Qualité de développement

Lambdas et streams en java

by Cédric Lhoussaine (Université de Lille / IUT de Lille)
on 2024-25

» Outline

» Classes internes

Définition

Une classe interne est une classe définie à l'intérieur du code d'une autre classe (au même niveau que ses méthodes et ses attributs) :

- par rapport à la classe principale, elle s'utilise comme n'importe quelle autre classe ;
- les méthodes de la classe interne ont *directement accès* aux méthodes et attributs de la classe principale (même s'ils sont privés !) ;
- si la classe interne est publique, elle peut être utilisée depuis une autre classe.

» Exemple de classe interne

```
public Interface Agregat<E> {
    public Iterateur<E> creerIterateur();
}

public Interface Iterateur<E> {
        public boolean aUnSuivant();
        public E elementSuivant();
}
```

» Exemple de classe interne

```
public class AgregatConcret implements Agregat<String> {
    private String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

    public Iterateur<String> creerIterateur() {
        return new MonIterateur();
    }

    private class MonIterateur implements Iterateur<String> {
        int index;

        public boolean aUnSuivant() {
            return (index < names.length)?true:false;
        }

        public String elementSuivant() {
            if (this.aUnSuivant()) return names[index++];
            return null;
        }
    }
}
```

» Classes anonymes

Définition

Une classe anonyme est une classe dérivée d'une autre ou qui implémente une interface :

- qui ne reçoit pas de nom ;
- qui est créée en même temps que son instance, en donnant tout son code après l'appel `new SuperClasse() {...}` ou `new Interface() {...}`
- et dont la persistance est limitée à la durée de vie de son instance ;
- à utiliser avec modération

» Exemple de classe anonyme

```
public class AgregatConcret implements Agregat<String> {
    private String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

    public Iterateur<String> creerIterateur() {
        return new Iterateur<String>() {
            int index;

            public boolean aUnSuivant() {
                return (index < names.length)?true:false;
            }

            public String elementSuivant() {
                if (this.aUnSuivant()) return names[index++];
                return null;
            }
        };
    }
}
```

» Avant les lambdas

Il arrive souvent, en particulier dans les bibliothèques graphiques, de fournir à des méthodes des fonctions qui réalisent des actions particulières. Avant java 8, il fallait passer par des classes anonymes implantant une seule méthode :

```
monBouton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            System.out.println("clic");
        }
    });
```

» Avant les lambdas

Il arrive souvent, en particulier dans les bibliothèques graphiques, de fournir à des méthodes des fonctions qui réalisent des actions particulières. Avant java 8, il fallait passer par des classes anonymes implantant une seule méthode :

```
monBouton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            System.out.println("clic");
        }
    });
```

À partir de java 8, on peut passer directement la méthode, sous forme de functions anonymes (*lambda expressions*), au lieu d'une classe « complète » :

```
monBouton.addActionListener(event -> System.out.println("clic"));
```

» Objectifs des lambda expressions

- Réduire le « boilerplate code »
- Introduire dans java, le paradigme fonctionnel

» Les lambda expressions, une vieille histoire...

- λ-calcul : Alonzo Church $\sim$ 1930
- Modèle de calcul, alternatif aux machines de Turing et à la théorie des fonctions récursives mais tout aussi expressif.
- Langage très « simple », qui repose sur les notions de variable (x), fonctions ($\lambda x.t$) et application (fg).
- Variantes typées
- Isomorphisme de Curry-Howard : preuve = programme

» Principe général

- Les lambda expressions sont des fonctions (ou méthodes) anonymes, i.e. des expressions qui associent à une suite (éventuellemment vide) de paramètres, un bloc d'instructions.

» Principe général

- Les lambda expressions sont des fonctions (ou méthodes) anonymes, i.e. des expressions qui associent à une suite (éventuellemment vide) de paramètres, un bloc d'instructions.
- En Java, une lambda expression est un objet ! Cet objet comporte une unique méthode qui permet d'évaluer la lambda expression.

» Principe général

- Les lambda expressions sont des fonctions (ou méthodes) anonymes, i.e. des expressions qui associent à une suite (éventuellemment vide) de paramètres, un bloc d'instructions.
- En Java, une lambda expression est un objet ! Cet objet comporte une unique méthode qui permet d'évaluer la lambda expression.
- Une lambda expression doit être typée avec une interface fonctionnelle, i.e. une interface qui ne déclare qu'une seule méthode abstraite (dont la signature doit évidemment être compatible avec le type de la lambda expression).

» Principe général

Losqu'une méthode reçoit un object de type interface fonctionnelle, par exemple :

```
interface Increment {
    int inc(int i);
}
...
void foo(Increment i) {
    e = i.inc(e);
}
```

la méthode `foo` peut être appelée avec un argument de type `Increment` sous la forme de :

- objet ou classe anonyme
- lambda expression
- référence de méthode (i.e. méthode de classe ou objet existant).

» Exemple (version classe anonyme)

```java
public class Entier {
    private int e;


    void foo(Increment i) {
        e = i.inc(e);
    }

    public String toString(){
        return "" + e;
    }
}
```

```java
Entier e = new Entier();
System.out.println("e = " + e);

e.foo(new Increment(){
        public int inc(int x) {
            return x+1;
        }
    });
System.out.println("e = " + e);

e.foo(new Increment(){
        public int inc(int x) {
            return x+2;
        }
    });
System.out.println("e = " + e);
```

» Exemple (version lambda expression)

```java
public class Entier {
    private int e;

    // déclaration inchangée!!
    void foo(Increment i) {
        e = i.inc(e);
    }

    public String toString(){
        return "" + e;
    }
}
```

```java
Entier e = new Entier();
System.out.println("e = " + e);

// seul le paramètre effectif change
e.foo(x -> x+1);
System.out.println("e = " + e);

e.foo(x -> x+2);
System.out.println("e = " + e);
```

» Exemple (version références de méthodes)

```java
public class MesIncrements {
    public static int plus1(int x) {
        return x+1;
    }

    public static int plus2(int x) {
        return x+2;
    }
}
```

```java
Entier e = new Entier();
System.out.println( "e = " + e);

e.foo(MesIncrements::plus1);
System.out.println( "e = " + e);

e.foo(MesIncrements::plus2);
System.out.println( "e = " + e);
```

» Syntaxe des lambda expression

Expressions de la forme

```
(parametres) -> expression // ou bien
(parametres) -> { traitements; }
```

- paramètres sont séparés par des virgules
- parenthèses optionnelles s'il n'y a qu'un seul paramètre
- ( ) si aucun paramètre
- préciser le type de paramètre s'il ne peut pas être inféré :

```
afficher = (Integer x) -> System.out.println(x)
```

- attention : si plusieurs paramètres, il faut soit spécifier tous les types soit aucun.

» Portée des variables

Dans le corps d'une lambda expression, on ne peut pas faire référence à des variables non (*effectivement*) `final` du contexte englobant.

```
public void test(){
    int v = 1;
    Entier e = new Entier();
    e.foo(x -> {return x+v;});
}
```

la variable v est « effectivement » final car jamais modifiée.

» Portée des variables

Dans le corps d'une lambda expression, on ne peut pas faire référence à des variables non (*effectivement*) `final` du contexte englobant.

```
public void test(){
    int v = 1;
    Entier e = new Entier();
    e.foo(x -> {return x+v;});
}
```

la variable v est « effectivement » final car jamais modifiée. En revanche :

```
public void test(){
    int v = 1;
    Entier e = new Entier();
    v++;
    e.foo(x -> {return x+v;}); // error
}
```

provoque une erreur à la compilation !

» Les interfaces fonctionnelles

* une interface fonctionnelle est une interface dans laquelle une seule méthode *abstraite* est déclarée.

» Les interfaces fonctionnelles

- une interface fonctionnelle est une interface dans laquelle une seule méthode *abstraite* est déclarée.
- utiliser d'annotation `@FunctionalInterface` pour indiquer au compilateur que l'interface définie est fonctionnelle (et qu'il doit en vérifier les contraintes).

» Les interfaces fonctionnelles

- une interface fonctionnelle est une interface dans laquelle une seule méthode *abstraite* est déclarée.
- utiliser d'annotation `@FunctionalInterface` pour indiquer au compilateur que l'interface définie est fonctionnelle (et qu'il doit en vérifier les contraintes).
- est utilisée pour typer
  - une référence de méthode
  - une lambda expression
  - une classe anonyme

» Le package `java.util.function`

- contient les interfaces fonctionnelles d'usage courant
- `Function<T,R>` : fonction unaire qui réalise une « transformation »

```
R apply(T t)
```

» Le package `java.util.function`

* contient les interfaces fonctionnelles d'usage courant
* `Function<T,R>` : fonction unaire qui réalise une « transformation »

```
R apply(T t)
```

* `Consumer<T>` : fonction qui réalise une action (i.e. ne renvoie pas de valeur)

```
void accept(T)
```

» Le package `java.util.function`

- contient les interfaces fonctionnelles d'usage courant
- `Function<T,R>` : fonction unaire qui réalise une « transformation »

```
R apply(T t)
```

- `Consumer<T>` : fonction qui réalise une action (i.e. ne renvoie pas de valeur)

```
void accept(T)
```

- `Predicate<T>` : fonction unaire qui renvoie un booléen

```
boolean test(T t)
```

» Le package `java.util.function`

- contient les interfaces fonctionnelles d'usage courant
- `Function<T,R>` : fonction unaire qui réalise une « transformation »

```
R apply(T t)
```

- `Consumer<T>` : fonction qui réalise une action (i.e. ne renvoie pas de valeur)

```
void accept(T)
```

- `Predicate<T>` : fonction unaire qui renvoie un booléen

```
boolean test(T t)
```

- `Supplier<T>` : fonction sans paramètre qui renvoie une instance

```
T get()
```

» Les références de méthodes

Il est possible d'utiliser des méthodes déjà déclarées en lieu et place de lambda expressions :

- référence à une méthode statique :

```
classe::methodeStatique // equiv. x -> nomClasse.methodeStatic(x)
```

» Les références de méthodes

Il est possible d'utiliser des méthodes déjà déclarées en lieu et place de lambda expressions :

- référence à une méthode statique :

```
classe::methodeStatique // equiv. x -> nomClasse.methodeStatic(x)
```

- référence à une méthode d'instance :

```
objet::nomMethode // equiv. x -> objet.nomMethode(x)
```

» Les références de méthodes

Il est possible d'utiliser des méthodes déjà déclarées en lieu et place de lambda expressions :

- référence à une méthode statique :

```
classe::methodeStatique // equiv. x -> nomClasse.methodeStatic(x)
```

- référence à une méthode d'instance :

```
objet::nomMethode // equiv. x -> objet.nomMethode(x)
```

- référence à une méthode d'un objet arbitraire d'un type donné :

```
String::compareToIgnoreCase // equiv. (x, y) -> x.compareToIgnoreCase(y)
```

» Les références de méthodes

Il est possible d'utiliser des méthodes déjà déclarées en lieu et place de lambda expressions :

- référence à une méthode statique :

```
classe::methodeStatique // equiv. x -> nomClasse.methodeStatic(x)
```

- référence à une méthode d'instance :

```
objet::nomMethode // equiv. x -> objet.nomMethode(x)
```

- référence à une méthode d'un objet arbitraire d'un type donné :

```
String::compareToIgnoreCase // equiv. (x, y) -> x.compareToIgnoreCase(y)
```

- référence à un constructeur :

```
MaClasse::new // equiv. () -> new MaClasse();
```

» Stream (vs List)

* les streams sont des variantes de listes (éventuellement infinies)

» Stream (vs List)

- les streams sont des variantes de listes (éventuellement infinies)
- elles permettent de se concentrer sur le « quoi » (calculer) plutôt que le « comment » (calculer)

» `Stream` (vs `List`)

- les streams sont des variantes de listes (éventuellement infinies)
- elles permettent de se concentrer sur le « quoi » (calculer) plutôt que le « comment » (calculer)
- elles utilisent « massivement » les *lambda expressions*

» `Stream` (vs `List`)

- les streams sont des variantes de listes (éventuellement infinies)
- elles permettent de se concentrer sur le « quoi » (calculer) plutôt que le « comment » (calculer)
- elles utilisent « massivement » les lambda expressions
- elles ne stockent pas leurs données, celles-ci viennent « d'ailleurs » (fichier, BDD, internet, etc.)

» Stream (vs List)

- les streams sont des variantes de listes (éventuellement infinies)
- elles permettent de se concentrer sur le « quoi » (calculer) plutôt que le « comment » (calculer)
- elles utilisent « massivement » les lambda expressions
- elles ne stockent pas leurs données, celles-ci viennent « d'ailleurs » (fichier, BDD, internet, etc.)
- elles sont non mutables : la plupart des méthodes de streams créent de nouvelles streams

» `Stream` (vs `List`)

- les streams sont des variantes de listes (éventuellement infinies)
- elles permettent de se concentrer sur le « quoi » (calculer) plutôt que le « comment » (calculer)
- elles utilisent « massivement » les lambda expressions
- elles ne stockent pas leurs données, celles-ci viennent « d'ailleurs » (fichier, BDD, internet, etc.)
- elles sont non mutables : la plupart des méthodes de streams créent de nouvelles streams
- les opérations sur les streams sont réalisées de façon paresseuse (i.e. le plus tardivement possible) : lazy evaluation

» Exemple (Stream vs. List)

Compter le nombre de valeurs supérieures à 2. Avec des listes :

```
List<Integer> l = Arrays.asList(1,2,3,4);
long cpt = 0;
for (Integer x : l) {
    if (x > 2) cpt++;
}
```

» Exemple (Stream vs. List)

Compter le nombre de valeurs supérieures à 2. Avec des listes :

```
List<Integer> l = Arrays.asList(1,2,3,4);
long cpt = 0;
for (Integer x : l) {
    if (x > 2) cpt++;
}
```

avec les streams :

```
Stream<Integer> s = Stream.of(1,2,3,4);
long cpt = s.filter(x -> x>2).count();
```

» Construction de streams

- à l'aide de la méthode static `Stream.of` :

```
Stream<Integer> s = Stream.of(3, 1, 4, 1, 5, 9);
// ou bien
Integer[] tabEntiers = { 3, 1, 4, 1, 5, 9 };
Stream<Integer> s = Stream.of(tabEntiers);
```

» Construction de streams

- à l'aide de la méthode static `Stream.of` :

```
Stream<Integer> s = Stream.of(3, 1, 4, 1, 5, 9);
// ou bien
Integer[] tabEntiers = { 3, 1, 4, 1, 5, 9 };
Stream<Integer> s = Stream.of(tabEntiers);
```

- par conversion de `Collection` avec la methode `stream()`

```
Stream<Integer> s = l.stream();
```

» Construction de streams

* à l'aide de la méthode static `Stream.of` :

```
Stream<Integer> s = Stream.of(3, 1, 4, 1, 5, 9);
// ou bien
Integer[] tabEntiers = { 3, 1, 4, 1, 5, 9 };
Stream<Integer> s = Stream.of(tabEntiers);
```

* par conversion de `Collection` avec la methode `stream()`

```
Stream<Integer> s = l.stream();
```

* Stream infinie à partir d'une fonction d'itération avec la méthode statique `iterate`

```
Stream<Integer> s = Stream.iterate(0, x->x+1);
```

» Transformation de streams

* les streams sont transformées par des opérations dites intermédiaires et produisent de nouvelles streams pour être transformées (en pipeline) ou extraites.

» Transformation de streams

* les streams sont transformées par des opérations dites intermédiaires et produisent de nouvelles streams pour être transformées (en pipeline) ou extraites.
* appliquer une fonction à chaque élément d'une stream et renvoyer la stream des éléments ainsi obtenus

```
Stream.of(1,2,3,4).map(x -> 2*x)
```

» Transformation de streams

* les streams sont transformées par des opérations dites intermédiaires et produisent de nouvelles streams pour être transformées (en pipeline) ou extraites.
* appliquer une fonction à chaque élément d'une stream et renvoyer la stream des éléments ainsi obtenus

```
Stream.of(1,2,3,4).map(x -> 2*x)
```

* filtrer les éléments d'une liste satisfaisant une condition donnée :

```
Stream.of(1,2,3,4).filter(x -> x%2==0)
```

» Transformation de streams

* les streams sont transformées par des opérations dites intermédiaires et produisent de nouvelles streams pour être transformées (en pipeline) ou extraites.
* appliquer une fonction à chaque élément d'une stream et renvoyer la stream des éléments ainsi obtenus

```
Stream.of(1,2,3,4).map(x -> 2*x)
```

* filtrer les éléments d'une liste satisfaisant une condition donnée :

```
Stream.of(1,2,3,4).filter(x -> x%2==0)
```

* tronquer une stream :

```
Stream.of(1,2,3,4).limit(2)
Stream.of(1,2,3,4).skip(2)
```

» Transformation de streams

* Trier une stream

```
Stream.of(2,3,1,4).sorted()
```

» Transformation de streams

* Trier une stream

```
Stream.of(2,3,1,4).sorted()
```

* Eliminer les doublons d'une stream

```
Stream.of(2,2,1,2,3,4).distinct()
```

» Extraire les éléments

* extraction sous forme d'un tableau :

```
String[] result = stream.toArray(String[]::new);
```

» Extraire les éléments

* extraction sous forme d'un tableau :

```
String[] result = stream.toArray(String[]::new);
```

* extraction sous forme de collection :

```
List<String> result = stream.collect(Collectors.toList());
Set<String> result = stream.collect(Collectors.toSet());
```

» Extraire les éléments

* extraction sous forme d'un tableau :

```
String[] result = stream.toArray(String[]::new);
```

* extraction sous forme de collection :

```
List<String> result = stream.collect(Collectors.toList());
Set<String> result = stream.collect(Collectors.toSet());
```

* attention : les opérations d'extraction détruisent la stream originale, ce sont des opérations dites terminales

```
Stream<Integer> s = Stream.of(1,2,3,4,5);
List<Integer> result1 = s.limit(2).collect(Collectors.toList());
List<Integer> result2 = s.skip(2).collect(Collectors.toList());
// java.lang.IllegalStateException thrown: stream has already been operated upon c
```

» Extraire les éléments

* extraction sous forme d'un tableau :

```
String[] result = stream.toArray(String[]::new);
```

* extraction sous forme de collection :

```
List<String> result = stream.collect(Collectors.toList());
Set<String> result = stream.collect(Collectors.toSet());
```

* attention : les opérations d'extraction détruisent la stream originale, ce sont des opérations dites terminales

```
Stream<Integer> s = Stream.of(1,2,3,4,5);
List<Integer> result1 = s.limit(2).collect(Collectors.toList());
List<Integer> result2 = s.skip(2).collect(Collectors.toList());
// java.lang.IllegalStateException thrown: stream has already been operated upon c
```

» Type `Optional`

* il est courant d'utiliser la valeur `null` pour indiquer qu'une méthode n'a pas de résultat. Doit être traité par la fonction appelante sinon un `NullPointerException` est déclenché.

» Type `Optional`

- il est courant d'utiliser la valeur `null` pour indiquer qu'une méthode n'a pas de résultat. Doit être traité par la fonction appelante sinon un `NullPointerException` est déclenché.
- le type générique `Optional<T>` est un wrapper qui encapsule une valeur de type T ou une indication d'absence de valeur.

» Type Optional

- il est courant d'utiliser la valeur `null` pour indiquer qu'une méthode n'a pas de résultat. Doit être traité par la fonction appelante sinon un `NullPointerException` est déclenché.
- le type générique `Optional<T>` est un wrapper qui encapsule une valeur de type T ou une indication d'absence de valeur.
- les valeurs o de ce type possèdent une méthode `orElse(e)` qui
  - renvoie son argument e si o n'encapsule pas de valeur, ou bien
  - la valeur encapsulée

```
Stream.of(1,2).skip(3).max((x,y)->x-y).orElse(-1)
```

» Type `Optional`

* la méthode `isPresent()` teste la présence d'une valeur

» Type Optional

- la méthode `isPresent()` teste la présence d'une valeur
- la méthode `ifPresent(...)` applique la fonction passée en paramètre s'il y a une valeur, sinon ne fait rien.

```
Stream.of(1,2,3,4)
      .max((x,y)->x-y)
      .ifPresent(x -> System.out.println("le max est " + x))
```

» Type `Optional`

* la méthode `isPresent()` teste la présence d'une valeur
* la méthode `ifPresent(...)` applique la fonction passée en paramètre s'il y a une valeur, sinon ne fait rien.

```
Stream.of(1,2,3,4)
      .max((x,y)->x-y)
      .ifPresent(x -> System.out.println("le max est " + x))
```

* pour construire une valeur de type `Optional` :

```
Optional<Integer> o = Optional.of(1);
Optional<Integer> o = Optional.empty();
```

» Streams de type primitif

- 3 types de stream dont les éléments sont de type primitif :
  - `IntStream` (pour `int`),
  - `LongStream` (pour `long`),
  - `DoubleStream` (pour `double`),
- exemples

```
IntStream s1 = IntStream.of(3, 1, 4, 1, 5, 9);
// ou bien
int[] values = {3, 1, 4, 1, 5, 9};
s1 = IntStream.of(values);
// ou bien
IntStream s2 = IntStream.range(1, 10);
// ou bien
Random generator = new Random();
IntStream s3 = generator.ints(1, 7); // Stream infinie!
// ou bien
Stream<String> mots = Stream.of("Jean-Michel", "est", "cool")
IntStream s4 = mots.mapToInt(w -> w.length());
```

Emacs 29.4 (Org mode 9.7.11)