IPRO

GÉNÉRICITÉ ET COLLECTIONS

Guillaume Bouyer

ENSIIE

www.ensiie.fr/~bouyer

GÉNÉRICITÉ

PROGRAMMATION GÉNÉRIQUE

La programmation générique consiste à écrire du code qui puisse être réutilisé pour des objets de types différents

Elle concerne essentiellement les **collections**, objets dont la principale fonctionnalité est de contenir d'autres objets (ex. liste, ensemble...)

PROGRAMMATION GÉNÉRIQUE "À LA MAIN" (AVANT JAVA 5)

Méthode 1 : utiliser le mécanisme d'héritage et la classe `Object`

- Avantage
  - souplesse : collections pouvaient contenir n'importe quel objet (sauf types primitifs)
- Inconvénients
  - impossible de limiter à certains types
  - pas de vérification des erreurs à la compilation (ex. ajout de n'importe quel objet)
  - besoin de caster chaque élément pour en utiliser les membres

Méthode 2 : écrire du code spécifique à chaque type

JAVA GENERICS (JDK 5.0 - 2004)

Principe : paramétrer les classes, les interfaces et les méthodes par un (ou plusieurs) type(s) de données

Exemple et syntaxe :

```
public class MyClass<E> {
    private E id;
    public E getID() {
        return id;
  }
}
```

```
MyClass<String> s = new MyClass<String>();
String sID = s.getID();

MyClass<Integer> i = new MyClass<Integer>();
Integer iID = i.getID();
```

AVANTAGES

- code plus lisible
- pas de duplication de code
- pas de cast
- vérifications du compilateur : une erreur de compilation (mécanisme statique) vaut mieux qu'une `ClassCastException` à l'exécution (ex. type correct des objets)

CLASSE GÉNÉRIQUE : EXEMPLE

```java
public class Pair<T> {
    private T first;
    private T second;

    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() { return this.first; }
    public void setFirst(T first) { this.first = first; }
    public T getSecond() { return this.second; }
    public void setSecond(T second) { this.second = second; }
}
```

VOCABULAIRE

`MyClass<E>`

type générique
(ici classe générique, il existe des interfaces génériques)

`E`

paramètre de type (ou variable de type)

`MyClass<String>`

type paramétré (ici classe paramétrée)
concret, instanciation du type générique `MyClass<E>`
`E` est remplacé par un type d'objet (ici `String`) appelé
argument de type

UTILISATION DES PARAMÈTRES DE TYPE

Dans le code du type générique, les paramètres de type peuvent être utilisés comme les autres types pour :

- déclarer des variables, des paramètres, des tableaux ou des types retour de méthodes

```
E element;
E[] elements;
E getElement(){...}
void setElement(E e){...}
```

- caster (avertissement à la compilation car non sûr)

```
E element = (E) o;
```

UTILISATION **INTERDITE** DES PARAMÈTRES DE TYPE

Ils ne peuvent pas être utilisés :

- pour créer des objets ou des tableaux
- comme super-type d'un autre type
- à droite de `instanceof`

```
new E()             //INTERDIT
new E[10]           //INTERDIT
class C extends E   //INTERDIT
x instanceof E      //INTERDIT
```

UTILISATION DES ARGUMENTS DE TYPE

Au moment de la création d'une classe paramétrée, on fournit un argument de type pour chaque paramètre de type

```
// syntaxe jdk < 7
Pair<Double> p = new Pair<Double>();
List<String> liste = new ArrayList<String>();
Map<String,Integer> map = new HashMap<String,Integer>();
```

```
// syntaxe "diamant" jdk >= 7
Pair<Double> p = new Pair<>();
List<String> liste = new ArrayList<>();
Map<String,Integer> map = new HashMap<>();
```

UTILISATION DES ARGUMENTS DE TYPE

- Ils peuvent être des classes, même abstraites, ou des interfaces

```
List<EmployeI> l = new ArrayList<EmployeI>();
```

- Ils peuvent être eux-mêmes des paramètres de type

```
public class C<E> {
  ...
  f = new ArrayList<E>();
}
```

- Ils ne peuvent pas être un type primitif (`int` ...)

PARAMÈTRES DE TYPE MULTIPLES

Un type générique peut avoir plusieurs paramètres de type

```
public class KeyValue<T,U> {
    private T key;
    private U value;

    public KeyValue(T key, U value) {
        this.key = key;
        this.value = value;
    }

    public T getKey() { return this.key; }
    public void setKey(T key) { this.key = key; }
    public U getValue() { return this.value; }
    public void setValue(U value) { this.value = value; }
}
```

```
KeyValue<String, Integer> contact = new KeyValue<>("John", 1823);
```

RAW TYPES

Pour des raisons de compatibilité, Java a conservé les anciens types non génériques.

On peut donc instancier une classe générique sans donner de type en argument

```
File<Integer> fileInt = new File<Integer>(); // ok instance de type paramétré
File file = fileInt;                         // warning "should be parameterized"
file = new File();                           // idem
File<Integer> fileInt = file;                // warning "unchecked conversion"
```

En réalité les instanciations des types génériques ne donnent à l'exécution qu'un seul type dans la JVM : le type “raw”

- Ex. : `ArrayList<Integer>` et `ArrayList<Employe>` sont représentés dans la JVM par une seule classe `ArrayList`

TYPE ERASURE

De plus le compilateur transforme le code générique en enlevant presque toute trace de généricité.

Les paramètres de type représentent des types inconnus au moment de la compilation du type générique

CONSÉQUENCES DU *TYPE ERASURE*

- 2 méthodes d'une classe ne peuvent donner la même signature après effacement de type
  - Ex : `void m(List<Message> messages)` et `void m(List<Email> emails)`
- On ne peut utiliser `instanceof` sur un type paramétré car à l'exécution la JVM ne connaît pas le type paramétré d'une instance (seulement le type raw et le type générique)
- Un paramètre de type ne peut être utilisé
  - pour créer un objet (ex. `new T()` –> INTERDIT)
  - comme classe mère d'une classe

TYPE PARAMÈTRE BORNÉ (OU CONTRAINT)

Pour limiter l'usage d'un type générique à des arguments de type particulier (ex. pour garantir que les objets possèderont une certaine méthode) on peut restreindre le paramètre de type T :

- à une classe qui implémente une certaine interface
- à une classe qui hérite d'une certaine classe
- dans les 2 cas avec le même mot-clé `extends`

```
public class PairOfNumbers<T extends Number> {
  ...
}

public class MyCollection<E extends Comparable> {
  ...
}
```

TYPE PARAMÈTRE BORNÉ

Plusieurs contraintes possibles

- Types limitants séparés par **&** (rappel : virgule utilisée pour séparer les variables de type)
- Comme pour l'héritage Java, plusieurs interfaces possibles mais une seule classe (doit être en 1er dans la liste)

```
<T extends Comparable & Serializable >
```

GÉNÉRICITÉ ET HÉRITAGE

- Une classe générique peut hériter d'une autre classe générique
- Une classe générique peut hériter d'une classe non générique
- Une classe générique peut implémenter une interface générique

```
public class A<E> implements I<E>...
public class B<T, U> extends A<T>...
public class C<E> extends G...
```

Alors

- `A<E>` est un sous-type de `I<E>`
- idem pour tous les types paramétrés issus de A et I :
  - *ex.* `A<Integer>` est un sous-type de `I<Integer>`

GÉNÉRICITÉ ET HÉRITAGE

Par contre, si 2 classes ont un lien d'héritage, les classes paramétrées par ces classes **n'ont aucun lien de sous-typage** entre elles

=> Pose des problèmes de réutilisation

Ex `Double` hérite de `Number` mais `ArrayList<Double>` n'a aucun lien de sous-typage avec `ArrayList<Number>`

```
public static void printElements(ArrayList<Number> list) {
  for(Number n : list) {
    System.out.println(n.intValue());
  }
}

ArrayList<Double> list = new ArrayList<Double>();
printElements(list); //ne compile pas : liste n'est pas un sous-type de ArrayList<Number>
```

MÉTHODES GÉNÉRIQUES

Comme une classe ou une interface, une méthode (ou un constructeur) peut être paramétrée par un ou plusieurs types :

```
visibilité modificateurs <T> typeRetour nomMéthode(parametres)
```

Indique que la méthode dépend de types non connus au moment de l'écriture

Une méthode générique peut être incluse dans une classe non générique, ou dans une classe générique (si paramètre différent)

MÉTHODES GÉNÉRIQUES : EXEMPLES

```
//dans l'interface Collection<E>
//Returns an array containing all of the elements in this collection;
//the runtime type of the returned array is that of the specified array
public abstract <T> T[] toArray(T[] a)
```

```
class TableUtils {
  //retourne l'élément central de n'importe quel tableau
  public static <T> T getMiddleElement(T[] tab) {
    return tab[tab.length / 2];
  }
}
```

MÉTHODES GÉNÉRIQUES : UTILISATION

On peut appeler une méthode paramétrée en la préfixant par le (ou les) type(s) qui doi(ven)t remplacer le paramètre de type

```
String[] names = {"Marie", "possède", "une", "petite", "lampe"};
String middle = TableUtils.<String>getMiddleElement(names);    // rappel : static
```

MÉTHODE GÉNÉRIQUE : INFÉRENCE DE TYPE

Le plus souvent le compilateur peut faire une inférence de type d'après le contexte d'appel de la méthode

```
ArrayList<Personne> list;
...
Employe[] res = list.toArray(new Employe[0]); // inutile de préfixer <Employe>toArray(...)
String middle = TableUtils.getMiddleElement(names);  //idem
```

JOKER : ?

Permet de relâcher les contraintes sur les types paramétrés pour rendre des méthodes plus réutilisables

- `<?>` désigne un type inconnu
- `<? extends A>` : type inconnu qui est `A` ou un sous-type de `A`
  - `A` peut être une classe, une interface, ou même un paramètre de type
- `<? super A>` : type inconnu qui est `A` ou un sur-type de `A`

JOKER : EXEMPLE

```
public static void printElements(ArrayList<? extends Number> list) {
  for(Number n : list) {
    System.out.println(n.intValue());
  }
}

ArrayList<Double> list = new ArrayList<Double>();
printElements(list); // Compile
```

```
static double sum(List<? extends Number> list){
  double res = 0.0;
  for(Number n : list)
      res += n.doubleValue();
  return res;
}
```

UTILISATION INTERDITES DU JOKER

Ne peut pas être utilisé

- lors de la création des objets

```
new ArrayList<? extends Integer>() // interdit
```

- pour indiquer le type des éléments d'un tableau au moment de sa création

```
new List<? extends Number>[10] // interdit
```

  - Cependant le type d'un tableau peut être une instanciation avec joker non contraint

```
new List<?>[10] // autorisé
```

GÉNÉRICITÉ ET CLASSE INTERNE

- Une classe interne non `static` peut utiliser un paramètre de type de la classe générique
- Ne pas mettre le paramètre de type dans la définition de la classe interne (considéré comme un paramètre de type différent de celui de la classe englobante)

```
public class MyClass<K,T> {
private class InternalClass {
  T t;
  ...
}
}
```

- Une classe interne `static` n'a pas accès aux paramètres de type de la classe englobante

COLLECTIONS

JAVA COLLECTIONS FRAMEWORK

Collection = objet qui contient un ensemble d'objets appelés *éléments*

Java 1 : structures `Vector, Stack, HashTable, Enumeration`

Java 2 : `java.util.Collection`

- ensemble d'interfaces et de classes structurées qui implémentent des fonctionnalités de gestion des collections (accès, ajout, suppression, parcours des éléments...)

Java 5 et + : `java.util.Collection<E>`

CAS DES TABLEAUX

- Suite d'éléments
- Taille et type des éléments fixés à la création
- Acceptent les objets et les types primitifs
- Ne sont pas des `Collection`
- Non thread safe
- Toujours mutables

MÉTHODES DES TABLEAUX

- Méthodes `static` de la classe `java.util.Arrays`
  - Trier : `sort`
  - Chercher dans un tableau trié : `binarySearch`
  - Comparer 2 tableaux : `equals` et `deepEquals`
  - Représenter un tableau sous forme de `String` : `toString`
  - Remplir un tableau avec des valeurs : `fill`
  - Copier un tableau (java 6) : `copyOf`
  - ...
- `System.arraycopy` permet de copier les éléments d'un tableau dans un tableau existant

JAVA COLLECTIONS FRAMEWORK : HIÉRARCHIE

JAVA COLLECTIONS FRAMEWORK : HIÉRARCHIE

- 2 interfaces de base
  - `Collection<E>` : pour gérer un groupe d'objets
  - `Map<K,V>` : pour gérer des éléments de type paires de clé/valeur
- Des sous-interfaces (utiles comme type générique)
  - ex `List<E>`, `Set<E>` ...
- Des classes abstraites qui implémentent les méthodes de base, permettant de faire ses propres collections
  - ex. `AbstractCollection<E>`, `AbstractList<E>`...
- Des collections concrètes directement utilisables qui ajoutent
  - les supports pour stocker les objets
  - les méthodes d'accès

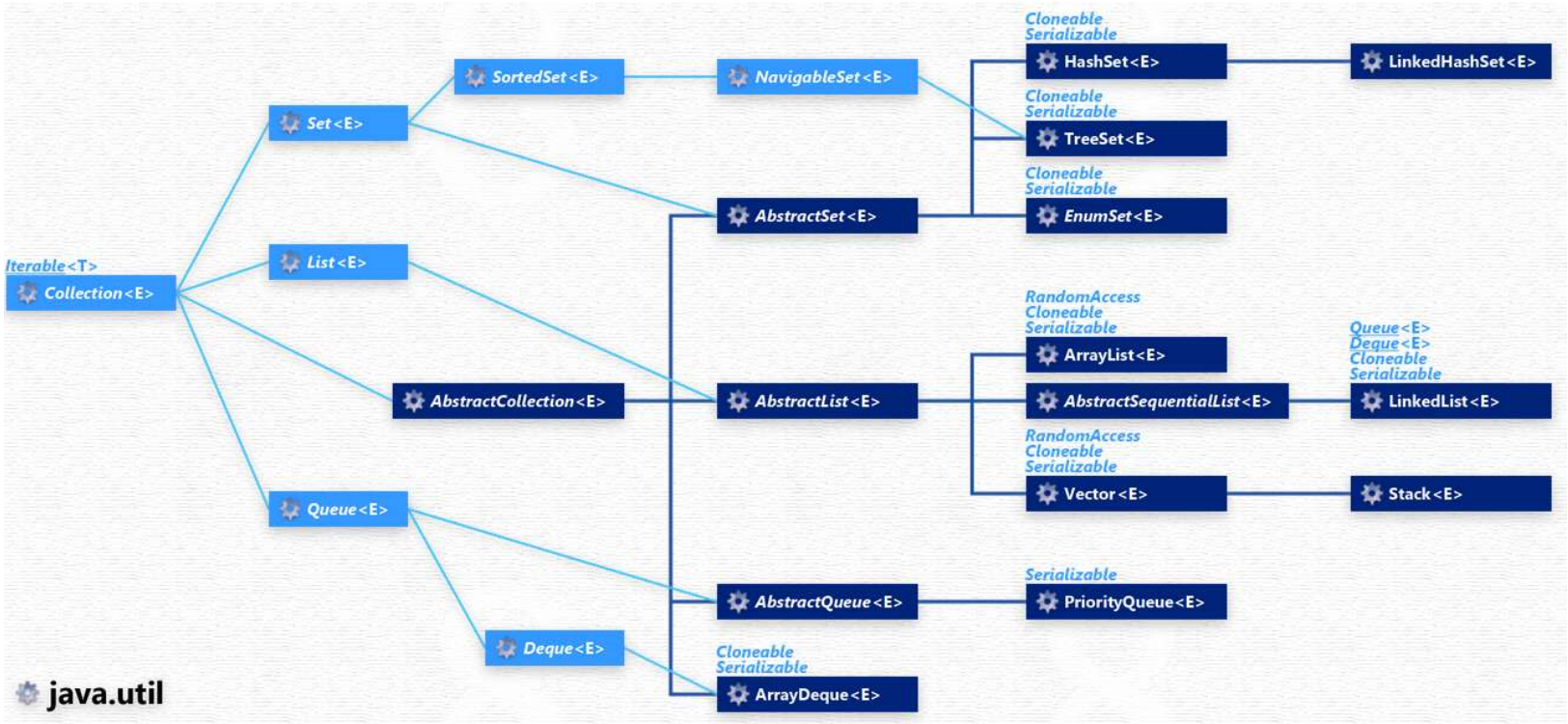
AUTRES CLASSES ET INTERFACES UTILES

Dans `java.util` :

- La classe `Collections` fournit des méthodes `static` notamment pour :
  - trier une collection
  - rechercher dans une collection triée
- 2 interfaces pour le parcours :
  - `Iterator`
  - `ListIterator`
- 1 classe utilisée pour le tri :
  - `Comparator`

Dans `java.lang`

INTERFACE COLLECTION<E> : HIÉRARCHIE



COLLECTION<E> : MÉTHODES (JAVA 8)

```
boolean   add(E e)
boolean   addAll(Collection<? extends E> c)
void  clear()
boolean   contains(Object o)
boolean   containsAll(Collection<?> c)
boolean   equals(Object o)
int   hashCode()
boolean   isEmpty()
Iterator<E>   iterator()
default Stream<E>   parallelStream()
boolean   remove(Object o)
boolean   removeAll(Collection<?> c)
default boolean   removeIf(Predicate<? super E> filter)
boolean   retainAll(Collection<?> c)
int   size()
default Spliterator<E>  spliterator()
default Stream<E>   stream()
Object[]  toArray()
<T> T[]   toArray(T[] a)
```

COLLECTION<E> : MÉTHODES OPTIONNELLES

Pour tenir compte des nombreux cas particuliers de collections, les interfaces possèdent des *méthodes optionnelles* qui évitent de fournir une interface pour chaque cas particulier

- Renvoient une `java.lang.UnsupportedOperationException` (qui est une `RuntimeException`) dans une classe d'implantation qui ne la supporte pas
- Ex : pour faire une collection non mutable, on ne redéfinit pas les méthodes `add`, `clear`, `remove`...

COLLECTION<E> : CONSTRUCTEURS

Par convention, toute classe d'implantation des collections doit fournir au moins 2 constructeurs :

- sans paramètre : crée une collection vide
- qui prend une collection de type compatible : permet le passage par copie d'une implémentation à l'autre

INTERFACE LIST<E>

Collection indexée par un entier (1er élément = 0)

Mêmes méthodes + nouvelles

```
boolean   add(E e)
void  add(int index, E element)
boolean   addAll(Collection<? extends E> c)
boolean   addAll(int index, Collection<? extends E> c)
void  clear()
boolean   contains(Object o)
boolean   containsAll(Collection<?> c)
boolean   equals(Object o)
E   get(int index)
int   hashCode()
int   indexOf(Object o)
boolean   isEmpty()
Iterator<E>   iterator()
int   lastIndexOf(Object o)
ListIterator<E>   listIterator()
ListIterator<E>   listIterator(int index)
E   remove(int index)
boolean   remove(Object o)
boolean   removeAll(Collection<?> c)
default void  replaceAll(UnaryOperator<E> operator)
boolean   retainAll(Collection<?> c)
E   set(int index, E element)
int   size()
```

CLASSE ARRAYLIST<E>

- Tableau dynamique
- Peut contenir un nombre quelconque d'instances d'une classe `E`
  - facteur d'agrandissement à 1.5 par défaut
- Implémente toutes les méthodes de `List`
- Peut contenir `null`
- Implémente *l'interface marqueur* `RandomAccess` : indique qu'une classe permet un accès direct rapide à un de ses éléments (ex. pour un parcours indexé)
- Rq : remplace `Vector<E>` (deprecated)

ARRAYLIST<E> : EXEMPLE

```
List<Employe> le = new ArrayList<>();
Employe e = new Employe("Dupond");
le.add(e);
...
for (int i = 0 ; i < le.size() ; i++) {
    System.out.println(le.get(i).getName());
}
```

Rq : Il est intéressant d'utiliser l'interface comme type de variable si on veut changer d'implémentation plus tard

CLASSE `LINKEDLIST<E>`

- Liste doublement chaînée
  - Chaque élément contient une référence à l'élément précédent et à l'élément suivant
  - L'élément précédent du 1er est `null`
  - L'élément suivant du dernier est `null`
- Implémente toutes les méthodes de `List`
- Peut contenir `null`

LIST<E> : COMPLEXITÉS

List	add	remove	get	contains
ArrayList	O(1) en queue - O(n) en tête	O(n)	O(1)	O(n)
LinkedList	O(1)	O(1)	O(n)	O(n)

INTERFACE SET<E>

- Représentation des ensembles mathématiques
- Pas de doublons (au sens de `equals()`)
- Mêmes méthodes que l'interface `Collection` mais les "contrats" des méthodes sont adaptés aux ensembles
  - la méthode `add` n'ajoute pas un élément si un élément égal est déjà dans l'ensemble (`return false`)
  - quand on enlève un objet, tout objet égal à l'objet passé en paramètre sera enlevé

Rq : si on ajoute des objets modifiables, l'absence de doublons n'est plus garantie

SET<E> : IMPLÉMENTATIONS

- `HashSet<E>` : table de hachage
- `LinkedHashSet<E>` : table de hachage + liste chainée

SET<E> ORDONNÉS

- `SortedSet<E>` :
  - éléments ordonnés
  - ordre naturel sur E ou fourni par un comparateur (voir plus loin)
  - méthodes ajoutées à `Set` : `E first(), E last(), SortedSet<E> subSet(E debut, E fin)...`
- `NavigableSet<E>`
  - permet de naviguer dans un `Set` à partir d'un de ses éléments
  - dans l'ordre du `Set` ou dans l'ordre inverse : `E higher(E)...`
    - Implémentations : `TreeSet<E>` (arbre ordonné)

SET<E> : COMPLEXITÉS

Set	add	remove	contains
HashSet	O(1)	O(1)	O(1)
LinkedHashSet	O(1)	O(1)	O(1)
TreeSet	O(log n)	O(log n)	O(log n)

INTERFACE QUEUE<E>

Représente une file à usage général, le plus souvent une file d'attente (FIFO)

- Eléments ajoutés en queue de liste
- 1er élément = tête
  - sera retourné/retiré en premier
- Méthodes
  - `add, remove, element` : lancent exception
  - `offer, poll, peek` : même fonction mais retournent `false` ou `null`

Implémentations notables : `LinkedList<E>` et `PriorityQueue<E>`

INTERFACE DEQUE<E>

Sous-interface dont les éléments peuvent être ajoutés en tête et en queue

Implémentation notable : `ArrayDeque<E>`, conseillée pour pile LIFO (et pas `java.util.Stack`)

QUEUE<E> : COMPLEXITÉS

Queue	push	peek	pop	contains
PriorityQueue	O(1)	O(1)	O(1)	O(n)
ArrayDeque	O(1)	O(1)	O(1)	O(n)

INTERFACE `MAP<K,V>`

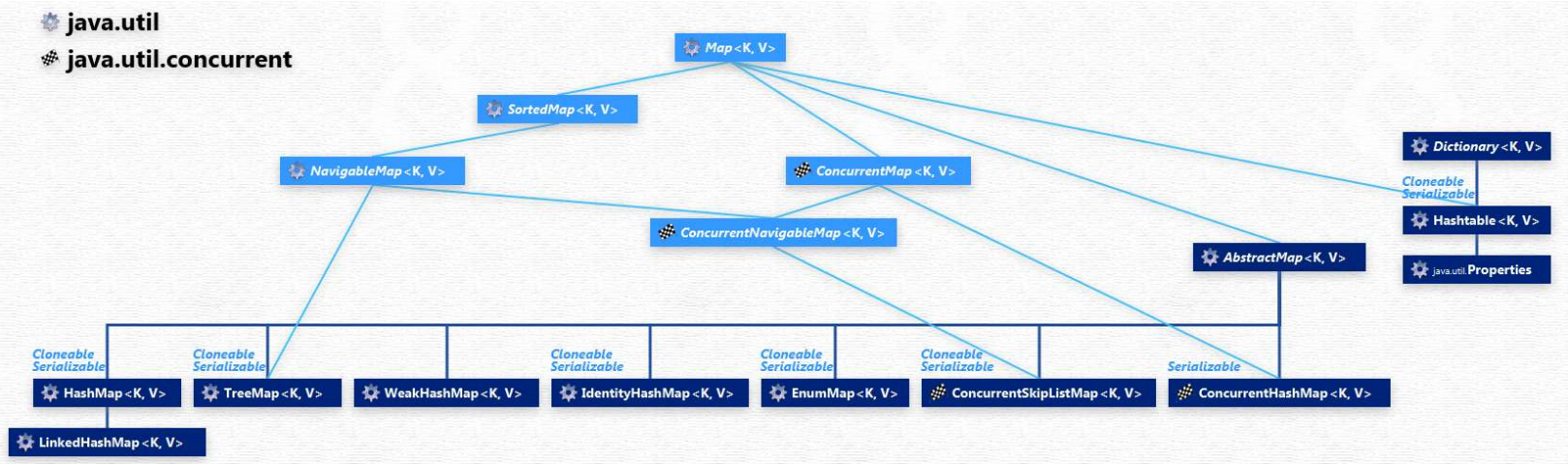
Problème fréquent : rechercher des informations en connaissant une clé qui permet de les identifier

- nom -> numéro de téléphone
- immatriculation -> informations sur le véhicule

`Map` (aussi appelée table associative ou dictionnaire) : associe une clé à une valeur

- La clé est unique au sens de `equals()`
- Une clé repère une et une seule valeur
- La valeur peut être associée à plusieurs clés

MAP<K,V> : HIÉRARCHIE



MAP<K,V> : SOUS-INTERFACES

- `SortedMap<K,V>`
  - fournit un ordre total sur ses clés (ordre naturel sur K ou comparateur associé à la Map)
  - méthodes pour extraire des sous-map, la 1ère ou la dernière clé (cf `SortedSet`)
- `NavigableMap<K,V>`
  - Ajoute des fonctionnalités à un `SortedMap`
  - Permet de naviguer à partir d'une de ses clés, dans l'ordre du `Set` ou dans l'ordre inverse

MAP<K,V> : IMPLÉMENTATIONS NOTABLES

- `HashMap`
  - table de hachage
  - `hashCode()` (héritée de Object ou redéfinie) est utilisée comme fonction de hachage
- `TreeMap`
  - arbre ordonné suivant les valeurs des clés

MAP.ENTRY<K,V>

Interface interne publique qui correspond à un couple clé-valeur

```
static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>>  comparingByKey()
static <K,V> Comparator<Map.Entry<K,V>>   comparingByKey(Comparator<? super K> cmp)
static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>>  comparingByValue()
static <K,V> Comparator<Map.Entry<K,V>>   comparingByValue(Comparator<? super V> cmp)
boolean   equals(Object o)
K   getKey()
V   getValue()
int   hashCode()
V   setValue(V value)
```

MAP<K,V> : MÉTHODES

```
void  clear()
default V   compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
default V   computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)
default V   computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunc
boolean   containsKey(Object key)
boolean   containsValue(Object value)
Set<Map.Entry<K,V>>   entrySet()
boolean   equals(Object o)
default void  forEach(BiConsumer<? super K,? super V> action)
V   get(Object key)
default V   getOrDefault(Object key, V defaultValue)
int   hashCode()
boolean   isEmpty()
Set<K>  keySet()
default V   merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFuncti
V   put(K key, V value)
void  putAll(Map<? extends K,? extends V> m)
default V   putIfAbsent(K key, V value)
V   remove(Object key)
default boolean   remove(Object key, Object value)
default V   replace(K key, V value)
default boolean   replace(K key, V oldValue, V newValue)
default void  replaceAll(BiFunction<? super K,? super V,? extends V> function)
int   size()
Collection<V>   values()
```

MAP<K,V> : EXEMPLE

```
Map<String, Integer> frequencies = new HashMap<>();
for (String  word : args) {
    Integer freq = frequencies.get(word);
    if (freq == null)
        freq = 1;
    else
        freq = freq + 1;
    frequencies.put(word, freq);
}
System.out.println(frequencies);
```

MAP<K,V> : COMPLEXITÉS

Map	put	remove	get	contains(e)	contains(k)
HashMap	O(1)	O(1)	O(1)	O(n)	O(1)
TreeMap	O(log n)	O(log n)	O(log n)	O(n)	O(log n)

FONCTIONNALITÉS DES COLLECTIONS

PARCOURS DES COLLECTIONS : `ITERABLE<E>` ET `ITERATOR()`

Toutes les collections héritent de l'interface `Iterable<E>` : indique qu'elles peuvent être parcourues

Pour cela elles utilisent un itérateur : une instance d'une classe qui implémente l'interface `Iterator<E>`

On obtient cet itérateur par la méthode :

```
Iterator<E> iterator()
```

NB : si la collection n'implémente pas `RandomAccess`, ne jamais la parcourir avec un index entier, il faut utiliser l'itérateur

ITERATOR<E>

Un itérateur permet de parcourir une collection et de retourner chaque élément 1 par 1

Il encapsule la structure de la collection : on pourrait changer de type de collection sans avoir à réécrire le code qui utilise l'itérateur

ITERATOR<E> : MÉTHODES

```
E next()
```

- renvoie l'élement **courant** s'il y en un, et passe au suivant
- sinon lance `NoSuchElementException`

```
boolean hasNext()
```

- renvoie true s'il y a encore au moins 1 élement (qui peut être renvoyé par `next()`)

ITERATOR<E> : MÉTHODES

```
void remove()
```

- supprime le dernier élément récupéré par `next()`
  - `next()` doit être appelé avant
  - `remove()` ne peut pas être appelé 2 fois de suite
  - sinon lance `IllegalStateException`
- optionnel : lance `NoSuchOperationException` si la méthode n'est pas implémentée

Rq : `List<E>` contient en plus la méthode `ListIterator<E> listIterator()` qui offre plus de possibilités pour

- parcourir une liste dans les 2 sens (`hasPrevious()`, `previous()`)
- la modifier (`add`, `set`)

ITERATOR<E> : EXEMPLE

```
List<Employe> le = new ArrayList<>();
Employe e = new Employe("Dupond");
le.add(e);
... // Ajoute d'autres employés
Iterator<Employe> it = le.iterator();
while (it.hasNext()) {
    // le 1er next() fournit le 1er élément
    System.out.println(it.next().getName());
}
```

BOUCLE “FOR EACH”

Autre syntaxe possible “implicite” de l'iterator

```
for (typeElements e : objetIterable)
    instructions
```

Limites

- On ne dispose pas de la position dans le tableau ou la collection pendant le parcours
- On ne peut pas modifier la collection pendant qu'on parcourt la boucle (contrairement à l'itérateur “explicite”)

```
for (Employe e : le) {
    System.out.println(e.getName());
}
```

ITERATOR : CAS DES MAP

On récupère une collection itérable des entrées, des clés ou des valeurs

```
Map<Integer, Integer> map = new HashMap<Integer, Integer>();

//iterating over entries
for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());
}

//iterating over keys only
for (Integer key : map.keySet()) {
    System.out.println("Key = " + key);
}
```

ITERATOR : CAS DES MAP

```
//iterating over values only
for (Integer value : map.values()) {
    System.out.println("Value = " + value);
}

//iterator
Iterator<Map.Entry<Integer, Integer>> entries = map.entrySet().iterator();
while (entries.hasNext()) {
    Map.Entry<Integer, Integer> entry = entries.next();
    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());
}
```

COLLECTIONS ET TYPES PRIMITIFS

Les collections de `java.util` ne peuvent pas contenir de valeurs des types primitifs

- Avant le JDK 5, il fallait utiliser explicitement les classes enveloppantes des types primitifs (ex. `Integer`)
- A partir du JDK 5, les conversions entre les types primitifs et les classes enveloppantes peuvent être implicites avec le concept de “boxing/unboxing”

```
List<Integer> l = new ArrayList<>();
l.add(10); // boxing : au lieu de add(new Integer(10)) ou add(Integer.valueOf(10))
l.add(-678);
l.add(87);
l.add(7);
int i = l.get(0); // unboxing : au lien de l.get(0).intValue()
```

COLLECTIONS ET ÉLÉMENT NULL

Selon les collections et les versions de Java, un élément `null` peut être accepté ou non. Dans tous les cas il n'est pas recommandé d'utiliser `null` pour représenter une collection vide

=> utiliser les constantes : `Collections.emptySet()`, `Collections.emptyList()`, `Collections.emptyMap()`

TRI ET RECHERCHE DANS UNE COLLECTION

La classe `Collections` contient des méthodes `static` utilitaires pour travailler avec des collections :

- tris (sur listes)
- recherches (sur listes triées)
- copie non modifiable d'une collection
- synchronisation
- minimum et maximum

Ex: `Collections.sort(l)`

- Les éléments de la liste doivent implémenter `java.lang.Comparable<T>` (`T` ancêtre du type `E` de la collection)

COMPARABLE<T>

Correspond à l'implantation d'un ordre naturel dans les instances d'une classe. Une seule méthode :

```
int compareTo(T t)
```

Renvoie :

- un entier positif si l'objet est > t
- 0 si les 2 objets sont égaux
- un entier négatif si l'objet est < t
- lance une `NullPointerException` si le paramètre est `null`

COMPARABLE<T>

Certaines classes Java l'implémentent déjà :

- classes enveloppes (ex. `Integer`)
- `String, Date, Calendar, BigInteger, BigDecimal, File, Enum...`
- pas `StringBuffer` ou `StringBuilder`

NB : fortement conseillé d'avoir une méthode `compareTo` compatible avec `equals` :

`e1.compareTo(e2) == 0` SSi `e1.equals(e2)`

COMPARATOR<T>

Si les éléments n'implémentent pas `Comparable` ou si on veut les trier suivant un autre ordre que celui donné par `Comparable`

- Construire une classe qui implémente `java.util.Comparator<T>`
  - Implémenter `int compare(T t1, T t2)` qui renvoit
    - un entier positif si t1 est « plus grand » que t2
    - 0 si t1 a la même valeur (au sens de `equals`) que t2
    - un entier négatif si t1 est « plus petit » que t2
- Passer une instance de cette classe en paramètre à la méthode `sort`

COMPARATOR<T> : EXEMPLE

```
public class CompareSize implements Comparator<People> {
  public int compare(People e1, People e2) {
    double s1 = e1.getSize();
    double s2 = e2.getSize();
    if (s1 > s2)
      return 1;
    else if (s1 < s2)
      return -1;
    else
      return 0;
    }
}
```

```
List<People> p = new ArrayList<>();
// On ajoute les personnes
. . .
Collections.sort(p, new CompareSize());
System.out.println(p);
```

CONVERSION ENTRE COLLECTIONS

2 méthodes

1. Copie des données d'une collection à l'autre
   - les données sont dupliquées
2. *Vue* d'une collection *comme* une autre
   - les données restent dans la structure initiale

TABLEAU VERS COLLECTION

Tableau vers liste : `<T> List<T> Arrays.asList(T... array)`

```
String[] words = { "a", "b", "c" };
List<String> l = Arrays.asList(words);
List<String> l1 = Arrays.asList("a", "b", "c"); //java 5
```

COLLECTIONS VERS TABLEAU

Liste vers tableau :

- `Object[] Collection.toArray()`
  - renvoie une instance de `Object[]` qui contient les éléments de la collection
- `<T> T[] toArray(T[] tableau)`
  - si le tableau est assez grand, les éléments sont rangés dans le tableau
  - sinon, un nouveau tableau du même type est créé pour recevoir les éléments

COLLECTION VERS COLLECTION

- Toutes les collections ont un constructeur qui prend une `Collection<? extends E>`
- `addAll(Collection<? extends E>)` permet d'ajouter les élements d'une collection vers l'autre
- List vers List
  - `list.subList(int start, int end)` permet d'obtenir une sous-liste d'une liste
- Deque vers Queue
  - `Queue<T> Collections.asLifoQueue(Deque<T> deque)`

MAP VERS COLLECTION

- Map vers l'ensemble des clés
  - `Set<K> map.keySet()`
- Map vers la collection des valeurs
  - `Collection<V> map.values()`
- Map vers l'ensemble des couples clé/valeur
  - `Set<Map.Entry<K,V>> map.entrySet()`
- Map vers Set
  - `Set<E> Collections.newSetFromMap(Map<E, Boolean> map)`

LEGACY COLLECTIONS

Java 1	Java >= 2
Vector	ArrayList
Stack	ArrayDeque
HashTable	HashMap
Enumeration	Iterator

COLLECTIONS ET CONCURRENCE

Les `Collections` ne sont pas *thread safe*

Java 5 propose plusieurs collections dans le package `java.util.concurrent` qui permettent des modifications lors de leur parcours, telles que :

- `CopyOnWriteArrayList`
- `ConcurrentHashMap`
- `CopyOnWriteArraySet`

NOTION DE STREAM

Introduits dans Java 8, dans `java.util.stream`

Permettent de manipuler assez naturellement les données des collections

Un stream :

- ne stocke pas les données, mais les transfère d'une source vers une suite d'opérations
- ne modifie pas les données
- n'est pas réutilisable : une fois parcouru il faut en recréer un

NOTION DE STREAM : EXEMPLE

```
List<String> strings = Arrays.asList("girafe", "chameau", "chat", "poisson", "cachalot");
strings.stream() //obtention du stream sur la collection
    .filter(x -> x.contains("cha"))        // filtrage
    .map(x -> x.substring(0, 1).toUpperCase() + x.substring(1)) // mapping : reformatage des
    .sorted() // tri par ordre alphabétique
    .forEach( System.out::println ); // parcours et affichage
```

STREAM : OPÉRATIONS

- Intermédiaires : renvoient un nouveau stream pour les opérations suivantes (on a donc une succession de stream appelée “stream pipeline”)
  - exemples : `map` (conversion, réduction des élements), `filter` (filtre les éléments avec prédicat), `distinct` (enlever doublon), `max`, `average`, `sorted`...
- Terminales : “consomment” l'ensemble des streams, appliquent les opérations
  - ex : `collect` (stocker les élements dans une collection), `foreach` (appliquer une opération sur chaque élement), `count`...

STREAM ET "LAMBDA"

Les expressions lambda (ou closure ou fonctions anonymes) permettent de passer en paramètre les traitement sur le modèle de la prog. fonctionnelle, sous la forme :

```
(arguments) -> traitement;
//ou
(arguments) -> { traitements ; }
```

La plupart des opérations des stream prennent des lambda en paramètre

COLLECTIONS EN PRATIQUE

Présentes dans une large majorité des programmes objet

Il existe de nombreuses collections, chacune adaptée à un usage particulier

- Choisir la bonne collection
- CheatSheet complexité
- Résumé des implémentations

		Classes d'implantations			
		Table de hachage	Tableau	Arbre balancé	Liste chaînée
Interfaces	`Set<E>`	`HashSet<E>`		`TreeSet<E>`	
	`List<E>`		`ArrayList<E>`		`LinkedList<E>`
	`Map<K,V>`	`HashMap<K,V>`		`TreeMap<K,V>`	
	`Queue<E>`		`ArrayDequeue<E>`		`LinkedList<E>`