

Lambda Expression

Lambda Expression

- ❑ Le espressioni Lambda sono il più importante miglioramento alla **sintassi** di Java in **Java 8** iniziato con Java 7.
- ❑ Nonostante l'apparente aspetto criptico della notazione Lambda in realtà, comprendendo da cosa sono originate, le espressioni Lambda sono piuttosto semplici
- ❑ Le espressioni Lambda sono giusto un modo **per passare un blocco di codice** a un metodo. Questo è tutto quello che sono.
- ❑ Tipica situazioni dove si può avere bisogno di passare del codice in un metodo è per esempio con **Thread**, per eseguirlo nel suo proprio thread oppure, con una interfaccia grafica di Java FX, per passare codice da eseguire in modo asincrono a un **onClick** su un button
- ❑ Per capire le espressioni Lambda è necessario avere una comprensione delle **interface** e della implementazione di **classi anonime**.

Passare codice a una classe anonima

- ❑ Le espressioni Lambda sono il più importante miglioramento alla **sintassi** di Java in **Java 8** iniziato con **Java 7**.
- ❑ Nonostante l'apparente aspetto criptico della notazione Lambda in realtà, comprendendo da cosa sono originate, le espressioni Lambda sono piuttosto semplici
- ❑ Le espressioni Lambda sono giusto un modo **per passare un blocco di codice** a un metodo. Questo è tutto quello che sono.

Passare codice a una classe anonima

- ❑ Iniziamo creando una classe che possiamo chiamare Runner() e immaginiamo di voler passare del codice al metodo run()

```
3 class Runner {  
4     public void run() {  
5  
6     }  
7 }  
8 public class LambdaDemo {  
9  
10     public static void main(String[] args) {  
11  
12     }  
13  
14 }
```

Passare codice a una classe anonima

- ❑ Cosa avremmo fatto nelle precedenti versioni di Java?
 - Prima avremmo definito una interface con un metodo che dichiara il codice da eseguire, passandola al metodo run()

```
3 interface Executable {  
4     void execute();  
5 }  
6  
7 class Runner {  
8     public void run(Executable e) {  
9         System.out.println("Executing code block ...");  
10        e.execute();  
11    }  
12 }
```

Passare codice a una classe anonima


- Poi istanziando la classe Runner avremmo eseguito il metodo run() al quale avremmo passato il codice da eseguire, attraverso una anonymous class come parametro

```
7 class Runner {  
8     public void run(Executable e) {  
9         System.out.println("Executing code block ...");  
10        e.execute();  
11    }  
12 }  
13 public class LambdaDemo {  
14     public static void main(String[] args) {  
15         Runner runner = new Runner();  
16         runner.run();  
17     }  
18 }
```

Passare codice a una classe anonima

- Implementando l'interface Executable nel parametro di run(...)

```
14 public class LambdaDemo {
15     public static void main(String[] args) {
16         Runner runner = new Runner();
17         runner.run(new Executable() {
18             @Override
19             public void execute() {
20                 System.out.println("Hello from execute ...");
21             }
22         });
23     }
24 }
25 }
```



The screenshot shows the IDE's console window with the following output:

```
<terminated> LambdaDemo [Java Application] C:\Program Files\Java\
Executing code block ...
Hello from execute ...
```

Passare codice a una classe anonima

- Il punto è proprio passare in qualche modo del codice al metodo `run()`
- Il metodo può e fa tutto ciò che desidera con il codice e in questo caso esegue giusto il codice passato nel metodo `execute()`
- Quindi ci sono un mucchio di istruzioni solo per specificare come passare un blocco di codice al metodo `run()`
- Vediamo ora come fare la stessa cosa con le espressioni Lambda in Java 8

Lambda expression

Il termine «Lambda expression» viene apparentemente dalla matematica dove la lettera greca lambda è storicamente associata a situazioni analoghe, passando una funzione a una funzione.

Lambda expression

- Con la lambda expression si ottiene lo stesso di quello ottenuto utilizzando la vecchia sintassi di Java con la classe anonima
- E' realmente importante da tenere a mente che tutto questo è un modo per passare del codice al metodo run()
- Blocchi di codice, ovviamente, potrebbero avere valori di ritorno e accettare in input dei parametri
- Per passare un blocco di istruzioni come lambda expression deve essere racchiuse fra parentesi {} e pertanto le istruzioni vanno terminate con ;

```
runner.run( () -> {  
    System.out.println("Code passed in Lambda expression| ...");  
    System.out.println("Hello from execute ...");  
});
```

Functional Interfaces

- ❑ Le Lambda expression sono sempre associate con interface che hanno un solo metodo
- ❑ Java 8 non introduce alcuna nuova sintassi per specificare che il parametro di un metodo si aspetta un blocco di codice
- ❑ Si devono ancora usare le interface, come nelle versioni precedenti di Java
- ❑ La cosa nuova è nella terminologia, una interface con giusto **un solo metodo**, come *Executable* nell'esempio, è chiamata **functional interface**
- ❑ Altri esempi di functional interface sono *Comparable* o la *Runnable* di Java

Return Values in Lambda Expression

- ❑ Come è possibile ottenere un valore di ritorno dal blocco di codice che si vuole passare al metodo per l'esecuzione ?
- ❑ Nelle precedenti versioni di Java, con le classi anonime e anche in java 8, quello che è necessario fare è cambiare l'interface
- ❑ Modifichiamo l'interface in modo che restituisca un int

```
interface Executable {  
    int execute();  
}
```

Return Values in Lambda Expression

- ❑ Modifichiamo anche il metodo run() per stampare il valore di ritorno

```
class Runner {  
    public void run(Executable e) {  
        System.out.println("Executing code block ...");  
        int value = e.execute();  
        System.out.println("Return value is: " + value);  
    }  
}
```

Return Values in Lambda Expression

- ❑ La classe anonima diventa, per supportare un valore di ritorno:

```
Runner runner = new Runner();

runner.run(new Executable() {

    public int execute() {
        System.out.println("Hello from execute ...");
        return 7;
    }

});
```

Return Values in Lambda Expression

- ❑ Con la Lambda expression è necessario solo inserire il valore di ritorno
- ❑ Non è più necessario specificare il valore di ritorno e in effetti non c'è nessun modo per fare ciò
- ❑ Tutto quello che c'è da fare è letteralmente restituire un valore
- ❑ Java conosce il tipo di ritorno in quanto definito nell'interfaccia funzionale trattata dalla Lambda expression

```
runner.run( () -> {  
    System.out.println("Code passed in Lambda expression ...");  
    System.out.println("Hello from execute ...");  
    return 8;  
});
```

<terminated> LambdaDemo [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.e

```
Code passed in Lambda expression ...  
Hello from execute ...  
Return value is: 8
```

Return Values in Lambda Expression

- ❑ Talvolta si vuole restituire un literal value o, più frequentemente, il risultato di un metodo che ritorna un valore
- ❑ Nel caso di una singola espressione che potrebbe essere un literal value, non è necessario specificare nemmeno return e parentesi graffe
- ❑ Le due Lambda expression sono equivalenti

```
runner.run( () -> {return 8;});
```

```
runner.run( () -> 8);
```

<terminated> LambdaDemo [Java Application] C:\Program F

Executing code block ...

Return value is: 8

Lambda Expression Parameters

- ❑ Facciamo in modo che la nostra interface *Executable* accetti un *parametro*, per esempio un *int a*
- ❑ Il metodo effettivo che richiama il blocco di codice deve fornire il parametro
- ❑ Per supportare il parametro si dovrà modificare la vecchia classe anonima oltre che l'espressione lambda
- ❑ Vogliamo solo che il blocco di codice prenda in input il parametro *int a*, ci aggiunga 7 e lo ritorni al chiamante

Lambda Expression Parameters

- ❑ Sotto le modifiche per richiamare il codice con il nuovo parametro

```
interface Executable {  
    int execute(int a);  
}
```

```
class Runner {  
    public void run(Executable e) {  
        System.out.println("Executing code block ...");  
        int value = e.execute(7);  
        System.out.println("Return value is: " + value);  
    }  
}
```

Lambda Expression Parameters

- ❑ Sotto le modifiche per l'esecuzione con la classe anonima

```
runner.run(new Executable() {  
  
    public int execute(int a) {  
        System.out.println("Hello from execute ...");  
        return a + 7;  
    }  
});
```

Lambda Expression Parameters

- ❑ Sotto le modifiche per l'esecuzione con la lambda expression
- ❑ La lambda expression **necessita solo di dare un nome al parametro**, mentre il tipo viene generalmente **inferito** dall'interface

```
runner.run( (a) -> {  
    System.out.println("Code passed in Lambda expression ..."  
    System.out.println("Hello from execute ...");  
    return a + 7;  
});
```

<terminated> LambdaDemo [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.e

Code passed in Lambda expression ...

Hello from execute ...

Return value is: 14

Lambda Expression Parameters

- ❑ Sotto le modifiche per l'esecuzione con la classe anonima

```
runner.run(new Executable() {  
  
    public int execute(int a) {  
        System.out.println("Code passed Anonymous class ...");  
        System.out.println("Hello from execute ...");  
        return a + 7;  
    }  
});
```

<terminated> LambdaDemo [Java Application] C:\Program Files\Java\jdk-17

```
Code passed Anonymous class ...  
Hello from execute ...  
Return value is: 14
```

Lambda Expression Parameters

- ❑ Le lambda expression sono in realtà semplici, basta ricordare che stiamo passando un blocco di codice, che potrebbe ricevere dei parametri e restituire un valore di ritorno
- ❑ Naturalmente possiamo specificare e passare al blocco di codice tutti i parametri che si vuole
- ❑ Da notare che, passando dei parametri multipli, è **necessario** indicarli fra parentesi tonde
- ❑ Come sempre, qualsiasi modifica alla espressione lambda (e alla classe anonima), è preceduta dalla modifica della functional interface

Lambda Expression Parameters

- ❑ Sotto l'interfaccia funzionale e la classe Runner modificata

```
interface Executable {  
    int execute(int a, int b);  
}
```

```
class Runner {  
    public void run(Executable e) {  
        System.out.println("Executing code block ...");  
        int value = e.execute(10, 7);  
        System.out.println("Return value is: " + value);  
    }  
}
```

Lambda Expression Parameters

- ❑ Sotto la classe anonima modificata

```
// Anonymous class|

runner.run(new Executable() {

    public int execute(int a, int b) {
        System.out.println("Code passed Anonymous class ...");
        System.out.println("Hello from execute ...");
        return a + b;
    }
});
```


Lambda Expression Parameters

- ❑ Sotto l'espressione Lambda modificate

```
// Lambda expression

runner.run( (a, b) -> {
    System.out.println("Code passed in Lambda expression ...")
    System.out.println("Hello from execute ...");
    return a + b;
});
```

<terminated> LambdaDemo [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.e

```
Executing code block ...
Code passed in Lambda expression ...
Hello from execute ...
Return value is: 17
```

Lambda Parameters Ambiguity

- ❑ Talvolta necessita specificare il tipo dei parametri nella lambda expression
- ❑ Creiamo una nuova interface StringExecutable, modifichiamo Executable per avere di nuovo solo un parametro e creiamo un nuovo metodo in Runner

```
interface Executable {  
    int execute(int a);  
}  
  
interface StringExecutable {  
    int execute(String a);  
}
```

Lambda Parameters Ambiguity

- ❑ Sotto la classe Runner con il nuovo metodo Run

```
class Runner {  
  
    public void run(Executable e) {  
        System.out.println("Executing code block ...");  
        int value = e.execute(7);  
        System.out.println("Return value is: " + value);  
    }  
  
    public void run(StringExecutable e) {  
        System.out.println("Executing code block ...");  
        int value = e.execute("Execute with string..");  
        System.out.println("Return value is: " + value);  
    }  
}
```

Lambda Parameters Ambiguity

- ❑ La classe anonima non deve essere modificata, avendo contezza dell'interfaccia da utilizzare

```
// Anonymous class

runner.run(new Executable() {

    public int execute(int a) {
        System.out.println("Code passed Anonymous class ...");
        System.out.println("Hello from execute ...");
        return a;
    }

});
```

Lambda Parameters Ambiguity

- ❑ La lambda expression dà errore in quanto ci sono due metodi in Runner con nome *run* e tipo parametro differente, uno *int* e l'altro *String*

```
68      // Lambda expression
69
70      runner.run( (a) -> {
71          System.out.println("Code passed in Lambda expression ...")
72          System.out.println("Hello from execute ...");
73          return a;
74      });
75  }
```

Lambda Parameters Ambiguity

- ❑ La lambda expression deve essere modificata per risolvere l'ambiguità dichiarando il tipo

```
// Lambda expression

runner.run( (int a) -> {
    System.out.println("Code passed in Lambda expression ...");
    System.out.println("Hello from execute ...");
    return a;
});
```

```
<terminated> LambdaDemo [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.e
Executing code block ...
Code passed in Lambda expression ...
Hello from execute ...
Return value is: 7
```

Lambda Parameters Ambiguity

- ❑ Proviamo ora a modificare l'interface Executable indicando un secondo parametro dello stesso tipo
- ❑ Vogliamo verificare se ci sono ambiguità nella lambda expression

```
interface Executable {  
    int execute(int a, int b);  
}
```

Lambda Parameters Ambiguity

- ❑ Modifichiamo la classe Runner con i due parametri di run()

```
class Runner {  
  
    public void run(Executable e) {  
        System.out.println("Executing code block ...");  
        int value = e.execute(7, 10);  
        System.out.println("Return value is: " + value);  
    }  
  
    public void run(StringExecutable e) {  
        System.out.println("Executing code block ...");  
        int value = e.execute("Execute with string..");  
        System.out.println("Return value is: " + value);  
    }  
}
```


Lambda Parameters Ambiguity

- ❑ Non ci sono ambiguità nella lambda expression in quanto i due metodi `run()` hanno firme diverse e differente numero di parametri

```
runner.run( (a, b) -> {  
    System.out.println("Code passed in Lambda expression ...");  
    System.out.println("Hello from execute ...");  
    return a + b;  
});
```

<terminated> LambdaDemo [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.e

Executing code block ...

Code passed in Lambda expression ...

Hello from execute ...

Return value is: 17

Lambda Expression And Scope: Effectively Final

- ❑ Cosa succede se nel blocco di codice si vogliono usare variabili locali del metodo chiamante o di istanza?
- ❑ In JDK7, per usare una variabile locale in una anonymous class, doveva essere dichiarata **final** ma, in JDK8 final non era più necessario, assegnando alla stessa un valore alla dichiarazione, senza successive modifiche
- ❑ Quindi possiamo usare variabili locali nella Lambda expression purchè siano **effettivamente final**
- ❑ Creiamo la variabile **int c** e verifichiamo la sua visibilità nella classe anonima e nella lambda expression
- ❑ In ogni caso sia da anonymous class sia da lambda expression sono visibili le variabili di istanza

Lambda Expression And Scope: Effectively Final

```
67 public class LambdaDemo {
68     public static void main(String[] args) {
69         int c = 9;
70
71         // c può essere usata in anonymous class e Lambda expression
72         // a condizione di non cambiare il valore dopo l'assegnazione
73         // Si dice che c è "effectively final"
74
75         Runner runner = new Runner();
76
77         // Anonymous class
78
79         runner.run(new Executable() {
80
81             public int execute(int a, int b) {
82                 System.out.println("Code passed Anonymous class ...");
83                 System.out.println("Hello from execute ...");
84                 return a + b + c;
85             }
86     });
```

Lambda Expression And Scope: Effectively Final

```
87
88     System.out.println("=====");
89
90     // Lambda expression
91
92     runner.run( (a, b) -> {
93         System.out.println("Code passed in Lambda expression ...");
94         System.out.println("Hello from execute ..."); |
95         return a + b + c;
96     });
97 }
98
99
100 }
```

Lambda Expression And Scope: Effectively Final

```
class Runner {  
  
    public void run(Executable e) {  
        System.out.println("Executing code block ...");  
        int value = e.execute(7, 10);  
        System.out.println("Return value is: " + value);  
    }  
  
    public void run(StringExecutable e) {  
        System.out.println("Executing code block ...");  
        int value = e.execute("Execute with string..");  
        System.out.println("Return value is: " + value);  
    }  
}
```

Lambda Expression And Scope: Effectively Final

- ❑ Sia l'anonymous class che la Lambda expression *vedono* la variabile locale *c* definita nel metodo chiamante e il risultato $a + b + c = 7 + 10 + 9 = 26$

```
<terminated> LambdaDemo [Java Application] C:\Program Files\Java\jdk-17\bin\javaw.exe
```

```
Executing code block ...
```

```
Code passed Anonymous class ...
```

```
Hello from execute ...
```

```
Return value is: 26
```

```
=====
```

```
Executing code block ...
```

```
Code passed in Lambda expression ...
```

```
Hello from execute ...
```

```
Return value is: 26
```

Lambda Expression And Scope: Effectively Final

- ❑ La variabile `int d=24` nel metodo della classe anonima ridefinisce quella esterna `d=11` definita nel metodo che la ospita

```
int d = 11;

Runner runner = new Runner();

// Anonymous class

runner.run(new Executable() {

    public int execute(int a, int b) {
        int d = 24;    // Questa è una d completamente nuova
        System.out.println("Code passed Anonymous class ...");
        System.out.println("Hello from execute ...");
        return a + b + c;
    }
});
```

24/04/2023

Lambda Expression And Scope: Effectively Final

- ❑ Dentro la lambda expression **non è invece possibile** ridefinire la variabile **d**.
- ❑ Questo significa che lo scope (campo di validità) dentro una lambda expression è lo stesso del metodo che la ospita

```
// Lambda expression

runner.run( (a, b) -> {
//      int d = 7;    // Syntax error
    System.out.println("Local variable d = " + d);
    System.out.println("Code passed in Lambda expression ...");
    System.out.println("Hello from execute ...");
    return a + b + c;
});
}
```


Le Lambda Expression Sono Objects!

- ❑ Nonostante le lambda expression rappresentino un modo molto più sintetico per passare del codice attraverso un oggetto, parametro in un metodo, non rappresentano di fatto una vera innovazione
- ❑ Dobbiamo ancora passare dalle interface per queste operazioni che in ogni caso si potevano fare anche senza lambda expression
- ❑ In ogni caso le lambda expression sono molto comode per passare codi a un thread o a un button

Le Lambda Expression Sono Objects!

- ❑ E' possibile memorizzare la lambda expression in una variabile

```
// Lambda expression in variable  
  
Executable ex = (a, b) -> {  
    return a+b;  
};  
runner.run(ex);
```

- ❑ O in un generico oggetto

```
// Lambda expression in un oggetto (da castare)  
  
Object codeBlock = (Executable)(a, b) -> {  
    return a+b;  
};  
runner.run((Executable) codeBlock);
```

Lambda Method Reference

- ❑ L'operatore `::` è conosciuto come **'method reference'** in Java 8
- ❑ I method references sono espressioni che hanno lo stesso trattamento come le lambda expression ma, invece di fornire un blocco di codice, riferiscono a un metodo esistente per nome
- ❑ L'utilizzo di method references rende il codice più conciso
- ❑ In sintesi l'operatore `::` è uno **'shortend'** per chiamare uno specifico metodo per nome

Lambda Method Reference

- ❑ Per esempio, per stampare tutti gli elementi di una List , usiamo il metodo [forEach\(\)](#)

```
list.stream.forEach( s-> System.out.println(s));
```

- ❑ In modo equivalente per incrementare la leggibilità rimpiazziamo la lambda expression con l'operatore Method Reference

```
list.stream.forEach( System.out::println);
```

Lambda Method Reference

- ❑ Si può usare il method reference **::** per riferirsi a:
- ❑ Un metodo **statico** di un particolare oggetto (`ClassName::methodName`)
 - **`System.out::println`**
- ❑ Un metodo di **istanza** (`instanceRef::methodName`)
 - **`Person::getAge`**
- ❑ Un **costruttore di classe** (`ClassName::new`)
 - **`ArrayList::new`**
- ❑ Un **costruttore di array** (`Type Name [] ::new`)
 - **`String [] ::new`**
- ❑ Un **super** metodo di un particolare oggetto
 - **`super::methodName`**

Lambda Method Reference

```
listOfNumbers.stream().sorted().forEach(number -> {  
    System.out.println(number);  
});
```

Si può riscrivere come

```
listOfNumbers.stream().sorted().forEach(System.out::println);
```

Lambda Method Reference

```
(x, y) -> x.compareToIgnoreCase(y)
```

Si può riscrivere come

```
String::compareToIgnoreCase
```

Lambda Method Reference

```
(x, y) -> x.compareToIgnoreCase(y)
```

Si può riscrivere come

```
String::compareToIgnoreCase
```

```
(x -> this.equals(x))
```

Si può riscrivere come

```
this::equals
```