Collections

Pratique de la programmation orientée-objet

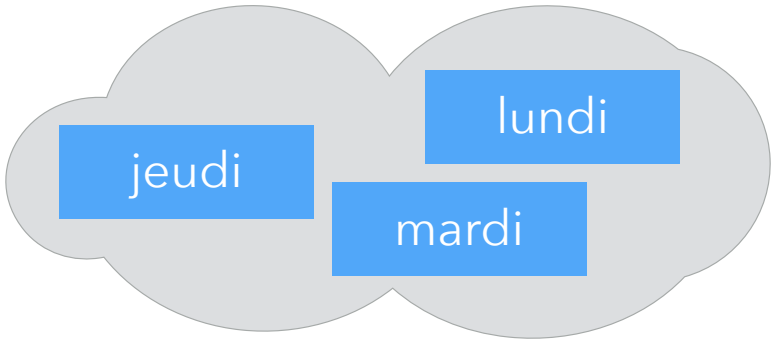
Michel Schinz

Collections étudiées

Listes
(*lists*)

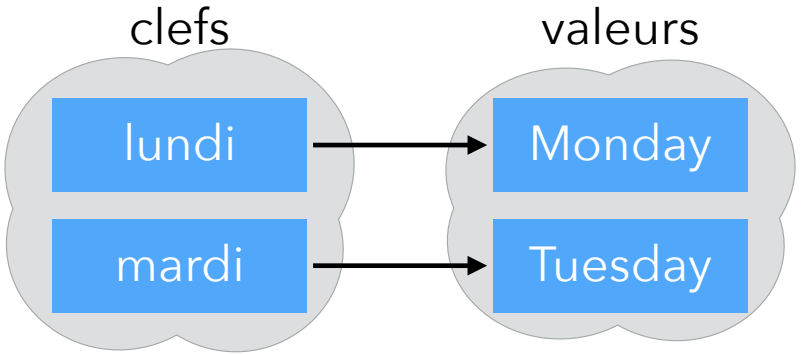
ordre : oui
doublons : oui

Ensembles
(*sets*)



ordre : non
doublons : non

Tables associatives
(*maps*)



association
clefs/valeurs

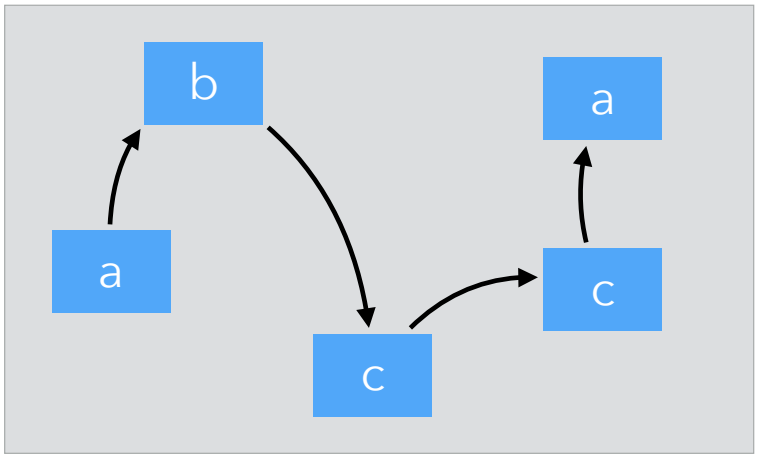
Mises en œuvre

La représentation en mémoire de la liste [a,b,c,c,a] dépend de la mise en œuvre choisie :

« Tableau-liste »
(`ArrayList`)

Accès aléatoire
aux éléments

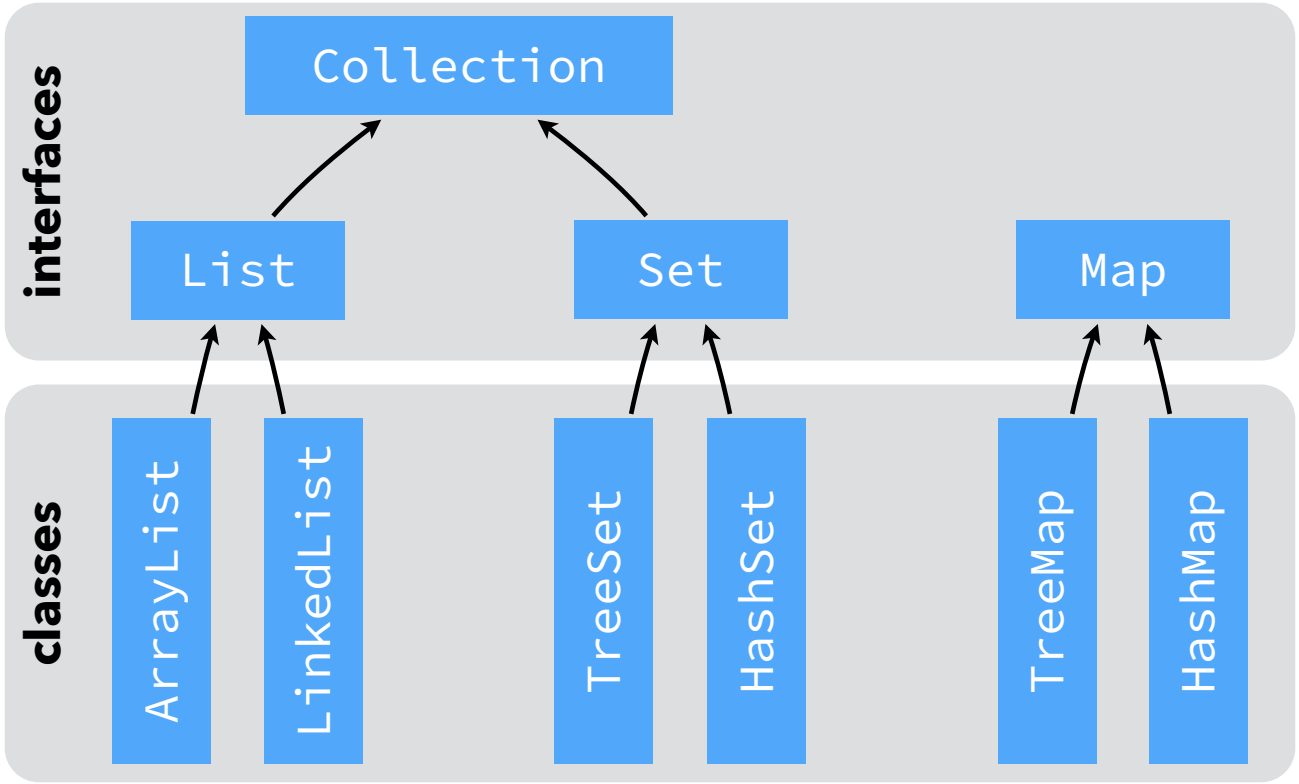
Liste chaînée
(`LinkedList`)



Accès séquentiel
aux éléments

Collections en Java

(vue *très* partielle et simplifiée du paquetage `java.util`)



\+ classes utilitaires `Collections` et `Arrays`

L'interface Collection

```
public interface Collection<E> {
  // méthodes de consultation :
  boolean isEmpty();
  int size();
  boolean contains(Object e);
  boolean containsAll(Collection<E> c);

  // … à suivre
```

L'interface Collection

```
  // … suite

  // méthodes d'ajout :
  boolean add(E e);
  boolean addAll(Collection<E> c);

  // méthodes de suppression :
  void clear();
  boolean remove(E e);
  boolean removeAll(Collection<E> c);
  boolean retainAll(Collection<E> c);
}
```

Modifiabilité

Toutes les méthodes qui modifient le contenu des collections (`add`, `remove`, `set`, etc.) sont **optionnelles** et peuvent lever `UnsupportedOperationEx…` !
En général, une collection est soit :

- **modifiable** : *aucune* de ses méthodes de modification ne lève `UnsupportedOperationException`, soit
- **non modifiable** : *toutes* ses méthodes de modification lèvent `UnsupportedOperationException`.

Listes

L'interface `List`

```
public interface List<E> extends Collection<E> {
  // méthodes de consultation :
  E get(int i);
  int indexOf(E e);
  int lastIndexOf(E e);
  // méthodes d'ajout :
  void add(int i, E e)
  boolean addAll(int i, Collection<E> c);
  // méthodes de suppression :
  E remove(int i);
  E set(int i, E e)
  // … + quelques autres méthodes
}
```

Mises en œuvre de `List`

Opération	`ArrayList`	`LinkedList`
ajout (`add`), suppression (`remove`)	O(*n*)	O(1)
accès (`get`), modification (`set`)	O(1)	O(*n*)

Attention : beaucoup de cas particuliers !

Vues

La méthode `subList` de `List` permet d'obtenir une vue sur une sous-liste :

```
List<E> subList(int f, int t)
```

La méthode `asList` de `Arrays` permet de voir un tableau comme une liste (de taille fixe) :

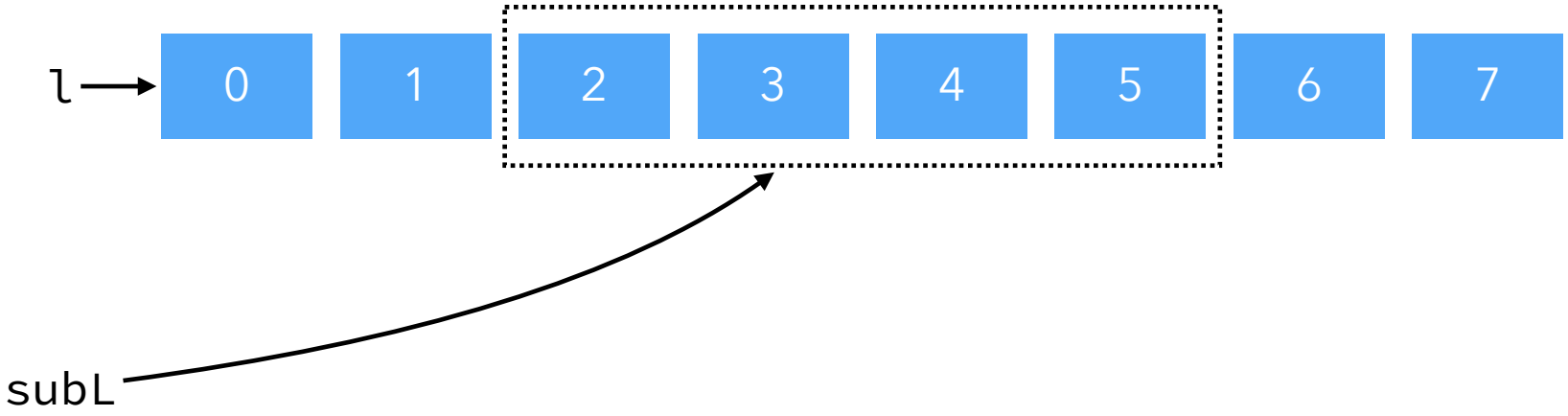
```
<E> List<E> asList(E... a)
```

La méthode `unmodifiableList` de `Collections` permet d'obtenir une vue non modifiable sur une liste :

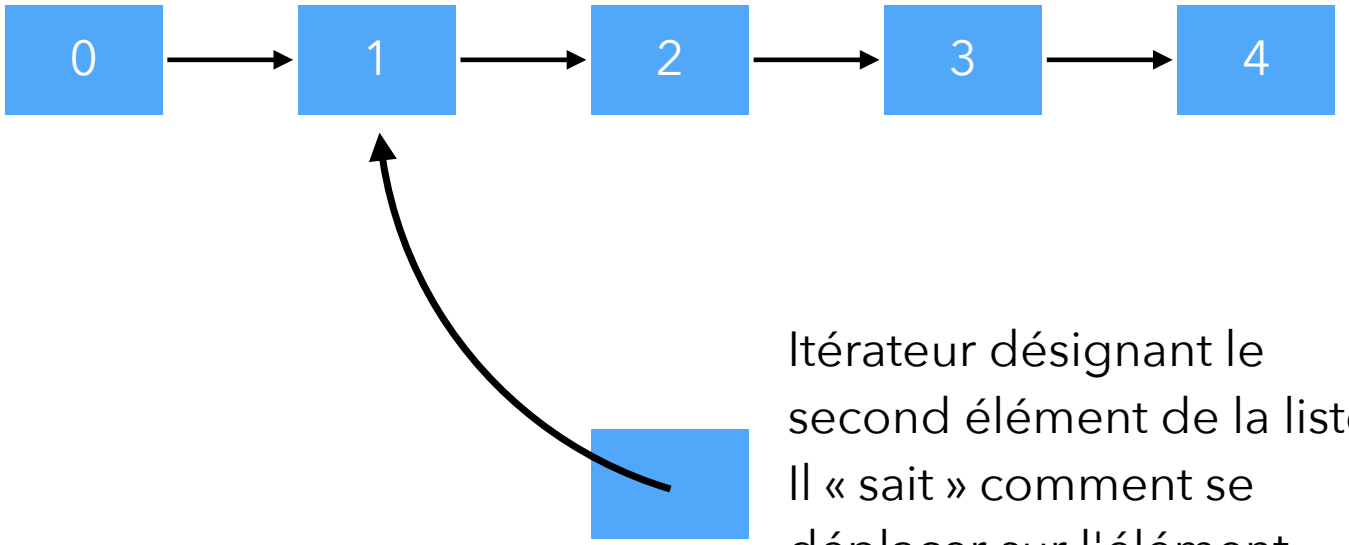
```
<E> List<E> unmodifiableList(List<E> l)
```

Attention : une vue n'est pas une copie !

Vue sur une sous-liste



Itérateur



Itérateur désignant le second élément de la liste. Il « sait » comment se déplacer sur l'élément suivant.

L'interface `Iterator`

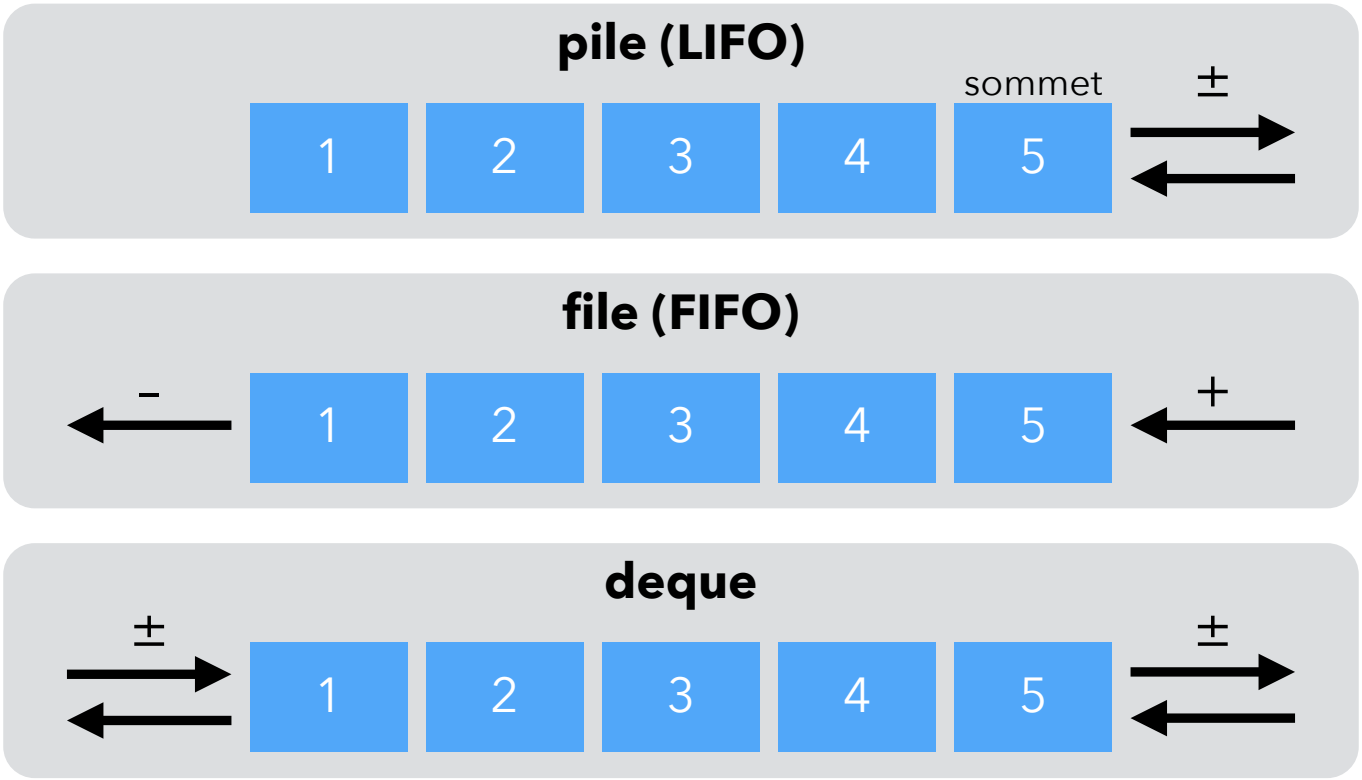
```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

La méthode `iterator` de `List` permet d'obtenir un itérateur pointant avant le premier élément.

L'interface Iterable

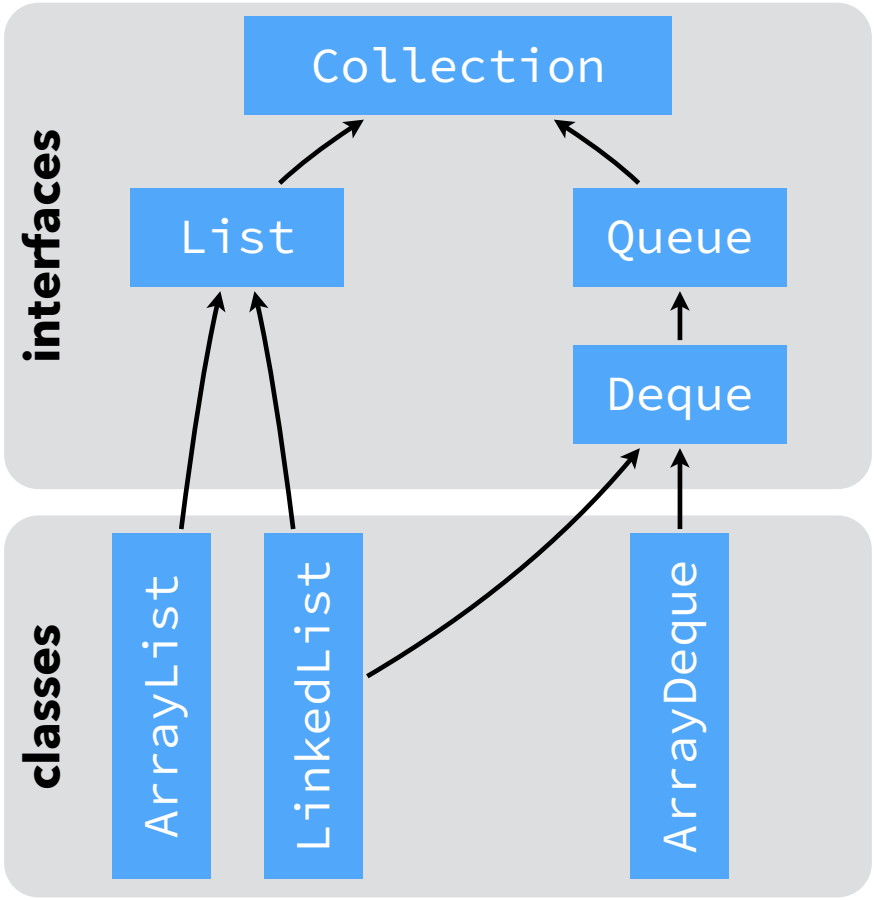
```
public interface Iterable<E> {
  Iterator<E> iterator();
}
```

Piles, files, deques



+ : ajout, – : suppression, ± : ajout/suppression

Piles, files, deques Java



Règle des listes

Pour représenter une pile, une file ou une « deque », utilisez `ArrayDeque`.
Pour représenter une liste dans toute sa généralité, utilisez `ArrayList` si les opérations d'indexation (`get`, `set`) dominent, sinon `LinkedList`.

Note : `ArrayList` peut également s'utiliser comme une pile, pour peu que les ajouts/suppressions se fassent toujours à la fin de la liste et pas au début.

Collection.removeIf

L'interface `Collection` offre une méthode `removeIf` :

```
public interface Collection<E> {
  boolean removeIf(Predicate<E> filter);
}
```

où `Predicate` est définie ainsi :

```
public interface Predicate<T> {
  boolean test(T t);
}
```

L'idée est que tous les éléments pour lesquels la méthode `test` de `filter` retourne vrai sont supprimés.

Ensembles

L'interface Set

```
public interface Set<E> extends Collection<E> {
  // n'ajoute aucune méthode !
}
```

Mises en œuvre de `Set`

Opération	`TreeSet`	`HashSet`
ajout (`add`), test (`contains`), etc.	O(log *n*)	O(1)

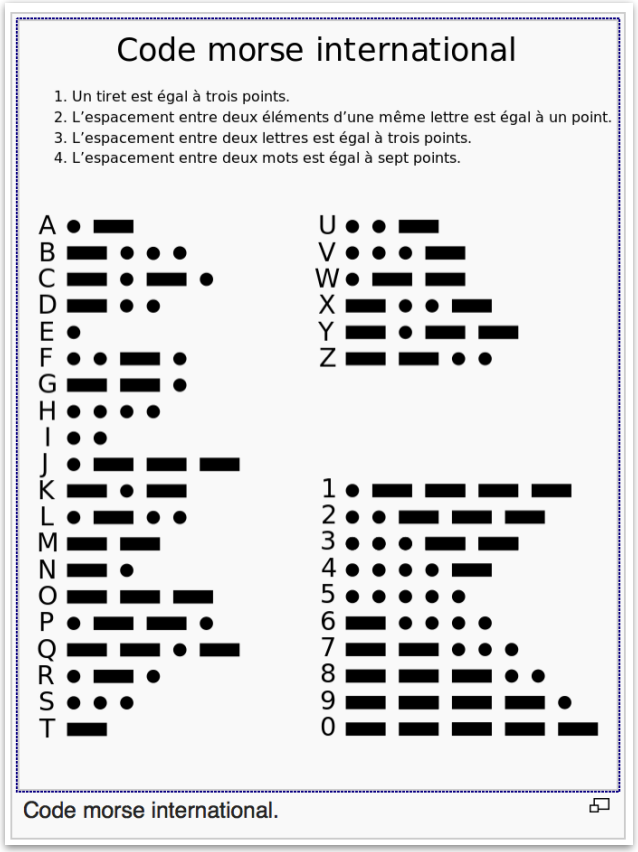
Différence importante :

- `TreeSet` stocke ses éléments de manière triée,
- `HashSet` stocke ses éléments dans un ordre qui semble quelconque.

Tables associatives

Table associative

Une table associative associe des valeurs à des clefs. Par exemple, la table ci-dessous associe leur représentation en morse aux lettres (et chiffres) de l'alphabet latin.



Code morse international.

L'interface Map (1/4)

```
public interface Map<K, V> {
  // méthodes de consultation
  boolean isEmpty();
  int size();

  boolean containsKey(Object k);
  V get(Object k);
  V getOrDefault(Object k, V d);

  // … à suivre
}
```

L'interface Map (2/4)

```
public interface Map<K, V> { // … suite
  // méthodes d'ajout/remplacement
  V put(K k, V v);
  V putIfAbsent(K k, V v);
  void putAll(Map<K, V> m);

  // … aussi : compute, computeIfAbsent, …

  // … à suivre
}
```

L'interface Map (3/4)

```
public interface Map<K, V> { // … suite
  // méthodes de remplacement
  V replace(K k, V v);
  boolean replace(K k, V v1, V v2);

  // méthodes de suppression
  void clear();
  V remove(Object k);
  boolean remove(Object k, Object v);

  // … à suivre
}
```

L'interface Map (4/4)

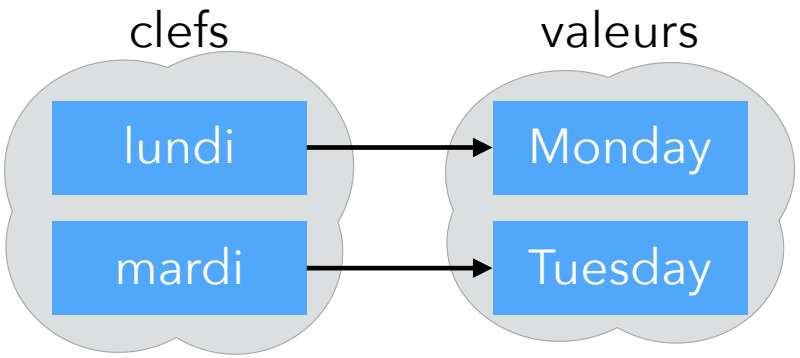
```
public interface Map<K, V> { // … suite
  // vues
  Set<K> keySet();
  Collection<V> values();
  Set<Map.Entry<K, V>> entrySet();

  // paire clef/valeur
  public static interface Entry<K, V> {
    K getKey();
    V getValue();
    void setValue(V v);
  }
}
```

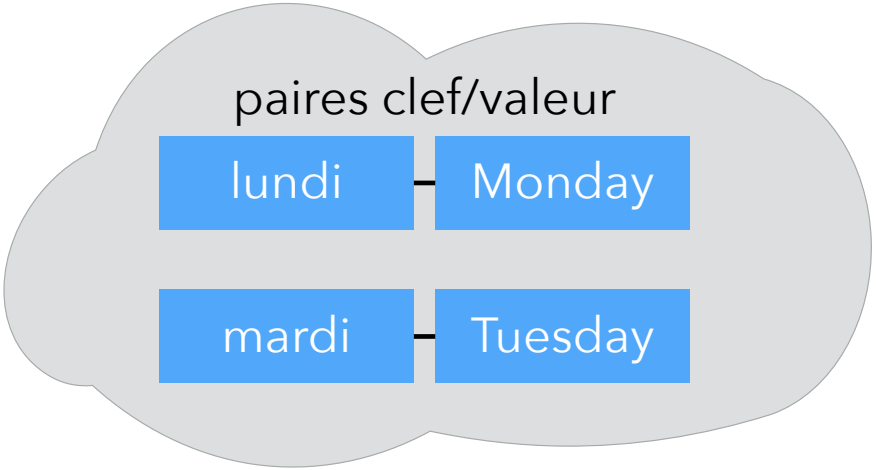
Table associative

Une table associative peut être vue comme…

…un ensemble de clefs, et un ensemble de valeurs, liés entre eux, ou



…un ensemble de paires clef/valeur.



Mises en œuvre de `Map`

Opération	`TreeMap`	`HashMap`
ajout (`put`), consultation (`get`), etc.	O(log *n*)	O(1)

Différence importante :

- `TreeMap` stocke ses clefs de manière triée,
- `HashMap` stocke ses clefs dans un ordre quelconque.

Ensembles / tables assoc.

Ensembles et tables associatives sont très similaires :

`Set<E>` ≈ `Map<E, Void>`

`Map<K, V>` ≈ `Set<Map.Entry<K,V>>`

Conséquences :

1. tout ce qui a été dit sur les éléments des ensembles s'applique aux clefs des tables associatives,
2. les uns peuvent être mis en œuvre au moyen des autres
   (en pratique en Java : `TreeSet` mis en œuvre au moyen de `TreeMap`, idem pour `HashSet`/`HashMap`).