



# Formation Java 17

## Expression Lambda

# Sommaire

Approche Fonctionnelle

Classe anonyme

Expression Lambda



# Chapitre 1

## L'approche fonctionnelle

# Un exemple

Soit une liste de comptes courants.

Notre objectif est de calculer la moyenne des soldes.

```
public class CompteCourant {  
    String numero;  
    String intitule;  
    double solde;  
    double montDecouvertAutorise;  
  
    // GET + SET  
}
```

# Avec une approche impérative

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

# Avec uniquement les soldes > 0

```
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 (nbComptes != 0) {
    moyenne = somme / nbComptes;
}
```

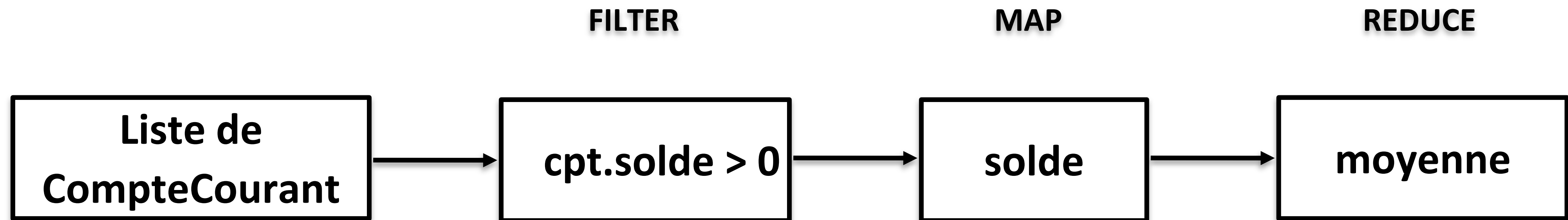
# Approche fonctionnelle (SQL)

Même chose en SQL

```
SELECT AVG (solde)  
FROM COMPTE_COURANT  
WHERE SOLDE > 0
```

# Approche fonctionnelle

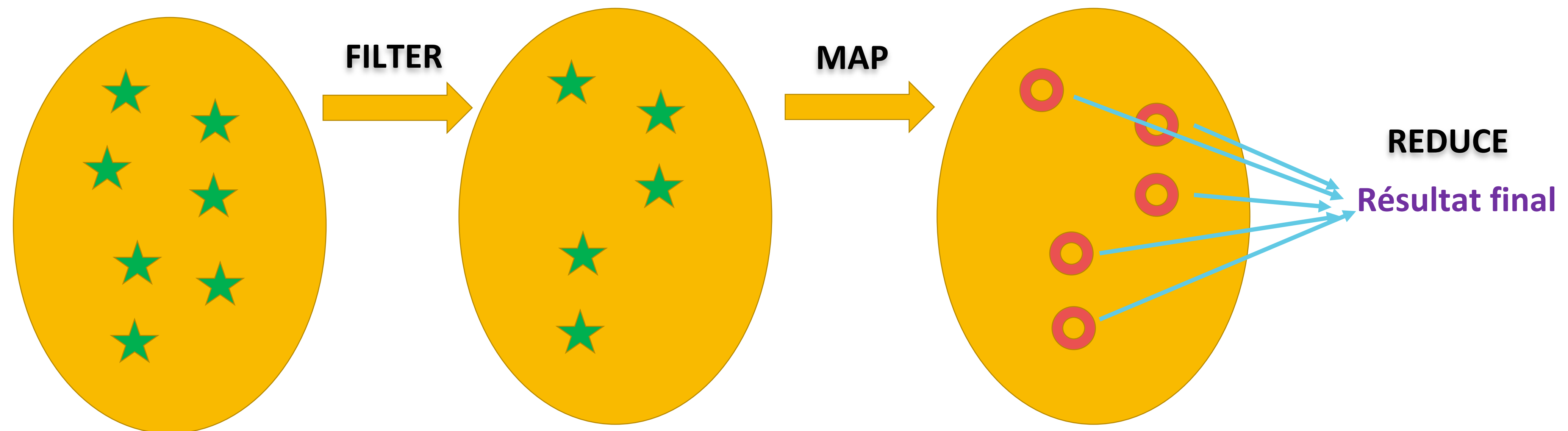
Application d'une suite de transformation à la collection





# Filter – Map - Reduce

Application d'une suite de transformation à la collection



# Chapitre 2


## Classe anonyme

# Comment implémenter une interface ?

Soit la méthode suivante :

```
public class Calcul {  
    public void exec(double a, double b, Operation op) {  
        // Code ici  
    }  
}
```

```
public interface Operation {  
    double apply(double a, double b);  
}
```



La méthode exec prend en 3<sup>ème</sup> paramètre une "Operation".

```
Calcul calcul = new Calcul();  
calcul.exec(10.0, 10.0, ???);
```

# Comment implémenter une interface ?

Première possibilité, je crée une classe qui implémente l'interface Operation:

Exemple:

```
public class Addition implements Operation {  
    public double apply(double a, double b) {  
        return a + b;  
    }  
}
```

```
public interface Operation {  
    double apply(double a, double b);  
}
```

Exemple d'invocation avec la classe Addition:

```
Calcul calcul = new Calcul();  
Addition addition = new Addition();  
calcul.exec(10.0, 10.0, addition);
```

# Comment implémenter une interface ?

**Solution 1:** La classe d'implémentation

**Avantages:**

- souple,
- écriture élégante

**Inconvénients:**

- chaque nouvel algorithme de calcul nécessite une nouvelle classe.
- Lourdeur en terme de nombre de classes.
- Pas la possibilité d'envoyer un algorithme de calcul directement en paramètre de méthode

# Comment implémenter une interface ?

Deuxième possibilité, je crée une classe anonyme qui implémente l'interface **Operation**:

- Utilisation de l'opérateur **new**
- Le corps de la classe anonyme est définie après l'appel du constructeur.

```
Operation addition = new Operation() {  
    public double apply(double a, double b) {  
        return a + b;  
    }  
};
```

```
public interface Operation {  
    double apply(double a, double b);  
}
```

# Comment implémenter une interface ?

Deuxième possibilité, mise en oeuvre:

```
Calcul calcul = new Calcul();
```

```
Operation addition = new Operation() {  
    public double apply(double a, double b) {  
        return a + b;  
    }  
};
```

```
calcul.exec(10.0, 10.0, addition);
```

```
public interface Operation {  
    double apply(double a, double b);  
}
```

# Comment implémenter une interface ?

## Solution 2: La classe anonyme

### Avantages:

- je peux définir un calcul à la volée
- Pas de nouvelle classe à chaque nouveau calcul.

### Inconvénients:

- écriture lourde.



# Comment implémenter une interface ?

**Solution: l'expression lambda !!**

# Chapitre 3

# Expression lambda

# Qu'est ce qu'une expression lambda ?

C'est l'écriture simplifiée d'une classe anonyme...une lambda est une implémentation d'une interface fonctionnelle (i.e. qui ne possède qu'une seule méthode abstraite).

Avec classe anonyme:

```
Calcul calcul = new Calcul();
```

```
Operation addition = new Operation() {  
    public double apply(double a, double b) {  
        return a + b;  
    }  
};
```

```
calcul.exec(10.0, 10.0, addition);
```

Avec une lambda:

```
Calcul calcul = new Calcul();
```

```
Operation addition = (a, b) -> a + b;
```

```
calcul.exec(10.0, 10.0, addition);
```

# Constitution lambda depuis une interface

Comment créer une expression lambda à partir d'une interface ?

Plusieurs étapes de simplification.

Remarques:

- Vous ne pouvez créer une lambda que pour une interface qui n'a qu'une méthode abstraite.
- Comme java va devoir déduire des informations, il est nécessaire qu'il n'y ait pas d'ambiguïté.

# Constitution lambda depuis une interface

**Etape 1:** on commence par écrire la classe anonyme à partir de l'interface fonctionnelle

```
Operation op = new Operation() {  
    public double apply(double a, double b) {  
        return a + b;  
    }  
};
```

```
public interface Operation {  
    double apply(double a, double b);  
}
```

# Constitution lambda depuis une interface

Etape 2: on supprime le code déclaratif (ce qui est en orange)

```
Operation op = new Operation() {  
    public double apply(double a, double b) {  
        return a + b;  
    }  
};
```

- On supprime la signature de la classe et celle de la méthode (à l'exception des paramètres).
- On supprime la dernière accolade fermante
- On ajoute l'opérateur "flèche" entre les paramètres et le corps de la méthode.

```
Operation op = (double a, double b) -> {  
    return a + b;  
};
```

# Constitution lambda depuis une interface

**Etape 3:** Java sait que les paramètres sont des doubles puisqu'il n'y a qu'une seule méthode. On supprime donc le type des paramètres.

```
Operation op = (a, b) -> {  
    return a + b;  
};
```

# Constitution lambda depuis une interface

## Etape 4 (finale):

- Si le corps de la méthode ne contient qu'une seule instruction, on peut supprimer les accolades.
- Si cette instruction unique est un return, il **faut** aussi supprimer la clause return.

```
Operation op = (a, b) -> {  
    return a + b;  
};
```

```
Operation op = (a, b) -> a + b;
```



# Constitution lambda depuis une interface

## Quelques compléments d'informations:

- Si la méthode de l'interface ne possède qu'un seul paramètre alors les parenthèses sont facultatives.

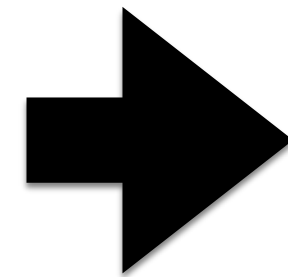
Predicate `p = a -> a.getSolde() >=1;`

- Si la méthode de l'interface ne possède pas de paramètre, on utilise la notation suivante:

Supplier `s = () -> new Compte();`

# Approche fonctionnelle (JDK 8)

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



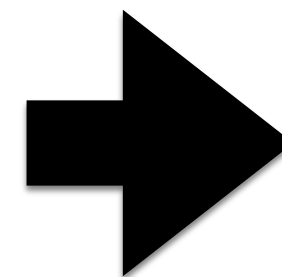
**(CompteCourant t) -> t.getSolde()**

**ou**

**t -> t.getSolde()**

# Approche fonctionnelle (JDK 8)

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



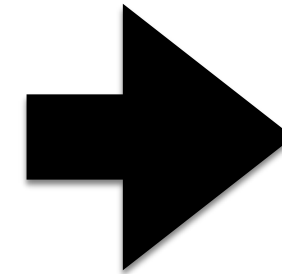
**(Double t) -> t > 0**

**ou**

**t -> t > 0**

# Approche fonctionnelle (JDK 8)

```
new Reducer<Double>() {  
    @Override  
    public Double reduce(Double t1, Double t2) {  
        return t1+t2;  
    }  
}
```



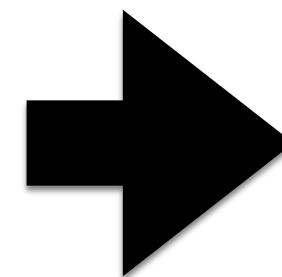
**(Double t1, Double t2) -> t1 + t2**

**ou**

**(t1,t2) -> t1 + t2**

# Approche fonctionnelle (JDK 8)

```
list.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;  
    }  
});
```



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

# Si la lambda nécessite plusieurs lignes de code ?

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

Dans ce cas on est obligé de conserver les  **accolades**  du bloc de la méthode.

# Rappels pour utiliser une lambda ?

Une seule méthode abstraite dans l'interface. On parle alors d'**interface fonctionnelle**.

Les types de paramètres de l'unique méthode doivent être compatibles avec les types de l'expression lambda.

# Une lambda dans une variable ?

```
Mapper<CompteCourant, Double> mapper = t -> t.getSolde();
```

```
Predicate<Double> filter = t -> t > 0;
```

```
Reducer<Double> reduce = (t1, t2) -> t1+t2;
```

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



# Travaux Pratiques