Metodologie di Programmazione: Gli stream

Roberto Navigli

Dipartimento
di Informatica

Sapienza
Università di Roma

Stanco delle NullPointerException?

- java.util.Optional è un contenitore di un riferimento che potrebbe essere o non essere null
- Un metodo può restituire un Optional invece di restituire un riferimento potenzialmente null

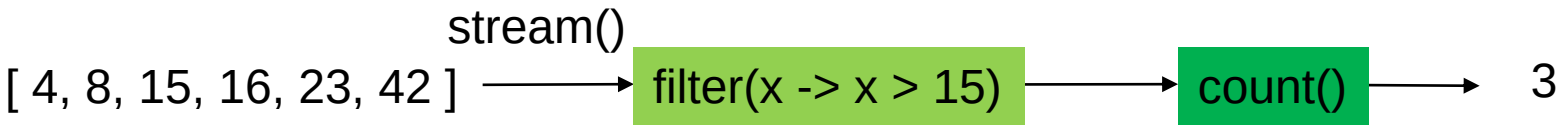

```
Optional<String> optional = Optional.ofNullable("bam");
optional.isPresent(); // true
optional.get(); // "bam"
optional.orElse("fallback"); // "bam"
optional.ifPresent(s -> System.out.println(s.charAt(0))); // "b"
```

Un nuovo meccanismo per le sequenze di elementi: gli Stream

- Una nuova interfaccia java.util.stream.Stream
- Rappresenta una sequenza di elementi su cui possono essere effettuate una o più operazioni
- Supporta operazioni sequenziali e paralelle
- Uno Stream viene creato a partire da una sorgente di dati, ad esempio una java.util.Collection
- Al contrario delle Collection, uno Stream non memorizza né modifica i dati della sorgente, ma opera su di essi

Stream: operazioni intermedie e terminali

- Le operazioni possono essere intermedie o terminali
  - **Intermedie:** restituiscono un altro stream su cui continuare a lavorare
  - **Terminali:** restituiscono il tipo atteso



- Una volta che uno stream è stato consumato (**operazione terminale**), non può essere riutilizzato
- **Builder pattern**: si impostano una serie di operazioni per configurare (op. intermedie) e infine si costruisce l'oggetto (op. terminale)

Metodi principali dell'interfaccia java.util.stream.Stream

Metodo	Tipo	Descrizione
<R, A> R **collect**(Collector<? super T, A, R> *collectorFunction*)	**T**	Raccoglie gli elementi di tipo T in un contenitore di tipo R, accumulando in oggetti di tipo A
long **count**()	**T**	Conta il numero di elementi
Stream<T> **filter**(Predicate<? super T> *predicate*)	**I**	Fornisce uno stream che contiene solo gli elementi che soddisfano il predicato
void **forEach**(Consumer<? super T> *action*)	**T**	Esegue il codice in input su ogni elemento dello stream
<R> Stream<R> **map**(Function<? super T, ? extends R> *mapFunction*)	**I**	Applica la funzione a tutti gli elementi dello stream, fornendo un nuovo stream di elementi mappati
IntStream **mapToInt/ToDouble/ToLong**(ToIntFunction<? super T> *mapFunction*)	**I**	Come sopra, ma la mappatura è su interi/ecc. (**operazione ottimizzata**)
Optional<T> **max/min**(Comparator<? super T> *comparator*)	**T**	Restituisce il massimo/minimo elemento all'interno dello stream
T **reduce**(T *identityVal*, BinaryOperator<T> *accumulator*)	**T**	Effettua l'operazione di riduzione basata sugli elementi dello stream
Stream<T> **sorted**()	**I**	Produce un nuovo stream di elementi ordinati
Stream<T> **limit**(long k)	**I**	Limita lo stream a k elementi

Stream: operazioni intermedie e terminali (2)

- Una volta che uno stream è stato consumato (**operazione terminale**), non può essere riutilizzato
- Comportamento pigro (lazy behavior): Le operazioni intermedie non vengono eseguite immediatamente, ma solo quando si richiede l'esecuzione di un'operazione terminale
- Le operazioni possono essere:
  - **senza stato** (*stateless*): l'elaborazione dei vari elementi può procedere in modo indipendente (es. filter)
  - con **stato** (*stateful*): l'elaborazione di un elemento potrebbe dipendere da quella di altri elementi (es. sorted)

Stream, IntStream, DoubleStream, LongStream

- Poiché Stream opera su oggetti, esistono analoghe versioni ottimizzate per lavorare con 3 tipi primitivi:
  - Su int: IntStream
  - Su double: DoubleStream
  - Su long: LongStream
- Tutte queste interfacce estendono l'interfaccia di base BaseStream

Come ottenere uno stream?

- Direttamente dai dati: con il metodo statico generico Stream.of(elenco di dati di un certo tipo)
- In Java 8 l'interfaccia Collection è stata estesa per includere due nuovi metodi di default:

- default Stream<E> stream() – restituisce un nuovo stream sequenziale
- default Stream<E> parallelStream() – restituisce un nuovo stream parallelo, se possibile (altrimenti restituisce uno stream sequenziale)

- E' possibile ottenere uno stream anche per un array, con il metodo statico Stream<T> Arrays.stream(T[ ] *array*)
- E' possibile ottenere uno stream di righe di testo da BufferedReader.lines() oppure da Files.lines(Path)

Stream vs. collection

- Lo stream permette di utilizzare uno **stile dichiarativo**
  - Iterazione interna sui dati
- La collection impone l'utilizzo di uno stile imperativo
  - Iterazione esterna sui dati (tranne con forEach)
- Lo stream si focalizza sulle **operazioni di alto livello** da eseguire (mattoncini o building blocks) eventualmente anche in parallelo, senza specificare **come verranno eseguite**
- **Stream:** dichiarativo, componibile, parallelizzabile

Metodi di java.util.stream.Stream: min e max

- I metodi min e max restituiscono rispettivamente il minimo e il massimo di uno stream sotto forma di Optional
- Prendono in input un Comparator sul tipo degli elementi dello stream

```
List<Integer> p = Arrays.asList(2, 3, 4, 5, 6, 7);
Optional<Integer> max = p.stream().max(Integer::compare);
// se c'è, restituisce il massimo; altrimenti restituisce -1
System.out.println(max.orElse(-1));
```

Metodi di java.util.stream.Stream: filter (intermedio), forEach (terminale)

- filter è un metodo di Stream che accetta un predicato (Predicate) per filtrare gli elementi dello stream
  - Operazione intermedia che restituisce lo stream filtrato
- forEach prende in input un Consumer e lo applica a ogni elemento dello stream
  - Operazione terminale

Esempi di Stream (con filter e forEach)

- Filtra gli elementi di una lista per iniziale e stampa ciascun elemento rimanente:

```
List<String> l = Arrays.asList("da", "ab", "ac", "bb");
l.stream()
  .filter(s -> s.startsWith("a"))
  .forEach(System.out::println);
```

- Filtra gli elementi di una lista di interi mantenendo solo quelli dispari e stampa ciascun elemento rimanente:

```
List<Integer> l = Arrays.asList(4, 8, 15, 16, 23, 42);
l.stream()
  .filter(k -> k % 2 == 1)
  .forEach(System.out::println);
```

Altro esempio di Stream (con filter e forEach)

- Filtra gli elementi di una lista per iniziale e lunghezza della stringa e stampa ciascun elemento rimanente:

```
Predicate<String> startsWithJ = s -> s.startsWith("J");
Predicate<String> fourLetterLong = s -> s.length() == 4;

List<String> l = Arrays.asList("Java", "Scala", "Lisp");
l.stream()
  .filter(startsWithJ.and(fourLetterLong))
  .forEach(s -> System.out.println("Inizia con J ed e' lungo
4 caratteri: "+s);
```

Metodi di java.util.stream.Stream: count (terminale)

- count è un'operazione terminale che restituisce il numero long di elementi nello stream
- Esempio:

```
long startsWithA = l.stream().filter(s -> s.startsWith("a")) .count();
System.out.println(startsWithA); // 2
```

- Esempio di conteggio del numero di righe di un file di testo:

```
long numberOfLines = Files.lines(Paths.get("yourFile.txt")).count();
```

Metodi di java.util.stream.Stream: sorted (intermedia)

- sorted è un'operazione intermedia sugli stream che restituisce una vista ordinata dello stream **senza modificare la collezione sottostante**
- Esempio:

```
List<String> l = Arrays.asList("da", "ac", "ab", "bb");
l.stream()
  .sorted()
  .filter(s -> s.startsWith("a"))
  .forEach(System.out::println);
```

- Stampa:

```
ab
ac
```

Metodi di java.util.stream.Stream: map (intermedia)

- map è un'operazione intermedia sugli stream che restituisce un nuovo stream in cui ciascun elemento dello stream di origine è convertito in un altro oggetto attraverso la funzione (Function) passata in input
- Esempio: restituire tutte le stringhe (portate in maiuscolo) ordinate in ordine inverso

```
// equivalente a .map(s -> s.toUpperCase())
l.stream().map(String::toUpperCase).sorted(Comparator.<String>naturalOrder().reversed()) .forEach(System.out::println);
```

- Stampa:

```
DA
BB
AC
AB
```

java.util.stream.Stream.map: esempio

- Si vuole scrivere un metodo che aggiunga l'IVA a ciascun prezzo:

```
List<Integer> ivaEsclusa = Arrays.asList(10, 20, 30);

// In Java 7:
for (int p : ivaEsclusa)
{
    double pIvaInclusa = p*1.22;
    System.out.println(pIvaInclusa);
}
// In Java 8:
ivaEsclusa.stream().map(p -> p*1.22).forEach(System.out::println);
```

Metodi di java.util.stream.Stream: collect (terminale)

- collect è un'operazione terminale che permette di raccogliere gli elementi dello stream in un qualche oggetto (ad es. una collection)
- Ad esempio, per ottenere la lista dei prezzi ivati:

```
List<Integer> ivaEsclusa = Arrays.asList(10, 20, 30);
// In Java 7:
List<Double> l = new ArrayList<>();
for (int p : ivaEsclusa) l.add(p*1.22);
// In Java 8:
List<Double> l = ivaEsclusa.stream().map(p -> p*1.22)
                                    .collect(Collectors.toList());
```

Esempio: creare una stringa che concatena stringhe in una lista, rese maiuscole e separate da virgola

```
List<String> l = Arrays.asList("RoMa", "milano", "Torino");
String s = "";

// in Java 7:
for (String e : l) s += e.toUpperCase()+", ";
s = s.substring(0, s.length()-2);

// in Java 8:
s = l.stream().map(e -> e.toUpperCase())
              .collect(Collectors.joining(", "));
```

Esempio: trasformare una lista di stringhe in una lista delle lunghezze delle stesse

```
List<String> words = Arrays.asList("Oracle", "Java",
"Magazine");
List<Integer> wordLengths =
        words.stream()
        .map(String::length)
        .collect(toList());
```

java.util.stream.Collectors

- "Ricette" per ridurre gli elementi di uno stream e raccoglierli in qualche modo
- Per rendere più leggibile il codice: import static java.util.stream.Collectors.*
  - In questo modo possiamo scrivere il nome del metodo senza anteporre Collectors. (es. toList() invece di Collectors.toList())

java.util.stream.Collectors: riduzioni a singolo elemento

- counting() – restituisce il numero di elementi nello stream (risultato di tipo long)

```
List<Integer> l = Arrays.asList(2, 3, 5, 6);
// k == 2
long k = l.stream().filter(x -> x < 5).collect(Collectors.counting()));
```

- maxBy/minBy(comparator) – restituisce un Optional con il massimo/minimo valore

```
// max contiene 6
Optional<Integer> max = l.stream().collect(maxBy(Integer::compareTo));
```

- summingInt(lambda che mappa ogni elemento a intero)/averagingInt, summingDouble, averagingDouble

java.util.stream.Collectors: riduzioni a singolo elemento

- joining(), joining(separatore), joining(separatore, prefisso, suffisso) – concatena gli elementi stringa dello stream in un'unica stringa finale

```
List<Integer> l = Arrays.asList(2, 3, 5, 6, 2, 7);
// str.equals("2,3,5,6,2,7")
String str = l.stream().map(x -> ""+x).collect(joining(","));
```

- toList, toSet e toMap – accumulano gli elementi in una lista, insieme o mappa (non c'è garanzia sul tipo di List, Set o Map)

```
Set<String> set = l.stream().map(x -> ""+x).collect(toSet());
```

- toCollection – accumula gli elementi in una collezione scelta

```
ArrayList<String> str = l.stream().map(x -> ""+x)
                                  .collect(toCollection(ArrayList::new));
```

Collectors.toMap: riduzione a una mappa

- toMap prende in input fino a 4 argomenti:
  - la funzione per mappare l'oggetto dello stream nella chiave della mappa
  - la funzione per mappare l'oggetto dello stream nel valore della mappa
  - **opzionale:** la funzione da utilizzare per unire il valore preesistente nella mappa a fronte della chiave con il valore associato all'oggetto dalla seconda funzione (non devono trovarsi due chiavi uguali o si ottiene un'eccezione IllegalStateException)
  - **opzionale:** il Supplier che crea la mappa

```
Map<Integer, String> map = persons
    .stream()
    .collect(Collectors.toMap(
        Person::getAge,
        Person::getName,
        (name1, name2) -> name1 + ";" + name2));
```