JAVA
Les collections

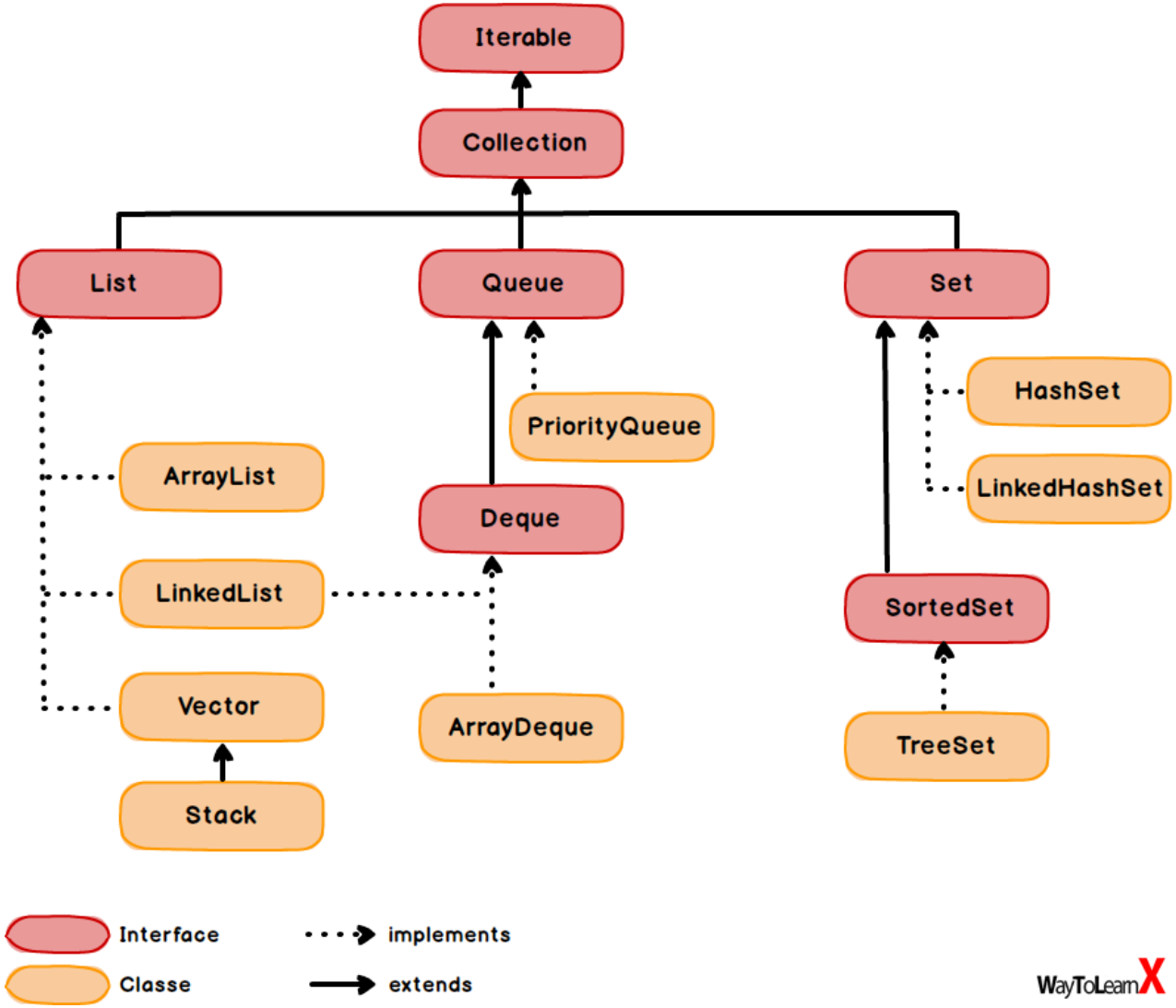
Les collections

- Comment gérer un ensemble d'éléments en Java ?
  - On fait des tableaux : `int[] array = {1, 2, 3};`
    - C'est bien, mais on est vite limités, les tableaux ont une taille fixe
  - On crée nos propres collections :
    `public class maCollection(int... number){ }`
    - C'est bien, il faut tout refaire de zéro…
  - On utilise les (très) nombreuses collections que nous offrent l'API Java

Les collections

- On appelle collections un ensemble de classes et d'interfaces fournies par l'API standard et disponibles pour la plupart dans le package **java.util**
- Java nous met à disposition le **Java Collections Framework**
- Les principaux types de collections :
  - Les listes (lists)
  - Les ensembles (sets)
  - Les tableaux associatifs (maps)
- Toutes les collections héritent de l'interface Iterable : elles peuvent-être parcourues avec une structure de **for** amélioré (foreach)

Java Collections Framework



1. Les listes

- C'est une collection ordonnée d'éléments
- Différentes façons d'implémenter les listes donc les performances sont optimisées :
  - soit pour les accès aléatoires aux éléments
  - soit pour les opérations d'insertion et de suppression d'éléments

Les listes : ArrayList

- La classe **java.util.ArrayList** est une implémentation de l'interface **List**
- Elle stocke les éléments de la liste sous la forme de blocs en mémoire. Cela signifie que la classe ArrayList est très performante pour les accès aléatoire en lecture aux éléments de la liste.
- Par contre, les opérations d'ajout et de suppression d'un élement se font en temps linéaire. Elle est donc moins performante que la classe LinkedList sur ce point

```
List<String> liste = new ArrayList<String>();
liste.add("une première chaîne");
liste.add("une troisième chaîne");
String premierElement = liste.get(0);
```

Les listes : ArrayList

- Il est possible de réserver de l'espace mémoire pour une liste pouvant contenir n éléments. Pour cela, on peut passer la taille voulue à la création d'une instance de **ArrayList** ou en appelant la méthode **ArrayList.ensureCapacity**.
- La liste ne change pas de taille pour autant, un espace mémoire est simplement alloué en prévision

```
// capacité de 10
ArrayList<String> liste = new ArrayList<String>(10);
// capacité d'au moins 100
liste.ensureCapacity(100);
System.out.println(liste.size()); // 0
```

Les listes : LinkedList

- La classe **java.util.LinkedList** est une implémentation de l'interface **List**
- Sa représentation interne est une liste doublement chaînée. Cela signifie que la classe **LinkedList** est très performante pour les opérations d'insertion et de suppression d'éléments.
- Par contre, l'accès aléatoire en lecture aux éléments se fait en temps linéaire. Elle est donc moins performante que la classe **ArrayList** sur ce point

```
List<String> liste = new LinkedList<String>();
liste.add("une première chaîne");
liste.add("une troisième chaîne");
String premierElement = liste.get(0);
```

Les listes : LinkedList

- La classe **LinkedList** implémente également les interfaces **Queue** et **Deque** (double ended queue), elle peut donc représenter des structures de type **LIFO** (Last In First Out) ou **FIFO** (First In First Out).

```
Queue<String> queue = new LinkedList<String>();
// insère un élément dans la file
queue.offer("un élément");
// lit l'élément en tête de la file sans l'enlever
de la file
System.out.println(queue.peek());  // "un élément"
// lit l'élément en tête de la file et l'enlève de
la file
System.out.println(queue.poll()); // "un élément"
System.out.println(queue.isEmpty()); // true
```

Les listes : LinkedList

```
Deque<String> deque = new LinkedList<String>();

// empile deux éléments
deque.push("élément 1");
deque.push("élément 2");


// lit le premier élément de la file sans l'enlever
System.out.println(deque.peekFirst()); // élément 2
// lit le dernier élément de la file sans l'enlever
System.out.println(deque.peekLast()); // élément 1
// lit l'élément de tête de la file sans l'enlever
System.out.println(deque.peek()); // élément 2
// lit l'élément de tête de la file et l'enlève
System.out.println(deque.pop()); // élément 2
System.out.println(deque.pop()); // élément


System.out.println(deque.isEmpty()); // true
```

Les listes : ArrayDeque

- La classe **java.util.ArrayDeque** est une implémentation des interfaces **Queue** et **Deque** (mais elle n'implémente pas **List**).
- Elle est conçue pour être plus performante que **LinkedList** pour les opérations d'ajout et de suppression en tête et en fin de liste.
- Si vous voulez utiliser une collection uniquement pour représenter une file ou une pile de type **LIFO** (Last In First Out) ou **FIFO** (First In First Out), alors il est préférable de créer une instance de la classe **ArrayDeque**.
- Comme pour la classe **ArrayList**, il est possible de réserver un espace mémoire pour n éléments au moment de la création d'une instance de **ArrayDeque :**
- `ArrayDeque<String> arrayDeque = new ArrayDeque<>(100);`

Les listes : ArrayDeque

```
Queue<String> queue = new ArrayDeque<String>();

// insère un élément dans la file
queue.offer("un élément");

// lit l'élément en tête de la file sans l'enlever de la file
System.out.println(queue.peek()); // "un élément"
// lit l'élément en tête de la file et l'enleve de la file
System.out.println(queue.poll()); // "un élément"

System.out.println(queue.isEmpty()); // true
```

Les listes : PriorityQueue

- La classe **java.util.PriorityQueue** permet d'ajouter des éléments dans une file selon un ordre naturel : soit parce que les éléments de la file implémentent l'interface **Comparable**, soit parce qu'une instance de **Comparator** a été fournie à la création de l'instance de **PriorityQueue**.
- Quel que soit l'ordre d'insertion, les éléments seront extraits de la file selon l'ordre naturel.

Les listes : PriorityQueue

```
Queue<String> queue = new PriorityQueue<>();

queue.add("i");
queue.add("e");
queue.add("u");
queue.add("o");
queue.add("a");
queue.add("y");

System.out.println(queue.poll()); // a
System.out.println(queue.poll()); // e
System.out.println(queue.poll()); // i
System.out.println(queue.poll()); // o
System.out.println(queue.poll()); // u
System.out.println(queue.poll()); // y
```

Les listes : Vector et Stack

- La version 1.0 de Java a d'abord inclus les classes **java.util.Vector** et **java.util.Stack**.
- La classe **Vector** permet de représenter une liste d'éléments comme la classe **ArrayList**.
- La classe **Stack** qui hérite de **Vector** permet de représenter des piles de type **LIFO** (Last In First Out).
- Ces deux classes sont toujours présentes dans l'API pour des raisons de compatibilité ascendante mais il ne faut surtout pas s'en servir. En effet, ces classes utilisent des mécanismes de synchronisation internes dans le cas où elles sont utilisées pour des accès concurrents (programmation parallèle ou multithread).
- Or, non seulement ces mécanismes de synchronisation pénalisent les performances mais en plus, ils se révèlent largement inefficaces pour gérer les accès concurrents (il existe d'autres façons de faire en Java).
- Les classes **ArrayList** et **ArrayDeque** se substituent très bien aux classes **Vector** et **Stack**.

2. Les ensembles : Set

- Les ensembles (*set*) sont des collections qui ne contiennent aucun doublon. Deux élements e1 et e2 sont des doublons si : `e1.equals(e2) == true`
- Il existe également un **EnumSet** qui représente un ensemble d'énumérations. Son implémentation est très compacte et très performante mais n'est utilisable que pour des énumérations.

Les ensembles : TreeSet

- La classe **TreeSet** contrôle l'unicité de ces éléments en maintenant en interne une liste triée par ordre naturel des éléments. L'ordre peut être donné soit parce que les éléments implémentent l'interface **Comparable** soit parce qu'une implémentation de **Comparator** est passée en paramètre de constructeur au moment de la création de l'instance de **TreeSet**.
- La classe **TreeSet** a donc comme particularité de toujours conserver ses éléments triés.

Les ensembles : TreeSet

```
Set<String> ensemble = new TreeSet<String>();
ensemble.add("élément");
ensemble.add("élément");
ensemble.add("élément");
ensemble.add("élément");
System.out.println(ensemble.size()); // 1
ensemble.remove("élément");
System.out.println(ensemble.isEmpty()); // true
```

Les ensembles : HashSet

- La classe **HashSet** utilise un code de hachage (hash code) pour contrôler l'unicité de ces éléments.
- Un code de hachage est une valeur (index) générée automatiquement et associée à l'objet.
- Deux objets identiques doivent obligatoirement avoir le même code de hachage. Par contre deux objets distincts ont des codes de hachage qui peuvent être soit différents soit identiques.
- Un ensemble d'éléments différents mais qui ont néanmoins le même code de hachage forment un **bucket**.
- La classe HashSet maintient en interne un tableau associatif entre une valeur de hachage et un bucket.
- Lorsqu'un nouvel élément est ajouté au HashSet, ce dernier calcule son code de hachage et vérifie si cette valeur a déjà été stockée. Si c'est le cas, alors les éléments du bucket associé sont parcourus un à un pour vérifier s'ils sont identiques ou non au nouvel élément.

Les ensembles : HashSet

- L'implémentation de la classe **HashSet** a des performances en temps très supérieures à **TreeSet** pour les opérations d'ajout et de suppression d'élément. Elle impose néanmoins que les éléments qu'elle contient génèrent correctement un code de hachage avec la méthode **hashCode**.
- Contrairement à **TreeSet**, elle ne garantit pas l'ordre dans lequel les éléments sont stockés et donc l'ordre dans lequel ils peuvent être parcourus.

Les ensembles : HashSet

```
Set<String> ensemble = new HashSet<String>();
ensemble.add("élément");
ensemble.add("élément");
ensemble.add("élément");
ensemble.add("élément");
System.out.println(ensemble.size()); // 1
ensemble.remove("élément");
System.out.println(ensemble.isEmpty()); // true
```

Les ensembles : LinkedHashSet

- La classe **LinkedHashSet**, comme la classe **HashSet**, utilise en interne un code de hachage mais elle garantit en plus que l'ordre de parcours des éléments sera le même que l'ordre d'insertion.
- Cette implémentation garantit également que si elle est créée à partir d'un autre **Set**, l'ordre des éléments sera maintenu.
- La classe **LinkedHashSet** a été créée pour réaliser un compromis entre la classe **HashSet** et la classe **TreeSet** afin d'avoir des performances proches de la première tout en offrant l'ordre de parcours pour ses éléments.

Les ensembles : LinkedHashSet

```
Set<String> ensemble = new LinkedHashSet<String>();
ensemble.add("premier élément");
ensemble.add("premier élément");
ensemble.add("premier élément");
ensemble.add("premier élément");
ensemble.add("deuxième élément");
ensemble.add("premier élément");
ensemble.add("troisième élément");
ensemble.add("premier élément");


// [premier élément, deuxième élément, troisième élément]
System.out.println(ensemble);
```

3. Les tableaux associatifs : Maps

- Un tableau associatif (parfois appelé dictionnaire) ou **Map** permet d'associer une clé à une valeur.
- Un tableau associatif ne peut pas contenir de doublon de clés.
- Les classes et les interfaces représentant des tableaux associatifs sont génériques et permettent de spécifier un type pour la clé et un type pour la valeur.
- Le **Java Collections Framework** fournit plusieurs implémentations de tableaux associatifs : **TreeMap**, **HashMap**, **LinkedHashMap**.
- La classe **EnumMap** représente un tableau associatif dont les clés sont des énumérations. Son implémentation est très compacte et très performante mais n'est utilisable que pour des clés de type énumération.

Les tableaux associatifs : TreeMap

- La classe **TreeMap** est basée sur l'implémentation d'un arbre bicolore pour déterminer si une clé existe ou non dans le tableau associatif.
- Elle dispose d'une bonne performance en temps pour les opérations d'accès, d'ajout et de suppression de la clé.
- Cette classe contrôle l'unicité et l'accès à la clé en maintenant en interne une liste triée par ordre naturel des clés.
- L'ordre peut être donné soit parce que les éléments implémentent l'interface Comparable soit parce qu'une implémentation de Comparator est passée en paramètre de constructeur au moment de la création de l'instance de TreeMap.
- La classe TreeMap a donc comme particularité de conserver toujours ses clés triées.

Les tableaux associatifs : TreeMap

```
Map<String, Integer> tableauAssociatif = new TreeMap<>();
tableauAssociatif.put("un", 1);
tableauAssociatif.put("deux", 2);
tableauAssociatif.put("trois", 3);

System.out.println(tableauAssociatif.get("deux")); // 2

int resultat = 0;
for (String s : "un deux trois".split(" ")) {
      resultat += tableauAssociatif.get(s);
}

System.out.println(resultat); // 6

tableauAssociatif.remove("trois");
tableauAssociatif.put("deux", 1000);

System.out.println(tableauAssociatif.keySet()); // [deux, un]
System.out.println(tableauAssociatif.values()); // [1000, 1]
```

Les tableaux associatifs : HashMap

- La classe **HashMap** utilise un code de hachage (hash code) pour contrôler l'unicité et l'accès aux clés. Un code de hachage est une valeur associée à un objet.
- Deux objets identiques doivent obligatoirement avoir le même code de hachage. Par contre deux objets distincts ont des codes de hachage qui peuvent être soit différents soit identiques.
- Un ensemble de clés différentes mais qui ont néanmoins le même code de hachage forment un bucket.
  - La classe HashMap maintient en interne un tableau associatif entre une valeur de hachage et un bucket.
  - Lorsqu'une nouvelle clé est ajoutée au HashMap, ce dernier calcule son code de hachage et vérifie si ce code a déjà été stocké.
  - Si c'est le cas, alors la valeur passée remplace l'ancienne valeur associée à cette clé.
  - Sinon la nouvelle clé est ajoutée avec sa valeur.
- L'implémentation de la classe HashSet a des performances en temps supérieures à TreeSet pour les opérations d'ajout et d'accès. Elle impose néanmoins que les éléments qu'elle contient génèrent correctement un code de hachage avec la méthode hashCode. Contrairement à la classe TreeMap, elle ne garantit pas l'ordre dans lequel les clés sont stockées et donc l'ordre dans lequel elles peuvent être parcourues.

Les tableaux associatifs : HashMap

```
Map<String, Integer> tableauAssociatif = new HashMap<>();
tableauAssociatif.put("un", 1);
tableauAssociatif.put("deux", 2);
tableauAssociatif.put("trois", 3);

System.out.println(tableauAssociatif.get("deux")); // 2


int resultat = 0;
for (String s : "un deux trois".split(" ")) {
        resultat += tableauAssociatif.get(s);
}
System.out.println(resultat); // 6


tableauAssociatif.remove("trois");
tableauAssociatif.put("deux", 1000);

System.out.println(tableauAssociatif.keySet()); // [deux, un]
System.out.println(tableauAssociatif.values()); // [1, 1000]
```

Les tableaux associatifs : LinkedHashMap

- La classe **LinkedHashMap**, comme la classe **HashMap**, utilise en interne un code de hachage mais elle garantit en plus que l'ordre de parcours des clés sera le même que l'ordre d'insertion.
- Cette implémentation garantit également que si elle est créée à partir d'une autre **Map**, l'ordre des clés sera maintenu.
- La classe **LinkedHashMap** a été créée pour réaliser un compromis entre la classe **HashMap** et la classe **TreeMap** afin d'avoir des performances proches de la première tout en offrant l'ordre de parcours pour ses clés.

Les tableaux associatifs : LinkedHashMap

```
Map<String, Integer> tableauAssociatif = new
                                LinkedHashMap<>();
tableauAssociatif.put("rouge", 0xff0000);
tableauAssociatif.put("vert", 0x00ff00);
tableauAssociatif.put("bleu", 0x0000ff);

// affichera : rouge puis vert puis bleu
for (String k: tableauAssociatif.keySet()) {
    System.out.println(k);
}
```

En résumé

- Les listes :
  - Interface pour des objets qui autorisent des doublons et un accès direct à un élément
- Les ensembles :
  - Interface qui n'autorise pas des doublons dans l'ensemble
- Les tableaux associatifs :
  - Interface qui définit des objets qui gères des collections sous la forme clé/valeur

Opérations de tri : les méthodes statiques sort

- Tri des tableaux de type primitifs
  - Classe Arrays
- Tri des Objets
  - Classe Arrays et Collection
  - Utilisable sur les collections d'instances de classe implémentant l'interface Comparable

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

- e1.compareTo(e2) retourne
  - un int < 0 si e1<e2
  - 0 si e1=e2
  - un int > 0 si e1>e2

Tri suivant un ordre quelconque

- Si vous souhaitez trier des éléments issus d'une classe n'implémentant pas **Comparable** ou si vous voulez effectuer un tri suivant une logique différente de celle impliquée par **Comparable**, tri dit naturel, vous devez utiliser un **Comparator**
- Un **Comparator** est un objet qui encapsule une relation d'ordre

```
public interface Comparator<T> {
       int compare(T o1, T o2);
} //retourne un entier négatif, Zéro ou positif selon que o1 est
  //plus petit, égal ou plus grand que o2. Si le type d'un des
  //arguments est impropre il y a une ClassCastException.
```