Les listes

Exercices obligatoires

A Exercice d'entraînement au parcours itératif d'une liste chaînée

Faites le *codeRunner Liste - parcours itératifs* proposé sur moodle.
Le document *codeRunner* vous donne des conseils d'utilisation et donne le contenu des listes testées.

B Implémentation de l'interface *ListeSimple* via une liste chaînée

B1 Complétez la classe Java *ListeSimpleImpl*.
Pensez à faire des schémas !!!
Cette classe implémente l'interface *ListeSimple*. Référez-vous à la JavaDoc de cette interface.
Testez cette classe avec la classe *TestListeSimpleImpl*.

Voici quelques indications concernant l'itérateur :

Un itérateur permet de parcourir les données qui se trouvent dans une structure de données.
Dans le cas de la liste, les éléments seront parcourus en respectant l'ordre.
(Et ceci, sans oublier le premier !)

L'interface *ListeSimple*<*E*> étend l'interface *Iterable*<*E*> qui impose la méthode `iterator()`. Ceci permet l'utilisation du `foreach` !

La méthode `iterator()` renvoie un objet de type *Iterator*<*E*>

L'interface *Iterator*<*E*> impose les méthodes `hasNext()`, `next()` et `remove()`.
Pour plus de détails concernant ces méthodes, suivez le lien :

https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/Iterator.html

La méthode `iterator()` construit un objet de type *Iterator*<*E*>.
Il est donc nécessaire d'implémenter l'interface *Iterator*<*E*>.
La classe qui implémente cette interface sera une classe interne. Elle doit pouvoir accéder à la liste !

B2 Sur moodle, répondez au questionnaire à choix multiples *ListeSimpleImpl*.

C Application Gestion des séries

On vous demande d'écrire une application qui va faciliter la tâche de la personne chargée de répartir les étudiants dans les différentes séries.

Nous n'allons pas introduire de classe *Etudiant*. Pour chaque étudiant, on ne retiendra que son nom (*String*).

Vous allez compléter 3 classes.

La classe *SerieEtudiants.*
Une série d'étudiants possède un numéro et une liste d'étudiants.
Pour la liste, vous utiliserez un objet de la classe *LinkedList* de l'API Java.

La classe *ClasseEtudiants*.
La classe compte plusieurs séries d'étudiants. Celles-ci sont placées dans une table.
.
La classe *GestionDesSeries*.
C'est cette classe qui contient la méthode `main()`.Celle-ci-ci propose un menu.
.

On ne vous demande pas d'écrire des classes de tests.
Mais, testez bien votre application en utilisant la classe *GestionDesSeries*.

Complétez les tables suivantes en donnant les coûts :

SerieEtudiants :

METHODE	COUT
`nombreEtudiants()`	
`contientEtudiant(String nom)`	
`ajouterEtudiant(String nom)`	
`supprimerEtudiant(String nom)`	
`toString()`	

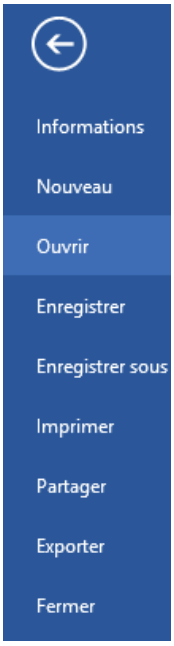
ClasseEtudiants :

METHODE	COUT
`getSerieEtudiants(int numeroSerie)`	
`numeroSerie(String nom)`	
`changerSerie(String nom, int nouveauNumeroSerie)`	
`toString()`	

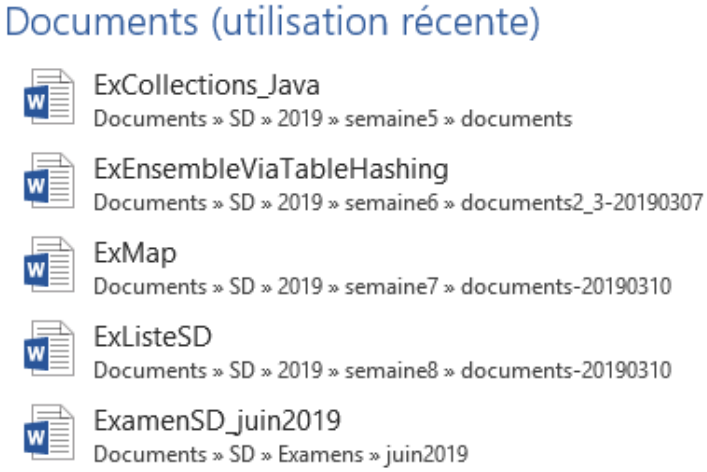
Les solutions se trouvent sur moodle dans le dossier solutions. Le document s'appelle *CSol.*

D Application DocumentsLRU

Le menu *ouvrir* de Word présente les documents récemment utilisés.

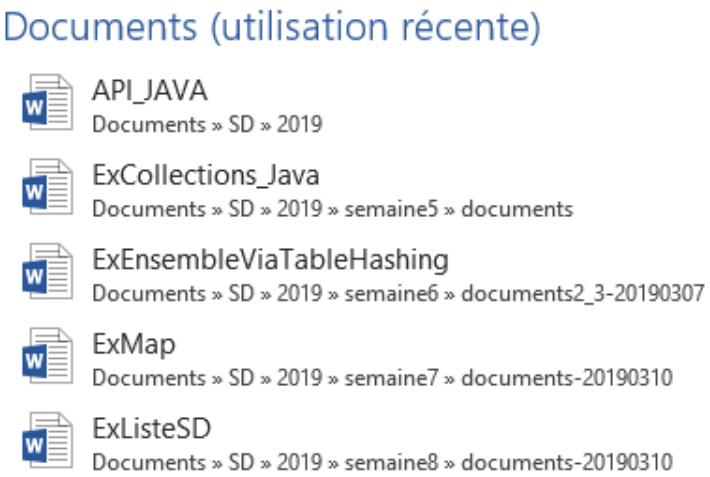


Exemple :



L'ordre d'apparition est important. C'est le document *ExCollections_Java* qui apparait en premier, car il est le document qui a été ouvert le plus récemment. Ensuite c'est *ExEnsembleViaTableHashing*, l'avant dernier à avoir été ouvert et ainsi de suite.

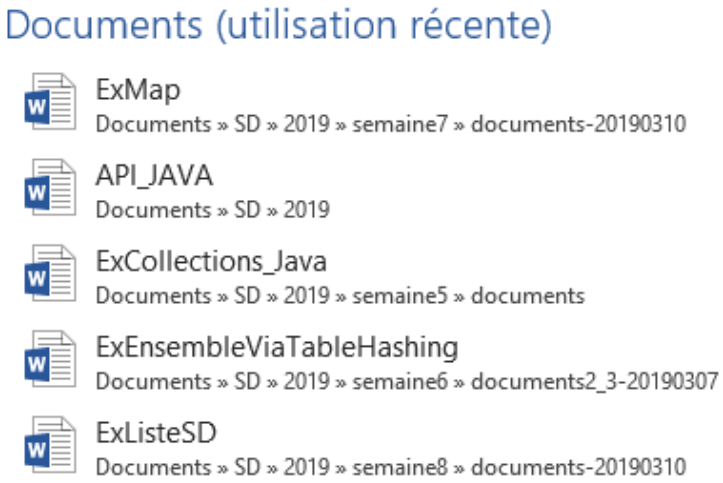
Le fait d'ouvrir un document qui n'était pas encore présent dans la liste a comme conséquence que celui-ci apparaît maintenant en premier lieu (MRU – Most Recently Used) et que le dernier document de la liste a disparu.
Ce document est celui qui a été le moins récemment utilisé (LRU – Least Recently Used).



Dans l'exemple, le document *API_JAVA* vient d'être ouvert. Il apparaît en premier lieu.
Le document *ExCollections_Java* apparait maintenant en deuxième, …
Le document *ExamenSD_juin2019* n'apparait plus.

Si on ouvre un document qui se trouve déjà dans la liste, il est déplacé en premier.

Il devient le document le plus récemment utilisé.



Dans l'exemple, le document *ExMap* vient d'être ouvert. Il apparaît en premier lieu.

Au cours de SD, nous allons nous intéresser à la structure de données qui permet d'optimiser le mécanisme de **LRU**.

Version semaine 4 :

La structure de données utilisée est une liste de documents (String).

Cette liste est préremplie avec autant de documents que nécessaires pour atteindre la taille que l'on a décidé préalablement et qu'on ne peut pas dépasser.
Comme, au départ, il n'y a pas de document, la liste contiendra des documents bidon aisément reconnaissables.

Lorsqu'on ouvre un document qui ne se trouve pas dans la liste, il est placé en début de liste et il faut retirer celui qui se trouve en fin de liste.
Lorsqu'on ouvre un document se trouvant dans la liste, celui-ci devient le plus récent.
Il faut donc le déplacer en tête.
La tête de la liste contient donc toujours le document le plus récemment utilisé (Most Rencently Used).

D1 Exercice préliminaire :

Après chaque ouverture d'un document, donnez le contenu de la liste.

```
Au départ :
 doc1 doc2 doc3 doc4 doc5

ouvrir doc3


ouvrir doc4


ouvrir doc4


ouvrir doc5


ouvrir doc6


ouvrir doc3


ouvrir doc6


ouvrir doc7
```

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

D2 Implémentation :

Nous vous demandons de compléter la classe *DocumentsLRU*.

Cette classe possède une liste de documents (String).
Pour la liste, vous utiliserez la classe *LinkedList* de l'API Java.

Suivez bien la *JavaDoc*.

Le jeu de tests de la classe *TestDocumentsLRU* correspond à l'exercice préliminaire.

D3 Remplissez la table suivante :

METHODE	**COUT**
`ouvrirDocument()`	
`toString()`	

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

E Encore quelques parcours de liste

E1 Complétez la classe *ListeCaracteres*.
Toutes ces méthodes doivent avoir au maximum un ordre de complexité O(N).
La classe *TestListeCaracteres*, via un menu, permet de tester ces méthodes.

Exercice ❤️ (à ne pas soumettre)

B3 Le bon sens nous demande de ne pas modifier une liste tout en la parcourant via un itérateur.
Par sécurité, une exception de type *ConcurrentModificationException* pourrait être levée par la méthode `next()` si la liste a été modifiée, autrement que via l'itérateur, depuis le moment où l'itérateur a été créé.
Une façon de gérer ce problème est d'introduire un numéro de version de la liste.
Chaque fois que la liste est modifiée, le numéro de version change.
Il suffit de s'assurer que ce numéro de version n'a pas changé lors de chaque appel à la méthode `next()`.
Testez cette classe avec la classe *TestListeSimpleImplSuite.*

Exercices défis

B4 Acceptez les suppressions dans l'itérateur.

E2 Ecrivez les méthodes de la classe *ListeCaracteres* mises en défi.