Programmation en Java : collections

Lors de cette séance de TP, nous verrons l'utilisation des collections Java permettant de gérer un ensemble de données.

Notions

Structure de données

Le package `java.util` nous fournit deux hiérarchies particulièrement importantes pour la programmation. La première permet de grouper des éléments relativement homogènes. La seconde permet de créer des dictionnaires, c'est-à-dire un ensemble de valeurs auxquelles nous accédons à partir d'une clé.

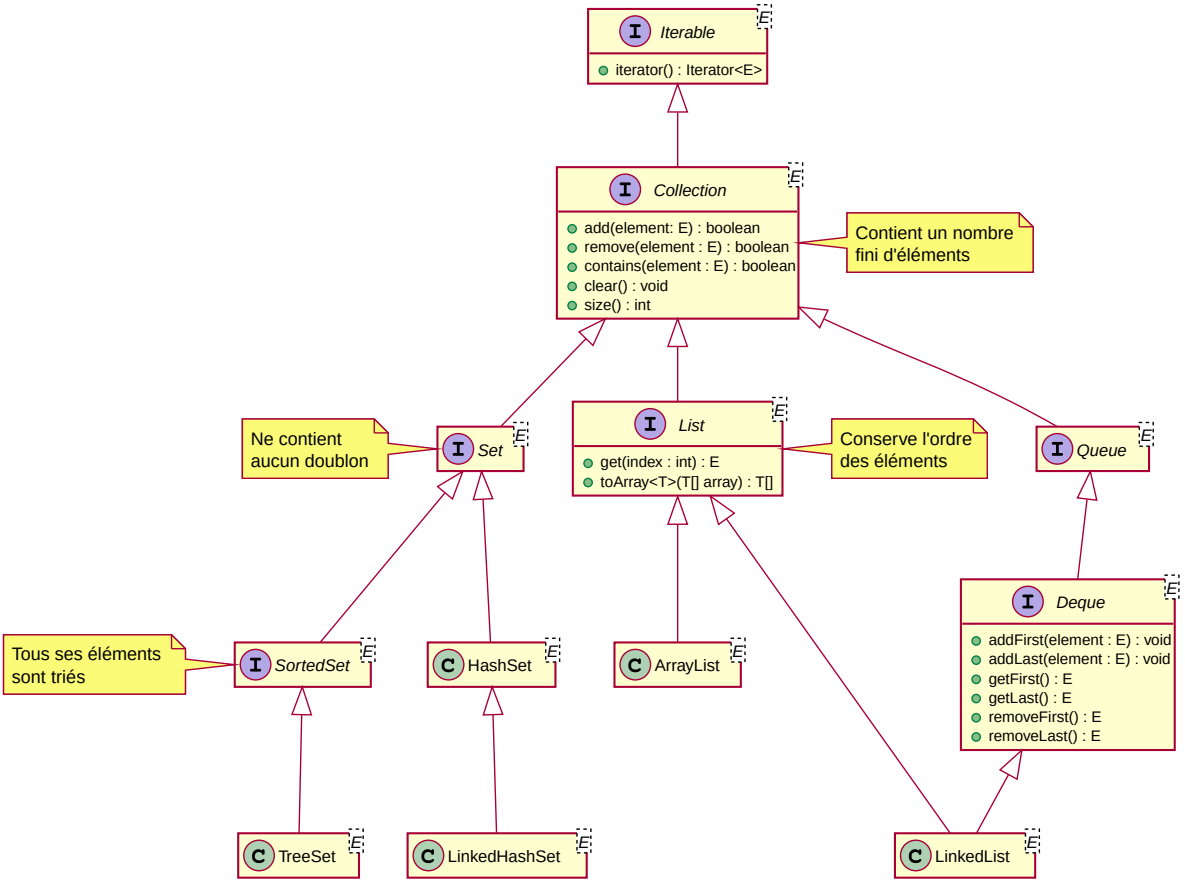

Figure 1. Interfaces et classes principales de la hiérarchie de `Collection`

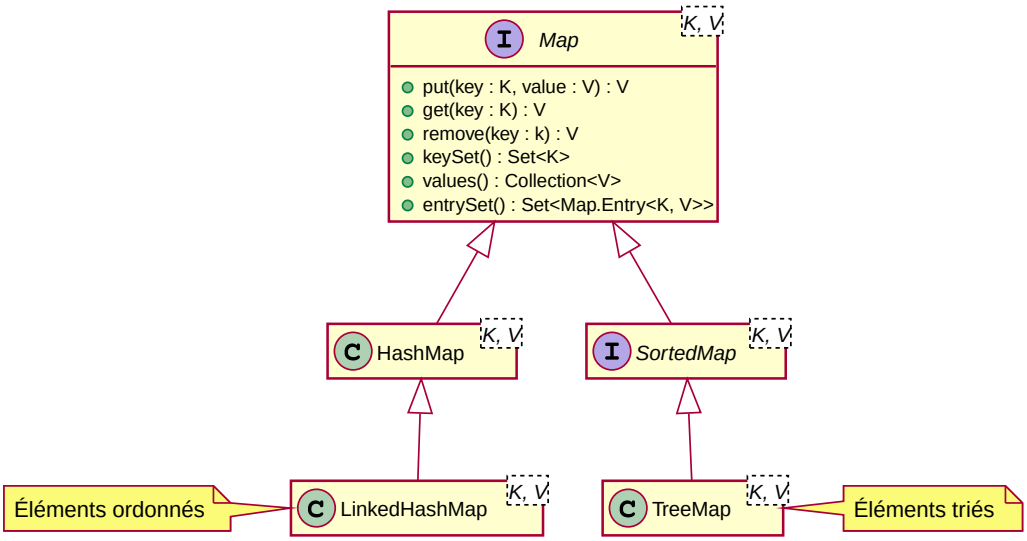

Figure 2. Interfaces et classes principales de la hiérarchie de `Map`

Chaque interface apporte ses propres fonctionnalités. Chaque implémentation possède ses spécificités et ses propres contraintes. En fonction de votre usage, vous choisirez l'interface appropriée. Vous choisirez ensuite une implémentation de cette interface pour optimiser vos performances.

Je ne passerai pas en revue les spécifités des interfaces et classes du framework. Ce serait bien trop long. Je vous renvoie vers l'aperçu du framework collection, sa présentation et la Javadoc du package `java.util`. Vous trouverez notamment des indications sur les performances des différentes méthodes au début de la page de la Javadoc des classes d'implémentation.

Certaines map ou collections font des hypothèses sur les éléments qu'elles contiennent. Les hypothèses les plus courantes portent sur :

- l'implémentation des méthodes `hashCode` et `equals`. Celles-ci sont utilisées par les collections et maps qui utilisent un algorithme de hachage.
- l'implémentation de l'interface `Comparable` ou d'un `Comparator`. Ceux-ci sont utilisés par les collections et maps qui trient leurs éléments.

L'implémentation par défaut des méthodes `hashCode` et `equals` se base sur l'adresse mémoire de l'objet. Deux instances différentes seront donc toujours différentes, même si leur contenu est unique. Le contrat des ces deux méthodes est plus compliqué que vous pourriez le croire. Référez-vous à la Javadoc de la classe `Object` pour connaître leur spécification. Les méthodes statiques `equals(Object valueA, Object valueB)` et `hash(Object… values)` vous aideront à implémenter correctement ces méthodes. La plupart des IDE permettent de les générer facilement en sélectionnant les champs de votre classe.

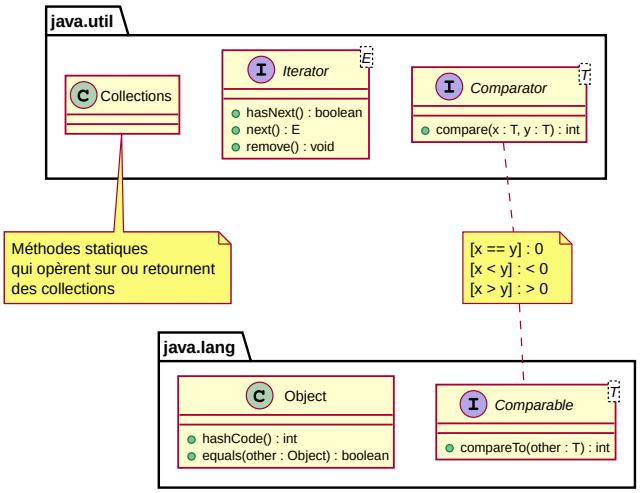

Figure 3. Méthodes et interfaces utilisées dans le framework

Example d'implémentation des méthodes `equals` et `hashCode`

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Person person = (Person) o;
    return Objects.equals(firstName, person.firstName) &&
        Objects.equals(lastName, person.lastName);
}

@Override
public int hashCode() {
    return Objects.hash(firstName, lastName);
}
```

Il est dangereux de baser les méthodes `equals` et `hashCode` de la même classe sur des champs différents. Les classes du framework collections font l'hypothèse que ces méthodes se comportent de manière consistante. Cette remarque vaut aussi si vous implémentez l'une de ces méthodes mais pas l'autre.

Paramètre générique

Les classes et interfaces du framework collections sont prévues pour contenir n'importe quel objet. Afin de garder une certaine facilité d'utilisation, celles-ci utilisent des paramètres génériques. Sur les figures Interfaces et classes principales de la hiérarchie de `Collection` et Interfaces et classes principales de la hiérarchie de `Map`, vous pouvez observer le paramètre `E` pour la hiérarchie de `Collection` et `K` et `V` pour la hiérarchie de `Map`. Ils désignent respectivement le type des éléments de la collection et les types de la clé et de la valeur du dictionnaire.

Lors de l'initialisation d'une collection ou d'une map, il n'est pas nécessaire de répéter les paramètres génériques de la déclaration. On se content alors d'écrire `<>`, aussi appelé *diamond operator*.

Example d'utilisation d'un paramètre générique.

```
Map<String, Student> studentsByName = new HashMap<>(); ①
studentsByName.put("Bridges", new Student("Lisandra", "Bridges"); ②
studentsByName.put("Zimmerman", new Student("Sawyer", "Zimmerman");
...
Student wilcoxStudent = studentsByName.get("Wilcox"); ③
```

① Une map est déclarée. Ses clés sont de type `String`. Ses valeurs de type `Student`.

② Le compilateur refusera de compiler ce code si vous essayer d'appeler la méthode `put` avec les mauvais types d'arguments.

③ La méthode `get` retourne une valeur de type `Student`.

Cette introduction devrait être suffisante pour réaliser les exercices du TP. Si vous souhaitez en savoir plus, vous pouvez consulter le tutorial d'Oracle sur le sujet.

Utilisation des collections

Il existe plusieurs manières d'itérer une collection.

Itération d'une collection à l'aide d'un itérateur

```
Collection<Element> elements = ...
for (Iterator<Element> elementIt = elements.iterator(); elementIt.hasNext(); ) {
    ...
}
```

Itération d'une collection à l'aide d'une boucle foreach

```
Collection<Element> elements = ...
for (Element element: elements) {
    ...
}
```

Iteration d'une collection à l'aide de l'interface `Stream` et d'une lambda expression

```
Collection<Element> elements = ...
elements.stream().forEach(element -> ...)
```

La première version permet un accès à l'itérateur alors que la seconde utilise un itérateur implicite. La troisième version utilise les lambda expressions qui sont disponibles depuis Java 8. L'interface `Stream` offre bien plus que la simple possibilité d'appliquer une fonction à tous les éléments d'une collection.

Exemple de filtrage et de conversion d'une collection

```
Collection<Person> people = ...
Collection<String> maleFirstnames = people.stream()
        .filter(person -> person.getGender().equalsTo(Gender.MALE)) ①
        .map(Person::getFirstname) ②
        .sorted() ③
        .distinct() ④
        .collect(Collectors.toList()); ⑤
```

① Ne garde que les personnes de sexe masculin.

② Transforme le `Stream<Person>` en `Stream<String>` contenant le prénom. Le paramètre est une référence de méthode.

③ Trie les prénoms par ordre alphabétique.

④ Supprime les doublons.

⑤ Transforme le `Stream<String>` en `Collection<String>`.

Exemple de calcul de l'age moyen d'une population

```
double averageAge = people.stream()
        .mapToInt(Person::getAge)
        .average()
        .orElse(0.0);
```

La Javadoc du package `java.util.stream` (en bas de la page) et des classes de ce package vous fourniront plus d'informations.

Optional

La classe `java.util.Optional` est un conteneur de données particulier. Cette classe peut-être considérée comme une collection qui peut soit être vide, soit contenir un élément unique non-null. Elle est souvent utilisée comme valeur de retour d'une méthode pour signifier à son utilisateur qu'il n'est pas toujours possible de retourner une valeur. On la retrouve notamment dans l'API de `java.util.Stream`.

Exercices

Exercice 1

On souhaite gérer une classe d'étudiants. Chaque étudiant reçoit les cotes de plusieurs cours. Votre objectif est d'implémenter la classes `Student` du diagramme Classes de l'exercice 1 et ses méthodes. Vous êtes libres d'en ajouter d'autres et de déplacer l'implémentation tant que les méthodes présentes peuvent être appelées par mes tests unitaires. Le comportement de certaines méthodes est décrit dans la Javadoc du code fourni. Veillez à ce que votre implémentation soit en accord avec ces contraintes. Vous trouverez des tests unitaires dans `src/test/java/exercise1` pour vous aider à évaluer votre progression.

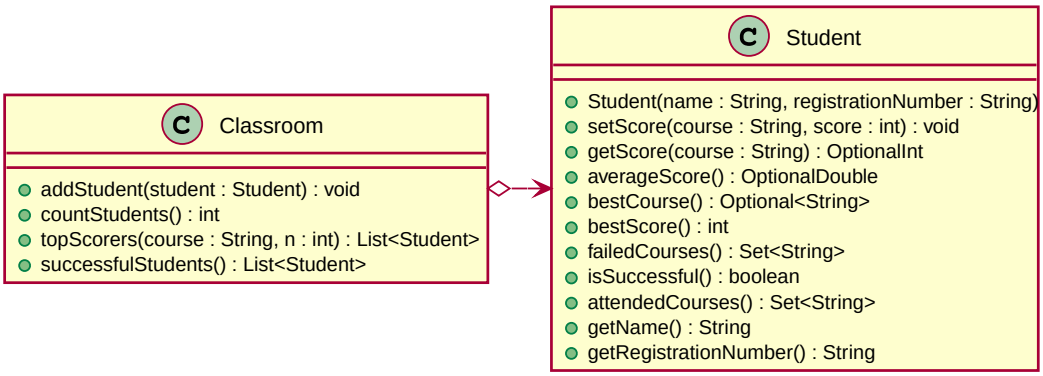

Figure 4. Classes de l'exercice 1

Exercice 2

Implémentez maintenant la classe `Classroom`. Les consignes sont les mêmes que pour l'exercice 1. Vous trouverez des tests unitaires dans `src/test/java/exercise2` pour vous aider à évaluer votre progression.

Références

- Deitel, H. M., & Deitel, P. J. (2002). *Java : comment programmer (4th ed.)*. Les éditions Reynald Goulet INC.
- Deitel, P. J., & Deitel, H. M. (2007). *Java: how to program (7th ed.)*. Les éditions Reynald Goulet INC.
- Evans, E. (2003). *Domain-driven design: tackling complexity in the heart of software (1st ed.)*. Addison-Wesley Professional.
- Manneback, P., & Frémal, S. (2014-2015). *Travaux pratiques de Méthodologie et Langage de Programmation*. UMons.
- Manneback, P. (2005-2006). *Méthodologie et Langages de Programmation*. UMons.
- *Java Platform Standard Edition 8 Documentation*. Récupéré de https://docs.oracle.com/javase/8/docs/