TME 4 : Implantation d'une structure de multi-ensembles

Objectifs pédagogiques :

- création d'une nouvelle collection : les multi-ensembles ;
- utilisation de la collection `Map` ;
- création d'un itérateur.

But

Les collections de la bibliothèque standard Java proposent de nombreuses interfaces de conteneurs et plusieurs implantations de chaque interface. Dans ce TME, nous allons construire un nouveau type de collection : les multi-ensembles, `MultiSet`. Le prochain TME reprendra cette collection et nous proposera de développer des tests.

Un multi-ensemble est une liste non-ordonnée d'éléments. Contrairement à un ensemble classique, un élément peut apparaître plusieurs fois dans un multi-ensemble, et il convient de se souvenir du nombre d'occurrences. Une application intéressante des multi-ensembles que nous allons explorer dans ce TME est le comptage de la fréquence d'apparition des mots dans un texte pour déterminer les mots les plus fréquents.

Plusieurs implantations des multi-ensembles sont possibles. Une implantation naïve consisterait à utiliser une implantation de l'interface `List` (par exemple `ArrayList`), mais celles-ci ne possèdent pas une bonne complexité pour notre application (le comptage de mots). Nous allons à la place utiliser une implantation de l'interface `Map`, qui associe à chaque élément son nombre d'occurrences. Notez que nous ne pouvons pas utiliser `Set`, car il ne permet pas à un élément d'apparaître plusieurs fois.

Notre multi-ensemble est polymorphe. Comme pour les collections Java, nous utilisons les types génériques. Nous écrirons donc `MultiSet<T>`.

4.1 Mise en place du TME

Nous allons travailler au cours des TME 4 et 5 dans un nouveau projet du serveur GitLab nommé `MultiSet` qui donne, après import, un projet Eclipse `MultiSet` sur votre machine locale.

Pour mettre en place le TME, nous allons répéter les opérations effectuées au début du TME 1 et décrites dans le document *Mise en place et rendu des TME* sur la page Moodle du cours. Nous les rappelons succinctement :

1. **Fork du squelette de projet GitLab.**
   Connectez-vous à GitLab et sélectionnez le menu « Your work > Projects ».
   Vous trouverez dans le groupe `POBJ-XXXX` (où `XXXX` est le semestre en cours, par exemple `POBJ-2023oct`) un nouveau projet, `MultiSet`, auquel vous pouvez accéder en lecture seule.
   Un des membres du binôme fait une copie privée en sélectionnant le projet et en cliquant sur « Forks » en haut à droite. Sur la page, ouvrez le menu « Select a namespace » et sélectionnez votre *username*. Sur la page de ce projet, ajoutez votre binôme et votre chargé de TME avec l'option « Manage > Members » en leur donnant le rôle « Maintainer ».
2. **Importation (clone) du projet dans Eclipse.**
   Lancez Eclipse et importez le projet localement. Dans Eclipse, choisissez dans le menu déroulant : « File > Import... > Git > Projects from Git » puis « Next ». Dans l'écran suivant, choisissez l'option « Clone URI » puis « Next ». Dans le champ « URI » de l'écran suivant, entrez l'URI du dépôt git du projet tel que trouvé dans le champ « Clone with HTTPS » sous GitLab : `https://stl.algo-prog.info/`*`username`*`/MultiSet.git` (où *username* est le numéro de l'étudiant qui a fait le *fork*). La fenêtre se remplit automatiquement. Donnez comme nom votre numéro d'étudiant et le mot de passe associé dans GitLab. Cliquez sur « Next ». Les écrans suivants, « Branch Selection » et « Local Destination », sont pré-remplis ; cliquez sur

« Next ». Choisissez « Import existing Eclipse projects » et « Next », puis « Finish ».

Dans ce TME, nous travaillons dans le package `pobj.multiset`.

4.2 Opérations de base

Un multi-ensemble contenant des éléments de classe `T` obéira à l'interface suivante :

```
package pobj.multiset;

public interface MultiSet<T> {
   public boolean add(T e, int count);
   public boolean add(T e);
   public boolean remove(Object e);
   public boolean remove(Object e, int count);
   public int count(T o);
   public void clear();
   public int size();
}
```

disponible dans le squelette de projet, où :

- `add(e,count)` et `remove(e,count)` ajoutent et enlèvent respectivement `count` occurrences de l'objet `e` ;
- `add(e)` et `remove(e)` sont des raccourcis correspondant au cas fréquent où `count` = 1 (une seule occurrence est ajoutée ou enlevée) ;
- les deux versions de `add` et de `remove` retournent un booléen indiquant si le multi-ensemble a été modifié par l'opération (`true`) ou non (`false`) ; pour un multi-ensemble, tout ajout d'un nombre non-nul d'occurrences modifie la collection, tandis qu'une suppression d'un nombre non-nul d'occurrences modifie la collection si celle-ci contient au moins une occurrence de l'élément ;
- `count(e)` indique le nombre d'occurrences de l'objet `e` dans la collection (0 si l'objet n'est pas dans le multi-ensemble) ;
- `size()` indique la taille du multi-ensemble, c'est-à-dire la somme des nombres d'occurrences des éléments ;
- `clear()` vide le multi-ensemble.

Notez le choix des noms de méthode (`add`, `remove`, `size`, `clear`), le booléen retourné par les méthodes d'ajout et de suppression, et le fait que `remove` prenne un objet de classe `Object` et non `T`. Ces choix sont motivés par le désir d'être aussi compatible que possible avec l'interface standard `Collection` de Java (vue en cours), ce qui nous servira dans les questions suivantes.

Nous allons créer une implantation concrète de cette interface, la classe `HashMultiSet<T>`, qui :

- s'appuie par délégation sur une `HashMap<T,Integer>` associant à un élément apparaissant au moins une fois dans le multi-ensemble un nombre d'occurrences strictement positif ;
- maintient dans un attribut `size` séparé la taille du multi-ensemble, ce qui permettra d'implanter la méthode `size()` en temps constant sans avoir à parcourir la table.

N'oubliez pas qu'il vaut mieux programmer le plus possible vis à vis d'une interface, donc ici `Map`, pour minimiser les dépendances aux classes concrètes, ici `HashMap`.

⇒ **Travail demandé :** Programmez la classe `HashMultiSet<T>` implantant l'interface `MultiSet<T>` dans le package `pobj.multiset` à l'aide d'une `HashMap<T,Integer>`. Cette classe aura deux constructeurs :

1. un constructeur sans argument, qui construit un multi-ensemble vide ;
2. un constructeur de copie, prenant un objet `Collection<T>` en argument et initialisant le nouveau multi-ensemble avec le contenu de la collection (voir encadrés suivants).

Une classe de test `pobj.multiset.test.HashMultiSetTest` est fournie dans le projet `MultiSet`. Notez qu'elle est, à dessein, limitée à tester quelques cas simples. L'enrichissement de la banque de tests fera l'objet du TME 5.

`Map<K,V>` EN JAVA. `java.util.Map` est une interface de la bibliothèque standard Java correspondant à une collection de type dictionnaire, associant des valeurs (de type `V`) à des clés (de type `K`). Les méthodes importantes de l'interface `Map` sont :

- `V put(K key, V value)`, qui associe la valeur à la clé, et retourne la valeur précédemment associée (ou `null` si la clé n'était pas déjà dans le dictionnaire). Une clé ne peut être associée qu'à une valeur au plus : `put` supprimera toute association préexistante avec la clé.
- `V remove(Object key)`, qui supprime la valeur éventuellement associée à la clé (et retourne cette valeur, ou `null` s'il n'y avait pas d'association).
- `V get(Object key)`, qui retourne la valeur associée à la clé, ou `null` s'il n'y a pas d'association.
- `boolean containsKey(Object key)`, qui retourne `true` si le dictionnaire contient une valeur associée à la clé.
- `void clear()`, qui vide complètement le dictionnaire.
- `int size()`, qui retourne le nombre d'associations dans le dictionnaire.

Le type générique `Map<K,V>` indique que les clés sont forcément de classe `K` (ou dérivée), et les données de classe `V` (ou dérivée). Ces informations servent à préciser le type de certaines opérations. Ainsi, `put` demande un objet `key` de type `K` et un objet `value` de type `V`, et retourne un objet de type `V`. Les opérations `get`, `containsKey` et `remove` sont plus laxistes : elles acceptent un objet arbitraire (`Object`). Plusieurs implantations obéissant à l'interface `Map<K,V>` existent dans la bibliothèque standard Java, offrant différentes complexités pour différentes opérations. Nous utiliserons ici `HashMap<K,V>`, qui est souvent un bon choix.

CLASSE `Integer`. Dans un dictionnaire `Map<K,V>`, le type `K` des clés et le type `V` des valeurs sont donnés par une classe (ou une interface). Or `int` est un type primitif, et pas un nom de classe ou d'interface : il est donc impossible d'écrire `Map<T,int>`. Nous remplaçons donc `int` par `Integer`, une classe de la bibliothèque standard Java qui « enrobe » un entier primitif dans un objet (attribut immuable), ce qui permet de l'utiliser dans un contexte où les types primitifs sont interdits. Convertir un `int` en `Integer` se fait grâce à la méthode statique `Integer valueof(int)` de `Integer`. Retrouver la valeur d'un `Integer` se fait grâce à la méthode `int intValue()`. Cela dit, l'appel à ces méthodes est souvent omis car le compilateur Java ajoute des conversions implicites entre `int` et `Integer` dès que nécessaire.

CONSTRUCTEUR DE COPIE. Le constructeur de copie demandé, `HashMultiSet(Collection<T>)`, permet de facilement convertir en multi-ensemble une collection arbitraire. Par exemple, si `list` obéit à l'interface `List<T>`, qui est une forme de collection, alors `new HashMultiSet<T>(list)` convertira une liste en un multi-ensemble. Nous verrons par ailleurs en question 4.4 comment étende `HashMultiSet` pour obéir à l'interface collection, ce qui permettra de copier une `HashMultiSet set` arbitraire par un appel à `new HashMultiSet<T>(set)`, et explique le terme *constructeur de copie*.
Les collections de la bibliothèque standard Java implantent elles-mêmes des constructeurs de copie. C'est attendu du contrat d'une collection, mais non spécifié dans l'interface `Collection` puisqu'une interface ne peut pas fournir de constructeur. Quand `HashMultiSet` implantera `Collection<T>`, en question 4.4, il sera possible, par exemple, de convertir facilement un multi-ensemble arbitraire `set` en liste par un simple appel à `new ArrayList<T>(set)`.

Alerte ***Unckeched cast.*** Il est tentant, voir nécessaire, dans `remove(Object elem)`, de commencer par convertir l'objet `elem` en type `T`. Normalement, un test `elem instanceof T` devrait précéder une telle conversion de type pour s'assurer que la conversion n'échouera pas. Néanmoins, il est impossible de faire un test `instanceof` vis à vis d'un paramètre `T` de classe générique, car la valeur exacte de `T` n'est pas disponible à l'exécution (ce point est détaillé au cours 7) ! Pour la même raison, l'exception `ClassCastException` ne sera pas signalée si `elem` n'est pas de type `T` lors de la conversion `(T)elem`. Pour ce type de conversion, non vérifiable, le compilateur Java indique un avertissement *Unchecked cast from Object to T*. Nous verrons les détails précis du typage des collections et des génériques Java dans un cours ultérieur. Il suffit pour l'instant de savoir que nous pouvons ignorer cet avertissement.

4.3 Itérateur

Toute collection a besoin d'être parcourue pour faire des traitements sur ses éléments. Cependant, les collections n'ont pas toutes la même structure interne (e.g. : un tableau, une liste chaînée, un arbre binaire, ...), ce qui implique que la manière de parcourir la collection va dépendre de son implantation. Pour éviter d'avoir un code client qui dépend de la structure interne de la collection, il existe un *Design Pattern* appelé *Iterator*. Un itérateur est un objet qui implante l'interface `java.util.Iterator<T>`, ce qui impose la définition de deux méthodes (voir l'encadré pour plus de détails) :

```
public boolean hasNext();
public T next();
```

Puisque chaque collection a un objet de type itérateur qui lui est propre, l'interface `Collection<T>` étend en fait l'interface Java `java.lang.Iterable<T>`, qui impose à toute collection de définir la méthode suivante permettant de construire un itérateur sur la collection :

```
public Iterator<T> iterator();
```

Un objet `Iterable<T> collection` bénéficie d'une boucle « for » améliorée :

```
for (T elem : collection) { action(elem); }
```

qui est en fait du « sucre syntaxique » pour le code suivant :

```
Iterator<T> iter = collection.iterator();
while (iter.hasNext()) {
   T elem = iter.next();
   action(elem);
}
```

Pour résumer :

- la classe collection implante l'interface `Iterable<T>` indiquant qu'elle peut créer des itérateurs sur les objets collection ;
- la classe itérateur sur une collection implante l'interface `Iterator<T>` indiquant qu'un objet itérateur permet d'énumérer les éléments, de type `T`, de l'objet collection qui l'a créé.

⇒ **Travail demandé :**

1. Définissez une classe `HashMultiSetIterator` qui implante `Iterator<T>` en respectant les spécifications suivantes (il est notamment conseillé de définir `HashMultiSetIterator` comme une classe interne à `HashMultiSet<T>`, voir encadré ci-dessous) :
   - Si un élément `e` apparaît n fois dans un multi-ensemble, alors l'itérateur retournera l'élément `e` pour les n prochains appels à `next()`, avant de passer à l'élément suivant.
   - Afin d'élaborer notre itérateur de multi-ensembles, nous nous appuierons par délégation sur un itérateur parcourant la collection `Map<T,Integer>` au cœur de l'implantation de `HashMultiSet<T>`.

     Attention : un appel à `next()` de l'itérateur de multi-ensemble ne se traduira pas systématiquement par un appel à `next()` de l'itérateur sous-jacent : il faut tenir compte des nombres d'occurrences. Considérons, par exemple le multi-ensemble à 7 éléments $\{a, a, a, b, b, b, c\}$. Celui-ci est encodé par le dictionnaire à 3 éléments $[a \mapsto 3, b \mapsto 3, c \mapsto 1]$. Une itération du dictionnaire retournera, avec les 3 premiers appels à `next` : $(a, 3)$, $(b, 3)$, $(c, 1)$ puis s'arrêtera. Une itération du multi-ensemble retournera, lors des 7 premiers appels à `next` : a, a, a, b, b, b, puis c, avant de s'arrêter.
2. Modifiez l'interface `MultiSet<T>` pour qu'elle étende `Iterable<T>`.
3. Enrichissez la classe `HashMultiSet<T>` afin qu'elle implante la méthode `iterator()` imposée par `Iterable<T>`. Cette méthode devra retourner une nouvelle instance de `HashMultiSetIterator`.
4. Assurez-vous que la classe de test `pobj.multiset.test.IteratorTest` valide votre itérateur.

Itérateur. L'interface `Iterator<T>` comporte en réalité trois méthodes :

- `boolean hasNext()`, qui retourne `true` tant qu'il existe des éléments à parcourir dans la collection.
- `T next()`, qui retourne le prochain élément dans la collection. Tant que `hasNext()` retourne vrai, `next()` retourne un élément ; sinon, `next()` signale une exception `NoSuchElementException`. `next()` a également pour effet de déplacer le curseur de la collection d'un cran, donc le prochain appel à `next()` retournera l'élément suivant.
- `void remove()`, qui modifie la collection en supprimant le dernier élément retourné par `next()`.

L'opération `remove` est cependant optionnelle. Il est possible (et c'est que nous conseillons de faire ici) d'omettre sa définition, auquel cas le comportement par défaut sera utilisé : signaler une exception `UnsupportedOperationException`.

Itérateur de clés et d'associations. Pour un objet `Map<K,V>`, un itérateur sur les clés `Iterator<K>` peut être obtenu grâce à `keySet().iterator()`. Mais il est également possible d'obtenir directement un itérateur sur les paires clé-valeur d'un dictionnaire grâce à `entrySet().iterator()`. L'itérateur est alors de type `Iterator<Map.Entry<K,V>>`. L'interface `Map.Entry<K,V>`, qui est une interface publique interne à `Map` (d'où la notation `.` similaire à celle d'un attribut), dénote une paire clé-valeur. La clé s'obtient avec `getKey()`, et la valeur avec `getValue()`. Un itérateur sur les paires clé-valeur remplace avantageusement une boucle qui itère sur les clés et fait un `get` à chaque itération.

Classes internes. Une classe interne est une classe déclarée au sein d'une autre classe. Dans l'exemple suivant :

```java
public class Contact
{
   private String nom;
   public Contact(String nom) { this.nom = nom; }

   public class Telephone
   {
      private String numero;
      public Telephone(String numero) { this.numero = numero; }
      public String toString() { return nom + " : " + numero; }
   }
   public Telephone nouveauTelephone(String numero) { return new Telephone(numero); }
}
```

la classe `Telephone` est interne à la classe `Contact`. La caractéristique principale d'une classe interne est qu'une instance de `Telephone` ne peut être créée que par une instance de `Contact`, dans une méthode de `Contact` (ici, `nouveauTelephone`). L'objet `Telephone` ainsi créé garde une référence implicite sur l'objet `Contact` qui l'a engendré, et peut ainsi accéder à tous ses attributs et toutes ses méthodes, *même privés*. C'est le cas de la méthode `toString`. Ainsi `(new Contact("Tom")).nouveauTelephone("3615").toString()` retournera la chaîne `"Tom : 3615"`.
Attention : si vous définissez une classe `TelephoneGen` interne à une classe avec paramètre générique `ContactGen<T>`, alors la classe interne hérite du paramètre de type `T`, que vous pouvez utiliser dans les types des attributs et méthodes. Il ne faut pas alors définir la classe interne comme `class TelephoneGen<T>`, mais simplement comme `class TelephoneGen` ; sinon, cela crée un nouveau paramètre de type `T`, qui masque le paramètre de même nom dans `ContactGen<T>`.

4.4 Respect de l'interface collection

Notre classe de multi-ensembles semble respecter les principes fondamentaux des collections Java : il est possible d'ajouter un élément, d'ôter un élément, de tester sa présence. Nous souhaitons donc que `HashMultiSet<T>` implante l'interface `Collection<T>`. Ajouter `implements Collection<T>` dans le fichier sous Eclipse (à essayer) montre qu'un grand nombre de méthodes manquent encore afin de respecter cette interface. Pour corriger ce problème avec le minimum d'efforts, nous utilisons la classe abstraite `AbstractCollection` fournie par Java. Celle-ci contient une implantation générique des collections qui s'appuie sur un cœur minimal de méthodes abstraites, supposées implantées dans une classe concrète qui étend cette classe abstraite, pour synthétiser le comportement des autres méthodes.

⇒ **Travail demandé :**

1. Faites dériver la classe `HashMultiSet<T>` de `AbstractCollection<T>` et faites étendre l'interface `MultiSet<T>` de l'interface `Collection<T>`. Corrigez les erreurs éventuelles que cette manipulation a mis au jour.
2. Lisez la documentation de `AbstractCollection` sur le site Java d'Oracle. Déterminez, pour chacune des méthodes fournies par `AbstractCollection` et que nous ne redéfinissons pas dans `HashMultiSet`, si elle a bien le comportement souhaité et fournit une implantation optimale en terme d'efficacité ; corrigez si nécessaire. Y-a-t-il des méthodes définies dans `HashMultiSet<T>` mais dont nous aurions pu nous passer car elles sont déjà définies correctement et efficacement dans `AbstractCollection` ?
3. Assurez-vous que la classe de test `pobj.multiset.test.CollectionTest` fonctionne ; celle-ci vérifie que nos collections implantent bien l'interface `Collection`.

ORGANISATION DES COLLECTIONS. L'interface `Collection` est à la racine des collections Java. Elle correspond à un ensemble non ordonné d'éléments, supposés comparables par la méthode `equals`. Toutes les autres interfaces de collections dérivent de `Collection` en y ajoutant des contraintes : `List` est une collection ordonnée, `Set` est une collection non ordonnée où un élément apparaît au plus une fois, etc. Chaque interface collection possède une ou plusieurs implantations.
Par ailleurs, chaque interface possède une ou plusieurs classes abstraites fournissant un squelette d'implantation, synthétisant la plus part des méthodes d'une collection à partir d'un noyau de méthodes laissées abstraites. Ainsi, il existe `AbstractCollection`, mais aussi `AbstractList` (implantant `List` à partir de `get` et `set`), `AbstractSeqentialList` (implantant `List` à partir d'un itérateur) et `AbstractSet` (similaire à `AbstractCollection`, mais faisant l'hypothèse supplémentaire qu'un élément n'existe qu'en au plus un exemplaire dans la collection pour satisfaire au contrat de `Set`).

4.5 Application : comptage de fréquences de mots

Notre application principale consiste à déterminer les mots les plus fréquents dans un texte. Cette application sera contenue dans une classe `WordCount` du package `pobj.multiset`.

La classe `WordCount` contiendra une méthode statique `wordcount(MultiSet<String> ms)` qui fera le traitement suivant :

1. charger un fichier ligne par ligne et le découper en mots ; pour cela vous pourrez utiliser le fragment de code suivant :

```
String file = "MonFichier.txt";
BufferedReader br = new BufferedReader(new FileReader(file));
String line;
while ((line = br.readLine())!=null) {
   for (String word : line.split("\\P{L}+")) {
      if (word.equals("")) continue; // ignore les mots vides
      // TODO: traitement à faire pour le mot word
   }
}
br.close();
```

2. accumuler les mots dans le multi-ensemble `ms` passé en paramètre ;
3. extraire du multi-ensemble la liste, sans doublon, des éléments ; pour cela vous pourrez ajouter une méthode `List<T> elements()` dans l'interface `MultiSet<T>` qui retourne la liste des éléments du multi-ensemble ;
4. trier cette liste par fréquence décroissante dans le multi-ensemble (voir encadré) ;
5. afficher les 10 premières entrées de cette liste (i.e., les 10 mots les plus fréquents).

La classe `WordCount` sera exécutable : elle contiendra une méthode statique publique `main`. Celle-ci se contentera d'appeler `wordcount` en lui passant en paramètre un `HashMultiSet` nouvellement créé.

Notez que `wordcount` est programmée vis à vis de l'interface `MultiSet` et non de l'implantation `HashMultiSet`, ce qui nous permettra de varier l'implantation choisie dans les questions suivantes sans avoir à modifier `wordcount` !

⇒ **Travail demandé :** Programmez la classe `WordCount` et exécutez-là sur un petit fichier texte de votre choix pour la tester.

Tri de liste. Toute collection ordonnée (c'est-à-dire obéissant à l'interface `List<T>`) contenant des éléments comparables (c'est-à-dire obéissant à l'interface `Comparable<T>`) peut être triée très simplement, en exploitant la méthode statique `sort` de la classe utilitaire `java.util.Collections`. Si l'ordre naturel, défini par la méthode `compareTo`, n'est pas l'ordre souhaité par le tri, il est possible de passer à `sort` en argument supplémentaire un objet obéissant à `Comparator`. Un `Comparator<T>` demande d'implanter une méthode `int compare(T o1, T o2)` qui compare deux objets de classe `T` et retourne un entier qui doit être strictement négatif si `o1` est strictement plus petit que `o2`, 0 si `o1` et `o2` sont égaux, et strictement positif si `o1` est strictement plus grand que `o2`. Dans notre application, nous devons trier une liste d'objets, de type `List<T>`, en fonction du nombre d'occurrences de chaque objet dans un multi-ensemble `MultiSet<T>`, ce qui nécessitera l'implantation d'un `Comparator<T>` dédié.
La classe standard `java.util.Collections` comporte de nombreuses autres méthodes statiques très utiles pour manipuler les collections (recherche, copie, extraction de tranches, etc.).

4.6 Comparaison d'implantations

La structure `HashMultiSet` est bien adaptée au problème de comptage de fréquence de mots. Pour nous en assurer, nous allons la comparer à une implantation naïve de la même interface, `MultiSet`, mais utilisant une simple liste, par exemple `ArrayList`. Rappelons qu'une liste Java peut tout à fait contenir de multiples copies d'un élément. Cependant, compter le nombre d'occurrences d'un élément nécessitera à chaque fois de parcourir l'intégralité de la liste...

⇒ **Travail demandé :** Programmez `NaiveMultiSet`, une version naïve de multi-ensemble respectant l'interface `MultiSet` et utilisant des `ArrayList` en interne. Comparez les performances de `wordcount` avec `NaiveMultiSet` et avec `HashMultiSet` sur des exemples de taille variée.

Nous fournissons une classe `pobj.util.Chrono` pour vous aider à chronométrer le temps d'exécution des différents appels à `wordcount`. Cette classe s'utilise de la manière suivante :

```
Chrono chrono = new chrono();
/* code à chronométrer */
chrono.stop();
```

4.7 Rendu de TME (OBLIGATOIRE)

Vous suivrez pour le rendu de TME les mêmes consignes qu'aux TME 1 et suivants : vous propagerez vos modifications dans votre projet privé `MultiSet` sur le serveur GitLab et leur associerez une *release* avec un nom de *tag* adapté. Dans la partie « Release notes » de la *release*, vous fournirez :

- La trace d'exécution de `wordcount` sur le fichier texte `data/WarAndPeace.txt` (qui est une version anglaise de *Guerre et Paix*) en utilisant les deux implantations de `MultiSet` : `HashMultiSet` et `NaiveMultiSet`.
- La réponse aux questions suivantes : Quels avantages y-a-t-il à implanter l'itérateur sous forme de classe interne ? Quelles sont les autres approches possibles ? Quels sont leurs avantages et inconvénients ?

La classe `HashMultiSet` sera exploitée au prochain TME. Il est donc indispensable de terminer votre implantation avant le début du TME 5.