Programmation Objet et UML

TP - Les Collections en Java et le modèle de conception “Itérateur”

(inspiré de sujets de TP de E. Asarin/A. Degorre et E. Bruno)

Le Modèle de conception “Itérateur” a été introduit en 1994 dans le célèbre livre du “Gang of Four” (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides) : *Design Patterns - Catalogue de modèles de conceptions réutilisables* (titre français). Ce livre a été le premier à formaliser la notion de “Modèle de Conception” (“Design Pattern” en anglais). L'itérateur permet de parcourir de façon générique des collections d'objets (listes, files, ensembles, tableaux associatifs), voir Fig. 1 (les noms des méthodes sont ceux de Java 1.6).

Fig. 1 – Interface Iterator

Il est aujourd'hui utilisé nativement dans les librairies standard de C++ et Java, notamment les Collections Java. La Fig. 2 montre la structure des interfaces héritant de Collection (à noter que Collection hérite de l'interface Iterable). L'interface List définit une Liste ordonnée d'éléments. L'interface Set définit un Ensemble non ordonné d'éléments où chaque élément n'apparait qu'une unique fois (ensemble mathématique). L'interface Queue définit une File FIFO où l'on n'a accès qu'à une extrémité ou aux deux (Deque, pour “double ended queue”). L'interface Map, bien qu'elle ne soit pas une Collection et n'est pas itérable, définit des tableaux associatifs (association clé/valeur) qu'il est utile de connaître.

Chaque interface est implémentée de façon native par plusieurs classes différentes (les plus courantes sont données dans la Fig. 2) Il existe également des implémentations robustes pour le multi-threading (ConcurrentLinkedQueue par exemple), ou pour d'autres usages spécifiques (voir BlockingQueue ou NavigableSet par exemple).

Attention, la complexité des opérations élémentaires diffère grandement suivant l'implémentation (voir Fig. 3). Il convient donc de savoir de quelle implémentation on a besoin en fonction de ces critères !

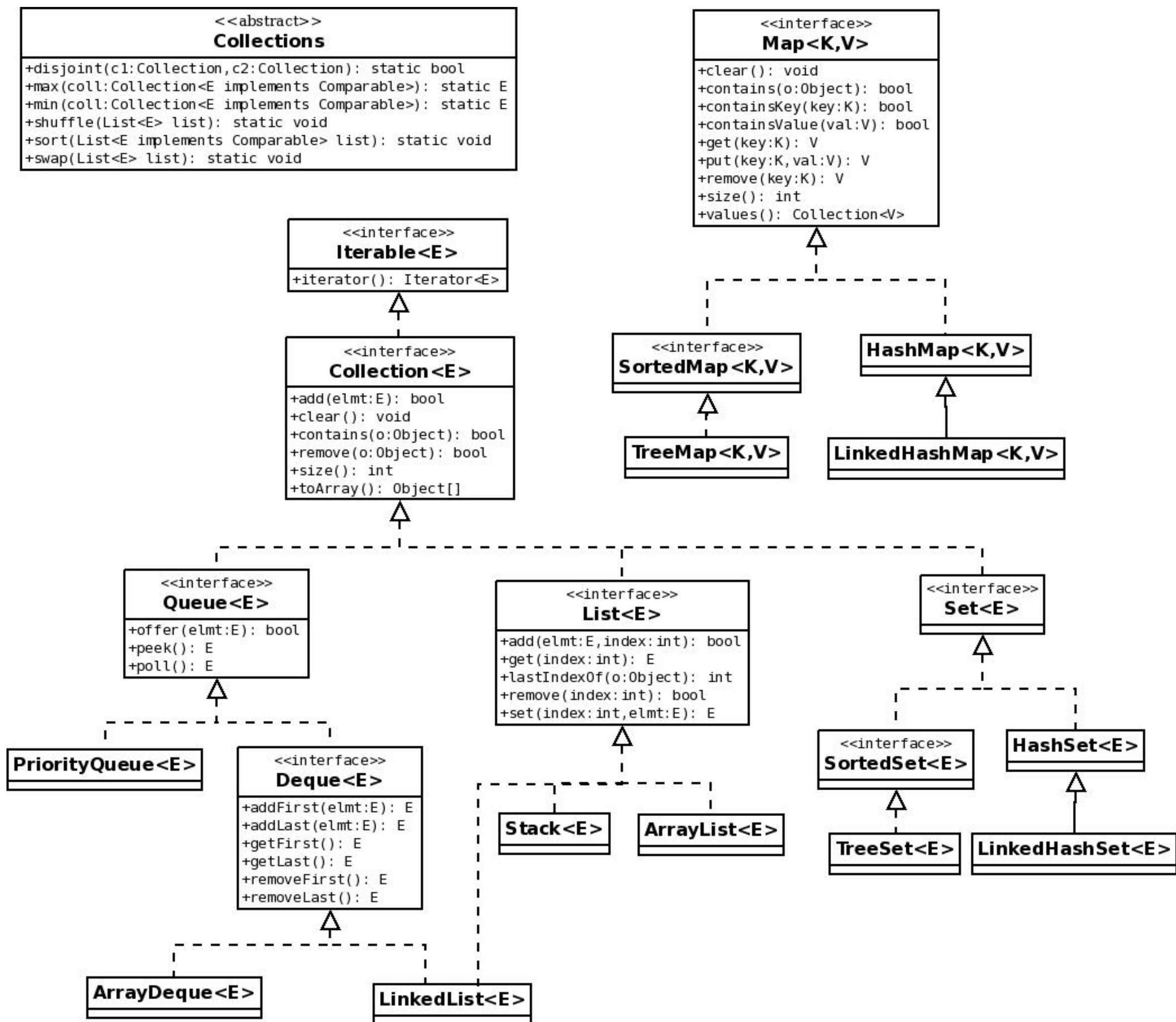

Fig. 2 – Diagramme de Classes (partiel) des collections et des map dans Java 1.6

	get	*add*	*contains*	*next*	*remove(O)*	*Iterator.remove*
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)
CopyOnWriteArrayList	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

(a) Implémentations de List

	add	*contains*	*next*	*Note*
HashSet	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashSet	O(1)	O(1)	O(1)	
CopyOnWriteArraySet	O(n)	O(n)	O(1)	
EnumSet	O(1)	O(1)	O(1)	
TreeSet	O(log n)	O(log n)	O(log n)	
ConcurrentSkipListSet	O(log n)	O(log n)	O(1)	

(b) Implémentations de Set

	offer	*peek*	*poll*	*size*
PriorityQueue	O(log n)	O(1)	O(log n)	O(1)
ConcurrentLinkedQueue	O(1)	O(1)	O(1)	O(n)
ArrayBlockingQueue	O(1)	O(1)	O(1)	O(1)
LinkedBlockingQueue	O(1)	O(1)	O(1)	O(1)
PriorityBlockingQueue	O(log n)	O(1)	O(log n)	O(1)
DelayQueue	O(log n)	O(1)	O(log n)	O(1)
LinkedList	O(1)	O(1)	O(1)	O(1)
ArrayDeque	O(1)	O(1)	O(1)	O(1)
LinkedBlockingDequeue	O(1)	O(1)	O(1)	O(1)

(c) Implémentations de Queue

	get	*containsKey*	*next*	*Note*
HashMap	O(1)	O(1)	O(h/n)	h is the table capacity
LinkedHashMap	O(1)	O(1)	O(1)	
IdentityHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
EnumMap	O(1)	O(1)	O(1)	
TreeMap	O(log n)	O(log n)	O(log n)	
ConcurrentHashMap	O(1)	O(1)	O(h/n)	h is the table capacity
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)	

(d) Implémentations de Map

FIG. 3 – Complexités des opérations pour certaines implémentations de Collection et Map (source : *Java Generics and Collections*, M. Naftalin, P. Wadler, O' Reilly 2006)

Partie 1 - ensembles

1. Ouvrez l'URL `http://download.oracle.com/javase/6/docs/api/` (ou tapez "java api 6" dans votre moteur de recherche favori). Il s'agit de l'API de java 1.6, très pratique pour les programmeurs Java. Il existe également un tutorial officiel (en anglais) sur les Collections à l'adresse `http://download.oracle.com/javase/tutorial/collections/index.html`.
2. Créez une classe Animal et un ensemble d'animaux en utilisant un HashSet *non générique*.
3. Tous les animaux ont un nom. Indiquez que deux animaux sont égaux s'ils ont le même nom en redéfinissant la méthode equals.
4. Peuplez votre ensemble d'animaux.
5. Affichez l'ensemble d'animaux en utilisant l'itérateur fourni par la méthode iterator()
6. Ajoutez plusieurs animaux ayant le même nom. Affichez de nouveau. Que constatez-vous ?
7. Transformez le support physique de cet ensemble en un TreeSet et testez-le.
8. Rendez votre ensemble spécifique à la classe Animal (utilisez la généricité présente dans Collection)

Partie 2 - listes

1. Créez une liste d'animaux à partir des animaux de l'ensemble de la Partie 1 (utilisez le bon constructeur de la classe ArrayList)
2. Ajoutez plusieurs animaux ayant le même nom dans la liste, et affichez-la.
3. Ecrivez une méthode statique prenant en paramètre une Collection d'Animaux et qui affiche le nom de ceux dont le nom commence par "C". Utilisez pour cela l'itérateur fourni dans toute classe implémentant l'interface Collection. Testez-la avec l'ensemble d'animaux et la liste d'animaux.
4. On veut comparer les animaux en fonction de leur nom via l'interface Comparable. Notez que la classe String est elle-même comparable, profitez-en !
5. Triez la liste d'animaux avec la méthode statique sort(List l) de la classe Collections (avec un "s" !) et affichez-la.

Partie 3 - maps

1. On associe à chaque animal un tatouage composé de son nom et d'un entier unique. Ecrivez la classe Tatouage correspondante. L'entier unique pourra être généré via un compteur entier statique.
2. Créez une Map d'animaux (avec la généricité) contenant des animaux tatoués, et indexés par leur tatouage (le choix de la classe utilisée vous est laissé).
3. Créez une nouvelle classe qui hérite de la classe implémentant Map que vous venez d'utiliser, et faites-lui implémenter l'interface Iterable. Testez-la avec la méthode statique écrite à la question 2.3.