

Collections

Rémi Forax

Historique

Java 1.0: pas de collections (1995)

Vector, Stack, Hashtable

Java 1.2: API des collections (1998)

List, Set, Map et ArrayList, HashSet, HashMap et Arrays

Java 1.5/1.6: Generics + queue + collections concurrentes (2004-2007)

Queue/Deque, CopyOnWriteArrayList, ConcurrentHashMap

Java 1.8: Lambdas (2014)

Collection.removeIf(), Map/Collection.forEach(), List.sort(), Map.computeIfAbsent()

Java 11: Collection non modifiable (2018)

List.of()/copyOf(), Set.of()/copyOf(), Map.of()/copyOf()

Java 21: SequencedCollection + List improvements (2023)

List.reversed() / List.getFirst() / List.getLast()

Plan

Partie 1: Design de l'API des collections

- types abstraits
- vue
- interface et mutations
- parcours

Parties 2 : Implantations

- différentes classes
- itérateurs
- classes abstraites

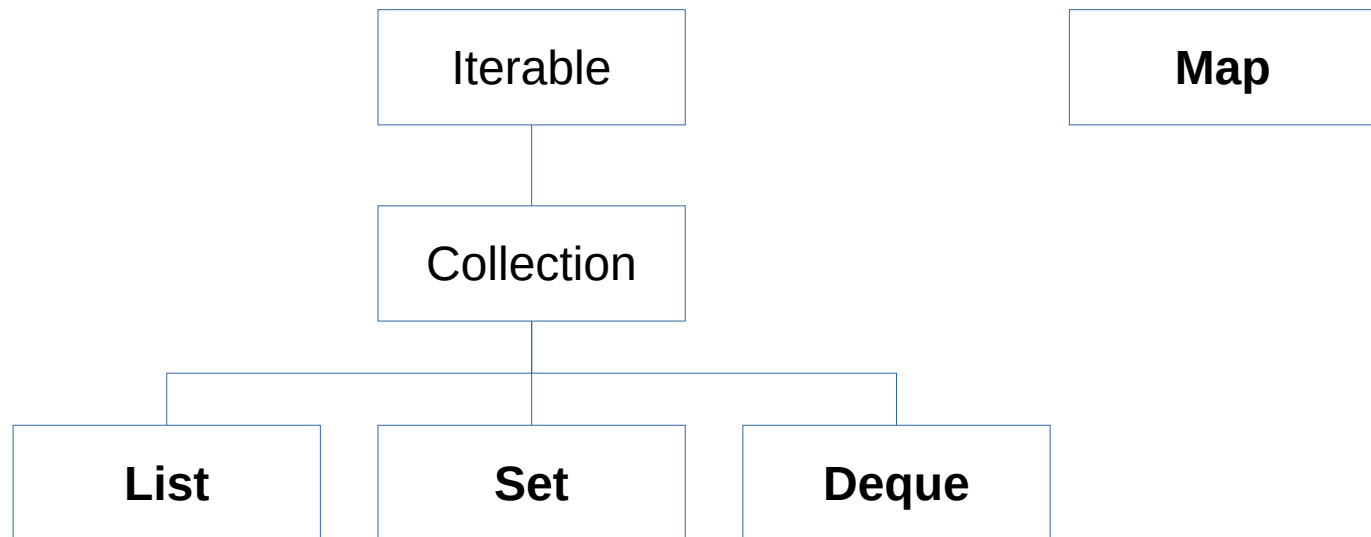
Partie 3 : Contrats des API

- différentes méthodes et leurs subtilités
- le concept d'ordre

Partie 4 : Class Legacy

Types abstraits

L'API définit les types abstraits (interfaces) :
List, Set, Deque et Map



Interfaces et Implantations

L'API définit les interfaces

List, Set, Deque et Map

Deux sortes d'implantations

Les implantations nommées, par exemple :

- ArrayList, HashSet, ArrayDeque, HashMap, ...

Les implantations anonymes :

- List.of(), Arrays.asList(), List.subList(), Map.copyOf(), ...

Typage : Interface vs. Implantation

Doit-on typer une collection par son interface ou par son implantation ?

```
List<String> list = new ArrayList<>();
```

ou

```
ArrayList<String> list = new ArrayList<>();
```

Typage : Interface vs. Implantation

Deux problèmes contradictoires :

Si une méthode prend en paramètre une implantation, on ne peut pas lui en passer une autre

```
public void m(ArrayList<String> list) { ... }
```

Une opération définie par une interface peut avoir une complexité différente suivant l'implantation

```
public void m(Set<String> list, String value) {  
    set.contains(value);  
    // HashSet ou Set.of() : complexité O(1)  
    // TreeSet : complexité O(ln n)  
}
```

Bonne pratique

Si la collection fait partie de l'**API** de la classe, on utilise l'interface (paramètres, type de retour des méthodes publiques)

```
public class Library {  
    public List<Book> getAllBooks() { ... }  
}
```

Il faut faire attention à la complexité pire cas !!

Sinon, on utilise l'implantation (champ privé, variable locale)

```
public class Library {  
    private final ArrayList<Book> books = new ArrayList<>();  
  
    public void compute() {  
        ArrayList<Book> result = new ArrayList<>(books); // var ?  
        ...  
    }  
}
```


Constructeurs

Une Collection/Map nommée doit toujours avoir au moins deux constructeurs :

- Un constructeur sans paramètre
 - `new ArrayDeque()`, `new HashMap()`, etc
- Un constructeur qui prend une autre collection/map en paramètre et recopie tous les éléments
 - `ArrayDeque<E>(Collection<? extends E>)`
 - `ArrayList<E>(Collection<? extends E>)`
 - `HashMap<K,V>(Map<? extends K, ? extends V>)`

Pré-dimensionnement

Performances : les implantations mutables ont un redimensionnement (resize, rehash) qui peut coûter cher.

Pré-dimensionner (static factory sauf ArrayList):

<E> HashSet.newHashSet(capacity)

<E> LinkedHashSet.newLinkedHashSet(capacity)

<K,V> HashMap.newHashMap(capacity)

<K,V> LinkedHashMap.newLinkedHashMap(capacity)

ArrayList<E>(capacity)

Contrat sur les éléments/clés

Les interfaces sont paramétrées

List<E>, Set<E>, Deque<E> et Map<K,V>

Les éléments (E) ou les clés (K) doivent implanter les méthodes

equals(Object) : teste si un élément est égal à n'importe quel objet

hashCode() : un entier résumant les champs de l'objet

toString() : un affichage

Exemple si on oublie hashCode()

```
final class Person {
    private int age;

    public Person(int age) { this.age = age; }
    public boolean equals(Object o) {
        return o instanceof Person p && age == p.age;
    }
}

...
var person = new Person(32);
var set = Set.of(person);
System.out.println(set.contains(new Person(32))); // false
```

Contrat sur les éléments/clés (2)

Les éléments ne doivent **pas** être **modifiés** après insertion

```
final class Person {  
    ...  
    public int hashCode() { return age; }  
}  
  
...  
var person = new Person(32);  
var set = Set.of(person);  
person.age = 23; // mutation ahhh  
System.out.println(set.contains(person)); // false
```

Élément **null** ?

Suivant les versions de Java, les implantations des collections supportent ou pas null :(

- Java 1.0 : null est interdit (Vector, Hashtable)
- Java 1.2 : null est ok (ArrayList, HashMap)
- Java 1.5+ : null est interdit (ArrayDeque, List.of())

La bonne pratique est de ne jamais mettre de null dans une Collection/Map.

Null veut dire quelque chose :(

Il y a 4 méthodes de Map bizarres

Si il n'y a pas de valeur associée à la clé :

- `map.get(clé)` renvoie null (`getOrDefault()` est mieux)
- `map.compute(clé, biFunction)` appelle la `biFunction` avec null en 2^e paramètre (`merge()` est mieux)

Si la `biFunction` renvoie null, on supprime l'élément :

- `map.compute(clé, biFunction)`,
- `map.computeIfPresent(clé, biFunction)` et
- `map.computeIfAbsent(clé, biFunction)`

On ne renvoie pas null !

Une méthode qui renvoie une Collection/Map ne renvoie pas null mais une Collection/Map vide

```
public static Set<String> findBlahBlah() {  
    if (...) {  
        return null; // ahhhh  
    }  
    ...  
}
```

```
public static Set<String> findBlahBlah() {  
    if (...) {  
        return Set.of(); // ok !  
    }  
    ...  
}
```


Les Vues

Vue

Une vue est une implantation qui partage ses éléments avec une collection/tableau déjà existant

- Permet de voir une collection comme une autre
 - `Arrays.asList(array)`, `map.keySet()`, `map.entrySet()`
- Plus performant (pas de copie des éléments)
 - Mais ne respecte pas le principe d'encapsulation

Une vue peut être modifiable ou non

List.asList() + List.subList() (slice)

Une implantation créée à partir d'une autre implantation peut ne pas stocker les données

```
var array = new String[] { "A", "B", "C", "D" };  
var list = Arrays.asList(array);  
var subList = list.subList(1, 3); // entre 1 et 3 exclu  
subList.set(0, "Z");  
println(Arrays.toString(array)); // [A, Z, C, D]
```

List.subList() est modifiable si la List est modifiable

Interface et Mutations

Interface et Mutations

L'API utilise une même interface pour les implantations modifiables et non modifiables

- Ça évite d'avoir trop d'interfaces...

Mais ça implique que les méthodes modifiant la collection sont *optionnelles*

- add, remove, set, replace, clear, etc

Elles peuvent lever une UnsupportedOperationException

Encapsulation

Prendre une collections en paramètre d'un constructeur public ?

- On doit prévoir pour le pire cas, c'est à dire que la collection est modifiable

```
public class Library {  
    private final ArrayList<Book> books;  
    public Library(List<Book> books) {  
        this.books = new ArrayList<>(books); // defensive copy  
    }  
}
```

Le champ books **ne doit pas** être visible de l'extérieur !

Encapsulation

Renvoyer une collection par une méthode publique ?

- La collection doit être dupliquée ou non-modifiable

```
public class Library {  
    private final ArrayList<Book> books = new ArrayList<>();  
    public List<Book> books() {           // accesseur  
        return new ArrayList<>(books); // defensive copy  
        // ou  
        return List.copyOf(books);      // defensive copy  
        // ou  
        return Collections.unmodifiableList(books); // non modifiable view  
    }  
}
```

La meilleure façon de résoudre ce problème est de ne pas avoir d'accesseur !

Encapsulation

Dans le cas d'un record, on a toujours un accesseur, donc il faut dupliquer dans une liste non modifiable

```
public record Library(List<Book> books) {  
    Library {  
        books = List.copyOf(books); // defensive copy  
    }  
}
```

Le champ books **doit** stocker un version non-modifiable !!

Collection que l'on ne possède pas

Recevoir une collection en paramètre d'une méthode publique :

- On doit prévoir le pire cas, c'est à dire que la collection n'est pas mutable

```
class Person {  
    private Book book;  
  
    public void addToBooks(List<Book> books) {  
        books.add(book); // ahhh  
    }  
}
```

Très mauvais design, on ne doit pas supposer qu'une collection est modifiable

Parcours

Interne vs. externe

Il y a deux sortes de parcours

– Itération interne

- Collection/Map possède une méthode **forEach**(consumer)
 - Plus efficace mais avec les limitations des lambdas

– Itération externe

- Collection possède une méthode **iterator**()
 - Un peu moins efficace mais plus expressif

Itération Interne

La méthode `forEach(consumer)` prend un `consumer` qui est appelé sur tous les éléments (resp. clé/valeur) de la collection (resp. map)

- **var** list = List.of(1, 2, 3);
list.forEach(v -> println(v));
- **var** map = Map.of(1, "cat", 2, "dog");
map.forEach((k, v) -> println(k + " " + v));

Limitation de l'itération interne

Une lambda ne capture que des valeurs *effectivement finales*

```
var list = List.of(1, 2, 3);  
var sum = 0;  
list.forEach(v -> sum += v); // compile pas  
println(sum);
```

“sum” est modifiée, donc pas final

Itération externe

L'interface Collection possède une méthode `iterator()` qui renvoie un `Iterator`

- Un `Iterator` est un curseur qui parcourt les éléments

```
interface Iterator<E> {  
    boolean hasNext(); // y-a-t-il un suivant ?  
    E next(); // renvoie l'élément courant et passe au suivant  
}
```

Utilisation d'un itérateur

Avant Java 1.5 : on utilise une boucle **while**

```
- var list = List.of(1, 2, 3);  
  var sum = 0;  
  var iterator = list.iterator(); // Iterator creation  
  while(iterator.hasNext()) {  
    var value = iterator.next();  
    sum += value;  
  }  
  println(sum);
```

Boucle améliorée

Java 1.5+ : on utilise **for(:)** si l'objet est Iterable

```
var list = List.of(1, 2, 3);  
var sum = 0;  
for(var value : list) {  
    sum += value;  
}  
println(sum);
```

Le compilateur génère le while avec l'itérateur

Iterable

Interface de tous les objets sur lesquels on peut faire un **for(:)**

```
interface Iterable<E> {  
    Iterator<E> iterator();  
  
    default void forEach(Consumer<? super E> consumer) {  
        for(var value : this) {  
            consumer.accept(value);  
        }  
    }  
}
```

Iterable possède aussi une méthode `forEach()` par défaut !

Et avec un index ?

On ne peut/veut pas parcourir une collection avec un index

- car les Set et les Queue n'ont pas de notion d'index
- car `List.get(index)` peut avoir une complexité **$O(n)$**

Un itérateur (un curseur) permet de parcourir une collection en complexité **$O(n)$**

Index => problème de complexité

Exemple

```
LinkedList<Integer> list = ... // doublement chaînée  
for (var i = 0; i < list.size(); i++) { // O(n2) ahhh  
    var element = list.get(i);  
    ...  
}
```

```
LinkedList<Integer> list = ... // doublement chaînée  
for (var element : list) { // O(n) ok  
    ...  
}
```

Parcourir en sens opposé

List, Queue et certains Set possèdent une méthode **reversed()** qui renvoie une vue inversée

```
List<String> list = ...  
for (var element : list.reversed()) {  
    ...  
}
```

comme `reversed()` est une vue, les éléments ne sont pas dupliqués !

Modification **structurelle**

L'API définit deux sortes de modifications

- Les modifications qui changent la structure (*size change*)
ex: List.add(), Set.remove()
- Les modifications non structurelles
 - ex: List.set() ou Map.replace()

exemple : Arrays.asList() ne permet que les modifications non structurelles

```
var list = Arrays.asList(1, 2, 3);  
list.add(4); // UnsupportedOperationException  
list.set(0, 42); // ok !
```

Itérateur et Mutation

Que se passe-t-il si on modifie structurellement la collection lors du parcours ?

- Si c'est une collection non modifiable
 - UnsupportedOperationException lors de la mutation
- Sinon, ça plante au prochain iterator.next() (*failfast*)
 - Lève une ConcurrentModificationException

Exemple qui plante

Supprimer des éléments

```
public class Group{
    private final ArrayList<Person> persons...

    void removeAllJohns() {
        for(var person : persons) { // CME
            if (person.firstName().equals("John")) {
                persons.remove(person);
            }
        }
    }
}
```

Ce code ne fonctionne pas !!

Exemple qui fonctionne

Supprimer un élément

```
void removeAJohn() {  
    for(var person : persons) {  
        if (person.firstName().equals("John")) {  
            persons.remove(person);  
            return;  
        }  
    }  
}
```

On n'utilise plus l'iterator après la suppression => OK !

ConcurrentModificationException

Comme son nom ne l'indique pas, ce n'est pas un problème de concurrence

- Le problème est une mutation de la collection lors du parcours de cette collection

Résoudre le problème

- Enregistrer les mutations et les appliquer *a posteriori*
- Recréer une nouvelle collection avec un Stream
- Utiliser les méthodes de l'itérateur

Méthodes de l'itérateur

Un itérateur possède des méthodes pour modifier la collection dont il est issu (pas de problème de CME)

- Iterator.**remove()**
Supprime l'élément renvoyé par next()
- ListIterator.**add()**
Ajoute un élément après celui renvoyé par next()
- ListIterator.**set()**
Change l'élément renvoyé par next()

ListIterator est un itérateur spécifique sur les List

Exemple

Supprimer des éléments sur l'itérateur

```
void removeAllJohn() {  
    var iterator = persons.iterator();  
    while(iterator.hasNext()) {  
        var person = iterator.next();  
        if (person.name().equals("John")) {  
            iterator.remove();  
        }  
    }  
}
```

Note : ici, on devrait utiliser `persons.removeIf(...)` qui est plus efficace

Plan

Partie 1: Design de l'API des collections

- types abstraits
- vue
- interface et mutations
- parcours

Parties 2 : Implantations

- différentes classes
- itérateurs
- classes abstraites

Partie 3 : Contrats des API

- différentes méthodes et leurs subtilités
- le concept d'ordre

Partie 4 : Class Legacy

Implantations de List

Implantations les plus importantes :


- `ArrayList<E>` : tableau dynamique

`E E E`  ...

- `Arrays.asList(E...)` : vue d'un tableau (taille fixe)

`E E E E E E`

- `List.of(E...)`, `List.copyOf(Collection)` : implantation non modifiable

`E E E E E E` 

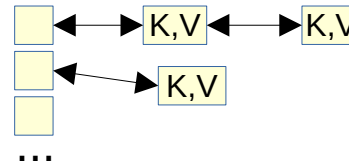
- `Collections.nCopies(int n, E value)` : n fois le même élément

n `E`

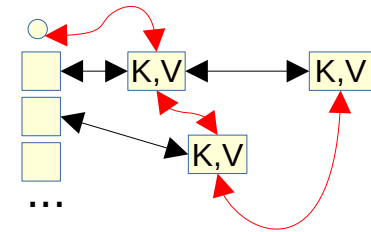
Implantations de Map

Implantations les plus importantes :

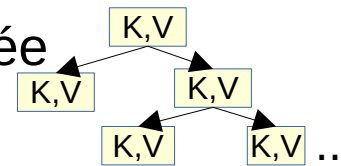
- `HashMap<K,V>` : table de hachage



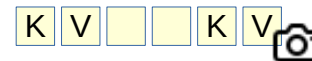
- `LinkedHashMap<K,V>` : maintien ordre d'insertion



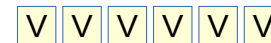
- `TreeMap<K,V>` : arbre rouge/noir, triée



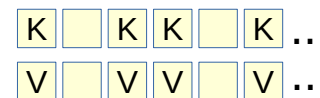
- `Map.of(K,V, ...)`, `Map.copyOf(Map)` : implantation non modifiable



- `EnumMap<K extends Enum<K>, V>` : les clés viennent d'un même enum



- `IdentityHashMap<K,V>` : utilise `System.identityHashCode()` et `==`, au lieu de `hashCode()` et `equals()`.



Implantations de Set

Implantations les plus importantes :

- HashSet<E> : table de hachage
 - Utilise HashMap en interne
- LinkedHashSet<E> : maintien ordre d'insertion
 - Utilise LinkedHashMap en interne
- TreeSet<E> : arbre rouge/noir, trié
 - Utilise TreeMap en interne
- Set.of(E...), Set.copyOf(Set) : implantation non modifiable

E E E E E



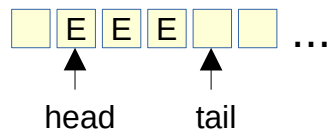
- EnumSet<E extends Enum<E>> : les éléments viennent d'un même enum

long long long

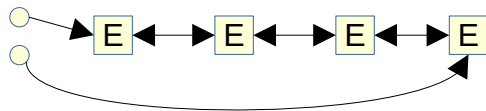
Implantation des Deque

Implantations les plus importantes :

- `ArrayDeque<E>` : tableau circulaire



- `LinkedList<E>` : liste doublement chaînée



Écrire sa propre implantation

Il existe des classes abstraites qui implément la plupart des méthodes à partir de quelques autres

Offrent uniquement une version non modifiable

- `AbstractList` pour les `List`
- `AbstractSet` pour les `Set`
- `AbstractMap` pour les `Map`
- `AbstractQueue` pour les `Deque`

Implanter une List

La classe abstraite `AbstractList` fournit une implantation pour les listes non modifiables, qui n'est efficace que si le `get(index)` est en $O(1)$

Méthodes à implanter :

- `int size()`
- `E get(index)`

Il faut aussi implanter l'interface `RandomAccess` pour indiquer que l'implantation de `get(index)` est en $O(1)$

Iterator<E>

Écrire un Iterator demande d'écrire au moins les méthodes

- boolean hasNext()
 - y a-t-il un élément suivant ?
- E next()
 - renvoie l'élément courant et passe au suivant.

Attention, la méthode next() doit lever une exception NoSuchElementException si il n'y a pas d'élément courant.

Comment écrire son itérateur

- ```
String[] array = ...
for (int i = 0; i < array.length; i++) {
 var element = array[i];
 ...
}
```
  - ```
Iterator<String> iterator() {  
    return new Iterator<E> () {  
        private int i = 0;  
        public boolean hasNext() {  
            return i < array.length;  
        }  
        public E next() {  
            if (!hasNext()) { throw new NoSuchElementException(); }  
            var element = array[i];  
            i++;  
            return element;  
        }  
    };  
}
```
- initialisation
- test
- incréméntation
-

Implanter un Set

La classe abstraite `AbstractSet` fournit une implémentation pour les ensembles non modifiables

Méthodes à implanter :

- `int size()`
- `Iterator<E> iterator()`

Mais attention, `contains()` est en $O(n)$, donc il faut la redéfinir :

- `boolean contains(Object)`

Implanter une Map

La classe abstraite `AbstractMap` fournit une implantation pour les dictionnaires non modifiables.

Méthodes à implanter

- `int size()`
- `Set<Map.Entry<K,V>> entrySet()`

Mais `get/getOrDefault()/containsKey()` sont $O(n)$, donc il faut les redéfinir :

- `V get(Object)`
- `V getOrDefault(Object, V defaultValue)`
- `boolean containsKey(Object)`

Plan

Partie 1: Design de l'API des collections

- types abstraits
- vue
- interface et mutations
- parcours

Parties 2 : Implantations

- différentes classes
- itérateurs
- classes abstraites

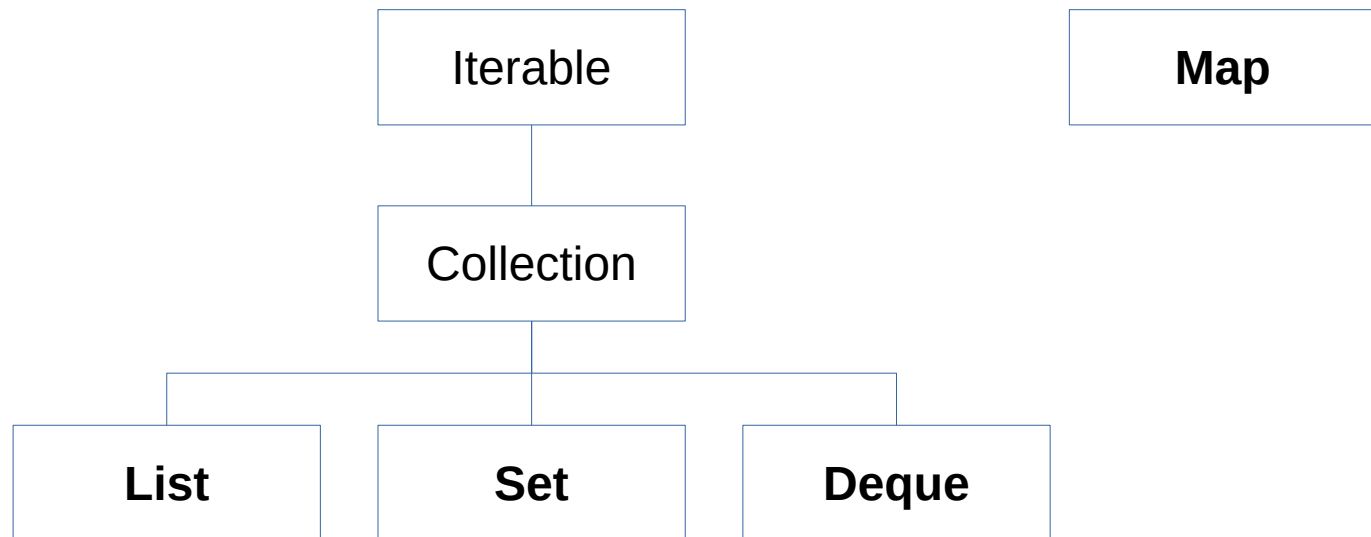
Partie 3 : Contrats des API

- différentes méthodes et leurs subtilités
- le concept d'ordre

Partie 4 : Class Legacy

Interface & Contrat

Les interfaces ont des contrats différents



java.util.Collection<E>

Interface de base des collections

- isEmpty()/size()
- boolean add*(E)
- contains(Object), boolean remove*(Object)
- equals(Object), hashCode(), toString()
- toArray(), toArray(intFunction), toArray(T[])

Il n'y a pas d'ordre spécifié

* méthode qui fait des mutations structurelles

E/K vs. Object ?

Les méthodes de recherche `contains()`, `remove()` ou `Map.get()` prennent un `Object` en paramètre pas un E/K

- Car on peut rechercher un objet avec un type différent que celui des éléments / clés

```
class Girafe implements Animal, Tall { }
```

```
...  
Girafe bob = new Girafe()  
Animal animal = bob; // sous-typage  
Tall tall = bob; // sous-typage  
List<Animal> list = List.of(animal);  
list.contains(tall) // contains doit être typé Object
```

Valeur de retour des méthodes

Les méthodes qui modifient une Collection/Map renvoient un booléen, pas void (beurk !)

- true si la collection/map a été modifiée

Permet de savoir si l'opération a été effectuée

- Set.add(E) renvoie false si l'élément est déjà présent dans l'ensemble
- List.addAll(Collection) renvoie false si l'élément n'est pas ajouté à la liste

equals()

equals() est définie de façon étrange

- Il faut que les éléments soit égaux
- Pour une List, l'autre collection doit être une List
- Pour un Set, l'autre collection doit être un Set

```
List.of(1, 2, 3).equals(Set.of(1, 2, 3)) // false
```

La méthode toArray()

Les tableaux connaissent la classe de leurs items à l'exécution mais avec l'érasure la classe de E est perdue.

Object[] toArray()

Renvoie un tableau d'objets, pas un tableau de E !

```
List<String> list = ...  
String[] array = (String[]) list.toArray(); // CCE
```

Autres méthodes toArray()

`<T> T[] toArray(IntFunction<T[]> function)`

- Prend une fonction de création de tableau en paramètre (par ex: `String[]::new`)
- Le type du tableau peut être différent du type de la collection (mais `ArrayStoreException` à l'exécution)

`<T> T[] toArray(T[] array)`

- Si le tableau est plus petit, créé un tableau de la bonne taille
- Si le tableau est plus grand, ajoute un null après le dernier élément (`ahhhh`)

Exemple d'utilisation de toArray()

Avant Java 11,

```
List<Object> list = List.of("foo", "bar");  
String[] array = list.toArray(new String[0]); // astuce !
```

Après Java 11,

```
List<Object> list = List.of("foo", "bar");  
String[] array = list.toArray(String[]::new);
```

Le fait que les éléments sont des String est testé à l'insertion de chaque élément dans le tableau

java.util.List

Impose l'ordre d'insertion

Méthodes supplémentaires

- `set(index, E)`, `E get(index)`, `E getFirst()`, `E getLast()`,
`boolean add*(index, E)`, `addFirst*(E)`, `addLast*(E)`,
`E remove*(index)`, `E removeFirst*()`, `E removeLast*()`
- `int indexOf(E)`, `int lastIndexOf(E)`
- `replaceAll(unaryOp)`
- `List<E> subList(start, end)`, `List<E> reversed()`
- `sort(comparator)`

Méthodes indexées et contains()

Méthodes rarement utilisées en pratique

- Les méthodes `get(index)` et `set(index, E)` peuvent être en $\mathbf{O}(n)$
- Les méthodes `add(index, E)`, `addFirst(E)`, `remove(index)` et `removeFirst()` sont en $\mathbf{O}(n)$

Rechercher dans une List est lent !

- `contains(Object)` est aussi en $\mathbf{O}(n)$

Premier et dernier élément

Depuis Java 21, il existe des méthodes pour obtenir le premier ou le dernier élément

premier élément

```
var first = list.getFirst();  
var first = list.get(0); // avant Java 21
```

dernier élément

```
var last = list.getLast();  
var last = list.get(list.size() - 1); // avant Java 21
```

Chaque implantation de List garantit que l'on accède au premier/dernier élément en temps constant

Relation avec les tableaux

Normalement, si on connaît le nombre d'éléments à l'avance, on devrait utiliser un tableau et pas une List.

Mais comme créer un tableau de type paramétré est impossible

- On utilise une List même si on connaît la taille à l'avance

java.util.Set

Ensemble sans doublons

Ne possède pas de méthodes supplémentaires par rapport à l'interface Collection

Permet la recherche et la suppression rapide :

- en $O(1)$ ou $O(\ln n)$

java.util.Deque

Pile, file, etc, ...

Par défaut, on ajoute en fin et retire au début.

Méthodes supplémentaires :

- boolean offer(E) / offerFirst(E) / **offerLast(E)**
ajoute si c'est possible
- E poll() / E **pollFirst()** / E pollLast()
retire si possible, sinon renvoie null
- E peek() / E **getFirst()** / E getLast()
valeur sans suppression, ou null si pas de valeur

Relation avec les collections

Les méthodes `add()` et `remove()` sont les équivalents de `offer()` et `poll()` mais elles lèvent une exception si l'opération n'est pas possible.

La méthode `element()` lève une exception si il n'y a pas de premier élément.

java.util.Map

Dictionnaire qui associe une valeur à une clé. Les clés n'ont pas de doublons.

Méthodes :

- V get(Object clé), V getOrDefault(Object clé, V defaultValue),
- V put*(K, V), V putIfAbsent*(K, V)
- V replace(K, V), boolean replace(K, V, V), V remove*(Object), boolean remove*(Object, Object)
- V computeIfAbsent*(K, Function<K,V>)
- V merge*(K,V, BiFunction<V,V,V>)
- Set<Map.Entry<K, V>> entrySet(), Set<K> keySet(), Collection<V> values()
- forEach(BiConsumer<K,V>)

Map.computeIfAbsent(clé, fonction)

Pour ne pas recalculer une valeur déjà calculée (cache)

```
class Cache {  
    static int myFunction(int value) { ... }  
  
    private final HashMap<Integer, Integer> map =  
        new HashMap<>();  
  
    public int lookup(int value) {  
        return map.computeIfAbsent(value, v -> myFunction(v));  
    }  
}
```

Et pour les mises à jour de Map à valeurs mutables

Map.merge(K, V, BiFunction<V, V, V>)

Calculer le nombre d'occurrences des mots d'une liste.

```
public static Map<String, Integer> group(List<String> list) {  
    var map = new HashMap<String, Integer>();  
    for(var word: list) {  
        map.merge(word, 1, (oldV, v) -> oldV + v);  
    }  
    return map;  
}
```

Note : on utilise plutôt `Collectors.groupingBy()`

Map.Entry<K,V>

Interface représentant une paire (un couple) clé/valeur

```
interface Map {  
    interface Entry<K,V> { // à l'intérieur de l'interface Map  
        K getKey();  
        V getValue();  
        default void setValue(V value) { throw new UOE(); }  
    }  
    ...  
}
```

Si la Map est modifiable, Entry est modifiable

entrySet()

Il n'existe pas d'itérateur sur une Map

La méthode `entrySet()` est une vue des couples clé/valeur de la Map

```
for (var entry : map.entrySet()) {  
    var key = entry.getKey();  
    var value = entry.getValue();  
    ...  
}
```

Toutes modification de la Map est reflétée dans la vue renvoyée par `entrySet()` et vice-versa

keySet() et values()

Comme `entrySet()`, `keySet()` et `values()` sont des vues de la Map

Ensemble des clés :

```
for (var key : map.keySet()) {  
    ...  
}
```

Collection des valeurs :

```
for (var value : map.values()) {  
    ...  
}
```

Ordre

L'ordre des éléments/clés dépend de l'interface ou de l'implantation

- Sans ordre (Set)
- Ordre d'insertion (SequencedCollection) : par exemple List, Deque, LinkedHashSet/Map)

Cas particulier : ordre d'accès

- LinkedHashMap(capacity, factor, true))
- Triée avec fonction de comparaison (TreeSet/TreeMap)

Comparator<T>

Fonction de comparaison entre deux éléments de même type

```
interface Comparator<T> {  
    int compare(T t1, T t2);  
}
```

Même sémantique que strcmp

- <0 si $t1 < t2$
- >0 si $t1 > t2$
- ==0 si $t1.equals(t2)$

Attention aux overflow !

On ne peut pas utiliser '-' dans l'implantation

```
record Person(int id) {}
```

```
Comparator<Person> comparator =  
    (p1, p2) -> p1.id() - p2.id(); // ahhh
```

Si id est Integer.MIN_VALUE ou proche,
problème !

```
Comparator<Person> comparator =  
    (p1, p2) -> Integer.compare(p1.id(), p2.id()); // ok
```

Ordre « naturel »

On peut définir un ordre naturel sur une classe avec l'interface Comparable

```
record Person(int id) implements Comparable<Person> {  
    public int compareTo(Person p) {  
        return Integer.compare(id, p.id);  
    }  
}
```

Attention, compareTo doit fonctionner avec equals() !

$a.compareTo(b) == 0 \iff a.equals(b)$

Comparator.comparing()

Méthode statique qui crée un Comparator à partir d'une fonction de projection

Attention au boxing :

- Comparator<Person> comparator =
 Comparator.**comparingInt**(Person::id); // ok
- Comparator<Person> comparator =
 Comparator.comparing(Person::id); // ahh

Le type de id doit être un Comparable ou un type primitif.

Concurrence

Le package `java.util.concurrent` définit des implantations concurrent et lock-free

- List (`CopyOnWriteArrayList`)
- Set (`CopyOnWriteArraySet`, `ConcurrentSkipListSet`)
- Map (`ConcurrentHashMap`, `ConcurrentSkipListMap`)

Algos :

- copy on write = on duplique le tableau à chaque mutation
- concurrent = on utilise l'instruction CAS et volatile
- skip list = liste triée (comparator) en $O(\ln n)$

java.util.concurrent et size

Pour les collections

- de java.util
 - La complexité de la méthode size() est **O(1)**
- de java.util.concurrent
 - La complexité de la méthode size() peut être **O(n)**

La complexité de isEmpty() est toujours en **O(1)**

Concurrence et parcours

Si la collection est concurrente, pas de CME lors du parcours

Deux sémantiques :

- On ne voit pas la modification (*Snapshot At The Beginning*)
 - CopyOnWriteArrayList, CopyOnWriteArraySet
- On peut voir la modification ou pas (*Weakly Consistent*)
 - ConcurrentHashMap, ConcurrentSkipList/Set

Plan

Partie 1: Design de l'API des collections

- types abstraits
- vue
- interface et mutations
- parcours

Partie 2 : Contrats des API

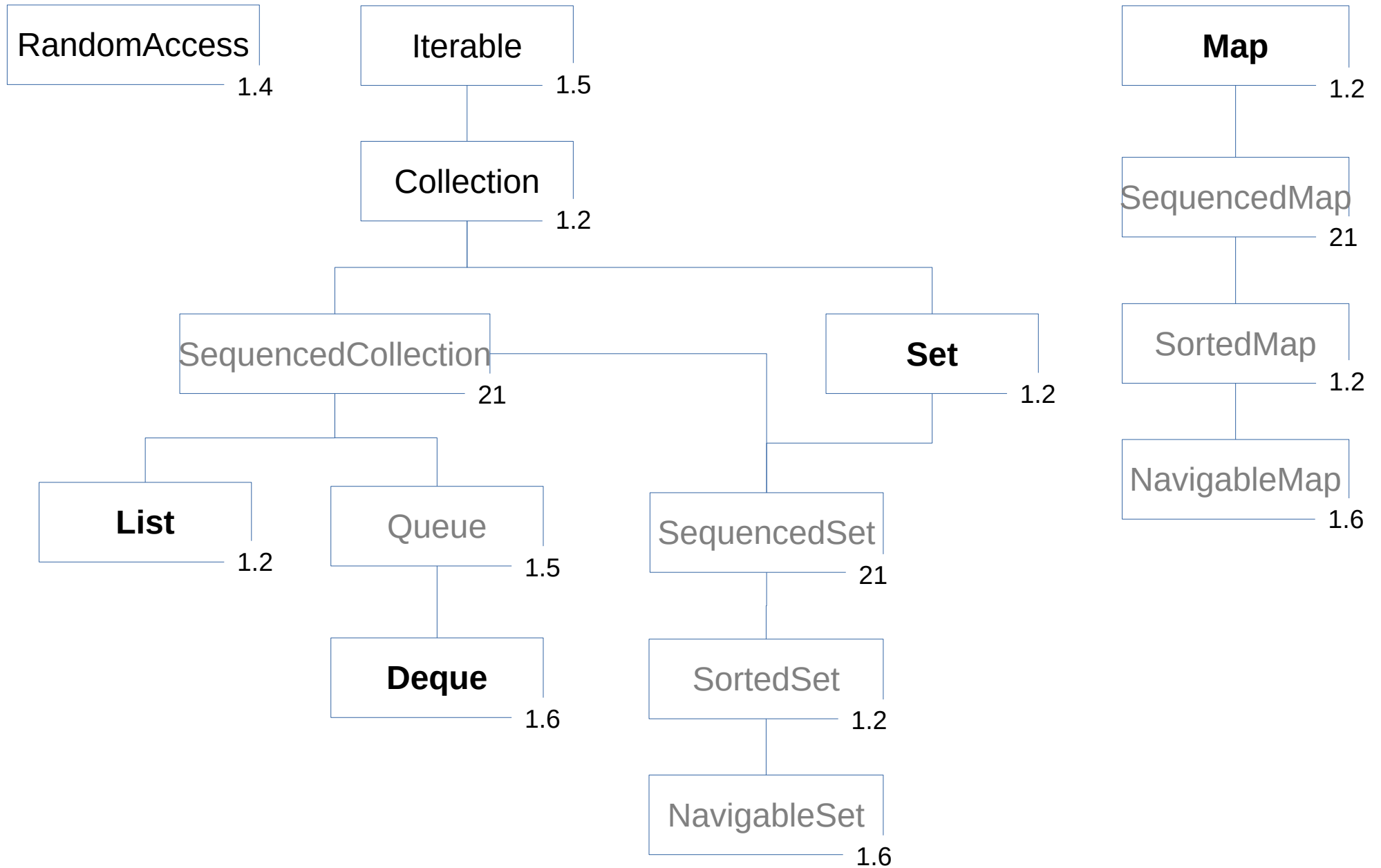
- différentes méthodes et leurs subtilités
- le concept d'ordre

Parties 3 : Implantations

- différentes classes
- itérateurs
- classes abstraites

Partie 4 : Class Legacy

API (trop détaillée)



Interfaces/Classes démodées

Certaines interfaces/classes...

- ... ne sont pas beaucoup utilisées ;
- ... ont été utilisées mais remplacées
 - À cause de problèmes algorithmique et/ou
 - À cause de problèmes de performance

Avant Java 1.2 (avant 1998)

Les classes avant l'API des collections ont leurs méthodes synchronized sauf l'itérateur

- Lent si on n'a pas besoin de concurrence
- Lent par rapport aux classes de `java.util.concurrent`
- Dangereuses quand on utilise un itérateur

Classes et leurs classes de remplacement :

- `Vector` est remplacée par `ArrayList`
- `Stack` est remplacée par `ArrayDeque` (il y a les méthodes `push/pop/isEmpty`)
- `Hashtable` est remplacée par `HashMap`
- `Enumeration` est remplacée par `Iterator`

Classes pas assez efficaces

LinkedList

- Liste doublement chaînée, trop lente par rapport aux implantations à base de tableau
- à remplacer par :
 - ArrayList ou ArrayDeque (si insertion au début)

AbstractSequentialList

classe abstraite implantée seulement par LinkedList

Classe avec un problème d'algo

PriorityQueue

- Permet de trier les éléments avec un tas
- Mais l'algo n'est pas *stable*
 - Deux objets égaux au sens de equals() peuvent être insérés dans un ordre et récupérés dans l'autre.

Interfaces peu utilisées

SequencedCollection

- est l'interface commune entre List, Queue et SequencedSet

SequencedSet, SequencedMap

- sont les interfaces pour LinkedHashSet/LinkedHashMap

SortedSet, NavigableSet, SortedMap, NavigableMap

- sont des interfaces pour TreeSet et TreeMap

Il est rare, mais pas impossible, d'avoir des API qui prennent ces interfaces en paramètre

Collections à zéro ou un élément, non modifiables

Anciennes versions

- `Collections.emptyList()`, `emptySet()`, `emptyMap()` et
- `Collections.singletonList()`, `singleton()`,
`singletonMap()`

Remplacées par `List.of()`, `Set.of()` et `Map.of()`

Qui fonctionnent de concert avec `List/Set/Map.copyOf()`

Implantations de Map.Entry

Obscures implantations jamais vraiment utilisées

- `AbstractMap.SimpleImmutableEntry<K,V>`,
`AbstractMap.SimpleEntry<K,V>`

Remplacées pour la version non modifiable par `Map.entry(K, V)` (attention c'est une méthode !)