Programmazione 2

Nona esercitazione

17/05/2024

Java streams

Java streams

- Il package java.util.stream definisce classi e interfacce per costruire delle "pipeline" di operazioni su flussi di valori, potenzialmente infiniti

Java streams

- Il package java.util.stream definisce classi e interfacce per costruire delle "pipeline" di operazioni su flussi di valori, potenzialmente infiniti
- Una pipeline si compone di:

Java streams

- Il package java.util.stream definisce classi e interfacce per costruire delle "pipeline" di operazioni su flussi di valori, potenzialmente infiniti
- Una pipeline si compone di:
  - Una sorgente (`Collection`, funzione generatrice, un canale di I/O, ...)

Java streams

- Il package java.util.stream definisce classi e interfacce per costruire delle "pipeline" di operazioni su flussi di valori, potenzialmente infiniti
- Una pipeline si compone di:
  - Una sorgente (`Collection`, funzione generatrice, un canale di I/O, ...)
  - Una sequenza di operazioni intermedie da applicare ai suoi valori

Java streams

- Il package java.util.stream definisce classi e interfacce per costruire delle "pipeline" di operazioni su flussi di valori, potenzialmente infiniti
- Una pipeline si compone di:
  - Una sorgente (`Collection`, funzione generatrice, un canale di I/O, ...)
  - Una sequenza di operazioni intermedie da applicare ai suoi valori
  - Una operazione terminale

Java streams

- Il package java.util.stream definisce classi e interfacce per costruire delle "pipeline" di operazioni su flussi di valori, potenzialmente infiniti
- Una pipeline si compone di:
  - Una sorgente (`Collection`, funzione generatrice, un canale di I/O, ...)
  - Una sequenza di operazioni intermedie da applicare ai suoi valori
  - Una operazione terminale
- Ogni operazione intermedia genera un nuovo stream per la successiva

Java streams

- Il package java.util.stream definisce classi e interfacce per costruire delle "pipeline" di operazioni su flussi di valori, potenzialmente infiniti
- Una pipeline si compone di:
  - Una sorgente (`Collection`, funzione generatrice, un canale di I/O, ...)
  - Una sequenza di operazioni intermedie da applicare ai suoi valori
  - Una operazione terminale
- Ogni operazione intermedia genera un nuovo stream per la successiva
- Una volta eseguita l'operazione terminale, la pipeline è consumata
  - Per eseguire altre operazioni sulla sorgente, occorre ri-convertirla in stream

Java streams: esempio

Java streams: esempio

- Un esempio di pipeline:
  - Uso un array di `String` come sorgente

Java streams: esempio

- Un esempio di pipeline:
  - Uso un array di `String` come sorgente
  - Estraggo le iniziali da ogni elemento

Java streams: esempio

- Un esempio di pipeline:
  - Uso un array di `String` come sorgente
  - Estraggo le iniziali da ogni elemento
  - Converto le iniziali in maiuscolo

Java streams: esempio

- Un esempio di pipeline:
  - Uso un array di `String` come sorgente
  - Estraggo le iniziali da ogni elemento
  - Converto le iniziali in maiuscolo
  - Rimuovo i duplicati

Java streams: esempio

- Un esempio di pipeline:
  - Uso un array di `String` come sorgente
  - Estraggo le iniziali da ogni elemento
  - Converto le iniziali in maiuscolo
  - Rimuovo i duplicati
  - Termino stampando a video lo stream finale

Java streams: esempio

```
String[] sorgente = { "albero", "casa", "palla",
"cane", "aereo" };

java.util.Arrays.stream(sorgente)
                .map(s -> s.substring(0, 1))
                .map(s -> s.toUpperCase())
                .distinct()
                .forEach(e -> System.out.print(e));

// Output: ACP
```

Java streams

- Ogni operazione produce un risultato, e non modifica la sorgente

Java streams

- Ogni operazione produce un risultato, e non modifica la sorgente
- Le operazioni intermedie sono generalmente *lazy*, ovvero consumano un valore solo quando necessario per proseguire

Java streams

- Ogni operazione produce un risultato, e non modifica la sorgente
- Le operazioni intermedie sono generalmente *lazy*, ovvero consumano un valore solo quando necessario per proseguire
- Di conseguenza, gli stream possono essere infiniti

Java streams

- Ogni operazione produce un risultato, e non modifica la sorgente
- Le operazioni intermedie sono generalmente *lazy*, ovvero consumano un valore solo quando necessario per proseguire
- Di conseguenza, gli stream possono essere infiniti
- Ogni elemento dello stream viene visitato una sola volta

Java streams

- Alcune importanti operazioni intermedie:

Java streams

- Alcune importanti operazioni intermedie:
  - map: genera un nuovo stream applicando una trasformazione a ogni elemento dello stream corrente

Java streams

- Alcune importanti operazioni intermedie:
  - map: genera un nuovo stream applicando una trasformazione a ogni elemento dello stream corrente
  - filter: il nuovo stream conterrà solo gli elementi che aderiscono a un criterio

Java streams

- Alcune importanti operazioni intermedie:
  - map: genera un nuovo stream applicando una trasformazione a ogni elemento dello stream corrente
  - filter: il nuovo stream conterrà solo gli elementi che aderiscono a un criterio
  - concat: accoda due stream

Java streams

- Alcune importanti operazioni intermedie:
  - map: genera un nuovo stream applicando una trasformazione a ogni elemento dello stream corrente
  - filter: il nuovo stream conterrà solo gli elementi che aderiscono a un criterio
  - concat: accoda due stream
  - distinct: rimuove duplicati

Java streams

- Alcune importanti operazioni intermedie:
  - map: genera un nuovo stream applicando una trasformazione a ogni elemento dello stream corrente
  - filter: il nuovo stream conterrà solo gli elementi che aderiscono a un criterio
  - concat: accoda due stream
  - distinct: rimuove duplicati
- Alcune importanti operazioni terminali:
  - count: rende il numero di elementi nello stream

Java streams

- Alcune importanti operazioni intermedie:
  - map: genera un nuovo stream applicando una trasformazione a ogni elemento dello stream corrente
  - filter: il nuovo stream conterrà solo gli elementi che aderiscono a un criterio
  - concat: accoda due stream
  - distinct: rimuove duplicati
- Alcune importanti operazioni terminali:
  - count: rende il numero di elementi nello stream
  - forEach: applica una funzione con side-effect agli elementi dello stream

Java streams

- Alcune importanti operazioni intermedie:
  - map: genera un nuovo stream applicando una trasformazione a ogni elemento dello stream corrente
  - filter: il nuovo stream conterrà solo gli elementi che aderiscono a un criterio
  - concat: accoda due stream
  - distinct: rimuove duplicati
- Alcune importanti operazioni terminali:
  - count: rende il numero di elementi nello stream
  - forEach: applica una funzione con side-effect agli elementi dello stream
  - collect: utile per riconvertire uno stream in una collezione

Java streams

- Modi interessanti per generare stream:

Java streams

- Modi interessanti per generare stream:
  - Da un oggetto `Collection` usando il suo metodo `stream()`

Java streams

- Modi interessanti per generare stream:
  - Da un oggetto `Collection` usando il suo metodo `stream()`
  - Da un array, con `Arrays.stream(Object[])`

Java streams

- Modi interessanti per generare stream:
  - Da un oggetto `Collection` usando il suo metodo `stream()`
  - Da un array, con `Arrays.stream(Object[])`
  - Da un file, con `BufferedReader.lines()`

Java streams

- Modi interessanti per generare stream:
  - Da un oggetto `Collection` usando il suo metodo `stream()`
  - Da un array, con `Arrays.stream(Object[])`
  - Da un file, con `BufferedReader.lines()`
  - Dai metodi statici dei vari stream

Java streams

- Modi interessanti per generare stream:
  - Da un oggetto `Collection` usando il suo metodo `stream()`
  - Da un array, con `Arrays.stream(Object[])`
  - Da un file, con `BufferedReader.lines()`
  - Dai metodi statici dei vari stream

- In particolare, tra questi ultimi:
  - `Stream.iterate(T seed, UnaryOperator<T> f)`
  - `IntStream.range(int startInclusive, int endExclusive)`

Java streams: esercizio

Java streams: esercizio

- Scaricate da E-Learning il file TestStream.java

- Generate gli stream da `miaCollection` e manipolateli come richiesto
  - Suggerimento: tra gli import trovate l'operazione terminale da stream a lista

```
List<String> collectionMaiuscola = /* TRASFORMAZIONE
                DELLE STRINGHE IN MAIUSCOLO */
List<String> collectionBrevi = /* RIMOZIONE DELLE
                STRINGHE PIÙ LUNGHE DI 4 */
```