JAVA Lambda

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

JAVA

Lambda

Sommaire

Introduction

- Pour faciliter, entre autres, cette mise à oeuvre, Java 8 propose les expressions lambda. Les expressions lambda sont aussi nommées closures ou fonctions anonymes : leur but principal est de permettre de passer en paramètre un ensemble de traitements.

introduction

Affichage d'une liste sans lambda:

- ```
  for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i))
  }
  ```

Avec Lambda :

- ```
  list.forEach(System.out::println)
  ```

Historique

Lors de la conception des expressions lambda, le choix a été fait de ne pas ajouter un type spécial dans le langage, ce qui limite les fonctionnalités d'une expression lambda par rapport à leur équivalent dans d'autres langages mais réduit les impacts dans le langage Java.

Par exemple, il n'est pas possible d'assigner une expression lambda à une variable de type Object parce que le type Object n'est pas une interface fonctionnelle.

Definition du lambda

Une expression lambda est utilisée pour représenter une interface fonctionnelle sous la forme d'une expression de la forme :

- (arguments) -> corps
- L'opérateur -> sépare le ou les paramètres du bloc de code qui va les utiliser.
- Une expression lambda est typée de manière statique. Ce type doit être une interface fonctionnelle.

Pourquoi les Lambda

- Les expressions lambda permettent d'écrire du code plus concis, donc plus rapide à écrire, à relire et à maintenir. C'est aussi un élément important dans l'introduction de la programmation fonctionnelle dans le langage Java qui était jusqu'à la version 8 uniquement orienté objet.
- Une expression lambda est une fonction anonyme : sa définition se fait sans déclaration explicite du type de retour, ni de modificateurs d'accès ni de nom.

- Une expression lambda est donc un raccourci syntaxique qui simplifie l'écriture de traitements passés en paramètre. Elle est particulièrement utile notamment lorsque le traitement n'est utile qu'une seule fois : elle évite d'avoir à écrire une méthode dans une classe.
- Elles permettent d'écrire du code plus compact et plus lisible. Elles ne réduisent pas l'aspect orienté objet du langage qui a toujours été une force mais au contraire, rendent celui-ci plus riche et plus élégant pour certaines fonctionnalités.

Synthaxe

L'écriture d'une expression lambda doit respecter plusieurs règles générales :

- zéro, un ou plusieurs paramètres dont le type peut être déclaré explicitement ou inféré par le compilateur selon le contexte
- les paramètres sont entourés par des parenthèses et séparés par des virgules. Des parenthèses vides indiquent qu'il n'y a pas de paramètre

- lorsqu'il n'y a qu'un seul paramètre et que son type est inféré alors les parenthèses ne sont pas obligatoires
- le corps de l'expression peut contenir zéro, une ou plusieurs instructions.

Synthaxe

- Avec parametre inféré :

```
Consumer<String> afficher = (param) -> System.out.println(param);
```

```
Consumer<String> afficher = param -> System.out.println(param);
```

- Avec paramètre implicite (parenthèses obligatoire) :

```
Consumer<String> afficher = (String param) -> System.out.println(param);
```

Portée des variables

Comme pour les classes anonymes internes, une expression lambda peut avoir accès à certaines variables définies dans le contexte englobant.

Dans le corps d'une expression lambda, il est donc possible d'utiliser :

- les variables passées en paramètre de l'expression
- les variables définies dans le corps de l'expression
- les variables final définies dans le contexte englobant

Portée des variables

L'accès aux variables du contexte englobant est limité par une contrainte forte : seules les variables dont la valeur ne change pas peuvent être accédées.

```
int compteur = 0;
bouton.addActionListener(event -> compteur++);
```

```
referenced from a lambda expression must be final or effectively final
bouton.addActionListener(event -> compteur++);
                                  ^
```

Portée des variables

Il est possible de passer en paramètre un objet mutable et de modifier l'état de cet objet. Le compilateur vérifie simplement que la référence que contient la variable ne change pas.

Comme avec les classes anonymes internes, il est aussi possible de définir un tableau d'un seul élément de type int et d'incrémenter la valeur de cet élément.

```
int[] compteur = new int[1];
bouton.addActionListene(event -> compteur[0]++);
```

L'utilisation de cette solution de contournement n'est pas recommandée d'autant qu'elle n'est pas thread-safe. Il est préférable d'utiliser une variable de type AtomicXXX pour le compteur.

```
AtomicInteger compteur = new AtomicInteger(0);
bouton.addActionListener(event -> compteur.incrementAndGet());
```

Atomicité

En Java, une variable de type atomique (ou une classe atomique) fait référence à un ensemble de classes fournies par la bibliothèque Java dans le package java.util.concurrent.atomic, qui permet de manipuler des variables de manière sûre dans des environnements multithread, sans nécessiter d'utiliser explicitement des mécanismes de synchronisation tels que les verrous (synchronized, Lock, etc.).

```
AtomicInteger atomicInt = new AtomicInteger(0);

// Opérations atomiques
atomicInt.incrementAndGet();  // Incrémente de 1 et retourne la nouvelle valeur
atomicInt.decrementAndGet();  // Décrémente de 1 et retourne la nouvelle valeur
atomicInt.getAndSet(5);       // Définit la valeur à 5 et retourne l'ancienne valeur
```

Atomicité

Pourquoi des variables atomiques ?

- Dans un contexte multithread, il est important d'assurer que l'accès à certaines variables soit fait de manière atomique, c'est-à-dire qu'une opération sur une variable se fasse en une seule étape indivisible. Sans cela, il peut y avoir des situations de concurrence où plusieurs threads modifient simultanément une même variable, ce qui pourrait entraîner des résultats imprévisibles ou incorrects.

Atomicité

Les variables atomiques résolvent ce problème en offrant des opérations atomiques pour des types de données simples, sans avoir besoin de synchroniser manuellement les accès, car elles utilisent des instructions de bas niveau fournies par le processeur, telles que Compare-And-Swap (CAS), qui garantit que l'opération sur la variable se fait de manière sécurisée même en présence de plusieurs threads.

Avantages des variables atomiques :

- Sécurité dans les environnements multithread
- Performances optimisées

Utilisation

Une expression lambda ne peut être utilisée que dans un contexte où le compilateur peut identifier l'utilisation de son type cible (target type) qui doit être une interface fonctionnelle

Par exemple, la déclaration d'une expression lambda peut être utilisée directement en paramètre d'une méthode.

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

Utilisation

Une expression lambda est définie grâce à une interface fonctionnelle : une instance d'une expression lambda qui implémente cette interface est un objet. Cela permet à une expression lambda :

- de s'intégrer naturellement dans le système de type de Java
- d'hériter des méthodes de la classe Object

Attention cependant, une expression lambda ne possède pas forcément d'identité unique : la sémantique de la méthode equals() n'est donc pas garantie.

Les references de méthodes

Les références de méthodes permettent d'offrir une syntaxe simplifiée pour invoquer une méthode comme une expression lambda : elles offrent un raccourci syntaxique pour créer une expression lambda dont le but est d'invoquer une méthode ou un constructeur.

Les references de méthodes

type	syntaxe	Exemple
Référence à une méthode statique	nomClasse::nomMethodeStatique	String::valueOf
Référence à une méthode sur une instance	objet::nomMethode	personne::toString
Référence à une méthode d'un objet arbitraire d'un type donné	nomClasse::nomMethode	Object::toString
Référence à un constructeur	nomClasse::new	Personne::new

Les references de méthodes

type	Référence de méthode	Expression lambda
Référence à une méthode statique	System.out::println	x -> System.out.println(x)
Référence à une méthode sur une instance	objet::nomMethode	x -> monObject.maMethode(x)
Référence à une méthode d'un objet arbitraire d'un type donné	String::compareToIgnoreCase	(x, y) -> x.compareToIgnoreCase(y)
Référence à un constructeur	MaClasse::new	() -> new MaClasse();

Les interfaces fonctionnelles

Le package java.util.function propose en standard des interfaces fonctionnelles d'usage courant. Toutes les interfaces de ce package sont annotées avec @FunctionalInterface.

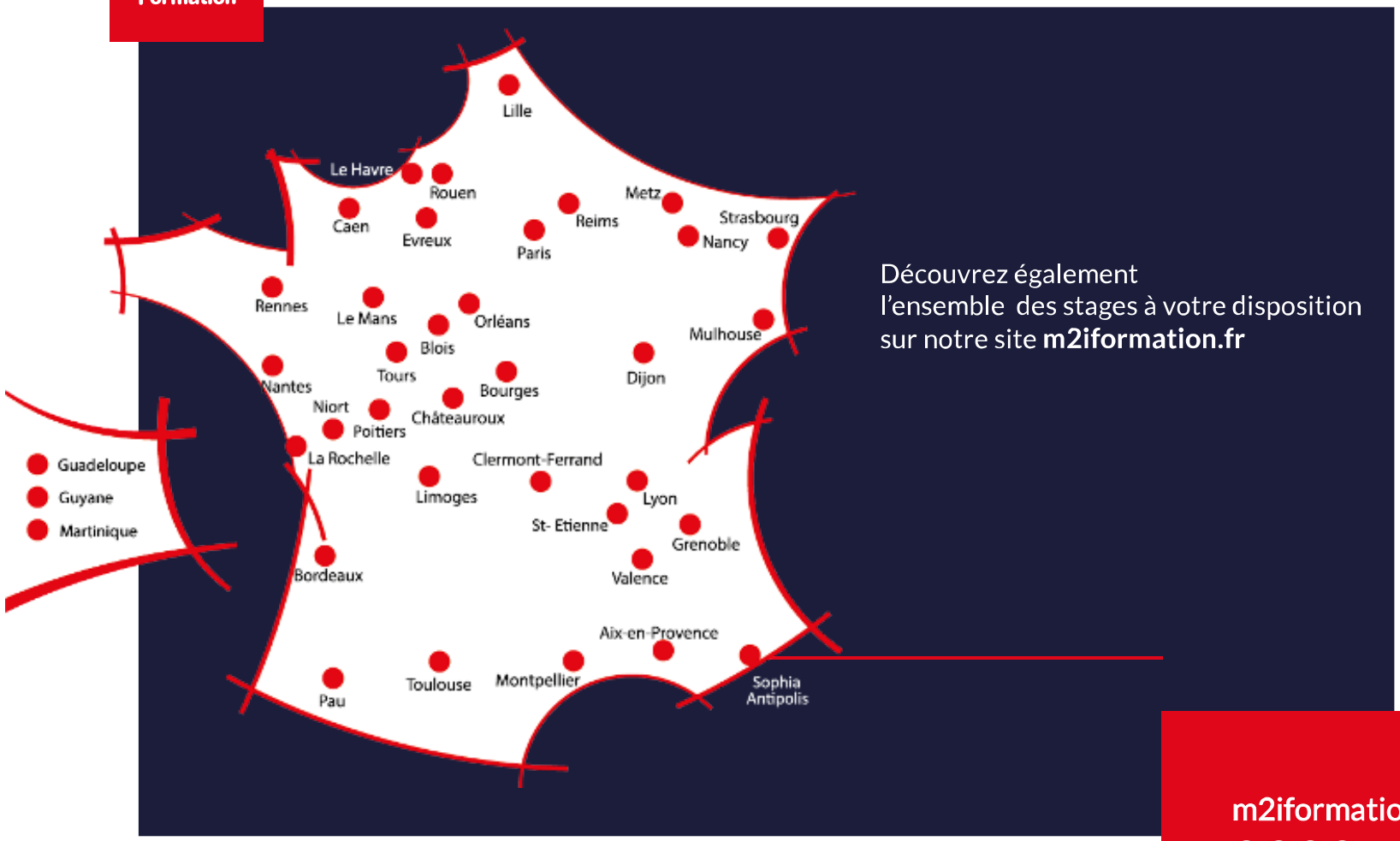
Les interfaces fonctionnelles

- Function : une fonction unaire qui permet de réaliser une transformation. Elle attend un ou plusieurs paramètres et renvoie une valeur. La méthode se nomme apply()
- Consumer : une fonction qui permet de réaliser une action. Elle ne renvoie pas de valeur et attend un ou plusieurs paramètres. La méthode se nomme accept()
- Predicate : une fonction qui attend un ou plusieurs paramètres et renvoie un booléen. La méthode se nomme test()
- Supplier : une fonction qui renvoie une instance. Elle n'attend pas de paramètre et renvoie une valeur. La méthode se nomme get(). Elle peut être utilisé comme une fabrique

Les interfaces fonctionnelles

Le nom des interfaces fonctionnelles qui attendent en paramètre une ou plusieurs valeurs primitives sont préfixées par le type primitif.

Le nom des interfaces fonctionnelles qui renvoient une valeur primitive sont suffixées par toXXX.



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr