Développement Logiciel Java

07/02/2025

Les Interfaces

L'héritage permet de factoriser des paramètres et comportements entre objets ayant une forte relation fonctionnelle (généricité -> spécialisation).

Il peut être nécessaire de caractériser certains comportements communs mais applicables à une vaste catégorie d'objets qui ne partagent pas nécessairement un ancêtre commun.

Exemple: le caractère *déplaçable* peut s'appliquer à presque tous les éléments de la classe Animal mais également Vehicules. Cependant, il est difficile d'imaginer un héritage commun à ces deux catégories d'objets.

Les Interfaces

Les interfaces viennent répondre à cette problématique. Elles représentent un contrat à remplir par quiconque implémente l'interface.

```
public interface Movable {

      void move(int x, int y);

}
```

Les interfaces ne spécifient en général pas le corps de leurs méthodes car c'est la responsabilité de l'implémentation de les définir (possibilité d'implémenter un comportement par défaut depuis java 8 avec le mot-clé *default*).

Les implémentations d'interfaces DOIVENT implémenter toutes les méthodes définies dans l'interface qui ne sont pas annotées *default*.

Les implémentations PEUVENT implémenter les méthodes *default* définies dans l'interface.

Les Interfaces

- Une interface peut hériter d'autres interfaces. C'est alors la somme de toutes les signatures de méthodes de la hiérarchie qui doivent êtres implémentées par les implémentations.
- L'héritage d'une classe implémentant une interface est également considéré comme implémentant cette interface.
- Les interfaces participent à la notion de polymorphisme. Il est possible de manipuler une liste d'objets typés comme implémentation d'une interface sans se préoccuper de leur implémentation.

Les Tableaux

Instancier un tableau:

```
type[] nomTableau = new type[taille];
```

Un tableau est une suite de types références ou de valeurs de taille fixe.

Les éléments du tableau sont accessible par leur index: `nomTableau[i]` (index à 0).

Un tableau est un Objet Java, et possède donc des méthodes et attributs (notamment length).

La classe Arrays de l'api Java.collections propose un certain nombre de méthodes utilitaires pour la manipulation de tableaux.

Les Tableaux

Initialiser un tableau:

- Directement à la déclaration:

```
String[] myarray = new String[]{"a", "b", "c"};
```

Ou

```
String[] myarray = {"a", "b", "c"};
```

- Element par élément:

```
String[] myArray = new String[10];

myArray[0] = "a";
```

Les Tableaux

Pour parcourir un tableau:

```
for (int i=0; i<myArray.length; i++) {
    System.out.println(myArray[i]);
}
```

ou :

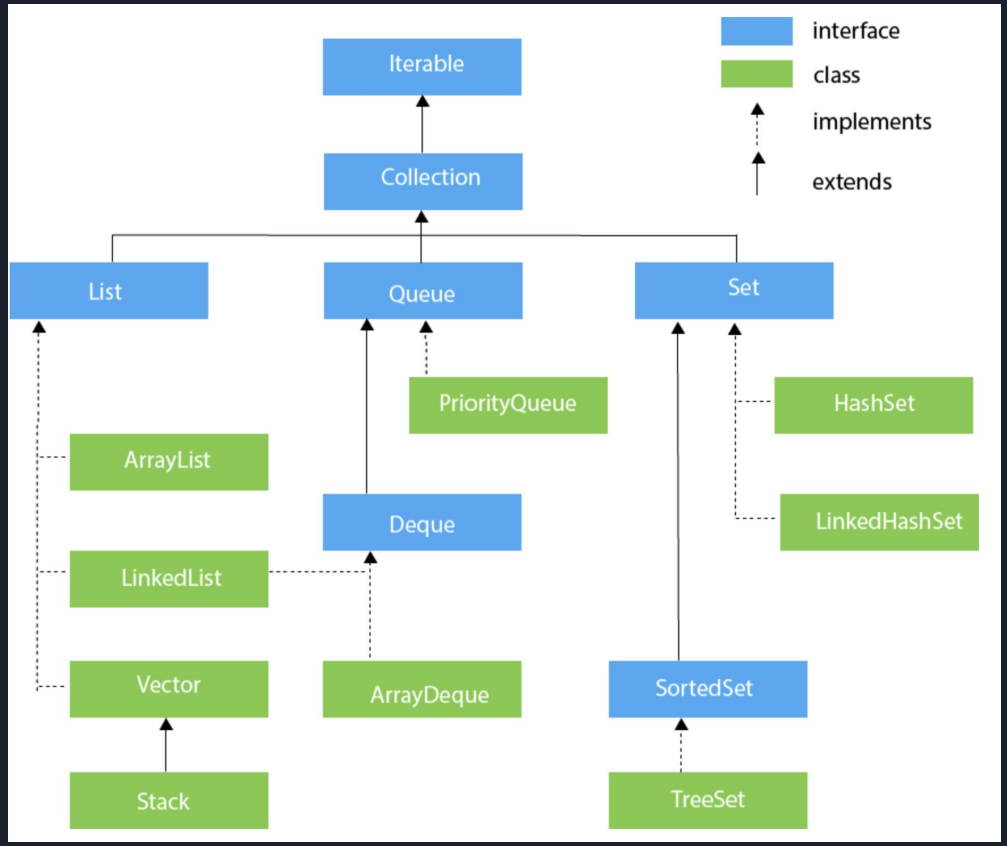
```
for (String myString: myArray) {
    System.out.println(myString);
}
```

Les Tableaux

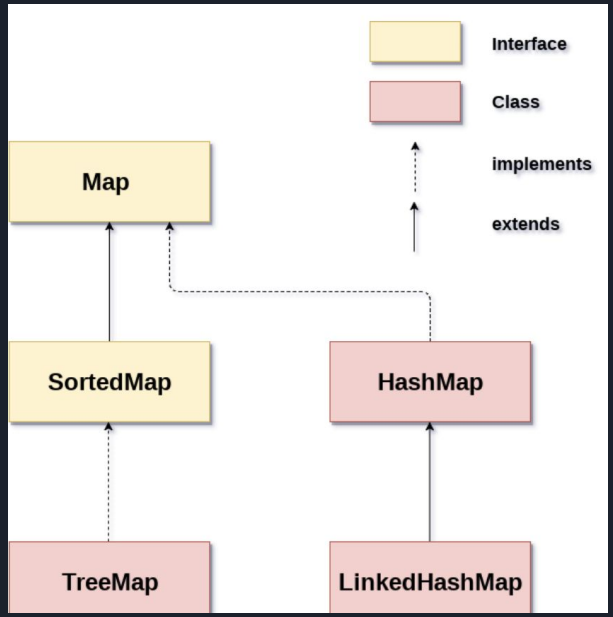
Il est également possible de faire des tableaux à plusieurs dimensions:

```
int[][] myArray = new int[10][10];
for (int i=0; i<myArray.length; i++) {
    for (int j=0; j<myArray[i].length; j++) {
        myArray[i][j] = i*j;
    }
}
```

Les Collections



[Javadoc]



Les Collections

- Interface Collection: interface de base de la hiérarchie des collections Java. Pas d'Implémentation.
- Interface Set: gère des listes sans doublon possible.
- Interface List: gère des listes ordonnées sans rejeter les doublons.
- Interface Queue: ajoute des méthodes d'insertion, d'extraction et d'inspection pour permettre un mode d'exploitation de type FIFO (First In - First Out : premier entré, premier sorti)
- Interface Deque hérite également de l'interface Collection et ajoute des méthodes d'insertion, d'extraction et d'inspection pour permettre des modes de fonctionnement de type FIFO ou LIFO (Last In - First Out).
- Une seconde branche de l'API commence par l'interface Map. Le principe de fonctionnement de ce type d'objet est d'associer des clés uniques (donc sans doublons) à des valeurs. Chaque clé ne peut être associée qu'à une seule valeur.

Les Collections

```
List<String> maCollection = new ArrayList<String>();
```

Entre chevrons: **Type Générique**, sert à spécifier que la collection est destinée à recevoir des *string*.

Le second chevron peut être omis car inféré par le compilateur (diamond operator)

```
List<String> maCollection = new ArrayList<>();
```

Les Collections

ArrayList<T>:

- Implémentation de List, couramment utilisée.
- Proche d'un tableau "classique".
- Rapidité d'accès aux cellules par index.

LinkedList<T>:

- Liste chaînée.
- Insertions, suppressions et changements d'ordres rapides
- Accès direct par index plus couteux.

Les Collections

- Queue<T>:
  - Implémente un comportement de type FIFO (First In First Out).
  - L'ordre de lecture correspond à l'ordre d'entrée dans la Queue
  - LinkedList<T> Implémente l'interface Queue.
- Stack<T>:
  - Implémente un comportement de type LIFO (Last In First Out).

Les Collections

HashMap<K,V>:

Permet d'indexer rapidement des éléments de type *V* en fonction d'une clé de type *K*.

```
Map<String, String> usersById = new HashMap<>();

usersById.put("1A", "John");

usersById.put("2A", "Jack");

usersById.put("3C", "Jane");

usersById.get("2A"); // "Jack"
```

Iterable & Iterator

Les descendants de *Collection* implémentent l'interface *Iterable.*

C'est ce qui les autorise à utiliser la syntaxe *for (Type element: Liste)*

On peut donc en obtenir un *Iterator* permettant de parcourir rapidement la collection:

```
List<String> myList = new ArrayList<>();
Iterator<String> iterator = myList.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```