TME 5 : Robustesse des multi-ensembles

Objectifs pédagogiques :

- robustesse des implantations ;
- définition, signalement et rattrapage des exceptions ;
- assertions, invariants ;
- tests avec JUnit 4.

But

Nous avons développé dans le TME précédent une structure de données pour gérer des multi-ensembles. Mais, cette implantation est-elle vraiment correcte ? Est-elle robuste dans tous ses contextes d'utilisation ? Dans ce TME, nous allons améliorer la robustesse de notre implantation afin d'éviter qu'une erreur de manipulation de la part du client (i.e., de l'application qui utilise le multi-ensemble) ne corrompe notre structure de données. Nous allons également tester notre implantation pour vérifier qu'elle respecte bien les attentes de l'utilisateur (i.e., qu'elle respecte sa spécification).

Nous allons travailler dans un package `pobj.tme5` au sein du projet `MultiSet` du précédent TME. Vous commencerez par recopier l'interface `MultiSet` et les classes `HashMultiSet` et `WordCount` du précédent TME (package `pobj.tme4`) dans notre nouveau package (`pobj.tme5`) : nous allons en effet modifier ces fichiers.

5.1 Validation des arguments

Toutes les opérations ne sont pas permises sur les multi-ensembles. Par exemple, il est interdit d'ajouter ou de soustraire un nombre strictement négatif d'occurrences d'un élément. C'est bien sûr au client de s'assurer que les méthodes `add` et `remove` sont appelées avec des arguments valides, mais il est indispensable que notre classe de multi-ensembles vérifie les arguments en tête des méthodes et lance proprement une exception en cas d'utilisation incorrecte, plutôt que d'exécuter du code à l'effet imprévisible ou dangereux (nous ne faisons pas tout à fait confiance au client...). Nous utiliserons pour cela l'exception standard Java adaptée à ce type de situations : `IllegalArgumentException`. Nous préciserons dans le constructeur de l'exception un message d'erreur intelligible et informatif qui aidera le programmeur à corriger le problème.

⇒ **Travail demandé :** Modifiez la classe `HashMultiSet` (on se limitera aux fonctions de retrait et d'ajout) pour vérifier les arguments et signaler, si besoin, l'exception `IllegalArgumentException`. Exécutez l'application `WordCount` du TME précédent avec cette nouvelle implantation et vérifiez qu'elle ne lance pas d'exception.

> **Exceptions non vérifiées.** L'exception `IllegalArgumentException` dérive de `RuntimeException`. C'est une exception non vérifiée, dans le sens où une méthode qui signale une telle exception n'a pas besoin de le déclarer avec le mot-clé `throws` dans sa signature.

5.2 Affichage

Une des méthodes les plus simples, mais tout de même très utile, pour déboguer une application consiste à afficher régulièrement des informations sur l'état du programme (*log*), par exemple la valeur de certaines variables, le contenu d'objets, etc. Par défaut, la méthode `toString` héritée d'`Object` donne le nom de la classe et un *hash* (par défaut, l'adresse mémoire), ce qui est peu exploitable lors du débogage ; elle ne donne en particulier aucune information sur les attributs de

l'objet. Il est donc important de redéfinir la méthode `toString` afin de la rendre plus informative et agréable à lire. Dans cette question, nous demandons d'implanter une méthode `toString` pour `HashMultiSet` qui affiche la liste des éléments avec leur nombre d'occurrences, de la manière suivante : `[a:1; b:2; c:99]`, qui signifie que `a` apparaît une fois, `b` deux fois et `c` 99 fois.

⇒ **Travail demandé :** Ajoutez une méthode `public String toString()` à `HashMultiSet` et modifiez `WordCount` afin qu'elle affiche le dictionnaire obtenu après la lecture du fichier `WarAndPeace.txt` utilisé lors du précédent TME.

Conversion implicite en chaîne. La méthode publique `toString` est définie dans `Object`. Cela signifie que, même si aucune implantation de `toString` n'est définie explicitement, toute classe hérite forcément d'une implantation. Cette méthode est en particulier appelée implicitement par l'opérateur `+` de Java. Rappelons que `+` permet d'ajouter deux nombres mais aussi de concaténer deux chaînes de caractères. Si au moins un des arguments de `+` est une chaîne et l'autre est un objet, alors ce dernier est converti en chaîne par un appel à `toString`. C'est pour cela que nous pouvons par exemple écrire : `"multi-ensemble : " + (new HashMultiSet())`. Il est même possible d'écrire `"" + o` pour convertir un objet `o` en chaîne !

`StringBuilder`. Créer une longue chaîne par concaténation, avec l'opérateur `+`, de nombreuses petites chaînes est très peu efficace. Considérez, par exemple, que `String s = "x: "; for (int i=0; i<100; i++) s = s+"a";` a un coût quadratique, puisque l'itération au rang `i` copiera une chaîne de $i+3$ caractères dans une nouvelle chaîne fraîche...
La solution est d'utiliser un `StringBuilder`, qui est une chaîne de caractères mutable et extensible avec une bonne complexité amortie. Après avoir créé un `StringBuilder` vide, des chaînes sont accumulées avec la méthode `append`. Enfin, le contenu du `StringBuilder` est converti en chaîne en appelant sa méthode `toString`. Voici, à titre d'illustration, un exemple d'utilisation de `StringBuilder` :

```
StringBuilder b = new StringBuilder();
b.append("x: ");
for (int i=0; i<100; i++) b.append("a");
return b.toString();
```

5.3 Tests JUnit 4

Nous allons maintenant vérifier que notre classe implante bien un multi-ensemble, en proposant un jeu de tests unitaires au format JUnit 4 : c'est la correction fonctionnelle. Rappelons qu'un test consiste à appeler un certain nombre de méthodes de `HashMultiSet` avec des arguments bien choisis, puis à vérifier avec une assertion JUnit que le résultat est bien conforme à nos attentes. Voici un exemple de deux tests simples pour la méthode `add` ; un premier test doit placer le multi-ensemble dans un état précis (le nombre d'occurrences de `a` est 6), tandis que le deuxième test doit se terminer en exception `IllegalArgumentException` :

```
@Test
public void testAdd1() throws InvalidCountException {
  MultiSet<String> m = new HashMultiSet<>();
  m.add("a");
  m.add("a",5);
  assertEquals(m.count("a"), 6);
}
```

```
@Test(expected = IllegalArgumentException.class)
public void testAdd2() throws InvalidCountException {
 MultiSet<String> m = new HashMultiSet<>();
 m.add("a");
 m.add("a",-1);
}
```

La classe fournie `pobj.tme5.test.HashMultiSetTest` contient ces deux méthodes de test. Bien sûr, il nous faudrait bien d'autres tests. Plus précisément, nous demandons de :

1. tester individuellement les méthodes `remove`, `size`, `toString` et `clear`, sur le modèle de `add` ;
2. tester une séquence complexe de `add` suivis de `remove` dans un multi-ensemble contenant plusieurs éléments différents (rappelons qu'une méthode `@Test` peut contenir plusieurs appels à `assertEquals`) ;
3. tester les cas particuliers suivants : `add` et `remove` avec un argument `count` à zéro ; cas `add(x,n)` suivi de `remove(x,n)` ; cas `count` appelé sur un élément jamais ajouté.

Tests JUnit 4. Rappelons que, pour utiliser JUnit 4, il est nécessaire de :

1. s'assurer que la bibliothèque JUnit 4 est activée ; pour cela, clic droit sur le projet, puis « Build Path > Configure Build Path… », onglet « Libraries », bouton « Add Library… », choisir « JUnit », puis « JUnit 4 » ;
2. programmer une classe séparée pour contenir toutes les méthodes de test ;
3. dans cette classe, créer des méthodes utilisant l'annotation `@Test`, qui serviront de points d'entrée de JUnit 4 ;
4. dans chaque méthode, utiliser la famille de fonctions `assert···` (en particulier `assertEquals`, `assertTrue`, `assertFalse`) ;
5. lancer le test avec un clic droit sur la classe, option « Run As > JUnit Test » ;
6. vérifier que la barre JUnit à gauche s'affiche en vert ; si ce n'est pas le cas, cliquez à gauche sur un test ayant échoué, cherchez la cause dans l'onglet « Failure trace » et corrigez l'erreur.

Notez qu'il vaut bien mieux utiliser `assertEquals(a,b)` que `assertTrue(a==b)`, puisque `assertEquals` utilise la comparaison d'objets `equals` et non l'égalité physique `==`.
Notez également qu'une méthode de test peut faire plusieurs appels à `assert···` pour vérifier plusieurs conditions, les unes à la suite des autres. Toutefois, il ne faut pas en abuser : toutes les vérifications d'une méthode ne comptent que pour un seul test, et JUnit interrompt un test au premier échec, donc les vérifications suivantes dans ce test ne seront pas effectuées.
Pensez également qu'une classe JUnit 4 peut tout à fait définir des attributs et des méthodes outre les méthodes de test. Chaque test sera effectué par un objet fraîchement créé pour l'occasion, et les méthodes ayant l'annotation `@Before` seront exécutées avant chaque méthode de test. Ceci est très utile pour factoriser du code commun à tous les tests (la création d'une `HashMultiSet` vide est un bon candidat).

⇒ **Travail demandé :** Créez une classe `pobj.tme5.test.HashMultiSetTest2` avec tous les tests demandés. Exécutez ces tests sur votre classe `HashMultiSet` et corrigez les éventuelles erreurs trouvées par les tests. Modifiez le fichier de configuration `.gitlab-ci.yml` pour que votre classe de test soit exécutée à chaque *push* sur le serveur GitLab (test de régression en intégration continue). Pour ajouter une classe de test `MonTest`, il suffit d'ajouter les deux lignes :

```
TME5 MonTest:
  script: ./scripts-ci/run.sh pobj.tme5.test.MonTest
```

5.4 Vérification de la cohérence interne avec des assertions

La classe `HashMultiSet<T>` délègue la gestion de son état interne à une `HashMap<T,Integer>` et à un entier `size`. Mais toutes les combinaisons d'états de la `HashMap` et de valeurs de `size` ne sont pas valides. L'état interne d'un objet est cohérent uniquement si :

- toutes les valeurs stockées dans notre `HashMap` sont *strictement positives* ;
- `size` est égal à la somme des occurrences de tous les objets de notre `HashMap`.

Sauf erreur de programmation, ces propriétés sont maintenues par toutes les méthodes : ce sont des *invariants*. Pour vérifier notre implantation, nous allons nous assurer que c'est bien le cas en testant ces invariants après chaque modification du multi-ensemble, c'est-à-dire essentiellement à la fin des méthodes `add` et `remove`. Ces tests étant néanmoins assez coûteux, il nous faut un moyen de les activer uniquement en phase de débogage, et pas en production. Nous utilisons pour cela les *assertions* de Java, c'est-à-dire le mot-clé `assert` (à ne pas confondre avec les méthodes statiques de JUnit fournies par `org.junit.Assert`). L'instruction `assert` suivie d'une expression booléenne testera l'expression (sous réserve que l'option `-ea` est passée à la machine virtuelle Java) et signalera une exception `AssertionException` en cas d'échec. Par ailleurs, le code contenant les deux vérifications de cohérence ci-dessus sera encapsulé dans une méthode `isConsistent`, qui renvoie `true` si la structure de multi-ensemble est cohérente, `false` sinon. Tester l'invariant se réduira donc à une instruction `assert isConsistent()`, insérée à la fin des méthodes `add` et `remove`.

⇒ **Travail demandé :** Ajoutez à `HashMultiSet` la méthode `isConsistent` et les assertions `assert isConsistent()` en fin de méthodes existantes. Exécutez l'application `WordCount` ainsi que les tests unitaires de la question précédente afin de vérifier que la classe `HashMultiSet` respecte bien ses invariants. Attention à bien activer la prise en compte des assertions lors de l'exécution (option `-ea`, voir encadré ci-dessous) !

Assertions. L'instruction `assert` est suivie d'une expression booléenne. Par exemple : `assert i>=0;` (note : contrairement aux `if` et `while`, les parenthèses autour de l'expression booléenne ne sont pas nécessaires).
Quand l'assertion est exécutée, l'expression est évaluée. Si le résultat est `true`, rien ne se passe. Mais, si le résultat est `false`, alors une exception `AssertionException` est signalée.
Pour éviter que les assertions soient activées dans le code en production, le comportement par défaut de la JVM est d'ignorer les instructions `assert`. Nous devons donc les activer manuellement. Pour cela, il faut faire un clic droit sur la classe à exécuter avec les assertions activées, choisir l'option « Run As > Run Configurations », l'onglet « Arguments », et ajouter dans la boîte « VM arguments » l'option `-ea`.
Notez que le fait d'évaluer une assertion ne devrait pas modifier l'état de l'objet (l'expression est dite *pure*, ou *sans effet de bord*). C'est au programmeur de s'assurer que l'évaluation de l'expression, donc dans notre cas la méthode `isConsistent`, ne modifie pas l'état de l'objet en cours d'examen.
Pour plus d'information, vous pouvez consulter la documentation Java spécifique sur les assertions.

5.5 Vérification de la cohérence interne avec un décorateur

Les assertions Java peuvent être activées à une granularité de classe (ou package). Toutefois, si les assertions sont activées pour une classe, elles seront activées pour toutes les instances de cette classe, durant toute l'exécution du programme. Nous allons proposer ici une solution alternative, qui permet de choisir au cas par cas pour quels multi-ensembles la vérification de l'état interne sera effectuée. Pour cela, nous allons définir une classe `MultiSetDecorator<T>` qui :

– obéit à l'interface `MultiSet<T>` ;
– contient un attribut `decorated` de type `MultiSet<T>`, fixé lors de la construction ;
– délègue toutes les opérations de `MultiSet<T>` à l'attribut `decorated` ;
– après avoir délégué le travail effectif à `decorated` dans `add` et `remove`, appelle `isConsistent` et signale une exception (par exemple `InternalError`) en cas d'échec.

Ainsi, `MultiSetDecorator` s'appuie sur une implantation de multi-ensemble existante, et mime parfaitement son comportement, à cela près qu'elle y ajoute un comportement supplémentaire : la vérification systématique de la cohérence. Il est alors facile de mélanger, dans une même application, des multi-ensembles vérifiés et des multi-ensembles non vérifiés. Voici un exemple :

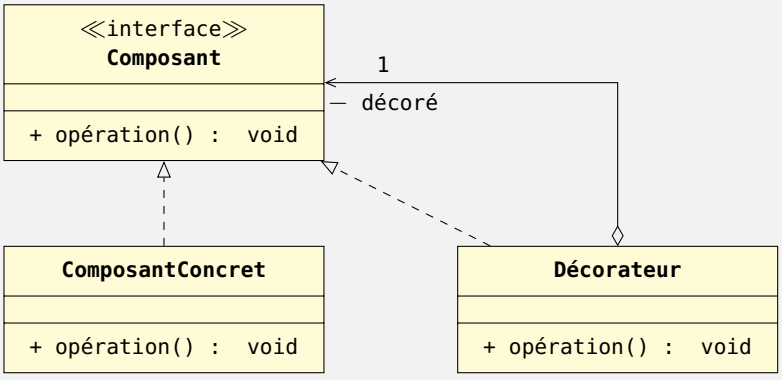
```
MultiSet<String> unchecked = new HashMultiSet<>();                          1
MultiSet<String> checked = new MultiSetDecorator<>(new HashMultiSet<>());   2
unchecked.add("x");                                                         3
checked.add("y");                                                           4
```

Puisque `MultiSetDecorator` ne fait aucune hypothèse sur l'implantation du multi-ensemble sous-jacent (il est programmé vis-à-vis de l'interface `MultiSet`), il est utilisable également avec la classe `NaiveMultiSet` du TME précédent. Puisque `MultiSetDecorator` obéit lui-même à l'interface `MultiSet`, il est facile de remplacer un multi-ensemble par un multi-ensemble vérifié dans une application client (à condition bien sûr d'avoir programmé l'application client vis-à-vis de l'interface `MultiSet` et non vis-à-vis d'une implantation spécifique).

⇒ **Travail demandé :**

1. Implantez la classe `MultiSetDecorator`.
2. Servez-vous du décorateur pour vérifier la cohérence de `HashMultiSet` et de `NaiveMultiSet` dans l'application `WordCount`.
3. Modifiez vos classes de test unitaire pour qu'elles utilisent le décorateur afin de vérifier la cohérence de votre implantation lors des tests.

Motif décorateur. Ajouter une responsabilité à un objet sans changer son interface est un besoin fréquent en programmation. La solution que nous avons utilisée dans cette question a pour nom le *motif décorateur*. Ce motif est très fréquent. Il se retrouve en particulier dans la gestion des entrées-sorties et du réseau dans la bibliothèques standard Java. Par exemple, la classe `BufferedInputStream` prend un flux arbitraire et y ajoute un tampon pour améliorer ses performances. Le flux original obéit à l'interface `InputStream`, et le flux avec tampon obéit également à l'interface `InputStream`.



5.6 Lecture dans un fichier

Nous proposons maintenant de créer une classe qui, à l'inverse de `toString`, prend une représentation textuelle d'un multi-ensemble et la convertit en objet `MultiSet<T>`. Pour simplifier, nous nous limiterons au cas des multi-ensembles de chaînes, `MultiSet<String>`, et nous utiliserons un format de fichier simpliste : chaque ligne comportera une chaîne, suivie du caractère « `:` », suivi d'un nombre entier indiquant sa fréquence (sans espace). Par exemple :

```
chaîne1:12
chaîne2:49
chaîne3:13
```

Bien sûr, ce format a des limitations. En particulier, une chaîne ne peut pas comporter de caractère « `:` », puisque tout caractère « `:` » sera considéré comme un séparateur entre la chaîne et la fréquence. Nous allons encapsuler cette conversion dans une classe dédiée, `MultiSetParser`, qui comportera donc simplement une méthode statique `parse` prenant le nom du fichier à lire en argument.

```
public static MultiSet<String> parse(String fileName);
```

De nombreuses erreurs peuvent se produire lors de l'exécution de `parse` : fichier introuvable, erreur d'entrées-sorties, ligne ne contenant pas de caractère « `:` », caractère « `:` » suivi d'une chaîne ne représentant pas un entier strictement positif, etc. Nous allons signaler toutes ces erreurs à l'aide d'une exception dédiée, `InvalidMultiSetFormat`. Cette classe d'exception aura les caractéristiques suivantes :

- `InvalidMultiSetFormat` dérive directement de la classe `Exception` ;
- chaque exception est créée avec un message d'erreur d'intelligible.

L'exception `InvalidMultiSetFormat` peut être signalée en réponse à une autre exception, par exemple `IOException` en cas d'erreur d'entrées-sorties. Il est important de s'assurer que toutes les `IOException` sont interceptées par la méthode `parse` et qu'aucune ne s'échappe vers le client. Le client ne veut voir que des `InvalidMultiSetFormat`.

⇒ **Travail demandé :** Implantez les classes `InvalidMultiSetFormat` et `MultiSetParser`. Ajoutez une classe `MultiSetParserTest` contenant une méthode `main` qui utilise `MultiSetParser` pour lire un multi-ensemble depuis un fichier, puis affiche le contenu du multi-ensemble dans la console avec `System.out.println`, afin de vérifier que la lecture s'est bien déroulée.

Lecture de fichiers texte. Nous rappelons les principes de la lecture d'un fichier texte :

- un fichier est ouvert par la création d'un objet `FileReader` mais, afin d'être lu ligne à ligne, celui-ci doit être encapsulé dans un objet `BufferedReader`, ce qui donne : `new BufferedReader(new FileReader(fileName))` ;
- une ligne est lue par la méthode `readLine` de `BufferedReader` ; en fin de fichier, `null` est retourné ;
- le fichier doit être fermé par la méthode `close` de `BufferedReader` ;
- ces opérations peuvent signaler une erreur d'entrées-sorties, avec la hiérarchie d'exceptions `IOException`.

Par ailleurs, les méthodes suivantes pourront être utiles :

- `split` dans la classe `String` permet de découper une chaîne en un tableau de sous-chaînes séparées par un séparateur spécifié ;
- la méthode statique `Integer.decode` permet de convertir une chaîne représentant un entier en une valeur entière ; la méthode signale une exception `NumberFormatException` si la chaîne ne représente pas un entier.

Classes d'exception. Pour que ses instances puissent être utilisées comme des exceptions, une classe doit dériver directement ou indirectement de la classe `Throwable`. En pratique, nous dérivons notre exception de `Exception`, qui dérive de `Throwable` et qui regroupe les exceptions ayant vocation à être rattrapées, par opposition aux erreurs graves qui ne sont pas remédiables. Pour toute exception `E`, il est important de définir au moins deux constructeurs :

1. `E(String msg)`, qui spécifie un message d'erreur intelligible ;
2. `E(String msg, Throwable cause)`, qui spécifie un message d'erreur et une exception représentant la cause de notre exception. Ceci est utile quand une exception est rattrapée, puis relancée sous une autre forme, pour garder précisément l'origine des erreurs. C'est le cas pour nous quand une exception `InvalidMultiSetFormat` est signalée en réponse à une `IOException`. Dans une clause `catch (InvalidMultiSetFormat x)`, l'exception originale peut être obtenue par `x.getCause()`. Notez que cette exception peut également avoir une cause, et ainsi de suite : il existe en réalité une *chaîne de causes*.

Quand vous créez une classe d'exception, Eclipse conseille de plus la redéfinition de l'attribut `serialVersionUID`. Vous pouvez utiliser l'auto-correction d'Eclipse pour générer une valeur automatiquement, et ainsi supprimer l'alerte. En pratique, cet attribut sert à distinguer des classes sérialisées (i.e. stockées sur disque ou communiquées sur réseau) de même nom mais d'implantation différente. Ceci ne sera donc pas très important pour notre application.
Puisque notre exception n'est pas une `RuntimeException`, une méthode qui signale mais ne rattrape pas `InvalidMultiSetFormat` devra l'indiquer dans sa signature à l'aide de `throws`. Nous utilisons la vérification statique d'exceptions de Java pour nous assurer que `parse` ne peut signaler que `InvalidMultiSetFormat`, et aucune autre exception (qui ne soit pas une `RuntimeException`) ; par exemple, `parse` ne doit pas signaler de `IOException`.

5.7 Rendu du TME (OBLIGATOIRE)

Vous ferez un rendu de TME sur le même modèle que pour les TME précédents : avec un *push* sur le serveur GitLab et un *tag*. N'oubliez pas d'inclure tous les tests que vous avez écrits dans le fichier de configuration de l'intégration continue `.gitlab-ci.yml`.

Dans la partie « Release notes » du *tag*, vous fournirez :

- les traces d'exécution de la question 5.2 (affichage du dictionnaire de `WarAndPeace.txt`) et de la question 5.6 (fichier texte d'entrée que vous avez utilisé comme exemple et sortie correspondante après chargement puis affichage) ;
- une liste des tests JUnit 4 que vous avez écrits à la question 5.3, ce qu'ils testent, et s'ils passent avec succès votre implantation de `HashMultiSet`.