TP 3

Manipulation sur des mots.

Le but de cet exercice est :

▷ d'apprendre à écrire une classe,

▷ d'utiliser une méthodologie rigoureuse basée sur les tests unitaires,

▷ de savoir exploiter la documentation d'une classe pour l'utiliser correctement (ici la classe `String`).

d'apprendre à réutiliser des classes existantes en s'appuyant sur la documentation disponible. De plus, les classes que vous allez découvrir sont souvent utilisées.

Nous cherchons dans cet exercice à modéliser des mots sur lesquels nous ferons un certain nombre d'opérations.

Documentation. Pour pouvoir implémenter la classe décrite ci-dessous, vous devrez utiliser les fonctionnalités de la classe `String`[1].
La classe `String` permet de représenter des chaînes de caractères non mutables, aucune méthode de la classe ne permet de modifier la chaîne manipulée.
N'oubliez pas cependant que les chaînes de caractères sont des objets !

On va utiliser la documentation de la classe `String`. Cette documentation est volumineuse. Pour éviter que vous ne vous perdiez dans le grand nombre de méthodes disponibles, pour chaque question il sera indiqué les (nouvelles) méthodes dont vous devriez consulter la documentation avant de chercher à répondre à la question car elles devraient vous aider. Il est cependant sans doute possible de trouver des solutions qui n'utilisent pas ces méthodes, n'en faites donc pas une fixation.
Ainsi, dans la suite, l'information :
`(dans NomDeClasse) nomMethode`
signifie : "*consultez la documentation pour la méthode de nom* ***`nomMethode`*** *dans la classe* ***`NomDeClasse`***"
Cela n'implique pas que vous deviez obligatoirement utiliser cette méthode pour traiter la question, mais qu'il est possible qu'elle vous soit utile pour cette question (et/ou une suivante).

Méthodologie. Une fois l'analyse effectuée on arrive à la phase d'écritude du code dans le langage choixi("codage"). Pour produire efficacement du code fiable et correct il faut respecter une démarche rigoureuse.

Il faut travailler une méthode à la fois en appliquant la démarche suivante :

1. **écrire la signature de la méthode,**
2. **écrire la documentation (javadoc) de la méthode,**
3. **écrire les tests qui permettront de vérifier que le code produit pour la méthode est correct,**
4. **écrire le code,**
5. **exécuter les tests prévus à l'étape 3, en vérifiant que les tests des méthodes précédemment écrites (et testées) restent réussis**[a]**,**
6. **si les tests sont réussis passer à la méthode suivante (étape 1) sinon recommencer à l'étape 4.**

[a] On s'assure que le nouveau code écrit ne remet pas en cause les codes précédents.

Pour ce premier TP, le squelette de la classe `Word` vous est fourni dans le fichier `Word.java`, ainsi que deux classes pour les tests `WordTest.java` et `WordTest1.java`. Vous n'aurez donc pas pour cette fois à créer ces tests, mais à les comprendre et les exécuter.
Vous devez récupérer sur le portail et placer dans un même répertoire les fichiers :

- `Word.java`, qui est le fichier de travail et donc est à compléter,

[1] Un certain nombre des méthodes demandées pourraient être simplifiées si l'on utilisait les fonctionnalités offertes par la classe `StringBuffer` du paquetage `java.lang`. Cependant on ne les utilisera pas a priori dans ce TP. Vous êtes cependant fortement invités à regarder la documentation et à réfléchir à comment vous auriez pu utiliser cette classe - par exemple pour la méthode `inverse`, il y a ce qu'il faut dans `StringBuffer` pour peu que l'on comprenne – ce qui n'est pas très compliqué – comment passer d'une instance de `String` à une instance "équivalente" de `StringBuffer` et réciproquement.

- le fichier de tests `WordTest1.java` à étudier et sa version compilée `WordTest1.class`[2] à exécuter,
- le fichier de tests `WordTest.java` à étudier et sa version compilée `WordTest.class` à exécuter,
- le fichier `test-1.7.jar` qui permet l'exécution des tests.

Premiers tests.

Q 1 . Dans un éditeur, ouvrez le fichier `Word.java` qui définit la classe `Word`.

L'état des objets de cette classe est représenté par l'attribut `value`, une chaîne de caractères. Le constructeur prend en paramètre une chaîne de caractères pour initialiser l'attribut.

> **Attention :** il est important par la suite de ne pas confondre les objets la classe `Word` et leur attribut `value`. Dans la suite du sujet quand on parlera d'un mot, il faudra comprendre *une instance de la classe* `Word`. Cependant il est probable que la plupart des traitements se feront sur l'attribut `value` qui est une chaîne de caractères.

Vous devez donc être vigilant, rigoureux et attentif et bien réfléchir à la notion que vous référencez/manipulez : l'objet `Word` ou son attribut `value`.

Les constantes chaînes de caractères seront notées entre `"`.

Les méthodes demandées dans les questions qui suivent sont des méthodes de cette classe `Word`. L'écriture de certaines est triviale, pour d'autres un peu plus de réflexion sera nécessaire.

Q 2 . La documentation et le code de la méthode `equals` est fourni. Consultez-le puis compilez la classe `Word`.

Q 3 . La documentation peut être générée à l'aide de l'outil `javadoc`. Dans le répertoire contenant le fichier `Word.java` créez un dossier `docs`, puis dans le terminal exécutez depuis ce répertoire la commande `javadoc Word.java -d docs`. L'option `-d docs` permet dans placer les fichier créé dans le dossier `docs`. Consultez le contenu de ce dossier, en particulier ouvrez le fichier `index.html` qu'il contient dans un navigateur. Dans la page affichée cliquez sur les liens proposés par le constructeur et la méthode `equals` et faites le lien entre les informations affichées et le code de documentation qui apparaît dans le fichier source `Word.java`.

Dans la suite vous aurez à écrire des documentations qui respecteront ces formats.

Q 4 . Consultez le contenu du fichier de test `WordTest1.java`.

Il n'y a qu'une méthode de test `testEquals`. Elle contient trois assertions de test identifiées par les instructions `assertTrue` et `assertFalse`. On comprend facilement que la première assertion est vérifiée si l'expression paramètre vaut `true` et les deux autres si elle vaut `false`. Un test est réussi quand toutes les assertions qu'il contient sont vérifiées.

> L'objectif d'une telle méthode de test est de vérifier la correction de l'implémentation de la méthode `equals`. Elle constitue une **spécification exécutable** de la méthode `equals` : on considère qu'une implémentation de la méthode `equals` est correcte dès qu'elle passe avec succès ce test. Evidemment cela fait l'hypothèse que ces tests sont corrects et suffisamment complet pour considérer les différentes situations possibles. Cela explique les différentes assertions dans la méthode `testEquals` : la première pour vérifier le cas « positif », les deux autres pour considérer deux cas négatifs : quand le paramètre `o` est une instance de `Word` ou non.
> Il est possible, et parfois plus simple voire nécessaire, d'écrire plusieurs méthodes de test pour tester une seule méthode (ce sera le cas pour la méthode `extractBefore` dans `WordTest`).
> La donnée de la documentation et d'une ou plusieurs méthodes de test doit suffire pour réaliser l'implémentation correcte d'une méthode.

Q 5 . Exécutez les tests `WordTest1` grâce à la commande (la compilation préalable des classes `Word` et `WordTest1` est nécessaire si les fichiers `.class` n'existent pas) :

```
java -jar test-1.7.jar WordTest1
```

Une fenêtre s'ouvre. Vous y trouvez une barre verte qui signifie que tous les tests ont été passés avec succès. Ce qui est confirmé par les indications fournies juste dessous :

- le nombre de tests exécutés (ici 1), celui de la seule méthode de test `testEquals`,
- le nombre d'erreurs (ici 0),

Q 6 . Consultez le fichier de tests `WordTest` et exécutez les tests qu'il contient :

```
java -jar test-1.7.jar WordTest
```

[2] Ce fichier comilé peut être obtenu par la commande `javac -classpath .:test-1.7.jar WordTest1.java`.

Cette fois la barre est rouge car il y a des échecs dans les tests et on constate qu'il y a 9 erreurs sur 10 tests. La fenêtre Results mentionne les erreurs et on peut y lire les méthodes de test qui posent problème. Ici, toute sauf `testEquals`.

Pour la suite l'objectif est "*obtenir une barre verte*"

NB : pour réaliser ces tests unitaires nous utilisons la bibliothèque `JUnit4` qui est très largement utilisée et fait référence pour les tests unitaires en java. Les tests tels que vous les apprenez sont des tests `JUnit4`.

Cependant le fichier `test-1.7.jar` en est une adaptation locale pour vous permettre de disposer de la fenêtre graphique des résultats.

Ecriture des méthodes.
Dans la suite du TP l'objectif est d'avoir un test en erreur de moins après chaque question et donc à la fin du TP d'obtenir une barre verte qui indique que tous les tests ont été passés avec succès.
Pour l'évaluation de votre TP, ce test sera exécuté !

Dans les questions suivantes, pour chacune des méthodes à implémenter, vous commencerez par étudier la ou les méthodes de test qui lui sont associées dans `WordTest` puis par en écrire la documentation avant de procéder à l'implémentation. Vous validerez votre implémentation en vérifiant que le test correspondant est réussi.
Attention : **Il est nécessaire de recompiler le code source après chaque modification pour que celle-ci soit prise en compte.**

Q 7 . Définissez une méthode `nbOfChars`, sans paramètre, qui a pour résultat le nombre de caractères (la longueur) du mot.

(dans `String`) `int length()`, [3]

Q 8 . Définissez une méthode `toString`[4] qui renvoie une chaîne de caractères correspondant au mot.

Q 9 . Réaliser un "exécutable" :

Comme vue dans la partie 1 du TP, il faut ajouter à votre classe `Word` une méthode `main`[5] qui :

1. déclare une référence de type `Word`,
2. initialise cette référence par une instance de la classe `Word` créée à partir de la valeur du premier argument passée en ligne de commande (`args[0]`),
3. invoque la méthode `nbOfChars()` sur cette référence et affiche le résultat (utilisez `System.out.println`).
4. affiche le résultat de l'invocation de la méthode `toString` sur cet objet.

Après compilation exécutez votre programme par : "`java Word` *`uneChainePourDefinirLeWord`*" (par exemple `java Word timoleon`).

Pour chacune des méthodes suivantes, une fois les tests réussis, vous pourrez compléter cette méthode `main`, pour invoquer les différentes méthodes. le travail que vous rendrez devra contenir une telle méthode avec les différents appels.

Q 10 . Définissez une méthode, `nbOccurrencesOfChar` qui calcule le nombre d'occurrences d'un caractère donné (en paramètre) dans le mot.

(dans `String`) `int indexOf(char c)`,
(dans `String`) `int indexOf(char c, int fromIndex)`
(dans `String`) `char charAt(int index)`,

Q 11 . Après en avoir écrit la documentation, définissez une méthode `reverse` qui retourne un objet `Word` (pas `String` !) dont la valeur (l'attribut) est l'inverse de la valeur du mot initial (c-à-d. du mot sur lequel cette méthode a été appelée).

---

[3] à ne pas confondre avec la propriété `length` des tableaux !

[4] Cette méthode est automatiquement utilisée lors de l'affichage par `System.out.println`. Ainsi si `m` est une référence de type `Word` initialisée avec une instance de cette classe, les instructions `System.out.println(m.toString());` ou `System.out.println(m);` sont équivalentes. Dans la seconde, l'invocation de la méthode `toString` est implicite et automatiquement gérée par Java.

[5] Attention à la signature de cette méthode qui doit être scrupuleusement respectée !

Q 12 . Définissez une méthode `contains` qui vérifie si un mot donné est un "sous-mot" du mot courant.

Par exemple les mots correspondants à `tim`, `mole` et `leon` sont des sous-mots du mot `timoleon`. Ce n'est pas le cas des mots `ile` ou `hobbit`.

(dans `String`) `int indexOf(String str),`
(dans `String`) `int indexOf(String str, int fromIndex)`
(dans `String`) `String substring(int beginIndex),`
(dans `String`) `String substring(int beginIndex, int endIndex)`

Q 13 . Définissez une méthode `rhymesWith` (*rime avec*), qui regarde si le mot rime avec un autre mot. On dira que deux mots riment si leur 3 derniers caractères sont identiques (il faut donc au moins 3 caractères, sinon le résultat vaut forcément faux...).

(dans `String`) `boolean endsWith(String suffix)`

Q 14 . Définissez une méthode `extractBefore` qui retourne un tableau de deux mots, le premier correspond au plus petit préfixe du mot précédant la première occurrence d'un caractère donné (passé en paramètre) inclus, le second correspond au reste du mot (sans le caractère). Si le caractère n'existe pas dans le mot, le premier élément du tableau résultat est le mot vide et le second est le mot lui-même.

Par exemple pour le mot `timoleon` et le caractère `o`, on veut obtenir `timo` et `leon`, alors que pour le caractère `i` on obtient `ti` et `moleon`.

Q 15 . Définissez une méthode `isPalindrom` qui teste si le mot est un palindrome. Une telle méthode renvoie évidemment un booléen.

Q 16 . Définissez une méthode `isAnagram` qui indique si un mot donné est un anagramme du mot (les deux mots ont les mêmes caractères en même nombre).

Q 17 . Définissez une méthode qui indique si un nom est un nom propre (*proper noun*), c'est-à-dire commence par une majuscule.

(dans `Character`) `static boolean isUpperCase(char ch)`

NB : "upper-case" signifie "lettre majuscule" en anglais...