par la référence null.



Question 4.a Commencez par écrire une classe nommée **Node** contenant deux champs : un champ **monster** référençant le monstre associé au nœud et un champ **next** référençant le nœud suivant de la liste. Ajoutez à cette classe une méthode nommée **create** prenant en argument un monstre et un nœud suivant et renvoyant un nouveau nœud initialisé de façon adéquate.

Question 4.b Dans la classe **Army**, ajoutez un champ **first** de type **Node**. Dans la méthode **create** de **Army**, initialisez **first** avec la valeur null, ce qui indique à Java que cette référence ne pointe nulle part.

Attention : Ne supprimez pas encore les anciens champs qui stockent les monstres sous la forme d'un tableau extensible de façon à conserver un code correct tout au long de l'exercice.

Question 4.c Dans la méthode **addMonster** de la classe **Army**, sans supprimer le code qui s'y trouve déjà, modifiez la liste de façon à ajouter le monstre passé en paramètre au début de la liste (à gauche sur la figure ci-dessus).

Aide

Techniquement, il vous suffit d'appeler la méthode **create** de **Node** de façon adéquate et d'affecter le résultat de cet appel au champ first de l'armée.

Question 4.d Quelle est la complexité de addMonster ?

Question 4.e Dans la méthode **display** de la classe **Army**, en plus d'afficher les monstres se trouvant dans le tableau extensible, affichez le contenu de la liste chaînée. Vérifiez que votre programme affiche maintenant bien deux fois votre armée.

Vous devriez remarquer que, lors du parcours de la liste chaînée, les monstres sont affichés dans l'ordre inverse de celui de leur insertion. Ce phénomène est tout à fait normal puisque vous ajoutez les nouveaux nœuds au début de la liste et que vous affichez votre liste en commençant par le début.

Question 4.f Vous pouvez maintenant nettoyer le code et supprimer le tableau extensible qui est devenu inutile. Pour cela, commencez par supprimer les champs **monsters** et **top** de la classe **Army**, et corrigez toutes les erreurs éventuelles que vous indique votre environnement de développement.

Question 4.g Pour les étudiants aventuriers, nous vous proposons d'écrire une méthode **delMonster** dans la classe **Army** permettant de supprimer un monstre de l'armée. Cette méthode doit prendre en paramètre une référence vers une armée et une référence vers un monstre.

Aide

Votre méthode doit traiter trois cas distincts :
- Si le premier nœud de la liste (**first**) vaut null, il n'y a rien à faire.
- Si le premier nœud de la liste référence un nœud qui référence le monstre passé en paramètre (l'opérateur == permet de savoir si deux références sont identiques), alors il faut affecter **first** à **first.next**.
- Sinon, votre méthode doit parcourir la liste de façon à identifier le prédécesseur du nœud qui référence le monstre passé en paramètre. Une fois identifié, si le prédécesseur est stocké dans une variable locale nommée **pred**, votre méthode doit affecter **pred.next.next** à **pred.next** de façon à sauter le nœud que vous souhaitez supprimer.

Question 4.h Quelle est la complexité de **delMonster** dans le pire des cas ?

Question 4.i Nous proposons maintenant d'écrire une fonction **getMonster** qui prend en entrée une armée et un index **i** et retourne le monstre à l'index **i**, s'il existe. Sinon, nous retournons null.

Question 4.j Quelle est la complexité de la fonction **getMonster** qui permet d'accéder à l'élément **i** d'une liste chaînée ? Comment se compare-t-elle à la complexité dans le cas d'un tableau extensible ?

Question 4.k Comme nous l'avons fait avec le tableau extensible plus haut, nous voulons rendre notre conception plus modulaire en poussant la logique de la liste chaînée directement dans la classe Node. Écrivez directement dans **Node** les fonctions :
- **add** qui prend en entrée un **Node** et un **Monster** et retourne une liste commençant par le monstre et continuant vers l'ancienne liste.
- **del** qui prend en entrée un **Node** et un **Monster** et retourne une liste sans le monstre.
- **get** qui prend en entrée un **Node** et un index et retourne le monstre à l'index ou null si ce n'est pas possible.
- **display** qui doit afficher tous les éléments de la liste.

Ensuite, réécrivez **addMonster**, **delMonster**, **getMonster**, et **display** dans **Army** pour n'utiliser que les fonctions dans **Node**.

Exercice 5 : Pour aller plus loin

Les questions qui suivent sont libres. À vous de gérer vos projets et classes pour répondre au problème !

Question 5.a Dans cet exercice, nous allons utiliser la classe **Node** de la dernière question de l'exercice précédent. Cependant, nous voulons maintenant manipuler des entiers et non des monstres. Modifiez **Node** en conséquence dans un nouveau projet. On réutilisera cette nouvelle classe par la suite. Pour rendre la lecture des listes plus claire, on affichera les listes chaînées sous la forme :

$$20 \to 12 \to 10 \to 7 \to 5$$

Pour cela, on pourra utiliser la fonction **System.out.print** qui ne rajoute pas de retour à la ligne à la fin de chaque ligne.

Question 5.b Pour faciliter la création des listes, implémentez une fonction **toNode** prenant en entrée un tableau et retournant une liste des éléments du tableau.
Exemple : $\{5, 7, 10, 12, 20\}$ devient $5 \to 7 \to 10 \to 12 \to 20$.

Question 5.c Écrivez une fonction **reverse** qui prend en entrée une liste chaînée et retourne la liste inversée.
Exemple : $20 \to 12 \to 10 \to 7 \to 5$ devient $5 \to 7 \to 10 \to 12 \to 20$.

Question 5.d Écrivez une fonction qui retire les doublons d'une liste chaînée triée. Exemple : $0 \to 0 \to 1 \to 2 \to 2 \to 3 \to 4 \to 4$ devient $0 \to 1 \to 2 \to 3 \to 4$.

Question 5.e Pour cette question, nous supposons que nous représentons les nombres par la liste chaînée de leurs chiffres en ordre inversé. Par exemple, 123 devient $3 \to 2 \to 1$. Écrivez une fonction capable d'additionner deux nombres représentés par une liste. Exemple : $3 \to 2 \to 1 + 2 \to 1 = 5 \to 3 \to 1$.

Question 5.f Ecrivez une fonction **Node getMiddleNode(Node nodeFrom, Node nodeTo)** qui retourne le nœud situé à mi-distance entre deux nœuds donnés en entrée. Essayez d'écrire cette fonction sans calculer la taille totale de la liste.

Aide

Il faut utiliser deux curseurs qui parcourent la liste, dont un qui se déplace deux fois plus vite.

Question 5.g Implémentez le tri fusion pour les listes. Attention, il ne faut pas tout faire dans une seule fonction ! À vous de vous organiser pour créer des fonctions auxiliaires et faire le code le plus lisible possible.

Aide

Voici quelques fonctions auxiliaires qui pourraient vous être utiles :
- **Node getLastNode(Node node)** qui retourne le dernier nœud d'une liste.
- **Node mergeSorted(Node first, Node second)** qui fusionne deux listes triées en une nouvelle liste triée elle aussi.
- **Node mergeSortAux(Node nodeFrom, Node nodeTo)** qui prend le nœud au milieu entre nodeFrom et nodeTo, trie chaque demie-liste et fusionne le résultat.