Classes et Interfaces utiles en Java

Documentation Java

- Sur le Web :
  - Java 7 (ou 1.7) : http://docs.oracle.com/javase/7/docs/api/
  - Java 8 (ou 1.8) : http://docs.oracle.com/javase/8/docs/api/

La classe Object

- En Java la classe **Object** est la classe mère de toutes les classes
  - Toute nouvelle classe dérive implicitement de la classe Object : Arbre d'héritage.
  - La classe Object peut donc être utilisée pour la généricité grâce au polymorphisme d'héritage.
    - Exemple : la classe Vector (avant le JDK1.5) stockait une collection de références vers des Object.
      - Avantages :
        - » généricité simple à gérer.
      - Inconvénients :
        - » Hétérogénéité possible des collections
        - » Obligation d'utiliser l'introspection pour contrôler les types si l'on souhaite n'avoir qu'un seul type d'instances dans une collection.
  - La classe Object ne contient pas de données mais définit un certain nombres de méthodes utiles.
    - Ces méthodes peuvent et/ou doivent être surchargées par les classes filles.

Les méthodes de la classe Object

- **`protected Object clone()`**
  - Pour créer une copie de soi-même au sens du contenu (deep copy)
  - `x.clone() == x` ⇒ false (deux instances distinctes)
  - `x.clone().equals(x)` ⇒ true (deux instances identiques en terme de contenu)
- **`public boolean equals(Object o)`**
  - Pour tester l'égalité de soi-même avec une autre instance
    - Au sens du contenu (deep compare) et non plus seulement au sens des références (shallow compare).
  - Réflexivité : `x.equals(x)` ⇒ true; (au sens du contenu)
  - Symétrie : `x.equals(y)` == true ⇒ `y.equals(x)` (si x et y sont de même type effectif et que leur contenu est indentique)
  - Transitivité : si `x.equals(y)` == true et `y.equals(z)` == true alors `x.equals(z)` == true.

Les méthodes de la classe Object

- **`protected void finalize() throws Throwable`**
  - Permet le nettoyage de l'instance avant garbage collecting : appelée par le garbage collector quand il n'y a plus de références pointant vers cette instance.
- **`public Class<? extends Object> getClass()`**
  - Permet d'obtenir une instance de la classe Class correspondant au type effectif de l'instance interrogée (Introspection).
- **`public int hashCode()`**
  - Renvoie un code de hashage (unique) pour l'objet au sens du contenu.
  - Utilisé dans les `class Hashtable<K,V>`
  - Si `x.equals(y)==true` alors `x.hashcode()==y.hashCode()`.

Les méthodes de la classe Object

- **`public String toString()`**
  - Permet de créer une chaîne de caractères représentant l'objet.
  - Utilisé par les méthodes d'affichages comme `println`

```
Object o = new Object();
System.out.println(o);
```

Appel implicite à `o.toString()`

- Méthodes spécifiques utilisées par les Threads
  - `public final void notify()`
  - `public final void notifyAll()`
  - `public final void wait()`
  - `public final void wait(long timeout)`
  - `public final void wait(long timeout, int nanos)`

La méthode equals

- Algorithme `boolean equals(Object o)`
  - Si `o == null` ⇒ renvoyer false
  - Si `this == o` ⇒ renvoyer true
  - Utiliser `instanceof` ou `getClass` pour vérifier que o est du bon type
    - Caster o dans le type voulu: `Type oCast` = (`Type`)o;
      - Comparer les attributs
        - » Types simples sauf float et double : ==
        - » float : utiliser `Float.floatToIntBits` avant de comparer avec ==
        - » double : utiliser `Double.doubleToLongBits` avant de comparer avec ==
        - » Autres : utiliser la méthode equals sur chaque attribut

```
(attribut == null ? oCast.attribut == null :
  attribut.equals(oCast.attribut))
```

A cause de `[Float|Double].NaN` et -0.0f

La relation `equals` ↔ `hashCode` (1/2)

- Le contrat de `equals` :
  - Réflexivité : pour un x non null x.equals(x) == true
  - Symétrie : si x.equals(y) == true alors y.equals(x) aussi
  - Transitivité : si x.equals(y) et y.equals(z) alors x.equals(z)
  - Cohérence (temporelle) : si x.equals(y) ils doivent le rester indéfiniment tant que l'un des deux n'est pas modifié.
  - Non-nullité : pour toute référence x non nulle x.equals(null) doit être faux : un objet non null (x) sur lequel on peut déclencher la méthode equals ne peut pas être null !

La relation `equals` ↔ `hashCode` (2/2)

- Le contrat de `hashCode` :
  - Cohérence (temporelle) : la valeur retournée par `hashCode` pour un objet doit rester la même tout au long de l'exécution d'une application java tant que les éléments utilisés pour la comparaison dans `equals` n'ont pas changé.
  - Si deux objets sont égaux au sens de « `equals` », alors ils doivent renvoyer le même `hashCode`.
  - Il n'est pas requis que deux objets différents au sens du `equals` renvoie deux `hashCode` différents, **néanmoins** cela permet d'accroitre les performances des tables de hachage utilisées dans les `HashMap` par exemple.

- Conclusion : si on réimplémente `equals` on doit aussi réimplémenter `hashCode`.

La méthode hashCode

- Algorithme `int hashCode()`

```
public int hashCode()
{
    final int prime = 31;
    int hash = 1;
    hash = prime * hash + (valBool ? 1231 : 1237);
    hash = prime * hash + valChar;
    long dtlb = Double.doubleToLongBits(valDouble);
    hash = prime * hash + (int) (dtlb ^ (dtlb >>> 32));
    hash = prime * hash + Float.floatToIntBits(valFloat);
    hash = prime * hash + valInt;
    hash = prime * hash
           + ((valObject == null) ? 0 : valObject.hashCode());
    :
    return hash;
}
```

unsigned right shift operator

- ~~Collections : Somme des hashCodes des éléments~~

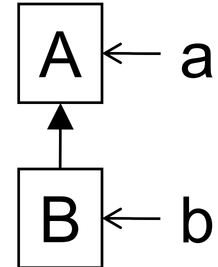
La classe Class<T>

- Les instances de la classe Class représentent les classes et les interfaces dans une application Java en cours d'exécution.
  - Toute classe ou interface instanciée dans une application a donc sa propre instance de la classe Class et celle-ci est **unique**.
  - Exemple :

```
MonObjet o = new MonObject();
Integer e = new Integer(3);
Class classe = o.getClass();
classe.equals(e.getClass()) ⇒ false
```

  - Les types simples (`boolean, byte, char, short, int, long, float, double`), et le mot clé `void` peuvent aussi être représentés par des instances de la classe Class.
  - Obtention des instances de la classe Class
    - Pas de constructeurs
    - Au travers de la méthode `getClass()` de la classe Object
    - Au travers du literal « class » : Type`.class`;
      `Integer.class == e.getClass()` ⇒ true

Utilisation de la classe Class



- A partir du langage lui-même
  - `if (b instanceof A) {…}` ⇒ true
  - `if (a instanceof B) {…}` ⇒ false
- En utilisant les méthodes de la classe Class
  - ```
    Class ca = a.getClass();
    Class cb = b.getClass();
    if (ca.equals(cb)) {
      System.out.println("de la même classe " + ca.get[Simple]Name());
    }
    ```
    ⇒ false
  - `if (ca.isInstance(b)) {…}` ⇒ true
  - `if (B.class.isInstance(a)) {…}` ⇒ false
  - ```
    Object monObject = new Object();
    if (Class.forName("java.lang.Object").isInstance(monObject))
    ```
    ⇒true

Utilisation de la classe Class

- Chaque instance de la classe Class correspondant aux types instanciés à un instant de l'exécution est **unique** :
  - ```
    Integer a = new Integer(…);
    Integer b = new Integer(…);
    Class ca = a.getClass();
    Class cb = b.getClass();

    if((ca == cb) == (ca.equals(cb))) {…}
    ```
    ⇒ true
    ```

Le package java.lang

- Interfaces
  - *Appendable*
    - xxxWriter(s), …, PrintStream, StringBuffer, StringBuilder, …
  - *CharSequence*
    - CharBuffer, String, StringBuffer, StringBuilder
  - *Cloneable*
  - *Comparable<T>*
    - Toutes les classes présentant des relations d'ordre (<) : p.ex. Integer, Boolean, Double, etc
  - *Iterable<T>*
    - *Toutes les interfaces des collections*
    - Toutes les collections
  - *Readable*
    - xxxReader(s),
  - *Runnable*

- Classes
  - Types scalaires : Number et classes filles : Byte, Boolean, Character, Double, Float, Integer, Long…
  - Class<T>
    - Introspection
  - Math
    - Fonctions mathématiques
  - String, String[Buffer|Builder]
    - Chaînes de caractères
  - System
  - Thread

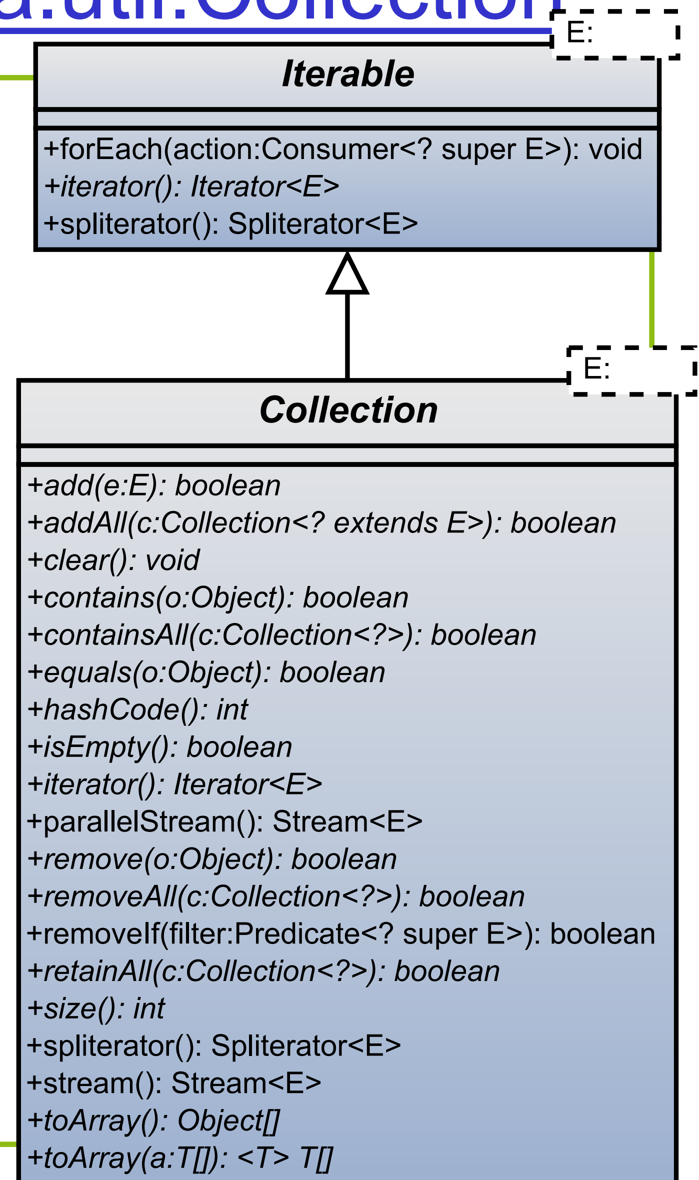
Le package java.util

- Interfaces
  - *Collection<E>*
    - *List<E>, Queue<E>, Set<E>*
    - AbstractCollection<E>, …, Vector<E>
  - *Comparator<E>*
    - *comparaison*
  - *Enumeration<E>*
    - Énumération des éléments d'une collection
  - *Iterator<E>*
    - Itération sur une collection
  - *List<E>*
    - *AbstractList, ArrayList, …*
  - *Map<K,V>*
    - *AbstractMap, HashMap, TreeMap, HashTable*
  - *Queue<E>*
    - File d'attente
  - *Set<E>*
    - Ensemble d'éléments sans doublons

- Classes
  - Implémentations des interfaces
    - Abstract[Collection,List,Queue]<E>
    - [AbstractMap,Dictionary]<K,V>
  - Collections
  - Calendar, Date
  - Formatter
  - HashXxx
    - Hash[Map<K,V>,Set<E>,table<K,V>]
  - LinkedXxx
    - Linked[HashMap<K,V>, HashSet<E>, List<E>]
  - Locale
  - Observable
  - Stack<E>, Vector<E>

Interface Collection : java.util.Collection

- Interface mère de toutes les collections en Java
- Différentes implémentations autorisant (ou pas) :
  - l'insertion de doublons (evt avec UnsupportedOperationException)
  - l'insertion d'éléments null (evt avec NullPointerException)
  - les opérations concurrentes (par plusieurs thread)
- La recherche d'éléments se fait implicitement par l'utilisation de la méthode « equals » plutôt que « == »



Parcours de collections

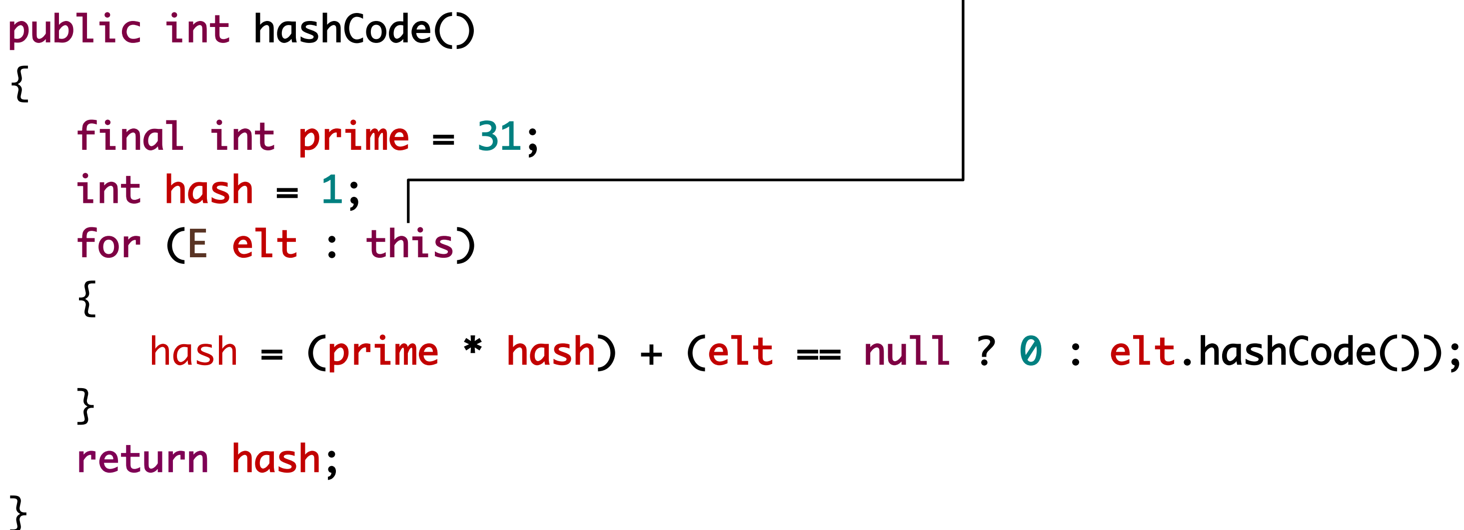
```
// Exemple sur un Vector
Vector<E> elts = new Vector<E>();
// Ancienne forme de parcours
for (int i = 0; i < elts.size(); i++)
{
    … elts.elementAt(i) … ;
}
// Parcours par énumération (obsolète)
for (Enumeration<E> en = elts.elements(); en.hasMoreElements(); )
{
    … en.nextElement() … ;
}
// Parcours par itérateur
for (Iterator<E> it = elts.iterator(); it.hasNext();)
{
    … it.next() … ;
}
// Parcours par boucle "foreach" (utilise implicitement l'itérateur)
for (E elt : elts) // elts doit être Iterable<E>
{
        … elt … ;
}
```

Ou plus exactement des Iterable<E>

Les tableaux sont aussi itérables

La méthode hashCode pour les collections

- HashCode des Collection<E>
  - Plus exactement des Iterable<E>
    - qui fournissent la méthode
      `public Iterator<E> iterator()`



```java
public int hashCode()
{
   final int prime = 31;
   int hash = 1;
   for (E elt : this)
   {
      hash = (prime * hash) + (elt == null ? 0 : elt.hashCode());
   }
   return hash;
}
```

Attention, le hashage dépend ici de l'ordre des éléments : ne conviendra pas pour la comparaison de collections dans lesquelles l'ordre des élément n'a pas de sens : on utilisera alors la somme des hashcodes.

http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Les flux : java.util.stream.Stream<T>

- Stream pipeline

- Sources
  - Collection
    - `c.stream()`
  - Arrays
    - `Arrays.stream(T[])`
  - Générateur (méthode)
    - `generate`
    - `iterate`
    - `of(T…)`
  - I/O Channel

- Opération(s) intermédiaire(s)
  - *Opérations dont le résultat est un flux*

- Opération terminale
  - *Opération produisant un résultat ou un effet de bord*

- Collections != Streams
  - Les collections sont destinées à la gestion et à l'accès à leurs éléments
  - Les flux sont destinés à la description déclarative de leur source et surtout aux **opérations** qui seront effectuées sur l'ensemble d'une telle source.

≠ java.io.InputStream
OutputStream

Opérations sur les flux

Source | Opération intermédiaire | Opération Terminale

- Les paramètres des opérations sur ces flux sont des instances de « functional interface » qui peuvent être des *lambda-expressions* ou des *références de méthodes*.
- Exemples

```java
CollectionListe<String> c = …
// [Coucou, Aujourd'hui, Bonjour, non, ?]

Predicate<String> capitalized = (String s) ->
{
   if (s.length() > 0)
      return Character.isUpperCase(s.charAt(0));
   return false;
};

boolean isC = c.stream().allMatch(capitalized);

isC = c.stream()
       .filter(capitalized).allMatch(capitalized);

String concat = c.stream()
   .collect(StringBuilder::new,
            StringBuilder::append,
            StringBuilder::append)
   .toString();

long nbMots = c.stream().count();
```

faux

vrai

Références de méthodes

```java
Integer nbLetters1 = c.stream()
                      .mapToInt(String::length)
                      .sum();
Integer nbLetters2 = c.stream()
                      .map(String::length)
                      .reduce(0, (a, b) -> (a + b));

Consumer<String> sPrinter =
   (String s) -> System.out.println(s);
c.stream()
 .filter(capitalized)
 .forEach(sPrinter);

Function<String, String> lowerCaser = (String s) ->
{
   return s.toLowerCase();
};
c.stream().map(lowerCaser).forEach(sPrinter);

Set<String> dictionary = c.stream()
   .distinct()
   .sorted(String::compareToIgnoreCase)
   .collect(Collectors.toSet());
dictionary.stream().forEach(sPrinter);
```

28

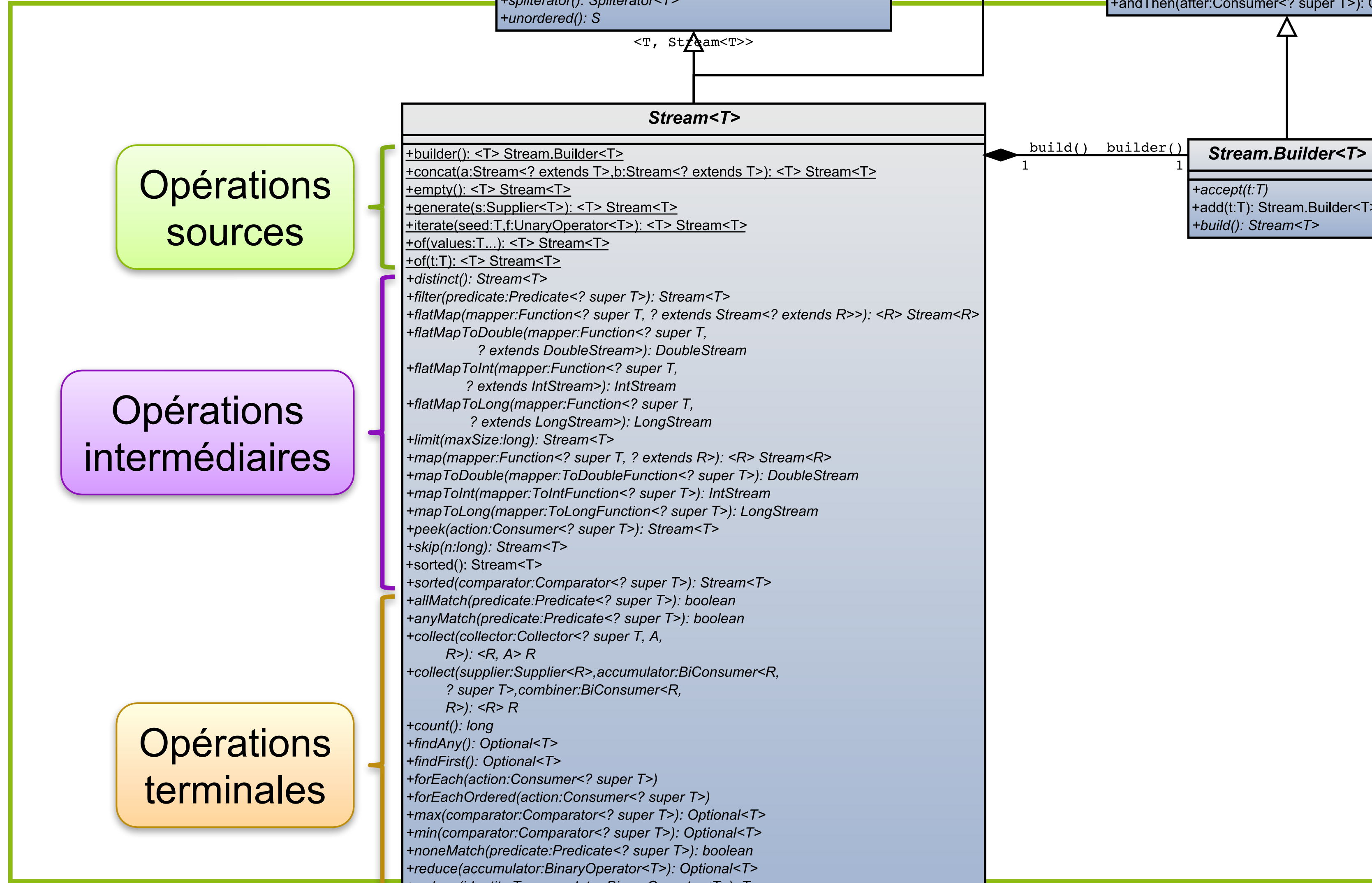
Références de méthodes

```
Coucou
Aujourd'hui
Bonjour
```

```
coucou
aujourd'hui
bonjour
non
?
```

```
Aujourd'hui
Bonjour
Coucou
non
?
```

Stream<T>



Interfaces fonctionnelles : java.util.function

Function<T, R>

+apply(t:T): R
+andThen(after:Function<? super R, ? extends V>): <V> Function<T, V>
+compose(before:Function<? super V, ? extends T>): <V> Function<V, R>
+identity(): Function<T, T>

- Ensemble d'interfaces fonctionnelles* utilisables, entre autres, sur les flux éventuellement avec des Lambda expressions ou des références de méthodes.
- Function<T, R> : R apply(T t)
  - UnaryOperator<T>
- BiFunction<T, U, R> : R apply(T t, U u)
  - BinaryOperator<T>
- Predicate<T> : boolean test(T t)
- BiPredicate<T, U> : boolean test(T t, U u)
- Supplier<T> : T get()
- Consumer<T> : void accept(T t)
- ...

* : une seule méthode abstraite + [méthodes de classe] + [méthodes par défaut]

Optionel : java.util.optional

- Conteneur pouvant contenir (ou pas) une valeur non null
  - Pas de constructeur
  - Si une valeur non null est présente
    - `isPresent()` est `true`
    - `get()` fournit cette valeur

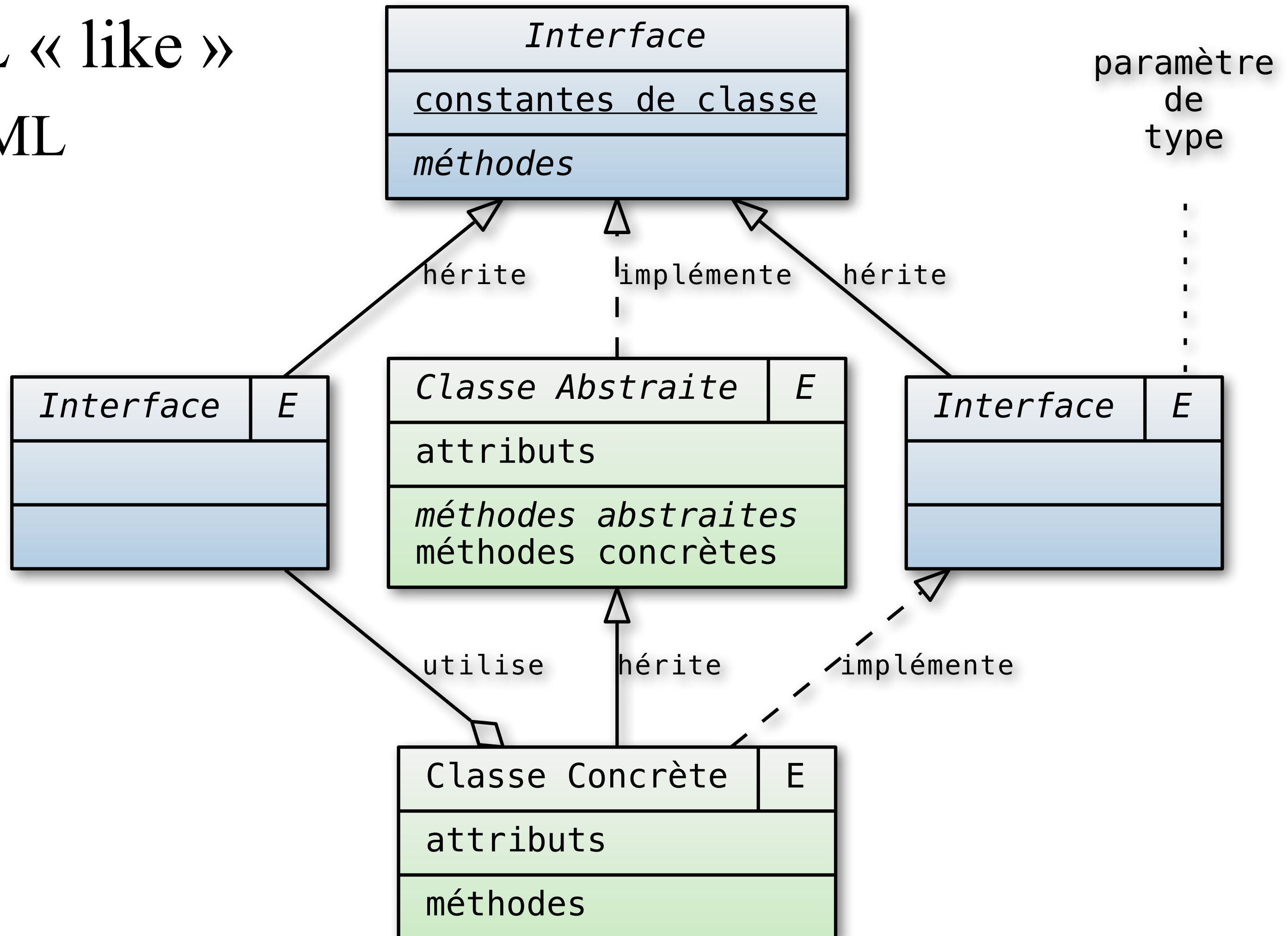
T:

Optional

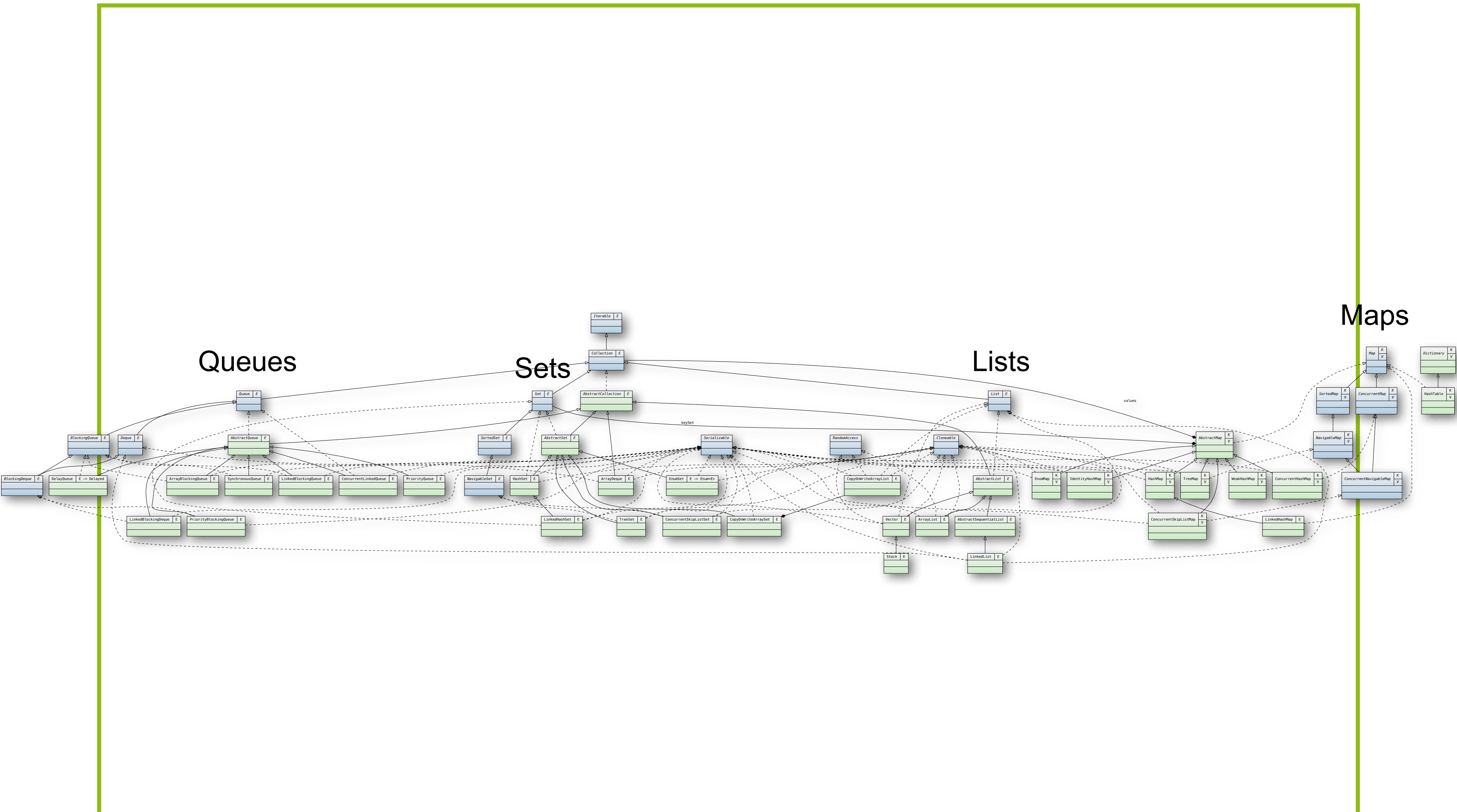
+empty(): <T> Optional<T>
+equals(o:Object): boolean
+filter(predicate:Predicate<? super T>): Optional<T>
+flatMap(mapper:Function<? super T, Optional<U>>): <U> Optional<U>
+get(): T
+hashCode(): int
+ifPresent(consumer:Consumer<? super T>)
+isPresent(): boolean
+map(mapper:Function<? super T, ? extends U>): <U> Optional<U>
+of(value:T): <T> Optional<T>
+ofNullable(value:T): <T> Optional<T>
+orElse(other:T): T
+orElseGet(other:Supplier<? extends T>): T
+orElseThrow(exceptionSupplier:<? extends X>): <X extends Throwable> T
+toString(): String

Notations graphes de classes

- UML « like »
  ≠ UML

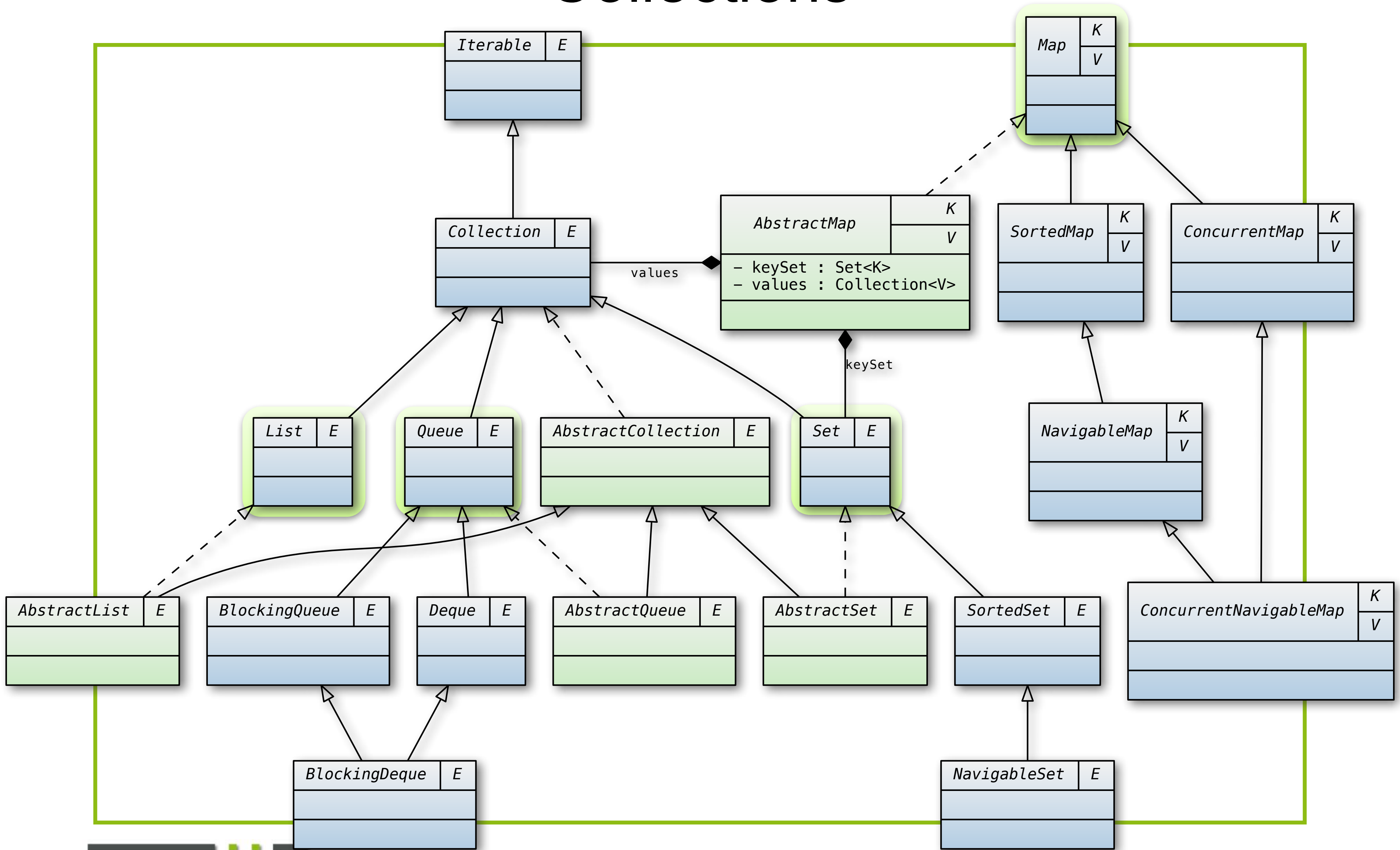


Java Collections Framework

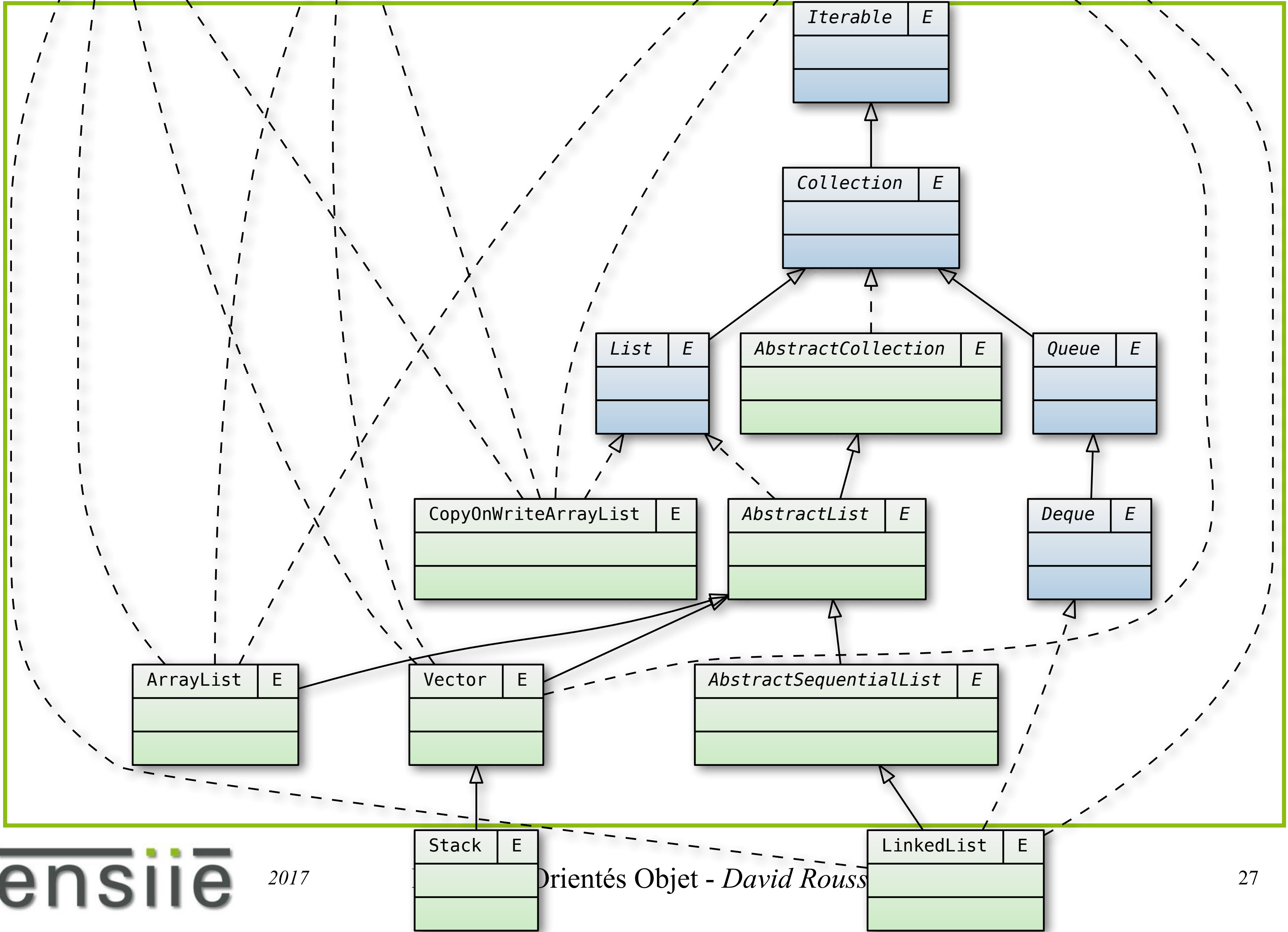


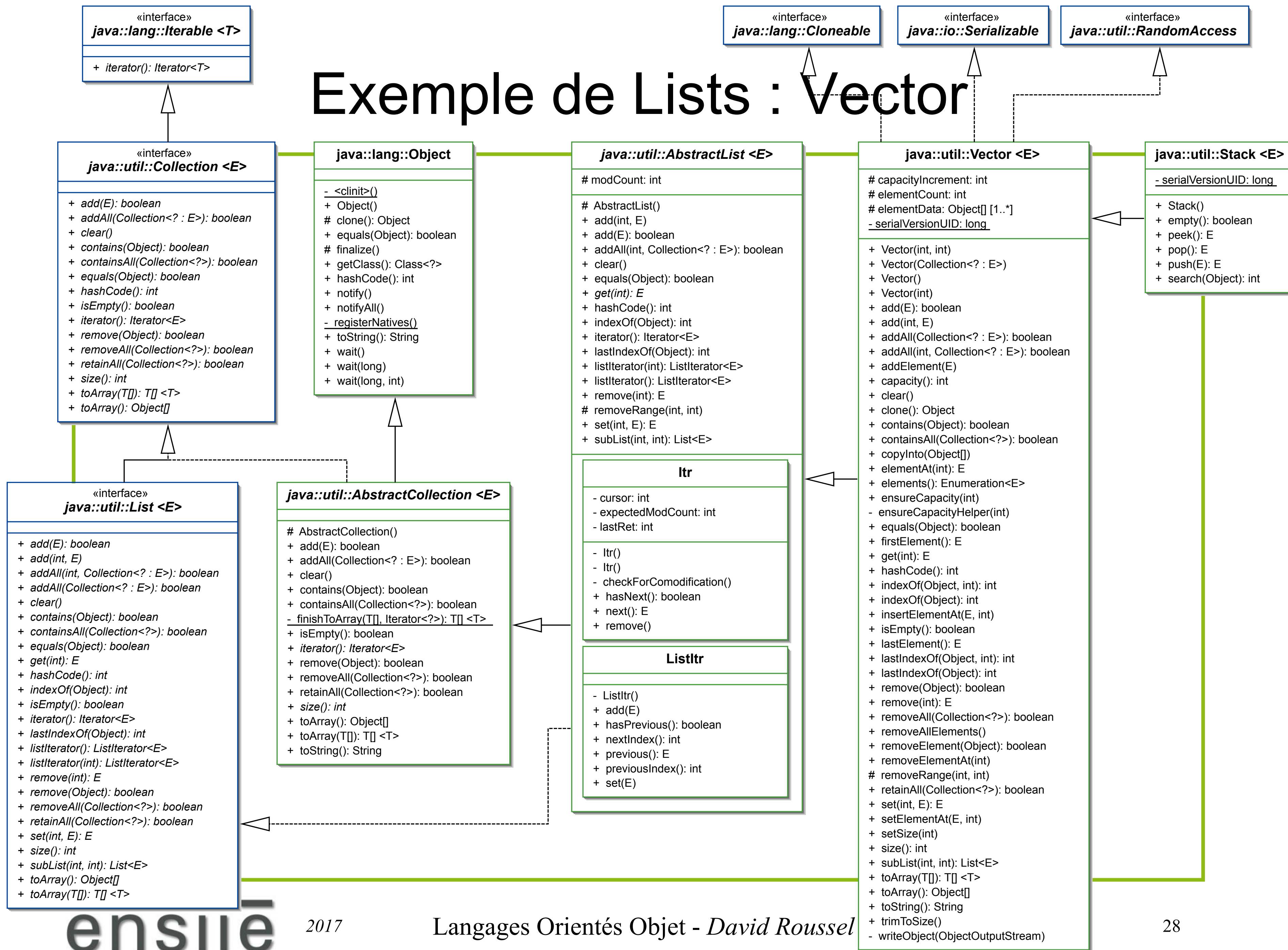
http://docs.oracle.com/javase/8/docs/technotes/guides/collections/reference.html

Collections



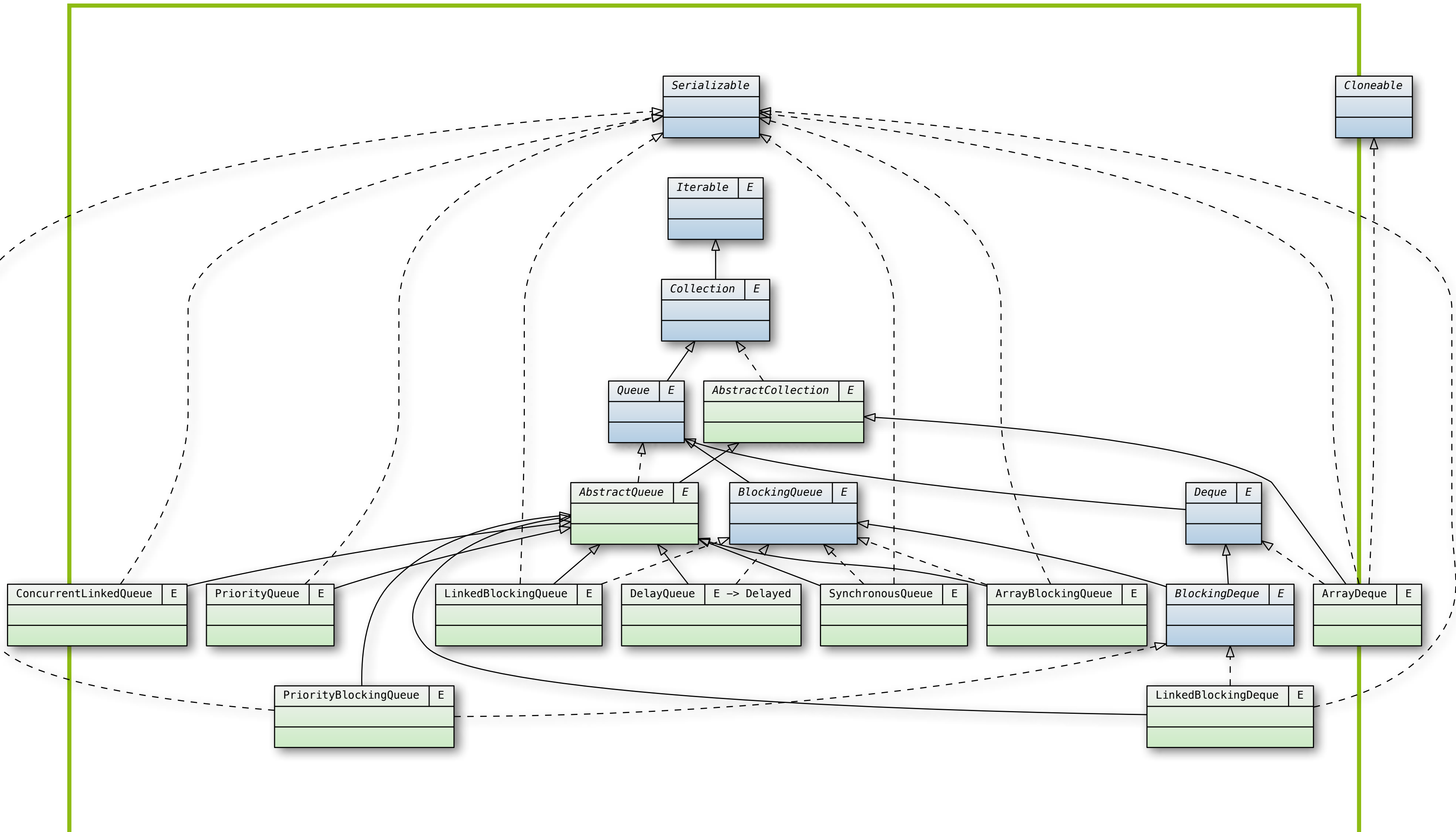
Lists : séquences



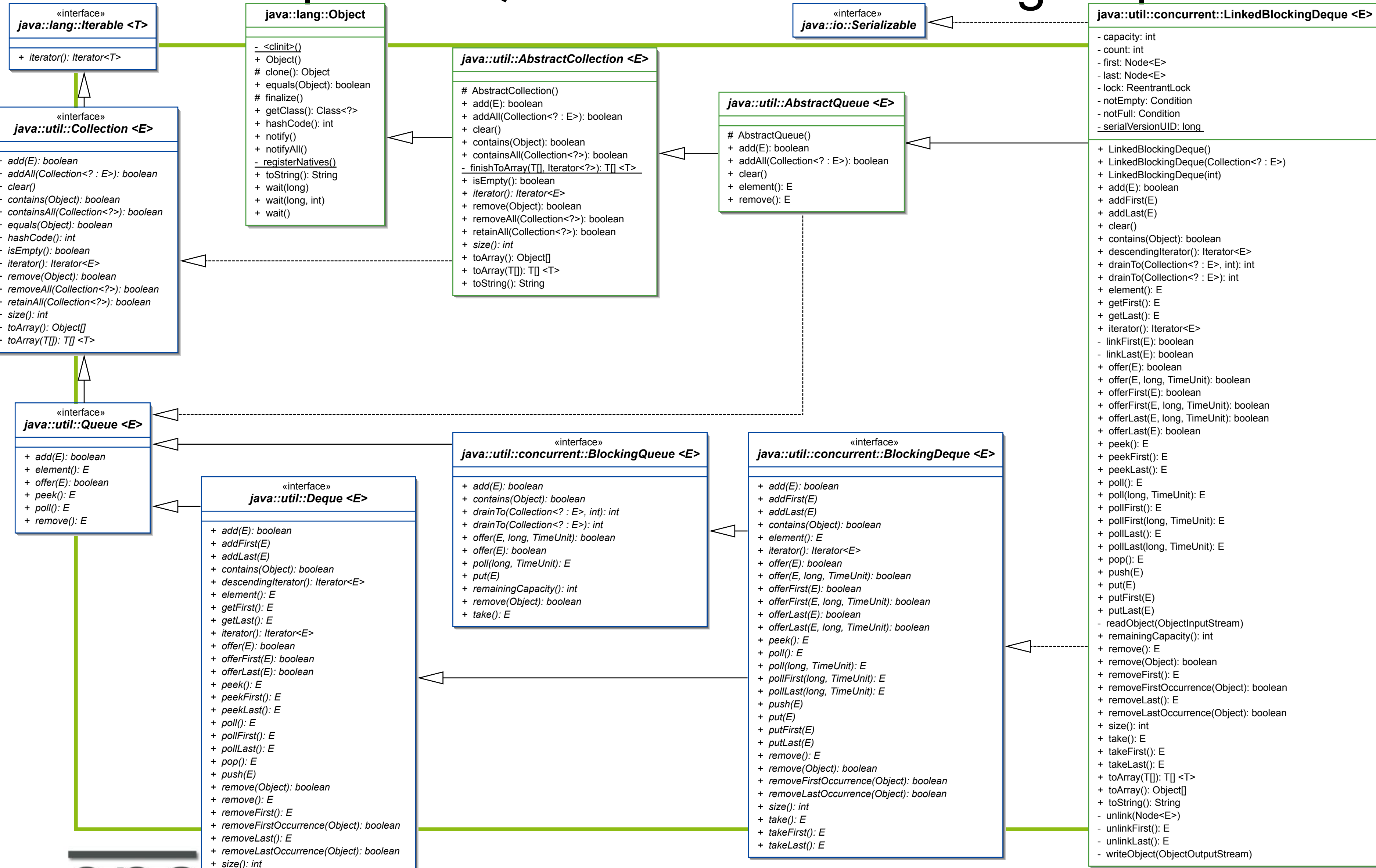


Exemple de Lists : Vector
«interface»
java::lang::Iterable <T>
+ iterator(): Iterator<T>
«interface»
java::lang::Cloneable
«interface»
java::io::Serializable
«interface»
java::util::RandomAccess
«interface»
java::util::Collection <E>
+ add(E): boolean
+ addAll(Collection<? : E>): boolean
+ clear()
+ contains(Object): boolean
+ containsAll(Collection<?>): boolean
+ equals(Object): boolean
+ hashCode(): int
+ isEmpty(): boolean
+ iterator(): Iterator<E>
+ remove(Object): boolean
+ removeAll(Collection<?>): boolean
+ retainAll(Collection<?>): boolean
+ size(): int
+ toArray(T[]): T[] <T>
+ toArray(): Object[]
java::lang::Object
- <clinit>()
+ Object()
# clone(): Object
+ equals(Object): boolean
# finalize()
+ getClass(): Class<?>
+ hashCode(): int
+ notify()
+ notifyAll()
- registerNatives()
+ toString(): String
+ wait()
+ wait(long)
+ wait(long, int)
java::util::AbstractList <E>
modCount: int
AbstractList()
+ add(int, E)
+ add(E): boolean
+ addAll(int, Collection<? : E>): boolean
+ clear()
+ equals(Object): boolean
+ get(int): E
+ hashCode(): int
+ indexOf(Object): int
+ iterator(): Iterator<E>
+ lastIndexOf(Object): int
+ listIterator(int): ListIterator<E>
+ listIterator(): ListIterator<E>
+ remove(int): E
# removeRange(int, int)
+ set(int, E): E
+ subList(int, int): List<E>
Itr
- cursor: int
- expectedModCount: int
- lastRet: int
- Itr()
- Itr()
- checkForComodification()
+ hasNext(): boolean
+ next(): E
+ remove()
ListItr
- ListItr()
+ add(E)
+ hasPrevious(): boolean
+ nextIndex(): int
+ previous(): E
+ previousIndex(): int
+ set(E)
java::util::Vector <E>
capacityIncrement: int
elementCount: int
elementData: Object[] [1..*]
- serialVersionUID: long
+ Vector(int, int)
+ Vector(Collection<? : E>)
+ Vector()
+ Vector(int)
+ add(E): boolean
+ add(int, E)
+ addAll(Collection<? : E>): boolean
+ addAll(int, Collection<? : E>): boolean
+ addElement(E)
+ capacity(): int
+ clear()
+ clone(): Object
+ contains(Object): boolean
+ containsAll(Collection<?>): boolean
+ copyInto(Object[])
+ elementAt(int): E
+ elements(): Enumeration<E>
+ ensureCapacity(int)
- ensureCapacityHelper(int)
+ equals(Object): boolean
+ firstElement(): E
+ get(int): E
+ hashCode(): int
+ indexOf(Object, int): int
+ indexOf(Object): int
+ insertElementAt(E, int)
+ isEmpty(): boolean
+ lastElement(): E
+ lastIndexOf(Object, int): int
+ lastIndexOf(Object): int
+ remove(Object): boolean
+ remove(int): E
+ removeAll(Collection<?>): boolean
+ removeAllElements()
+ removeElement(Object): boolean
+ removeElementAt(int)
# removeRange(int, int)
+ retainAll(Collection<?>): boolean
+ set(int, E): E
+ setElementAt(E, int)
+ setSize(int)
+ size(): int
+ subList(int, int): List<E>
+ toArray(T[]): T[] <T>
+ toArray(): Object[]
+ toString(): String
+ trimToSize()
- writeObject(ObjectOutputStream)
java::util::Stack <E>
- serialVersionUID: long
+ Stack()
+ empty(): boolean
+ peek(): E
+ pop(): E
+ push(E): E
+ search(Object): int
«interface»
java::util::List <E>
+ add(E): boolean
+ add(int, E)
+ addAll(int, Collection<? : E>): boolean
+ addAll(Collection<? : E>): boolean
+ clear()
+ contains(Object): boolean
+ containsAll(Collection<?>): boolean
+ equals(Object): boolean
+ get(int): E
+ hashCode(): int
+ indexOf(Object): int
+ isEmpty(): boolean
+ iterator(): Iterator<E>
+ lastIndexOf(Object): int
+ listIterator(): ListIterator<E>
+ listIterator(int): ListIterator<E>
+ remove(int): E
+ remove(Object): boolean
+ removeAll(Collection<?>): boolean
+ retainAll(Collection<?>): boolean
+ set(int, E): E
+ size(): int
+ subList(int, int): List<E>
+ toArray(): Object[]
+ toArray(T[]): T[] <T>
java::util::AbstractCollection <E>
AbstractCollection()
+ add(E): boolean
+ addAll(Collection<? : E>): boolean
+ clear()
+ contains(Object): boolean
+ containsAll(Collection<?>): boolean
- finishToArray(T[], Iterator<?>): T[] <T>
+ isEmpty(): boolean
+ iterator(): Iterator<E>
+ remove(Object): boolean
+ removeAll(Collection<?>): boolean
+ retainAll(Collection<?>): boolean
+ size(): int
+ toArray(): Object[]
+ toArray(T[]): T[] <T>
+ toString(): String
ensiie

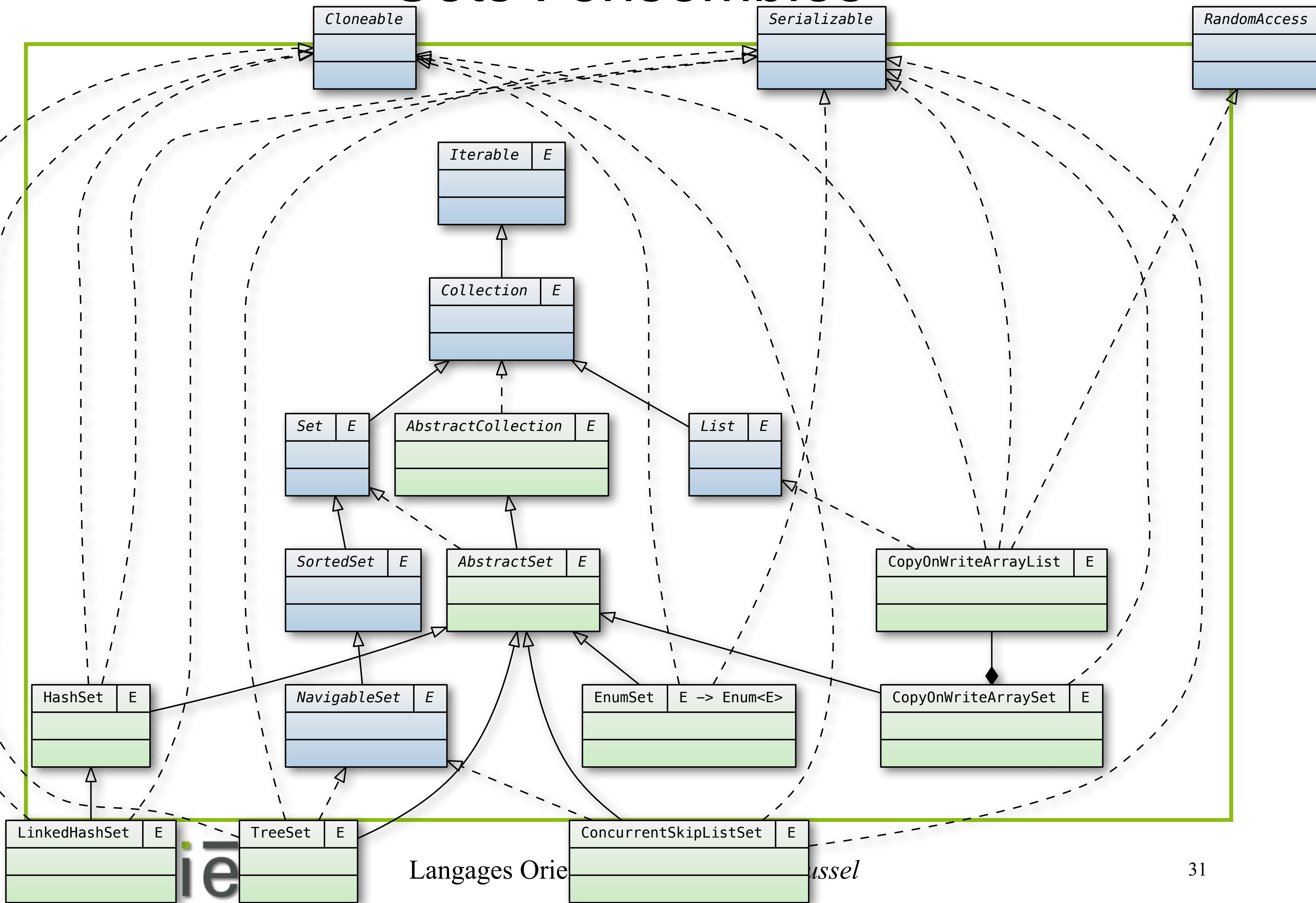
Queues : files d'attente



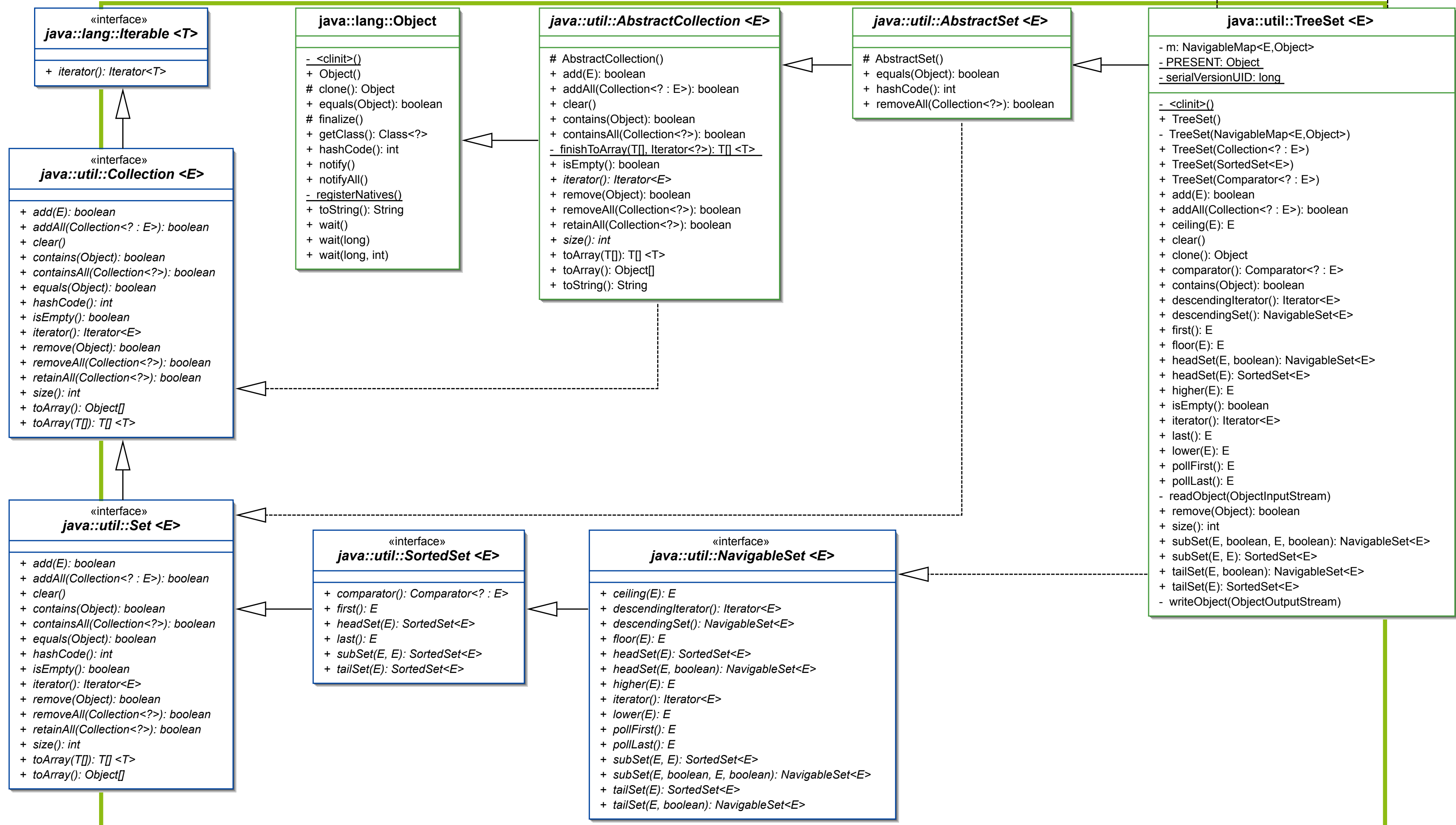
Exemple de Queue : LinkedBlockingDeque



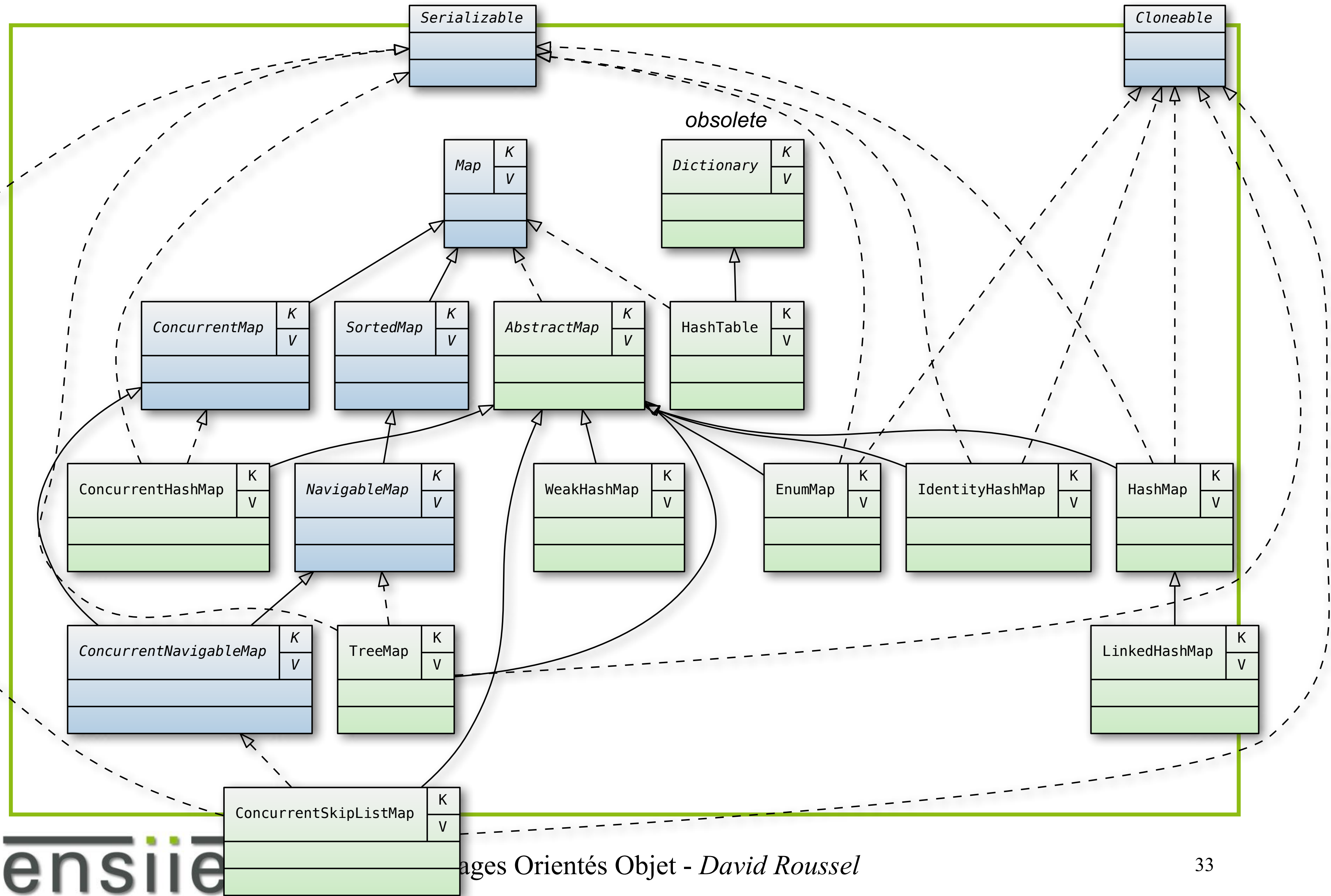
Sets : ensembles



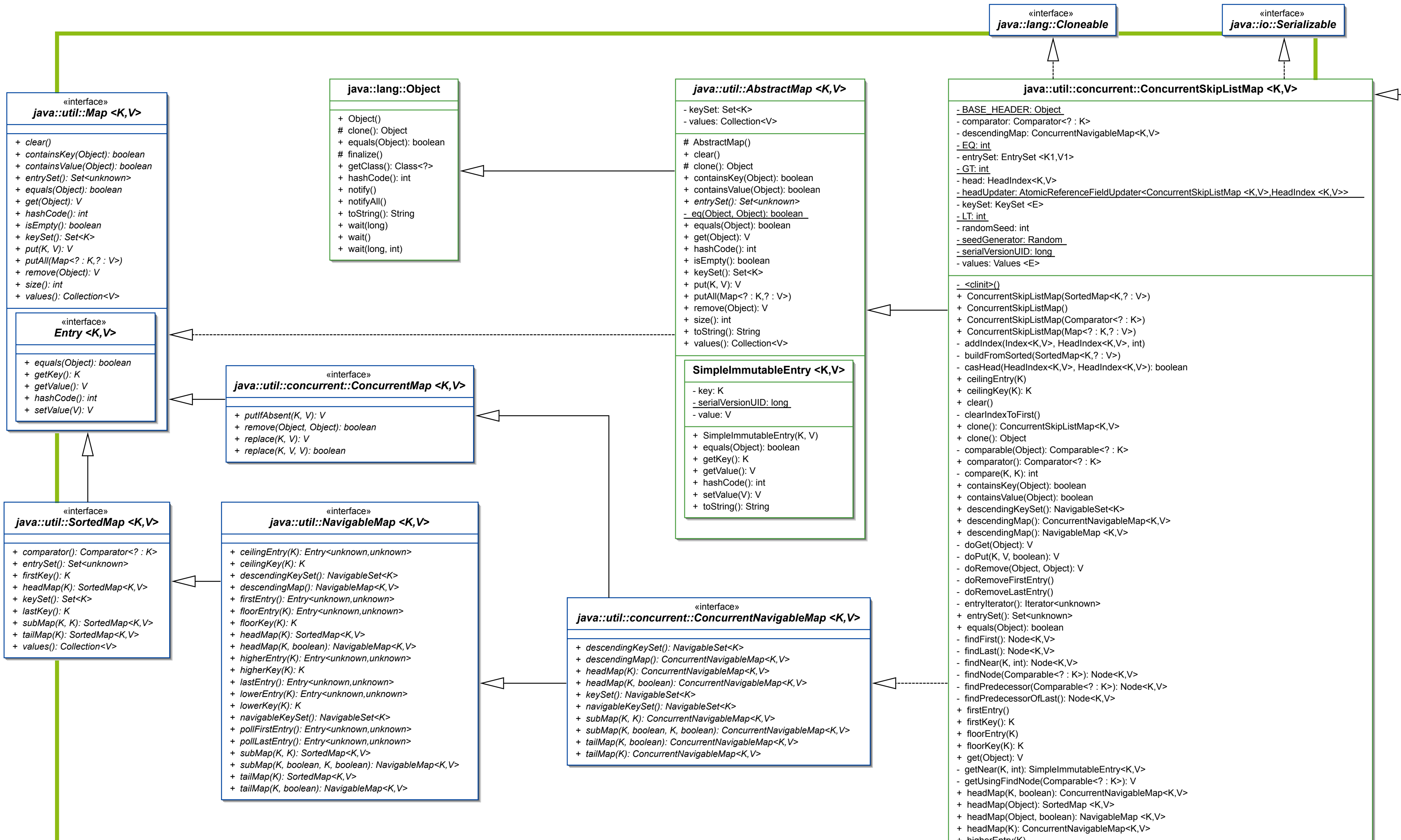
Exemple de Set : TreeSet



Maps : indexs, dictionnaires



Exemple de Map : ConcurrentSkipListMap



Algorithmes de java.util.Collections

- **sort(List)** - Sorts a list using a merge sort algorithm, which provides average-case performance comparable to a high-quality quicksort, guaranteed O(n*log n) performance (unlike quicksort), and *stability* (unlike quicksort). (A stable sort is one that does not reorder equal elements.)
- **binarySearch(List, Object)** - Searches for an element in an ordered list using the binary search algorithm.
- **reverse(List)** - Reverses the order of the elements in the a list.
- **shuffle(List)** - Randomly permutes the elements in a list.
- **fill(List, Object)** - Overwrites every element in a list with the specified value.
- **copy(List dest, List src)** - Copies the source list into the destination list.
- **min(Collection)** - Returns the minimum element in a collection.
- **max(Collection)** - Returns the maximum element in a collection.
- **rotate(List list, int distance)** - Rotates all of the elements in the list by the specified distance.
- **replaceAll(List list, Object oldVal, Object newVal)** - Replaces all occurrences of one specified value with another.
- **indexOfSubList(List source, List target)** - Returns the index of the first sublist of source that is equal to target.
- **lastIndexOfSubList(List source, List target)** - Returns the index of the last sublist of source that is equal to target.
- **swap(List, int, int)** - Swaps the elements at the specified positions in the specified list.
- **frequency(Collection, Object)** - Counts the number of times the specified element occurs in the specified collection.
- **disjoint(Collection, Collection)** - Determines whether two collections are disjoint, in other words, whether they contain no elements in common.
- **addAll(Collection<? super T>, T...)** - Adds all of the elements in the specified array to the specified collection.
- **newSetFromMap(Map)** - Creates a general purpose Set implementation from a general purpose Map implementation.
- **asLifoQueue(Deque)** - Returns a view of a Deque as a Last-in-first-out (Lifo) Queue.