Collections : tables associatives

Pratique de la programmation orientée-objet

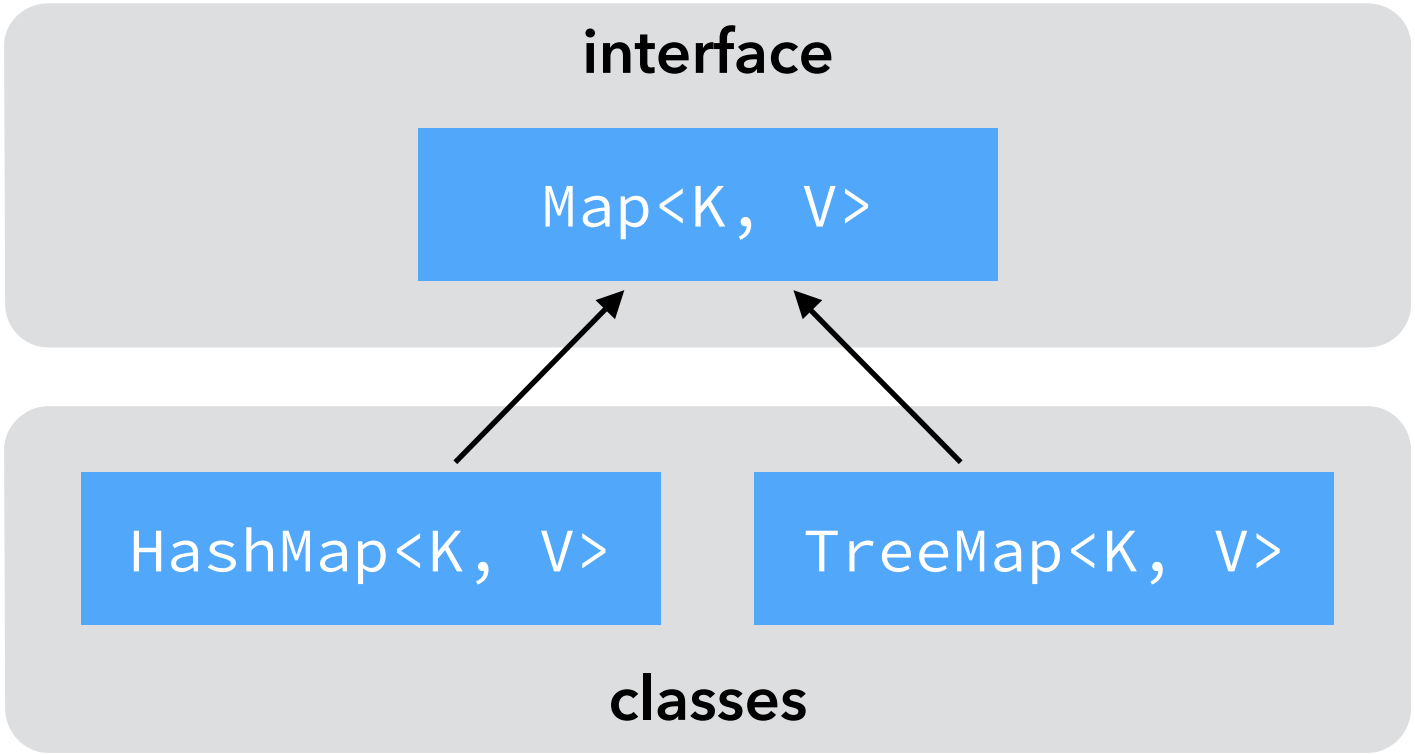
Michel Schinz – 2015-03-02

Collection n°2 :
la table associative

Tables associatives

Une **table associative** (*map*) ou **dictionnaire** (*dictionary*) est une collection qui associe des **valeurs** (*values*) à des **clefs** (*keys*).
Par exemple, l'index d'un livre est une table associative qui associe à différents mots (les clefs) la liste des numéros de pages sur lesquelles ce mot apparaît (les valeurs).
En informatique, un tableau – ou une liste – peut être vu comme un cas particulier d'une table associative dont les clefs sont les entiers compris entre 0 et le prédécesseur de la taille du tableau, et les valeurs sont les éléments du tableau.

Tables associatives en Java



(D'autres mises en œuvre existent, mais sont rarement utilisées, donc ignorées ici.)

L'interface `Map`

Le concept de table associative est représenté dans l'API Java par l'interface `Map` du paquetage `java.util`. Cette interface est générique et prend *deux* paramètres de type nommés `K` et `V` qui représentant respectivement le type des clefs et celui des valeurs :

```
public interface Map<K, V> {
  // … méthodes
}
```

A noter que cette interface n'hérite ni de `Collection` – ce qui est normal – ni de `Iterable` – ce qui l'est moins.

Les principales méthodes de cette interface sont présentées ci-après, parfois avec un type simplifié pour des raisons pédagogiques.

Méthodes de consultation

Les méthodes ci-dessous, comme toutes celles qui ne modifient pas la table associative, sont obligatoires :

- `boolean isEmpty()` : retourne vrai ssi la table associative est vide.
- `int size()` : retourne le nombre d'associations clef/valeur contenues dans la table associative.
- `boolean containsKey(Object k)` : retourne vrai ssi la table contient la clef donnée.
- `boolean containsValue(Object v)` : retourne vrai ssi la table contient la valeur donnée (rarement utile !).

Le type de l'argument des deux dernières méthodes devrait être `K` et `V` respectivement, mais est `Object` pour de malheureuses raisons historiques.

Méthodes de consultation

Les méthodes ci-dessous permettent d'obtenir la valeur associée à une clef et sont probablement les méthodes les plus utilisées des tables associatives :

- `V get(Object k)` : retourne la valeur associée à la clef donnée, ou `null` si cette clef n'est pas présente dans la table.
- `V getOrDefault(Object k, V d)` : retourne la valeur associée à la clef donnée, ou la valeur `d` si la clef n'est pas présente dans la table.

Là aussi, le type de la clef devrait être `K` et pas `Object`.

Méthodes de modification

Les méthodes d'ajout/modification ci-dessous sont comme d'habitude optionnelles :

- `V put(K k, V v)` : associe la valeur donnée avec la clef donnée et retourne la valeur qui lui était associée ou `null` s'il n'y en avait aucune,
- `V putIfAbsent(K k, V v)` : si la clef donnée n'est pas encore associée à une valeur, l'associe à la valeur donnée et retourne `null` ; sinon, retourne la valeur associée à la clef,
- `void putAll(Map<K, V> m)` : copie toutes les associations clef/valeur de la table donnée dans la table à laquelle la méthode est appliquée.

Méthodes de remplacement

Les méthodes de remplacement ci-dessous sont optionnelles :

- `V replace(K k, V v)` : si la table contient une valeur associée à la clef donnée, la remplace par la valeur donnée et retourne l'ancienne valeur ; sinon, ne modifie pas la table et retourne `null`,
- `boolean replace(K k, V v1, V v2)` : si la table associe actuellement la clef donnée à la première valeur donnée, lui associe la seconde valeur donnée et retourne vrai ; sinon, ne modifie pas la table et retourne faux.

Méthodes de suppression

Les méthodes de suppression ci-dessous sont optionnelles :

- `void clear()` : supprime toutes les associations clef/valeur de la table,
- `V remove(Object k)` : supprime la clef donnée de la table ainsi que la valeur qui lui était associée ; retourne cette dernière ou `null` si la clef n'était pas présente,
- `boolean remove(Object k, Object v)` : supprime la clef donnée de la table ssi elle est associée à la valeur donnée ; retourne vrai ssi la table a été modifiée en conséquence.

Comme précédemment, l'utilisation de `Object` en lieu et place de `K` ou `V` est une erreur historique.

Vues sur les clefs et valeurs

Les méthodes ci-dessous permettent d'obtenir des vues sur les clefs, les valeurs ou les paires clefs/valeurs.

- `Set<K> keySet()` : retourne une vue sur l'ensemble des clefs de la table,
- `Collection<V> values()` : retourne une vue sur les valeurs de la table,
- `Set<Map.Entry<K, V>> entrySet()` : retourne une vue sur l'ensemble des associations clef/valeur de la table.

Si la table est modifiable, ces vues le sont également et les modifications qu'on y apporte sont reportées sur la table – et inversement.

L'interface `Map.Entry`

L'interface `Map.Entry` – imbriquée statiquement dans l'interface `Map` – représente une association entre une clef et une valeur, aussi appelée paire clef/valeur.

Tout comme l'interface `Map`, `Entry` est générique et prend deux paramètres de type : le type `K` de la clef et le type `V` de la valeur qui lui est associée.

```
public interface Map<K, V> {
  public static interface Entry<K, V> {
    // … méthodes
  }
}
```

L'interface `Map.Entry`

L'interface `Map.Entry` offre deux méthodes permettant respectivement d'accéder à la clef et à la valeur de la paire clef/valeur qu'elle représente :

- `K getKey()` : retourne la clef de la paire,
- `V getValue()` : retourne la valeur de la paire.

De plus, elle offre une méthode optionnelle permettant de modifier la valeur associée à la clef :

- `void setValue(V v)` : remplace la valeur de la paire par celle donnée.

Itération

L'interface `Map` n'étend pas l'interface `Iterable` et il n'est donc pas possible d'itérer directement sur les paires clef/valeur d'une table associative.

On peut par contre itérer sur ces paires clef/valeur par l'intermédiaire de la vue fournie par `entrySet`, p.ex. :

```
Map<String, Integer> m = …;
for (Map.Entry<String, Integer> e:
        m.entrySet()) {
  System.out.println(e.getKey() + " => "
                     + e.getValue());
}
```

Attention : l'ordre des paires est – en général – quelconque et peut même changer d'une exécution à l'autre !

Tables immuables

Tout comme pour les listes, la classe `Collections` offre des méthodes permettant de créer des tables associatives immuables :

- `<K, V> Map<K, V> emptyMap()` : retourne une table associative vide immuable.
- `<K, V> Map<K, V> singletonMap(K k, V v)` : retourne une table associative contenant uniquement l'association entre la clef et la valeur donnés.

Vues non modifiables

Tout comme pour les listes, la classe `Collections` offre une méthode permettant d'obtenir une vue non modifiable sur une table associative :

```
<K,V> Map<K, V> unmodifiableMap(Map<K, V> m)
```

Comme toujours, prenez garde au fait qu'il s'agit d'une vue et que toute éventuelle modification à la table sous-jacente sera répercutée sur la vue ! Celle-ci est donc non modifiable mais pas forcément immuable.

Règle des tables immuables

Pour obtenir une table associative immuable à partir d'une table associative quelconque, obtenez une vue non modifiable d'une copie de cette table.

La copie peut se faire au moyen du constructeur de copie de l'une des mises en œuvre. Par exemple, en utilisant `HashMap`, on obtient :

```
Map<…> immutableMap =
  Collections.unmodifiableMap(
    new HashMap<>(map));
```

Exemple

Une table associative de traduction de français en anglais des noms de saisons peut être construite et utilisée ainsi :

```
Map<String, String> s = new HashMap<>();
s.put("printemps", "spring");
s.put("été", "summer");
s.put("automne", "autumn");
s.put("hiver", "winter");
for (Map.Entry<String,String> e: s.entrySet())
  System.out.println("En anglais, " +
    e.getKey() + " se dit " + e.getValue());
```

Mise en œuvre n°1 : listes associatives

Listes associatives

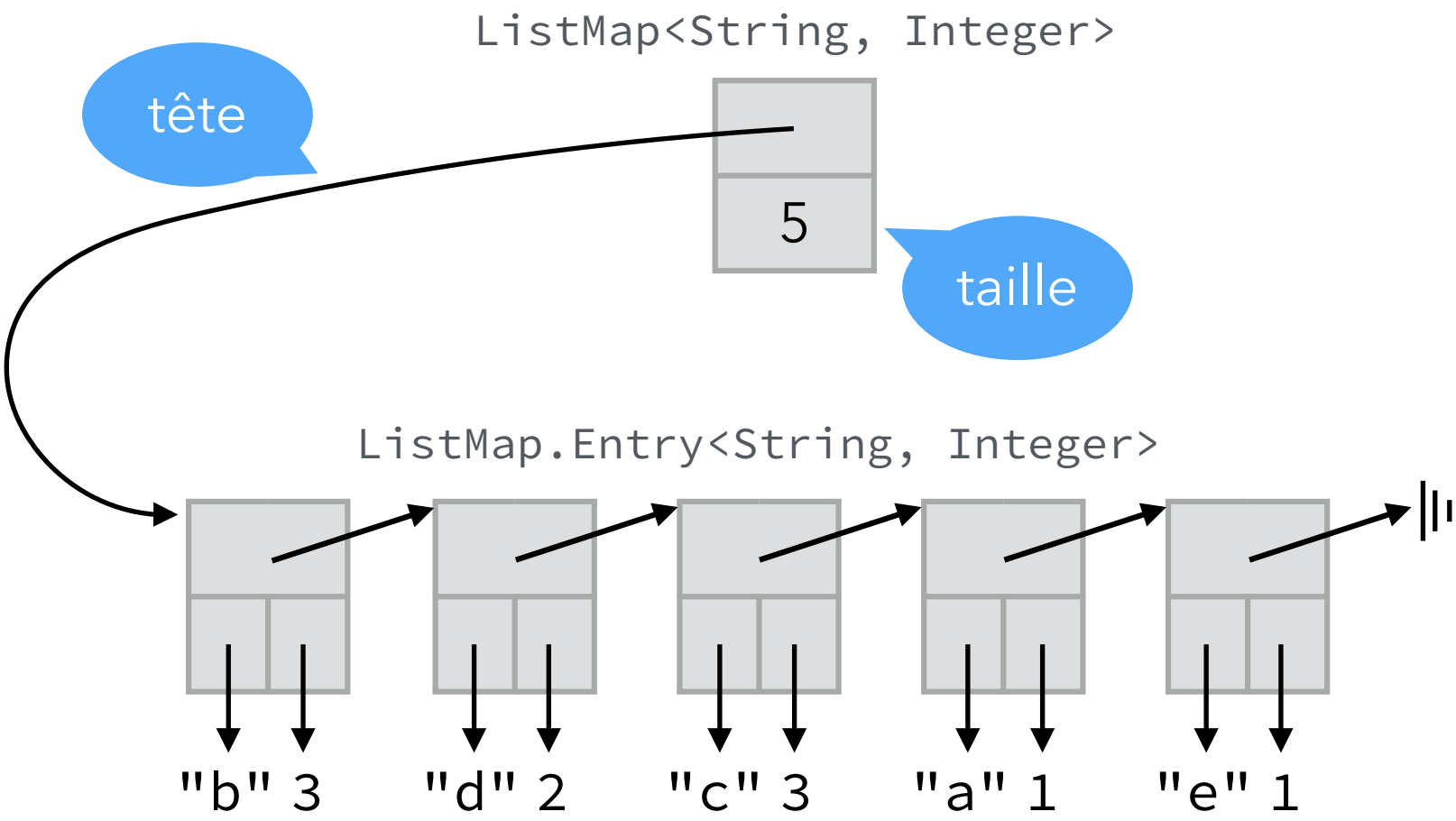
Une table associative peut être mise en œuvre très simplement au moyen d'une liste de paires clef/valeur. Une telle liste s'appelle **liste associative**.
En pratique, les listes associatives sont trop peu efficaces pour être utilisées, sauf peut-être pour de petites tables. Elles sont néanmoins pédagogiquement intéressantes car elles facilitent la compréhension des autres mises en œuvre.

ListMap

La bibliothèque Java n'offre pas de mise en œuvre des tables associatives sous la forme de listes associatives. Toutefois, pour illustrer l'idée des listes associatives, nous ferons ici l'hypothèse qu'une telle mise en œuvre est fournie sous la forme d'une classe nommée `ListMap`.

ListMap

(Utilisant une liste simplement chaînée)



Listes associatives

Les listes associatives sont peu efficaces, dans le sens où les opérations principales – ajout/remplacement d'une association, recherche de la valeur associée à une clef – ont une complexité de $O(n)$.

Peut-on faire mieux ? Sous certaines conditions, oui :

- s'il est possible de « hacher » les clefs, on peut utiliser une table de hachage pour représenter une table associative,
- s'il est possible d'ordonner les clefs, on peut utiliser un arbre binaire de recherche pour représenter une table associative.

Ces deux possibilités sont examinées plus loin.

Mise en œuvre n°2 : tables de hachage

Hachage

Le **hachage** (*hashing*) est une technique permettant de transformer une donnée quelconque en un entier compris dans un intervalle borné.
Cette transformation est faite par une **fonction de hachage** (*hash function*) qui, appliquée à une donnée, produit sa **valeur de hachage** (*hash value*).

Exemple

Un exemple de fonction de hachage utilisable sur les chaînes de caractères composées uniquement des 26 lettres non accentuées de l'alphabet latin est :

h(*c*) = somme du code des lettres de *c*, modulo 100

où le code d'une lettre est sa position dans l'alphabet – a étant à la position 1, b à la position 2, etc.

Par exemple :

h(`chien`) = (3 + 8 + 9 + 5 + 14) mod 100 = 39

h(`niche`) = (14 + 9 + 3 + 8 + 5) mod 100 = 39

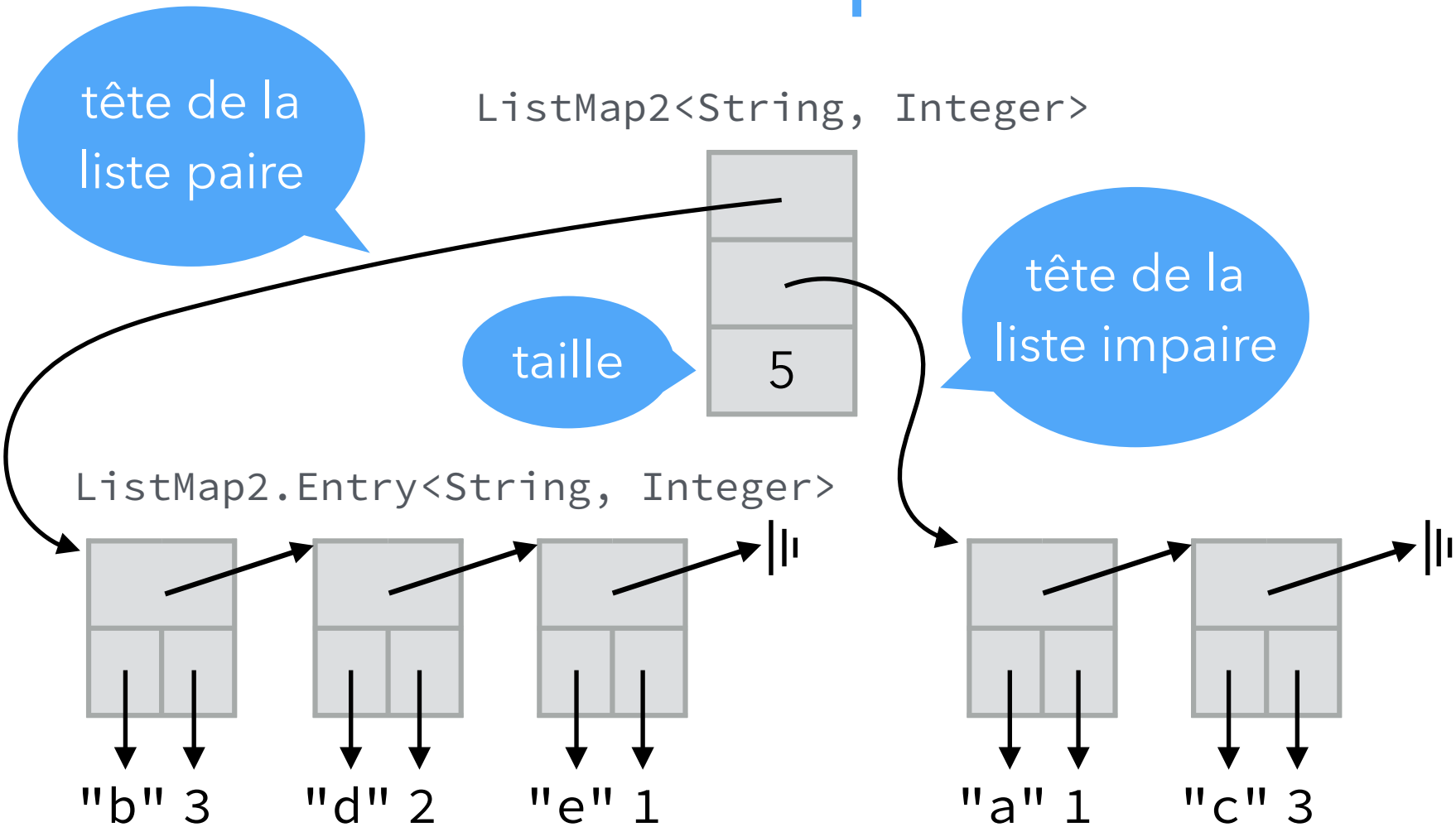
h(`zoologie`) =

(26 + 15 + 15 + 12 + 15 + 7 + 9 + 5) mod 100 = 4

Hachage et liste associative

Au moyen d'une fonction de hachage sur les clefs, il est aisé de multiplier par deux les performances d'une table associative mise en œuvre au moyen de listes associatives.
L'idée est de représenter la table associative par *deux* listes associatives plutôt qu'une seule. On détermine dans laquelle de ces deux listes stocker ou rechercher une clef en fonction de la parité de sa valeur de hachage.
Comme précédemment, faisons l'hypothèse qu'une telle mise en œuvre existe dans l'API Java, sous la forme de la classe `ListMap2`.

ListMap2



Hypothèse : la valeur de hachage des clefs b, d et e est paire, celle des clefs a et c impaire.

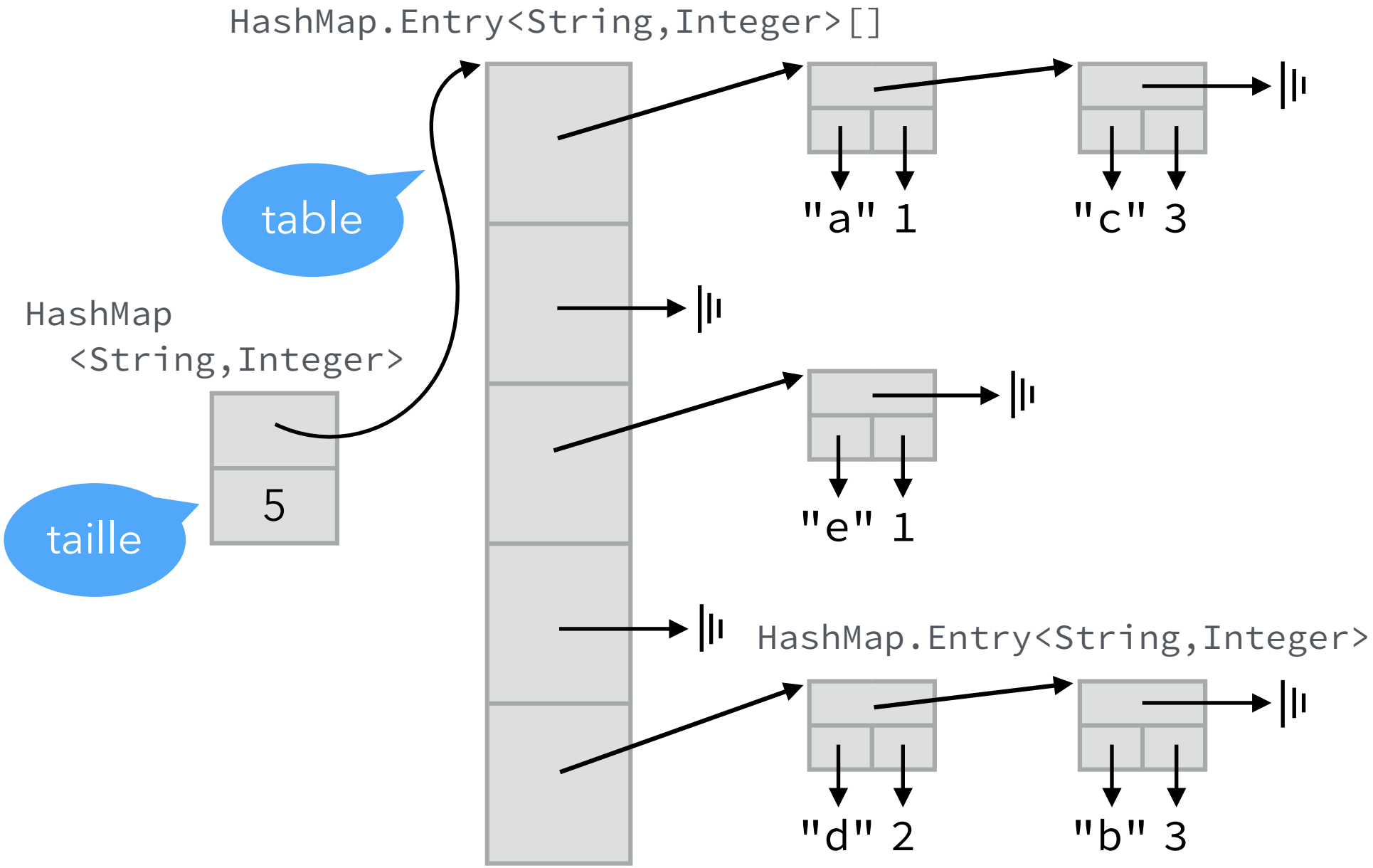
Table de hachage

Cette idée peut bien entendu se généraliser à *n* listes associatives stockées dans un tableau.
L'indice de la liste à utiliser pour rechercher ou stocker une clef est simplement la valeur de hachage de cette clef modulo le nombre de listes !
Cette structure de données, composée d'un tableau de listes (ici associatives) indexé en fonction de la valeur de hachage de ses éléments (ici les clefs) s'appelle une **table de hachage** (*hash table*).

HashMap

La classe `HashMap` de l'API Java met en œuvre les tables associatives au moyen d'une table de hachage.
Dans une table de hachage, le rapport entre le nombre d'éléments présents dans la table et sa capacité est appelé le **facteur de charge** (*load factor*). La classe `HashMap` s'arrange – en redimensionnant au besoin la table – pour que ce facteur reste proche de 0.75. Cette valeur offre normalement un bon compromis entre utilisation mémoire et efficacité.

HashMap


HashMap.Entry<String,Integer>[]
table
HashMap
<String,Integer>
5
taille
"a" 1
"c" 3
"e" 1
HashMap.Entry<String,Integer>
"d" 2
"b" 3

HashMap

La mise en œuvre des tables associatives au moyen de tables de hachage est très efficace si :

1. la fonction de hachage est en O(1), et
2. les listes stockées dans la table ont une longueur proche de 1.

Sous ces conditions, les principales opérations sur la table de hachage (insertion, recherche) sont en O(1) !

Cette caractéristique fait des tables de hachage la mise en œuvre la plus efficace qui soit des tables associatives – et, comme nous le verrons, des ensembles.

Mise en œuvre n°3 : arbres de recherche

Ordre et tables associatives

En dehors du hachage, une autre caractéristique des clefs peut être utilisée pour mettre en œuvre les tables associatives de manière efficace : la possibilité de les ordonner (totalement).
L'idée consiste à stocker les paires clef/valeur de la table *triées* par clef. Il devient ainsi possible de rechercher une clef par dichotomie, avec une complexité en $O(\log n)$ plutôt que $O(n)$ si les paires ne sont pas triées.
Une structure de données permettant de stocker ainsi un ensemble trié de valeurs est l'arbre de recherche.

Arbre de recherche

Un **arbre** (**binaire**) **de recherche** (*binary search tree*) est un arbre composé de nœuds ayant chacun deux fils, qui peuvent être vides.

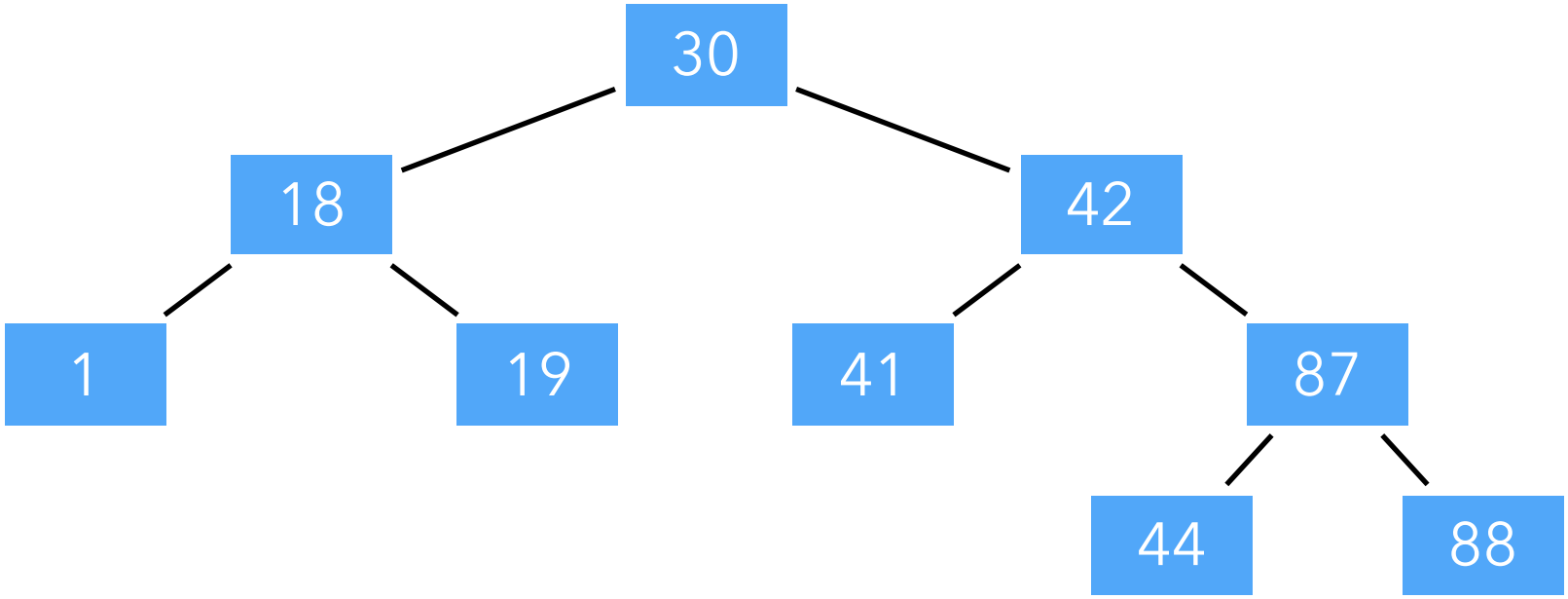
A chaque nœud est associé un élément et l'arbre est organisé de manière à ce que l'invariant suivant soit respecté :

- tous les éléments du sous-arbre gauche d'un nœud sont strictement plus petits que celui du nœud, et
- tous les éléments du sous-arbre droite d'un nœud sont strictement plus grands que celui du nœud.

Cette organisation permet l'accès rapide à un élément, car soit il se trouve à la racine, soit on sait dans lequel des deux fils poursuivre la recherche.

Arbre de recherche

Invariant : tous les éléments du fils gauche d'un nœud sont strictement plus petits que celui du nœud, ceux du fils droit strictement plus grands.



Note : cet exemple d'arbre de recherche ne représente *pas* une table associative !

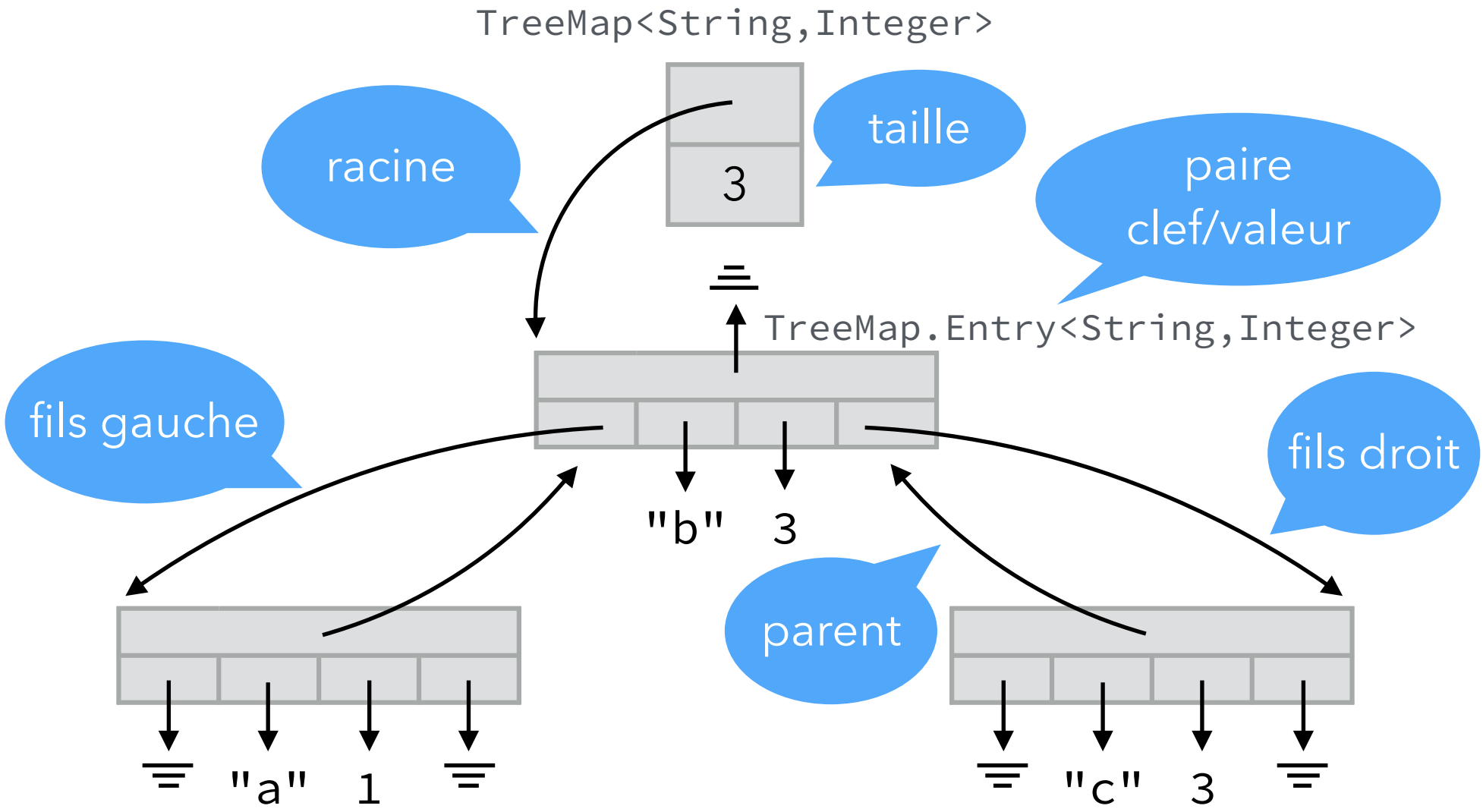
TreeMap

Un arbre de recherche peut être utilisé pour stocker les paires clef/valeur d'une table associative.
Les nœuds d'un tel arbre sont les paires clef/valeur, et ils sont organisés de manière à ce que, pour chaque nœud :

- les clefs de tous les nœuds du sous-arbre gauche soient strictement plus petites que celle du nœud, et
- les clefs de tous les nœuds du sous-arbre droit soient strictement plus grandes que celle du nœud.

La classe `TreeMap` de l'API Java utilise cette technique pour mettre en œuvre les tables associatives.

TreeMap


TreeMap<String,Integer>
racine
taille
3
paire
clef/valeur
TreeMap.Entry<String,Integer>
fils gauche
fils droit
"b"
3
parent
"a"
1
"c"
3

Ordre d'itération

Etant donné que les paires clef/valeur sont stockées par ordre croissant de clefs dans l'arbre binaire de recherche, la classe `TreeMap` garantit que l'itération sur ses clefs – ou ses paires clef/valeur – se fait dans cet ordre.

Pour sa part, `HashMap` ne fait aucune garantie à ce sujet, au point que l'ordre de parcours d'une même table associative de ce type peut varier d'une exécution à l'autre !

TreeMap

La mise en œuvre des tables associatives au moyen d'un arbre binaire de recherche est bien plus efficace que celle basée sur les listes associatives, puisque les principales opérations (insertion, recherche) ont une complexité de $O(\log n)$ plutôt que $O(n)$.
Toutefois, étant donné que ces mêmes opérations ont une complexité de $O(1)$ avec les tables de hachage, ces dernières restent généralement préférables.
La classe `TreeMap` est surtout intéressante dans le cas où il est important de pouvoir parcourir les clefs dans l'ordre.

Règle HashMap/TreeMap

Utilisez HashMap comme mise en œuvre des tables associatives en Java, sauf lorsqu'il est utile de parcourir les clefs en ordre croissant, auquel cas vous pourrez leur préférer TreeMap.

HashMap a d'une part l'avantage d'être généralement plus rapide que TreeMap, et d'autre part celui de ne pas demander à ce que les clefs de la table puissent être ordonnées.

Egalité

Tables associatives et égalité

Même si cela n'a pas été mentionné explicitement jusqu'ici, il est clair que les tables associatives font l'hypothèse suivante au sujet de leurs clefs :

> Etant données deux clefs, il est toujours possible de déterminer si elles sont égales ou non.

Par exemple, la méthode `get` – qui retourne la valeur associée à une clef donnée – doit déterminer si oui ou non la clef qu'on lui passe est égale à l'une des clefs stockée dans la table.

Il convient donc d'examiner en détail la notion d'égalité et sa mise en œuvre en Java.

Formes d'égalité

On peut distinguer deux formes d'égalité dans un langage comme Java :

1. l'égalité **par référence** (ou **par identité**), qui considère deux objets égaux si et seulement si il s'agit d'un seul et même objet,
2. l'égalité **par structure** (ou **structurelle**), qui considère deux objets égaux si et seulement si les valeurs de leurs attributs – ou d'un sous-ensemble d'entre-eux – sont égales.

L'égalité par référence est la plus discriminante : deux objets égaux par référence sont toujours égaux par structure, mais deux objets égaux par structure ne sont pas forcément égaux par référence.

Formes d'égalité

Par exemple, dans l'extrait de programme suivant :

```
class C {
  private final int x;
  public C(int x) { this.x = x; }
}
C c1 = new C(10);
C c2 = c1;
C c3 = new C(10);
```

les objets `c1`, `c2` et `c3` sont structurellement égaux – leur unique attribut, `x`, ayant la même valeur – mais seuls `c1` et `c2` sont de plus égaux par référence.

Formes d'égalité en Java

En Java, l'égalité par référence est prédéfinie sous la forme de l'opérateur ==. De plus, la méthode `equals` définie dans `Object` – et donc héritée par toute classe qui ne la redéfinit pas – effectue une comparaison par référence.
L'égalité structurelle n'est par contre pas prédéfinie en Java. Pour que les instances d'une classe soient comparées par structure, il faut que celle-ci redéfinisse la méthode `equals` et mette en œuvre l'égalité structurelle.

Stabilité de `equals`

Lors d'une redéfinition de `equals`, il est important de s'assurer que celle-ci est **stable**, dans le sens où deux objets considérés comme égaux à un instant donné le sont aussi à n'importe quel instant futur.
Cette stabilité évite les comportements surprenants pour l'utilisateur et, surtout, permet aux méthodes `hashCode` et `compareTo` – examinées plus loin – d'être stables à leur tour. La stabilité de ces dernières est fondamentale à la préservation des invariants des tables de hachage et des arbres de recherche !
Le seul moyen de garantir qu'une mise en œuvre de `equals` soit stable est qu'elle ne se base que sur des attributs immuables de la classe.

Règle de `equals`

Toute redéfinition de `equals` ne doit se baser que sur des attributs immuables de la classe.

Corollaire : une classe dénuée d'attributs immuables ne doit jamais redéfinir `equals`. Cela garantit que ses instances sont comparées par identité, la seule caractéristique immuable d'une telle classe !
Notez que beaucoup de classes de l'API Java violent cette règle, à commencer par toutes les collections.

Hachage

Hachage

Rappel : on désigne par hachage toute technique permettant de transformer des données quelconques en une valeur entière comprise dans un intervalle donné, au moyen d'une fonction de hachage.
Le hachage est une technique fondamentale en informatique qui rencontre de nombreuses applications, p.ex. en cryptographie.
Ici, nous n'examinerons que son utilisation dans le cadre des tables de hachage.

Propriété du hachage

Une fonction de hachage est une fonction mathématique, au sens où lorsqu'on l'applique à deux valeurs égales, elle produit la même valeur de hachage.
En d'autres termes, toute fonction de hachage h satisfait la propriété suivante :

$$\forall x, y : x = y \Rightarrow h(x) = h(y)$$

L'implication inverse n'est vraie que pour le cas – très rare en pratique – des fonctions de hachage parfaites.

Hachage parfait

Une fonction de hachage h est dite **parfaite** si elle fait correspondre une valeur de hachage différente à chaque donnée à laquelle on peut l'appliquer, c-à-d si :

$\forall x, y : x \neq y \Rightarrow h(x) \neq h(y)$ ou encore $h(x) = h(y) \Rightarrow x = y$

En termes mathématiques, une fonction de hachage est parfaite ssi elle est injective.

En pratique, cette condition n'est que rarement satisfaite. Selon le principe des tiroirs (*pigeonhole principle*), elle ne peut d'ailleurs pas l'être si le nombre de données possibles est plus grand que le nombre de valeurs de hachage possible, ce qui est fréquemment le cas.

Collisions de hachage

Les fonctions de hachage ne pouvant généralement pas être parfaites, elle doivent être conçues pour distribuer aussi bien que possible les valeurs rencontrées en pratique.
Cette condition est malheureusement très difficile à spécifier et à garantir, la définition de bonnes fonctions de hachage étant plus un art qu'une science.
Lorsque deux données distinctes possèdent la même valeur de hachage, c-à-d lorsque $x \neq y$ mais $h(x) = h(y)$, on dit qu'il y a **collision de hachage**.

Méthode hashCode

Les concepteurs de Java ont fait le choix – discutable – de fournir dans la classe `Object` la méthode `hashCode` qui retourne, sous la forme d'un entier `int`, une valeur de hachage de l'objet auquel on l'applique.
La mise en œuvre par défaut de cette méthode – héritée par toute classe qui ne la redéfinit pas – retourne une valeur qui dépend de l'identité de l'objet. Par défaut, deux objets distincts ont donc généralement une valeur de hachage distincte. (Mais attention : cela n'est pas garanti !)

Compatibilité avec `equals`

Il est absolument fondamental que deux objets considérés comme égaux par `equals` aient la même valeur de hachage d'après `hashCode`, faute de quoi `hashCode` ne définit pas une fonction de hachage au sens mathématique. En d'autres termes, la propriété suivante doit être satisfaite pour toute paire d'objets o_1 et o_2 :

o_1`.equals(`o_2`)` $\Rightarrow$ o_1`.hashCode() == `o_2`.hashCode()`

Les méthodes `hashCode` et `equals` d'une classe donnée sont dites **compatibles** si cette condition est satisfaite.

Règle hashCode/equals

Si vous redéfinissez `hashCode` dans une classe, redéfinissez également `equals` – et inversement – afin que ces deux méthodes restent compatibles.

Notez que cette règle, combinée à la règle de `equals`, implique que `equals` et `hashCode` ne doivent être redéfinies qu'ensemble, et ne se baser que sur des attributs immuables.

Redéfinition de hashCode

L'écriture de fonctions de hachage de qualité étant très difficile, il est préférable de laisser cette tâche à des spécialistes.
Heureusement, depuis peu la bibliothèque Java offre dans la classe `Objects` une méthode statique permettant de calculer une valeur de hachage pour une combinaison arbitraire d'objets :

```
int hash(Object... values)
```

Règle hashCode

Lorsque vous redéfinissez hashCode, utilisez la méthode statique hash de la classe `Objects` pour la mettre en œuvre, en lui passant tous les attributs à hacher.

Par exemple, pour une classe `Person` immuable dont les attributs seraient `firstName`, `lastName` et `birthDate`, la méthode hashCode pourrait être redéfinie ainsi :

```
public int hashCode() {
  return Objects.hash(
    firstName, lastName, birthDate);
}
```

Ordre

Ordre en Java

Contrairement au hachage qui est disponible pour tout objet via la méthode `hashCode` de `Object`, la possibilité d'ordonner les valeurs d'un type donné n'est pas prédéfini en Java.
Au lieu de cela, deux interfaces sont fournies pour ordonner des valeurs d'un type donné. L'une permet aux valeurs de se comparer elle-mêmes, tandis que l'autre permet à un objet externe de comparer deux valeurs.

Ordre en Java

La bibliothèque Java offre deux manières d'exprimer un ordre entre les instances d'une classe *C* donnée :

1. la classe *C* elle-même peut implémenter l'interface `Comparable<C>`, signalant ainsi qu'elle possède la méthode `compareTo` permettant d'ordonner deux de ses propres instances,
2. une classe indépendante de *C* peut implémenter l'interface `Comparator<C>`, signalant ainsi qu'elle possède la méthode `compare` permettant de comparer deux objets de type *C*.

Les interfaces `Comparable` et `Comparator`, qu'il est important de ne pas confondre, sont présentées ci-après.

L'interface Comparable

L'interface `Comparable` – du paquetage `java.lang` – peut être implémentée par toute classe dont les instances sont comparables entre-elles.

Cette interface contient une seule méthode, qui compare l'objet auquel on l'applique (le récepteur) à un objet d'un type donné par le paramètre de type de l'interface :

```
public interface Comparable<T> {
  int compareTo(T that);
}
```

La méthode `compareTo` retourne un entier négatif si l'objet auquel on l'applique est inférieur à l'argument, nul si les deux sont égaux, et positif dans les autres cas.

Paramètre de `Comparable`

Le paramètre de type de `Comparable` représente le type des objets avec lesquels le récepteur – l'objet auquel on applique la méthode, c-à-d `this` – est comparable. Dès lors, les classes qui implémentent `Comparable` passent leur propre type en argument !

Par exemple, la classe `Integer` est définie ainsi en Java :

```
class Integer implements Comparable<Integer>
```

Aussi surprenante que cette déclaration puisse paraître, elle est tout à fait logique : elle signifie qu'un objet de type `Integer` n'est comparable qu'avec un autre objet de type `Integer`.

Exemple

Beaucoup de classes de la bibliothèque Java implémentent l'interface `Comparable`. Il en va par exemple ainsi de la classe `String` dont les instances sont ordonnées selon l'ordre lexicographique (du dictionnaire), et donc :

- `"le".compareTo("la")` retourne un entier positif,
- `"le".compareTo("le")` retourne zéro, et
- `"mont".compareTo("montagne")` retourne un entier négatif.

Règle de `Comparable`

Lorsque vous définissez une classe qui implémente l'interface `Comparable`, assurez-vous que sa méthode `compareTo` soit compatible avec sa méthode `equals`.

Les méthode `compareTo` et `equals` d'une classe donnée sont compatibles ssi elles considèrent exactement les mêmes couples d'objets comme étant égaux, donc si, pour toutes paires d'instances o_1 et o_2 de cette classe, on a :

o_1`.equals(`o_2`)` $\Leftrightarrow$ o_1`.compareTo(`o_2`) == 0`

Ordre naturel

Il est évident qu'une classe donnée ne peut avoir qu'une seule version de la méthode `compareTo`. L'ordre qui est ainsi « attaché » à une classe via sa méthode `compareTo` est appelé **l'ordre naturel** de cette classe en Java.
Par exemple, l'ordre lexicographique a été choisi par les concepteurs de la classe `String` comme étant l'ordre naturel des chaînes de caractères.
Parfois, plusieurs notions d'ordres peuvent être utiles pour un même type, et il est donc important d'avoir un moyen d'ordonner des valeurs qui soit externe à leur classe.
C'est le but de la notion de comparateur.

L'interface Comparator

L'interface `Comparator` – du paquetage `java.util` – décrit un comparateur, c-à-d un objet capable de comparer deux autres objets d'un type donné. Le paramètre de type de cette interface spécifie le type des objets que le comparateur sait comparer.

```
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

La méthode `compare` doit retourner un entier négatif si le premier objet est inférieur au second, nul si les deux sont égaux et positif dans les autres cas.

`Comparable/Comparator`

Attention : les interfaces `Comparable` et `Comparator` ont un rôle similaire mais sont néanmoins différentes.
La méthode `compareTo` de `Comparable` prend un seul argument, et compare le récepteur (`this`) avec cet argument.
Par contre, la méthode compare de `Comparator` prend *deux* arguments, et les compare entre eux.

Tri

La différence entre `Comparable` et `Comparator` – et leur utilité respective – peut s'illustrer via les variantes de la méthode de tri de liste de la classe `Collections`.

La première variante ne prend qu'un seul argument – la liste à trier – et la trie selon l'ordre naturel de ses éléments, qui doivent donc en posséder un (voir page suivante) :

```
<T> void sort(List<T> l)
```

La seconde variante prend deux arguments – la liste à trier et un comparateur – et la trie selon l'ordre du comparateur :

```
<T> void sort(List<T> l, Comparator<T> c)
```

Cette variante est utilisable que les éléments aient un ordre naturel ou pas, car seul le comparateur est utilisé !

Borne de type

Le type de la première variante de la méthode `sort` est en réalité plus complexe qu'indiqué précédemment. En effet, cette méthode ne doit être utilisable que si le type `T` implémente `Comparable<T>`. Comment exprimer cette contrainte ?

En Java, cela peut se faire en plaçant une borne (supérieure) sur le paramètre de type `T` le forçant à implémenter `Comparable<T>`, au moyen de la syntaxe suivante :

```
<T extends Comparable<T>> void sort(List<T> l)
```

De manière informelle, cette contrainte peut se lire : « cette méthode n'est utilisable que si le type des éléments de la liste possède un ordre naturel. »