digitalent
FORMATION

Le langage Java

Java 8 - Lambda, Streams, Collectors

Expression Lambda

Expression Lambda

- Soit un compte courant et une liste

```
public class CompteCourant {
	String numero;
	String intitule;
	double solde;
	double montDecouvertAutorise;
}

List<CompteCourant> comptes = new ArrayList<>();
```

- Objectif => Calculer la moyenne des soldes

Expression Lambda

- Approche impérative

```
double somme = 0.0;
double moyenne = 0.0;

for(CompteCourant c : list) {
     somme += c.getSolde();
}

if(!list.isEmpty()) {
     moyenne = somme / list.size();
}
```

Expression Lambda

- Approche impérative (uniquement comptes positifs)

```
double somme = 0.0;
double moyenne = 0.0;
int nbComptes = 0;

for(CompteCourant c : list) {
	if(c.getSolde() > 0.0) {
		somme += c.getSolde();
		nbComptes++;
	}
}
if(!list.isEmpty()) {
	moyenne = somme / nbComptes;
}
```

Expression Lambda

- Approche fonctionnelle (SQL)

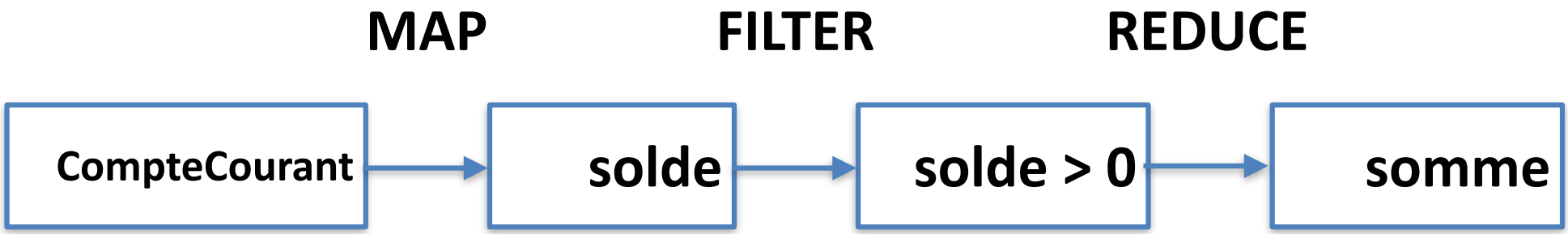
SELECT AVG (solde)

FROM COMPTE_COURANT

WHERE SOLDE > 0

Expression Lambda

- Approche fonctionnelle



Expression Lambda

- Approche fonctionnelle (JDK 7)

```
public interface Mapper<T, V> {
     public V map(T t);
}

public interface Predicate<T> {
     public boolean filter(T t);
}

public interface Reducer<T> {
     public T reduce(T t1, T t2);
}
```

```
list.stream().map(new Mapper<CompteCourant,Double>() {
     @Override
     public Double map(CompteCourant t) {
          return t.getSolde();
     }
}).filter(new Predicate<Double>() {
     @Override
     public boolean filter(Double t) {
          return t > 0;
     }
}).reduce(new Reducer<Double>() {
     @Override
     public Double reduce(Double t1, Double t2) {
          return t1+t2;
     }
});
```

Expression Lambda

- Approche fonctionnelle (JDK 8)

```
new Mapper<CompteCourant,Double>() {
      @Override
      public Double map(CompteCourant t) {
            return t.getSolde();
      }
}
```

devient

```
(CompteCourant t) -> t.getSolde()
```

ou

```
t -> t.getSolde()
```

Expression Lambda

- Approche fonctionnelle (JDK 8)

```
new Predicate<Double>() {
      @Override
      public boolean filter(Double t) {
            return t > 0;
      }
}
```

devient

```
(Double t) -> t > 0
```

ou

```
t -> t > 0
```

Expression Lambda

- Approche fonctionnelle (JDK 8)

```
list.stream()
        .map(t -> t.getSolde())
        .filter(t -> t > 0)
        .reduce((t1, t2) -> t1+t2);
```

Expression Lambda

- Si plusieurs lignes de code

```
list.stream()
          .map(t -> {
             System.out.println(t);
             return t.getSolde();
           })
          .filter(t -> t > 0)
          .reduce((t1, t2) -> t1+t2);
```

Expression Lambda

- Conditions pour utiliser une expression lambda
    - Une seule méthode dans l'interface
    - Les types de paramètres de l'unique méthode doivent être compatible avec les types de l'expression lambda

Expression Lambda

digitalent FORMATION

- Expression dans une variable

```
Function<CompteCourant, Double> mapper = t -> t.getSolde();
Predicate<Double> filter = t -> t > 0;
BinaryOperator<Double> reduce = (t1, t2) -> t1+t2;

list.stream()
     .map(mapper)
     .filter(filter)
     .reduce(reduce);
```

Expression Lambda

- Référence de méthode

```
mapper = t -> t.getSolde();

mapper = CompteCourant::getSolde;
```

Interface Fonctionnelle

Interface Fonctionnelle

- Les expressions lambda ne sont applicables que sur des interfaces dites “fonctionnelles”. Il s’agit d’interface avec **une seule méthode abstraite**.
    - Les interfaces existantes du JDK qui n’ont qu’une seule méthode abstraite peut-être vue comme des interfaces fonctionnelles

```
public interface Runnable {

    public abstract void run();

}
```

Interface Fonctionnelle

- L'annotation **@FunctionalInterface** permet de vérifier à la compilation qu'une interface est bien fonctionnelle au sens Java 8 (elle ne contient qu'une seule méthode abstraite).

```
@FunctionalInterface
public interface Runnable {

    public abstract void run();

}
```

Interface Fonctionnelle

- Java 8 fournit des interfaces fonctionnelles usuelles dans le package **java.util.function**.
    - Function<T,R>
    - BiFunction<T,U,R>
    - Consumer<T>
    - Supplier<T>
    - ...

Interface Fonctionnelle

- Function<T,R> : une méthode avec un paramètre de type T qui retourne une valeur de R.

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t);
…
}
```

Interface Fonctionnelle

- BiFunction<T,R> : une méthode avec un paramètre de type T et un autre de type U qui retourne une valeur de R.

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u);

}
```

Interface Fonctionnelle

- Consumer<T> : une méthode avec un paramètre de type T qui ne retourne aucun résultat.

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);
…
}
```

Interface Fonctionnelle

- Supplier<T> : une méthode sans paramètre qui retourne un résultat de type T.

```
@FunctionalInterface
public interface Supplier<T> {

    T get();
…
}
```

Interface Fonctionnelle

digitalent FORMATION

- Quel intérêt d'utiliser des interfaces fonctionnelles ?
  - Possibilité de déclarer une lambda comme paramètre d'une méthode ou comme variable.
  - Possibilité de chainer l'exécution d'expression lambda via les méthodes : **compose** & **andThen** fournis par certaines interfaces.

Interface Fonctionnelle

digitalent FORMATION

- Exemple **andThen** :

```
public String maMethode(BiFunction<Integer,Integer, Integer> b) {
   …
}


BiFunction<Integer, Integer, Integer> addition = (a,b) -> a+b;
Function<Integer, Integer> carre = (b) -> b*b;

maMethode(addition.andThen(carre));
```

API Stream

API Stream

- Les collections ont désormais une méthode :
    - **stream()** : création d'un flux pour traiter les données d'une source

```
somme = list.stream()
        .map(t -> t.getSolde())
        .filter(t -> t > 0)
        .reduce((t1, t2) -> t1+t2);
```

API Stream

- Un Stream ?
  - ne porte pas de données
  - ne peut pas modifier sa source
  - peut être infinie

API Stream

- Deux types d'opération
    - opérations **intermédiaires**
        - ne déclenchent pas de traitement
        - exemple : map, filter
    - opérations **terminales**
        - déclenchent un traitement
        - exemple : reduce, collect
- Un stream ne peut-être traité qu'une seule fois
- Une seule opération terminale est autorisée

API Stream

- Quelques opérations terminales
    - reduce()
    - count(), min(), max()
    - anyMatch(), allMatch(), noneMatch()
    - findFirst(), findAny()
    - toArray()
    - forEach(), forEachOrdered()

Collectors

Collectors

- Collectors ?
  - Classe utilitaire fournissant les réductions usuelles (30+ méthodes)
    - counting, minBy, maxBy
    - summing, averaging, summarizing
    - joining
    - toList, toSet
    - mapping, groupingBy, partionningBy

Collectors

digitalent FORMATION

```java
// Transformer un Stream en List
List<Person> liste1 = persons.stream().collect(Collectors.toList());

// Transformer un Stream en Set
Set<String> liste2 = persons.stream().collect(Collectors.toSet());

// Transformer un Stream en TreeSet
TreeSet<String> liste3 =
persons.stream().collect(Collectors.toSet(TreeSet::new));

// Concaténer les noms d'une liste de personnes
String names1 =
persons.stream().map(Person::getName).collect(Collectors.joining());
```

Collectors

digitalent FORMATION

```
// Concaténer les noms séparés par une virgule d'une liste de personnes
String names2 =
persons.stream().map(Person::getName).collect(Collectors.joining(","));

// Compter le nombre de personnes
int nbPersons = persons.stream().collect(Collectors.counting());

// Moyenne des ages des personnes
double moyenneAge =
persons.stream().collect(Collectors.averagingDouble(Person::getAge));

// Regroupement des personnes par age
Map<Integer, List<Person>> map =
persons.stream().collect(Collectors.groupingBy(Person::getAge));
```

Collectors

digitalent FORMATION

```
// Regroupement des personnes par age
Map<Integer, List<Person>> map =
persons.stream().collect(Collectors.groupingBy(Person::getAge));

// Regroupement des personnes par age en utilisant un Set
Map<Integer, Set<Person>> map =
persons.stream().collect(Collectors.groupingBy(Person::getAge,Collectors.toSet()
));

// Répartir les données en 2 ensembles : true -> liste des personnes age > 20 et
false -> le reste
Map<Boolean, List<Person>> map =
persons.stream().collect(Collectors.partitionningBy(p -> p.getAge() > 20));
```

Comparateurs

Comparateurs

- Nouvelle API pour construire les comparateurs

```
Comparator<Person> comp = Comparator.comparing(Person::getLastName)
                                    .thenComparing(Person::getFirstName)
                                    .thenComparing(Person::getAge);
```