Collections : choix de conception

Théorie et pratique de la programmation
Michel Schinz – 2013-03-25

Choix de conception

Lorsqu'on met en œuvre des collections, plusieurs questions se posent, parmi lesquelles :

- Comment représenter l'absence d'éléments dans une collection ? Par une valeur spéciale ou par `null` ?
- Comment visiter les éléments d'une collection ? Par itération ou par récursion ?
- Faut-il réutiliser la mise en œuvre d'une collection pour une autre ?
- Comment mettre à jour les collections ? En les modifiant ou en en produisant de nouvelles ?

Il n'y a pas forcément de meilleure réponse à chacune de ces questions, tout est affaire de compromis.

Absence d'éléments

Absence d'éléments

Toute collection est formée d'un certain nombre d'éléments. Il faut donc savoir comment représenter l'absence d'éléments.

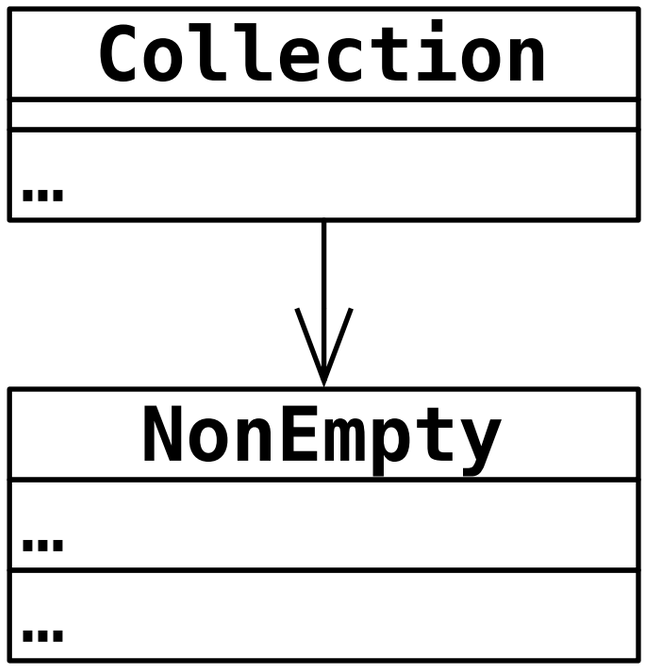
En Java, il y a deux possibilités de le faire :

- au moyen de la valeur `null`,
- au moyen d'un objet spécial.

Nous avons déjà utilisé les deux possibilités, puisque nos listes chaînées (y compris celles des tables de hachage) utilisent `null`, alors que nos arbres de recherche utilisent un objet spécial, à savoir une instance de la classe `Leaf`.

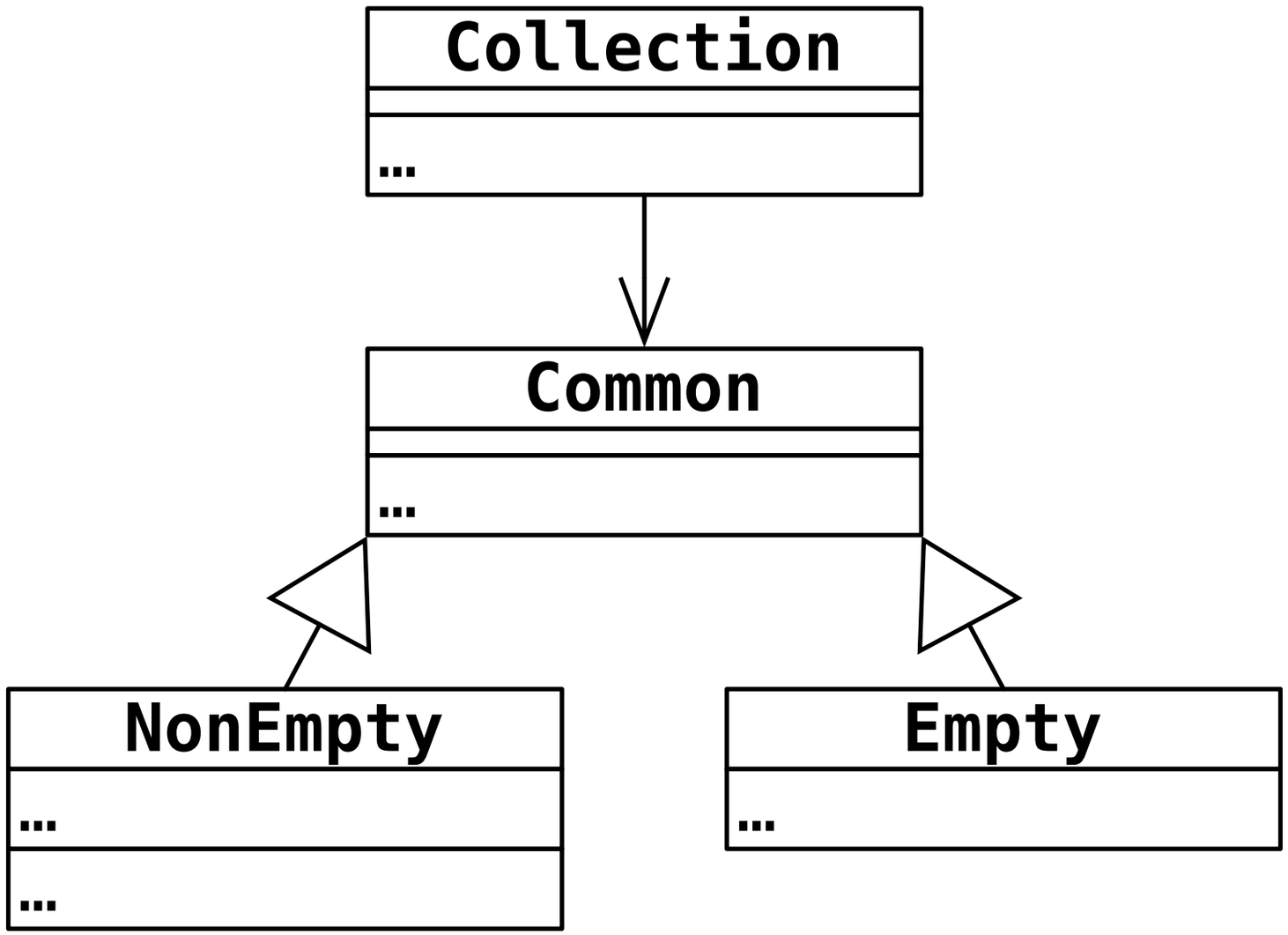
Utilisation de `null`

Utiliser `null` pour représenter l'absence d'éléments implique de ne définir une classe que pour le cas non vide (nœud ou entrée), et à utiliser `null` pour le cas vide.



Utilisation d'un objet

Utiliser un objet pour représenter l'absence d'éléments implique de définir une classe pour le cas non vide (nœud ou entrée), une pour le cas vide et une interface commune à ces deux classes :



Tests

Représenter l'absence d'éléments par `null` implique un grand nombre de tests de la forme :

```
if (node != null)
  // cas n°1 (non vide)
else
  // cas n°2 (vide)
```

L'utilisation d'un objet laisse la résolution dynamique des liens faire le test :

```
class NonEmpty { /* cas n°1 (non vide) */ }
class Empty { /* cas n°2 (vide) */ }
```

Comparaison

L'utilisation de `null` a l'avantage de ne pas demander la définition d'une classe pour le cas vide, ni d'une interface commune.

Son inconvénient principal est qu'elle implique de très nombreuses comparaisons avec `null`, comparaisons qu'il est facile d'oublier.

Parcours des collections

Parcours des collections

Les éléments d'une collection – ou un sous-ensemble d'entre-eux – doivent souvent être parcourus pour mettre en œuvre les opérations de base.

Ce parcours peut se faire de deux manières :

- itérative, c-à-d au moyen d'une boucle,
- récursive.

La représentation de l'absence d'éléments a une grande influence sur le type de parcours approprié : les collections utilisant `null` se prêtent bien à un parcours itératif, celles utilisant par un objet spécial à un parcours récursif.

Parcours récursif des arbres

Nos arbres de recherche utilisent un objet spécial (de type `Leaf`) pour représenter l'absence d'éléments. Le test d'appartenance est récursif :

```
class Node<E …> implements Tree<E> {
  boolean contains(E e) {
    int c = e.compareTo(v);
    if (c < 0) return s.contains(e);
    else if (c > 0) return g.contains(e);
    else return true;
  }
}
class Leaf<E …> implements Tree<E> {
  boolean contains(E e) { return false; }
}
```

Parcours récursif des arbres

La méthode `contains` de la collection (ici `TreeSet`) se contente d'appeler la méthode `contains` de la racine, qu'on sait être différente de `null` :

```
class TreeSet<E …> {
  private Tree<E> root = new Leaf<E>();
  public boolean contains(E elem) {
    return root.contains(elem);
  }
}
```

Parcours itératif des arbres

La recherche dans un arbre binaire de recherche terminé par `null` se fait aisément de manière itérative, directement dans la classe de la collection :

```
class TreeSetUsingNull<E …> {
  private Node<E> root = null;
  boolean contains(E e) {
    Node<E> node = root;
while (node != null) {
  int c = e.compareTo(node.v);
  if (c < 0) node = node.s;
  else if (c > 0) node = node.g;
  else return true;
}
return false; } }
```

Réutilisation de code

Réutilisation de code

La réutilisation de code est, généralement, une bonne chose. Dans les collections examinées, plusieurs opportunités de réutilisation existent, et les questions suivantes se posent donc :

- Faut-il mettre en œuvre les ensembles au moyen des tables associatives ?
- Faut-il mettre en œuvre les tables associatives au moyen des ensembles ?
- Faut-il réutiliser la mise en œuvre des listes chaînées pour les listes de collision des tables de hachage ?

Inconvénients

La réutilisation de code est généralement une bonne chose car elle limite la quantité de code à écrire, à tester et à déboguer.
Toutefois, elle peut avoir un coût en termes d'utilisation mémoire et de temps d'exécution. Ces facteurs sont importants à prendre en compte, particulièrement dans le cas des collections.
Pour quantifier le coût de la réutilisation, on peut compter le nombre de mots mémoire supplémentaires et les indirections additionnelles nécessaires pour accéder aux éléments qu'elle implique.

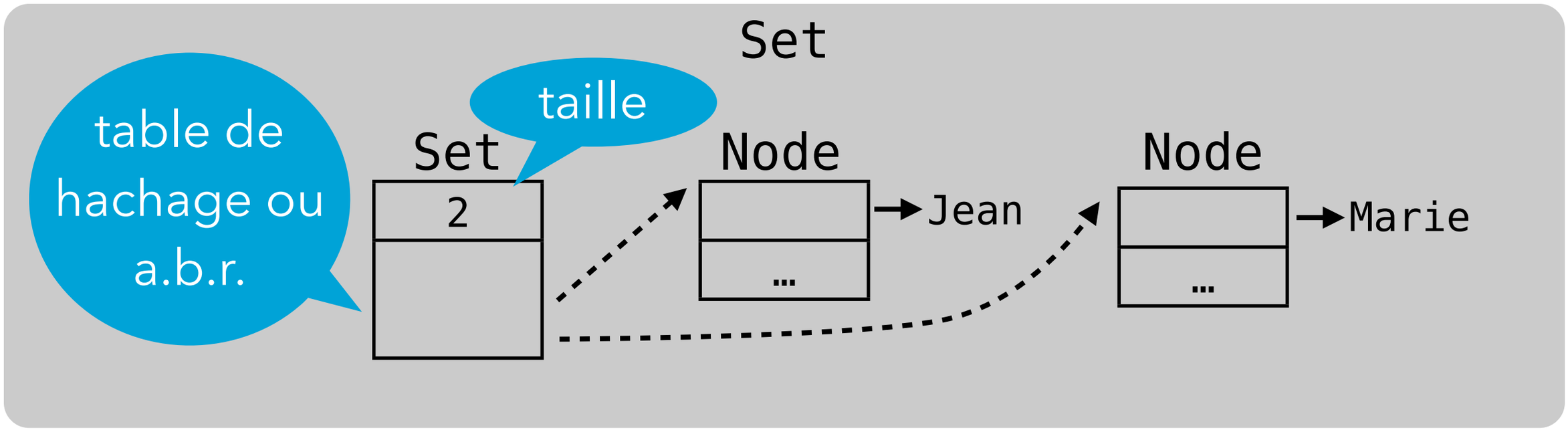
Ensembles par tables

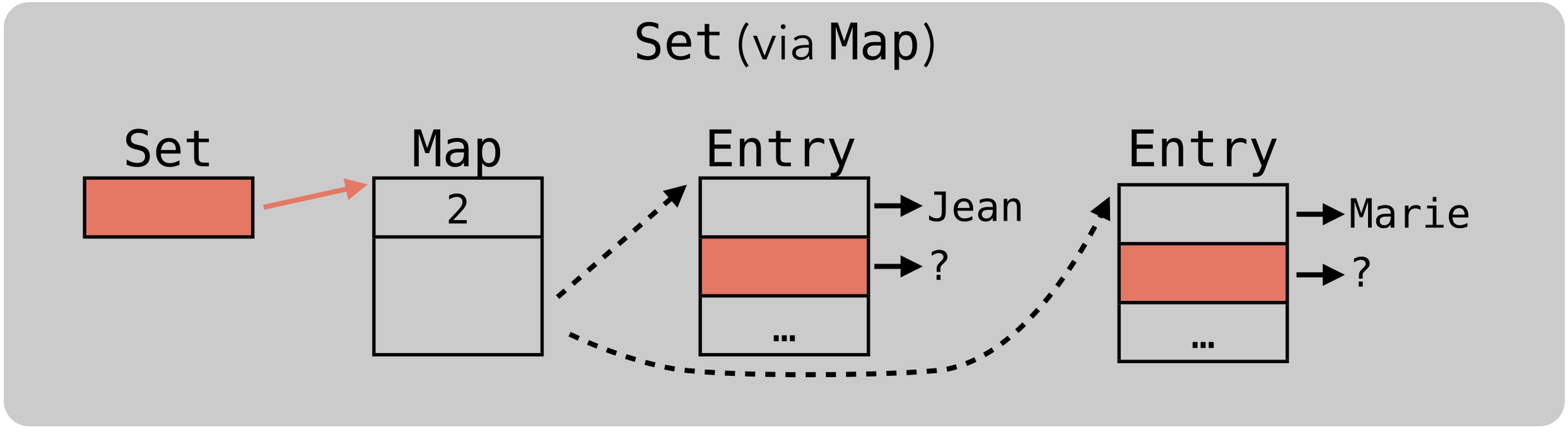
Les ensembles peuvent être mis en œuvre au moyen de tables associatives dans lesquelles les valeurs n'ont pas d'importance et sont ignorées.

En d'autres termes, on met en œuvre `Set<E>` au moyen de `Map<E, Object>`.

C'est la solution qui été retenue dans la mise en œuvre originale des collections Java.

Set via Map


Set
table de hachage ou a.b.r.
taille
Set
2
Node
…
Jean
Node
…
Marie


Set (via Map)
Set
Map
2
Entry
…
Jean
?
Entry
…
Marie
?

Tables par ensembles

Il est aussi possible de mettre en œuvre les tables associatives au moyen d'ensembles, en définissant judicieusement la notion d'égalité, de hachage et de comparaison des entrées.

En d'autre terme, une table associative de type `Map<K, V>` peut être mis en œuvre par un ensemble d'entrées (paires) de type `Set<Entry<K, V>>`.

Map via Set

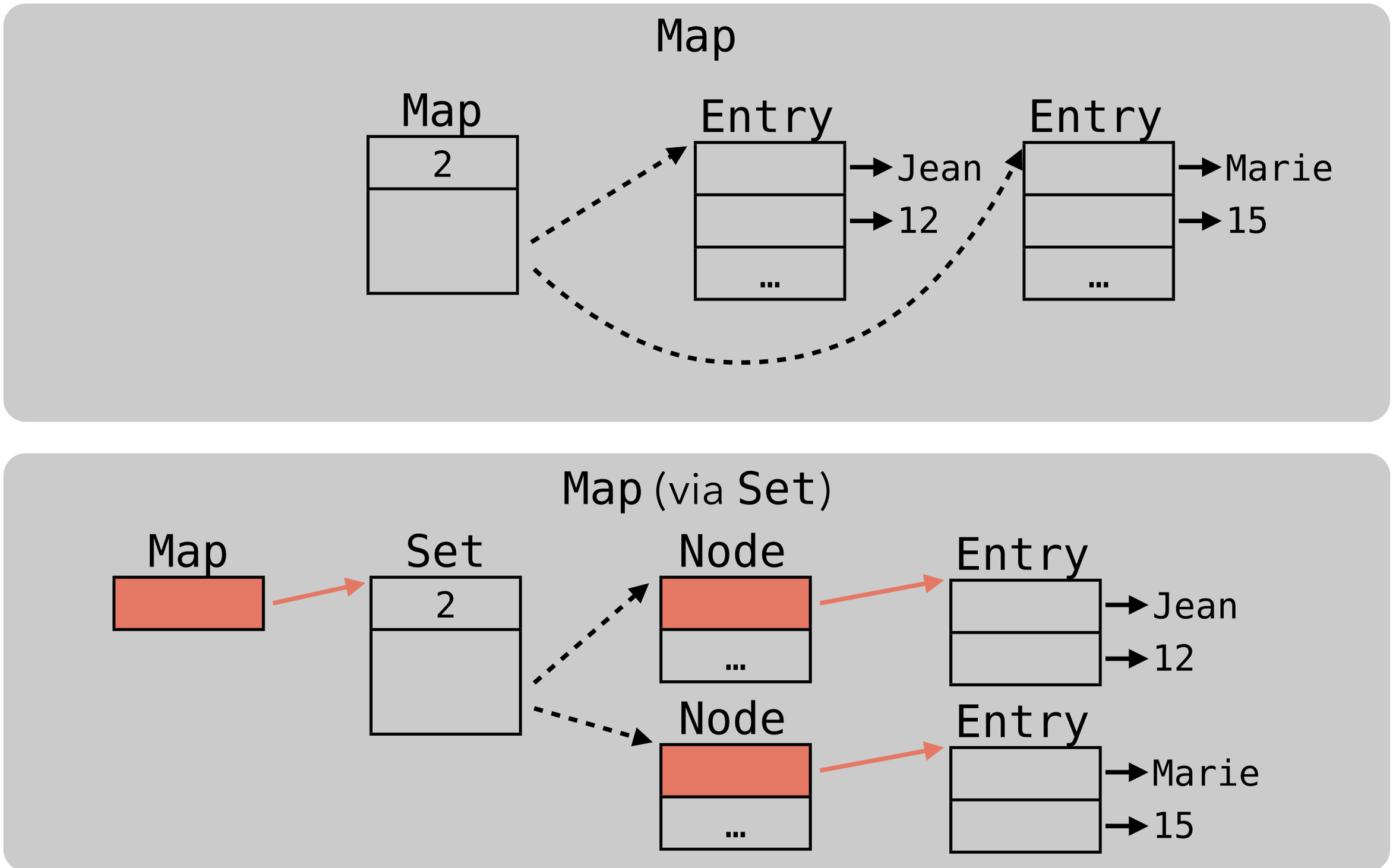

Map
Map
2
Entry
Jean
12
…
Entry
Marie
15
…
Map (via Set)
Map
Set
2
Node
…
Node
…
Entry
Jean
12
Entry
Marie
15

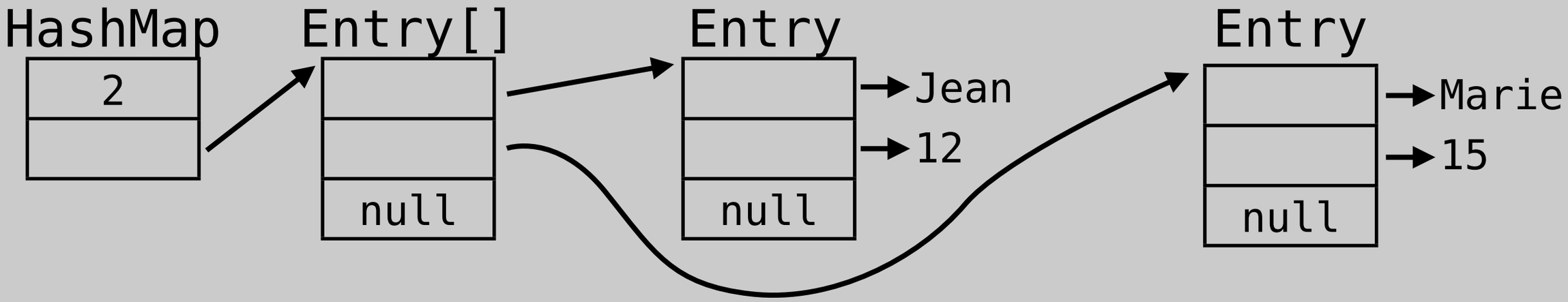
Table de hachage via listes

Les tables de hachage que nous avons examinées gèrent les collisions par chaînage.

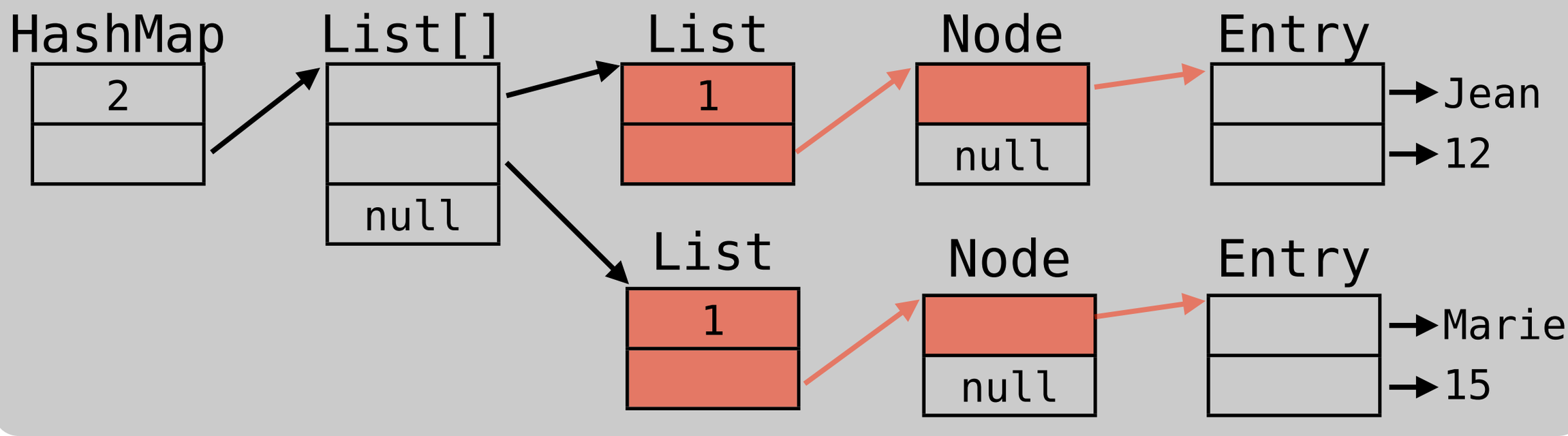
Les listes chaînées contenant les éléments en collision peuvent être mise en œuvre en réutilisant les listes chaînées.

HashMap réutilisant List

HashMap (sans réutilisation de **List**)



HashMap (avec réutilisation de **List**)



Coûts de la réutilisation

Technique	Coût mémoire	Coût accès
`Set` via `Map`	≅1+n	≅1
`Map` via `Set`	≅1+n	≅2
`List` pour `HashMap`	≅2c+n	≅2
`List` pour `HashSet`	≅2c	≅1

n = nombre d'éléments dans la collection

c = capacité de la table de hachage (taille du tableau)

Mutabilité des collections

Collections (im)mutables

Toutes les collections que nous avons examinées étaient modifiables (ou mutables), c-à-d que les opérations d'ajout, de suppression et de modification d'éléments changent la collection à laquelle elles sont appliquées.
Il est également possible de définir des collections non modifiables, dont les opérations sus-mentionnées produisent une nouvelle collection.

Listes non modifiables

Rendre une collection non modifiable implique de changer la signature des méthodes qui modifient son contenu. Généralement, le type de retour de ces méthodes change de `void` au type de la collection.

Exemple pour une variante non modifiables des listes :

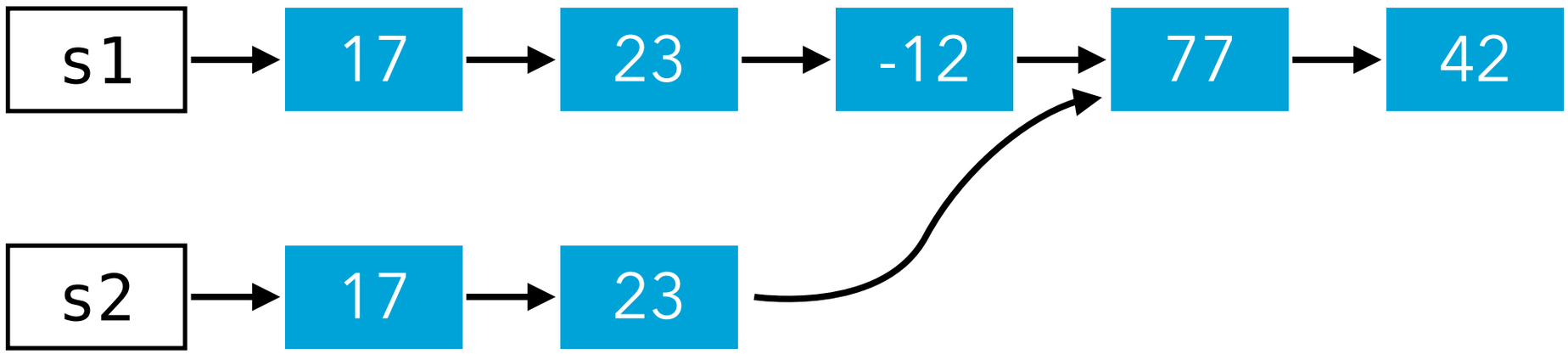
```
interface ImmutableList<E> {
  boolean isEmpty();
  int size();
  ImmutableList<E> add(E newElem);
  ImmutableList<E> remove(int index);
  E get(int index);
  ImmutableList<E> set(int index, E elem);
  Iterator<E> iterator();
}
```

Immutabilité et partage

Il peut sembler très coûteux de produire à chaque fois une nouvelle version de la collection complète.
Toutefois, cela n'est pas toujours nécessaire : étant donné que les nœuds (ou entrées) des collections non modifiables sont également non modifiables, il est souvent possible de les partager entre les versions successives de la collection.
Cela est très simple pour les listes (simplement chaînées) et les arbres binaires de recherche, pas du tout pour les tables de hachage...

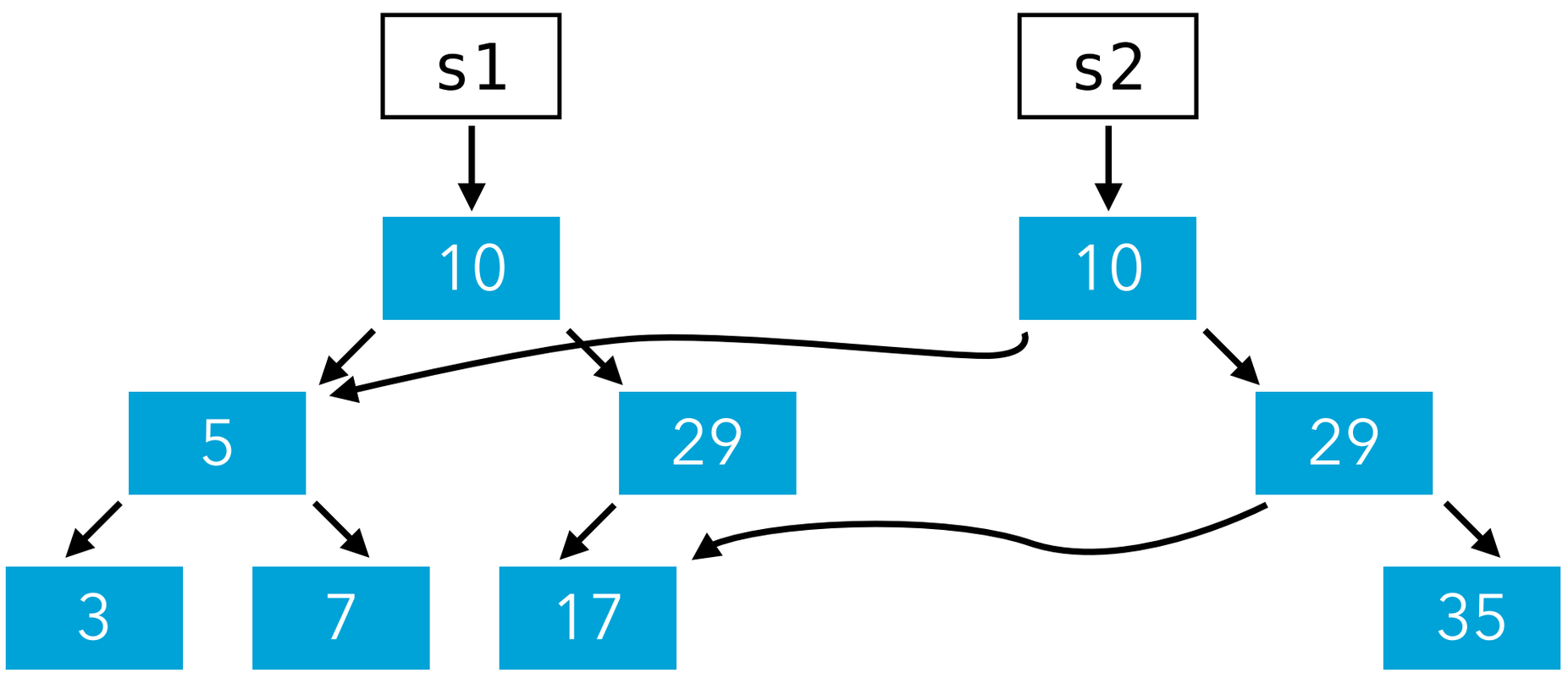
Partage dans une liste

```
ImmutableList<Integer> s1 = …;
ImmutableList<Integer> s2 = s1.remove(2);
```



Partage dans un a.b.r.

```
ImmutableSet<Integer> s1 = …;
ImmutableSet<Integer> s2 = s1.add(35);
```



Intérêt des collections n. m.

Les collections non modifiables sont très intéressantes dans plusieurs contextes :

- Lorsqu'on désire toujours avoir accès aux anciennes versions d'une collection, p.ex. pour mettre en œuvre l'annulation d'action (*undo*) dans un programme interactif. Les techniques de mise en œuvre des collections non modifiables sont d'ailleurs à la base de systèmes de gestion de version comme git et Subversion.
- Lorsqu'on fait de la programmation concurrente, car les collections non modifiables ne posent par définition aucun problème de synchronisation.

Collections Java non mod.

L'API Java n'offre pas de type séparé pour les collections non modifiables.

Toutefois, toutes les opérations de modification des collections sont désignées comme optionnelle, et peuvent donc simplement lever `UnsupportedOperationExc…`

Des méthodes statiques de la classe `Collections` permettent de rendre une collection non modifiable :

```
<T> List<T> unmodifiableList(List<T> l)
…
```

Mais attention : ces méthodes ne peuvent pas garantir que la collection sous-jacente (ici `l`) ne va pas changer !

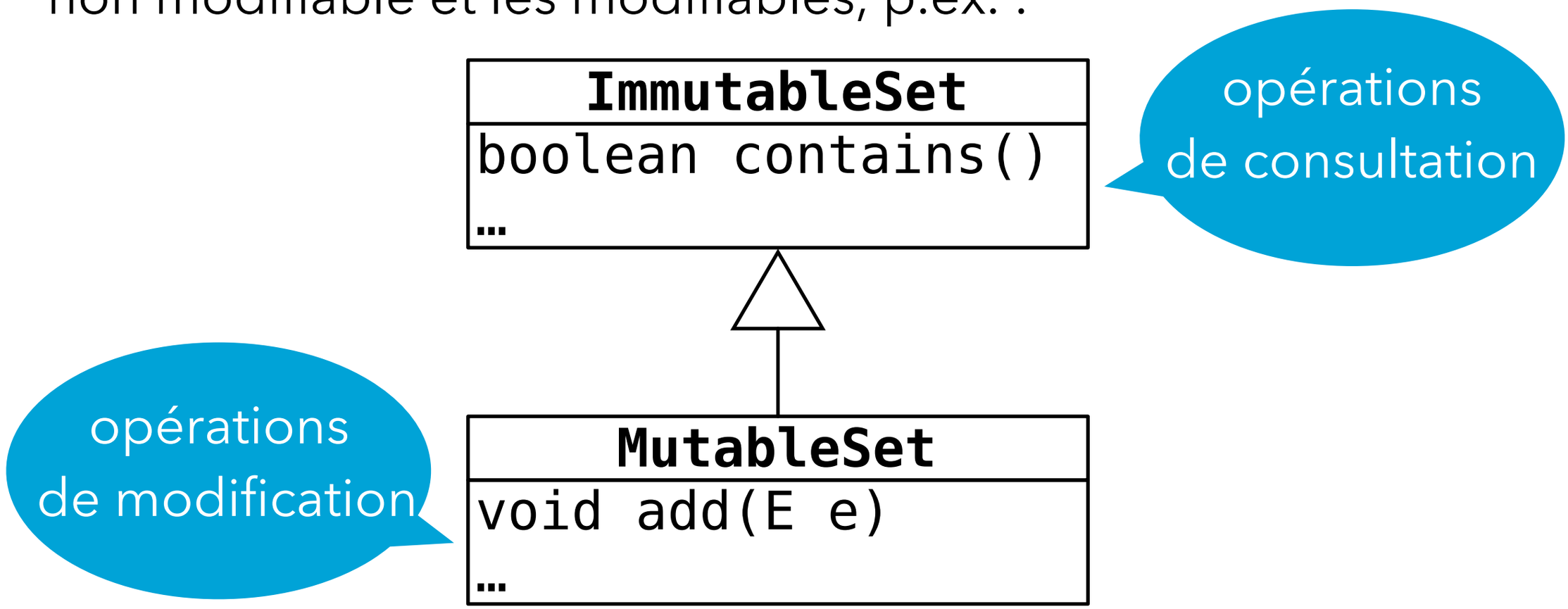
Collections Java non mod.

La notion de collection immutable offerte par Java n'est pas tout à fait la même que celle présentée auparavant.

En Java, les opérations de modification sont *interdites*. Avec de « vraies » collections immutables, ces opérations sont autorisées mais produisent une nouvelle collection.

Dangers de sous-typage

Certaines API de collections (comme celle d'Apple) définissent une relation de sous-typage entre les collections non modifiable et les modifiables, p.ex. :



L'idée peut paraître séduisante de prime abord, mais se révèle très dangereuse en pratique... Pourquoi ?

Résumé

Les choix de conception des collections faits dans les leçons précédentes ne sont en aucun cas les seuls possibles.

Concevoir une bonne API de collections implique de bien réfléchir aux réponses à donner aux différentes questions posées dans cette leçon.