

26-05-2023

Java ➡

Linguaggio Imperativo, permette la **PROGRAMMAZIONE FUNZIONALE** più ESPRESSIVA, CONCISA, PARALLELIZZATA

- Java è un linguaggio imperativo, permette (da marzo 2014 circa) la **programmazione funzionale** e ha introdotto diverse funzionalità.
- La programmazione funzionale è più **espressiva, concisa e facile da parallelizzare** rispetto alla programmazione ad oggetti

Classi anonime

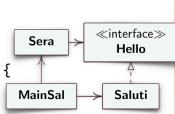
Da Java 1.1 è possibile avere una gerarchia di classi come segue:

```
public interface Hello {
    public void greetings(String s);
}

public class Sera {
    private Hello myh;
    public Sera(Hello h) {
        myh = h;
    }
    public void saluti() {
        myh.greetings("buonasera");
    }
}

public class Saluti implements Hello {
    public void greetings(String s) {
        System.out.println("Ciao, "+s);
    }
}

public class MainSal {
    public static void main(String[] args) {
        Sera sr = new Sera(new Saluti());
        sr.saluti();
    }
}
```

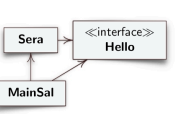


Questo codice può trasformarsi in un codice corretto che implementa classi anonime. Piuttosto che creare una classe che interfaccia che implementa l'interfaccia (come *Saluti*) si può implementare una **classe anonima**. Non si dà un nome alla classe e *non si mette il codice che implementa l'interfaccia in maniera separata*.

```
public interface Hello {
    public void greetings(String s);
}

public class Sera {
    private Hello myh;
    public Sera(Hello h) {
        myh = h;
    }
    public void saluti() {
        myh.greetings("buonasera");
    }
}

public class MainSal {
    public static void main(String[] args) {
        Sera sr = new Sera(new Hello() {
            public void greetings(String s) {
                System.out.println("Ciao, "+s);
            }
        });
        sr.saluti();
    }
}
```



La classe anonima che implementa Hello è scritta nel main ed è così scritta:

```
//istanzo classe anonima compatibile con Hello
Sera sr = new Sera(new Hello() {
    public void greetings(String s) {
        System.out.println("Ciao, "+s);
    }
})
```

Questa classe la uso solo in quel punto preciso e non può essere usata altrove.

Espressioni Lambda

E' una funzione anonima, quindi **senza nome**. La sintassi è così formata:

- **Parametri** da passare alla funzione (*a sinistra della freccia*)
- **Corpo della funzione**, cioè il codice che deve essere eseguito quando quella espressione Lambda va in esecuzione (*a destra della freccia*)

Esempio:

$(p, s) \rightarrow // \text{corpo funzione}$
parametri. corpo della funzione

- Si deve **CERCARE** DI EVITARE DI METTERE TROPPE ISTRUZIONI.
- Il tipo non viene SPECIFICATO, ci sono ma sono SOTTINTESI. (possono anche essere specificati)
- I parametri in ingresso sono **parametri FORMALI**.

```
s -> s=s+1
s -> System.out.println("Ciao " + s);
(p,s) -> //corpo funzione
(x,y) -> x+y;

//se non ci sono parametri in ingresso, allora:
() -> System.out.println("Ciao");

//in caso di insieme di istruzioni a destra della freccia, allora:
() -> {
    System.out.println("Ciao");
    System.out.println("mondo");
}
```

- Solitamente si deve **cercare di evitare di mettere troppe istruzioni** all'interno di una funzione anonima.
- Il **TIPO** non viene specificato fra i parametri e può avere senso solo se si trattano determinati tipi per il corpo della funzione
 - **I tipi ci sono ma sono SOTTINTESI**
 - Potrebbero anche essere esplicitati perchè ci sono informazioni derivate dal contesto dove si inserisce l'espressione lambda che definisce i tipi.

I **PARAMETRI IN INGRESSO** sono **parametri FORMALI** (e non attuali), per cui non hanno niente a che fare con le variabili scritte prima della dichiarazione della funzione anonima stessa.

Implementazione interfaccia

Quando si deve istanziare un'interfaccia si usava:

```
Sera sr = new Sera(new Hello() {
    public void greetings(String s) {
        System.out.println("Ciao, " + s);
    }
});
sr.saluti();
```

Questa funzione nascosta permette più espressività.

Interfacce con un solo metodo sono dette

INTERFACCIE FUNZIONALI.

↳ Possibile implementarle con funzioni LAMBDA.

Usando la funzione anonima diventa:

```
Sera sr = new Sera(s2 -> System.out.println("Ciao, " + s2));
sr.saluti();
```

• Si eliminano i frammenti di codice ripetitivi, Ripetere codice è inutile e viene detto BOILERPLATE

La funzione Anonima in questo caso è stata possibile da usare perchè la classe ha un solo metodo

SENZA dover usare le classi Anonime.

- Si eliminano dei frammenti di codice ridondanti, perchè già si capivano dal contesto (*l'implementazione dell'interfaccia conteneva solo un metodo e quindi non c'è bisogno di ripeterlo*)
- **Ripetere codice inutile viene detto *BOILERPLATE***
- Questa tecnica si è potuta usare perchè l'interfaccia ha **SOLO UN METODO** altrimenti si sarebbero dovute usare le *classi anonime* viste qualche riga prima.
- `greetings(String s)` prende un tipo `String`. `s2` sarà solo di tipo `String` e quindi viene determinato da questo contesto.
 - Di conseguenza se si passa un parametro non di tipo `String`, allora ci sarà un errore, ovviamente

Le interfacce che hanno un solo metodo vengono dette **INTERFACCE FUNZIONALI** perchè possono essere implementate utilizzando proprio le espressioni lambda anonime

Esempio: ricerca valori in una lista

```
public class Trova {
    private List<String> listaNomi = Arrays.asList("Nobita", "Nobi",
        "Suneo", "Honekawa", "Shizuka", "Minamoto", "Takeshi", "Gouda");

    // in stile imperativo
    public void trovaImper() {
        boolean trovato = false;
        for (String nome : listaNomi)
            if (nome.equals("Nobi")) {
                trovato = true;
                break;
            }
        if (trovato) System.out.println("Nobi trovato");
        else System.out.println("Nobi non trovato");
    }

    // in stile dichiarativo
    public void trovaDichiar() {
        if (listaNomi.contains("Nobi")) System.out.println("Nobi trovato");
        else System.out.println("Nobi non trovato");
    }
}
```

Con `contains()` si ha il confronto con ogni elemento della lista. Non si vede il `for` ma c'è ed è dentro la libreria JAVA. (nascondendo i dettagli)

5

Prof. Tramontana - Maggio 2022

Per controllare *se esiste un elemento all'interno di una lista* si potrebbe usare

- `trovaImper()` che è un metodo *imperativo* che scorre la lista e scorre la lista con un `for` avanzato e, per ogni elemento, viene fatto un confronto.
 - In questo caso si devono scrivere molte cose e, per capire bene il significato, bisogna leggere molto testo.
- **Potrei usare un metodo DICHIARATIVO `trovaDichiar()` che usa `contains()`.** Si sposta la funzionalità che si voleva avere dentro la libreria Java.
 - Il ciclo `for` esiste ugualmente, sì, ma dentro le librerie che *nasconde i dettagli* e quindi il programmatore non deve esplicitare il ciclo `for`

Conforme allo stile dichiarativo e aggiunge funzione di ordine superiore (permette di passare un intero pezzo di codice a `contains()`). Permette di implementare funzioni che lavorano su altre funzioni. Si possono passare funzioni oltre dati al metodo

Stile funzionale

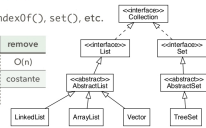
- E' conforme allo stile dichiarativo e aggiunge funzione di ordine superiore (in `contains()` si può passare solo il parametro da contenere e non si può passare un pezzo di codice).
- Permette di implementare funzioni che lavorano su altre funzioni che sono passate come parametri in ingresso.
- Quando si implementa un metodo, se c'è bisogno, **si può passare una funzione** (oltre a un singolo dato) a quel metodo.

- I metodi che prendono **in ingresso funzioni**, si parla di funzioni di **ordine più alto**

- Le librerie di Java hanno tante interfacce e classi collection. Collection è un'interfaccia che definisce i metodi add(), remove(), size(), contains(), containsAll(), etc.
- List definisce i metodi get(), indexOf(), set(), etc.

	get	add	contains	remove
ArrayList	costante	costante	O(n)	O(n)
LinkedList	O(n)	costante	O(n)	costante
TreeSet		O(log n)	O(log n)	

- ArrayList è un array espandibile: cresce del 50% quando non vi è spazio. Gli elementi sono contigui, quindi l'accesso al generico elemento è veloce
- LinkedList è una lista, ogni elemento ha i riferimenti al successivo e al prec
- Vector è simile ad ArrayList, ma è synchronized



è un Metodo di Collection implementato in Collection. Rappresenta un'eccezione per il tipo interfaccia.

il metodo **stream()** dà accesso alla prog funzionale, stream **restituisce un tipo Stream** che mette a disposizione un metodo che prende in ingresso **FUNZIONI LAMBDA**

Metodi di default

Il metodo **stream()**, a partire da una lista, permette di **accedere alla programmazione funzionale**. **stream()** restituisce un tipo **Stream** che mette a disposizione un metodo che **prende in ingresso funzioni lambda** (filter() ecc..) e, di base, **stream()** *non fa altro*.

- Serve un modo per trasformare una lista in un qualcosa che si adatti alla prog. funzionale. Questo lavoro lo fa proprio **stream()**: Stream
- stream()** è un metodo di **Collection** ed è implementato in **Collection** stesso e rappresenta un'eccezione per il tipo interfaccia (come regola di base) e si parla appunto di **METODI DI DEFAULT**
- i metodi di default non possono agire sullo stato perchè l'interfaccia non ha uno stato

Si può eseguire la conta di un certo numero di elementi in una lista in maniera **FUNZIONALE**:

```
List<String> nomi = List.of("Nobita", "Nobi", "Suneo");
long c = nomi.stream().filter(s -> s.equals("Nobi")).count();
```

③ Contiene solo gli elementi presenti nello stream di ingresso che soddisfanno **PREDICATE**

④ tipo String

⑤ in base alla condizione LAMBDA si scorre la lista, torna TRUE o FALSE se Parla di Predicate

⑥ Conto gli elementi della lista.

- filter(s -> s.equals("Nobi"))** seleziona alcuni elementi della lista sulla base della condizione della funzione lambda. Deve tornare true o false, quindi un **BOOLEANO** e quindi si parla di un predicato, detto **Predicate**
 - Se l'espressione lambda non ritorna un booleano, allora vi sarà un errore di compilazione
- s è di tipo String perchè si ha List<String> nomi;
- La funzione viene applicata a tutti gli elementi della lista
- All'uscita di **filter()** si ha un tipo **Stream** che contiene **solo gli elementi** presenti nello stream **in ingresso** e che **soddisfanno** la condizione il **Predicate**
- count()** conta gli elementi presenti nello Stream di uscita da **filter()**.
- Quindi **count()** e **filter()** sono metodi messi a disposizione dal tipo **Stream**.

Il pezzo di codice sopra è equivalente al seguente:

```
Stream<String> s1 = nomi.stream();
Stream<String> s2 = s1.filter(s -> s.equals("Nobi"));
long c = s2.count();
```

- s2 non contiene nulla
- Le operazioni terminali (**count()**) fanno finire **FORZATAMENTE** le operazioni intermedie (**filter()** , quindi **filter()** non si applica fino a quando non si conclude con un'operazione terminale)
- Dopo la **filter()** e quindi dopo aver selezionato gli elementi bisogna fare qualcos'altro, per esempio, contare o fare altro. Alcune operazioni sono **TERMINALI** (dette **eager**) e alcune **NON TERMINALI** o **INTERMEDIE** (dette **lazy**).

- Uno `Stream<T>` non viene riempito fino a quando non si hanno le operazioni terminali

OPERAZIONI INTERMEDIE: tornano un tipo Stream. // Una volta che è stata effettuata l'operazione terminale, lo Stream si chiude. OPERAZIONI TERMINALI: Non tornano un tipo Stream.

Le operazioni che *tornano uno `Stream<T>`* sono **operazioni intermedie**. Mentre le operazioni che *tornano un tipo NON `Stream<T>`*, allora si tratta **operazioni terminale**.

*Dopo aver **UTILIZZATO** lo Stream, quindi invocata un'operazione terminale, esso **SI CHIUDE**.
Per tale motivo uno Stream può essere usato **SOLO UNA VOLTA**

• Esempi con Filter

- Contare quanti elementi della lista `nomi` hanno lunghezza 5 caratteri
- Si noti che non si può usare il metodo `contains()` di `List` (che è invece utile per verificare se una lista contiene un certo elemento)

```
long c = nomi.stream()
    .filter(s -> s.length() == 5)
    .count();
```

- L'espressione lambda `s -> s.length() == 5` ha in ingresso `s`, ovvero un elemento dello stream; su `s` si invoca `length()` (metodo di `String` che restituisce la lunghezza di `s`), quindi si valuta se è pari a 5

- Contare quanti elementi della lista `nomi` sono stringhe vuote

```
long c = nomi.stream()
    .filter(s -> s.isEmpty())
    .count();
```

- `isEmpty()` è un metodo di `String` (non ha parametri di ingresso e restituisce un boolean). Tale metodo è passato a `filter()`, e sarà chiamato su ciascun elemento dello stream, quindi ciascun elemento dello stream è valutato da `isEmpty()`

Tipo Predicate<T>

Piuttosto che scrivere ogni volta il predicato dell'espressione Lambda, si potrebbe salvare in un'apposita variabile di tipo `Predicate<T>` per poi venire usato in un'altra espressione lambda. Chiaramente `Predicate` avrà esclusivamente **un valore booleano**.

è un'interfaccia FUNZIONALE: contiene il metodo test(Object obj)

```
Predicate<String> p = s -> s.equals("...");
```

```
str.filter(s -> s.equals("..."));
```

```
//equivale a
```

```
str.filter(p); //operazione non terminale
```

Predicate è un'interfaccia funzionale e quindi implementa **SOLO UNA FUNZIONE** che si chiama **`test(Object obj)`**

Metodo `reduce()` Applicato ad uno stream e fornisce in uscita un **UNICO risultato** che è **CALCOLATO SULL'INTERO STREAM()**.
Lo valuta interamente sulla base dell'espressione LAMBDA PRESENTA allo `STREAM()`.

E' applicato a uno `Stream` e fornisce in uscita un *UNICO risultato* che è **calcolato sull'intero `Stream`**. Lo *valuta interamente* sulla base dell'espressione lambda passatagli.

```
reduce(T identity, BinaryOperator<T> accumulator)
//identity compatibile con il tipo passato allo Stream. Vale 0 perchè non cambia il valore
finale dello Stream (in una somma il valore nullo è 0).

Predicate<Integer> positive = x -> x>=0;
Stream<Integer> result = Stream.of(2,5,10,-1).filter(positive);

redue(0, (accum,v) -> accum+v); // ritorna 2+5+10=17
// passo 0 -> accum = 0; v = primo valore della lista
// passo 1 -> accum = valore precedente della somma; v = secondo valore della lista
// passo 3 -> accum = valore precedente della somma; v = terzo valore della lista
// passo 4 -> ecc...
```

- Identità = non deve cambiare il risultato finale.
 - Nelle somme è `0`
 - Nei prodotti è `1`
 - Nelle stringhe è `" "`
 - ecc...