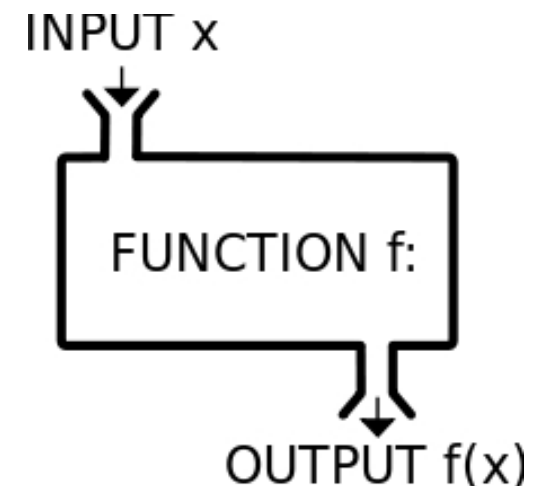


Programmazione Funzionale in Java

Programmazione funzionale

- E' uno **stile di programmazione** che

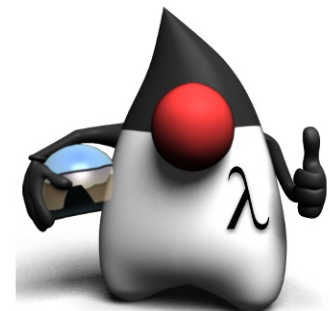
- Favorisce l'espressione della computazione come l'applicazione di una o più funzioni sui dati
- Funzioni nel **senso matematico**
 - No side-effects!
 - Il valore di una funzione dipende solo dagli input



- Origina da **lambda calculus**, un sistema formale inventato negli anni 30 da Alonzo Church per formalizzare il calcolo delle funzioni
 - Alternativa a macchina di Turing per investigare problemi di decidibilità

Linguaggi “funzionali”

- Così come diversi linguaggi supportano (o forzano) il paradigma OO a gradi diversi...
- ...così diversi linguaggi favoriscono, o meno, lo stile funzionale!
- Linguaggi funzionali “puri”: Erlang, Lisp, Haskell, Clojure, ...
- Linguaggi che supportano **alcune** caratteristiche funzionali: PHP, Perl, Python, ...
 - ...e **Java** dalla versione 8!



Esempio: stile imperativo

- Stampare tutti gli elementi di una lista di stringhe
- Fino ad adesso avremmo scritto:

```
List<String> friends =  
    Arrays.asList( "Brian", "Nate", "Neal", "Sara", "Scott" );  
  
for(int i = 0; i < friends.size(); i++) {  
    System.out.println(friends.get(i));  
}
```

- Oppure, se siamo bravi:

```
for(String name : friends) {  
    System.out.println(name);  
}
```

Esempio: stile funzionale

- Usiamo il supporto di Java 8 per lo stile funzionale:

```
List<String> friends =  
    Arrays.asList( "Brian", "Nate", "Neal", "Sara", "Scott" );
```

Applica la **funzione** data come parametro
a ciascun elemento della collezione

```
friends.forEach(String name -> System.out.println(name));
```

Espressione lambda: descrive la computazione da
applicare a ciascun elemento

Confronto

```
for(String name : friends) {  
    System.out.println(name);  
}
```

- Stile imperativo:
 - Descrive **come** raggiungere un determinato scopo

```
friends.forEach(  
    String name ->  
    System.out.println(name));
```

- Stile funzionale:
 - Descrive **quale** scopo vogliamo raggiungere

- **What** vs. **how**, programmazione **dichiarativa**
- L'output sarà comunque identico...

...quindi, perchè disturbarsi?

- Applicare (correttamente) lo stile funzionale permette di:
 - Ridurre il **gap** tra la specifica e la realizzazione
 - ...grazie alle caratteristiche dichiarative
 - Nascondere gli aspetti di **basso livello**
 - Iteratori, conversioni di tipo, ...
 - Assegnamenti e ri-assegnamenti di variabili
 - Facilitare il **test** e **manutenzione** del codice
 - Sfruttare il **parallelismo**
 - N-core CPUs a disposizione ormai ovunque...
 - Implementare facilmente **design pattern**
 - Quindi, rendere il codice più **conciso, elegante, efficiente, e meno prone** ad errori

Un altro esempio...

- Computiamo la somma di tutti i prezzi maggiori di 20 e scontati del 10%
- Stile imperativo:

```
List<BigDecimal> prices =  
    Arrays.asList(new BigDecimal("30"), new BigDecimal("20"),  
        new BigDecimal("15"), new BigDecimal("45"));  
  
BigDecimal total = BigDecimal.ZERO;  
for(BigDecimal price : prices) {  
    if(price.compareTo(BigDecimal.valueOf(20)) > 0)  
        total = total + price.multiply(BigDecimal.valueOf(0.9));  
}
```

Provate a leggere ad alta voce il codice come se foste il calcolatore che deve eseguirlo...

...ora applicando lo stile funzionale

```
List<BigDecimal> prices =  
    Arrays.asList(new BigDecimal("30"), new BigDecimal("20"),  
        new BigDecimal("15"), new BigDecimal("45"));  
  
BigDecimal total =  
    prices.stream()  
        .filter(price -> price.compareTo(BigDecimal.valueOf(20))>0)  
        .map(price -> price.multiply(BigDecimal.valueOf(0.9)))  
        .reduce(BigDecimal.ZERO, (sum, x) -> sum + x);
```

Filtra gli elementi della collezione
in base ad un **predicato**...

...e li **somma** tra di loro
partendo da zero

...**applica** una funzione a tutti gli
elementi che "sopravvivono" al filtro...

Ora provate di nuovo a leggere il codice
ad alta voce come foste il calcolatore!

Concetti fondamentali in Java 8

- Le espressioni lambda diventano “**cittadine di prima classe**” nel linguaggio
- La tipizzazione forte di Java **rimane**
 - Viene introdotto un nuovo tipo Function e varianti come Predicate, Comparator, ...
 - Questi tipi (**interfacce funzionali**) sono parametrizzate con i tipi concreti in base ai casi specifici
 - Le API che supportano lo stile funzionale effettuano il type-check in base alla **conformità rispetto al tipo Function e varianti**

Sintassi espressioni lambda (1/2)

- Espressione lambda con un **singolo** input di tipo String
`friends.map(String name -> name.toUpperCase()));`
- L'input è **immutable**: assicura la mancanza di side-effect
`friends.map(final String name -> name.toUpperCase()));`
- Il compilatore può **inferire** il tipo del dato di input (per tutti o per nessuno dei parametri)
`friends.map(name -> name.toUpperCase()));`
- Questo è **illegale**:
`friends.map(final name -> name.toUpperCase()));`
 - Se chiediamo aiuto al compilatore per inferire il tipo, dobbiamo essere noi ad assicurare la mancanza di side-effects!

Sintassi espressioni lambda (2/2)

- Espressione lambda con **più parametri** in input

```
friends.doSomething((name1,name2) -> name1.concat(name2));
```

- Espressione lambda con più di uno statement

```
friends.doSomething(final String name ->
    {
        if (name.equals("Boo")
            return "Foo";
        else
            return name;
    });
```



Necessario un **return** esplicito!

Stile funzionale: dove?

- Lo stile funzionale è applicabile ovunque le interfacce funzionali (Function e varianti) siano parte dei prototipi di metodi
- Risultano particolarmente efficaci in ambiti specifici
 - Manipolazione delle **Collection**
 - Processing di **String(s)**
 - **Multi-threading**
 - Gestione delle **risorse** (file, sockets, ...)
- I concetti che vedremo applicati alle Collection(s) rimangono gli stessi quando applicati in altri ambiti

Alcuni interfacce funzionali

- **Predicate<T>**: una funzione da T a Boolean
- **Function<T, R>**: una funzione con parametri di tipo T e R, ritorna un tipo R
- **BiFunction<T, U, R>**: una funzione con parametri di tipo T ed U, ritorna un tipo R
- Si consiglia di usare queste interfacce quando si vogliono passare funzioni come parametro

Streams Java 8 vs Iteratori

- Gli iteratori prevedono una specifica strategia di visita della collezione, impedendo un'efficiente esecuzione concorrente
- Gli **streams** sono un'alternativa che fornisce più libertà alla JVM per migliorare l'efficienza della computazione
 - Parallelismo...
- Gli streams possono essere definiti da collezioni, arrays, generatori o iteratori
 - ...da non confondere con InputStreams, ...

Esempio: trasformare una Lista

Rendiamo i dati
immutabili !

```
final List<String> friends =  
    Arrays.asList("Brian", "Nate", "Neal", "Sara", "Scott");  
List<String> uppercaseNames = new ArrayList<String>();
```

```
// Versione imperativa  
for(String name : friends) {  
    uppercaseNames.add(name.toUpperCase());  
}
```

```
// Versione funzionale  
uppercaseNames = friends.stream()  
    .map(name -> name.toUpperCase())  
    .collect(Collectors.toList());
```

Converte la lista in
un “flusso” di
elementi, sul quale
viene applicata una
espressione lambda

collect è un particolare tipo di
reduce che converte da uno
stream ad una collection

Esempio: trasformare una Lista

- Possiamo anche convertire da un tipo ad un altro...

```
List<String> friends =  
    Arrays.asList( "Brian", "Nate", "Neal", "Sara", "Scott");
```

```
friends.stream()  
    .map(name -> name.toUpperCase())  
    .forEach(name -> System.out.print(name));
```

Brian Nate Neal Sara Scott

```
friends.stream()  
    .map(name -> name.length())  
    .forEach(count -> System.out.print(count));
```

5 4 4 4 5

Genera una nuova Collection dove ogni elemento è la lunghezza dell'elemento corrispondente in friends

Stream e Collection

- Uno stream non memorizza i suoi elementi
 - ..che possono essere memorizzati in una collezione o generati on demand
- Le operazioni su stream:
 - Non possono modificare la loro sorgente
 - Possono restituire nuovi stream che contengono i risultati
 - Sono “lazy” quando possibile, cioè sono eseguiti solo quando il loro risultato è necessario
 - Es. se cerchiamo le prime cinque parole che iniziano per N, allora il filtro si fermerà dopo il quinto match...

Trovare un elemento in una lista

Nuovo tipo Optional
introdotta in Java 8

```
final Optional<String> foundName =  
    names.stream()  
        .filter(name -> name.startsWith("N"))  
        .findFirst();  
  
System.out.println(A name starting with  
    + startingLetter + ": "  
    + foundName.orElse("No name found"));  
  
foundName.ifPresent(  
    name -> System.out.println("Hello " + name));
```

Semplifica la gestione dei valori **null**
nelle Collection, ed è più efficace con lo
stile funzionale!

Ridurre una lista ad un elemento

- Come troviamo il nome più lungo in una lista?

```
final Optional<String> aLongName =  
    friends.stream()  
        .reduce((name1, name2) ->  
            name1.length() >= name2.length() ? name1 : name2);
```

- Utilizza un'ulteriore variante di Function, detta BinaryOperator
- Esiste un'ulteriore versione di reduce con un valore iniziale di default

```
final Optional<String> aLongName =  
    friends.stream()  
        .reduce("Steve", (name1, name2) ->  
            name1.length() >= name2.length() ? name1 : name2);
```

Per ciascun elemento, reduce esegue una espressione lambda dove name1 è il risultato precedente o il primo elemento, name2 è l'elemento successivo

Riferimenti a metodi

- Le espressioni lambda sono **funzioni anonime**
- Lo stile funzionale si applica anche quando le funzioni sono **named** e **pre-esitenti**
 - `object::instanceMethod`
 - `Class::staticMethod`
 - `Class::instanceMethod`
- Attenzione ai side-effects!
- Sono quindi equivalenti:

```
uppercaseNames = friends.stream()  
    .map(name -> name.toUpperCase());
```

```
uppercaseNames = friends.stream()  
    .map(String::toUpperCase);
```

La funzione passata è
`toUpperCase()` della
classe `String`

Function references

- Dovendo ri-usare la stessa espressione lambda in più posti, possiamo definirla una sola volta
 - Evita duplicazioni
 - Semplifica le modifiche

```
final List<String> friends =  
    Arrays.asList("Brian", "Nate", "Neal", "Sara", "Scott");  
final List<String> colleagues =  
    Arrays.asList("Luke", "Peter", "Paul");  
  
final Predicate<String> startsWithN =  
    name -> name.startsWith("N");  
  
friends.stream()  
    .filter(startsWithN)  
    .forEach(name -> System.out.print(name));  
  
colleagues.stream()  
    .filter(startsWithN)  
    .forEach(name -> System.out.print(name));
```

L'espressione
lambda, con
input una String,
diventa ri-usabile!

Possiamo fare di più...

- Se non è necessariamente “N” l’iniziale che stiamo cercando?
- Creiamo un metodo che **ritorna una espressione lambda** parametrizzata
 - Le Function (o Predicate) sono anch’essi tipi di dato: possono essere il tipo di ritorno di metodi!

```
public static Predicate<String>  
  checkIfStartsWith (final String letter) {  
    return name -> name.startsWith(letter);  
  }
```

A cosa corrisponderà
letter???

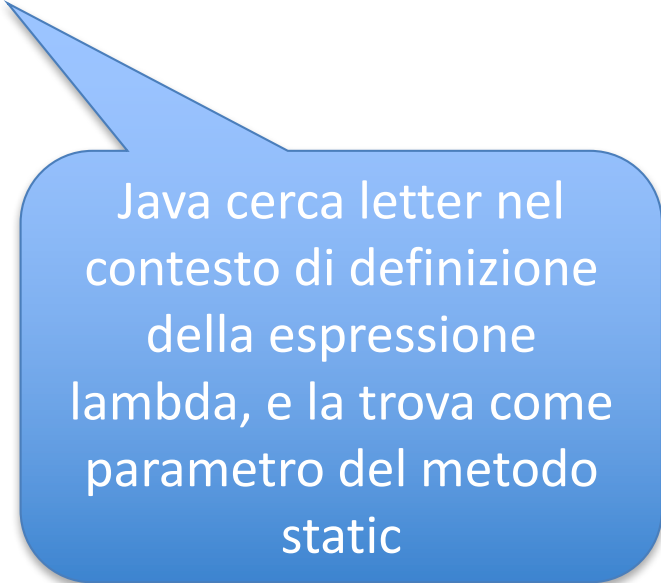
Non è un parametro!
L’espressione lambda viene
ritornata come fosse un dato,
customizzata rispetto a letter

Lexical scoping e closures

```
public static Predicate<String>
    checkIfStartsWith (final String letter) {
        return name -> name.startsWith(letter);
    }
```

```
friends.stream()
    .filter(checkIfStartsWith("N"))...
friends.stream()
    .filter(checkIfStartsWith("B"))...
```

- **Lexical scoping** permette di individuare staticamente le variabili di riferimento
- Si dice che la espressione lambda “closes over” lo scope della sua definizione
 - Viene quindi detta **closure**



Java cerca letter nel contesto di definizione della espressione lambda, e la trova come parametro del metodo static

Stile funzionale in Java

- Rispetto a tanti altri linguaggi, il supporto allo stile funzionale in Java rimane “regolato”
 - Il tipo Function e varianti assicura il type-checking
 - Le espressioni lambda sostanzialmente operano da “stand-in” dove classi anonime o interfacce con un singolo metodo svolgerebbero lo stesso compito (in maniera più verbosa)
 - Tutte le interfacce con un singolo metodo sono anche Functional in Java 8, e quindi possono essere utilizzate in maniera intercambiabile

```
class LengthComparator implements Comparator<String> {  
    public int compare(String first, String second) {  
        return Integer.compare(first.length(), second.length());  
    }  
}
```

```
Arrays.sort(strings, new LengthComparator());
```

...diventa invece...

```
Arrays.sort(strings, (first,second) ->  
    Integer.compare(first.length(), second.length()));
```

Parallelismo

- Tutte le chiamate a `stream()` per le collezioni possono essere convertite in `parallelStream()`
- La JVM si occupa, in maniera trasparente, di parallelizzare le operazioni ovunque sia possibile
- **Attenzione:**
 - Tutte le espressioni lambda ed equivalenti dovranno lavorare su **dati immutabili**
 - Non vi sarebbe, altrimenti, nessuna garanzia sulla correttezza del risultato finale, poichè l'accesso ai dati sarebbe **non thread-safe**

Sommario

Stile OO	Stile funzionale
I dati e le operazioni che li manipolano sono strettamente accoppiati (nella stessa classe)	I dati e le operazioni (funzioni) che li manipolano sono disaccoppiati e combinati in maniera ortogonale
Le astrazioni del linguaggio mirano ad aumentare l'incapsulamento	Le astrazioni del linguaggio mirano a fornire le modalità con cui combinare le funzioni
Stateful: il risultato della computazione è determinato tipicamente dallo stato dell'oggetto	Stateless: il risultato di una funzione dipende solo dagli input e, in assenza di side-effects, da null'altro