Espressioni Lambda & Java Stream

Alessandro Midolo, Ph.D. Student
alessandro.midolo@phd.unict.it

Tutorato Ingegneria del Software

A.A. 2021/2022

Classi Anonime

Le classi anonime sono delle classi interne senza un **nome** → Non è possibile istanziare una classe anonima

Vengono dichiarate ed utilizzate in una singola espressione nella porzione di codice in cui verranno usate

Si può sia **estendere** una classe già esistente oppure **implementare** un’interfaccia

Espressioni Lambda

E’ una funzione anonima che mantiene una struttura simile a quella di un metodo: parametri, corpo e tipo di ritorno

x -> System.out.println(“Stampo la variabile x: “ + x)

Può prendere anche due parametri come input

(x,y) -> x+y

Il tipo dei parametri passati come input è determinato automaticamente

Nel caso di più istruzioni nel corpo della EL, bisogna utilizzare le graffe

Stile Imperativo & Stile Dichiarativo

Lo stile imperativo dà un maggiore “controllo” allo sviluppatore su ciò che il programma deve fare. Ad esempio per eseguire un ciclo su una lista:

- Implementare svariate linee di codice
- Comprensione del comportamento difficoltosa → bisogna leggere tante linee di codice per capire cosa sta facendo l’istruzione
- Il ciclo è “**esterno**” rispetto al codice della lista

Nello stile dichiarativo, il codice che si occupa di implementare delle funzionalità è **interno** alla classe lista stessa

Stile Funzionale

A partire da Java 8 è possibile implementare lo stile funzionale all'interno del codice. Questo permette la creazione di **funzioni di ordine più alto**

Queste non sono altro che funzioni che prendono come parametri altre funzioni → E' possibile **passare funzioni ai metodi, creare funzioni dentro i metodi e ritornare funzioni dai metodi**

Una libreria che permette l'implementazione dello stile funzionale nelle proprio classi e metodi è l'interfaccia **Collection**

Stream

Java 8 introduce i metodi **default** per le interfacce → un metodo default è un metodo che non modifica lo stato dell'istanza della classe

Questo permette di inserire facilmente questi metodi all'interno di interfacce esistenti senza alterarne la compatibilità

stream() è un metodo di default dell'interfaccia **Collection**, restituisce un oggetto di tipo Stream<T> dove T è il tipo dei valori contenuti nella collection su cui si invoca il metodo

L'interfaccia Stream presenta dei metodi che prendono come parametri delle funzioni

Metodi dell'interfaccia Stream

Metodi Lazy / Operazioni intermedie

- *filter()*
- *map()*

Metodi Eager / Operazioni terminali

- *reduce()*
- *count()*
- *collect()*
- *findAny()*
- *forEach()*

Tutte le operazioni che restituiscono uno *stream* sono operazioni intermedie

Documentazione: Interfaccia Stream

Filter e Predicate

filter(Predicate<T> p) è un metodo intermedio di stream che prende in input una funzione che restituisce un booleano → **Predicate**

Restituisce uno Stream contenente tutti gli elementi che hanno soddisfatto il predicato

filter(s -> s.equals("Ciao"))

Predicate è un'interfaccia funzionale → presenta un solo metodo che prende in input un tipo generico Object e restituisce un booleano

Predicate<Integer> isPositive = x -> x>=0

Reduce

reduce(T identity, BinaryOperator<T> accumulator) è un metodo dell'interfaccia Stream che prende in input un valore identità di partenza dello stesso tipo degli elementi presenti nello stream, e una funzione di accumulazione che ritorna un singolo valore

La si usa se si vuole passare da un insieme di valori ad un singolo valore

reduce(0, (acc,v) -> acc+v)

E' possibile passare come accumulatore direttamente un metodo di una classe

reduce(0, Integer::sum)

reduce(Integer::sum)

Map e Function

map(Function<T, R> mapper) prende in input una funzione *mapper* la quale viene applicata per ogni elemento dello stream, restituendo uno stream che conterrà i nuovi valori restituiti dalla funzione mapper

```
map(x -> x * 2)
```

```
map(Persona::getEta)
```

```
map(p -> p.getEta())
```

Function è un'interfaccia funzionale → presenta un unico metodo che prende in input un oggetto di tipo T e restituisce un nuovo oggetto di tipo R (T e R possono essere uguali)

```
Function<Integer, Integer> multiply = x -> x*2
```

Collect

La funzione ***collect()*** di Stream permette di prendere in input uno stream e restituire una specifica *Collection*

Prende in input un ***Collector***, questa classe mette a disposizione diversi metodi per raggruppare dati all'interno di collections

Un esempio è il metodo ***toList()*** il quale permette di raggruppare i valori all'interno di una lista

```
collect(Collectors.toList())
```