# Applications utilisant les classes de l'API Java

Conseil : n'essayez pas n'importe quelle méthode sans vraiment la connaître.
Les méthodes utiles sont reprises dans le document API_JAVA !

## Exercices obligatoires

## A Combat de guerriers

Une équipe de guerriers s'est constituée pour vaincre une créature du mal.
A tour de rôle, chaque guerrier va combattre cette créature maléfique.
Lors de chaque combat, le guerrier et la créature frappent en même temps, s'affaiblissent et peuvent même mourir.
Les combats vont continuer tant que la créature n'est pas morte et qu'il y a des guerriers en vie.

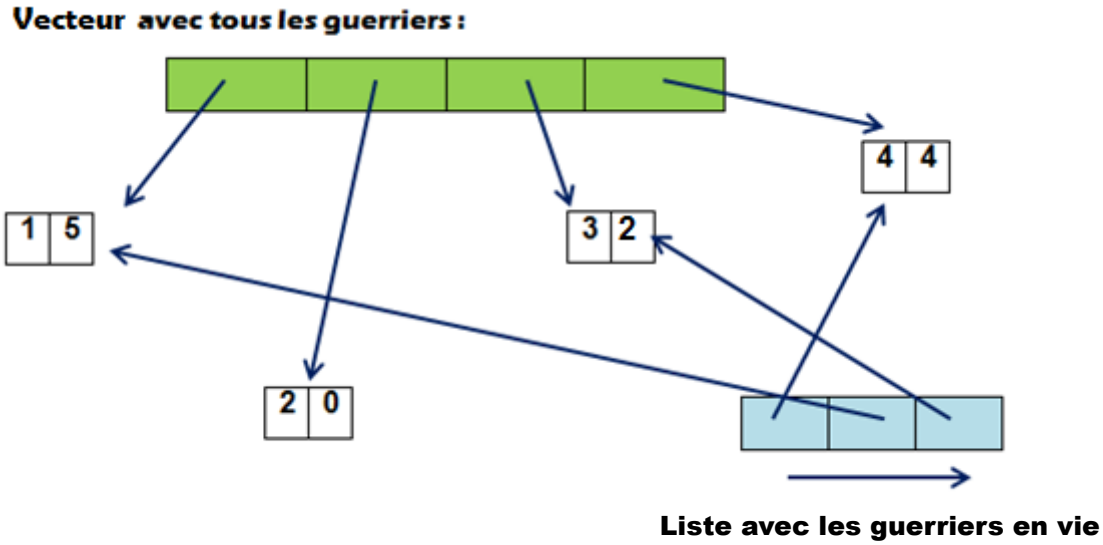
Il s'agit bien sûr d'un jeu. L'état d'un participant se calcule en points de vie.
C'est un lancer d'un dé qui va déterminer le nombre de points de vie perdus lors d'un combat.
Le participant est mort lorsqu'il n'a plus de points de vie.

A1 Implémentation :

Vous avez reçu une classe ***Guerrier***.
Tout objet de la classe *Guerrier* possède comme attributs un numéro et un nombre de points de vie.
**Vous allez compléter la classe *EquipeGuerriers*.**

Tous les guerriers sont sauvegardés dans un vecteur.
Pour ce vecteur, vous utiliserez un objet de la classe *ArrayList* de l'API Java.

Pour connaître l'ordre des combats, les guerriers sont placés dans une liste.
Si un guerrier est toujours vivant au terme de son combat, il est replacé en fin de liste.
Pour cette liste, vous utiliserez un objet de la classe *LinkedList* de l'API Java.
Voici un schéma pour vous aider à mieux visualiser un objet de la classe *EquipeGuerrier* :
Dans cet exemple, on y retrouve une *équipe* de 4 guerriers.
Le guerrier n°2 est mort (0 point de vie)
Les autres guerriers sont encore en vie.
Le prochain guerrier à affronter le mal porte le n° 4, il sera suivi par le n°1 et puis par le n°3




Le constructeur reçoit en paramètre le nombre de guerriers et le nombre de points de vie attribué au départ à chaque guerrier.
Il va créer les différents guerriers. C'est le constructeur qui attribue à chaque guerrier un numéro. La numérotation commence à 1.
Il va placer chaque guerrier simultanément dans la liste et le vecteur.
Ne perdez pas de vue que la liste et le vecteur référencient des mêmes objets !

La classe *EquipeGuerriers*, en plus du constructeur, possède les 3 méthodes suivantes :

- `int nombreGuerriersEnVie()`
- `Guerrier jouer(int pointsDeViePerdus)`
- `Guerrier getGuerrier(int numero)`

La méthode **`nombreGuerriersEnVie()`** renvoie le nombre de guerriers toujours en vie.
Un guerrier est en vie si son nombre de points de vie est > 0.

La méthode **`jouer()`** s'occupe du combat d'un guerrier.
Elle sélectionne le guerrier qui doit combattre, lui retire les points de vie perdus et le remet éventuellement dans la liste.
Elle renvoie le guerrier qui a combattu.
Cette méthode va lever une *NoSuchElementException* si tous les guerriers sont morts.

La méthode **`getGuerrier()`** renvoie le guerrier dont le numéro est passé en paramètre.
Ce guerrier peut être mort.
Cette méthode va lever une *IllegalArgumentException* si le numéro ne correspond pas à un guerrier.

**A2 Vous allez compléter la classe *JeuGuerrier*.**

L'équipe compte 3 guerriers.
Chaque guerrier a reçu 10 points de vie. La créature maléfique en a reçu 30.
Il faut pouvoir suivre les combats grâce à des affichages appropriés.
Voici un exemple de ce que pourrait être l'affichage à la suite d'une exécution de votre programme :

L'equipe compte 3 guerriers en vie
Suite au combat entre la creature du mal et le guerrier n°1 :
Le guerrier vient de perdre 4 points de vie
Il lui reste 6 points de vie
La creature du mal vient de perdre 5 points de vie
Il lui reste 25 points de vie

L’equipe compte 3 guerriers en vie
Suite au combat entre la creature du mal et le guerrier n°2 :
Le guerrier vient de perdre 6 points de vie
Il lui reste 4 points de vie
La creature du mal vient de perdre 4 points de vie
Il lui reste 21 points de vie

L’equipe compte 3 guerriers en vie
Suite au combat entre la creature du mal et le guerrier n°3 :
Le guerrier vient de perdre 4 points de vie
Il lui reste 6 points de vie
La creature du mal vient de perdre 4 points de vie
Il lui reste 17 points de vie

L’equipe compte 3 guerriers en vie
Suite au combat entre la creature du mal et le guerrier n°1:
Le guerrier vient de perdre 3 points de vie
Il lui reste 3 points de vie
La creature du mal vient de perdre 3 points de vie
Il lui reste 14 points de vie

L’equipe compte 3 guerriers en vie
Suite au combat entre la creature du mal et le guerrier n°2:
Le guerrier vient de perdre 4 points de vie
Le guerrier est mort
La creature du mal vient de perdre 4 points de vie
Il lui reste 10 points de vie

L’equipe compte 2 guerriers en vie
Suite au combat entre la creature du mal et le guerrier n°3:
Le guerrier vient de perdre 1 points de vie
Il lui reste 5 points de vie
La creature du mal vient de perdre 6 points de vie
Il lui reste 4 points de vie

L’equipe compte 2 guerriers en vie
Suite au combat entre la creature du mal et le guerrier n°1:
Le guerrier vient de perdre 6 points de vie
Le guerrier est mort
La creature du mal vient de perdre 2 points de vie
Il lui reste 2 points de vie

L’equipe compte 1 guerriers en vie
Suite au combat entre la creature du mal et le guerrier n°3:
Le guerrier vient de perdre 3 points de vie
Il lui reste 2 points de vie
La creature du mal vient de perdre 4 points de vie
La creature du mal est morte

A3 Ajoutons la possibilité d'ajouter un nouveau guerrier à l'équipe, même si les combats ont débuté.

Ce guerrier va recevoir un nouveau numéro et va prendre place dans la liste d'attente pour les combats.

Il ira se placer après le premier joueur encore vivant qui portait, avant son arrivée, le plus grand numéro.

Dans l'exemple ci-dessus, en cas d'ajout, le guerrier ajouté portera le numéro 5 et ira s'intercaler entre le guerrier 4 et le guerrier1 dans la liste d'attente.

Ecrivez la méthode **`ajouterNouveauGuerrier()`** qui s'occupe de cet ajout.
Cette méthode renvoie le numéro de ce nouveau guerrier.

La classe *TestAjoutNouveauGuerrier* permet de tester cette nouvelle méthode.

A4 Remplissez la table suivante :

| **METHODE** | **COUT** |
|---|---|
| **`nombreGuerriersEnVie()`** | O1 |
| **`jouer()`** | On |
| **`getGuerrier()`** | O1 |
| **`ajouterGuerrier()`** | On |

Vérifiez vos réponses avec celles du document *ASol* qui se trouve sur moodle.

# B Unshuffle sort

L'implémentation de cet algorithme de tri utilise une liste de « deque ».
La structure de données appelée « double-ended queue » ou « deque » possède les caractéristiques d'une file d'attente, mais elle permet des ajouts et des retraits aux 2 extrémités de la file.

Cet algorithme de tri comporte deux étapes. La première consiste à répartir les données à trier dans un nombre variable de *deques* triés.
Lorsque toutes les données auront été réparties la deuxième étape se chargera de remplir la table à renvoyer.

Les 2 étapes sont basées sur le principe suivant :
La liste des *deques* devra toujours être triée par ordre croissant des premiers éléments de chaque *deque*
Chaque *deque* aussi est trié.

Dans le cadre du cours, les éléments seront des entiers. Les doublons sont acceptés.

Etape 1 :
Chaque entier sera placé sur le premier *deque* qui pourra le contenir en respectant le tri imposé.
Si aucun *deque* ne peut contenir l'entier, un nouveau *deque* avec cet entier sera ajouté en fin de liste.

Exemple :

Table à trier :

| 3 | 12 | 2 | 4 | 18 | 17 | 6 | 1 | 15 | 14 |
|---|---|---|---|---|---|---|---|---|---|

Au départ la liste est vide.

```
placerEntier(3)
```

| 3 |
|---|

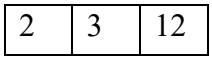
Un *deque* est placé dans la liste. L'entier 3 est ajouté en premier.

```
placerEntier(12)
```

| 3 | 12 |
|---|---|

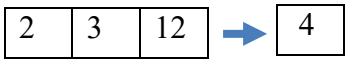
Le *deque* peut contenir l'entier 12 tout en respectant le tri du *deque*. 12 est placé en dernier.

`placerEntier(2)`



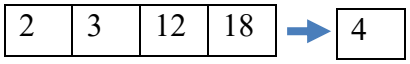

Le *deque* peut contenir l'entier 2 tout en respectant le tri du *deque*, 2 est placé en premier.

`placerEntier(4)`



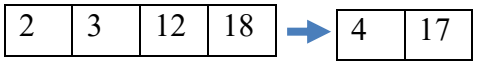

Le *deque* ne peut contenir l'entier 4. Le tri n'est pas respecté si on ajoute 4 en premier ou en dernier. Un nouveau *deque* avec 4 est placé en fin de liste.

`placerEntier(18)`



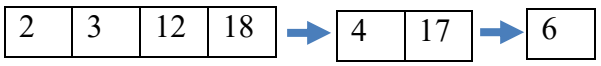

Le 1er *deque* peut contenir l'entier 18 tout en respectant le tri du *deque*. 18 est placé en dernier.

`placerEntier(17)`



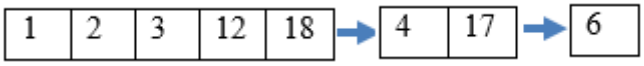

Le 1er *deque* ne peut contenir l'entier 17 tout en respectant le tri du *deque*.
Le 2ème *deque* peut contenir l'entier 17 tout en respectant le tri du *deque*. 17 y est placé en dernier.

`placerEntier(6)`



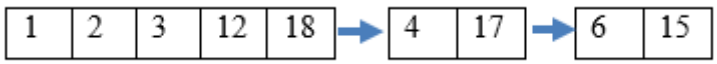

Le 1er *deque* ne peut contenir l'entier 6 tout en respectant le tri du *deque*.
Le 2ème *deque* ne peut contenir l'entier 6 tout en respectant le tri du *deque*.
Un nouveau *deque* avec l'entier 6 est placé en fin de liste.

`placerEntier(1)`



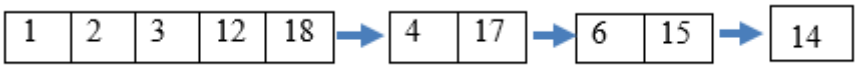

Le 1er *deque* peut contenir l'entier 1 tout en respectant le tri du *deque*.

`placerEntier(15)`




Le 1er *deque* ne peut contenir l'entier 15 tout en respectant le tri du *deque*.
Le 2ème *deque* ne peut contenir l'entier 15 tout en respectant le tri du *deque*.
Le 3ème *deque* peut contenir 15 tout en respectant le tri du *deque*. 15 y est placé en dernier.

`placerEntier(14)`




Le 1er *deque* ne peut contenir l'entier 14 tout en respectant le tri du *deque*.
Le 2ème *deque* ne peut contenir l'entier 14 tout en respectant le tri du *deque*.
Le 3ème *deque* ne peut contenir l'entier 14 tout en respectant le tri du *deque*.
Un nouveau *deque* avec l'entier 14 est placé en fin de liste.

Deuxième étape :

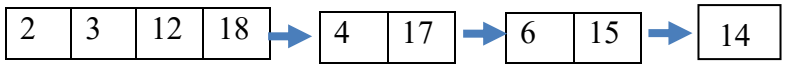
Tant que la liste des *deques* n'est pas vide, le premier entier du premier *deque* est retiré et placé dans la table à renvoyer.
Compte tenu du tri imposé, cet entier sera toujours le plus petit entier restant dans la liste.

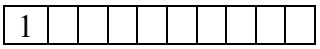
Après chaque suppression, la liste doit être revue :

- Lorsqu'un *deque* est vide, il est retiré de la liste.
- Le premier *deque* devra peut-être être retiré et réinséré au bon endroit dans la liste. La liste doit toujours être triée en utilisant le premier élément de chaque *deque* comme clef de tri.

Exemple suite :

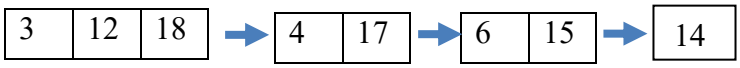
Suppression 1 :

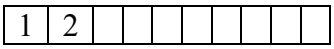



1 est placé dans la table et retiré du premier *deque*.
La liste ne doit pas être revue. Le tri imposé est vérifié.

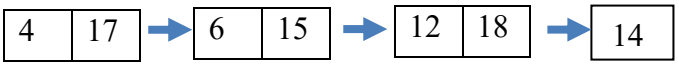
Suppression 2 :

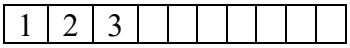



2 est placé dans la table et retiré du premier *deque*.
La liste ne doit pas être revue. Le tri imposé est vérifié.

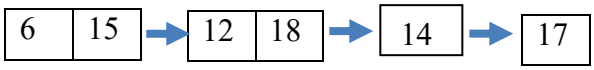
Suppression 3 :

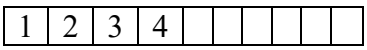



3 est placé dans la table et retiré du premier *deque*.
Pour respecter le tri, ce premier *deque* doit être déplacé.

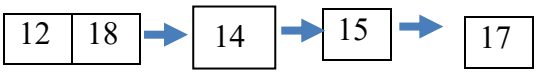
Suppression 4 :

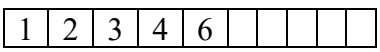



4 est placé dans la table et retiré du premier *deque*.
Pour respecter le tri, ce premier *deque* doit être déplacé.

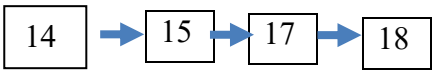
Suppression 6 :




6 est placé dans la table et retiré du premier *deque*.
Pour respecter le tri, ce premier *deque* doit être déplacé.

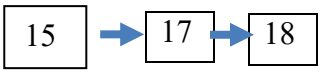
Suppression 12 :




| 1 | 2 | 3 | 4 | 6 | 12 | | | | |
|---|---|---|---|---|---|---|---|---|---|

12 est placé dans la table et retiré du premier *deque*.
Pour respecter le tri, ce premier *deque* doit être déplacé.

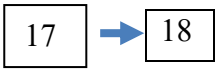
Suppression 14 :




| 1 | 2 | 3 | 4 | 6 | 12 | 14 | | | |
|---|---|---|---|---|---|---|---|---|---|

14 est placé dans la table et retiré du premier *deque*.
Le *deque* qui contenait 14 est vide. Il a été supprimé.
La liste ne doit pas être revue.

Suppression 15 :




| 1 | 2 | 3 | 4 | 6 | 12 | 14 | 15 | | |
|---|---|---|---|---|---|---|---|---|---|

15 est placé dans la table et retiré du premier *deque*.
Le *deque* qui contenait 15 est vide. Il a été supprimé.
La liste ne doit pas être revue.

Suppression 17 :


| 1 | 2 | 3 | 4 | 6 | 12 | 14 | 15 | 17 | |
|---|---|---|---|---|---|---|---|---|---|

17 est placé dans la table et retiré du premier *deque*.
Le *deque* qui contenait 17 est vide. Il a été supprimé.
La liste ne doit pas être revue.

Suppression 18 :

| 1 | 2 | 3 | 4 | 6 | 12 | 14 | 15 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|

18 est placé dans la table et retiré du premier *deque*.
Le *deque* qui contenait 18 est vide. Il a été supprimé.
La liste est vide et la table triée !

Complétez la classe *UnshuffleSort*.

La classe *TestEtape1UnshuffleSort* permet de vérifier la méthode `placerEntier()` utilisée lors de l'étape1.

La classe *TestUnshuffleSort* permet de vérifier votre méthode de tri.

Pour débugger votre programme, nous vous suggérons de placer l'instruction « `System.out.println(listeDeDeques) ;` » dans 2 méthodes de la classe *UnshuffleSort*.
Si votre méthode de tri est au point, voici les affichages attendus :

```
*******************************
Tests de la classe UnshuffleSort
*******************************
Test 1 : table a trier vide
etape1
etape2
Test 1 ok

Test 2 : table a trier avec 1 element
etape1
[[1]]
etape2
[]
Test 2 ok

Test 3 : table a trier de 6 entiers - 1 seul deque
etape1
[[4]]
[[3, 4]]
[[3, 4, 5]]
[[2, 3, 4, 5]]
[[1, 2, 3, 4, 5]]
[[1, 2, 3, 4, 5, 6]]
etape2
[[2, 3, 4, 5, 6]]
[[3, 4, 5, 6]]
[[4, 5, 6]]
[[5, 6]]
[[6]]
[]
Test 3 ok

Test 4 : la table a trier est celle de l'enonce
etape1
[[3]]
```

```
[[3, 12]]
[[2, 3, 12]]
[[2, 3, 12], [4]]
[[2, 3, 12, 18], [4]]
[[2, 3, 12, 18], [4, 17]]
[[2, 3, 12, 18], [4, 17], [6]]
[[1, 2, 3, 12, 18], [4, 17], [6]]
[[1, 2, 3, 12, 18], [4, 17], [6, 15]]
[[1, 2, 3, 12, 18], [4, 17], [6, 15], [14]]
etape2
[[2, 3, 12, 18], [4, 17], [6, 15], [14]]
[[3, 12, 18], [4, 17], [6, 15], [14]]
[[4, 17], [6, 15], [12, 18], [14]]
[[6, 15], [12, 18], [14], [17]]
[[12, 18], [14], [15], [17]]
[[14], [15], [17], [18]]
[[15], [17], [18]]
[[17], [18]]
[[18]]
[]
Test 4 ok

Test 5 : la table a trier contient des ex-aequos : 4 4 7 2 4
etape1
[[4]]
[[4, 4]]
[[4, 4, 7]]
[[2, 4, 4, 7]]
[[2, 4, 4, 7], [4]]
etape2
[[4, 4, 7], [4]]
[[4, 7], [4]]
[[4], [7]]
[[7]]
[]
Test 5 ok


Tous les tests ont reussi!
```