



## Lambda Expressions in Java 8: Uso e Tipaggio

Mattia D'Autilia - 5765968 - [mattia.dautilia@stud.unifi.it](mailto:mattia.dautilia@stud.unifi.it),  
Alex Foglia - 6336805 - [alex.foglia@stud.unifi.it](mailto:alex.foglia@stud.unifi.it)

# Indice presentazione

- 1 Java e Java 8
- 2 Lambda Expression in Java 8
- 3 Tipaggio Lambda Expression
- 4 Lambda Expression vs Design Pattern
- 5 Tipi Intersezione
- 6 Conclusioni

# Java e Java 8

# 1.1 : Java

- **Java** è un linguaggio di programmazione di alto livello, strongly typed, principalmente *orientato agli oggetti*, ma ammette 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 Contenere strumenti e librerie per il networking;
  - ❺ 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, che lo hanno reso un linguaggio sempre più potente e completo, in particolare con le versioni 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 processori 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 *Lambda Expression*. tuttavia molto richieste.

## 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 Java 8

## 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;
- Essa rappresenta una **Closure**, nel senso che all'interno di una lambda possono occorrere eventuali variabili libere dell'ambiente di definizione.
- 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** :
  - 1 il **tipo dei parametri** quando non c'è possibilità di errore;
  - 2 le parentesi tonde che circondano la lista dei parametri, nel caso quest'ultima fosse costituita da un unico elemento;
  - 3 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:

$$\lambda x.x$$

- $\lambda$ : rappresenta l'*astrazione*;
  - la prima  $x$  : rappresenta la *variabile di input*;
  - la seconda  $x$  : rappresenta il *corpo della funzione*.
- Con le regole sintattiche di Java precedentemente esposte, tale funzione può essere scritta in uno qualsiasi dei seguenti modi:

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

$$x \rightarrow (x)$$

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

## 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 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**.
- 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 funzionale.

## 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);
```

# Tipaggio Lambda Expression

## 3.1 : Tipaggio

- Finora abbiamo trattato le *Lambda Expression* senza specificare il modo in cui queste possono essere:
  - **Assegnate a una referenza;**
  - **Passate come parametro;**
  - **Ottenute come risultato di una valutazione.**

*Entriamo nel dettaglio*

## 3.1 : Tipaggio

- Java è un linguaggio **strongly typed**, ciò significa che il programmatore è tenuto a specificare il tipo di ogni elemento che durante l'esecuzione denota un valore, e il linguaggio garantisce che tale valore sia utilizzato in modo coerente con il tipo specificato.
- Quindi, ogni sotto-espressione di qualsiasi espressione deve essere **ben tipata**.
- Tornando alle *Lambda Expression*, il **tipo** di una *Lambda Expression* deve essere coerente con il suo **Target Type**.

## 3.1 : 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"

- Questo avviene, perchè la *referenza* a una *Lambda Expression* deve essere un' **Interfaccia Funzionale**.

## 3.2 : Interfacce funzionali

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

## 3.2 : Interfacce funzionali

- Adesso possiamo capire l'errore precedente:

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

- In questo caso il *type checker* dà errore in quanto si aspetta che il *target type* della *Lambda Expression* sia di *tipo Point*, quindi controlla che *Lambda Expression* sia di *tipo Point*, ma non può avere *tipo Point* perchè *Point* non è un'Interfaccia Funzionale.
- Può avvenire che *Point* sia effettivamente un'interfaccia funzionale, ma la lambda potrebbe non matchare con la signature del metodo astratto.

## 3.3 : Typing

- Una *Lambda Expression* può avere un *tipo solo*, se questo è un' *Interfaccia Funzionale*.
- Il *contesto* definisce il *target type* che bisogna assegnare alla *Lambda Expression*.
- Il **target type** è :
  - Il *tipo di ritorno* se il return di un metodo è una *Lambda Expression*;
  - Il *tipo della variabile* se assegno alla variabile una *Lambda Expression*;
  - Il *tipo del parametro* se passo una *Lambda Expression* come parametro attuale;
  - Il *tipo* a cui faccio *cast* se la *Lambda Expression* è un *type-cast*.



## 3.3 : Typing

- A questo punto, il *type-checker* considera il *target-type* e controlla se può assegnarlo alla *Lambda Expression*.
  - Se il *target-type* non è un'*Interfaccia Funzionale*, l'assegnazione viene subito **negata**.
  - Se il *target-type* è un'*Interfaccia Funzionale*, l'assegnazione avviene se e solo se, la *Lambda Expression* **matcha** la *signatura* del metodo astratto, la quale è definita in questo modo :

**T metodo(T1 x)**

- Si controlla se assegnando alla variabile *Lambda-astratta* il **tipo T1**, il *body* della *Lambda Expression* ha **tipo T**.

## 3.3 : Typing

- Java mette a disposizione le seguenti *Interfacce Funzionali*, che possono essere usate per descrivere la *signatura* di diverse *Lambda Expression*:
  - **Predicate**<T>     $T \rightarrow \text{boolean}$
  - **Function**<T,R>     $T \rightarrow R$
  - **BinaryOperator**<T>     $(T,T) \rightarrow T$
  - **BiFunction**<T,U,R>     $(T,U) \rightarrow R$
  - etc...
- Inoltre Java, ti permette di definire nuove *Interfacce Funzionali*, indicandole aggiungendo l'annotazione **@FunctionInterface**.

## 3.3 : Typing

- *Esempio* :  $\text{Function}\langle T, R \rangle \quad T \rightarrow R$

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

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

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

## 3.3 : Typing

- **Controllo tipaggio:**

- Assunto il *parametro*  $x$  di tipo Integer, il *body* della *Lambda Expression* è un Boolean.
- Dunque, assunto che  $x$  è un intero, è vero che  $x \geq 0$  è un booleano? Si, la *Lambda Expression* è *ben tipata*.

## 3.3 : Typing

- E' possibile che il tipo di una *Lambda Expression* sia compatibile con più *target type*.

Predicate<Integer> p = (x) -> (x%2==0);

Function<Integer, Boolean> f = (x) -> (x%2==0);

## 3.4 : Semantica

Supponiamo di aver definito la seguente interfaccia funzionale:

```
interface I{  
    boolean check(int n);  
}
```

Supponiamo inoltre di avere anche definito una siffatta classe:

```
class A<T> {  
    ...  
    T method(I i){  
        ...  
        int n;  
        ...  
        i.check(n);  
        ...  
    }  
}
```

## 3.4 : Semantica - Continuo:

Vediamo cosa avviene se in un client eseguo

```
A<Integer> a = new A<>();
```

```
...
```

```
a.method((x) -> (x > 0));
```

Quando nel body del metodo incontro l'invocazione di check sulla lambda, viene eseguita l'istruzione

```
return x > 0;
```

Con il parametro formale  $x$  sostituito dal parametro attuale  $n$  (definito dal contesto del metodo).

# Lambda Expression vs Design Pattern



## 4.1 : Perché?

- Finora abbiamo visto alcune semplici applicazioni delle *Lambda Expression*.
- Adesso vediamo come viene **semplificato** il codice nel caso del loro utilizzo al posto dei *Design Pattern*.
- Qui di seguito faremo l'esempio dei seguenti *Design Pattern*:
  - **Strategy**;
  - **Template Method**.

## 4.2 : Strategy

- Il *Design Pattern Strategy* con l'utilizzo delle *Lambda Expression*, permette di scegliere fra funzioni diverse con la stessa signature a run-time.
- Esempio:

```
public class Vendable {  
    private Strategy s;  
    private double basePrice;  
    public Vendable(double base) {  
        this.basePrice=base;  
    }  
    public void setStrategy(Strategy s) {  
        this.s=s;  
    }  
    public double getPriceWithStrategy() {  
        return s.getPrice(basePrice);  
    }  
}
```

```
public interface Strategy {  
    public double getPrice(double base);  
}
```

## 4.2 : Strategy

- Prima dell'introduzione delle *Lambda Expression*, un client che avesse voluto implementare uno *Strategy* avrebbe avuto due opzioni:
  - *Classe anonima*;
  - *Definire una classe concreta che implementa l'interfaccia Strategy*.
- **Classe anonima:**

```
Vendable v = new Vendable(100);  
v.setStrategy(new Strategy() {  
  
    @Override  
    public double getPrice(double base) {  
        return 0.75*base;  
    }  
  
});
```

## 4.2 : Strategy

- Definire una classe concreta che implementa l'interfaccia Strategy:

```
public class StrategyDiscount implements Strategy {  
    @Override  
    public double getPrice(double base) {  
        return 0.75*base;  
    }  
}  
  
Vendable v = new Vendable(100);  
Strategy ss = new StrategyDiscount();  
v.setStrategy(ss);
```

- Lambda Expression:

```
Vendable v = new Vendable(100);  
v.setStrategy((x)->(0.75*x));
```

## 4.3 : Template Method

- Il *Design Pattern Template Method* con l'utilizzo delle *Lambda Expression*, permette di rimpiazzare il polimorfismo del metodo astratto con la composizione, passando una funzione al costruttore;
- Esempio: Polimorfismo del metodo astratto

```
public abstract class Template {  
  
    public final void doTripTemplate(String dataComing,String dataReturning ){  
        busComing(dataComing);  
        doDay1();  
        doDay2();  
        busReturning(dataReturning);  
    }  
  
    public void busComing(String data){  
        System.out.println("Il Bus parte il : " + data);  
    }  
  
    public abstract void doDay1();  
    public abstract void doDay2();  
  
    public void busReturning(String data){  
        System.out.println("Il Bus torna il : " + data);  
    }  
}
```

## 4.3 : Template Method

- Prima dell'introduzione delle *Lambda Expression*, un client che avesse voluto implementare un *Template Method* avrebbe dovuto estendere in vari modi la classe concreta contenente il/i metodo/i astratto/i per differenziarlo/i:

```
public class ExtendOne extends Template {  
  
    @Override  
    public void doDay1() {  
        System.out.println("Il primo giorno i turisti visiteranno Lecce");  
    }  
  
    @Override  
    public void doDay2() {  
        System.out.println("Il secondo giorno i turisti visiteranno Foggia");  
    }  
}  
  
public class ExtendTwo extends Template {  
  
    @Override  
    public void doDay1() {  
        System.out.println("Il primo giorno i turisti visiteranno Firenze");  
    }  
  
    @Override  
    public void doDay2() {  
        System.out.println("Il secondo giorno i turisti visiteranno Empoli");  
    }  
}
```

## 4.3 : Template Method

- Nel metodo *main()* avremo:

```
ExtendOne trip1pt = new ExtendOne();
trip1pt.doTripTemplate("12/12/2012", "15/12/2012");
ExtendTwo trip1ps = new ExtendTwo();
trip1ps.doTripTemplate("14/04/2008", "17/04/2008");
```

- Ora implementiamo lo stesso programma con le **Lambda Expression**:

```
public class TemplateLambda {
    private whereDay wfd;
    private whereDay wsd;

    TemplateLambda(whereDay wfd, whereDay wsd) {
        this.wfd = wfd;
        this.wsd = wsd;
    }

    public final void doTripTemplate(String dataComing, String dataReturning) {
        busComing(dataComing);
        wfd.printWhereDay();
        wsd.printWhereDay();
        busReturning(dataReturning);
    }

    public void busComing(String data) {
        System.out.println("Il Bus parte il : " + data);
    }

    public void busReturning(String data) {
        System.out.println("Il Bus torna il : " + data);
    }
}
```

```
public interface whereDay {
    void printWhereDay();
}
```

## 4.3 : Template Method

- Nel metodo *main()*, a questo punto avremo:

```
whereDay fd;
whereDay sd;
TemplateLambda tl;
fd = ()->System.out.println("Il primo giorno i turisti visiteranno Lecce");
sd = ()->System.out.println("Il primo giorno i turisti visiteranno Foggia");
tl = new TemplateLambda(fd,sd);
tl.doTripTemplate("12/12/2012", "15/12/2012");

fd = ()->System.out.println("Il primo giorno i turisti visiteranno Firenze");
sd = ()->System.out.println("Il primo giorno i turisti visiteranno Empoli");
tl = new TemplateLambda(fd,sd);
tl.doTripTemplate("14/04/2008", "17/04/2008");
```



## 4.4 : Altri Pattern

- Oltre a quelli appena descritti, si possono elencare tanti altri *Design Pattern* e su come quest'ultimi possono essere implementati in modo diverso, possiamo dire *migliore*, con l'utilizzo delle *Lambda Expression*.
- Tra i tanti, uno che sarebbe giusto nominare, è il *Design Pattern Decorator*, che evitiamo di dimostrare.
  - Il *Design Pattern Decorator* con l'utilizzo delle *Lambda Expression*, permette di Passare una *Lambda Expression* al metodo interessato, per *modificarne il comportamento* (una *Lambda Expression* che può chiamare un'altra *Lambda Expression*, con la *stessa signature*, ma con *argomenti diversi*).;

# Tipi Intersezione

## 5.1 : I tipi intersezione

- In Java 8 è stato introdotto un nuovo tipo.
- I *tipi intersezione*.

## 5.2 : Regole dei tipi intersezione

- Se  $T$ ,  $T'$  sono tipi, allora  $T \& T'$  è un tipo.
- Un termine che ha tipo  $T$  e  $T'$ , ha anche il tipo  $T \& T'$ .
- Un termine che ha tipo  $T \& T'$ , ha sia tipo  $T$  che tipo  $T'$ , ossia ha tutte le proprietà del tipo  $T$  e le proprietà del tipo  $T'$ .
- I tipi *intersezione* sono particolarmente utili se usati assieme alle lambda, poichè castare una *Lambda Expression* a un tipo *interesezione* permette di aggiungere comportamenti diversi alla stessa *Lambda Expression*.
- Questo è l'argomento della successiva trattazione da parte della Prof.

# Conclusioni

## 6.1 : Conclusioni

- Abbiamo visto un'introduzione all'uso delle **Lambda Expression** in Java.
- Java è un linguaggio nato per essere semplice, familiare e sicuro.
- L'assenza delle *Lambda Expression* rappresentava una grande lacuna rispetto ad altri linguaggi.
- La loro aggiunta permette veramente di scrivere codice più conciso ed elegante.

## 6.1 : Conclusioni

- Eliminata la necessità di dover implementare *classi anonime* al volo.
- Semplificazione di molti *Design Pattern*.
- Eliminazione di alcuni *Design Pattern* (il Command può essere implementato con una *Lambda Expression*).
- L'utilizzo delle *Lambda Expression* si combina bene con altri miglioramenti apportati al linguaggio:
  - *Referenza ai metodi statici*;
  - *Tipi intersezione*.