DE: Department of Engineering Ferrara

Ingegneria del Software Avanzata A.A. 2021/2022

Stream Java

Damiano Azzolini
damiano.azzolini@unife.it

Laurea Magistrale in Ingegneria Informatica e dell'Automazione - Università di Ferrara

Recap: Collezioni

Strutture dati fornite dalle API Java.

Interfaccia comune (tranne per le mappe): `Collection`.

`List`: interfaccia per una collezione dove ciascun elemento ha un indice che parte da 0. Molti metodi tra cui `get`, `set`, `remove` e `indexOf`. `List` implementata da `ArrayList` e `LinkedList`.

`Set`: interfaccia per una collezione dove gli elementi non hanno una posizione particolare e non sono ammessi duplicati. `SortedSet`: aggiunge l'ordinamento.

`Map`: associazioni chiave-valore.

Si consiglia di utilizzare il più possibile le interfacce: *subtyping*. Per esempio, se viene creato un `ArrayList`, posso salvarne il riferimento in `List`: `List<String> parole = new ArrayList<>();`

Esistono moltissimi metodi già implementati.

Si può utilizzare l'interfaccia `Iterator` per definire degli iteratori coi quali si possono scorrere gli elementi di una collezione.

DE: Department of Engineering Ferrara

Java Stream

Un oggetto di tipo `Stream` rappresenta una sequenza di elementi sulla quale si possono eseguire delle operazioni definite a livello concettuale.

Con uno stream è possibile specificare ciò che si desidera fare, non il modo in cui farlo.

È possibile creare stream da collezioni, array, generatori o iteratori.

Per selezionare gli elementi si usa il metodo `filter`, per trasformarli si usa il metodo `map`.

Per ottenere un risultato da uno stream si utilizza un operatore di *riduzione* come `count`, `max`, `min`, `findFirst` o `findAny`. Alcuni di questi restituiscono un valore `Optional`.

La classe `Optional` è utile per la gestione dei valori `null`.

Caratteristiche degli stream

Uno stream non memorizza elementi.

Le operazioni con gli stream non modificano la sorgente: per esempio il metodo `filter` non rimuove elementi dallo stream iniziale, ma restituisce un nuovo stream senza gli elementi.

Le operazioni degli stream sono *lazy*: non vengono eseguite finché non è necessario il loro risultato.

Flusso di lavoro di uno stream

1. Si crea uno stream
2. Si specificano le *operazioni intermedie* che trasformano lo stream iniziale in altri stream.
3. Si applica un'*operazione terminale* che produce un risultato.

Creare uno Stream

Se si ha un array si può utilizzare il metodo statico `Stream.of`

```
1  Stream<String> wordsStream = Stream.of("Mi", "chiamo", "...");
```

Se si ha una collezione, è possibile trasformarla in stream con il metodo `stream`.

```
1  List<String> words = new ArrayList<>();
2  // ...
3  Stream<String> wordsStream = words.stream();
```

Col metodo `parallelStream` è possibile creare degli stream paralleli (sui quali possono essere eseguite operazioni in parallelo)

```
1  Stream<String> wordsStream = words.parallelStream();
```

Il metodo `forEach`

Con il metodo `forEach` è possibile applicare un'operazione (o serie di operazioni) a ciascun elemento dello stream.

L'argomento del metodo `forEach` è di tipo `Consumer<T>`, ossia un'interfaccia funzionale con una funzione che prende in ingresso un elemento di tipo `T` e restituisce `void`.

```
1  Stream<String> wordsStream = Stream.of("Mi", "chiamo", "...");
2  wordsStream.forEach(x -> System.out.println(x));
3  // oppure
4  // wordsStream.forEach(System.out::println);
```

Il metodo `filter`

Il metodo `filter` restituisce un nuovo stream con gli elementi che rispettano una determinata condizione.

L'argomento di `filter` è di tipo `Predicate<T>`, un'interfaccia funzionale con una funzione che prende in ingresso un oggetto di tipo `T` e restituisce `boolean`.

```
1  Stream<String> wordsStream = Stream.of("Mi", "chiamo", "...");
2
3  Stream<String> longWordsStream = wordsStream.filter(x -> x.
       length() > 5);
```

Il metodo `map`

Per trasformare i valori di uno stream si può usare il metodo `map`.

L'argomento di `map` è di tipo `Function<T,R>`, un'interfaccia con una funzione che prende in ingresso `T` e restituisce `R`.

```
1  Stream<String> lowerCaseWordsStream = words.stream().map(String
       ::toLowerCase);
2
3  Stream<String> firstLettersStream = words.stream().map(s -> s.
       substring(0,1));
```

Il risultato dell'applicazione di `map` è un nuovo stream con i valori ottenuti.

Generazione di Elementi

Posso generare uno stream di elementi *infinito* utilizzando `generate(Supplier<T>)` dove ciascun elemento è generato da `Supplier` (interfaccia funzionale).

Genero stream di numeri random:

```
1  Stream<Double> randomNumbers = Stream.generate(Math::random);
```

Come faccio ad estrarre gli elementi?

Estrazione di Sotto-stream

Posso utilizzare `stream.limit(n)` per estrarre i primi `n` elementi da uno stream di partenza (o tutti se lo stream ha meno di n elementi).

```
1  Stream<Double> randomNumbers = Stream.generate(Math::random).
       limit(10);
```

Posso utilizzare `stream.skip(n)` per saltare i primi n elementi di uno stream di partenza e collezionare i rimanenti.

```
1  Stream <String> paroleLungheUpper = paroleLunghe.map(s -> s.
       replace('a', 'k')).skip(1);
```

Riduzioni I

Le *riduzioni* sono *operazioni terminali* che riducono lo stream a un valore non stream.

Ne esistono diverse, `count` restituisce un intero, mentre altre restituiscono un valore `Optional<T>`.

Per esempio, il metodo `count()` conta gli elementi di uno stream.

```
1  System.out.println(randomNumbers.count());
```

Come gestire un valore di ritorno mancante? La classe `Optional<T>` funge da wrapper per un oggetto di tipo `T` o per nessun oggetto ed è utile per la gestione dei valori `null`.

```
1  Optional<Double> result = randomNumbers.filter(x -> x > 1).
       findFirst();
```

Riduzioni II

L'utilizzo principale è quello di fornire un'alternativa alla risposta cercata se questa non esiste utilizzando metodi come `orElse()` or `orElseGet()`.

```
1  Optional<Double> result = randomNumbers.filter(x -> x > 1).
       findFirst();
2  System.out.println(result.orElse((double) -1));
3
4  long count = words.stream().filter(w -> w.length() > 12).count
       ();
5  Optional<String> startsWithGFirst = words.stream().filter(s ->
       s.startsWith("G")).findFirst();
6  Optional<String> startsWithGAny = words.stream().filter(s -> s.
       startsWith("G")).findAny();
7  String s = startsWithGAny.orElse("");
8  s = startsWithGAny.orElseGet(() -> System.getProperty("user.
       name"));
```

Raggruppare i Risultati

Posso raccogliere gli elementi di uno stream in una lista utilizzando il metodo `collect` e i metodi della classe `Collectors`.

```
1  List<Double> coll = randomNumbers.collect(Collectors.toList());
2  System.out.println(coll);
```