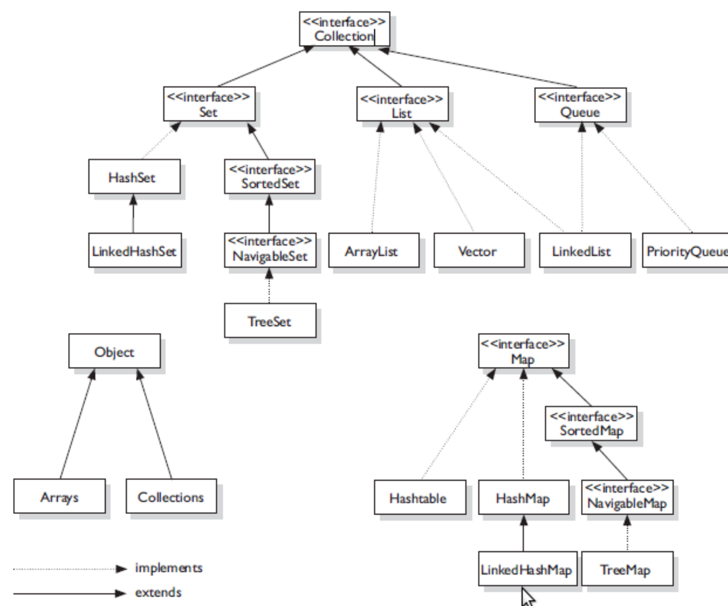
TD 8 L’utilisation d’un ensemble ordonné

Contexte

Aujourd’hui nous allons regarder de près un autre type de collection, notamment un ensemble ordonné de type TreeSet. Notre but sera de créer un ensemble de type TreeSet qui contiendra des éléments d’un type qu’on va définir, notamment le type Etudiant.

Exercice 1

Nous allons premièrement étudier le diagramme ci-dessous qui donne une hiérarchie des classes et interfaces utilisées dans la Java Collection Framework.



1. Trouvez dans ce diagramme les classes `HashSet` et `TreeSet`.
2. Quelle est la relation entre la classe `HashSet` et l’interface `Set` ? Et entre `TreeSet` et `SortedSet` ?
3. Voici un extrait de la documentation Java sur l’interface `SortedSet` :

A `Set` that further provides a *total ordering* on its elements. The elements are ordered using their natural ordering, or by a `Comparator` typically provided at sorted set creation time. The set's iterator will traverse the set in ascending element order. Several additional operations are provided to take advantage of the ordering. (This interface is the set analogue of `SortedMap`.)

All elements inserted into a sorted set must implement the `Comparable` interface (or be accepted by the specified comparator). Furthermore, all such elements must be *mutually comparable*: `e1.compareTo(e2)` (or `comparator.compare(e1, e2)`) must not throw a `ClassCastException` for any elements `e1` and `e2` in the sorted set. Attempts to violate this restriction will cause the offending method or constructor invocation to throw a `ClassCastException`.

Note that the ordering maintained by a sorted set (whether or not an explicit comparator is provided) must be *consistent with equals* if the sorted set is to correctly implement the `Set` interface. (See the `Comparable` interface or `Comparator` interface for a precise definition of *consistent with equals*.) This is so because the `Set`interface is defined in terms of the `equals` operation, but a sorted set performs all element comparisons using its `compareTo` (or `compare`) method, so two elements that are deemed equal by this method are, from the standpoint of the sorted set, equal. The behavior of a sorted set *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the `Set` interface.

Pouvez-vous expliquer le texte dans le deuxième et troisième paragraphes ?

4. Regardez la description de `compareTo` dans l'interface `Comparable` :

Method Detail

- *compareTo*

`int compareTo(T o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff`y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable`interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(`*expression*`)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

Parameters:

`o - the object to be compared.`

Returns:

`a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.`

Throws:

`NullPointerException - if the specified object is null`

`ClassCastException - if the specified object's type prevents it from being compared to this object.`

Expliquez le fonctionnement de cette méthode.

Exercice 2

Notre but sera d'implémenter un `TreeSet` d'objets de la classe `Etudiant`. Un étudiant aura les attributs prochains : un attribut `nom` de type `String`, un attribut `prenom` de type `String` et un attribut `numeroDEtudiant` de type `int` (on suppose que les numéros d'étudiant ne commencent jamais par un 0).

1. Pour pouvoir définir un `TreeSet` d'étudiants, la classe `Etudiant` devra implémenter l'interface `Comparable`. Comment peut-on indiquer ce fait dans le code de la classe `Etudiant` ?
2. La documentation Java de l'interface `Comparable` est donnée en annexe. Quelle méthode la classe `Etudiant` doit-elle implémenter ?
3. Vous allez supposer que la classe `Etudiant` aura des getters pour chaque attribut et un constructeur avec la signature `Etudiant(String nom, String prenom, int numeroDEtudiant)`.
   Je vous attire l'attention sur ce paragraphe de la documentation Java sur la méthode `compareTo` :

   It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`.

   Ce paragraphe fait référence à la méthode equals de la classe Object, donc la documentation est mise ci-dessous :

equals

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.

- It is *transitive*: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is *consistent*: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

Parameters:

obj - the reference object with which to compare.

Returns:

true if this object is the same as the obj argument; false otherwise.

See Also:

hashCode(), HashMap

Expliquez ce que fait la méthode equals de la classe Object.

4. Nous allons écrire méthode boolean equals(Object objet) dans la classe Etudiant. Pourquoi doit-on prendre un objet de la classe Object en paramètre (au lieu, par exemple, d'un objet de type Etudiant) ?

   Initialement nous allons considérer que deux étudiants sont égaux si, et seulement si leurs numéros d'étudiant coïncident.

   Attention : la méthode equals prend en entrée un paramètre de type Object. Par contre, nous voulons utiliser cette méthode seulement pour des objets de type Etudiant, et effectivement il va falloir utiliser pour cet objet des méthodes propres à la classe Etudiant. Pour pouvoir faire cela il va falloir premièrement transformer l'objet objet d'un Object à un Etudiant, en utilisant par exemple la technique de « type casting ». Celle-ci consiste en déclarer à Java qu'un certain objet est d'un certain type. Dans notre cas, on utilisera l'instruction :

```
Etudiant etudiant = (Etudiant) objet;
```

   Le méthode equals devra donc vérifier si l'objet en paramètre (objet) est null -- si c'est le cas, on retourne false pour la méthode equals. Puis on fait un type-cast pour l'objet o comme ci-dessus. Si celui-ci et l'objet actuel partagent la même adresse en mémoire la méthode rend true. Pour tout autre cas, on retourne true seulement si l'objet etudiant et l'objet actuel ont le même numéro d'étudiant.

   Ecrivez cette méthode.

5. Disons que dans une classe TD8, dans la méthode `public static void main(String[] args)` on a le code suivant :

```
public static void main(String[] args){
      final Etudiant jean = new Etudiant("Jean", "Dupont", 125);
      final Etudiant gabrielle = new Etudiant("Gabrielle", "Delacroix",
125);

      final Set<Etudiant> ensemble = new HashSet<Etudiant>();
      ensemble.add(jean);
      ensemble.add(gabrielle);
      System.out.println(ensemble.size());
      System.out.println(ensemble);
}
```

Est-ce que ce code compile ?
Quel est le résultat de l'exécution ?

6. Finalement nous voulons écrire la méthode `int compareTo(Comparable objet)`. Cette méthode est censée fonctionner selon les spécifications mentionnées dans l'exercice 1. Notamment :
   - Il faut faire un type cast pour l'objet objet pour le transformer dans un étudiant
   - Si l'étudiant en paramètre est égale (selon la méthode equals) à l'étudiant actuel, alors on retourne 0 ;
   - Si l'étudiant en paramètre a un numéro d'étudiant strictement inférieur à celui de l'objet actuel, alors on retourne 1 ;
   - Finalement si l'étudiant en paramètre a un numéro d'étudiant strictement supérieur à celui de l'objet actuel, on retourne -1.

7. On remplace le code de la méthode main (de l'exercice 5) par :

```
public static void main(String[] args){
      final Etudiant jean = new Etudiant("Jean", "Dupont", 125);
      final Etudiant gabrielle = new Etudiant("Gabrielle", "Delacroix",
122);

      final Set<Etudiant> ensemble = new TreeSet<Etudiant>();
      ensemble.add(jean);
      ensemble.add(gabrielle);
      System.out.println(ensemble.size());
      System.out.println(ensemble);
}
```

Dans un HashSet, les éléments sont listés à chaque fois dans un ordre quelconque. Dans un TreeSet, ils sont toujours ajoutés dans un ordre donné par la méthode compareTo. Quel est l'effet de faire exécuter le code ci-dessus ?

Exercice 3

Nous allons maintenant modifier les méthodes `equals` et `compareTo` pour comprendre quel est leur effet sur notre code.

1. Nous voulons modifier la méthode `equals` de la classe `Etudiant` tel qu'elle retourne `true` si, et seulement si, les deux attributs que ces objets stockent ont la même valeur.
2. Quel est l'effet de faire exécuter le code de l'exercice 2, question 5 ?
3. Modifiez la méthode `compareTo` tel qu'elle ordonne les attributs premièrement par leurs numéros d'étudiant, puis par leurs noms, puis par leurs prenoms.
4. Quel sera l'effet de faire exécuter le code de l'exercice 2, question 7 ?
5. La méthode `compareTo` lève deux types d'exception : `NullPointerException` (un des objets à comparer est null) et `ClassCastException` (on a fait un mauvais type casting). Cette méthode est appelée implicitement lorsqu'on appelle la méthode add, et explicitement si on veut la faire exécuter pour un objet de type `Etudiant`.
   - Qu'est-ce qui se passe si dans la méthode main ci-dessus on rajoute (à la fin) la ligne de code : `System.out.println(jean.compareTo("Gabrielle"));` ? Est-ce que le code compile ? Pourquoi ? Que lest le résultat de l'exécution ?
   - Ecrivez un bloc `try/catch` autour de cette instruction pour capturer l'exception levée.