BIENVENUE

TECHNIFUTUR®
CENTRE DE COMPETENCES

www.enmieux.be

FEDER-FSE
UNION EUROPEENNE
Wallonie
L'UNION EUROPÉENNE ET LA WALLONIE
INVESTISSENT DANS VOTRE AVENIR

Notions tranversales

Wrapper-class

Générique

Interfaces

Classes internes / anonyme

CLASSES DE TYPE-WRAPPER

TECHNIFUTUR
CENTRE DE COMPETENCES

- Les classes “enveloppes” fournissent un mécanisme pour convertir les types primitif en objet et les objets en type primitif.
- **autoboxing** and **unboxing**
  - depuis J2SE 5.0,
  - Page 245

Primitive Type	**Wrapper class**
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Notions transversales
Les génériques

Pourquoi faire

Des définitions

Déclaration de types génériques

Utilisation de types génériques

Type erasure (effacement de type)

Wildcards

POURQUOI UTILISER DES CLASSES GÉNÉRIQUES

TECHNIFUTUR
CENTRE DE COMPETENCES

- Permettre la détection à la compilation de bug qui sans cela ne seraient apparu que lors de l'exécution.
  - Renforcement de la vérification des types au moment de la compilation.
- Suppression d'opération de casting
- Permettre l'écriture des codes pouvant s'appliquer à plusieurs types de données, tout en conservant un contrôle fort sur les types au moment de l'utilisation de ses codes.

DÉFINITIONS

TECHNIFUTUR
CENTRE DE COMPETENCES

- Type générique
  - Classe ou interface paramétrable par un ou plusieurs types.

```
public interface List <E> {
  // corps
}
```

- Méthode générique
  - Méthode ou constructeur paramétrable par un ou plusieurs types.

```
static <T> List<T> asList(T... a)
```

- Paramètre de type
  - Type à définir représenté par un id (souvent une lettre majuscule « T,E,K,V,… »)
  - Un paramètre de type peut être limité par une classe et des interface
    - `<T extends ClassA, InterfaceB, …>`
- Raw Types
  - Est le nom d'une classe générique sans argument de type
    - `Box` est le type brut de `Box<T>`

DÉCLARATION D'UNE CLASSE GÉNÉRIQUE

TECHNIFUTUR
CENTRE DE COMPETENCES

- La déclaration des paramètres de type
  - … *className* **<** *id* [*extends typeName*[*&interfaceName*]] *,* [...]**>**
    - Le type peux être borner (limiter à la déclaration)
      - En indiquant que le type doit étendre une classe et/ou une ou plusieurs interfaces
      - Les classes et interfaces peuvent être elles-mêmes déclarer des génériques
      - Le type ne peux pas être borner par un Throwable
    - On peut déclarer plusieurs types génériques.
- Le(s) type(s) paramètre(s)
  - Utilisation
    - Comme type :
      - D'une variable d'instance
      - D'un paramètre
      - D'une variable locale

UTILISATION D'UN TYPE GÉNÉRIQUE

TECHNIFUTUR
CENTRE DE COMPETENCES

- Déclarer une variable
  - Box<Integer> box;
  - void open(Box<Number> box){}
  - Map<String, Box<Integer>> map;
- Instancier un type générique
  - new Box<Integer>(125)
  - Si l'instanciation est directement affectée à une variable de même paramètre de type
    - Box<Integer> box = new Box<>();
- Pour « borner » un paramètre de type
  - Interface BaseStream<T,S extends BaseStream<T,S>>

TYPE ERASURE (EFFACEMENT DE TYPE)

TECHNIFUTUR
CENTRE DE COMPETENCES

- Les définitions génériques sont effacées et remplacée au moment de la compilation.
  - Dans la classe ou la méthode ou le type T est déclaré
    - Si le type n'est pas borné T est remplacé par Object
    - Si le type est borné T est remplacé par le premier type de la liste
  - Dans le code utilisant une classe générique
    - Le type générique des variables est modifié dans leur type brut
    - Le passage de paramètres de type générique est vérifié par le compilateur.
    - Le type de retour des méthodes fait l'objet d'un casting

CONSÉQUENCES DU TYPE ERASURE

TECHNIFUTUR
CENTRE DE COMPETENCES

- Le type erasure a quatre conséquences :
  - la comparaison des classes génériques mène à des paradoxes ;
  - on ne peut pas instancier directement une variable à partir d'un type générique ;
  - les tableaux de types génériques sont interdits ;
  - il est illégal de créer des méthodes prenant des paramètres génériques, qui pourraient entrer en collision avec des méthodes existantes.

LA COMPARAISON DES CLASSES GÉNÉRIQUES MÈNE À DES PARADOXES ;

TECHNIFUTUR
CENTRE DE COMPÉTENCES

- Étant donné que l'on n'a qu'une unique classe de laquelle on a retiré toute référence au type générique, la méthode getClass() retourne toujours la même classe quelle que soit l'instance d'une classe générique que l'on regarde.

```
Holder<String> h1 = new Holder<String>("Bonjour !") ;
Holder<Float> h2 = new Holder<Float>(3.14) ;

boolean b = h1.getClass() == h2.getClass() ; // ce booléen vaut true
```

- On retrouve ce comportement sur l'utilisation de instanceof.
  - Même si l'on peut utiliser cet opérateur avec des types génériques, ceux-ci ne sont pas pris en compte.

ON NE PEUT PAS INSTANCIER DIRECTEMENT UNE VARIABLE À PARTIR D'UN TYPE GÉNÉRIQUE ;

- Le deuxième point signifie que l'on ne peut pas écrire

```
T t = new T()
```

- ou encore

```
T t = T.class.newInstance().
```

- Le pattern préconisé pour instancier une variable générique à partir de son type est le suivant.

```
public static <T> T newInstance(Class<T> clazz) {
    // ajouter la gestion des exceptions
    return clazz.newInstance() ;
}
```

  - Ce pattern utilise la classe de la variable générique.
  - Notons que la classe Class est elle-même une classe générique.

LES TABLEAUX DE TYPES GÉNÉRIQUES SONT INTERDITS

TECHNIFUTUR
CENTRE DE COMPETENCES

- Fonctionnement des tableaux en Java.

```
1  String [] tabString = new String [10] ;
2  Object [] tabObject = tabString ;
3  tabObject[0] = new Object() ; // ArrayStoreException
```

  - Une ArrayStoreException sera jetée à l'exécution de la ligne 3.
- Il faudrait donc qu'une exception soit soulevée dans ce cas

```
1  // la ligne suivante ne compile pas !!
2  Holder<String> [] tabString = new Holder<String> [10] ;
3  Object [] tabObject = tabString ;
4   tabObject[0] = new Holder<Integer>(1) ;
```

  - La ligne 4 devrais aussi soulever une ArrayStoreException
  - Les types Hodler<String> et Holder<Integer> sont de même type
- Plutôt que de laisser cette possibilité d'écrire des bugs, la décision logique d'interdire les tableaux de génériques a été prise.
  - Le code de la ligne 2 est donc illégal.

COLLISION AVEC DES MÉTHODES EXISTANTES.

- Le type erasure change la signature d'une méthode une fois celle-ci compilée.
  - Si la méthode compilée a une signature qui est déjà présente dans la classe, alors elle devient illégale.

```
public class Holder<T> {
    // méthode en collision avec equals(Object)
    // erreur de compilation !!
    public boolean equals(T t) {
        // corps de la méthode
    }
}
```

WILDCARDS

TECHNIFUTUR
CENTRE DE COMPETENCES

- Exemple
  - // variable contenant la référence d'une Box dont on ne connait pas le type du contenu
  - Box<?> unknownBox;
- Pour écrire des algorithmes génériques à plusieurs type de générique
  - Par exemple pour toutes les Box
- Les wilcards sont utilisés lors de la déclaration
  - des variables,
  - des paramètres
  - et des types de retour d'une méthode
- Les wildcards peuvent être bornées
  - Box< ? extends Number> numberBox;
  - Box< ? super Number> superNumberBox;

WILDCARDS ASSIGNATION

TECHNIFUTUR
CENTRE DE COMPETENCES

a = b;		Type de la variable b					
		Box <Object>	Box <Number>	Box <Integer>	Box <?>	Box <? Extends Number>	Box <? super Number>
Type de la variable a	Box<Object>	Ok					
	Box<Number>		Ok				
	Box<Integer>			Ok			
	Box<?>	Ok	Ok	Ok	Ok	Ok	Ok
	Box<? Extends Number>		Ok	Ok		Ok	
	Box<? super Number>	Ok	Ok				Ok

WILCARDS UTILISATION

		a.getValue() Type de retour	a.Set(value) type accepté		
			Object	Number	Integer
Type de la variable a	Box<Object>	:Object	Ok	Ok	Ok
	Box<Number>	:Number		Ok	Ok
	Box<Integer>	:Integer			Ok
	Box<?>	:Object			
	Box<? Extends Number>	:Number			
	Box<? super Number>	:Object		Ok	Ok

Notions transversales
Interface

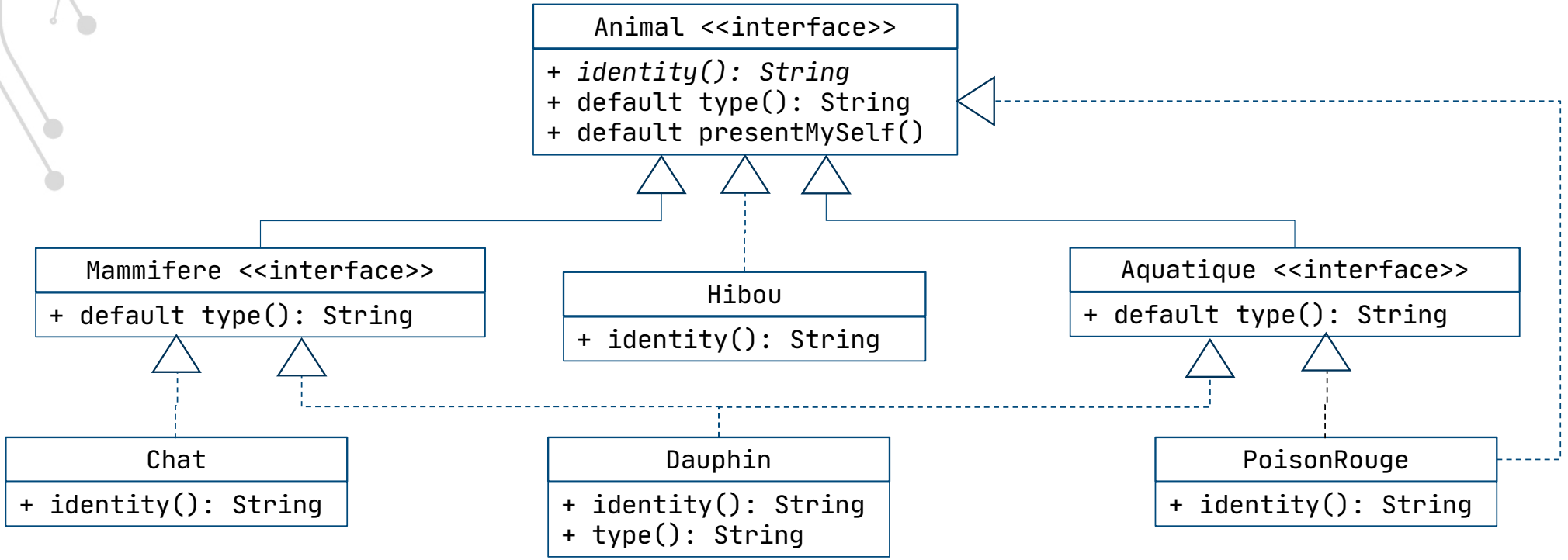
Type de méthodes abstraites, default, static

Implémentation : classe premier niveau, classe interne, classe anonyme

INTERFACE

- Méthode abstraites venant de Object
  - automatique
- Méthodes abstraites
  - Signature de méthode sans code
  - Les descendants non abstrait doivent les redéfinir
- Méthode default
  - Méthode d'instance avec un code par défaut
  - Les descendants n'ont pas d'obligation de redéfinir
- Méthode static
  - Méthode de classe avec un code
- Exemple java.util.Comparator

IMPLÉMENTATION DES INTERFACES



- new Hibou().type()
- new Chat().type()
- new PoisonRouge().type()
- new Dauphin().type()

- "animal"
- "mammifère"
- "aquatique"
- "mammifère aquatique" // obligation d'overrid

INTERFACE INTERNE

- Interface définie dans le corps d'une classe
  - Sa définition peut être public, potected, package-private, private.
  - Full name : packageName.ClassexterneName.InterfaceName

```
public class HelloWorldAnonymousClasses {

    /**
     * interface interne à la classe
     */
    interface HelloWorld {
        public void greet();

        public void greetSomeone(String someone);
    }
}
```

CLASSE INTERNE

- sayHello():
  - méthode englobante pour les classes internes
- EngilishGreeting:
  - classe interne locale à la méthode sayHello implementant HelloWorld
  - Le nom de la classe n'est accessible que dans la méthode

```
public class HelloWorldAnonymousClasses {

    public void sayHello(String name) {

        class EnglishGreeting implements HelloWorld {

            @Override
            public void greet() {
            }

            @Override
            public void greetSomeone(String someone) {
            }
        }
    }
    …
}
```

ACCÈS AUX VARIABLES

- name : variable d'instances de la classe englobante.
- other, format : variable locale (final dans le faits) de la méthode englobante.
- salutation : variable d'instance de la classe interne
- someone : variable locale de la méthode de la classe interne

```
public class HelloWorldAnonymousClasses {
    private String name;

    public HelloWorldAnonymousClasses(String name) { this.name = name; }

    public void setName(String name) { this.name = name;}

    public void sayHello(String other) {
        String format = "%s: \"%s %s\"\n";
        class FrenchGreeting implements HelloWorld {
             String salutation = "salut";
             @Override public void greet() { greetSomeone(« tout le monde"); }
             @Override public void greetSomeone(String someone) {
                System.out.printf(format, name, salutation, name);
            }
        }
    }
}
```

CLASSE ANONYME

- Variable d'instance de la classe englobante : `HelloWorldAnonymousClasses.this.name`
- Variable d'instance : `this.name`
- Variable locale : `name`

```
public class HelloWorldAnonymousClasses {
    private String name;

    public HelloWorldAnonymousClasses(String name) { this.name = name; }

    public void setName(String name) { this.name = name;}

    public void sayHello(String name) {
        String format = "%s: \"%s %s\"\n";
        EnglishGreeting englishGreeting = new HelloWorld() {
             String name = "Hello";
             @Override public void greet() { greetSomeone("world"); }
             @Override public void greetSomeone(String name) {
                System.out.printf(format, HelloWorldAnonymousClasses.this.name, this.name, name);
            }
        }
    }
}
```

EXÉCUTION DU CODE

TECHNIFUTUR
CENTRE DE COMPÉTENCES

```
public class HelloWorldAnonymousClasses {

    public static void main(String... args) {
        HelloWorldAnonymousClasses myApp = new HelloWorldAnonymousClasses("Yannick");
        myApp.sayHello("John");
        myApp.setName("dédé");
        myApp.sayHello("John");
    }

    public void sayHello(String other) {
        …
        englishGreeting.greet();
        englishGreeting.greetSomeone("toto");
        englishGreeting.greetSomeone(other);
        frenchGreeting.greet();
        frenchGreeting.greetSomeone("toto");
        frenchGreeting.greetSomeone(other);
    }
}
```

Collections

Introduction

Interfaces de l'API

Implémentation

INTRODUTION

- Le framework collection est une architecture unifiée pour représenter et manipuler des collections d'objets.
  - Des interfaces
    - Pour manipuler les collections indépendamment de leurs implémentations.
    - Les interfaces sont hiérarchisées pour regrouper les fonctionnalités communes
  - Des implémentations
    - Utilisations des techniques de stockage classique
    - Implémentions réutilisables dans un maximum de citrconstance
  - Des algorithmes
    - Ex recherche tri …
    - Réutilisable pour différentes implèmentations
- Bénéfices attendus
  - Réduction de l'effort de programmation
  - Amélioration de la rapidité et de la qualité des programmes
  - Interopérabilité des API
  - Réduction de l'effort d'apprentissage
  - …

DIFFÉRENCES ENTRE LES COLLECTIONS

TECHNIFUTUR
CENTRE DE COMPETENCES

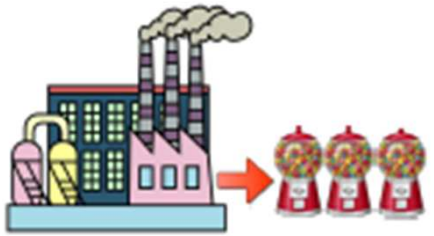
- Qualités différenciant les Collections
  - Taille limité ou non
  - Immuable ou non
  - Accepte les doublons ou non
  - Accepte la valeurs « null »
  - Synchronisée ou non
  - Ordonnée ou non
  - Triée ou non
- Implémentation
  - Centaines méthodes sont documentées *optional*
    - *Ces méthodes soulève alors une « UnsupportedOperationException »*

JAVA.LANG.ITERABLE<E>

TECHNIFUTUR
CENTRE DE COMPETENCES

- Toute les collections sont itérables

java.lang.Iterable<E>
Iterator<E> iterator() default void forEach(Consumer<? super T> action) default Spliterator<T> spliterator()

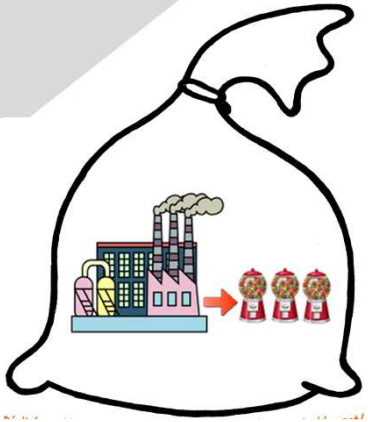


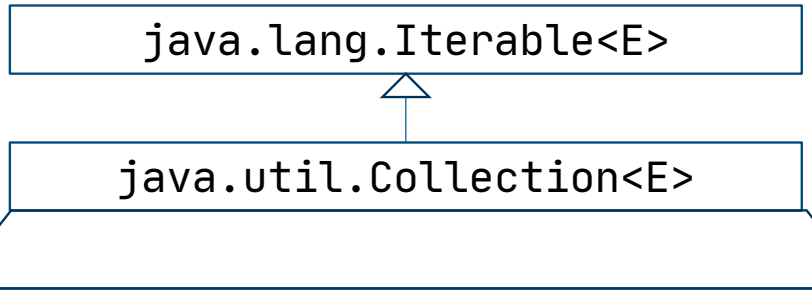
java.util.Iterator<E>
boolean hasNext() E next() default void remove() default void forEachRemaining(Consumer<? super E> action)

UTILISATION D'UN ITÉRABLE

```
public static void main(String[] args) {
    //Iterable<Personne> personnes = Arrays.asList(Personne.dataTest());
    Iterable<Personne> personnes = new ArrayList<>(Arrays.asList(Personne.dataTest()));

    Iterator<Personne> iterator = personnes.iterator();
    while (iterator.hasNext()){
        Personne p = iterator.next();
        System.out.println(p.getName());
    }

    iterator = personnes.iterator();
    while (iterator.hasNext()){
        Personne p = iterator.next();
        if(p.getName().endsWith("d"))
            iterator.remove();
     }

    for (Personne p : personnes){
        System.out.println(p.getPrenom());
    }
}
```

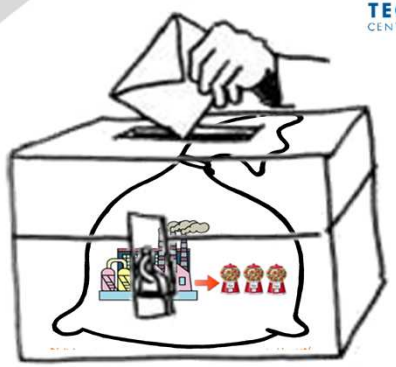
JAVA.UTIL.COLLECTION<E>

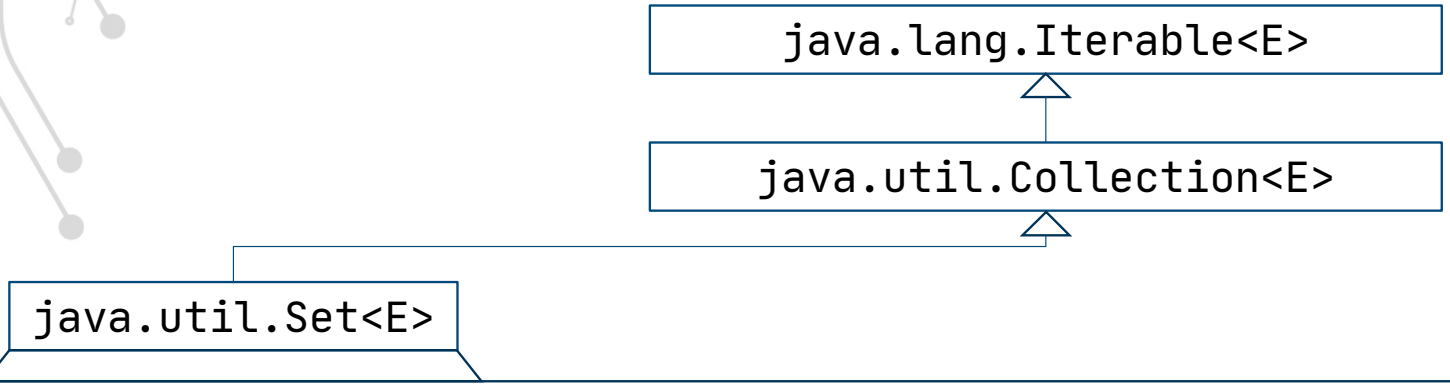




- Query operations
  - + size() :int
  - + isEmpty():boolean
  - + contains(o:Object):boolean
  - + iterator():Iterator
  - + toArray():Object[]
  - + toArray(a:T[]):T[]
- Modification Operations
  - + add(e:E):boolean
  - + remove(o:Object):boolean
- Bulk operations
  - + containsAll(c:Collection):boolean
  - + addAll(c:Collection):boolean
  - + removeAll(c:Collection):boolean
  - + retainsAll(c:Collection):boolean
  - + Clear()
- Comparison and hashing
  - + equals(o:Object):boolean
  - + hashCode():int

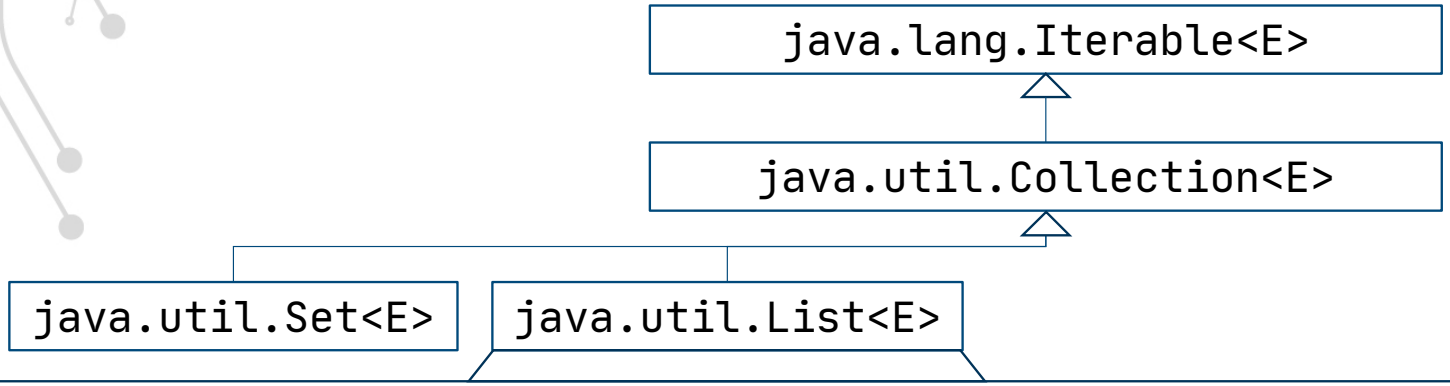
JAVA.UTIL.SET<E>

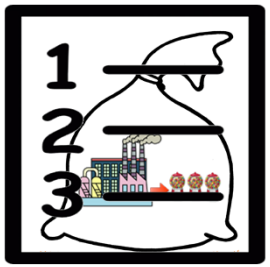




- Pas de nouvelles opérations
  Mais
- Les « Set » garantissent qu'il n'y a pas de doublons

JAVA.UTIL.LIST<E>

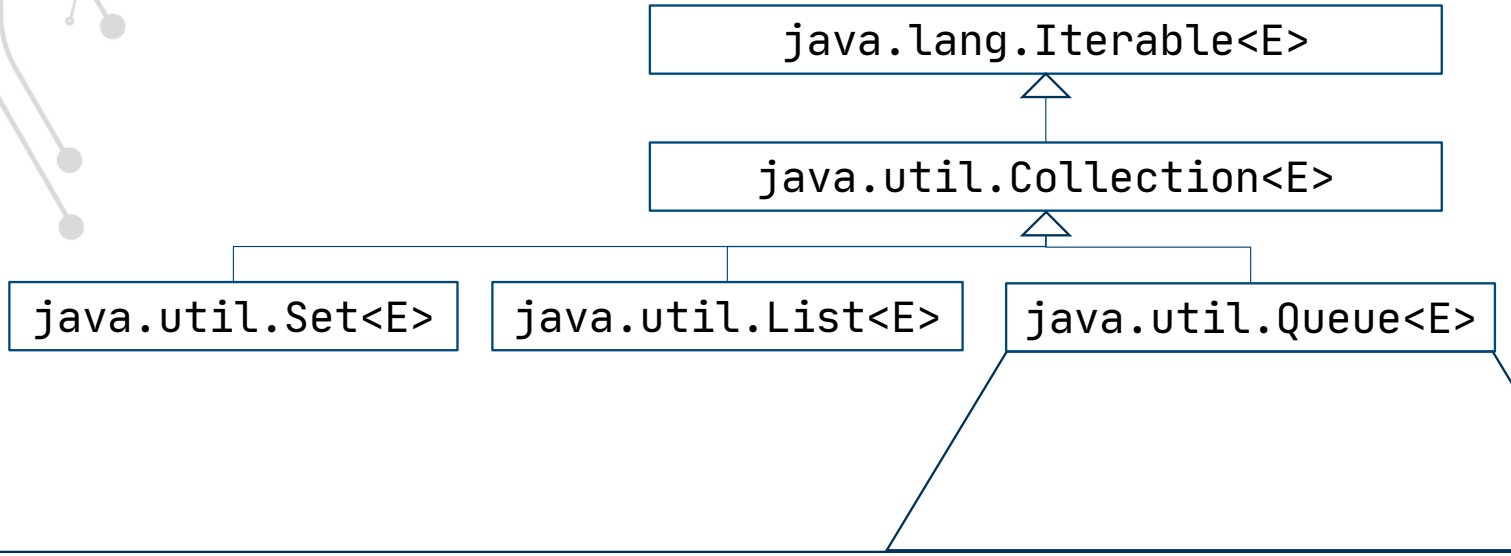




Les éléments d'une liste sont ordonnés

- Positional access operations
  - + get(index:int):E
  - + set(index:int,element:E):boolean
  - + add(index:int,element:E):boolean
  - + remove(index:int):E
  - + addAll(index:int,c:Collection) :boolean
- Search operations
  - + indexOf(o:Object):int
  - + lastindexOf(o:Objetc):int
  - + listIterator():ListIterator<E>
  - + listIterator(index:int):Listerator<E>
- View
  - + subList(fromIndex:int, toIndex:int):List<E>

JAVA.UTIL.QUEUE<E>

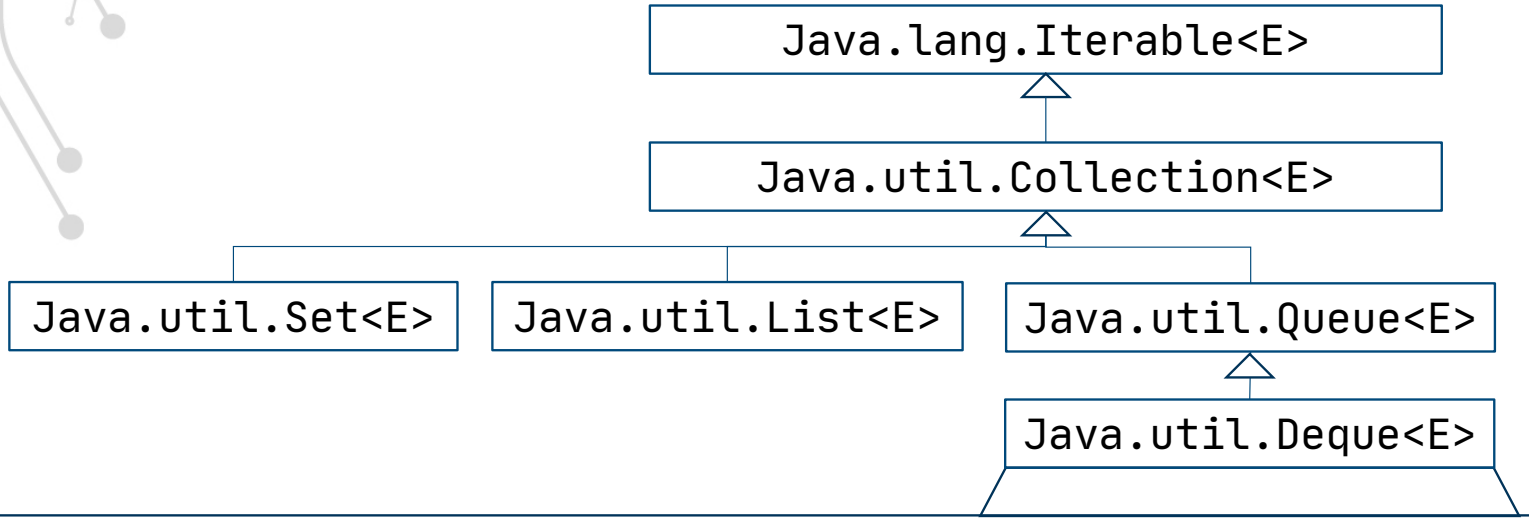


Collection permettant l'accès aux éléments suivant un ordre de priorité.
Par exemple: FIFO, LIFO

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

```
+ element():E
+ offer(e:E):boolean
+ peek():E
+ poll():E
+ remove():E
```

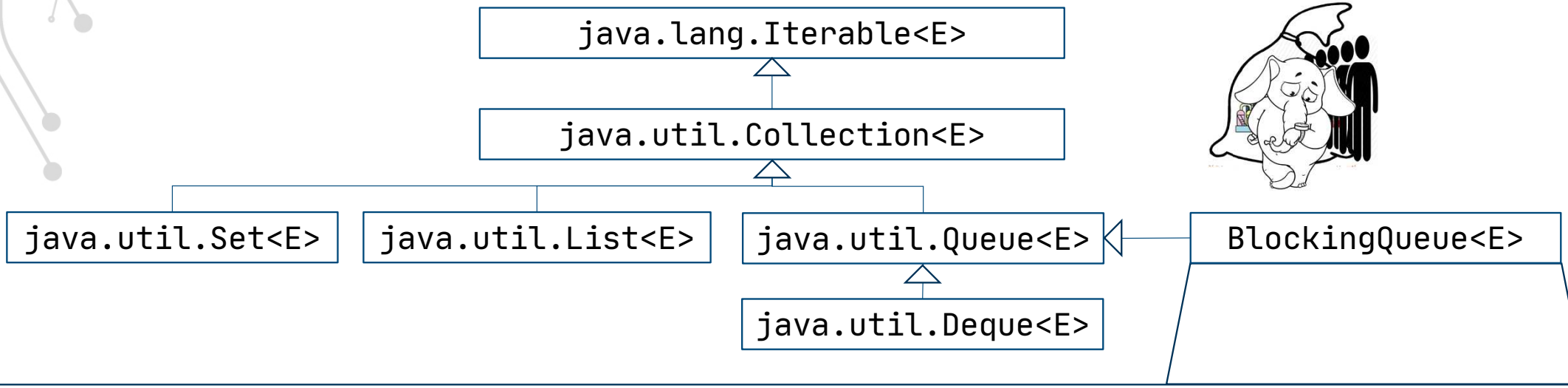
JAVA.UTIL.DEQUE<E>



Queue permettant l'accès, suivant un ordre de priorité, au premier et au dernier éléments.

	First Element (Head)		Last Element (Tail)	
	Throws exception	Special value	Throws exception	Special value
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

JAVA.UTIL.BLOCKINGQUEUE<E>

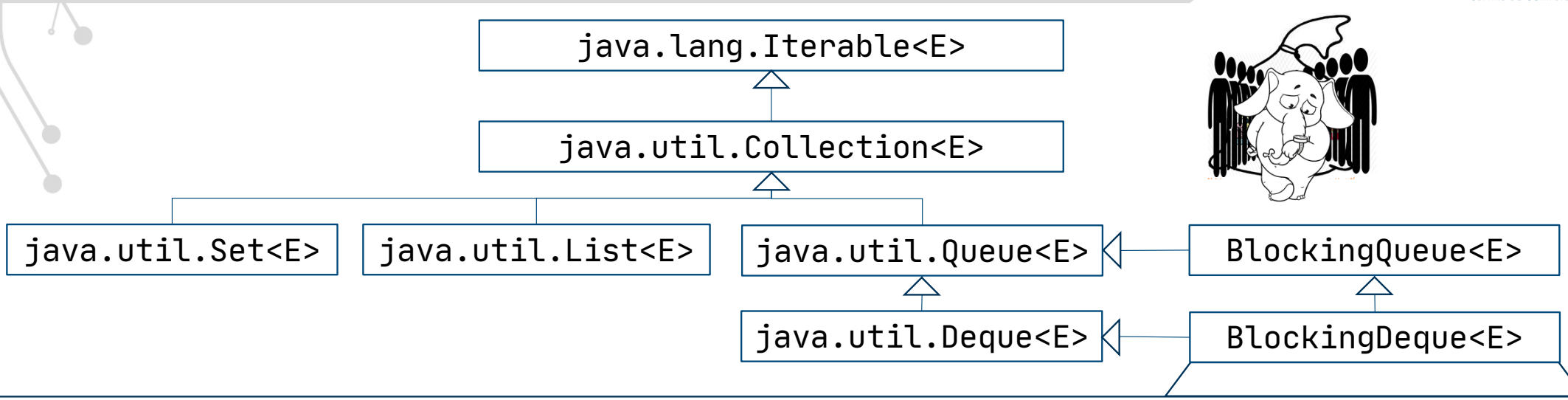


Queue permettant de mettre le thread utilisateur en attente :

- S'il n'y a pas d'éléments.
- Si la collection est remplie.

	Throws exception	Special value	Blocks	Times out
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	not applicable	not applicable

JAVA.UTIL.BLOCKINGDEQUE<E>



Deque permettant de mettre le thread utilisateur en attente :

- S'il n'y a pas d'éléments.
- Si la collection est remplie.

First Element (Head)				
	Throws exception	Special value	Blocks	Times out
Insert	addFirst(e)	offerFirst(e)	putFirst(e)	offerFirst(e, time, unit)
Remove	removeFirst()	pollFirst()	takeFirst()	pollFirst(time, unit)
Examine	getFirst()	peekFirst()	not applicable	not applicable
Last Element (Tail)				
Insert	addLast(e)	offerLast(e)	putLast(e)	offerLast(e, time, unit)
Remove	removeLast()	pollLast()	takeLast()	pollLast(time, unit)
Examine	getLast()	peekLast()	not applicable	not applicable

JAVA.UTIL.SORTEDSET<E>

- Range-view
  - `+ subSet(fromElement:E, toElement:E):SortedSet<E>`
  - `+ headSet(toElement:E):SortedSet<E>`
  - `+ tailSet(toElement:E):SortedSet<E>`
- Comparator access
  - `+ comparator():Comparator<? super E>`
- Endpoints
  - `+ first():E`
  - `+ last():E`

JAVA.UTIL.NAVIGABLESET<E>

- Range-view
  - + subSet(from:E,fromInclusive:boolean,to:E,toInclusive:boolean):NagigableSet<E>
  - + headSet(toElement:E,inclusive:boolean):SortedSet<E>
  - + tailSet(fromElement:E,inclusive:boolean):SortedSet<E>
- Query operations
  - + ceilling(e:E):E
  - + floor(e:E):E
  - + higer(e:E):E
  - + lower(e:E):E

JAVA.UTIL.MAP<K,V>

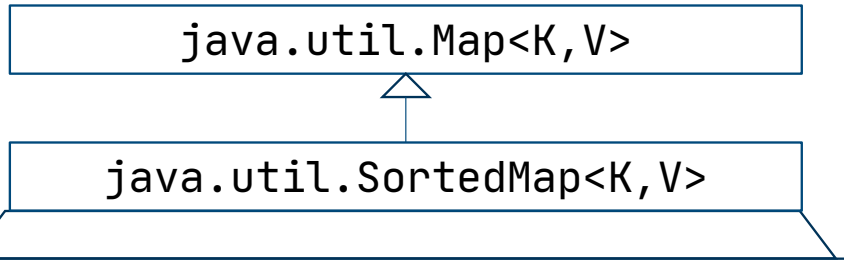
java.util.Map<K,V>

- Query Operations
  - + size():int
  - + isEmpty():boolean
  - + containsKey(key:Object):boolean
  - + containsValue(value:Object):boolean
  - + get(key:Object):V
- Modification operations
  - + put(key:K, value:V):V
  - + remove(key:Object):V
- Bulk Operations
  - + putAll(m:Map)
  - + clear()
- Views
  - + keySet():Set<K>
  - + values():Collection<V>
  - + entrySet():Set<Map.entry<K,V>>

java.util.Map.Entry<K,V>

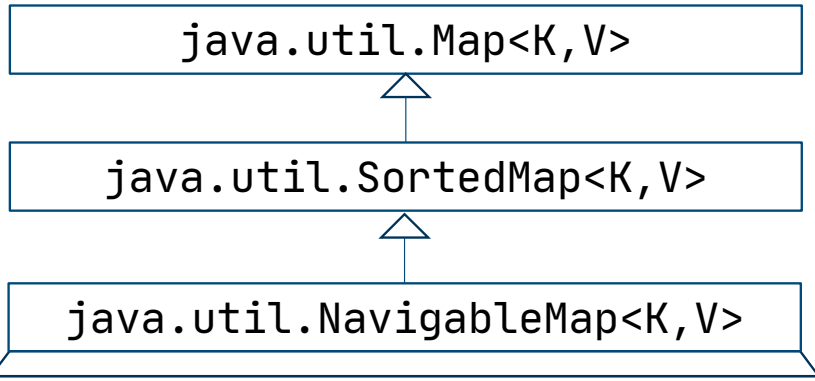
- + getKey():K
- + getValue():V
- + setValue(Value:V)

JAVA.UTIL.SORTEDMAP<K,V>



- Range-view
  - `+ subMap(fromKey:K, toKey:K):SortedMap<K,V>`
  - `+ headMap(toKey:K):SortedMap<K,V>`
  - `+ tailMap(fromKey:K):SortedMap<K,V>`
- Comparator access
  - comparator():Comparator<? Super K>
- Endpoints
  - firstKey():K
  - lastKey():K

JAVA.UTIL.NAVIGABLEMAP<K,V>



- Access operations
  - + lowerEntry(K key):Map.Entry<K,V>
  - + lowerKey(K key):K
  - + floorEntry(K key):Map.Entry<K,V>
  - + floorKey(K key):K
  - + ceilingEntry(K key)Map.Entry<K,V>
  - + ceilingKey(K key):K
  - + higherEntry(K key)Map.Entry<K,V>
  - + higherKey(K key):K
  - + firstEntry():Map.Entry<K,V>
  - + lastEntry():Map.Entry<K,V>
- Modification operations
  - + pollFirstEntry()Map.Entry<K,V>
  - + pollLastEntry():Map.Entry<K,V>
- Range-view
  - + descendingMap():NavigableMap<K,V>
  - + navigableKeySet():NavigableSet<K>
  - + descendingKeySet()NavigableSet<K>
  - + subMap(from:K, inclusive:boolean, to:K, inclusive:boolean): NavigableMap<K,V>
  - + tailMap(from:K, inclusive:boolean): NavigableMap<K,V>
  - + headMap(to:K, inclusive:boolean): NavigableMap<K,V>

IMPLEMENTATION GÉNÉRALE

TECHNIFUTUR
CENTRE DE COMPETENCES

Interfaces	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
SortedSet			TreeSet		
List		ArrayList		LinkedList	
Queue		ArrayDeque		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap
SortedMap			TreeMap		

- PriorityQueue est une implémentation sous forme de tas à priorité pour Queue.

IMPLÉMENTATIONS PARTICULIÈRES

TECHNIFUTUR
CENTRE DE COMPETENCES

- Set
  - EnumSet
  - CopyOnWriteArraySet
- List
  - Vector
  - CopyOnWriteArrayList
- Map
  - EnumMap
  - WeakHashMap
  - IdentyHashMap

WRAPPER IMPLEMENTATIONS

- Méthode static dans la classe utilitaire java.util.Collections
- Pour la Synchronisation
  + <T> synchronizedCollection(c:Collection<T>):Collection<T>
  + <T> synchronizedSet(s:Set<T>):Set<T>
  + <T> synchronizedList(list:List<T>):List<T>
  + <K,V> synchronizedMap(m:Map<K,V>):Map<K,V>
  + <T> synchronizedSortedSet(s:SortedSet<T>):SortedSet<T>
  + <K,V> synchronizedSortedMap(m:SortedMap<K,V>):SortedMap<K,V>
- Wrappers non modifiables
  + <T> unmodifiableCollection(c:Collection<? extends T>):Collection<T>
  + <T> unmodifiableSet(s:Set<? extends T>):Set<T>
  + <T> unmodifiableList(list:List<? extends T>):List<T>
  + <K,V> unmodifiableMap(m:Map<? extends K, ? extends V>):Map<K, V>
  + <T> unmodifiableSortedSet(s:SortedSet<? extends T>):SortedSet<T>
  + <K,V> unmodifiableSortedMap(m:SortedMap<K, ? extends V>):SortedMap<K, V>

ALGORITHMES

- Fonctionnalités réutilisable pour les Collections
  - Accessible dans la classe java.util.Collections
- Tri
  - Permet de réordonner les éléments d'une liste
    - Rapide : n log(n)
    - Stable
    - Ordre « naturel » / via Comparator
- Mélange
- Manipulation des données
  - Inverser : reverse
  - Remplir : fill
  - Copier : copy
  - Inversion d'éléments : swap
  - Ajout des éléments d'une collection ou d'un tableau dans une autre: addAll
- Recherche
  - Recherche dichotomique : binarySearch
- Composition
  - Fréquence d'un élément dans une collection
  - Tester si 2 collections sont disjointes
- Min max

TECHNIFUTUR
CENTRE DE COMPETENCES

PROGRAMMATION FONCTIONNELLE

Interfaces fonctionnelles

Expressions lambda

Collections de flux et filtres

INTRODUCTION

- Données partagées mutables vs Immutables
- Programmation déclarative
- Fonction sans effet secondaires
  - Effets secondaires internes
    - Effets non visibles en externe (attention multithread)
    - Masquer les effets secondaires des fonctions appelées
  - Effets secondaires externes
- Ne pas générer d'exceptions
  - Optional
- Transparence référentielle
  - Une fonction est référentiellement transparente si elle renvoie toujours la même valeur de résultat lorsqu'elle est appelée avec la même valeur d'argument.
    - Pas de structure de mutation visible
    - Pas E/S
    - Pas d'exception

INTERFACES FONCTIONNELLES

TECHNIFUTUR
CENTRE DE COMPETENCES

Interface ayant une et une seul méthode abstraite différente d'une méthode publique de Object.

- Valides
  - interface Runnable { void run(); }
- Non valides
  - interface NonFunc { boolean equals(Object obj); }
- Valides
  - interface Func extends NonFunc { int compare(String o1, String o2); }

  - interface Comparator<T> {
  - boolean equals(Object obj);
  - int compare(T o1, T o2);
  - }
- Non valides
  - interface Foo {
  - int m();
  - Object clone();
  - }

INTERFACES FONCTIONNELLES INTÉGRÉES DANS L'API

TECHNIFUTUR
CENTRE DE COMPETENCES

parm 1	param 2	Type de retour						
		void	boolean	int	long	double	T	R
		Runnable	BooleanSupplier	IntSupplier	LongSupplier	DoubleSupplier	Supplier<T>	
int		IntConsumer	IntPredicate	IntUnaryOperator	IntToLongFunction	IntToDoubleFunction		IntFunction<R>
long		LongConsumer	LongPredicate	LongToIntFunction	LongUnaryOperator	LongToDoubleFunction		LongFunction<R>
double		DoubleConsumer	DoublePredicate	DoubleToIntFunction	DoubleToLongFunction	DoubleUnaryOperator		DoubleFunction<R>
T		Consumer<T>	Predicate<T>	ToIntFunction<T>	ToLongFunction<T>	ToDoubleFunction<T>	UnaryOperator<T>	Function<T,R>
int	int			IntBinaryOperator				
long	long				LongBinaryOperator			
double	double					DoubleBinaryOperator		
T	T						BinaryOperator<T>	
T	double	ObjDoubleConsumer<T>						
T	int	ObjIntConsumer<T>						
T	long	ObjLongConsumer<T>						
T	U	BiConsumer<T,U>	BiPredicate<T,U>	ToIntBiFunction<T,U>	ToLongBiFunction<T,U>	ToDoubleBiFunction<T,U>		BiFunction<T,U,R>

Supplier | Consumer | Predicate | UnaryOperator | BinaryOperator

EXPRESSIONS LAMBDA

TECHNIFUTUR
CENTRE DE COMPETENCES

- Des classes anonymes aux expressions Lambda
  - Classes anonymes
  - Interface fonctionnelle
  - Expression Lambda
- Syntaxe des expressions lambda
- Référence à des méthodes existantes

CLASSE ANONYME

TECHNIFUTUR
CENTRE DE COMPETENCES

Implémentation dans une classe de premier niveau

```
public class MyPersonneComparator implements Comparator<Personne>
{
    @Override
    public int compare(Personne p1, Personne p2) {
        int diff = p1.getName().compareTo(p2.getName());
        if (diff == 0) diff =
p1.getPrenom().compareTo(p2.getPrenom());
        if (diff == 0) diff =
p1.getNaissance().compareTo(p2.getNaissance());
        return diff;
    }
}
```

Implémentation dans un classe anonyme

```
private static Comparator<Personne> getAnonymousComparator() {
    return new Comparator<Personne>() {
        @Override
        public int compare(Personne p1, Personne p2) {
            int diff = p1.getPrenom().compareTo(p2.getPrenom());
            if (diff == 0) diff = p1.getName().compareTo(p2.getName());
            if (diff == 0) diff =
p1.getNaissance().compareTo(p2.getNaissance());
            return diff;
        }
    };
}
```

TECHNIFUTUR
CENTRE DE COMPETENCES

Par une expression lambda sur plusieurs lignes

```
private static Comparator<Personne> getMultilineLambdaComparator()
{
    return (p1, p2) -> {
        int diff = p1.getNaissance().compareTo(p2.getNaissance());
        if (diff == 0) diff =
p1.getName().compareTo(p2.getName());
        if (diff == 0) diff =
p1.getPrenom().compareTo(p2.getPrenom());
        return diff;
    };
}
```

Par une expression lambda sur un ligne

```
(p1, p2) -> p1.getAge() - p2.getAge()
```

SYNTAXE DES EXPRESSIONS LAMBDA

TECHNIFUTUR
CENTRE DE COMPETENCES

Paramètres -> Corps

- Paramètres (listes de paramètres, entre parenthèses, séparé par des virgules).
  - Les paramètres sont
    - soit tous des identifiants
    - soit tous des déclarations de paramètres
      - Modificateur(final, annotation) optionel
      - Type
      - Identifiant
  - S'il n'y a qu'un paramètre sous la forme d'identifiant les parenthèses sont facultatives..
- Corps (soit une expression soit un bloc de code)
  - Expression
    - La valeur de retour de l'expression lambda est celle de l'expression.
  - Bloc de code (même syntaxe que le corps d'une fonction)
    - Si le type de retour est void et que le corps ne contient qu'une instruction les accolades sont facultatives.

RÉFÉRENCE À DES MÉTHODES EXISTANTES

TECHNIFUTUR
CENTRE DE COMPETENCES

- Lorsque qu'une expression lambda consiste à appeler une fonction en passant les paramètres reçus, Il est possible d'implémenter l'interface fonctionnelle avec la référence d'un méthode existante.
  - `Arrays.sort(roster, (a, b) -> Person.compareByAge(a, b) );`
  - `Arrays.sort(roster, Person::compareByAge);`
- Il existe 4 types de références de mèthodes :
  - Référence à une méthode « static »
    - `NomClasse::nomMethodeStatic | (a,b)->NomClasse.nomMethodeStatic(a,b)`
  - Référence à la méthode d'instance d'un objet particulier
    - `refObjet::nomMethodeInstance | (a,b)-> refObjet.nomMethodeInstance(a,b)`
  - Référence à la méthode d'instance d'un objet arbitaire
    - `NomClasse ::nomMethodeInstance | (a,b)-> a.nomMethodeInstance(b)`
  - Référence à un constructeur
    - `NomClasse ::new | (a,b)-> new NomClasse(a,b)`

COLLECTIONS DE FLUX ET FILTRES

TECHNIFUTUR
CENTRE DE COMPETENCES

- Pipelines et Streams
- Opérations de Stream
- Réduction
  - Stream.reduce()
  - Stream.collect()
- Parallélisme

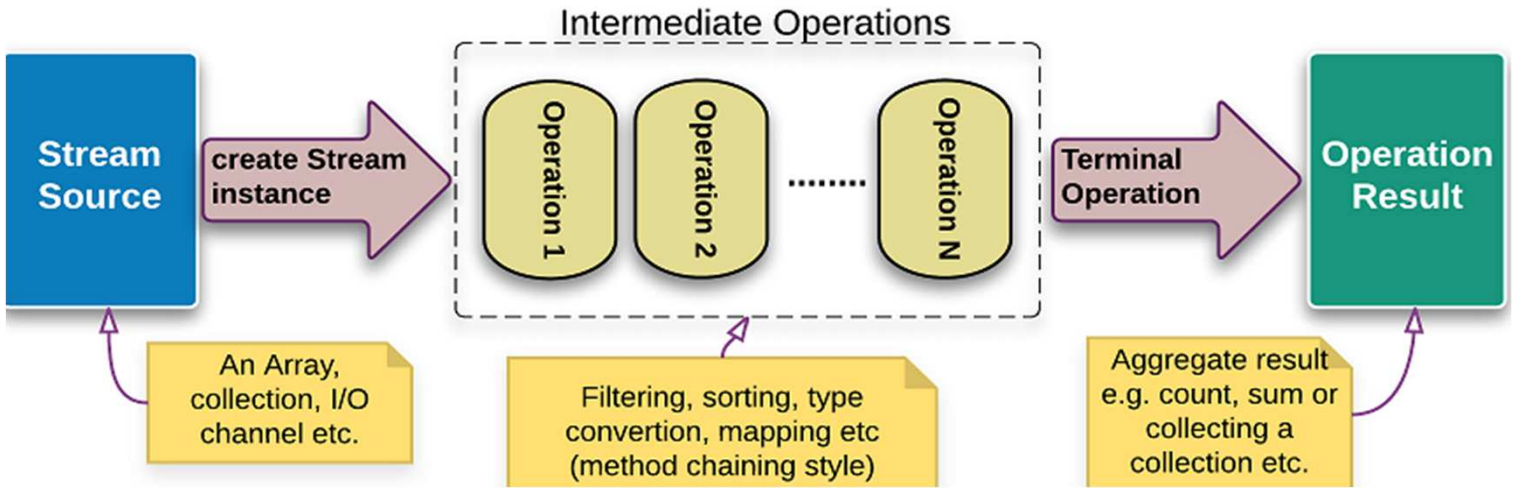
POURQUOI UTILISER LES STREAMS

TECHNIFUTUR
CENTRE DE COMPETENCES

- Utilisation des collections de manière déclarative
  - Spécification de l'intention (ce qui est à réaliser)
  - Utilisation d'une implémentation existante
- Composable
  - Il est possible de combiner des actions entre elles
- Traitement multithread très facile
- Concision du code

PIPELINES ET STREAMS

- Un Stream est un flux d'éléments.
  - Il ne stocke pas d'éléments.
  - Il transporte des valeurs d'une source via un pipeline.
- Un pipeline est une suite d'opérations d'agrégation composé de:
  - Une source à l'origine du Stream : une collection, un tableau, une fonction génératrice, Un canal d'entrées/sorties
  - De 0 à plusieurs opérations intermédiaires qui à partir d'un Stream produisent un nouveau Stream.
  - Une opération terminale qui produit un résultat qui n'est plus un Stream

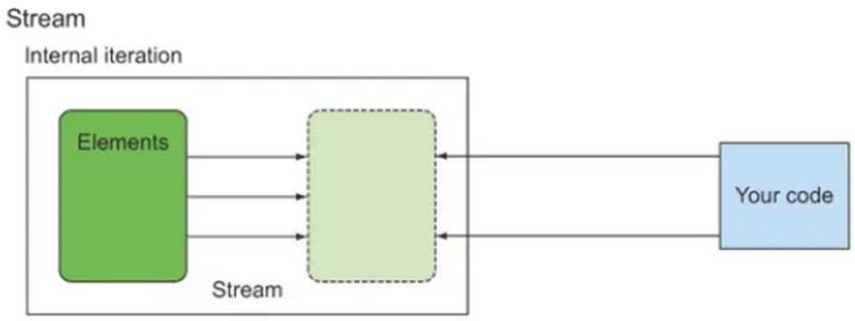


DIFFÉRENCES ENTRE LES COLLECTIONS ET LES STREAMS

TECHNIFUTUR
CENTRE DE COMPETENCES

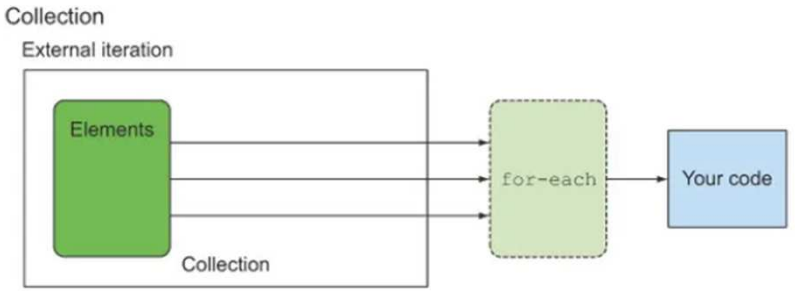
Stream

- Un Stream ne peut être utilisé qu'une seul fois.
- Concernent les calculs
- Ensemble de valeurs réparties dans le temps.
- Utilise une itération interne



Collection

- Une collection peut être utilisée plusieurs fois
- Concernent les données
- Ensemble de valeurs réparties dans l'espace.
- Nécessite une itération externe (par l'utilisateur)



PROPRIÉTÉS DES STREAMS

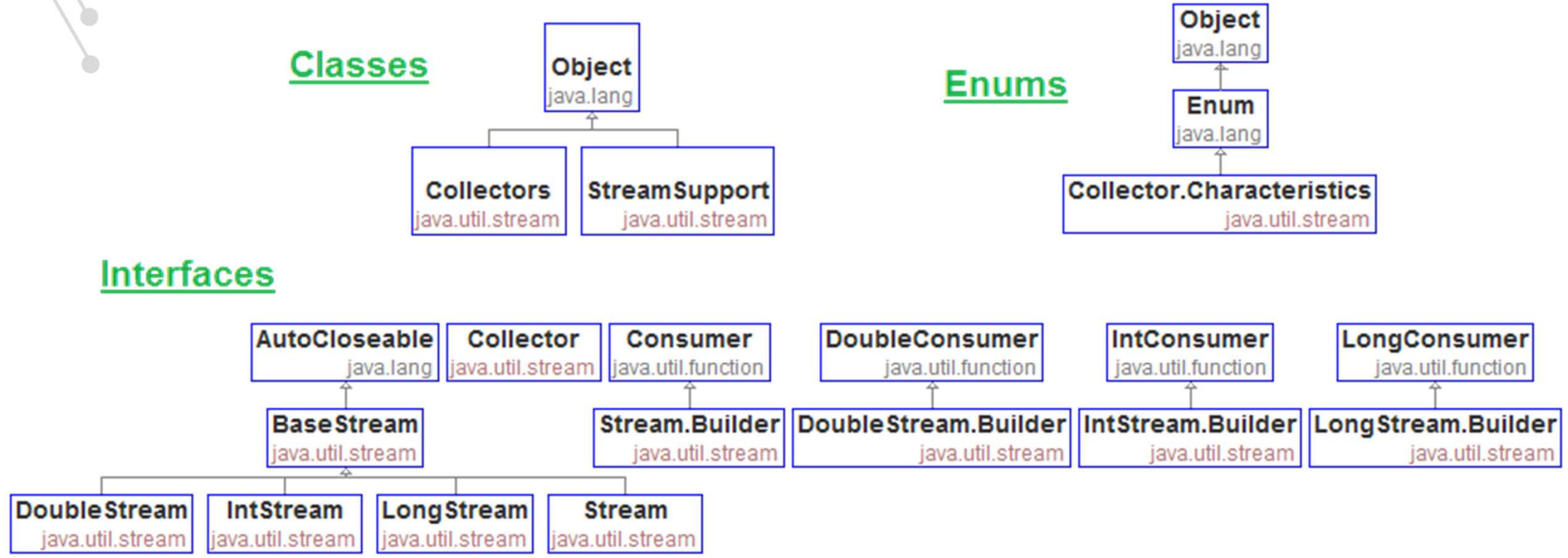
TECHNIFUTUR
CENTRE DE COMPETENCES

- Un Stream ne peut être parcouru qu'une seul fois
  - C'est l'exécution de l'opération terminal qui démarre et ferme l'utilisation du Stream.
  - Tous les éléments du Stream ne sont pas forcément parcouru (le parcours peut s'arrêter lorsque le résultat est trouvé).
- Le traitement par Pipeline est optimisé
  - Toutes les opérations intermédiaires sont « lazy ». Elle ne calcule un résultat qu'au fur et à mesure de la demande.
  - Les opérations intermédiaires sont soit stateless soit statefull.
- Le Parcours d'un Stream peut se faire soit séquentiellement soit parallèlement.
- Le parcours d'un Stream ne modifie pas les données de la source

L'API JAVA PROPOSE :

- 4 types de Stream en fonctions du type d'éléments parcourus :
  - Stream : éléments de type référence.
  - IntStream : éléments de type primitif int.
  - LongStream : éléments primitif de type long.
  - DoubleStream : élément primitif de type double.
- 2 types de parcours :
  - Séquentiel : Les éléments sont parcourus les un après les autres.
  - En Parallèle : Les éléments sont divisés en sous ensembles parcourus en parallèle dans des threads différents.
- De nombreuses techniques pour créer des Stream.
- Les Stream proposent des opérations combinables pour la création de pipelines
  - Des opérations intermédiaires de différents types (stateless , statefull, short-circuiting)
  - Des opérations terminales.
- Des classes utilitaires
  - Stream.Builder : pour la construction de Stream à partir d'éléments générer individuellement
  - Collectors : utilitaire proposant des implémentations d'opérations de réduction
  - StreamSupport : Utilitaire de bas niveau pour la création de Stream

java.util.stream



CRÉATION D'UN STREAM

TECHNIFUTUR
CENTRE DE COMPETENCES

- Un Stream vide
  - `Stream : static <T> Stream<T> empty()`
- Un Stream à partir d'une série de valeurs connues avant la construction du Stream
  - `Stream : static <T> Stream<T> of(T… values)`
  - `Arrays : static <T> Stream<T> stream( T[] array)`
  - `Collection : default Stream<E>stream()`
  - `Stream : static <T>Stream.Builder<T>builder()`
- Un Stream dont les éléments sont découverts pendant le parcours du Stream.
  - `Stream : static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)`
  - `Stream : static <T> Stream<T> iterate(T seed,UnaryOperator<T> f)`
  - `Stream : static <T> Stream<T> generate(Supplier<? extends T> s)`
- Stream créer par un objet de l'API
  - `BufferedReader, Files, Random, BitSet, Pattern, JarFile`

CRÉATION D'UN STREAM AVEC UN SPLITERATOR

TECHNIFUTUR
CENTRE DE COMPETENCES

- StreamSupport :
  - stream(Spliterator<T> spliterator, boolean parallel)
  - stream(Supplier<? extends Spliterator<T>> supplier, int characteristics, boolean parallel)
- La classe utilitaire Spliterators propose des factories pour la créations de Spliterators.
  - spliterator(Object[] array, int additionalCharacteristics)
  - spliterator(Object[] array, int fromIndex, int toIndex, int additionalCharacteristics)
  - spliterator(Collection<? extends T> c, int characteristics)
  - spliterator(Iterator<? extends T> iterator, long size, int characteristics)
  - spliteratorUnknownSize(Iterator<? extends T> iterator, int characteristics)
  - emptySpliterator()
- Autres méthodes
  - Les méthode équivalentes pour les Spliterator de type primitif (int, long, double)
  - Des méthodes pour construire un Iterator à partir de Spliterator

OPÉRATIONS INTERMÉDIAIRES

TECHNIFUTUR
CENTRE DE COMPETENCES

- Les opérations intermédiaires retournent un autre flux en tant que type de retour.
  - Sélectionner certains éléments :
    - Stream<T> skip(long n)
    - default Stream<T> dropWhile(Predicate<? super T> predicate)
    - (stateless) Stream<T> limit(long maxSize)
    - (statefull) default Stream<T> takeWhile(Predicate<? super T> predicate)
    - Stream<T> distinct()
    - Stream<T> filter(Predicate<? super T> predicate)
  - Transformer les éléments :
    - <R> Stream<R> map(Function<? super T,? extends R> mapper)
    - <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
  - Modifier l'ordre des éléments
    - Stream<T> sorted()
    - Stream<T> sorted(Comparator<? super T> comparator)
  - Action générique sur les éléments
    - Stream<T> peek(Consumer<? super T> action)

OPÉRATION FINALES PRÉDÉFINIES

TECHNIFUTUR
CENTRE DE COMPETENCES

- Tester la collection
  - boolean allMatch(Predicate<? super T> predicate)
  - boolean noneMatch(Predicate<? super T> predicate)
  - boolean anyMatch(Predicate<? super T> predicate)
- Sélectionner un éléments
  - Optional<T> findFirst()
  - Optional<T> findAny()
  - Optional<T> min(Comparator<? super T> comparator)
  - Optional<T> max(Comparator<? super T> comparator)
- Compter les élémnts
  - long count()
- Faire un opération sur chaque élément
  - void forEach(Consumer<? super T> action)
  - void forEachOrdered(Consumer<? super T> action)
- Stocker les résultats dans un tableau
  - Object[] toArray()
  - <A> A[] toArray(IntFunction<A[]> generator)

OPÉRATIONS FINALES CONFIGURABLES

TECHNIFUTUR
CENTRE DE COMPETENCES

- Opération qui parcours un Stream pour créer un résultat.
- L'opération est définie par :
  - De la valeur du résultat s'il n'y a pas d'éléments. (identity)
  - Une opération calculant un résultat à partir d'un résultat partiel et d'un élément (accumulator)
    - reduce() : création d'un nouveau résultat à chaque étape
      - L'opération retourne le nouveau résultat
      - Le résultat peut-être d'un type non modifiable
    - collect() : le résultat est modifier à chaque étape
      - L'opération met à jour le paramètre résultat
      - Le résultat doit être un type modifiable
  - D'une opération calculant un résultat à partir de 2 résultats partiels (combiner)
    - L'opération doit être associative : A op B op C = A op (B op C)
    - combiner.apply(identity, u) = u
    - combiner.apply(u, accumulator.apply(identity, t)) = accumulator.apply(identity, t)

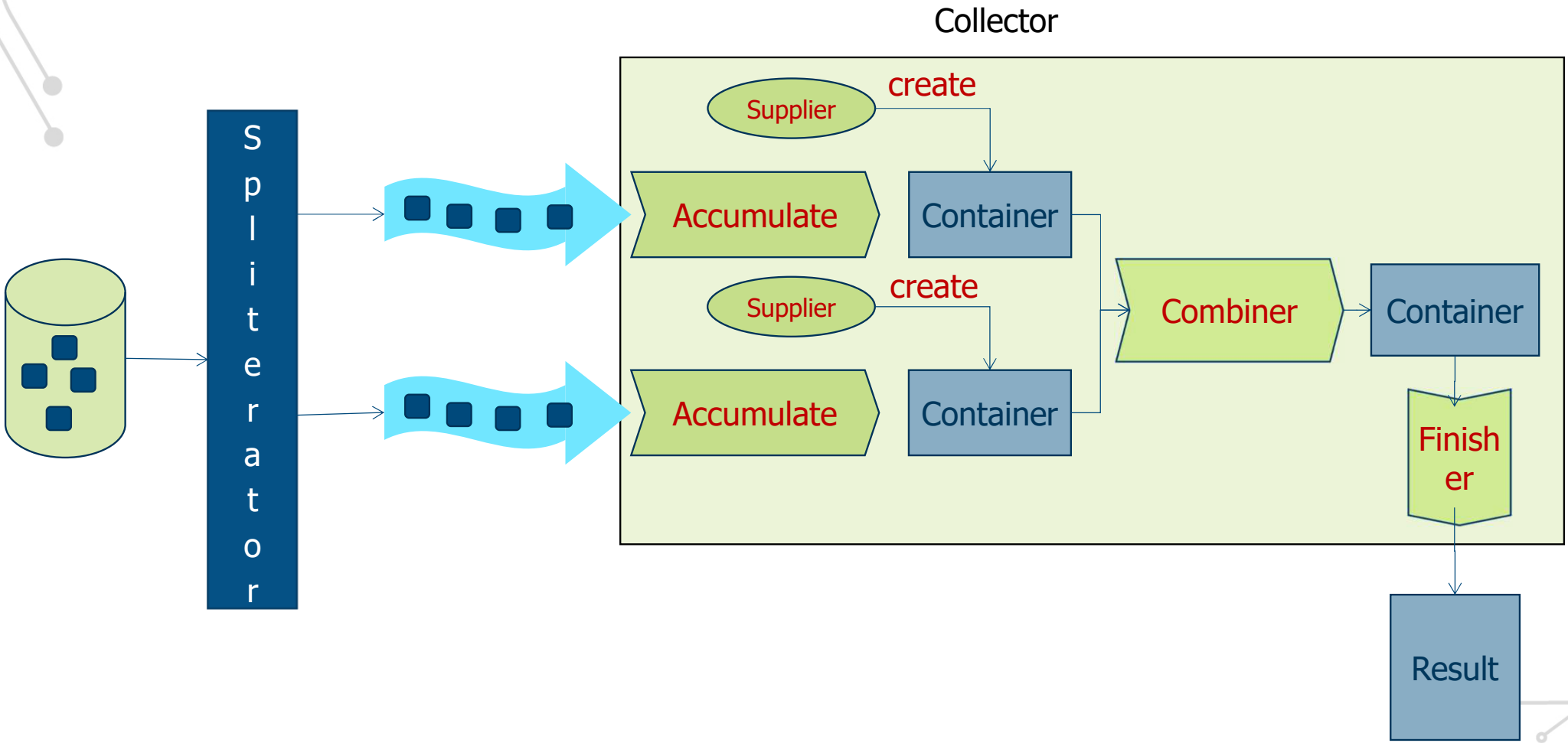
REDUCE

- <U> U reduce(U identity,
  BiFunction<U,? super T,U> accumulator,
  BinaryOperator<U> combiner)
  - U : type du résultat et des résultats partiels
  - T : type des éléments du Stream
  - identity : premier résultat partiel => valeur par défaut
  - accumulator : fonction calculant un résultat à partir
    - du résultat partiel précédent (1er paramètre)
    - et de l'élément suivant (2ème paramètre)
  - combiner : fonction calculant un résultat partiels à partir de 2 résultats partiels.
- Formes simplifiées :
  - T reduce(T identity, BinaryOperator<T> accumulator)
    - Le résultat est du même type que les éléments.
    - Le combiner = l'accumulator
  - Optional<T> reduce(BinaryOperator<T> accumulator)
    - Le premier résultat = le premier élément
    - Retourne Optinal.empty() si Stream vide

COLLECT

- <R> R collect(Supplier<R> supplier,
  BiConsumer<R,? super T> accumulator,
  BiConsumer<R,R> combiner)
  - R : type du résultat et des résultats partiels
  - T : type des éléments du Stream
  - Supplier : fonction sans paramètre qui construit le résultat dans son etat initial
  - accumulator : procédure modifiant le résultat (1er paramètre) à partir d'un élément (2ème paramètre)
  - combiner : modifiant le premier paramètre en le combinant avec le deuxième paramètre.
- <R,A> R collect(Collector<? super T,A,R> collector)

INTERFACE COLLECTOR<T,A,R>



COLLECTOR PRÉDÉFINIS

TECHNIFUTUR
CENTRE DE COMPETENCES

- La classe utilitaire java.util.stream.Collectors offre plusieurs Collectors pédéfinis.
  - Pour calculer un résultat
  - Pour stocker les éléments dans des collections
  - Pour calculer un résultat pour des sous ensembles de données
  - Pour adapter le comportement de Collector existant

COLLECTOR CAPABLE DE

- Concaténer des Strings
  - String: joining ([[CharSequence delimiter], CharSequence prefix, CharSequence suffix])
- Sélectionner un élément
  - Optional<T>: minBy(Comparator<? super T> comparator)
  - Optional<T>: maxBy(Comparator<? super T> comparator)
- D'effectuer une opération générique de réduction
  - T: reducing(T identity, BinaryOperator<T> op)
  - Optional<T>: reducing(BinaryOperator<T> op)
  - U: reducing(U identity, Function<? super T,? extends U> mapper, BinaryOperator<U> op)

COLLECTOR CAPABLE DE CALCULER UN RÉSULTAT NUMÉRIQUE

TECHNIFUTUR
CENTRE DE COMPETENCES

- Le nombre d'éléments
  - Long: counting()
- Une somme
  - Integer: summingInt(ToIntFunction<? super T> mapper)
  - Long: summingLong(ToLongFunction<? super T> mapper)
  - Double: summingDouble(ToDoubleFunction<? super T> mapper)
- Une moyenne
  - Double: averagingInt(ToIntFunction<? super T> mapper)
  - Double: averagingLong(ToLongFunction<? super T> mapper)
  - Double: averagingDouble(ToDoubleFunction<? super T> mapper)
- Un ensemble de données statistiques de base (nombre, somme, min, max, moyenne)
  - IntSummaryStatistics: summarizingInt(ToIntFunction<? super T> mapper)
  - LongSummaryStatistics: summarizingLong(ToLongFunction<? super T> mapper)
  - DoubleSummaryStatistics: summarizingDouble(ToDoubleFunction<? super T> mapper)

COLLECTOR CAPABLE DE CRÉER DES COLLECTIONS DE COLLECTIONS

TECHNIFUTUR
CENTRE DE COMPETENCES

- Collection<T>: toCollection(Supplier<C> collectionFactory)
- List<T>: toList()
- Set<T>: toSet()
- List<T>: toUnmodifiableList()
- Set<T>: toUnmodifiableSet()

COLLECTOR CAPABLE DE CRÉER UN DICTIONNAIRE DES ÉLÉMENTS

TECHNIFUTUR
CENTRE DE COMPETENCES

- toMap (keyMapper, valueMapper[, mergeFunction[, mapSupplier]]) ->M extends Map<K,U>
- toConcurrentMap (idem) ->M extends ConcurrentMap<K,U>
- toUnmodifiableMap(keyMapper, valueMapper[, mergeFunction]) -> Map<K,U>
  - keyMapper : Function<? super T,? extends K>
    - Retourne la clé à partir de l'élément
  - valueMapper : Function<? super T,? extends U>
    - Retourne la valeur à partir de l'élément
  - mergeFunction : BinaryOperator<U>
    - Retourne un valeur à partir des valeurs de deux éléments de même clé
  - mapSupplier : Supplier<M>
    - Retourne Retourne Map ou ConcurrentMap concrète du résultat
  - Avec
    - T : le type des éléments,
    - K : le type des clés,
    - U : le type des valeurs,
    - M : le type de la Map ou de la ConcurrentMap

COLLECTOR CAPABLE DE CRÉER 2 RÉSULTATS EN FONCTION D'UN TEST SUR LES ÉLÉMENTS

TECHNIFUTUR
CENTRE DE COMPETENCES

- partitioningBy(predicate [, downstream]) -> Map<Boolean,D>
  - predicate: Predicate<? super T>
    - Retourne vrai ou faux en fonction de l'élément
  - downstream : Collector<? super T,A,D>
    - Collector servant à créer le résultat de chacun de 2 groupes d'éléments
    - Par défaut : Collectors.toList()
  - Avec
    - T : le type des éléments,
    - A : le type intermédiaire de l'accumulateur,
    - D : le type des valeurs,

COLLECTOR CAPABLE DE CRÉER PLUSIEURS RÉSULTAT EN FONCTION D'UNE VALEUR CLÉ

TECHNIFUTUR
CENTRE DE COMPETENCES

- groupingBy(classifier [[, mapFactory,] downstream]) -> M extends Map<K,D>
- groupingByConcurrent (idem) ->M extends ConcurrentMap<K,D>
  - classifier : Function<? super T,? extends K>,
    - Retourne la clé à partir de l'élément
  - mapFactory : Supplier<M>,
    - Retourne Map ou ConcurrentMap concrète du résultat
  - downstream : Collector<? super T,A,D>
    - Retourne un collector qui construit une valeur à partir des éléments de mêmes clés
    - par défaut : Collectors.toList())
  - Avec
    - T : le type des éléments,
    - K : le type des clés,
    - D : le type des valeurs,
    - M : le type de la Map ou de la ConcurrentMap

COLLECTOR CAPABLE D'ADAPTER LE COMPORTEMENT D'AUTRES COLLECTORS

TECHNIFUTUR
CENTRE DE COMPETENCES

- En modifiant le type des éléments à collecter
  - <R>: mapping(Function<? super T,? extends U> mapper, Collector<? super U,A,R> downstream)
  - <R>: flatMapping(Function<? super T,? extends Stream<? extends U>> mapper, Collector<? super U,A,R> downstream)
- En modifiant le résultat d'un collector donné
  - Collector<T,A,RR> : collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)
- En filtrant les éléments reçus par le collector
  - <R>: filtering(Predicate<? super T> predicate, Collector<? super T,A,R> downstream)
- En combinant deux Collector
  - <R>: teeing(Collector<? super T,?,R1> downstream1, Collector<? super T,?,R2> downstream2, BiFunction<? super R1,? super R2,R> merger)

SPLITERATOR

- Objet pour utilisé pour
  - Manipuler les éléments de la sources de données
    - Individuellement: boolean tryAdvance(Consumer<? super T> action)
    - Globalement: default void forEachRemaining(Consumer<? super T> action)
  - Diviser en 2 les éléments de la source de données
    - Spliterator<T> trySplit()
- Chaque Spliterator possède un ensemble de caractéristiques :
  - ORDERED : un ordre est définit pour les éléments
  - DISTINCT : il n'y a pas de doublons dans les éléments
  - SORTED : les éléments sont triés
    - => Ordered est vrai
    - => getComparator() ne soulève pas d'exception, null si les éléments Comparable sont trié sur ce critère.
  - SIZED : si la taille de la source est connue
  - NONNULL : aucun éléments n'est null
  - IMMUTABLE : les éléments ne peuvent pas être modifiés durant l'utilisation du Spliterator
  - CONCURRENT : la source peut être modifiée durant l'utilisation du Spliterator
  - SUBSIZED : si la source est SIZED et que le résultat de trySplit sera SIZED et SUBSIZED