Programmation Par Objets

Lambda expressions
Traitement de flots (streaming)

Plan

I. Lambdas c'est quoi ?

- Expression d'un « pur » traitement
  - avec ses paramètres introduits par le symbole λ
  - sans nom, appelée aussi « fonction anonyme »

```
λ(x) x+1
λ(x,y) x<y
λ(message, x) {print message: x}
```

- A l'origine, concept mathématique de la théorie du **lambda calcul** de [A.Church, 1930] pour formaliser le calcul formel de fonctions
- Repris comme fondement du *paradigme de la **programmation fonctionnelle*** dans les langages Lisp (1958), Scheme, ML (CAML et OCAML), Haskell, ... où tout est fonction et composition de fonctions
- Intégré (récemment) dans la plupart des langages (notamment à objets, multi-paradigmes) : Java, C++, C#, Scala, Smalltalk, Python, ...
- Pourquoi ? ...

Lambdas pourquoi ? Motivations (1/3)

1. Paramétrer des traitements par des traitements

- Exemple : *cumul sur un tableau de nombres de calculs tels que :*
  `λ(x)x+1` , `λ(x)2*x` , ...
  même algo à la formule (lambda expression) près => la paramétrer

```
algo cumul(t, λ)
  pour tout x de t faire
  result = result + apply λ to x
```

- Lambda expression = « donnée fonctionnelle »
- « code as data »
- manipulable en paramètre (et plus généralement dans des variables)
- simplification de code

Lambdas pourquoi? Motivations (2/3)

2. Enchainement de traitements sur des flots de données

- « streaming » de données
- typique du Big Data (« map-reduce »)
- Exemple:

```
data_flot
| filtrer(λ_critere)
| extraire(λ_info)
| appliquer(λ_formule)
| cumuler
```


Lambdas pourquoi? Motivation (3/3)

3. Parallélisation

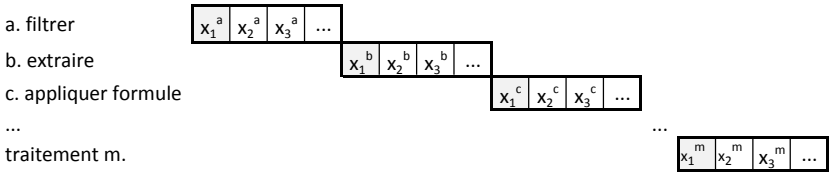
- suite à la généralisation des processeurs multi-cœurs
- faciliter la parallélisation de tels enchainements de traitements par « pipelining »
- principe : une donnée peut passer à l'étape suivante de traitement dès qu'elle est produite (« consommable »)

Exemple...

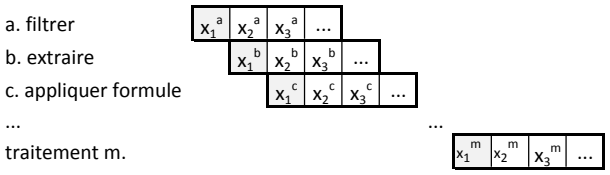
Lambdas pourquoi? Motivation (3/3)

Soit m traitements à effectuer sur n données

Enchainement séquentiel : O(nxm)



Enchainement pipeliné : O(n+m)



Lambdas pourquoi? Motivation (3/3)

Lambdas (programmation fonctionnelle) et parallélisme

- les lambdas (dans leur acceptation la plus stricte) représentent des « pures fonctions » (au sens mathématique)
- elles **produisent** des données indépendantes (sans modification ou « effet de bord »)
- et facilitent donc la parallélisation de traitements.

II. Lambdas en Java: syntaxe

```
<parametres> -> <expression> | {bloc de traitement}
```

```
x -> x+1 // λ(x) x+1

(x,y) -> x<y // λ(x,y) x<y

// bloc de traitement avec résultat (return)
(String message, Integer x, String) ->
        {int tmp;
         tmp = x + 10;
         return (message + " = " + tmp);}

// bloc de traitement sans résultat (action)
(String message, Integer x) ->
        {int tmp;
         tmp = x + 10;
         System.out.println(message + " = " + tmp);}

// lambda sans paramètre
() -> "done"
```


Représentation par objets

- Objets particuliers dits « objets fonctionnels »
  - représentation de fonctions
  - munis d'**une seule méthode : méthode d'application**
  - manipulable comme valeur (dite « fonctionnelle ») de **variables**, de **paramètres** ou en **résultat** de traitements
- Typage
  - comme tout objet, une lambda a un type défini par une interface dite **« interface fonctionnelle »** spécifiant sa **méthode d'application**
  - mais pas de classe, pas d'instanciation explicite : objets créés « à la volée » et gérés par le langage.
- Interfaces fonctionnelles
  - jeu varié de types de lambdas standards et génériques sur leurs paramètres d'application
  - fournis dans les bibliothèques notamment le package **java.util.function**
  - et il est possible d'en programmer : interfaces avec 1! méthode abstraite

Exploitation: Lambda valeur de variable

`x -> x+1, x -> 2*x, ...`

- sont du type (interface fonctionnelle) **`Function<T,R>`** avec
  - **`T`** : type du paramètre
  - **`R`** : type du résultat
  - méthode d'application : **`R apply(T)`**

- Exemple

```
Function<Integer,Integer> plus1, fois2;
plus1 = x -> x+1;
plus1.apply(10); // 11
fois2 = x -> 2*x;
fois2.apply(10); // 20
```


Lambda comme paramètre (exple. 1)

- Exemple de l'algo de `cumul`
- solution classique

```
int cumul=0;
for (int x : t) cumul += (x+1);

int cumul=0;
for (int x : t) cumul += (2*x);
```

autres formules : dupliquer le code

```
int cumul=0;
for (int x : t) cumul += (x*x);
```

Lambda comme paramètre (exple. 1)

- solution avec lambdas

```
int cumul(int[] t, Function<Integer, Integer> ld) {
  int cumul =0;
  for (int x : t) cumul += ld.apply(x);
  return cumul;
}

cumul = cumul(t, x -> x+1 ); // ou cumul(t, plus1)
cumul = cumul(t, x -> 2*x ); // ou cumul(t, fois2)
cumul = cumul(t, x -> x*x );
...
```


Lambda comme paramètre (exple. 2)

Tri d'ouvrages selon différents critères (titre, auteur, ...)

```
List<Ouvrage> ouvrages;
```

- solution classique par objet comparateur implémentant l'interface **Comparator<T>**

```
class ComparateurTitre implements Comparator<Ouvrage> {
 int compare(Ouvrage o1, Ouvrage o2) {
   return o1.getTitre().compareTo(o2.getTitre());
}}
class ComparateurAuteur implements Comparator<Ouvrage> {
 int compare(Ouvrage o1, Ouvrage o2) {
   return o1.getAuteur().compareTo(o2.getAuteur());
}}
Collections.sort(ouvrages, new ComparateurTitre());
Collections.sort(ouvrages, new ComparateurAuteur());
```

Lambda comme paramètre (exple. 2)

- Il s'avère que `Comparator<T>` a le statut d'« interface fonctionnelle » puisqu'elle ne spécifie qu'une seule méthode abstraite (relation d'ordre):
  `int compare(T x, T y);`
- elle peut donc typer des lambda expressions pour spécifier des critères de tri (paramètre `Comparator<T>` de `sort`)
- d'où la simplification de code

```
Collections.sort(ouvrages, // lambda Comparator % titre:
    (Ouvrage o1, Ouvrage o2)
      -> o1.getTitre().compareTo(o2.getTitre()) );
Collections.sort(ouvrages, // lambda Comparator % auteur:
    (Ouvrage o1, Ouvrage o2)
      -> o1.getAuteur().compareTo(o2.getAuteur()) );
```


Lambda comme résultat

- Composition de fonctions

```
f : X -> Y
g : Y -> Z
(g o f) : X -> Z
    x -> g(f(x))
```

- par lambdas :

```
<X,Y,Z> Function<X,Z>
   compose(Function<X,Y> f, Function<Y,Z> g) {
      return (x) -> g.apply(f.apply(x));
   }
```

- Exemple

```
compose(plus1,fois2).apply(10) = 22
compose(fois2,plus1).apply(10) = 21
```

Interfaces fonctionnelles standards

`java.util.function`

Type de lambda (interface fonctionnelle)	méthode d'application	exemple
`Function<T,R>`	`R apply(T)`	`x -> x+1`
`BiFunction<T,U,R>`	`R apply(T,U)`	`(x,y) -> x+y`
`Predicate<T>`	`boolean test(T)`	`st -> st.equals("OK")`
`BiPredicate<T,U>`	`boolean test(T,U)`	`(x,y) -> x<y`
`Consumer<T>`	`void accept(T)`	`x -> System.out.print("x:"+x)`
`BiConsumer<T,U>`	`void accept(T,U)`	`(text,x) -> System.out.print(text+x)`
`Supplier<T>`	T get()	`() -> "done"`
`...`		

III. Traitement de flots : « streams »

- « **Streaming** »: enchainement de traitements paramétrés par lambda expression sur des flots de données = « streams »
- ces traitements appliquent la lambda à chaque donnée et on distingue
  - les traitements du genre « map » dits **intermédiaires** qui produisent un **nouveau stream** sur lequel il est possible d'enchainer ...
  - les traitements du genre « reduce » dits **terminaux** qui ne produisent pas de stream (`void`, résultat de synthèse, ...)
- Exemple

```
data_flot
    | filtrer(λ_critere) // intermédiaire
    | extraire(λ_info) // intermédiaire
    | appliquer(λ_formule) // intermédiaire
    | cumuler // terminal
```

Opérations de traitement de flots

- Spécifiées par un jeu de types de streams : package `java.util.stream`
- dont le plus général (générique) : **`Stream<T>`**
- fourni par diverses **sources de données** :
  - les `Collection's (java.util.Collection)`
    - via leur méthode
      `Stream<T> stream()`
  - les tableaux
    - via l'utilitaire `static` de la classe `Arrays` (utilitaires de tableaux)
      `Stream<T> Arrays.stream(T[])`
  - les fichiers textes (`java.io.BufferedReader`)
    - via leur méthode
      `Stream<String> lines() // ligne par ligne`

L'interface **`Stream<T>`**

Traitement	genre
`Stream<T> filter(Predicate<T>)`	intermédiaire
`Stream<R> map(Function<T,R>)`	intermédiaire
`Stream<T> sorted(Comparator<T>)`	intermédiaire
`void forEach(Consumer<T>)`	terminal
int count()	terminal
`...`	

Exemple sur les collections (1/3)

- Soit la classe `Message`

```
public class Message {
     // fields
     protected String from, to;
     protected Date date;
     protected String body;
     protected int size;
...}
```

- et une collection de `Message`

```
Collection<Message> messages;
```

- Problème
  Filtrer les messages contenant le mot `"Polytech"` *dans leur* `body`*, les trier par expéditeur* (`from`) *et les afficher*

Exemple sur les collections (2/3)

Solution OO « classique »

```
List<Message> filtered = new ArrayList<>();
// 1. Filtrer
for (Message m : messages)
        if (m.getBody().contains("Polytech"))
                filtered.add(m);
// 2. Trier par expéditeur (from)
class ComparatorFrom implements Comparator<Message> {
        public int compare(Message m1, Message m2) {
                return m1.getFrom().compareTo(m2.getFrom());
}}
Collections.sort(filtered, new ComparatorFrom());
// 3. Afficher
for (Message m : filtered)
        System.out.println(m);
```

Exemple sur les collections (3/3)

Avec lambdas et streams

```
messages.stream()
.filter(m -> m.getBody().contains("Polytech")) //1.
.sorted((m1,m2)->m1.getFrom().compareTo(m2.getFrom())) //2.
.forEach(m -> System.out.println(m)); //3.
```

Remarquer le passage

- du contrôle « externe » (structures de contrôle « classiques »)
- à un contrôle « interne » pris en charge par le `Stream` lui-même (à sa façon)
- toujours + orienté objet !

Streams numériques

- `IntStream, DoubleStream` ...
  dédiés aux types primitifs
- opérations :
  `sum(), average(), min(), max()...`
- streams fournis par diverses sources dont :
  - les `Stream<T>` génériques par les méthodes :
    - `IntStream` **`mapToInt`**`(Function<T,Integer>)`
    - `DoubleStream` **`mapToDouble`**`(Function<T,Double>)`
  - les tableaux, par les méthodes `static` de `Arrays`
    - `IntStream Arrays.stream(int[])`
    - `DoubleStream Arrays.stream(double[])`

Exemple

Taille totale des messages contenant le mot `"Polytech"` *dans leur* `body`

```
messages.stream()
.filter(m->m.getBody().contains("Polytech"))
.mapToInt(m->m.getSize()) // IntStream
.sum()
```


Collecteurs

- Les collections et les tableaux sont des **sources** de streams
- inversement les **collecteurs** permettent de récupérer les résultats d'un streaming dans un tableau ou une collection (traitements terminaux)
- via les méthodes de streams
  - `toArray()`
  - `collect(Collector)`

    le paramètre `Collector` permet de définir le type de `Collection` (`List`, `Set`, ...) où ranger les résultats

Collecteurs: `toArray()`

Un tableau des tailles des messages contenant le mot `"Polytech"` *dans leur* `body`

```
int[] tSizes;
tSizes =
 messages.stream()
 .filter(m->m.getBody().contains("Polytech »))
 .mapToInt(m->m.getSize())
 .toArray();
```


Collecteurs : `collect(Collector)`

- Des `Collector` types sont fournis (en `static`) par la classe d'utilitaires `Collectors` :
  - `List<T> toList()`
  - `Set<T> toSet()`
  - `...`
- Exemple

```
List<String> listExpediteurs =
 messages.stream()
 .filter(m ->m.getBody().contains("Polytech"))
 .map(m -> m.getFrom())
 .collect(Collectors.toList());
```

IV. Lambdas références de méthodes

- Quand une lambda ne fait qu'invoquer une seule méthode (ou un constructeur)
- formulation simplifiée
  « method (constructor) invocation as lambda »

- Formes possibles
  - référence de méthode

    `<class>|<some object> :: <method name>`
  - référence de constructeur

    `<class>|<Type[]> :: new`

Référence de méthode

- Référence de méthode

```
messages.stream()
.mapToInt((Message m) -> m.getSize()) // lambda
<=>
messages.stream()
.mapToInt(Message::getSize) // method ref
```

- Référence de méthode sur un objet particulier

```
messages.stream()
.forEach(m -> System.out.println(m));
<=>
messages.stream()
.forEach(System.out::println) // method ref
```

Référence de constructeur

- Soit la classe représentant l'enveloppe d'un message: triplet (`from,to,date`) :

```
class Envelope {
   String from, to;
   Date date;
   Envelope(Message m) { // Constructeur d'enveloppe d'un message
     this.from = m.getFrom();
     this.to = m.getTo();
     this.date = m.getDate();
}}
```

- Production d'un `Stream<Envelope>` à partir d'un `Stream<Message>`

```
messages.stream()
.map((Message m) -> new Envelope(m)) // par lambda
<=>
messages.stream()
.map(Envelope::new) // reference de constructeur equivalente
```

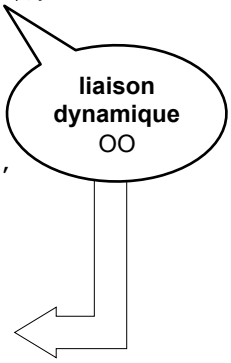

V. «Deep inside» Environnement d'une lambda

- Une lambda a accès à ses paramètres et à ses variables locales, dites *variables liées*
- mais elle s'inscrit aussi dans un environnement de variables dites *libres,* potentiellement *:*
  - **environnement d'introduction**: lieu où elle est introduite (définie)
  - **environnement d'exécution :** lieu où elle est exécutée
- Problème
  - à quel environnement de variables a-t-elle effectivement accès?
  - problème dit de *capture* ou de *clôture* (« closure ») d'environnement dans les langages
- Alternative :
  - **capture statique** (« lexicale ») : les variables libres accessibles par la lambda sont celles de son environnement d'introduction
  - **capture dynamique** : les variables libres accessibles par la lambda sont celles de son environnement d'exécution.
- En Java la capture effectuée par les lambdas est statique.

Capture : illustration (1/3)

Exemple OO classique : par méthodes

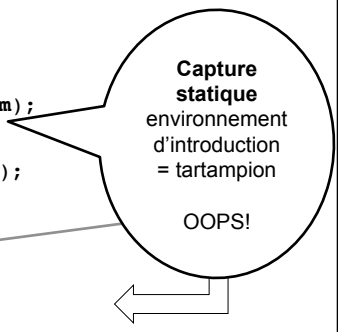
```
class Personne {
  String nom, tel;
  Personne collegue;
  void say(String text) {System.out.print(text + nom);}
  void quiEsTu() {
    this.say("je suis ");
    collegue.say(", collegue: ");
  }}
// Application (main)
Personne martine = new Personne("Martine", "3000 »),
  tartampion = new Personne("Tartampion", "4000"),
  albert = new Personne("Albert", "1000");

tartampion.setCollegue(martine);
tartampion.quiEsTu();
// je suis Tartampion, collegue: Martine
```



Capture : illustration (2/3)

Au travers d'une lambda

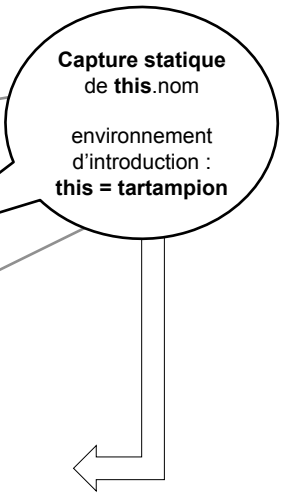
```
class Personne { ...
  void say(Consumer<String> lambda_text, String text) {
    lambda_text.accept(text);
  }
  void quiEsTu() {
    Consumer<String> lambda_text // lambda
      = (text) -> System.out.print(text + nom);
    this.say(lambda_text, "je suis ");
    collegue.say(lambda_text, ", collegue: ");
  }}
// Application (main)
tartampion.quiEsTu();
// je suis Tartampion, collegue: Tartampion
```



Capture : illustration (3/3)

Par référence de méthode: idem

```
class Personne { ...
  void say(String text) {
    System.out.print(text + nom);
  }
  void quiEsTu() {
    Consumer<String> lambda_text = this::say;
    this.say(lambda_text, "je suis ");
    collegue.say(lambda_text, ", collegue: ");
  }
// Application (main)
tartampion.quiEsTu();
// je suis Tartampion, collegue: Tartampion
```



Environnement d'une lambda

En conclusion : une lambda a accès

- à ses paramètres et à ses variables locales
- et **par capture** aux variables de son **environnement d'introduction**
  - si introduite par un objet: `this` et ses variables d'instances
  - paramètres et variables temporaires d'une méthode d'introduction
    - moyennant la restriction: uniquement les variables `final` (immutables)
    - en effet la durée de vie d'une lambda peut dépasser celle de la méthode (ne serait-ce que par `return` puis mémorisation)

«Deep inside» Lambdas, des objets sans état

- Rappel
  - les lambdas sont représentées par des objets particuliers dits « objets fonctionnels »
  - munis d'une seule méthode : méthode d'application spécifiée par leur type = « interface fonctionnelle »
  - référençables dans des variables ou comme paramètre
  - instanciées et gérées par le langage
- Particularité importante: **lambdas = objets sans état**
  en effet ces objets représentent des expressions
- Conséquences
  - pas de variables d'instances (pas d'état)
  - « pas de this »
    si ce n'est celui capturé (lambda introduite par un objet)

Lambdas vs. AIC : illustration (1/4)

Pour comparaison et en guise d'explication :
Lambdas vs. « Anonymous Inner Classes » (AIC)

On reprend le problème de tri par `Comparator<T>`

```
class Societe {
  List<Personne> personnel;
  void trier() {
    Collections.sort(personnel, new ComparateurNom());
  }

class ComparateurNom implements Comparator<Personne>{
  public int compare(Personne x, Personne y) {
    return x.getNom().compareTo(y.getNom());
  }
}
```

Lambdas vs. AIC : illustration (2/4)

« Inner class » (IC) de comparateur

```
class Societe { ...
  // IC Societe$ComparateurNom
  class ComparateurNom implements Comparator<Personne>{
    public int compare(Personne x, Personne y) {
      return x.getNom().compareTo(y.getNom());
    }
  }
  void trier() {
    Collections.sort(personnel, new ComparateurNom());
  }
}
```


Lambdas vs. AIC : illustration (3/4)

« Anonymous Inner Class » (AIC) de comparateur

```
class Societe { ...
  void trier() {
    Collections.sort(personnel,
      new Comparator<Personne>() { // AIC
        public int compare(Personne x, Personne y) {
          return x.getNom().compareTo(y.getNom());
        }
      }
    );
}
```

Lambdas vs. AIC : illustration (4/4)

`Comparator<T>` ayant le statut d'interface fonctionnelle (1! méthode abstraite: `int compare(T,T)`)

=> Simplification par lambda du même type

```
class Societe { ...
 void trier() {
  Collections.sort(personnel,
   (x, y) -> x.getNom().compareTo(y.getNom()) // lambda
  );
 }
}
```


Lambdas vs. AIC

- Mais cette simplification ne vaut que si l'AIC ne gère **pas d'état** (en pratique : pas de variables d'instance)
- Contre-exemple: comparateur avec compteur de comparaisons

```
class Societe { ...
 void trier() { Collections.sort(personnel,
  new Comparator<Personne>() {
   int compteur = 0; // AIC avec etat
   public int compare(Personne x, Personne y) {
    this.compteur++; // this= le comparateur instance de l'AIC
    System.out.println("compteur:"+this.compteur); //trace
    return x.getNom().compareTo(y.getNom());
   }
  });}
}
```

Lambdas vs. AIC : exemple

```
// application (main)
societe.trier();
compteur:1
compteur:2
compteur:3
compteur:4
Person [nom=Albert, tel=1000]
Person [nom=Martine, tel=3000]
Person [nom=Tartampion, tel=4000]

// re-trier
societe.trier();
compteur:1
compteur:2
```


Lambdas vs. AIC : exemple

Non simplifiable par une lambda (objet sans état).
Tout au mieux :

```
class Societe { ...
  int compteur; // commun à la societe
  void trier() {
    this.compteur = 0; // re-init necessaire (sinon cumul)
    Collections.sort(personnel, (x, y) -> { //lambda
        this.compteur++; //capture: this = la societe
        System.out.println("compteur:" + this.compteur);
        return x.getNom().compareTo(y.getNom());
      }
    );
  }
}
```

Lambdas vs. AIC

- En résumé

  Lambda expression

  <=>

  instance d'AIC **sans état**
  implémentant la même interface fonctionnelle

- En pratique
  - Toute instance d'AIC sans état implémentant une interface fonctionnelle peut se simplifier par une lambda
  - Mais il faut revenir aux AIC (classes de plein droit) si la gestion d'état est nécessaire (« full objects »).

Biblio

- http://www.angelikalanger.com/Lambdas/LambdaReference.pre-release.pdf

« Lambda Expressions and Streams in Java − Reference », A. Langer & K. Kreft

- « Aide-mémoire Java » V. Granet & J-P. Regourd, Editions Dunod.
- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

« The Java Tutorials - Lambda Expressions », site officiel Java (Oracle)

- Packages `java.util.function` et `java.util.stream` de la Javadoc