



Lambda Expressions in Java 8: Uso e Tipaggio

Mattia D'Autilia - 5765968 - mattia.dautilia@stud.unifi.it,

Alex Foglia - 6336805 - alex.foglia@stud.unifi.it

- 1 Java e Java 8
- 2 Lambda Expression Java 8

Java e Java 8

1.1 : Java

- **Java** è un linguaggio di programmazione di alto livello, principalmente *orientato agli oggetti*, ma accetta anche altri paradigmi come quello *funzionale* ed è a *tipizzazione statica*.
- E' stato creato per soddisfare cinque obiettivi primari:
 - ❶ Essere "semplice e familiare";
 - ❷ Essere "robusto e sicuro";
 - ❸ Essere indipendente dalla piattaforma, da qui il detto "*Write one, run everywhere*";
 - ❹ Contenente strumenti e librerie per il networking;
 - ❺ Essere progettato per eseguire codice da sorgenti remote in modo sicuro.

1.2 : Evoluzione di Java

- Quando Java nacque nel 1995, era un linguaggio molto semplice. Con il passare degli anni sono state introdotte gradualmente tante caratteristiche, diventando un linguaggio sempre più potente e completo, in particolare con la versione 5 e 7. Quello che però non era mai cambiato sino ad ora, era la coerenza d'essere un linguaggio orientato agli oggetti.
- Negli ultimi anni però la scena della programmazione mondiale è cambiata. In particolare con l'avvento di processore multi-core nell'uso domestico, la *programmazione funzionale* è stata rivalutata. Con linguaggi moderni come *Scala* e *Groovy* è possibile scrivere algoritmi con un numero di righe nettamente inferiore, rispetto a quello che si poteva fare con Java, che qualcuno stava già definendo un linguaggio morto in quanto con le versioni 6 e 7 aveva solo modernizzato alcune librerie, estromettendo le tanto richieste *Lambda Expression*.

1.3 : Java 8

- Con l'avvento di **Java 8** fu però apportata una vera e propria rivoluzione, la più innovativa in tutta la storia di Java. Con l'introduzione delle *Espressioni Lambda* e la possibilità di *referenziare i metodi*, la **filosofia funzionale**, fa il suo ingresso nella programmazione Java.
- Ora vedremo come affrontare la nuova sfida, che è quella di far convivere i due paradigmi, quello *orientato agli oggetti* e quello *funzionale*, in modo tale da ottenere il meglio della programmazione.

Lambda Expression in Java8

2.1 : Definizione

- Una **Lambda Expression** è detta:
 - **Funzione anonima** (in inglese "**anonymous function**"), in quanto si tratta proprio di una funzione, quindi non è un metodo appartenente a una classe e chiamato tramite un oggetto, ma è una funzione senza nome;
 - **Chiusa** (in inglese "**closure**"), in quanto fa uso di variabili che non sono parametri e non sono variabili locali al blocco di codice che definisce l'espressione.
- Inoltre le **Lambda Expression** permettono:
 - di scrivere codice più semplice, leggibile e meno verboso;
 - di adottare nuovi pattern di programmazione, basati sulle funzioni di **ordine superiore**.

2.2 : Sintassi

- In Java 8 la sintassi generale di una funzione è la seguente:

$$([\text{lista di parametri}]) \rightarrow (\text{espressione di ritorno})$$
$$([\text{lista di parametri}]) \rightarrow \{ \text{blocco di codice come da} \\ \text{prassi procedurale} \}$$

- Il vantaggio principale nell'uso di una *Lambda Expression*, risiede nella sinteticità dell'espressione. In alcuni casi è possibile **omettere** :
 - ❶ il **tipo dei parametri** quando non c'è possibilità di errore;
 - ❷ le parentesi tonde che circondano la lista dei parametri, nel caso quest'ultima fosse costituita da un unico elemento;
 - ❸ la keyword *return* quando esiste una singola espressione da valutare.

2.2 : Sintassi

- Abbiamo visto come una funzione viene definita in **Lambda Calcolo**. Facciamo l'esempio più semplice, la funzione identità:

$$\lambda x.x$$

- λ : rappresenta l'*astrazione*;
 - la prima x : rappresenta la *variabile di input*;
 - la seconda x : rappresenta il *corpo della funzione*.
- Con queste regole di sintassi, tale funzione può essere scritta in uno qualsiasi dei seguenti modi:

$$(x) \rightarrow (x)$$

Oppure:

$$x \rightarrow (x)$$

$$x \rightarrow \{ \text{return } x; \}$$

2.3 : Quando usare le Lambda Expression

- Dovremmo usare le *Lambda Expression*, quando il nostro obiettivo è quello di passare in maniera dinamica un certo algoritmo ad un altro metodo. Questo serve per eseguire l'algoritmo in un contesto definito dal metodo a cui stiamo passando l'algoritmo.
- In generale, passare una *Lambda Expression* a un metodo, significa delegare allo stesso la decisione sul *se* e sul *quando* valutare tale lambda.

2.3 : Quando usare le Lambda Expression

- In *Java 8* è possibile usare una *Lambda Expression* per:
 - **Assegnarla a una referenza** : trattarla come valore;
 - **Passarla come parametro** : parametrizzarla come comportamento;
 - **Ottenerla come risultato di una valutazione** : come un qualunque oggetto.

2.4 : Funzioni di ordine superiore

- Nello studio del *Lambda Calcolo*, abbiamo visto come le funzioni sono entità di prima classe: possono essere argomenti o anche risultato di una funzione.
- Le **funzioni di ordine superiore** (in inglese "**higher order functions**") sono funzioni che ammettono a loro volta funzioni come argomento e/o risultato. L'operatore matematico *derivata* è un esempio di funzione d'ordine superiore.

2.4 : Funzioni di ordine superiore

- In Java le *Lambda Expression* possono essere *funzioni di ordine superiore*.
- La possibilità di definire funzioni di ordine superiore rende Java 8 a tutti gli effetti un linguaggio che supporta anche il paradigma del *Lambda Calcolo*.
- Vediamo alcuni esempi di implementazione in Java del **Lambda Calcolo**.

2.5 : Lambda calcolo vs Lambda Expressions

- Funzione che riceve un intero x e restituisce $x+1$:

- **Lambda calcolo:**

$$\lambda x.x + 1$$

- **Java:**

```
(x) -> (x+1);
```

2.5 : Lambda calcolo vs Lambda Expressions

- La **curryficazione** di funzione binaria utilizzando *higher-order-functions*:

- Lambda calcolo:**

$$\lambda xy.x + y$$
$$\lambda x.\lambda y.x + y$$

- Java:**

```
(x, y) -> (x+y);
```

```
(x) -> (y) -> (x + y);
```


2.5 : Lambda calcolo vs Lambda Expressions

- Booleani:
 - Lambda calcolo:

$True = \lambda x. \lambda y. x$

$False = \lambda x. \lambda y. y$

- Java:

```
True = (x) -> (y) -> (x);  
False = (x) -> (y) -> (y);
```

2.5 : Lambda calcolo vs Lambda Expressions

- Ricorsione:

- Supponiamo di voler scrivere una lambda per calcolare il *fattoriale* di un numero intero:

- In **Lambda Calcolo**, sarebbe:

$$\text{let } Fact = \lambda n. \text{if}(n = 0) \quad 1 \quad \text{else} \quad n * Fact(n - 1)$$

- Che in **Java** sarebbe:

```
Fact = (n) -> (n == 0 ? 1 : Fact.apply(n - 1));
```

- Come sappiamo dal *Lambda Calcolo*, questa espressione non è corretta, poiché la lambda, essendo *funzione anonima*, non può sapere di chiamarsi *Fact*.

2.5 : Lambda calcolo vs Lambda Expressions

- **Ricorsione - Continuo:**
 - Possiamo quindi implementare un **Combinatore di Punto Fisso**.
 - Nel **Lambda Calcolo**, un noto *Combinatore di Punto Fisso* è il seguente:
- $$fix = \lambda f. (\lambda x. f(xxy)) (\lambda x. f(\lambda y. xxy))$$
- In **Java** possiamo definire suddetto *combinatore* come segue:

```
Y = y -> f -> x -> f
    .apply(y.apply(y).apply(f)).apply(x);
Fix = Y.apply(Y);
```

2.5 : Lambda calcolo vs Lambda Expressions

- Ricorsione - Continuo:
 - Applicando il *Combinatore di Punto Fisso* al fattoriale, otteniamo quindi:
 - In Lambda Calcolo:

$let\ G = fix\ Fact$

- In Java:

```
fixfactorial = Fix.apply(Fact);
```

2.6 : Tipaggio

- Finora abbiamo trattato le Lambda senza specificare il modo in cui queste possono essere assegnate a una variabile (o usarle come parametro di un metodo).
- Abbiamo inoltre visto come alcune lambda expressions possono essere valutate chiamando il metodo *apply()*
- Entriamo nel dettaglio

2.6 : Tipaggio

- Java è un linguaggio *strongly typed*
- Ogni sotto-espressione di qualsiasi espressione deve essere *ben tipata*
- Quindi il tipo di una lambda deve essere coerente con il suo *tipo atteso*

2.6 : Tipaggio

- Per esempio, se scriviamo:

```
Point p = (x) -> (x+1);|
```

- Otteniamo il seguente errore:

The target type of this expression
must be a functional interface

- Perchè la referenza a una lambda deve essere un' *Interfaccia Funzionale*

2.6.1 : Interfacce funzionali

- Una qualunque interfaccia è funzionale se e solo se contiene esattamente un solo metodo astratto.
- La signature di questo metodo descrive il *TIPO* di una Lambda Expression
- Quindi il tipo di una Lambda Expression è un' interfaccia funzionale
- Una lambda è come se fosse un'istanza di una classe concreta che implementa l'interfaccia funzionale

2.6.1 : Interfacce funzionali

- Adesso possiamo capire l'errore precedente:

```
Point p = (x)->(x+1);|
```

- In questo caso il type checker dà errore in quanto la lambda expression non matcha con la signature di alcun metodo astratto di un'interfaccia funzionale.

2.6.2 : Typing

- Il tipo di una lambda viene inferito rispetto alla sua interfaccia funzionale
- Il type-checker può stabilire se una lambda è ben tipata quando quest'ultima matcha con la signature del metodo astratto
- Vediamo un esempio

2.6.2 : Typing

- Java mette a disposizione la seguente interfaccia funzionale:

```
public interface Function<T,S> {  
    public S apply(T s);  
}
```

- Quindi come fa il type checker a stabilire se la seguente lambda è ben tipata?

```
Function<Integer , Boolean> fun=x->(x>=0);
```

2.6.2 : Typing

- Controlla se, assunto il parametro x di tipo Integer, il body della lambda è un Boolean.
- Dunque, assunto che x è un intero, è vero che $x \geq 0$ è un booleano? Sì, la lambda è ben tipata.