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 COMPÉTENCES

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 COMPÉTENCES

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 COMPÉTENCES

- 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

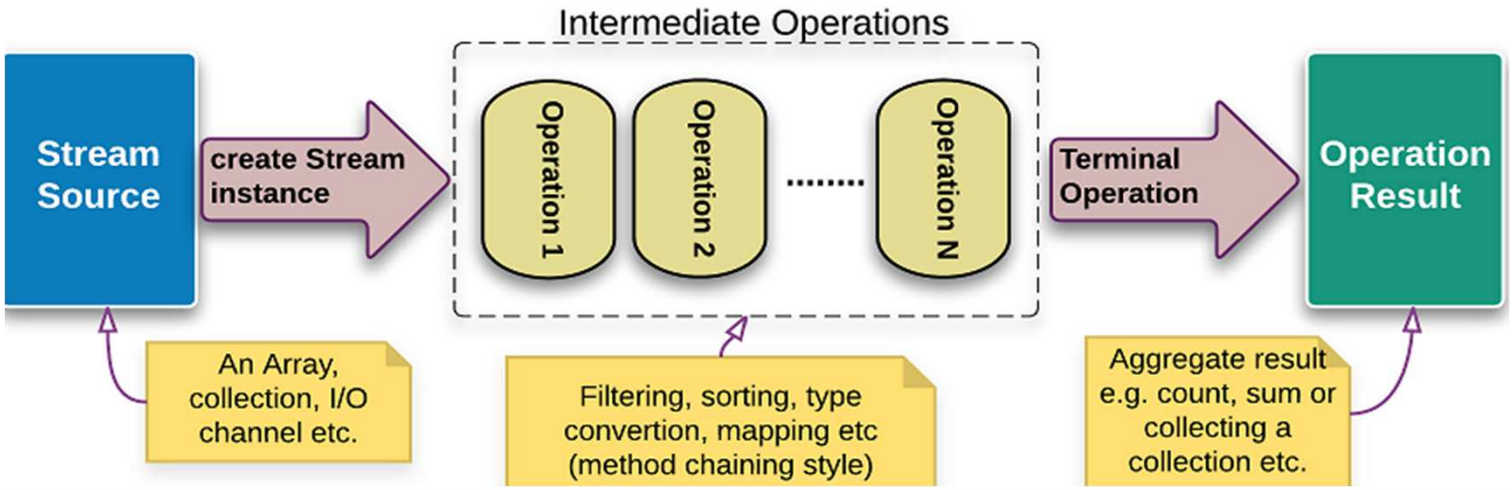
POURQUOI UTILISER LES STREAMS

TECHNIFUTUR
CENTRE DE COMPÉTENCES

- 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

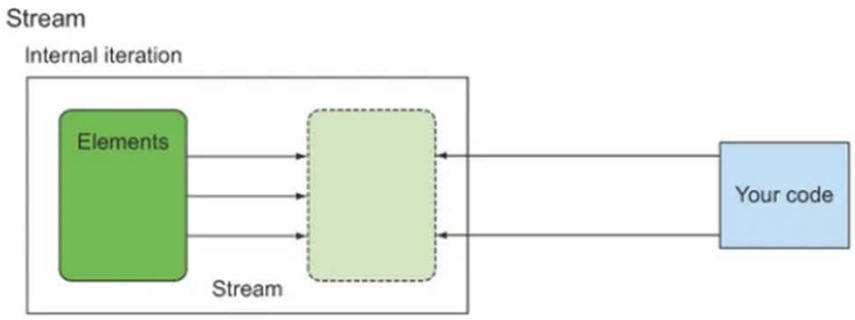


DIFFÉRENCES ENTRE LES COLLECTIONS ET LES STREAMS

TECHNIFUTUR
CENTRE DE COMPÉTENCES

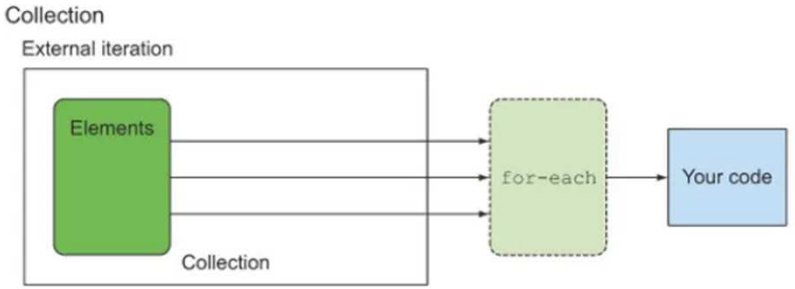
Stream

- 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

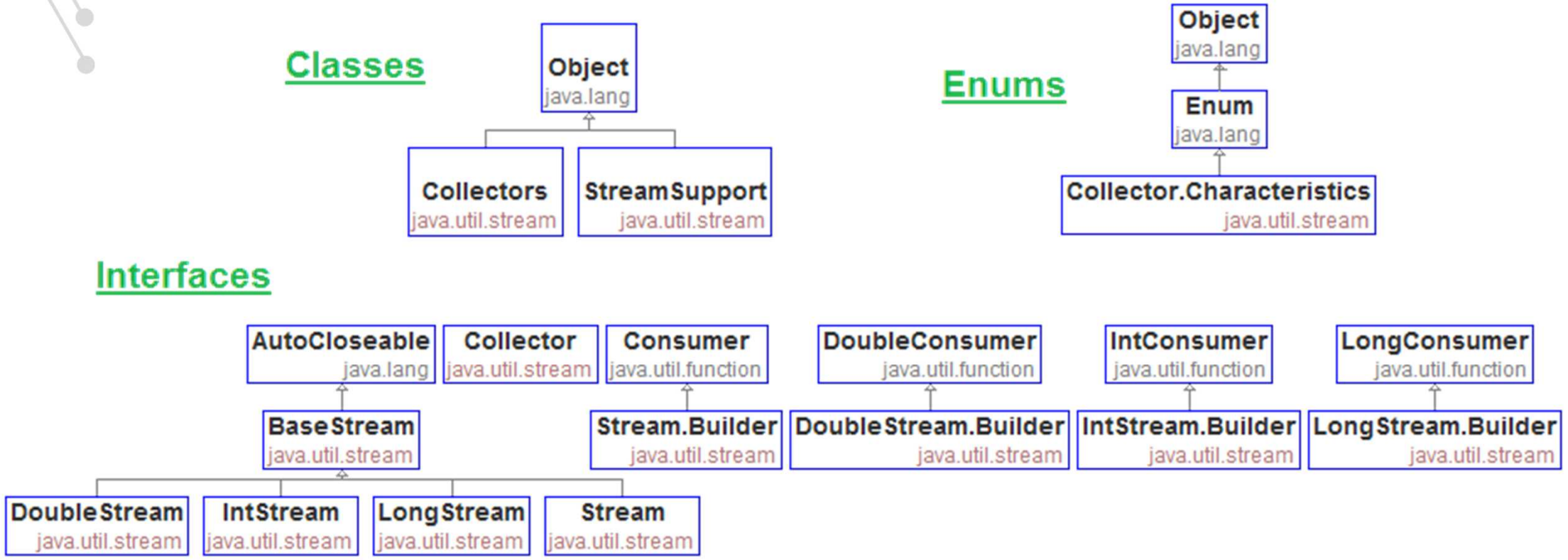
TECHNIFUTUR
CENTRE DE COMPÉTENCES

- 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 COMPÉTENCES

- 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 COMPÉTENCES

- 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 COMPÉTENCES

- 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 COMPÉTENCES

- 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)

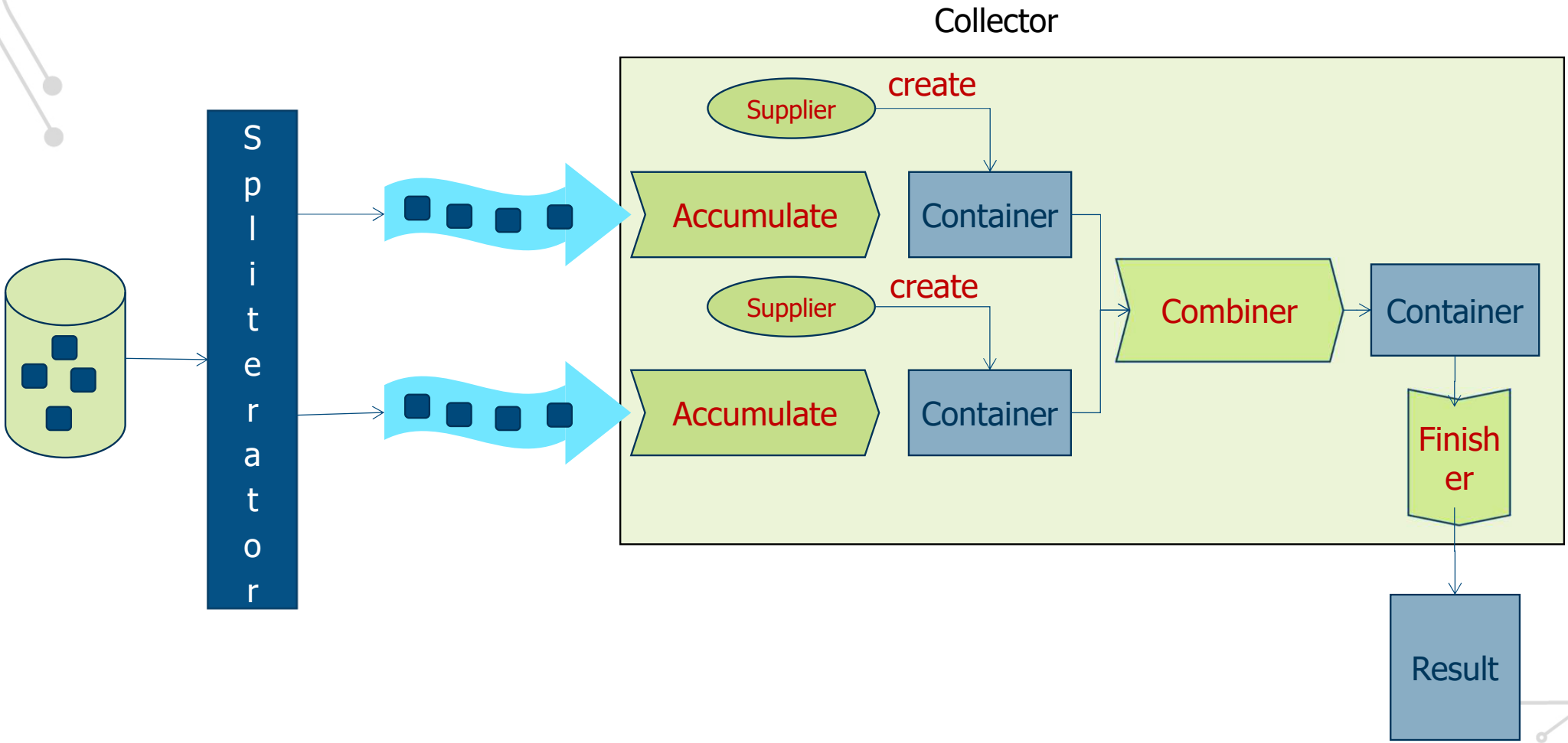
REDUCE

- <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 COMPÉTENCES

- 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 COMPÉTENCES

- 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 COMPÉTENCES

- 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 COMPÉTENCES

- 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 COMPÉTENCES

- 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 COMPÉTENCES

- 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

TECHNIFUTUR
CENTRE DE COMPETENCES

- 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