UNIVERSITÀ DEGLI STUDI DI BRESCIA · unibs.it

AA 2026/27

PAJC · PROGRAMMAZIONE AVANZATA IN JAVA E C

LO3

Stream, per intero

Pipeline, laziness, ordine di valutazione · e la metà che manca sempre



prof. Massimiliano Redolfi

Università degli Studi di Brescia · Ingegneria Informatica

Oggi

1. **Un flusso di dati**: sorgente, operazioni intermedie, operazione terminale
2. **Come lavora una pipeline**: laziness, ordine verticale, short-circuit, `sorted`
3. **Creare stream**: da collezioni, intervalli, array, `iterate`; stream primitivi e `Optional`
4. **La metà mancante**: `collect`, `Collectors`, `groupingBy`, `reduce`, `flatMap`
5. Il censimento del bimondo in una riga

01

Un flusso di dati



Prima degli stream

Da una lista di stringhe, stampare in ordine e in maiuscolo quelle che iniziano per `c`.

```
List<String> myList = List.of("a1", "a2", "b1", "c1", "c2");
List<String> conC = new ArrayList<>();
for (String s : myList) {
    if (s.startsWith("c")) conC.add(s);
}
List<String> maiuscole = new ArrayList<>();
for (String s : conC) {
    maiuscole.add(s.toUpperCase());
}
Collections.sort(maiuscole);
for (String s : maiuscole) {
    System.out.println(s);
}
```

Tre cicli, due liste d'appoggio, un `if`: il **come** copre il **che cosa**. Anche in un ciclo solo, l'ordinamento costringe a raccogliere prima e stampare dopo.

L’esempio guida

```java
List<String> myList = List.of("a1", "a2", "b1", "c1", "c2");

myList.stream()
      .filter(s -> s.startsWith("c"))
      .map(String::toUpperCase)
      .sorted()
      .forEach(System.out::println);
```

```
C1
C2
```

Immaginate i dati che **scorrono dall’alto**: a ogni passo vengono filtrati, trasformati, ordinati. Nessun indice, nessuna variabile d’appoggio, nessun `if` dentro un `for`.

La pipeline

```
myList.stream()                          // 1. sorgente
      .filter(s -> s.startsWith("c"))    // 2. operazione intermedia
      .map(String::toUpperCase)          // 2. operazione intermedia
      .sorted()                          // 2. operazione intermedia
      .forEach(System.out::println);     // 3. operazione terminale
```

- la **sorgente** produce gli elementi: una collezione, un array, un intervallo, un file
- le **operazioni intermedie** restituiscono un nuovo stream: si concatenano
- l'**operazione terminale** consuma lo stream e produce il risultato (o un effetto)

> Uno stream **non è una collezione**: non contiene dati, li fa passare. Si consuma una volta sola.

Le operazioni

Intermedie (→ `Stream`)	Terminali (→ risultato)
`filter(Predicate)`	`forEach(Consumer)`
`map(Function)`, `mapToInt(...)`	`collect(Collector)`, `toList()`
`sorted()`, `sorted(Comparator)`	`count()`, `sum()`, `average()`
`distinct()`, `limit(n)`, `skip(n)`	`anyMatch`, `allMatch`, `noneMatch`
`flatMap(Function)`	`findFirst()`, `min`, `max`
`peek(Consumer)`	`reduce(...)`

Tutte ricevono **lambda**: le interfacce funzionali della lezione scorsa sono il linguaggio delle pipeline.

02

Come lavora una pipeline



Laziness: senza terminale non succede niente

```
Stream.of("alfa", "beta", "gamma", "delta")
      .filter(s -> {
          System.out.println("Filtro: " + s);
          return true;
      });
```

```
(nessun output)
```

Le operazioni intermedie **descrivono** il lavoro; è l'operazione terminale a farlo partire. Fino ad allora lo stream è una ricetta, non un'esecuzione.

Elaborazione verticale

```
Stream.of("alfa", "beta", "gamma", "delta")
      .filter(s -> { System.out.println("Filtro: " + s); return true; })
      .forEach(s -> System.out.println("For:    " + s));
```

```
Filtro: alfa    For: alfa
Filtro: beta    For: beta
Filtro: gamma   For: gamma
Filtro: delta   For: delta
```

Non "prima tutti i filtri, poi tutti i for": **ogni elemento attraversa tutta la pipeline** prima che parta il successivo.

L’ordine conta

MAP, POI FILTER

```
Stream.of("alfa", "beta", "gamma")
 .map(s -> log("Map", s).toUpperCase())
 .filter(s -> log("Filter", s).startsWith("A")
 .forEach(s -> log("For", s));
```

```
Map: alfa    Filter: ALFA    For: ALFA
Map: beta    Filter: BETA
Map: gamma   Filter: GAMMA
```

6 chiamate (`log` stampa e restituisce `s`)

FILTER, POI MAP

```
Stream.of("alfa", "beta", "gamma")
 .filter(s -> log("Filter", s).startsWith("a")
 .map(s -> log("Map", s).toUpperCase())
 .forEach(s -> log("For", s));
```

```
Filter: alfa   Map: alfa   For: ALFA
Filter: beta
Filter: gamma
```

4 chiamate: **filtrare prima, trasformare dopo**

Short-circuit: si elabora solo ciò che serve

```
Stream.of("alfa", "beta", "gamma", "delta")
      .map(s -> { log("Map", s); return s.toUpperCase(); })
      .anyMatch(s -> { log("Match", s); return s.startsWith("B"); });
```

```
Map: alfa
Match: ALFA
Map: beta
Match: BETA
```

`anyMatch` si ferma al primo `true`: `gamma` e `delta` non vengono mai toccati. Lo stesso fanno `findFirst`, `limit`, `allMatch`, `noneMatch`.

`sorted` lavora in orizzontale

```
Stream.of("gamma", "alfa", "beta", "epsilon", "delta")
      .sorted((s1, s2) -> { log("Sort", s1 + ";" + s2); return s1.compareTo(s2); })
      .filter(s -> { log("Filter", s); return s.startsWith("a"); })
      .map(String::toUpperCase)
      .forEach(s -> log("ForEach", s));
```

```
Sort: alfa;gamma   Sort: beta;alfa   ...   (8 confronti, poi:)
Filter: alfa   Filter: beta   ...   Filter: delta   ForEach: ALFA
```

- `sorted` è **stateful**: deve vedere tutti gli elementi prima di emetterne uno; lì la verticalità si interrompe
- con il `filter` **prima** di `sorted`: un solo elemento da ordinare, zero confronti

Uno stream si consuma una volta sola

```
IntStream s = IntStream.iterate(2, f -> 2 * f);
s.anyMatch(f -> f > 15);
s.anyMatch(f -> f > 32);
```

```
Exception in thread "main" java.lang.IllegalStateException:
    stream has already been operated upon or closed
```

Serve un nuovo stream ogni volta: un `Supplier` lo fabbrica a richiesta.

```
Supplier<IntStream> supplier = () -> IntStream.iterate(2, f -> 2 * f);
supplier.get().anyMatch(f -> f > 15);
supplier.get().anyMatch(f -> f > 32);
```

03

Creare stream



Le sorgenti

```
myList.stream()                                  // da una collezione
Stream.of("alfa", "beta", "gamma")               // da valori
Arrays.stream(new int[] {1, 2, 3})               // da un array
IntStream.range(1, 10)                           // 1..9
IntStream.rangeClosed(1, 10)                     // 1..10
Stream.iterate(1, n -> n * 2)                    // infinito: 1, 2, 4, 8, ...
Files.lines(Path.of("dati.txt"))                 // le righe di un file
```

> Uno stream **infinito** va sempre chiuso da `limit`, `takeWhile` o da un'operazione short-circuit: altrimenti il programma non termina.

Fibonacci con `iterate`

Ciclico, ricorsivo, con un iteratore… oppure con uno stream:

```
LongStream fibs = Stream
        .iterate(new long[] {1, 1}, f -> new long[] {f[1], f[0] + f[1]})
        .mapToLong(f -> f[0]);

fibs.limit(10).forEach(System.out::println);
```

```
1 1 2 3 5 8 13 21 34 55
```

Lo **stato** (la coppia corrente) viaggia dentro l'elemento; `iterate` applica la funzione al precedente. Lo stream è infinito, `limit` lo chiude.

Stream primitivi

`Stream<T>` è generico; `IntStream`, `LongStream` e `DoubleStream` evitano il boxing e hanno operatori numerici dedicati.

```
IntStream.range(1, 10).filter(v -> v % 2 == 0).sum();          // 20

Arrays.stream(new int[] {1, 2, 3}).map(n -> n * n).average();  // 4.67

Stream.of("a1", "a2", "a3")
      .map(s -> s.substring(1))
      .mapToInt(Integer::parseInt)      // Stream<String> -> IntStream
      .max();                           // OptionalInt[3]

IntStream.range(1, 4)
         .mapToObj(i -> "a" + i)        // IntStream -> Stream<String>
         .toList();                     // [a1, a2, a3]
```

`Optional`: un risultato che può mancare

```
Stream.of("alfa", "beta", "gamma").findFirst();      // Optional[alfa]
Stream.<String>empty().findFirst();                  // Optional.empty
```

- `findFirst`, `min`, `max`, `average`, `reduce` senza identità: il risultato **potrebbe non esserci**
- `Optional` lo dice nel tipo, invece di restituire `null` e sperare

```
opt.ifPresent(System.out::println);          // fai qualcosa se c'è
String s = opt.orElse("nessuno");            // valore di riserva
String t = opt.orElseThrow();                // eccezione se manca
```

> `opt.get()` senza controllo è un `null` travestito. Se vi trovate a scriverlo, volevate `orElse` o `orElseThrow`.

04

La metà mancante



Raccogliere: `collect` e `toList`

`forEach` produce effetti. Per **costruire** un risultato si usa `collect`.

```
List<String> maiuscole = myList.stream()
        .map(String::toUpperCase)
        .toList();                                    // Java 16: lista immutabile

Set<Character> iniziali = myList.stream()
        .map(s -> s.charAt(0))
        .collect(Collectors.toSet());

String tutte = myList.stream()
        .collect(Collectors.joining(", "));           // "a1, a2, b1, c1, c2"
```

> Riempire una lista dentro un `forEach` funziona ma è il modo sbagliato: il risultato è del `collect`, non dell'effetto collaterale.

`groupingBy` : da lista a mappa

```
import static java.util.stream.Collectors.*;

Map<Character, List<String>> perIniziale = myList.stream()
        .collect(groupingBy(s -> s.charAt(0)));
// {a=[a1, a2], b=[b1], c=[c1, c2]}
```

Il secondo argomento dice **che cosa fare di ogni gruppo**:

```
Map<Character, Long> quante = myList.stream()
        .collect(groupingBy(s -> s.charAt(0), counting()));
// {a=2, b=1, c=2}

Map<Boolean, List<String>> pari = myList.stream()
        .collect(partitioningBy(s -> s.endsWith("2")));
// {false=[a1, b1, c1], true=[a2, c2]}
```

Il censimento del bimondo, in una riga

Alla prima lezione: una mappa, un `for`, un leggi-modifica-scrivi.

```
Map<Class<?>, Integer> census = new HashMap<>();
for (Shape2d s : shapes) {
    Integer c = census.get(s.getClass());
    census.put(s.getClass(), c == null ? 1 : c + 1);
}
```

```
Map<Class<?>, Long> census = shapes.stream()
        .collect(groupingBy(Shape2d::getClass, counting()));
```

Stessa mappa, stesso risultato. La differenza: la riga dice **che cosa** vogliamo, non **come** ottenerlo.

`reduce`: da molti a uno

```java
int somma = IntStream.rangeClosed(1, 5).reduce(0, (a, b) -> a + b);     // 15
int somma = IntStream.rangeClosed(1, 5).reduce(0, Integer::sum);         // 15
int somma = IntStream.rangeClosed(1, 5).sum();                           // 15
```

- `reduce(identità, operatore)`: parte dall'identità e combina un elemento alla volta
- senza identità restituisce un `Optional`: su uno stream vuoto non c'è nulla da combinare
- `sum`, `count`, `min`, `max`, `joining` sono **riduzioni già scritte**: usatele

```java
Optional<String> piuLunga = myList.stream()
        .reduce((a, b) -> a.length() >= b.length() ? a : b);
```

`flatMap`: appiattire

`map` produce un elemento per elemento. Quando ogni elemento ne produce **molti**, serve `flatMap`.

```
List<String> righe = List.of("alfa beta", "gamma", "delta epsilon zeta");

List<String> parole = righe.stream()
        .flatMap(r -> Arrays.stream(r.split(" ")))
        .toList();
// [alfa, beta, gamma, delta, epsilon, zeta]
```

Con `map` avremmo ottenuto uno `Stream<String[]>`: tre array, non sei parole. `flatMap` **srotola** ogni array nel flusso.

Lista di liste, righe di parole, ordini con più articoli: ogni volta che il risultato di `map` sarebbe una collezione, è un `flatMap`.

`toMap` e le chiavi duplicate

```
Map<String, Integer> lunghezze = myList.stream()
        .collect(toMap(s -> s, String::length));      // {a1=2, a2=2, ...}
```

```
Map<Character, String> prima = myList.stream()
        .collect(toMap(s -> s.charAt(0), s -> s));
// IllegalStateException: Duplicate key a
```

Con due elementi per chiave bisogna dire **come fonderli**:

```
        .collect(toMap(s -> s.charAt(0), s -> s, (x, y) -> x + "," + y));
// {a=a1,a2, b=b1, c=c1,c2}
```

05

In pratica



Stream o `for`?

Stream	`for`
trasformare, filtrare, raggruppare, contare	modificare la collezione mentre la si scorre
il risultato è un valore o una nuova collezione	servono l'indice o due collezioni in parallelo
ogni passo è una lambda breve e senza stato	il corpo del ciclo è lungo o ha `break` complessi

> Se la pipeline supera le cinque o sei operazioni, o le lambda crescono, spezzatela in metodi con un nome. Uno stream illeggibile non è più moderno di un ciclo chiaro.

Errori frequenti

- **effetti collaterali** nelle lambda intermedie: modificare una variabile esterna da `map` o `filter` non è permesso (*effectively final*) e, con `parallel()`, sarebbe un disastro
- **riusare uno stream**: `IllegalStateException`; serve un `Supplier` o una nuova chiamata a `stream()`
- **stream infinito senza** `limit`: il programma non termina
- `forEach` **per costruire** una lista: è un `collect`
- `get()` **su un** `Optional` senza controllo: è un `null` con un altro nome
- `sorted` **prima di** `filter`: si ordina anche ciò che si scarterà

E `parallel()`?

```
long n = shapes.parallelStream().filter(s -> s.area() > 10).count();
```

- una parola, e la pipeline gira su più core
- funziona **solo** se le lambda sono pure: niente stato condiviso, niente ordine assunto
- conviene solo con molti elementi e lavoro per elemento non banale

> Ci torneremo con i thread. Fino ad allora: sequenziale.