Collections — la bibliothèque de conteneurs

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-31

Table des matières

1 Position de la leçon et objectifs 1
2 1. Limites du tableau comme structure de données 2
3 2. La hiérarchie des contrats (List, Set, Map, Queue) 3
4 3. Les listes : ArrayList contre LinkedList 6
5 4. Les ensembles : hachage (HashSet), ordre (TreeSet) 6
6 5. Le contrat equals/hashCode 8
7 6. Les Map : clés, valeurs, Entry, parcours 9
8 7. Parcourir sans savoir : Iterator, Iterable, for-each 12
9 8. Critères de choix et convention de retour 13
10 9. Hors chemin noté : enrichissements 15
10.1 Ressources 15
10.2 Les classes abstraites du framework 15
10.3 Les classes utilitaires Collections et Arrays 15
10.4 La conversion vers un tableau 15
10.5 Recherche binaire 16
10.6 EnumSet et EnumMap 17
11 10. À retenir 18
12 Annexes 18
12.1 Signatures de l'interface Map (dump de référence) 18
12.2 Eclipse Collections (aperçu) 19

1 Position de la leçon et objectifs

Cette leçon enseigne l'usage des structures de conteneurs du JDK : la hiérarchie des contrats `java.util`, les implémentations qui les satisfont, les coûts associés à chaque famille et le protocole de parcours. Elle suppose acquis le mécanisme des types paramétrés établi dans `06a_java_L_genericite` : les signatures étudiées ici (`List<E>`, `Set<E>`, `Map<K,V>`) sont des types génériques ; les déclarations brutes et les transtypages analysés en §1 et §7 de cette leçon préalable ne seront

pas repris. Les exemples s'exécutent dans l'image du cours (Java 25, annexe `90_prerequis_environnement`).

Objectifs contractuels — à la fin de la leçon, l'étudiant sait :

1. **(COL-O1)** Énoncer trois limites du tableau qui motivent la bibliothèque (taille figée, absence de garantie d'unicité, absence d'accès par clé).
2. **(COL-O2)** Situer `List`, `Set`, `Map` et `Queue` dans la hiérarchie `java.util` et nommer la question que chacune tranche.
3. **(COL-O3)** Choisir une implémentation (`ArrayList`/`LinkedList`, `HashSet`/`TreeSet`, `HashMap`/`TreeMap`) en citant les complexités sourcées de la javadoc et la garantie échangée contre elles.
4. **(COL-O4)** Énoncer le contrat `equals`/`hashCode` de la javadoc `Object` et prédire le comportement d'une clé mutée après insertion (démonstration exécutée en §5).
5. **(COL-O5)** Parcourir une `Map` par ses vues et par `entrySet()` ; distinguer `put` (écrasement, ancienne valeur rendue) et `replace`.
6. **(COL-O6)** Expliquer le mécanisme fail-fast d'un `Iterator` (`modCount`) et reproduire la `ConcurrentModificationException` (§7).

2 1. Limites du tableau comme structure de données

Un tableau fixe sa taille à la création, s'adresse par indice entier et ne porte aucune garantie sur le contenu de ses cases : ni unicité des éléments, ni ordre de parcours autre que l'ordre des indices, ni accès par une clé arbitraire. Ces trois limites ne sont pas des défauts de fabrication ; elles délimitent ce que la structure promet, et donc ce qu'il faut chercher ailleurs quand le besoin dépasse la promesse.

Quatre familles de structures répondent à ces besoins : le **tableau** (accès indexé, taille figée), la **liste** (séquence ordonnée, insertion et suppression souples), l'**arbre** (hiérarchie ordonnée, recherche guidée par la comparaison) et la **table de hachage** (affectation clé→valeur, recherche par signature de clé). La bibliothèque `java.util` — le collections framework — fournit pour chaque famille des **interfaces** qui définissent le contrat (`List`, `Set`, `Queue`, `Map`), des **classes abstraites** qui en implémentent les parties communes, et des **classes concrètes** qui en précisent le rangement (`ArrayList`, `HashSet`, `TreeSet`, `HashMap`).

Le point de basculement pour le développeur est celui-ci : stocker des groupes d'objets est un problème résolu. Chaque structure — liste chaînée, table de hachage, arbre équilibré — existe, corrigée, mesurée, et régie par un contrat écrit dans la javadoc. Le travail d'implémentation cède la place à un travail de **choix** : sélectionner le contrat, puis l'implémentation, et inscrire ce choix dans le type déclaré — le plus étroit qui convienne à l'usage (§8). Les compétences mobilisées ici — souscrire à un contrat et reconnaître une hiérarchie de types — sont celles des séquences bases (boucle améliorée, sec-04) et héritage (réutilisation par contrat, sec-05).

3 2. La hiérarchie des contrats (List, Set, Map, Queue)

Un conteneur unique promettant à la fois l'ordre, l'unicité, l'accès par clé et l'insertion en tête ne pourrait refuser aucune de ces propriétés ; or une propriété non promise est ce qui permet de l'optimiser. La hiérarchie découpe donc le problème en questions indépendantes, et chaque interface de premier rang y répond par un contrat :

- **Dans quel ordre les éléments sont-ils rendus ?** `List` promet l'ordre d'insertion et l'accès par indice ; aucune autre interface ne le promet.
- **Un élément peut-il apparaître deux fois ?** `Set` l'exclut — cette exclusion est une garantie contractuelle, non une limitation accidentelle.
- **Peut-on retrouver un élément par une clé ?** `Map` le permet ; elle n'appartient pas à la hiérarchie `Collection`, car elle ne contient pas des éléments mais associe des clés à des valeurs (§6).
- **La structure modélise-t-elle une file d'attente ?** `Queue` ajoute un contrat d'entrée par l'arrière et de sortie par l'avant (`offer`, `poll`, `peek`), sans accès par indice ; elle est nommée ici pour fermer l'énumération des interfaces de premier rang et n'est pas développée dans le chemin noté.

> ℹ Définition — Collection
>
> `Collection` est le contrat commun de ce qui énumère : `size()`, `add(e)`, `remove(o)`, et surtout `iterator()` — un parcours indépendant du rangement intérieur. (Source : javadoc de `java.util.Collection`.) Les types `List<E>`, `Set<E>` et `Queue<E>` utilisés dans toute la leçon sont des types paramétrés : l'argument de type `E` a été établi dans `06a_java_L_genericite` (pivot de sa §6) ; une collection déclarée sans argument de type est un type **brut**, et les transtypages et erreurs différés que ce choix provoque y sont démontrés (sa §1) — ils ne seront pas repris ici.

Sur une collection, la boucle améliorée `for (T x : coll)` est l'écriture habituelle du parcours par `iterator()` ; sur un tableau, la même écriture produit un autre mécanisme — la §7 y revient.

Premier contact : deux conteneurs, le même contenu, deux comportements (`List.of` : fabrique de liste immuable, étudiée plus loin dans la séquence Après 8 ; le motif `new ArrayList<>(List.of(...))` employé plus bas la copie dans une `ArrayList` pour pouvoir la modifier).

```
import java.util.*;
List<String> liste = new ArrayList<>();
Set<String> ensemble = new HashSet<>();
for (String s : List.of("pommier", "poire", "pommier")) {
    liste.add(s);
```

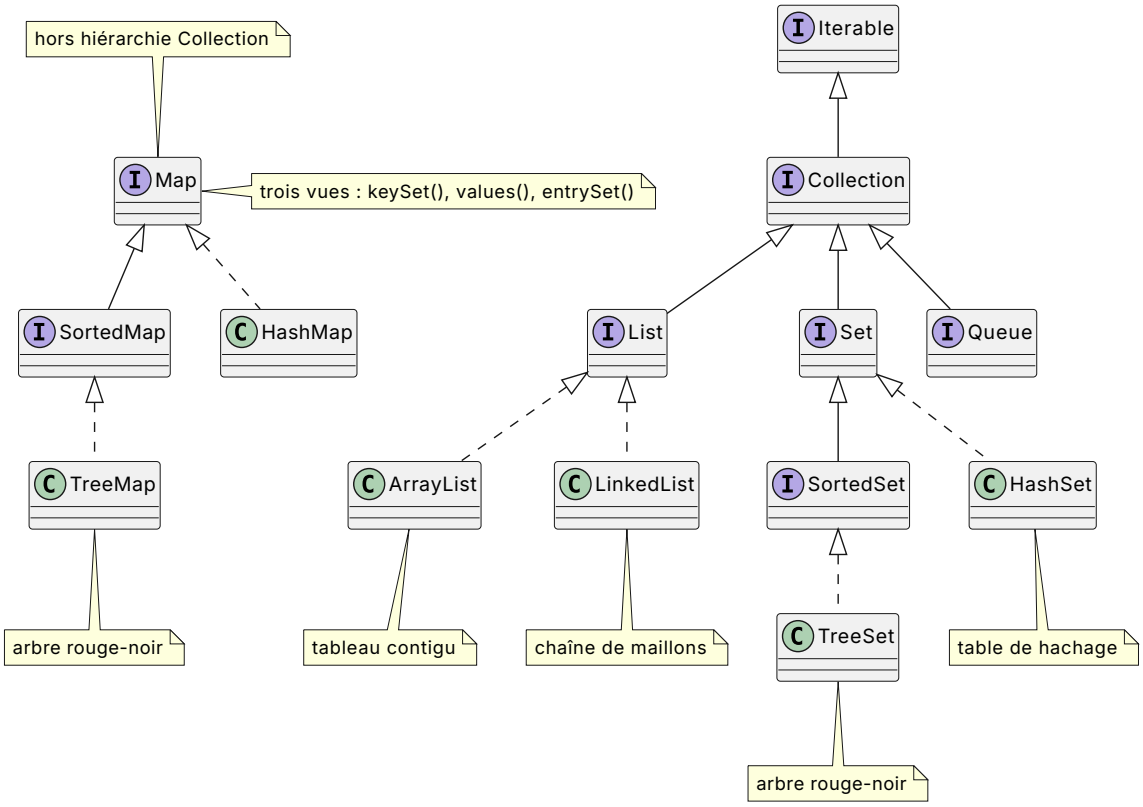
```
    ensemble.add(s);
}
System.out.println("liste    : " + liste);
System.out.println("ensemble : " + ensemble);
```

```
liste    : [pommier, poire, pommier]
ensemble : [pommier, poire]
```

La liste a conservé le doublon et l'ordre ; l'ensemble a supprimé le doublon et ne promet aucun ordre de parcours — celui-ci dépend du rangement intérieur, non de la séquence d'insertion. La figure ci-dessous donne la version unique de la hiérarchie étudiée (les implémentations de `Queue` et les classes abstraites en sont retirées pour la lisibilité) :

```
%%plantUML
@startuml
skinparam classAttributeIconSize 0
interface Iterable
interface Collection
interface List
interface Set
interface SortedSet
interface Queue
interface Map
interface SortedMap
class ArrayList
class LinkedList
class HashSet
class TreeSet
class HashMap
class TreeMap
Iterable <|-- Collection
Collection <|-- List
Collection <|-- Set
Collection <|-- Queue
Set <|-- SortedSet
Map <|-- SortedMap
List <|.. ArrayList
List <|.. LinkedList
Set <|.. HashSet
SortedSet <|.. TreeSet
Map <|.. HashMap
SortedMap <|.. TreeMap
```

```
note top of Map : hors hiérarchie Collection
note right of Map : trois vues : keySet(), values(), entrySet()
note bottom of ArrayList : tableau contigu
note bottom of LinkedList : chaîne de maillons
note bottom of HashSet : table de hachage
note bottom of TreeSet : arbre rouge-noir
note bottom of TreeMap : arbre rouge-noir
@enduml
```



Choisir le contrat n'est pas choisir le conteneur : derrière chaque interface de premier rang se tiennent plusieurs implémentations qui satisfont toutes le contrat — et qui diffèrent par le coût. Les sections 3 à 6 ferment la liste des paires étudiées : `ArrayList`/`LinkedList` sous `List`, `HashSet`/`TreeSet` sous `Set`, `HashMap`/`TreeMap` sous `Map`.

4 3. Les listes : ArrayList contre LinkedList

ℹ Propriété — coûts des deux implémentations de List

- `ArrayList` range les éléments dans un tableau contigu : l'accès par indice et le parcours complet s'exécutent en temps constant par élément ; `add(e)` en fin de liste s'exécute en temps constant amorti ; les insertions et suppressions en position déplacent les éléments suivants et coûtent un temps linéaire en le nombre d'éléments déplacés. (Source : javadoc Java SE 25, `java.util.ArrayList` — accès par indice en temps constant ; `add` et `addLast` amortis ; `getFirst`, `getLast`, `removeLast`, `reversed` en temps constant.)
- `LinkedList` range les éléments en maillons reliés : l'insertion et la suppression en tête, ou à une position déjà atteinte, s'exécutent en temps constant ; mais les opérations adressées par indice parcourent la chaîne depuis la tête ou la queue et coûtent un temps proportionnel à la distance parcourue (Source : javadoc Java SE 25, `java.util.LinkedList` — les opérations adressées par indice « parcourent depuis le début ou la fin, selon ce qui est le plus proche » ; coût proportionnel à la distance).

L'échange est net et symétrique : `ArrayList` favorise la lecture adressée et pénalise l'insertion en position ; `LinkedList` favorise l'insertion à position connue et pénalise la lecture par indice. Le contrat `List` ne tranche pas à la place du programmeur : il permet d'écrire le code sans trancher, et rend le choix lisible dans la déclaration.

5 4. Les ensembles : hachage (HashSet), ordre (TreeSet)

Sous `Set`, la garantie d'unicité s'obtient de deux façons différentes.

ℹ Définition — HashSet et TreeSet

- `HashSet` décide de l'unicité **par hachage** : chaque objet est placé dans une case (`bucket`) dont l'index est dérivé de `hashCode()` ; deux objets égaux au sens de `equals()` doivent être placés dans la même case. La garantie porte donc aussi sur les objets stockés (contrat du §5). En contrepartie, `HashSet` ne promet aucun ordre de parcours. (Source : javadoc de `java.util.HashSet` et de `java.lang.Object.hashCode()`.)
- `TreeSet` décide de l'unicité **par ordre** : les éléments sont maintenus dans un arbre binaire équilibré de type rouge-noir ; le parcours suit l'ordre trié, et `contains`, `add` et `remove` s'exécutent en temps logarithmique. Condition d'emploi : les éléments doivent être comparables d'eux-mêmes (`Comparable<T>`, bornage étudié en §3 de `06a_java_L_genericite`) ou par un `Comparator` fourni au constructeur (§8). (Source : javadoc Java SE 25, `java.util.TreeSet` — `add`, `remove`, `contains` garantis « log(n) » ; arbre rouge-noir : la page `java.util.TreeMap`, sur laquelle `TreeSet` repose.)

```
Set<String> arbre = new TreeSet<>();
arbre.add("Banane");
arbre.add("Pomme");
arbre.add("Orange");
System.out.println("TreeSet : " + arbre);
System.out.println("Second add de \"Pomme\" : " + arbre.add("Pomme"));
```

```
TreeSet : [Banane, Orange, Pomme]
Second add de "Pomme" : false
```

```
Set<String> hache = new HashSet<>();
hache.add("Banane");
hache.add("Pomme");
hache.add("Orange");
System.out.println("HashSet : " + hache);
System.out.println("Second add de \"Pomme\" : " + hache.add("Pomme"));
```

```
HashSet : [Banane, Pomme, Orange]
Second add de "Pomme" : false
```

Le `TreeSet` rend `[Banane, Orange, Pomme]` — l'ordre naturel alphabétique. L'ordre affiché par le `HashSet` n'est pas spécifié par le contrat et ne doit pas être prédit. En revanche,

le second ajout de `"Pomme"` rend `false` dans les deux implémentations : c'est la garantie d'unicité du contrat `Set`, vérifiée par `equals` — et par `hashCode` sous `HashSet`.

Deux relations d'ordre et une relation d'égalité interviennent dans ces structures ; il faut les distinguer formellement :

> ℹ Propriété — equals, compareTo, compare
>
> - `equals(Object)` : relation d'équivalence entre objets, régie par le contrat de `java.lang.Object` (§5). Elle gouverne l'unicité dans les deux familles de `Set`.
> - `compareTo(T)` (interface `Comparable<T>`) : ordre naturel d'une classe sur elle-même ; `TreeSet` l'utilise pour se ranger. La javadoc de `Comparable` recommande fortement la cohérence avec `equals` : `(x.compareTo(y) == 0) == x.equals(y)`.
> - `compare(T, T)` (interface `Comparator<T>`) : ordre externe à la classe, fourni au conteneur (§8). (Source : javadoc Java SE 25, `java.lang.Comparable` — cohérence avec `equals` « fortement recommandée » ; `java.util.Comparator` — cohérence « non strictement requise », formule plus faible ; contrat d'équivalence : `java.lang.Object`.)

6 5. Le contrat equals/hashCode

> ℹ Définition — contrat Object.hashCode (javadoc)
>
> La méthode `hashCode` doit respecter trois clauses (javadoc de `java.lang.Object.hashCode()`) :
>
> 1. **Stabilité** : appelé plusieurs fois sur le même objet au cours d'une même exécution, sans modification des champs utilisés par `equals`, elle doit retourner la même valeur.
> 2. **Parité avec equals** : si `a.equals(b)`, alors `a.hashCode() == b.hashCode()`.
> 3. **Pas de réciproque** : deux objets non égaux peuvent avoir le même code ; les collisions sont légales, mais coûtent en performances.
>
> Bonnes pratiques correspondantes (javadoc, documentation de référence) : redéfinir les deux méthodes ensemble ; inclure exactement les mêmes champs dans les deux ; calculer le code par `java.util.Objects.hash(...)` plutôt qu'à la main.

Ce contrat n'est pas une formalité : `HashSet` et `HashMap` s'en servent comme condition de fonctionnement (§4, §6). Une clé qui le viole devient introuvable dans la table. Démonstration, exécutée :

```java
class CleMutable {
    String valeur;
    CleMutable(String v) { this.valeur = v; }
    @Override public int hashCode() { return Objects.hash(valeur); }
    @Override public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() ≠ o.getClass()) return false;
        return valeur.equals(((CleMutable) o).valeur);
    }
}
```

```java
Map<CleMutable, String> table = new HashMap<>();
CleMutable cle = new CleMutable("alpha");
table.put(cle, "présente");
System.out.println("Avant mutation : " + table.get(cle));
cle.valeur = "beta";              // mutation APRÈS insertion
System.out.println("Après mutation : " + table.get(cle));
System.out.println("Éléments dans la table : " + table.size());
```

```
Avant mutation : présente
Après mutation : null
Éléments dans la table : 1
```

La mutation après insertion change le `hashCode` de l'objet : la recherche adresse une autre case, n'y trouve rien et rend `null` — alors que la table contient toujours l'entrée (`size()` vaut 1), devenue inatteignable par cette clé. Par conséquent : une clé doit être **immuable** (ou du moins voir ses champs de hachage non modifiés) pendant toute sa durée de vie dans une table de hachage. Limite du cas traité : la violation la plus fréquente reste l'absence de redéfinition conjointe (`equals` redéfini sans `hashCode`) ; le correctif est la parité des deux méthodes sur les mêmes champs, comme dans la classe ci-dessus.

7 6. Les Map : clés, valeurs, Entry, parcours

Une `Map<K,V>` associe des clés à des valeurs : chaque clé apparaît au plus une fois (deux clés distinctes ne sont jamais `equals`). Elle n'étend pas `Collection` : elle n'énumère pas des éléments, elle indexe par clé. Deux implémentations ferment la liste des `Map` étudiées :

> **Propriété — coûts de HashMap et TreeMap**
>
> - `HashMap` (table de hachage) : les opérations de base `get`, `put`, `remove` et `containsKey` s'exécutent en temps constant **en moyenne**, sous l'hypothèse que la fonction de hachage disperse suffisamment les clés. (Source : javadoc Java SE 25, `java.util.HashMap` — « get » et « put » en temps constant, « assuming the hash function disperses the elements properly ».)
> - `TreeMap` (arbre rouge-noir, implémentation de `SortedMap`) : mêmes opérations en temps logarithmique, avec un parcours dans l'ordre des clés. (Source : javadoc Java SE 25, `java.util.TreeMap` — `containsKey`, `get`, `put`, `remove` garantis en temps « log(n) ».) Condition d'emploi : clés `Comparable`, ou `Comparator` fourni au constructeur.

Le contrat des clés est celui du §5 : `HashMap` exige la parité `equals`/`hashCode` ; `TreeMap` exige en outre un ordre total entre les clés. Les types `Map<K,V>` et `TreeMap<String,Integer>` rencontrés ici sont des types paramétrés au sens de `06a_java_L_genericite` : `K` et `V` sont ses paramètres de type, et l'invariance (`Map<String,Integer>` n'est pas un `Map<Object,Object>`) y est établie (sa §7).

Comportement de base, avec la gestion de l'absence :

```
Map<String, Integer> ages = new HashMap<>();
ages.put("René", 34);
ages.put("Léa", 29);
System.out.println(ages.get("René"));
System.out.println(ages.get("Marcel"));
```

```
34
null
```

`get` sur une clé absente ne lève pas d'exception : il rend `null`. Trancher entre « absence normale » et « absence erreur » relève de l'appelant ; la `Map` refuse de le faire à sa place.

> **Définition — Map.Entry et les vues**
>
> Les couples clé→valeur sont manipulés via l'interface imbriquée `Map.Entry<K,V>` : `getKey()`, `getValue()`, et `setValue(V)` — cette dernière écrit la valeur dans la `Map` d'origine. La méthode `entrySet()` retourne la vue `Set<Map.Entry<K,V>>` des couples. Les trois vues `keySet()`, `values()`, `entrySet()` sont des collections **connectées** à la `Map` : elles reflètent ses modifications et y participent. (Source : javadoc de `java.util.Map` et de `java.util.Map.Entry`.)

Écrasement et remplacement sont deux comportements distincts, et `put` rend quelque chose :

```
Integer precedente = ages.put("Léa", 30);   // clé déjà présente
System.out.println("Valeur rendue par put : " + precedente);
System.out.println("replace sur clé absente : " + ages.replace("Marcel", 40));
System.out.println("Table après opérations : " + ages);
```

```
Valeur rendue par put : 29
replace sur clé absente : null
Table après opérations : {Léa=30, René=34}
```

`put` sur une clé existante écrase la valeur et **rend l'ancienne** : `precedente` vaut `29`, la table conserve une seule entrée pour `"Léa"`, désormais à `30`. `replace` sur une clé absente ne crée rien et rend `null`. La distinction `put`/`replace` porte donc précisément sur le cas limite « clé absente ».

Parcours effectif par la vue `entrySet()` — sur un `TreeMap`, pour que l'ordre affiché soit déterministe :

```
Map<String, Integer> notes = new TreeMap<>();
notes.put("Alice", 12);
notes.put("Bruno", 17);
notes.put("Chloé", 15);
for (Map.Entry<String, Integer> e : notes.entrySet()) {
    System.out.println(e.getKey() + " → " + e.getValue());
}
for (Map.Entry<String, Integer> e : notes.entrySet()) {
    if (e.getValue() < 14) {
        e.setValue(14);          // écriture via la vue
    }
}
System.out.println("Après relèvement : " + notes);
```

```
Alice → 12
Bruno → 17
Chloé → 15
Après relèvement : {Alice=14, Bruno=17, Chloé=15}
```

La boucle sur `entrySet()` effectue le parcours complet en une seule traversée ; parcourir `keySet()` puis rappeler `get(k)` pour chaque clé est plus coûteux sur une `HashMap` (une recherche par hachage de plus par clé). `setValue` sur l'entrée modifie la `Map` sous-jacente : la vue est connectée, non copiée.

8 7. Parcourir sans savoir : Iterator, Iterable, for-each

Le parcours n'est pas une opération propre à chaque conteneur : c'est un protocole à part entière.

> ℹ Définition — Iterator et Iterable
>
> - `Iterator<E>` répond à deux questions de lecture — `hasNext()` : reste-t-il un élément ? `next()` : le fournir et avancer — et à une opération de retrait, `remove()`, qui supprime l'élément rendu par le dernier `next()`.
> - `Iterable<E>` est l'interface d'un objet qui sait fournir un `Iterator` par `iterator()`. `Collection` étend `Iterable` : toute collection sait se parcourir quel que soit son rangement intérieur. (Source : javadoc de `java.util.Iterator` et de `java.lang.Iterable`.)

La boucle améliorée n'est pas un mécanisme distinct : elle admet deux traductions selon le type de l'expression itérable — parcours par `iterator()`, `hasNext()` et `next()` pour un `Iterable` ; parcours indexé par compteur pour un tableau (Java Language Specification, §14.14.2, Enhanced for Statement). Les deux formes, explicite et concise, parcourent la même séquence :

```java
import java.util.*;
List<String> files = new ArrayList<>(List.of("chêne", "hêtre"));
for (Iterator<String> it = files.iterator(); it.hasNext(); )
    System.out.println(it.next());
for (String f : files)
    System.out.println(f);
```

```
chêne
hêtre
chêne
hêtre
```

Le contrat promet la lecture, pas la cohérence face à une modification extérieure. Le mécanisme fail-fast, documenté dans la javadoc d'`Iterator`, est le suivant : la collection sous-jacente tient un compteur structurel (`modCount`), incrémenté à toute modification structurelle hors du chemin de l'itérateur ; l'itérateur mémorise la valeur de ce compteur à sa création et la revérifie à chaque `next()` — en cas d'écart, il lève `ConcurrentModificationException`. Démonstration, exécutée :

```
List<String> motifs = new ArrayList<>(List.of("un", "deux", "trois"));
for (String m : motifs) {
    motifs.remove(m);         // modification par un autre chemin que it.remove()
}
```

```
EvalException: null
------------------------------------------------------------------------
java.util.ConcurrentModificationException: null
                                                                        at
java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1096)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:1050)
    at .(#74:1)
```

Rejet attendu : `java.util.ConcurrentModificationException`. Le chemin légitime de retrait pendant un parcours est `remove()` sur l'itérateur lui-même — il met le compteur à jour ; le TP-C le met en œuvre. Le mécanisme protège contre un état incohérent ; ce n'est pas une synchronisation : un parcours réellement concurrent entre threads relève d'un autre outillage, hors du chemin noté.

9 8. Critères de choix et convention de retour

Trier une collection suppose un ordre : `Collections.sort(liste)` exige des éléments `Comparable` ; la surcharge `Collections.sort(liste, comparateur)` accepte un ordre externe.

> ℹ Définition — Comparator
>
> `Comparator<T>` est un **ordre externe** à la classe `T` : une méthode `int compare(T a, T b)` qui rend un entier négatif, nul ou positif selon que `a` précède, égale ou suit `b`. Elle permet plusieurs ordres sur une même classe sans la modifier ; un `TreeSet` ou un `TreeMap` construit avec un comparateur range ses éléments selon cet ordre plutôt que selon l'ordre naturel. Un comparateur devrait être cohérent avec `equals` : `(compare(x, y) == 0) == x.equals(y)` (javadoc de `java.util.Comparator`). La borne `Comparator<? super T>` des API `java.util` relève des wildcards (06a §4). La mise en œuvre de comparateurs sur des types du métier est exercée dans le TP-C.

```java
List<String> arbres = new ArrayList<>(List.of("sapin", "if", "chêne"));
Collections.sort(arbres, new Comparator<String>() {
    @Override public int compare(String a, String b) {
        return Integer.compare(a.length(), b.length());
    }
});
System.out.println("Trié par longueur : " + arbres);
```

```
Trié par longueur : [if, sapin, chêne]
```

Les algorithmes de tri eux-mêmes et l'écriture abrégée des comparateurs par expressions lambda appartiennent à la séquence Après 8 ; seule la définition du contrat est établie ici, car `TreeSet` et `TreeMap` la conditionnent.

! Convention — Convention de retour

Critère de choix des types déclarés (documentation de référence Java, The Collections Framework ; application du principe de réutilisation par contrat de la séquence héritage) :

1. **Paramètres** : déclarer le type aussi général que possible — une interface (`List`, `Set`, `Map`) — pour accepter toute implémentation.
2. **Retours** : la règle classique recommande la classe la plus spécifique (elle offre le plus de méthodes à l'appelant), **mais** elle lie l'appelant à cette implémentation : un changement d'implémentation casse alors les appels. La convention retenue dans ce cours est donc l'interface comme type de retour, sauf nécessité documentée d'une méthode propre à l'implémentation.
3. **Partage sans mutation** : pour exposer une collection interne sans céder sa modification, retourner une vue en lecture seule de la classe `Collections` (`unmodifiableList`, `unmodifiableSet`, `unmodifiableMap`) plutôt que la collection elle-même (javadoc de `java.util.Collections`).

Conséquence contractuelle de la clause 3 : les collections en lecture seule (`Collections.unmodifiableXxx`), à taille fixe (`Arrays.asList`) ou immuables (`List.of` — séquence Après 8) refusent les opérations de modification. L'appel de `add`, `remove` ou `set` sur une telle vue lève `UnsupportedOperationException` : le mécanisme est celui d'opérations optionnelles que l'implémentation choisit de ne pas prendre en charge — la javadoc de `Collection.remove` l'énonce expressément (« Optional operations »). Qui définit, déclenche et traite cette exception est enseignée dans la séquence consacrée aux exceptions (sec-07) ; il n'y est pas répété ici.

10 9. Hors chemin noté : enrichissements

10.1 Ressources

Le tutoriel officiel (The Collections Framework, docs.oracle.com) et la javadoc du paquetage `java.util` prolongent la leçon. Liens : javadoc `java.util` (Java SE 25) `https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/package-summary.html` ; tutoriel `https://docs.oracle.com/javase/tutorial/collections/index.html` — documentation legacy JDK 8 (le tutoriel en ligne n'a pas été réédité pour les versions récentes).

10.2 Les classes abstraites du framework

`java.util` fournit, en plus des interfaces et des classes concrètes, une famille close de squelettes d'implémentation : `AbstractCollection`, `AbstractList`, `AbstractQueue`, `AbstractSequentialList`, `AbstractSet`, `AbstractMap`. Chacune implémente les parties déductibles du contrat de son interface, laissant aux classes concrètes les seules méthodes propres au rangement. Le mécanisme général — hériter d'un squelette et compléter des méthodes abstraites — est celui étudié dans la séquence héritage (sec-05) ; seule son application aux collections est nommée ici.

10.3 Les classes utilitaires Collections et Arrays

`Collections` (à ne pas confondre avec l'interface `Collection`) centralise des algorithmes polymorphes sur les collections : `sort`, `min`, `max`, `binarySearch`, ainsi que les fabriques de vues (`unmodifiableXxx`, §8). `Arrays` rend le même service aux tableaux : `sort`, `binarySearch`, `toString`, et la fabrique `asList` qui présente un tableau comme une `List` à taille fixe.

```
List<Integer> valeurs = new ArrayList<>(List.of(7, 3, 5));
System.out.println("min : " + Collections.min(valeurs)
                 + ", max : " + Collections.max(valeurs));
int[] tableau = {7, 3, 5};
Arrays.sort(tableau);
System.out.println("Tableau trié : " + Arrays.toString(tableau));
```

```
min : 3, max : 7
Tableau trié : [3, 5, 7]
```

10.4 La conversion vers un tableau

`toArray()` rend un `Object[]` : la récupération des éléments typés exige alors un transtypage par élément — le défaut des types bruts montré en §1 de `06a_java_L_genericite`. La forme typée `toArray(T[] a)` écrit les éléments dans le

tableau fourni (ou dans un nouveau tableau du même type si `a` est trop petit) ; une collection vide rend un tableau vide, jamais `null`.

```
List<String> essences = new ArrayList<>(List.of("chêne", "hêtre"));
Object[] génériques = essences.toArray();
String[] typées = essences.toArray(new String[0]);
System.out.println("Object[] : " + Arrays.toString(génériques));
System.out.println("String[] : " + Arrays.toString(typées));
```

```
Object[] : [chêne, hêtre]
String[] : [chêne, hêtre]
```

Remarque : pour construire une liste à partir d'un flux d'éléments, la voie moderne `Stream.toList` appartient à la séquence Après 8 ; `toArray` reste l'idiome lorsqu'un tableau est réellement requis (API d'interoperabilité).

10.5 Recherche binaire

`Collections.binarySearch(liste, cible)` effectue une recherche dichotomique en temps logarithmique — **à la condition que la liste soit triée dans l'ordre naturel** (ou dans l'ordre d'un comparateur passé en troisième argument). Sur liste non triée, le résultat est indéfini ; une absence est rendue comme `(-point d'insertion) - 1`.

```
List<String> noms = new ArrayList<>(List.of("sapin", "chêne", "hêtre"));
Collections.sort(noms);
System.out.println("Index de \"hêtre\"    : " + Collections.binarySearch(noms,
"hêtre"));
System.out.println("Index de \"tilleul\" : " + Collections.binarySearch(noms,
"tilleul"));
```

```
Index de "hêtre"   : 1
Index de "tilleul" : -4
```

`contains` sur la liste (parcours linéaire) suffit quand le tri n'est pas une exigence par ailleurs ; la recherche binaire ne s'emploie que là où le tri est déjà entretenu.

10.6 EnumSet et EnumMap

> ℹ Définition — EnumSet et EnumMap
>
> - `EnumSet<E extends Enum<E>>` est un `Set` dont tous les éléments appartiennent à une seule classe d'énumération, représenté comme un **vecteur de bits** : un bit par constante d'énumération. Les opérations d'ensemble (`union`, `complementOf`, `contains`) sont des opérations bit-à-bit, donc en temps constant — la javadoc d'`EnumSet` en fait son argument de performance.
> - `EnumMap<K extends Enum<K>, V>` est une `Map` dont toutes les clés appartiennent à une seule énumération, représentée par un **tableau** indexé par la position des constantes : pas de hachage, parcours dans l'ordre de déclaration. (Source : javadoc Java SE 25, `java.util.EnumSet` — « bit vectors », opérations de base « en temps constant » ; `java.util.EnumMap` — « arrays », parcours en ordre naturel de déclaration des constantes.)

```
enum Jour { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI }
EnumSet<Jour>     joursOuvres     =     EnumSet.of(Jour.LUNDI,     Jour.MERCREDI,
Jour.VENDREDI);
System.out.println("Jours ouvrés : " + joursOuvres);
System.out.println("Complément   : " + EnumSet.complementOf(joursOuvres));
```

```
Jours ouvrés : [LUNDI, MERCREDI, VENDREDI]
Complément   : [MARDI, JEUDI]
```

```
Map<Jour, String> planning = new EnumMap<>(Jour.class);
planning.put(Jour.MARDI, "réunion");
planning.put(Jour.VENDREDI, "atelier");
System.out.println("Planning : " + planning);
System.out.println("Jeudi    : " + planning.get(Jour.JEUDI));
```

```
Planning : {MARDI=réunion, VENDREDI=atelier}
Jeudi    : null
```

Les deux structures sont à préférer à leurs équivalents génériques dès que la clé ou l'élément est une énumération : mêmes contrats (`Set`, `Map`), coût et déterminisme du parcours garantis par la représentation compacte.

11 10. À retenir

- Le tableau ne garantit ni la taille variable, ni l'unicité, ni l'accès par clé ; `java.util` organise ces garanties en contrats d'interface : `List` (ordre d'insertion, indice), `Set` (unicité), `Map` (association clé→ valeur, hors hiérarchie `Collection`), `Queue` (sortie par l'avant).
- Chaque implémentation échange une garantie contre un coût mesurable : `ArrayList` contre `LinkedList` (accès indexé contre insertion en position), `HashSet` contre `TreeSet` (constance moyenne sans ordre contre ordre rouge-noir logarithmique), `HashMap` contre `TreeMap` (même alternative sur les clés). Les valeurs proviennent de la javadoc citée en §3–§6.
- Le parcours est un protocole indépendant du rangement (`Iterator`, `Iterable`) ; la boucle améliorée admet deux traductions, par `iterator()` pour les collections, par indice pour les tableaux (JLS §14.14.2).
- Les promesses des tables de hachage reposent sur le contrat `equals`/`hashCode` (`Object.hashCode`, javadoc) : une clé mutée après insertion devient introuvable (§5, exécuté).
- Le type déclaré suit la convention de retour (§8) : interface en entrée, interface en sortie sauf nécessité documentée ; les vues en lecture seule refusent la modification par `UnsupportedOperationException` (traité sec-07).

La bibliothèque range des objets dans la mémoire du programme. Quand les conteneurs doivent survivre au programme et résider dans un système de gestion de bases de données, le problème change de nature : la séquence JDBC (sec-12) l'aborde, et l'exercice de synthèse P6 (DAO paramétré `GenericDAO`, révision sec-13) consommera pour cela les types paramétrés de `06a_java_L_genericite` et les structures de la présente leçon.

12 Annexes

12.1 Signatures de l'interface Map (dump de référence)

Surface complète de l'interface `java.util.Map` reprise de l'original ; bloc de référence, non exécuté par le rendu.

```
void clear()
boolean containsKey(Object key)
boolean containsValue(Object value)
Set<Map.Entry<K,V>> entrySet()
boolean equals(Object o)
V get(Object key)
int hashCode()
boolean isEmpty()
```

```
Set<K> keySet()
V put(K key, V value)
void putAll(Map<? extends K, ? extends V> t)
V remove(Object key)
int size()
Collection<V> values()
```

Bloc non exécuté par le rendu.

12.2 Eclipse Collections (aperçu)

Bibliothèque externe au JDK (Eclipse Collections, `https://eclipse.dev/collections/`), qui complète le framework par des structures supplémentaires, mutables et immuables, des collections spécialisées pour les types primitifs (`IntList` sans boxing de `Integer`) et une intégration directe aux lambdas et aux flux. Chargement et exemple — bloc non exécuté par le rendu : le pin de dépendance (11.1.0, date de l'original 2023) exige un contrôle d'actualité et une résolution réseau que le rendu du cours n'engage pas.

```
%maven org.eclipse.collections:eclipse-collections-api:11.1.0
%maven org.eclipse.collections:eclipse-collections:11.1.0
```

```
import org.eclipse.collections.impl.factory.*;
import org.eclipse.collections.impl.factory.primitive.IntLists;

MutableIntList intList = IntLists.mutable.of(1, 2, 3);
// intList : somme, filtre et transformations sans boxing des entiers
```

Bloc non exécuté par le rendu.