Collections

Université de Nice - Sophia Antipolis

Version 5.8 – 14/2/08

Richard Grin

Plan du cours

- Généralités sur les collections
- Collections et itérateurs
- Maps
- Utilitaires : trier une collection et rechercher une information dans une liste triée

Définition

- Une collection est un objet qui contient d'autres objets
- Par exemple, un tableau est une collection
- Le JDK fournit d'autres types de collections sous la forme de classes et d'interfaces
- Ces classes et interfaces sont dans le paquetage **`java.util`**

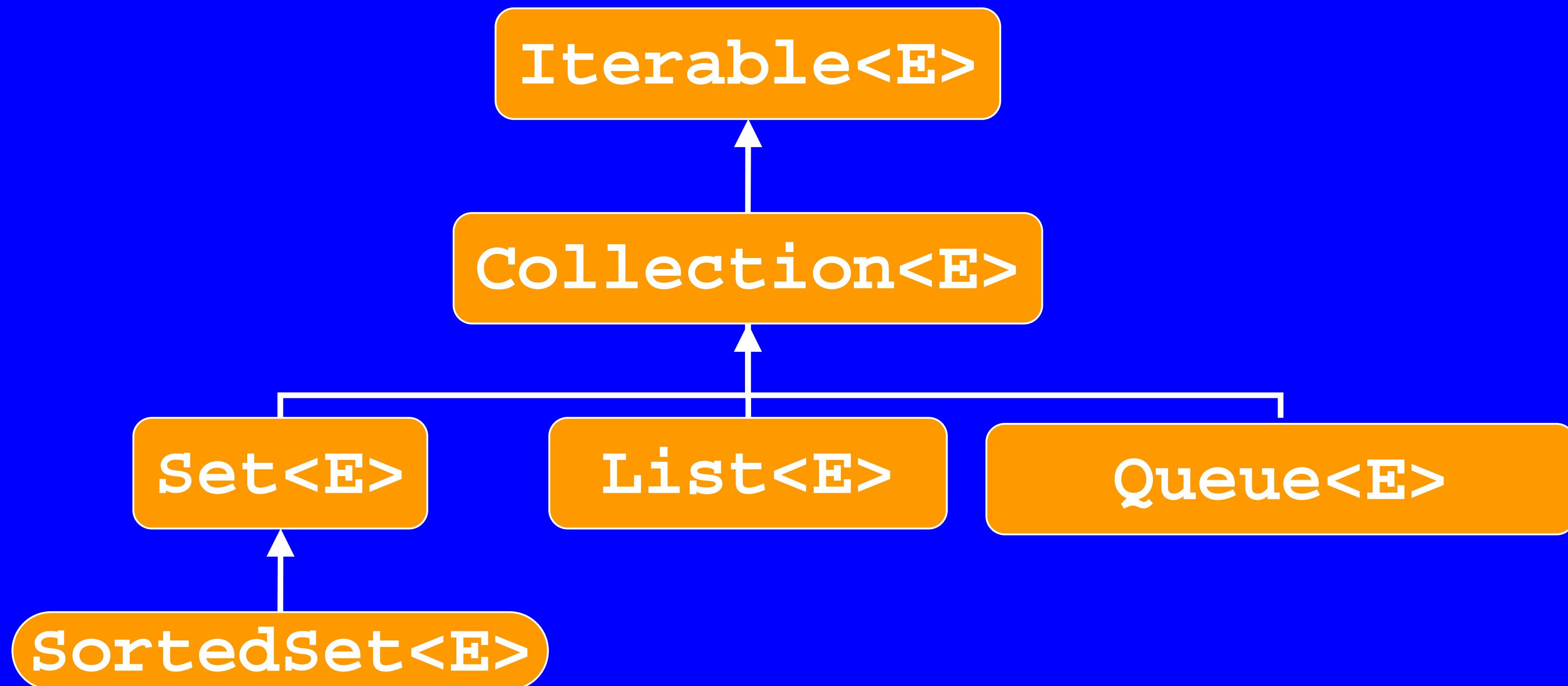
Généricité

- Avant le JDK 5.0, il n'était pas possible d'indiquer qu'une collection du JDK ne contenait que des objets d'un certain type ; les objets contenus étaient déclarés de type **`Object`**
- A partir du JDK 5.0, on peut indiquer le type des objets contenus dans une collection grâce à la généricité : **`List<Employe>`**
- Ce cours privilégie les collections génériques

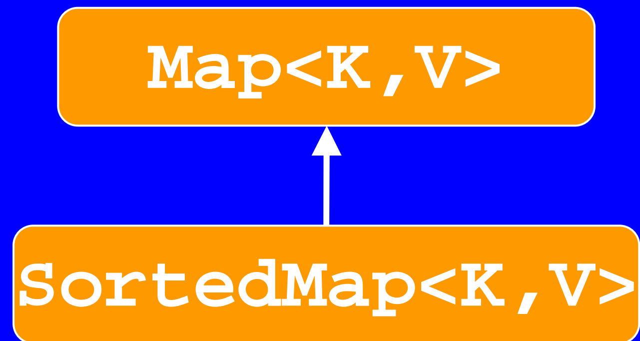
Les interfaces

- Des interfaces dans 2 hiérarchies principales :
  - `Collection<E>`
  - `Map<K,V>`
- **`Collection`** correspond aux interfaces des collections proprement dites
- **`Map`** correspond aux collections indexées par des clés ; un élément de type **`V`** d'une *map* est retrouvé rapidement si on connaît sa clé de type **`K`** (comme les entrées d'un dictionnaire ou les entrées de l'index d'un livre)

Hiérarchie des interfaces - `Collection`



Hiérarchie des interfaces – `Map`



Les classes abstraites

- **`AbstractCollection<E>`**, **`AbstractList<E>`**, **`AbstractMap<K,V>`**,… implantent les méthodes de base communes aux collections (ou *map*)
- Elles permettent de factoriser le code commun à plusieurs types de collections et à fournir une base aux classes concrètes du JDK et aux nouvelles classes de collections ajoutées par les développeurs

Les classes concrètes

- **`ArrayList<E>`**, **`LinkedList<E>`**, **`HashSet<E>`**, **`TreeSet<E>`**, **`HashMap<K,V>`**, **`TreeMap<K,V>`**,…
  héritent des classes abstraites
- Elles ajoutent les supports concrets qui vont recevoir les objets des collections (tableau, table de hachage, liste chaînée,…)
- Elles implantent ainsi les méthodes d'accès à ces objets (*get*, *put*, *add*,…)

Classes concrètes d'implantation des interfaces

		Classes d’implantations			
		Table de hachage	Tableau	Arbre balancé	Liste chaînée
Interfaces	Set<E>	HashSet<E>		TreeSet<E>	
	List<E>		ArrayList<E>		LinkedList<E>
	Map<K,V>	HashMap<K,V>		TreeMap<K,V>	
	Queue<E>		ArrayDequeue<E>		LinkedList<E>

Classes étudiées

- Nous étudierons essentiellement les classes **`ArrayList`** et **`HashMap`** comme classes d'implémentation de **`Collection`** et de **`Map`**
- Elles permettront d'introduire des concepts et informations qui sont aussi valables pour les autres classes d'implantation

Classes utilitaires

- `Collections` (avec un *s* final) fournit des méthodes `static` pour, en particulier,
  - trier une collection
  - faire des recherches rapides dans une collection triée
- `Arrays` fournit des méthodes `static` pour, en particulier,
  - trier
  - faire des recherches rapides dans un tableau trié
  - transformer un tableau en liste

Collections du JDK 1.1

- Les classes et interfaces suivantes, fournies par le JDK 1.1,
  - **`Vector`**
  - **`HashTable`**
  - **`Enumeration`**

  existent encore mais il vaut mieux utiliser les nouvelles classes du JDK 1.2
- Il est cependant utile de les connaître car elles sont utilisées dans d'autres API du JDK
- Elles ne seront pas étudiées ici en détails

2 exemples d'introduction à l'utilisation des collections et *maps*

Exemple de liste

```
List<String> l =
  new ArrayList<String>();
l.add("Pierre Jacques");
l.add("Pierre Paul");
l.add("Jacques Pierre");
l.add("Paul Jacques");
Collections.sort(l);
System.out.println(l);
```

Exemple de Map

```
Map<String, Integer> frequences =
  new HashMap<String, Integer>();
for (String  mot : args) {
  Integer freq = frequences.get(mot);
  if (freq == null)
    freq = 1;
  else
    freq = freq + 1;
  frequences.put(mot, freq);
}
System.out.println(frequences);
```

Collections et types primitifs

- Les collections de **`java.util`** ne peuvent contenir de valeurs des types primitifs
- Avant le JDK 5, il fallait donc utiliser explicitement les classes enveloppantes des types primitifs, **`Integer`** par exemple
- A partir du JDK 5, les conversions entre les types primitifs et les classes enveloppantes peuvent être implicite avec le « boxing » / « unboxing »

Exemple de liste avec (un)boxing

```
List<Integer> l = new ArrayList<Integer>();
l.add(10);
l.add(-678);
l.add(87);
l.add(7);
int i = l.get(0);
```

Exemple de liste sans (un)boxing

```
List<Integer> l = new ArrayList<Integer>();
l.add(new Integer(10));
l.add(new Integer(-678));
l.add(new Integer(87));
l.add(new Integer(7));
int i = l.get(0).intValue();
```

Interface `Collection<E>`

Définition

- L'interface **`Collection<E>`** correspond à un objet qui contient un groupe d'objets de type **`E`**
- Aucune classe du JDK n'implante *directement* cette interface (les collections vont implanter des sous-interfaces de **`Collection`**, par exemple **`List`**)

Méthodes de `Collection<E>`

renvoie `true` si la collection a été modifiée

```
* boolean add(E elt)
* void addAll(Collection<? extends E> c)
* void clear()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
Iterator<E> iterator()
* boolean remove(Object obj)
* boolean removeAll(Collection<?> c)
* boolean retainAll(Collection<?> c)
int size()
Object[] toArray()
<T> T[] toArray(T[] tableau)
```

étudié plus loin dans le cours

* : méthode optionnelle (pas nécessairement disponible)

Notion de méthode optionnelle

- Il existe de nombreux cas particuliers de collections ; par exemple
  - collections de taille fixe,
  - collections dont on ne peut enlever des objets
- Plutôt que de fournir une interface pour chaque cas particulier, l'API sur les collections comporte la notion de méthode optionnelle

Méthode optionnelle

- Méthode qui peut renvoyer une **`java.lang.UnsupportedOperationException`** (sous-classe de **`RuntimeException`**) dans une classe d'implantation qui ne la supporte pas
- Les méthodes optionnelles renvoient cette exception dans les classes abstraites du paquetage
- Exemple d'utilisation : si on veut écrire une classe pour des listes non modifiables, on ne redéfinit pas les méthodes **`set`**, **`add`** et **`remove`** de la classe abstraite **`AbstractList`**

Constructeurs

- Il n'est pas possible de donner des constructeurs dans une interface mais la convention donnée par les concepteurs des collections est que toute classe d'implantation des collections doit fournir au moins 2 constructeurs :
  - un constructeur sans paramètre
  - un constructeur qui prend une collection d'éléments de type compatible en paramètre (facilite l'interopérabilité)

Exemple de constructeur

- Un constructeur de **`ArrayList<E>`** :

```
public ArrayList(
    Collection<? extends E> c)
```

Transformation en tableau

- **`toArray()`** renvoie une instance de **`Object[]`** qui contient les éléments de la collection
- Si on veut un tableau d'un autre type, il faut utiliser la méthode paramétrée
  `<T> T[] toArray(T[] tableau)`
  à laquelle on passe un tableau du type voulu
  - si le tableau est assez grand, les éléments de la collection sont rangés dans le tableau
  - sinon, un nouveau tableau du même type est créé pour recevoir les éléments de la collection

Transformation en tableau (2)

- Pour obtenir un tableau de type **`String[]`** (remarquez l'inférence de type ; voir le cours sur la généricité) :

```
String[] tableau =
  collection.toArray(new String[0]);
```

- L'instruction suivante provoquerait une erreur de *cast* (car le tableau renvoyé a été créé par une commande du type **`new Object[n]`**) :

```
String[] tableau =
  (String[]) collection.toArray();
```

Erreur !

- Remarque : si la collection est vide, **`toArray`** renvoie un tableau de taille 0 (pas la valeur **`null`**)

Interface `Set<E>`

Définition de l'interface `Set<E>`

- Correspond à un groupe d'objets qui ne contient pas 2 objets égaux au sens de `equals` (comme les ensembles des mathématiques)
- On fera attention si on ajoute des objets modifiables : la non duplication d'objets n'est pas assurée dans le cas où on modifie les objets déjà ajoutés

Implémentation

- Classes qui implémentent cette interface :
  - **`HashSet<E>`** garantit un temps constant pour les opérations de base (**`set`**, **`add`**, **`remove`**, **`size`**)
  - **`TreeSet<E>`** garantit que les éléments sont rangés dans leur ordre naturel (interface **`Comparable<E>`**) ou suivant l'ordre d'un **`Comparator<? super E>`** ; implémente **`SortedSet`**

Méthodes de `Set<E>`

- Mêmes méthodes que l'interface **`Collection`**
- Mais les « contrats » des méthodes sont adaptés aux ensembles
- Par exemple,
  - la méthode **`add`** n'ajoute pas un élément si un élément égal est déjà dans l'ensemble (la méthode renvoie alors **`false`**)
  - quand on enlève un objet, tout objet égal (au sens de **`equals`**) à l'objet passé en paramètre sera enlevé

`HashSet<E>`

- Attention, les contrats ne fonctionnent avec **`HashSet`** que si les objets que l'on place dans **`HashSet`** respectent la règle (normalement obligatoire pour toutes les classes) « 2 objets égaux au sens de **`equals`** doivent avoir la même valeur pour la méthode **`hashCode`** »
- En effet, cette classe ne vérifie l'égalité que pour les objets qui ont le même *hashCode*

Interface `List<E>`

Définition

- L'interface **`List<E>`** correspond à un groupe d'objets indexés par des numéros (en commençant par 0)
- Classes qui implémentent cette interface :
  - **`ArrayList<E>`**, tableau à taille variable
  - **`LinkedList<E>`**, liste chaînée
- On utilise le plus souvent **`ArrayList`**, sauf si les insertions/suppressions au milieu de la liste sont fréquentes (**`LinkedList`** évite les décalages des valeurs)

Nouvelles méthodes de **List<E>**

```
* void add(int indice, E elt)
* boolean addAll(int indice,
                 Collection<? extends E> c)
E get(int indice)
* E set(int indice, E elt)
* E remove(int indice)
int indexOf(Object obj)
int lastIndexOf(Object obj)
ListIterator<E> listIterator()
ListIterator<E> listIterator(int indice)
List<E> subList(int depuis, int jusqua)
```

insertion avec décalage vers la droite

suppression avec décalage vers la gauche

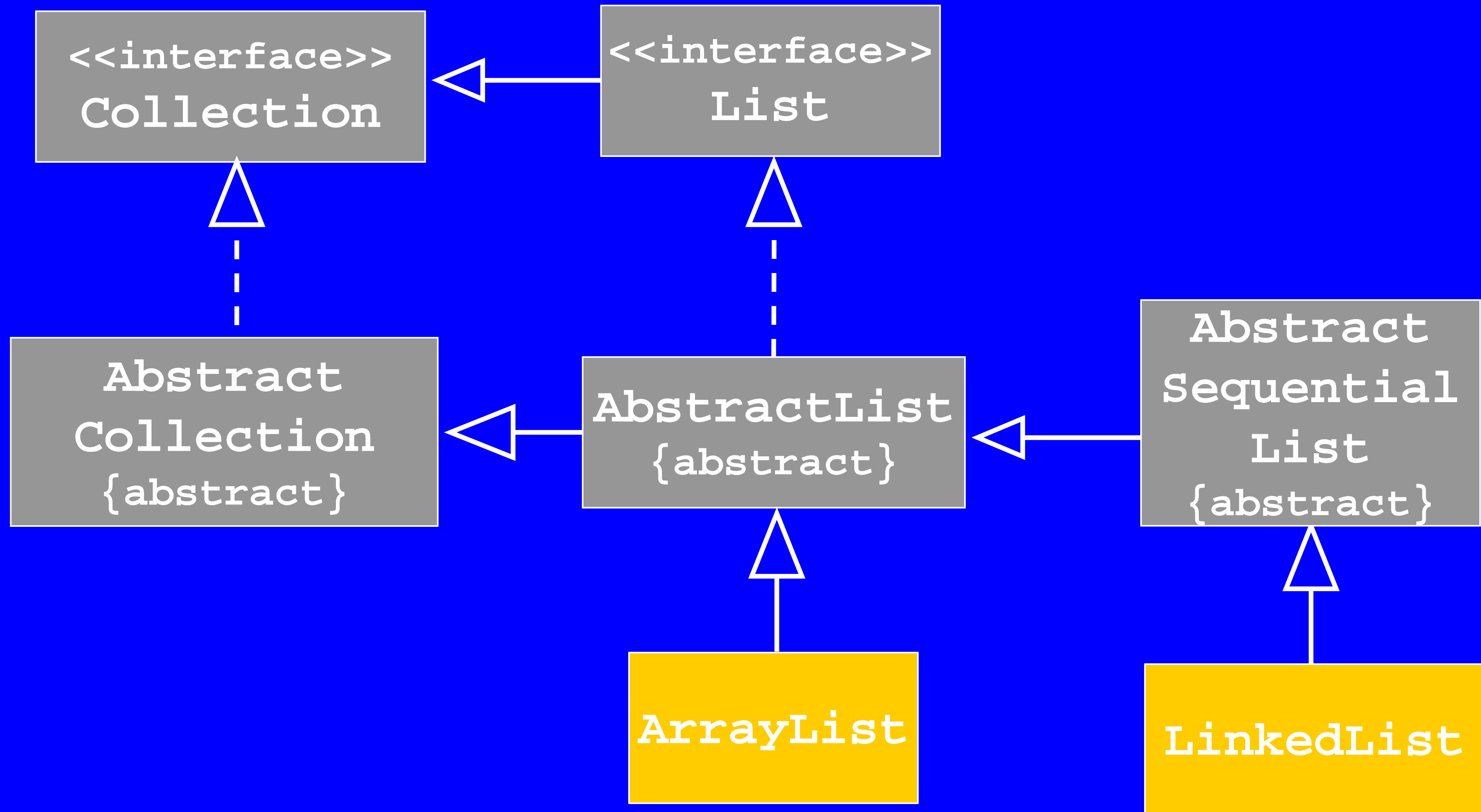
indice du 1er élément égal à **obj** (au sens de **equals**) (ou -1)

depuis inclus, **jusqua** exclu

Ne pas oublier !

- Il s'agit ici des *nouvelles* méthodes par rapport à **`Collection`**
- Dans les classes qui implémentent **`List`**, on peut évidemment utiliser toutes les méthodes héritées, en particulier
  - **`remove(Object)`**
  - **`add(E)`**

Interfaces et classes d'implantation



Classe **`ArrayList<E>`**

Fonctionnalités

- Une instance de la classe `ArrayList<E>` est une sorte de tableau qui peut contenir un nombre quelconque d'instances d'une classe E
- Les emplacements sont indexés par des nombres entiers (à partir de 0)

Constructeurs

- **`ArrayList()`**
- **`ArrayList(int taille initiale)`** : peut être utile si on connaît la taille finale la plus probable (évite les opérations d'augmentation de la taille)
- **`ArrayList(Collection<? extends E> c)`** : pour l'interopérabilité entre les différents types de collections

Méthodes principales

- Aucune des méthodes n'est « **`synchronized`** »

```
boolean add(E elt)
void add(int indice, E elt)
boolean contains(Object obj)
E get(int indice)
int indexOf(Object obj)
Iterator<E> iterator()
E remove(int indice)
E set(int indice, E elt)
int size()
```

Exemple d'utilisation de **ArrayList**

```
List<Employe> le =
  new ArrayList<Employe>();
Employe e = new Employe("Dupond");
le.add(e);
// Ajoute d'autres employés
. . .
// Affiche les noms des employés
for (int i = 0; i < le.size(); i++) {
  System.out.println(le.get(i).getNom());
}
```

Interface `RandomAccess`

- **`ArrayList`** implémente l'interface **`RandomAccess`**
- Cette interface est un marqueur (elle n'a pas de méthode) pour indiquer qu'une classe « liste » permet un accès direct *rapide* à un des éléments de la liste
- **`LinkedList`** n'implémente pas **`RandomAccess`**

Interfaces

`Iterator<E>` et `Iterable<E>`

Fonctionnalités

- Un itérateur (instance d'une classe qui implante l'interface `Iterator<E>`) permet d'énumérer les éléments contenus dans une collection
- Il encapsule la structure de la collection : on pourrait changer de type de collection (remplacer un `ArrayList` par un `TreeSet` par exemple) sans avoir à réécrire le code qui utilise l'itérateur
- Toutes les collections ont une méthode `iterator()` qui renvoie un itérateur

Méthodes de l'interface `Iterator<E>`

```
boolean hasNext()
E next()
* void remove()
```

- **`remove()`** enlève le dernier élément récupéré de la collection que parcourt l'itérateur (elle est optionnelle)

Exceptions lancées par les méthodes des itérateurs

- Ce sont des exceptions non contrôlées
- **`next`** lance **`NoSuchElementException`** lorsqu'il n'y a plus rien à renvoyer
- **`remove`** lance **`NoSuchOperationException`** si la méthode n'est pas implémentée (voir méthodes optionnelles) et lance **`IllegalStateException`** si **`next`** n'a pas été appelée avant ou si **`remove`** a déjà été appelée après le dernier **`next`**

Obtenir un itérateur

- L'interface `Collection<E>` contient la méthode
  `Iterator<E> iterator()`
  qui renvoie un itérateur pour parcourir les éléments de la collection
- L'interface `List<E>` contient en plus la méthode
  `ListIterator<E> listIterator()`
  qui renvoie un `ListIterator` (offre plus de possibilités que `Iterator` pour parcourir une liste et la modifier)

Exemple d'utilisation de `Iterator`

```
List<Employe> le = new ArrayList<Employe>();
Employe e = new Employe("Dupond");
le.add(e);
// Ajoute d'autres employés dans le
. . .
Iterator<Employe> it = le.iterator();
// le 1er next() fournira le 1er élément
while (it.hasNext()) {
  System.out.println(it.next().getNom());
}
```

Itérateur et modification de la collection parcourue

- Un appel d'une des méthodes d'un itérateur associé à une collection du JDK lance une **`ConcurrentModificationException`** si la collection a été modifiée directement depuis la création de l'itérateur (directement = sans passer par l'itérateur)

Itérateur et suppression dans la collection parcourue

- L'interface **`Iterator`** fournit la méthode *optionnelle* **`remove()`** qui permet de supprimer le dernier élément retourné par l'itérateur

Itérateur de liste et ajout dans la *liste* parcourue

- Si l'on veut faire des ajouts dans une liste (pas possible avec une collection qui n'implante pas l'interface **`List`**) pendant qu'elle est parcourue par un itérateur, il faut utiliser la sous-interface **`ListIterator`** de **`Iterator`** (renvoyée par la méthode **`listIterator()`** de **`List`**)
- Cette interface permet
  - de parcourir la liste sous-jacente dans les 2 sens
  - de modifier cette liste (méthodes *optionnelles* **`add`** et **`set`**)

Interface `Iterable<T>`

- Nouvelle interface (depuis JDK 5.0) du paquetage `java.lang` qui indique qu'un objet peut être parcouru par un itérateur
- Toute classe qui implémente `Iterable` peut être parcourue par une boucle « for each »
- L'interface `Collection` en hérite

Définition de `Iterable<T>`

```
public interface Iterable<T>  {
  Iterator<T> iterator();
}
```

Boucle « for each »

- Avant le JDK 5 :

```
for (Iterator it = coll.iterator();
      it.hasNext(); ) {
  Employe e = (Employe)it.next();
  String nom = e.getNom();
}
```

- Avec une boucle « for each » :

```
for (Employe e : coll) {
  String nom = e.getNom();
}
```

Syntaxe générale de « for each »

- ```
  for (Typev v : expression)
    instruction
  ```
- ***Typev v*** : déclaration d'une variable
- ***expression*** : une expression dont l'évaluation donne un tableau ***typeT*`[]`** ou un objet qui implémente l'interface **`Iterable<E>`**, tel que ***typeT*** ou **`E`** est affectable à ***Typev***
  ```

Restriction de « for each »

- On ne dispose pas de la position dans le tableau ou la collection pendant le parcours
- On ne peut pas modifier la collection (en passant par l'itérateur)
- L'utilisation d'une boucle ordinaire et d'un itérateur ou d'un compteur de boucle explicite est indispensable si ces 2 restrictions gênent

Implémentation **`Iterable`**

- Il n'est pas fréquent d'avoir à implémenter **`Iterable`** sur une des classes qu'on a créées
- Voici un exemple d'utilisation : une classe représente un document de type texte et on implémente **`Iterable<String>`** pour permettre du code du type suivant

```
for (String mot : texte) {
   // Traitement du mot
   . . .
}
```

Interface **Map<K,V>**

Définition

- L'interface **`Map<K,V>`** correspond à un groupe de couples clés-valeurs
- Une clé repère une et une seule valeur
- Dans la map il ne peut exister 2 clés égales au sens de **`equals()`**

Implémentation

- **`HashMap<K,V>`**, table de hachage ; garantit un accès en temps constant
- **`TreeMap<K,V>`**, arbre ordonné suivant les valeurs des clés, avec accès en log(n) ;
  La comparaison utilise l'ordre naturel (interface **`Comparable<K>`**) ou une instance de **`Comparator<? super K>`**

Fonctionnalités

- On peut (entre autres)
  - ajouter et enlever des couples clé – valeur
  - récupérer une référence à une des valeurs en donnant sa clé
  - savoir si une table contient une valeur
  - savoir si une table contient une clé

Méthodes de Map<K,V>

```
* void clear()
boolean containsKey(Object clé)
boolean containsValue(Object valeur)
V get(Object clé)
boolean isEmpty()
Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K,V>> entrySet()
* V put(K clé, V valeur)
* void putAll(Map<? extends K, ? extends V>
  map)
* V remove(Object key)
int size()
```

retourne **null** si la clé n'existe pas

retourne l'ancienne valeur associée à la clé si la clé existait déjà

Interface *interne* **`Entry<K,V>`** de **`Map`**

- L'interface **`Map<K,V>`** contient l'interface interne **`public Map.Entry<K,V>`** qui correspond à un couple clé-valeur
- Cette interface contient 3 méthodes
  - **`K getKey()`**
  - **`V getValue()`**
  - **`V setValue(V valeur)`**
- La méthode **`entrySet()`** de **`Map`** renvoie un objet de type « ensemble (**`Set`**) de **`Entry`** »

Constructeurs

- Il n’est pas possible de donner des constructeurs dans une interface ; mais la convention donnée par les concepteurs est que toute classe d’implantation de **`Map`** doit fournir au moins 2 constructeurs :
  - un constructeur sans paramètre
  - un constructeur qui prend une *map* de type compatible en paramètre (facilite l’interopérabilité)

Modification des clés

- La bonne utilisation d'une *map* n'est pas garantie si on modifie les valeurs des clés avec des valeurs qui ne sont pas égales (au sens de **`equals`**) aux anciennes valeurs
- Si on veut changer une clé, on enlève d'abord l'ancienne entrée (avec l'ancienne clé) et on ajoute ensuite la nouvelle entrée avec la nouvelle clé et l'ancienne valeur

Récupérer les valeurs d'une **`Map`**

1. On récupère les valeurs sous forme de **`Collection<V>`** avec la méthode **`values()`**
   La collection obtenue reflétera les modifications futures de la *map*
2. On utilise la méthode **`iterator()`** de l'interface **`Collection<V>`** pour récupérer un à un les éléments

Récupérer les clés d'une **`Map`**

1. On récupère les clés sous forme de **`Set<K>`** avec la méthode **`keySet()`**
2. On utilise alors la méthode **`iterator()`** de l'interface **`Set<K>`** pour récupérer une à une les clés

Récupérer les entrées d'une `Map`

1. On récupère les entrées (paires clé-valeur) sous forme de `Set<Entry<K,V>>` avec la méthode `entrySet()`
2. On utilise alors la méthode `iterator()` de l'interface `Set<Entry<K,V>>` pour récupérer une à une les entrées

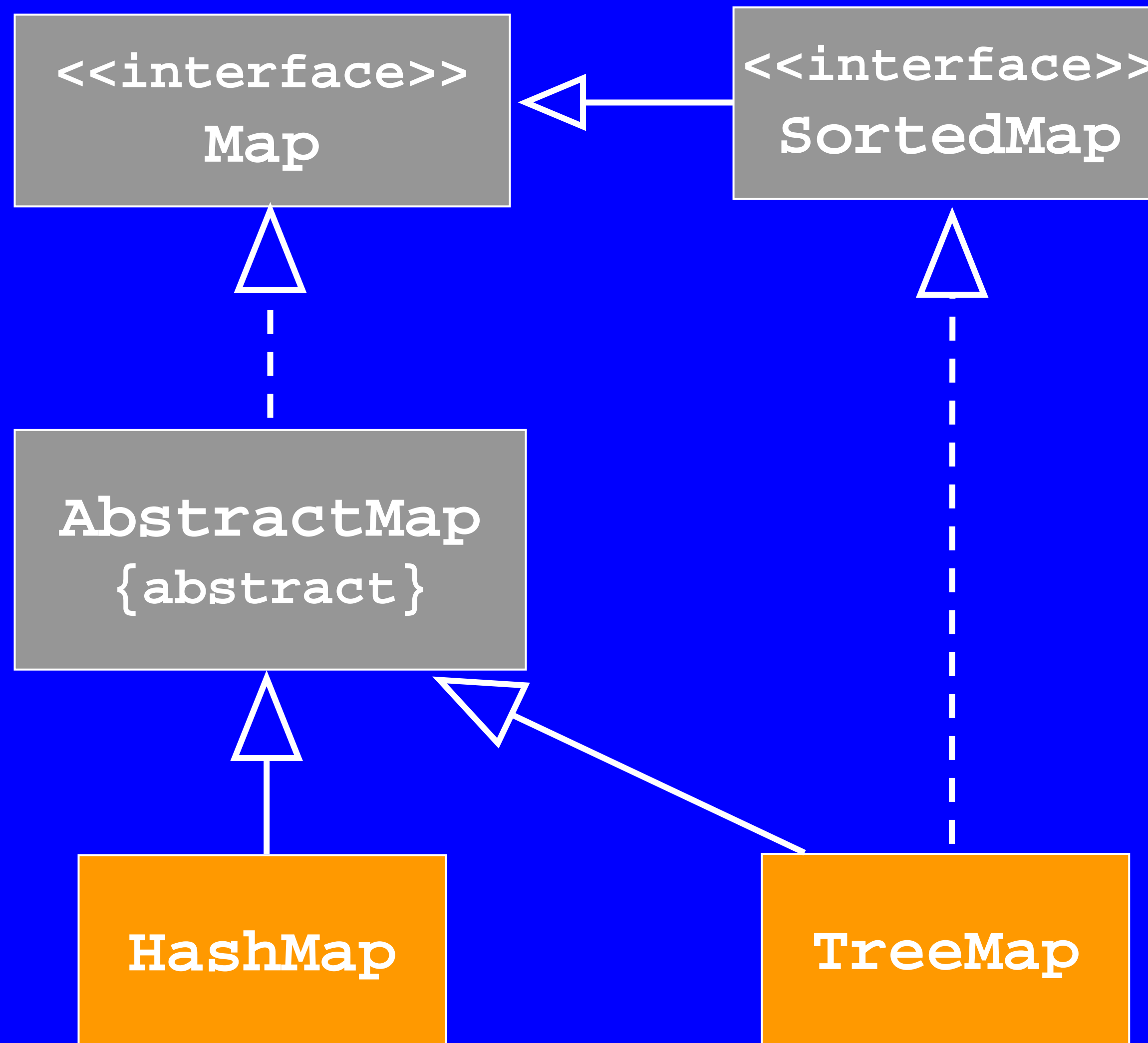
Itérateur et modification de la *map* parcourue

- Un appel d'une des méthodes d'un itérateur associé à une map du JDK lance une exception **`ConcurrentModificationException`** si la map a été modifiée directement depuis la création de l'itérateur
- L'itérateur peut avoir été obtenu par l'intermédiaire des **`Set`** renvoyés par **`keySet()`** ou **`entrySet()`** ou de la **`Collection`** renvoyée par **`values()`**

Itérateur et suppression dans la *map* parcourue

- Pendant qu'une *map* est parcourue par un des itérateurs associés à la *map*
  - On peut supprimer des éléments avec la méthode **`remove()`** de **`Iterator`** (si elle est implantée)
  - On ne peut ajouter des éléments dans la *map*

Interfaces et classes d'implantation



Classe **`HashMap<K,V>`**

Constructeurs

- **`HashMap()`**
- **`HashMap(int tailleInitiale)`** : idem ArrayList
- **`HashMap(int tailleInitiale, float loadFactor)`** : **`loadFactor`** indique à partir de quel taux de remplissage la taille de la map doit être augmentée (0,75 par défaut convient le plus souvent)
- **`HashMap(Map<? extends K, ? extends V)`** : pour l'interopérabilité entre les maps

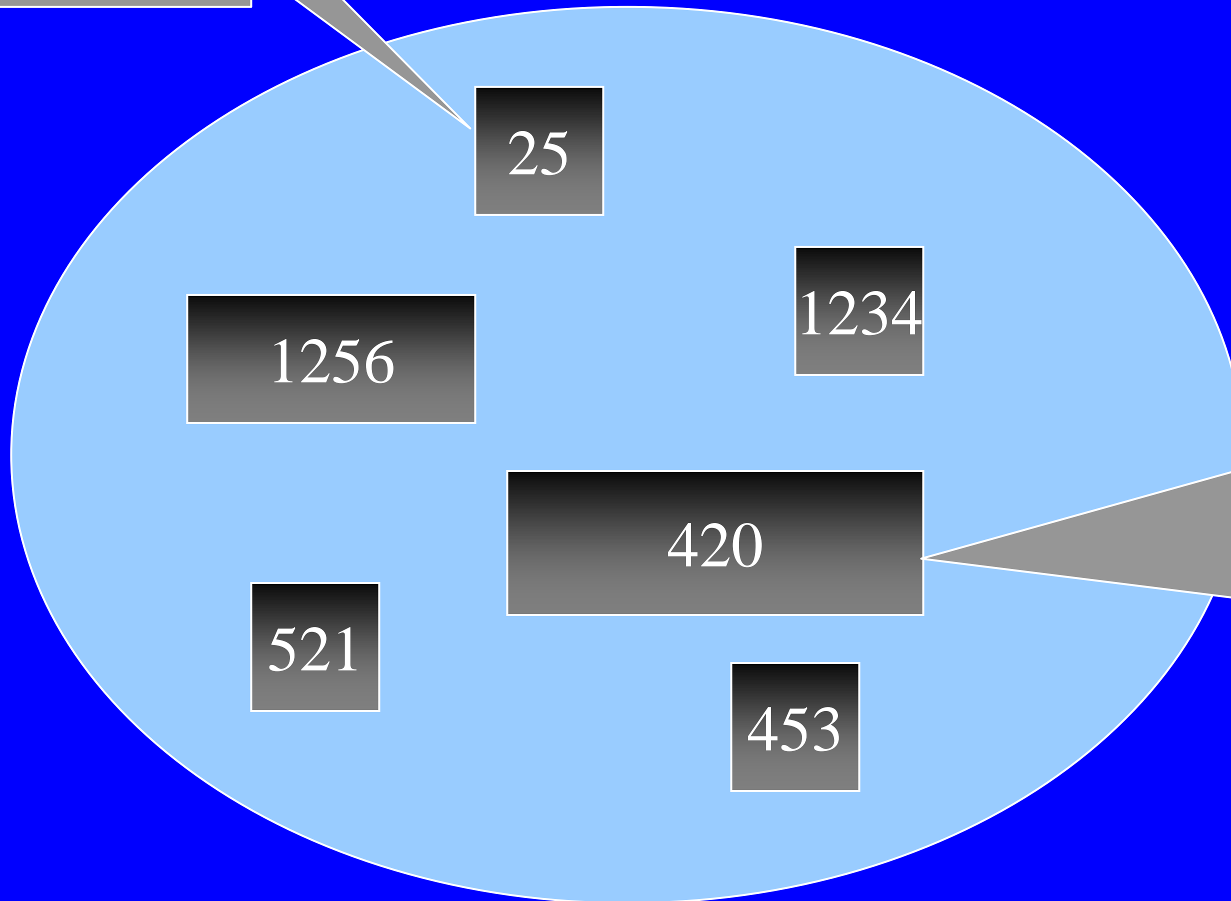
Implémentation

- La classe **`HashMap<K,V>`** utilise la structure informatique nommée « table de hachage » pour ranger les clés
- La méthode **`hashCode()`** (héritée de **`Object`** ou redéfinie) est utilisée pour répartir les clés dans la table de hachage

Table de hachage

- Une table de hachage range des éléments en les regroupant dans des sous-ensembles pour les retrouver plus rapidement
- Le regroupement dans les sous-ensembles dépend de la valeur d'un calcul effectué sur les éléments
- Pour retrouver un élément, on doit fournir cette valeur qui va déterminer dans quel sous-ensemble il est rangé ; l'élément est alors retrouvé très rapidement grâce à des techniques informatiques
- Pour que la recherche soit efficace, le calcul doit répartir les éléments uniformément sur les sous-ensembles

Table de hachage



Certains sous-ensembles contiennent plusieurs éléments qui donnent la même valeur calculée

Exemple d'utilisation de **HashMap**

```
Map<String,Employe> hm =
  new HashMap<String,Employe> ();
Employe e = new Employe("Dupond");
e.setMatricule("E125");
hm.put(e.getMatricule(), e);
// Crée et ajoute les autres employés dans la table de hachage
. .
Employe e2 = hm.get("E369");
Collection<Employe> elements = hm.values();
for (Employe employe : employes) {
  System.out.println(employe.getNom());
}
```

Types « *raw* »

Avant le JDK 5

- Les collections n'étaient pas génériques
- On trouvait les types **`Collection`**, **`List`**, **`ArrayList`**, **`Map`**, **`Iterator`**, etc., appelés type « *raw* » depuis le JDK 5
- Elles pouvaient contenir des objets de n'importe quelle classe
- Les signatures de l'API n'utilisaient que la classe **`Object`** et il fallait souvent *caster* les valeurs retournées si on voulait leur envoyer des messages

Exemple

```
List le = new ArrayList();
Employe e = new Employe("Dupond");
le.add(e);
. . .
for (int i = 0; i < le.size(); i++) {
  System.out.println(
      ((Employe) le.get(i)).getNom());
}
```

Remarquez les parenthèses pour le *cast*

Compatibilité (1)

- Java a toujours permis de faire marcher les anciens codes avec les nouvelles API
- On peut donc utiliser du code écrit avec les types « *raw* » avec le JDK 5

Compatibilité (2)

- Les collections non génériques n'offrent pas le même niveau de sécurité que les nouvelles collections et on reçoit alors des messages d'avertissement du compilateur
- Pour faire disparaître ces messages il suffit d'annoter la méthode avec **`@SuppressWarning("unchecked")`**

Utilisation des types *raw*

- Il est permis d'utiliser une collection *raw* là où on attend une collection générique, et réciproquement
- Évidemment, il faut l'éviter et ne le faire que pour utiliser un code ancien avec du code écrit avec les collections génériques
- Voir cours sur la généricité

Classes introduites par le JDK 1.4

- Les classes suivantes ont été introduites depuis le JDK 1.4 et ont été rendues génériques par le JDK 5

`LinkedHashMap<K,V>`

- **`Map`** qui maintient les clés dans l'ordre d'insertion ou de dernière utilisation (ordre choisi en paramètre du constructeur)
- Les performances sont justes un peu moins bonnes que **`HashMap`** (gestion d'une liste doublement chaînée)
- L'ordre de « la clé la moins récemment utilisée » est pratique pour la gestion de cache

LinkedHashMap<K,V>

- Il est possible de redéfinir une méthode
  `protected boolean removeEldestEntry(Map.Entry)`
  dans une classe fille pour indiquer si l'entrée la moins récemment utilisée doit être supprimée de la map (renvoie **`true`** si elle doit être supprimée)
- Cette possibilité est utile pour les caches dont la taille est fixe

`LinkedHashSet<E>`

- **`Set`** qui maintient les clés dans l'ordre d'insertion
- Les performances sont légèrement moins bonnes que pour un **`HashSet`**

`IdentityHashMap<K,V>`

- Cette **`Map`** ne respecte pas le contrat général de **`Map`** : 2 clés sont considérées comme égales seulement si elles sont identiques (et pas si elles sont égales au sens de equals)
- Elle doit être réservée à des cas particuliers où on veut garder des traces de tous les objets manipulés (sérialisation, objets « proxy », …)

Classes introduites par le JDK 5

Queue<E>

- L'interface **java.util.Queue<E>** représente une file d'attente (FIFO) qui contient des éléments de type **E**
- Le « premier » élément de la queue s'appelle la tête de la queue
- Elle hérite de **Collection<E>**
- Elle peut aussi être utilisée autrement que comme une file d'attente ; cela dépend de l'implémentation de l'ajout des éléments

`Queue<E>`

- Elle contient 2 groupes de méthodes qui exécutent les mêmes opérations de base (ajouter, supprimer, consulter)
- Un groupe, hérité de **`Collection`**, lance une exception en cas de problème (**`add`**, **`remove`**, **`element`**)
- L'autre groupe renvoie une valeur particulière dans les mêmes conditions (**`offer`**, **`poll`**, **`peek`**)
- Le développeur choisit ce qu'il préfère

Nouvelles méthodes `Queue<E>`

- **`boolean offer(E o)`** ajoute un élément dans la queue ; renvoie **`false`** si l'ajout n'a pas été possible
- **`E poll()`** enlève la tête de la queue (et la récupère) ; renvoie **`null`** si queue vide
- **`E peek()`** récupère une référence vers la tête de la queue, sans la retirer ; renvoie **`null`** si queue vide

Implémentations de `Queue<E>`

- Des classes implémentent `Queue` :
  - `LinkedList<E>`, déjà vu
  - `PriorityQueue<E>` : les éléments sont ordonnés automatiquement suivant un ordre naturel ou un `Comparator` ; l'élément récupéré en tête de queue est le plus petit
  - d'autres classes et interfaces sont liées aux problèmes de concurrence (multi-threads)
- Pour plus de détails, consultez l'API

Interface **`Deque<E>`**

- Sous interface de **`Queue<E>`**
- Représente une collection linéaire dont les éléments peuvent être ajoutés « aux 2 bouts » (la tête et la queue de la collection)
- Implémentée par la classe **`ArrayDeque<E>`**
- Pour plus de détails, consultez l'API

EnumMap

- La classe du paquetage **`java.util`** **`EnumMap<K extends Enum <K>, V>`** hérite de **`AbstractMap<K,V>`**
- Elle correspond à des *maps* dont les clés sont d'un type énumération
- Les clés gardent l'ordre naturel de l'énumération (l'ordre dans lequel les valeurs de l'énumération ont été données)
- L'implémentation est très compacte et performante

Principes généraux sur les déclarations de types pour favoriser la réutilisation

Principe général pour la réutilisation

- Le code doit fournir ses services au plus grand nombre possible de clients
- Les conditions d'utilisation des méthodes doivent être les moins contraignantes possible
- Ce principe et ce qui suit dans cette section est valable pour toute API (pas seulement pour l’API des collections)

Principe général pour la réutilisation

- Donc, si possible,
  - pour les types des paramètres, des interfaces les plus générales et qui correspondent au strict nécessaire pour que la méthode fonctionne
  - pour les types retour, des classes ou interfaces les plus spécifiques possible (dont les instances offrent le plus de services) ; mais il faut penser aussi à l'encapsulation

- Pour favoriser l'adaptation/extensibilité du code, il faut aussi déclarer les variables du type le plus général possible, compte tenu de ce qu'on veut en faire (des messages échangés entre les objets)
- Cette section étudie l’application de ces principes avec les collections, mais ils sont valables pour n'importe quel code

Paramètres des méthodes

- Quand on écrit une méthode, il vaut mieux déclarer les paramètres du type interface le plus général possible et éviter les types des classes d'implémentation :
  - `m(Collection)` plutôt que `m(List)` (quand c'est possible)
  - éviter `m(ArrayList)`
- On élargit ainsi le champ d'utilisation de la méthode

Type retour des méthodes (1)

- On peut déclarer le type retour du type le plus spécifique possible **si ce type ajoute des fonctionnalités** (mais voir transparent suivant) :
  « `List m()` » plutôt que « `Collection m()` »
- L'utilisateur de la méthode
  - pourra ainsi profiter de toutes les fonctionnalités offertes par le type de l'objet retourné
  - mais rien ne l'empêchera de faire un « *upcast* » avec l'objet retourné : `Collection l = m(...)`

Type retour des méthodes (2)

- On doit être certain que l'instance retournée par la méthode sera toujours bien du type déclaré durant l'existence de la classe
- En effet, si on déclare que le type retour est un `ArrayList` mais qu'on est amené plus tard à utiliser un `LinkedList` dans la méthode, ça posera des problèmes de maintenance

Variables

- Déclaration d'une collection (ou *map*) : on crée le type de collection qui convient le mieux à la situation, mais on déclare cette collection du type d'une interface :

```
List<Employe> employes =
  new ArrayList<Employe>();
```

- On se laisse ainsi la possibilité future de changer d'implémentation :

```
employes =
  new LinkedList<Employe>();
```

Conclusion

- Le plus souvent, les types déclarés des variables, des paramètres ou des valeurs retour doivent être des **interfaces**, les moins spécifiques possible, sauf pour les valeurs de retour

Compatibilité avec les classes fournies par le JDK 1.1

Classes du JDK 1.1

- Dans les API du JDK 1.1, on utilisait
  - la classe **`Vector`** pour avoir un tableau de taille variable (remplacée par **`ArrayList`**)
  - la classe **`Hashtable`** pour avoir un ensemble d'objets indexé par des clés (remplacée par **`HashMap`**)
  - l'interface **`Enumeration`** pour avoir une énumération des objets contenus dans une collection (remplacée par **`Iterator`**)

Classes du JDK 1.1

- Elles offrent moins de possibilités
- Les noms des méthodes sont semblables aux noms des méthodes de **`Collection`**, **`Map`** et **`Iterator`**, en plus long
- **`Vector`** et **`HashTable`** sont *synchronized*
- Il est conseillé de travailler avec les nouvelles classes
- Depuis le JDK 5 ces classes sont génériques ; les exemples qui suivent utilisent les versions antérieures au JDK 5

Exemple d'utilisation de **Vector**

```
Vector v = new Vector();
Employe e = new Employe("Dupond");
v.addElement(e);
. . .
for (int i = 0; i < v.size(); i++) {
  System.out.println(
      ((Employe)v.elementAt(i)).getNom());
}
```

Exemple d'utilisation de `Enumeration`

```
Vector v = new Vector();
Employe e = new Employe("Dupond");
v.addElement(e);
. . . // ajoute d'autres employés dans v
Enumeration enum = v.elements();
for ( ; enum.hasMoreElements(); ) {
  System.out.println(
    ((Employe)enum.nextElement()).getNom());
}
```

Exemple d'utilisation de **HashTable**

```
Hashtable ht = new Hashtable();
Employe e = new Employe("Dupond");
e.setMatricule("E125");
ht.put(e.matricule, e);
... // crée et ajoute les autres employés dans la table de hachage
Employe e2 = (Employe)ht.get("E369");
Enumeration enum = ht.elements();
for ( ; enum.hasMoreElements() ; ) {
  System.out.println(
      ((Employe)enum.nextElement()).getNom());
}
```

Tri et recherche dans une collection

Classe `Collections`

- Cette classe ne contient que des méthodes **`static`**, utilitaires pour travailler avec des collections :
  - tris (sur listes)
  - recherches (sur listes triées)
  - copies
  - synchronisation
  - minimum et maximum
  - ...

Trier une liste

- Si `l` est une liste, on peut trier `l` par :

```
Collections.sort(l);
```

- Cette méthode ne renvoie rien ; elle trie `l`
- Pour que cela fonctionne, il faut que les éléments de la liste implémentent l'interface `java.lang.Comparable<T>` pour un type `T` ancêtre du type `E` de la collection

Interface `Comparable<T>`

- Cette interface correspond à l'implantation d'un ordre naturel dans les instances d'une classe
- Elle ne contient qu'une seule méthode :
`int compareTo(T t)`
- Cette méthode renvoie
  - un entier positif si l'objet qui reçoit le message est plus grand que `t`
  - 0 si les 2 objets ont la même valeur
  - un entier négatif si l'objet qui reçoit le message est plus petit que `t`

Interface `Comparable` (2)

- Toutes les classes du JDK qui enveloppent les types primitifs (`Integer` par exemple) implantent l'interface `Comparable`
- Il en est de même pour les classes du JDK `String`, `Date`, `Calendar`, `BigInteger`, `BigDecimal`, `File`, `Enum` et quelques autres (mais pas `StringBuffer` ou `StringBuilder`)
- Par exemple, `String` implémente `Comparable<String>`

Interface `Comparable` (3)

- Il est conseillé d'avoir une méthode `compareTo` compatible avec `equals` : `e1.compareTo(e2) == 0` *ssi* `e1.equals(e2)`
- Sinon, les contrats des méthodes de modification des `SortedSet` et `SortedMap` risquent de ne pas être remplis si elles utilisent l'ordre naturel induit par cette méthode
- `compareTo` lance une `NullPointerException` si l'objet passé en paramètre est `null`

Exercice sur la généricité

- La méthode **`sort`** est une méthode paramétrée
- Sa signature peut être un bon exercice pour tester ses connaissances de la généricité :

```
<T extends Comparable<? super T>>
void sort(List<T> liste)
```

- L'idée : si on peut comparer les éléments d'une classe, on peut comparer ceux des classes filles

Question

- Que faire
  - si les éléments de la liste n'implantent pas l'interface `Comparable`,
  - ou si on ne veut pas les trier suivant l'ordre donné par `Comparable` ?

Interface `Comparator<T>`

- Réponse :
  1. on construit un objet qui sait comparer 2 éléments de la collection (interface `java.util.Comparator<T>`)
  2. on passe cet objet en paramètre à la méthode `sort()`

Interface `Comparator<T>` (2)

- Elle comporte une seule méthode :

```
int compare(T t1, T t2)
```

qui doit renvoyer

  – un entier positif si **`t1`** est « plus grand » que **`t2`**

  – 0 si **`t1`** a la même valeur (au sens de **`equals`**) que **`t2`**

  – un entier négatif si **`t1`** est « plus petit » que **`t2`**

Interface `Comparator<T>` (3)

- **Attention**, il est conseillé d'avoir une méthode `compare` compatible avec `equals` :
`compare(t1, t2) == 0` *ssi*
`t1.equals(t2)`
- Sinon, les contrats des méthodes de modification des `SortedSet` et `SortedMap` risquent de ne pas être remplis si elles utilisent ce comparateur

Exemple de comparateur

```
public class CompareSalaire
    implements Comparator<Employe> {
  public int compare(Employe e1, Employe e2) {
    double s1 = e1.getSalaire();
    double s2 = e2.getSalaire();
    if (s1 > s2)
      return +1;
    else if (s1 < s2)
      return -1;
    else
      return 0;
  }
}
```

Utilisation d'un comparateur

```
List<Employe> employes =
  new ArrayList<Employe>();
// On ajoute les employés
. . .
Collections.sort(employes,
                 new CompareSalaire());
System.out.println(employes);
```

Signature de la méthode pour trier avec un comparateur

- ```
  public static <T>
  void sort(List<T> list,
            Comparator<? super T> c)
  ```
- Autre exercice pour tester ses connaissances sur la généricité
  ```

Recherche dichotomique

- Une grande partie des méthodes de la classe Collection sont des méthodes génériques
- Si la liste l est triée par l'ordre naturel de ses éléments (**`Comparable`**), la méthode suivante retourne l'indice de **`elt`** dans la liste
- ```
  <T> int binarySearch(
    List<? extends Comparable<? super T>> l,
    T elt)
  ```
- Si **`elt`** n'est pas dans la liste, retourne -p -1 où p est le point d'insertion éventuel de **`elt`** dans la liste
  ```

Recherche dichotomique (2)

- Si la liste l est triée par l'ordre donné par un comparateur (`Comparator`), il faut utiliser la méthode suivante en passant le comparateur en 3ème paramètre

```
<T> int binarySearch(
  List<? extends T> l,
  T elt,
  Comparator <? super T> c)
```

Classe `Arrays`

- Elle contient des méthodes `static` utilitaires pour travailler avec des tableaux d'objets ou de types primitifs (les méthodes sont surchargées pour tous les types primitifs)

Méthodes de la classe **`Arrays`**

- Trier : **`sort`**
- Chercher dans un tableau trié : **`binarySearch`**
- Comparer 2 tableaux : **`equals`**
- Représenter un tableau sous forme de **`String`** : **`toString`**
- Remplir un tableau avec des valeurs :
- Copier un tableau (depuis 1.6) : **`copyOf`**
- Rappel : **`System.arraycopy`** permet de copier les éléments d'un tableau dans un tableau existant

Pas redéfinition des méthodes de **`Object`** !

Synchronisation

Synchronisation

- Si une collection peut être utilisée en même temps par plusieurs *threads*, la modification simultanée de ses éléments peut provoquer des problèmes
- Des méthodes de la classe **`Collections`** créent une collection synchronisée à partir d'une collection
- Quand une collection est synchronisée, les *threads* ne peuvent modifier la collection en même temps ; ils doivent le faire l'un après l'autre

Exemple de synchronisation

- Exemple de création d'une liste qui peut être utilisée sans danger par plusieurs *threads* :

```
List<String> listeS =
  Collections.synchronizedList(liste);
```

- Remarques :
  – il ne faut plus passer par liste pour effectuer des modifications
  – il faut synchroniser tout parcours de la liste
  – toute la collection est verrouillée à chaque accès. Ça n'est donc pas très performant

Collections synchronisées du JDK 5

- Le paquetage **`java.util.concurrent`** du JDK 5 fournit des collections synchronisées
- Ce sont les classes qui implémentent les interfaces **`BlockingQueue`** ou **`ConcurrentMap`**
- Elles sont plus performantes que les collections synchronisées par **`Collections`**

Écrire ses propres classes de collections

Utiliser les classes abstraites

- Il est rare d'avoir à écrire ses propres classes de collections
- Le plus souvent on ne part pas de zéro
- Il suffit d'implémenter les méthodes abstraites des classes abstraites fournies avec l'API : **`AbstractList`**, **`AbstractSet`**, **`AbstractMap`**, **`AbstractQueue`**, **`AbstractSequentialList`**, **`AbstractCollection`**

Redéfinir quelques méthodes

- Si la collection est modifiable il faut aussi redéfinir quelques méthodes : **`set`**, **`add`**,… qui sinon renvoient une **`UnsupportedOperationException`**
- Pour des raisons de performances on peut aussi avoir à redéfinir d'autres méthodes (voir l'API pour une description de l'implémentation des méthodes des classes abstraites)

L'agrégation en notation UML