Introduction à la programmation et au développement d'applications en Java

Collections

Introduction

- ❏ Les tableaux : une structure « bas niveau »
- ❏ Les collections
- ❏ Définir des collections et itérer
- ❏ Garantir un ordonnancement : les listes
- ❏ Garantir l'unicité : les ensembles
- ❏ Garantir un ordre de modification : piles et queues
- ❏ Association clés/éléments : les maps
- ❏ Opérations communes à toutes les collections

Collections

- **Les collections sont importantes**
  - Elle résolvent les problèmes posés par les tableaux simples

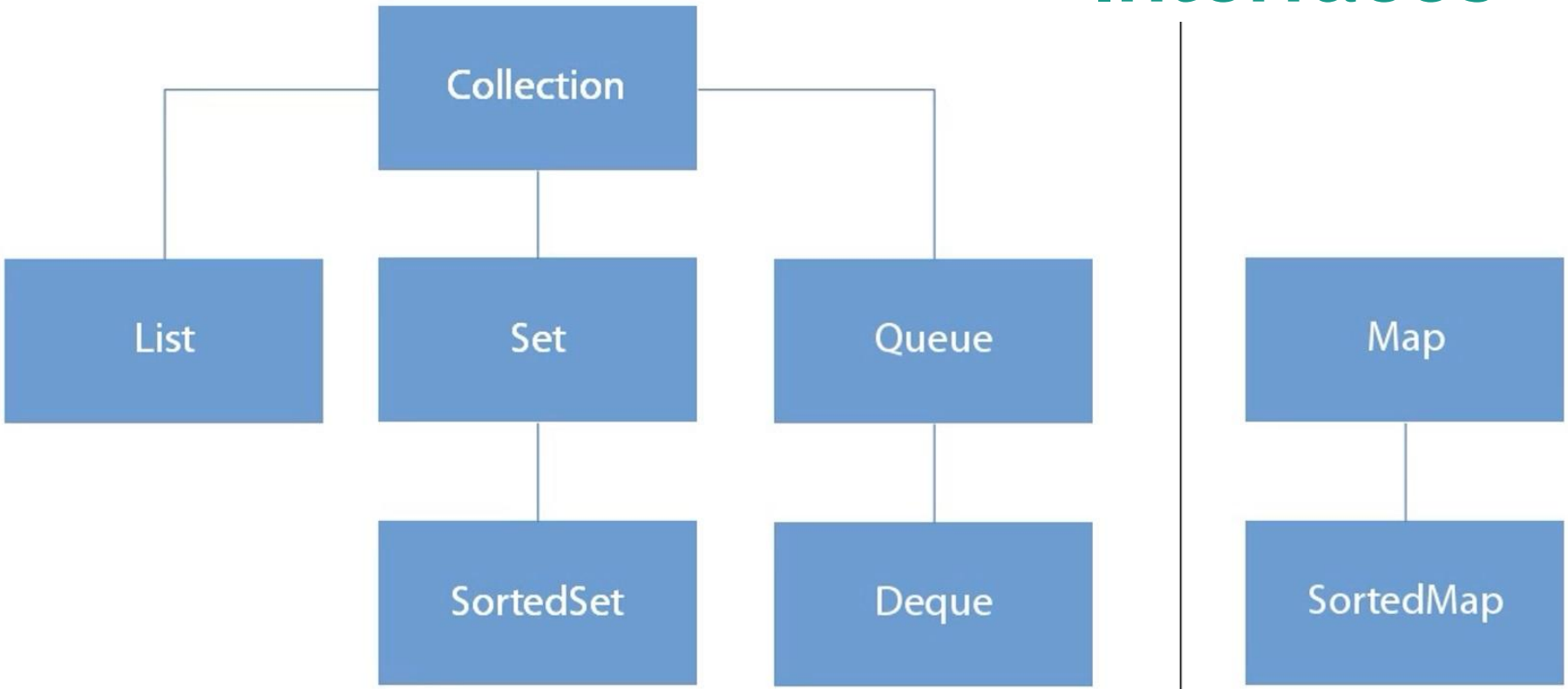
Collections

- **Les collections sont importantes**
  - Elle résolvent les problèmes posés par les tableaux simples
- **Les collections sont très importantes**
  - Dès sa conception, la *Java Class Library* disposait de classes collections standardisées
  - Tout langage de programmation moderne possède aujourd'hui sa propre implémentation des collections

Collections

- **Les collections sont importantes**
  - Elle résolvent les problèmes posés par les tableaux simples
- **Les collections sont très importantes**
  - Dès sa conception, la *Java Class Library* disposait de classes collections standardisées
  - Tout langage de programmation moderne possède aujourd'hui sa propre implémentation des collections
- **Les collections sont essentielles**
  - Quasiment tout programme faisant quelque chose d'intéressant devrait utiliser des collections

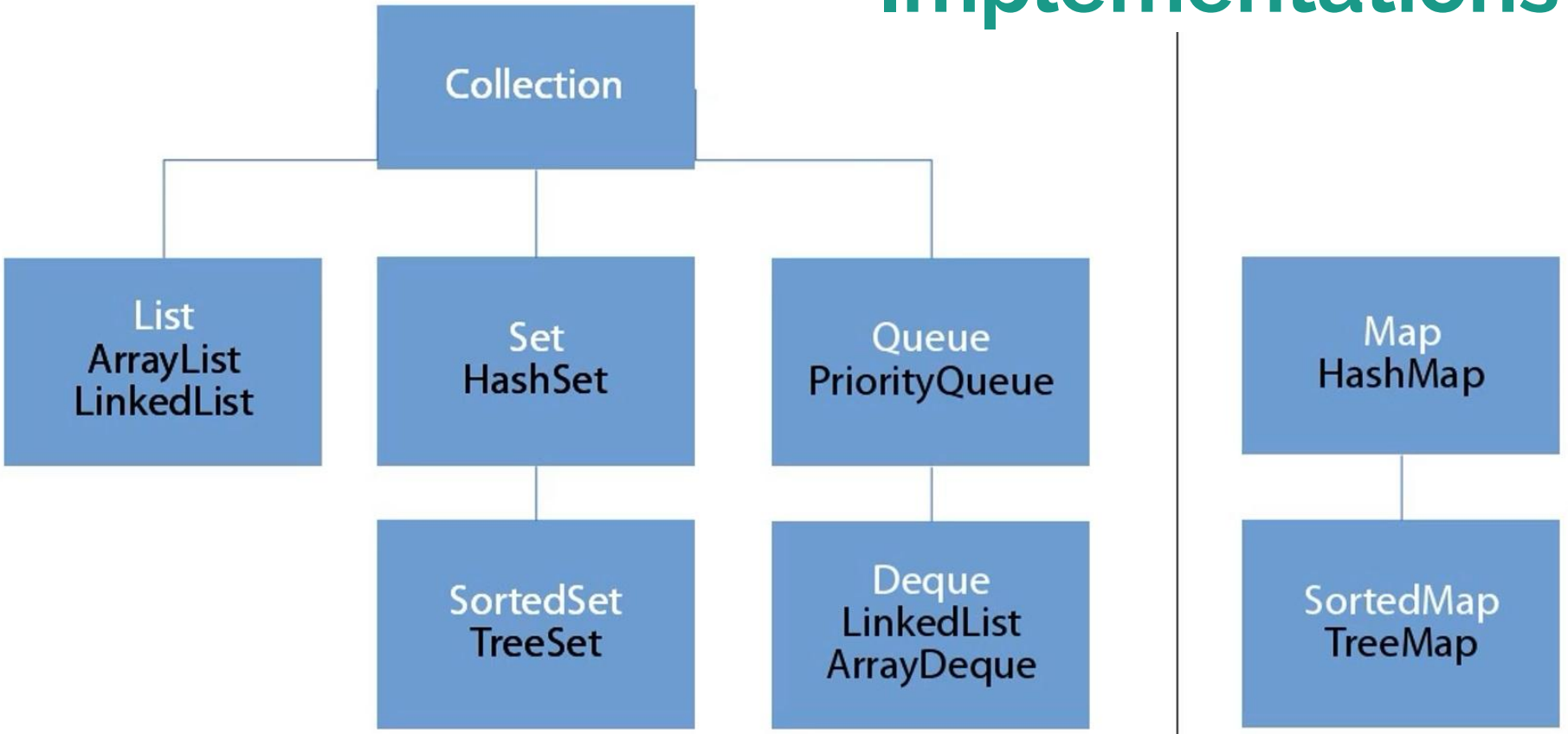
Interfaces



Interfaces vs. implémentations

- **Interface :**
  - Plusieurs structures de données implémentent la même interface (ex: `List`)
  - Décrit les caractéristiques fonctionnelles (ajouter, trouver…)
  - On utilise souvent l'interface comme type de variable plutôt qu'une implémentation spécifique
  - Souvent, une interface a une certaine implémentation très largement populaire (`ArrayList` pour `List`)
- **Implémentation :**
  - Une structure de données spécifique (tableau derrière `ArrayList`)
  - La structure de données utilisée implique des performances différentes en fonction des opérations
  - Concrète et instanciable

Implémentations



Choisir la collection appropriée

- Éléments rangés par clés ?
  - *Oui* : Ordre important ?
    - *Oui* : `SortedMap`
    - *Non* : `Map`
  - *Non* : Éléments uniques ?
    - *Oui* : Ordre important ?
      - *Oui* : `SortedSet`
      - *Non* : `Set`
    - *Non* : Premier entré, premier sorti (FIFO) ?
      - *Oui* : `Queue` ou `Deque`
      - *Non* : Dernier entré, premier sorti (LIFO) ?
        - *Oui* : `Deque`
        - *Non* : `List`

Fonctionnalités communes

- Toutes ces collections exposent des **fonctionnalités communes**
- Applicables quelles que soient la collection et l'implémentation spécifique choisie
- Cela concerne principalement :
  - **l'itération** (parcours)
  - **la taille**
  - **la mutation basique**

Fonctionnalités communes

`size()`	Retourne le nombre d'éléments de la collection
`isEmpty()`	Vrai si `size() == 0`
`add(element)`	Ajoute l'élément au début de la collection
`addAll(collection)`	Ajoute tous les éléments de `collection`
`remove(element)`	Supprime l'élément
`removeAll(collection)`	Supprime tous les éléments contenus dans `collection`
`retainAll(collection)`	Supprime tous les éléments qui ne sont pas contenus dans `collection`
`contains(element)`	Vrai si l'élément est dans la collection
`containsAll(collection)`	Vrai si tous les éléments de `collection` sont dans la collection
`clear()`	Supprime tous les éléments de la collection

Démo : interface `Collection`

Collections avec ordre : les listes

- Interface **`java.util.List`**
- Deux implémentations principales : **`ArrayList`** et **`LinkedList`**
- Une liste est une collection qui définit un **ordre d'itération**
- Donc : chaque élément de la liste a un **index** (comme pour un tableau)
- L'interface **`List`**, qui dérive de **`Collection`**, ajoute des méthodes au contrat qui sont propres à cette caractéristique

```
void add(int index, E e);
E get(int index);
E remove(int index);
E set(int index, E element);
boolean addAll(int index, Collection<? extends E> c);
```

Chaque élément a un index

L'index est un entier représentant la position de l'élément dans la liste

On peut modifier la liste en utilisant les index

```
int indexOf(Object o);
int lastIndexOf(Object o);
```

Recherche d'index (*Lookup*)

Retourne l'index du premier/dernier trouvé

Retourne `-1` si l'élément n'est pas trouvé

```
List<E> subList(int fromIndex, int toIndex);
```

Les *sublists* sont une « vue » de liste

Vue modifiée => Liste modifiée aussi

`fromIndex` est inclusif ; `toIndex` est exclusif

Démo : interface `List`

Implémentations de `List` : `ArrayList`

- **`ArrayList`** : en interne, stocke les éléments dans un **tableau Java classique**
- Principaux avantages / inconvénients :
  - **+** accès direct par index rapide
  - **+** parcours rapide (cache CPU notamment)
  - **-** obligation de redimensionner le tableau si on ajoute beaucoup d'éléments
  - **-** insertion/suppression d'éléments coûteuse (surtout en début de liste)

Implémentations de `List` : `LinkedList`

- **`LinkedList`** : en interne, stocke les éléments dans une **liste chaînée double**
- Principaux avantages / inconvénients :
  - **+** insertion/suppression peu coûteuse
  - **+** aucun redimensionnement nécessaire
  - **-** parcours sensiblement moins rapide
  - **-** accès direct impossible (sauf si on a une référence au noeud précis)

L'interface `List` : résumé

- **`List`** : collection avec **notion d'ordre et indexée**
- **`ArrayList`** est de loin l'implémentation la plus populaire : si vous ne savez pas trop quelles seront les caractéristiques d'accès à votre liste, c'est le meilleur choix par défaut
- Beaucoup d'ajouts vers la fin, d'itérations, d'accès par index ? **`ArrayList<E>`**
- Beaucoup d'ajouts/suppression arbitraires ou vers le début ? **`LinkedList<E>`**
- Il existe d'autres implémentations de **`List`** moins utilisées
  - notamment une variante de **`ArrayList`** spécialisée pour les contextes *concurrents* : **`CopyOnWriteArrayList`**

Collections garantissant l'unicité : les sets

- Interface **`java.util.Set`**
- Implémentations principales : **`HashSet, TreeSet, EnumSet`**
- Un **set** (ensemble) est une collection d'**éléments distincts**, qui garantit qu'aucun doublon n'est présent
- Cela pose la question : quand dit-on que deux éléments sont identiques ?
- L'interface **`Set`**, qui dérive de **`Collection`**, n'y ajoute pas de méthodes (contrairement à **`ArrayList`**) ; elle se contente de garantir l'unicité des éléments
- Deux sous-interfaces : **`SortedSet, NavigableSet`**

Démo : interface `Set`

Implémentations de `Set` : `HashSet`

- **`HashSet`** : en interne, stocke les éléments dans une **`HashMap`** **Java** (clé => valeur)
- Se redimensionne si nécessaire
- Nécessite que la classe des objets stockés possède une bonne implémentation de la méthode **`hashCode()`**
- **`hashCode()`** retourne un entier, un « index » qui va permettre de localiser l'élément dans la **`HashMap`**
- Si le **`hashCode`** est bien implémenté (entiers bien répartis), le **`HashSet`** aura de bonnes performances ; **`HashSet`** sera votre choix par défaut pour les sets, comme **`ArrayList`** l'est pour les listes

Le « contrat » hashCode/equals

- Règle primordiale à respecter :

  SI : `obj.equals(autre)`

  ALORS : `obj.hashCode() == autre.hashCode()`

- Le hashCode est utilisé pour localiser l'élément dans la structure interne
- Si les hashCode pour deux objets égaux ne retournent pas la même valeur, le HashSet pourra stocker deux objets identiques à deux locations différentes (doublon)
- Laissez votre IDE générer la méthode **`hashCode()`** pour vous ; vérifiez toujours que tous les champs utilisés pour évaluer **`equals()`** sont utilisés dans le calcul du hashCode (pas un de plus, pas un de moins)

Démo : `hashCode()` / `equals()`

Implémentations de `Set` : `TreeSet`

- **`TreeSet`** : en interne, stocke les éléments dans une **`TreeMap` Java** (arbre binaire)
- Conserve l'ordre des éléments (ils sont toujours triés)
- Les éléments devront donc définir un ordre (être comparable entre eux) :
  - soit en implémentant l'interface **`Comparable<E>`**
  - soit en fournissant au constructeur de **`TreeSet`** un **`Comparator`** qui sait comment comparer les éléments de ce type :

```
// Dans la classe élément (ici Produit)
public static final Comparator<Produit> TRI_PAR_POIDS
                        = Comparator.comparing(Produit::getPoids);
```

```
// Dans la classe qui contient la collection
private final Set<Produit> produits = new TreeSet<>(Produit.TRI_PAR_POIDS);
```

Démo : `TreeSet`

Spécialisation de `Set` : `SortedSet` et `NavigableSet`

- Étendent les capacités de l'interface **`Set`** (qui à la base ne contient rien de plus que **`Collection`**, excepté la caractéristique d'unicité des éléments)
- Ces deux interfaces forcent l'ensemble d'éléments à être **ordonné**
- Mais c'est différent d'une liste :
  - pas de doublon (**`Set`**)
  - pas de notion d'index

```
E first();
E last();

SortedSet<E> headSet(E toElement);
SortedSet<E> tailSet(E fromElement);
SortedSet<E> subSet(E fromElement, E toElement);
```

SortedSet

Définit un ordre, mais on ne peut pas utiliser d'index

On peut parler du premier, du dernier, des premiers jusqu'à un certain élément (exclu), des derniers à partir d'un certain élément (inclus), d'un sous-ensemble entre deux éléments
Ces sous-ensembles réagissent comme des subLists (modif affectent toutes les vues)

```
E lower(E e);            // <
E higher(E e);           // >
E floor(E e);            // <=
E ceiling(E e);          // >=
E pollFirst();
E pollLast();
```

NavigableSet

Dérive de l'interface SortedSet et étend encore ses capacités, pour permettre de « se déplacer » dans l'ensemble

On peut alors parler du précédent, du suivant (strictement ou pas), et retourner le premier ou le dernier élément (en le supprimant de l'ensemble)

Démo : interface `SortedSet`

L'interface `Set` : résumé

- **`Set`** : Éléments **distincts**
- Implémentations basées sur :
  - égalité/hachage (**`HashSet`**)
  - ou sur un arbre binaire (**`TreeSet`**)
- **`TreeSet`** implémente aussi **`SortedSet`** et **`NavigableSet`**, ce qui permet d'avoir une structure de données à éléments uniques, triée, et « navigable »

Collections avec ordre de modification : Piles et Queues

- Queues (FIFO)
- *Priority Queues*
- Piles ou *Stacks* (LIFO)
- *Deques* (queues à double entrées/sorties)

Queue : *First-In, First-Out*

- FIFO : **Premier arrivé, premier sorti**
- Autrement dit : les éléments **sortent dans l'ordre où ils sont arrivés**
- Métaphore évidente : la **file d'attente**
- Une queue peut avoir une taille bornée

```
boolean add(E e);

boolean offer(E e);
```

Queue - Ajout d'un élément

problème : `add` lance une exception si la queue (bornée) est pleine (renvoie `false` seulement si doublon — requis de l'interface `Collection`)

solution : l'interface `Queue` ajoute la méthode `offer`

⇒ retourne `false` si la queue est pleine

```
E remove();

E poll();
```

Queue - Enlever *et* retourner l'élément en tête

`remove` lance une **exception** si la queue est vide

`poll` retourne **null** si la queue est vide

```
E element();

E peek();
```

Queue - Lire l'élément en tête *sans supprimer*

`element` lance une **exception** si la queue est vide

`peek` retourne **null** si la queue est vide

Démo : interface `Queue`

Priority Queue

- Chaque élément a une « importance », donnée par vous-même
- ⇒ Notion de **priorité entre les éléments**
- ⇒ En pratique un **ordonnancement** (comme pour une collection triée)
- La queue va gérer en interne cet ordonnancement afin de pouvoir à tout moment retourner **l'élément dont la priorité est la plus importante**
- Autrement dit : les éléments **sortent dans leur ordre de priorité**, et non plus dans leur ordre d'arrivée
- Métaphore évidente : la file d'attente toujours, mais **aux urgences**

Démo : `PriorityQueue`

Piles : *Last In, First Out*

- Pile = *Stack* en anglais
- LIFO : **Dernier arrivé, premier sorti**
- Autrement dit : les éléments **sortent dans l'ordre inverse d'arrivée**
- Métaphore : em**pile**ment/dé**pile**ment de blocs Lego un à un
- L'implémentation `java.util.Stack` est *deprecated*, **ne l'utilisez pas**
- En Java, on implémentera une pile avec l'interface `Deque`

`Deque` : queues à double entrées/sorties

- **D-e-que** = *Double-Ended-Queue*
- **`Queue`** : une entrée d'un côté, une sortie de l'autre
- **`Deque`** : une entrée et une sortie des deux côtés
- On peut donc utiliser une **`Deque`** pour implémenter une queue ou une pile, en fonction des entrées/sorties que l'on décide d'utiliser
- L'interface **`Deque`** dérive de l'interface **`Queue`**, elle reprend les mêmes méthodes en ajoutant **`First/Last`** pour indiquer de quelle entrée/sortie on parle

```
boolean addFirst(E e);
boolean addLast(E e);

boolean offerFirst(E e);
boolean offerLast(E e);
```

Deque - Ajout d'un élément

`add*` lance une exception si la queue est pleine

`offer*` retourne `false` si la queue est pleine

```
E removeFirst();
E removeLast();

E pollFirst();
E pollLast();
```

Deque - Enlever *et* retourner un élément

`remove*` lance une exception si la queue est vide

`poll*` retourne `null` si la queue est vide

```
E getFirst();
E getLast();

E peekFirst();
E peekLast();
```

Deque - Lire l'élément d'un des deux côtés *sans supprimer*

`get*` lance une exception si la queue est vide

`peek*` retourne `null` si la queue est vide

(petite incohérence : l'interface `Queue` utilise le nom `element()` au lieu de `get()`)

```
void push(E e);

E pop();
```

Deque - noms usuels pour la *pile*

`push` pour mettre un élément sur la pile

`pop` pour enlever et retourner l'élément « du dessus »

Démo : pile avec `Deque`

`Deque` : les implémentations

- Deux implémentations principales pour les utilisations classiques FIFO/LIFO et *double ended queue* : **`ArrayDeque`** et **`LinkedList`**
- **`ArrayDeque`** est largement l'implémentation la plus fréquente
- N'utilisez pas **`LinkedList`**
- Une implémentation pour le contrat sémantique spécifique de la *priority queue* : **`PriorityQueue`**, que nous avons vue
- Il y a d'autres implémentations de **`Queue`** spécialisées notamment dans les accès concurrentiels

Les interfaces `Queue`/`Deque` : résumé

- Queue : FIFO (interface **`Queue`**, implémentation **`ArrayDeque`**)
- Pile : LIFO (interface **`Deque`**, implémentation **`ArrayDeque`** – et non **`Stack`**)
- *Priority Queue* : prioritaire d'abord (interface Queue, implémentation **`PriorityQueue`**)
- Deque : *double ended queue* (interface **`Deque`**, implémentation **`ArrayDeque`**)
- Ne pas utiliser **`LinkedList`** pour implémenter une queue
- De nombreuses situations de développement requièrent naturellement l'un de ces concepts, pensez-y et utilisez les implémentations que fournit la bibliothèque du langage

Collections de paires : `Map`

- Pourquoi utiliser une Map ?
- Vues sur Map
- **`SortedMap`** et **`NavigableMap`**
- Amélioration de Java 8
- Implémentations : laquelle ?

Map = Dictionnaire

- Dictionnaire : plutôt que d'avoir des éléments seuls, une entrée d'un dictionnaire associe une **clé** à une **valeur** (clé => valeur)
- Les **clés sont uniques** (pas de doublon), **mais** pas forcément les valeurs
- L'unicité des clés est garantie par le contrat **`equals/hashCode`** ou par **`Comparator/equals`**
- Métaphore : un vrai dictionnaire (mot => définition)
- En Java, on utilise le terme **Map** (interface) plutôt que dictionnaire (car il existe une classe **`Dictionary`** qui est *deprecated*)

Démo : interface `Map`

```
V put(K key, V value);

void putAll(Map<? extends K, ? extends V> values);
```

Map - Ajout et remplacement d'une paire (clé => valeur)

`put` ajoute la paire et retourne `null`
OU écrase la valeur si clé existante et retourne l'ancienne valeur

`putAll` pour ajouter toute une `Map` d'un coup (écrasement si nécessaire)

Comportement si clé ou valeur `null` est spécifique à l'implémentation

```
V get(Object key);

boolean containsKey(Object key);
boolean containsValue(Object value);
```

Map - Recherche de clés ou valeurs

`get` retourne `null` si clé non trouvée

`contains*` retourne `true/false` selon que la clé/valeur est dans la Map ou non

`Object` plutôt que `K` et `V` pour plus de flexibilité (?)

```
V remove(Object key);

void clear();
```

Map - Suppression d'éléments

`remove` retourne `null` si clé non trouvée

`clear*` supprime toutes les paires de la Map

```
int size();

boolean isEmpty();
```

Map - Taille ?

Même sémantique que l'interface `Collection`

Map != Collection ?

- **`Map`** est la seule collection dont l'interface ne dérive pas de **`Collection`**
- **`Map`** ne fonctionne pas bien avec le contrat de **`Collection`** à cause du fait qu'elle gère des paires clé => valeur, et non des « éléments unitaires »

Vues sur Map

- Comme pour les listes (`List - subList(2, 5)`) et les ensembles (`SortedSet - headSet(toElement)`), on peut avoir des « vues » sur une `Map`
- Rappel : une modification sur une vue affecte la collection, pas seulement la vue
- `keySet()` : vues sur **clés**
- `values()` : vues sur **valeurs**
- `entrySet()` : vues sur **paires**

Démo : vues sur `Map`

`SortedMap` et `NavigableMap`

- Ces deux interfaces dérivent de **`Map`**
  (comme **`SortedSet`** et **`NavigableSet`** dérivent de **`Set`**)
- **`SortedMap`** définit un ordre (ascendant) sur les clés de la Map
- **`NavigableMap`** dérive de **`SortedMap`** pour ajouter la notion de déplacement dans la Map
- **`NavigableMap`** est peu utilisée par les développeurs Java, surtout par manque de connaissances : elle remplace avantageusement **`SortedMap`**

```
K firstKey();
K lastKey();

SortedMap<K, V> tailMap(K fromKey);
SortedMap<K, V> headMap(K toKey);
SortedMap<K, V> subMap(K fromKey, K toKey);
```

SortedMap - Ordonnancement

Les vues sont basées sur les clés (from, to), comme d'habitude elles sont modifiables et les modifications sont refletées sur la collection et les autres vues

```
Entry<K, V> firstEntry();
Entry<K, V> lastEntry();

Entry<K, V> pollFirstEntry();
Entry<K, V> pollLastEntry();
```

NavigableMap - Récupérer les paires

`poll*` supprime l'élément de la Map et le retourne

```
Entry<K, V> lowerEntry(K key);    // <
Entry<K, V> higherEntry(K key);   // >

K lowerKey(K key);                // <
K higherKey(K key);               // >
```

NavigableMap - Navigation par clés (strictement différentes)

Aller à la paire ou la clé précédente/suivante,
en « sautant » celles qui sont égales à la clé courante

```
Entry<K, V> floorEntry(K key);    // <=
Entry<K, V> ceilingEntry(K key);  // >=

K floorKey(K key);                // <=
K ceilingKey(K key);              // >=
```

NavigableMap - Navigation par clés (différentes ou égales)

Aller à la paire ou la clé précédente/suivante,
en incluant celles qui sont évaluées comme égales

```
NavigableMap<K, V> descendingMap();
NavigableSet<K> descendingKeySet();

NavigableSet<K> navigableKeySet();
```

NavigableMap

`descending*` retourne une vue inversée de la Map

`navigableKeySet` retourne le `NavigableSet` que ne peut pas retourner `keySet()` (pour des raisons de rétro-compatibilité)

```
NavigableMap<K, V> tailMap(K fromKey, boolean inclusive);
NavigableMap<K, V> headMap(K toKey, boolean inclusive);

NavigableMap<K, V> subMap(K fromKey, boolean fromInclusive,
                          K toKey, boolean toInclusive);
```

NavigableMap - Vues

Les booléens permettent de préciser si la clé est incluse ou non

Un peu de pratique...

- Quelles interface/implémentation pour concevoir :
  - une collection ordonnée de patients attendant chez le médecin ?
  - un fichier non-ordonné de patients qui ont le même médecin traitant ?
  - un fichier ordonné de patients d'une ville, chacun associé à leur médecin traitant ?
  - une collection non-ordonnée d'objets **`CompteEnBanque`** uniques ?

Un peu de pratique...

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

- Pourquoi le type de la variable est **`Map`** et non **`HashMap`** ?
- Pourquoi la ligne suivante causerait-elle une erreur de compilation ?

```
etudiants.put("u0012345", "Sam");
```