LPP - +3CFU (corso da 9 CFU)

Operazioni sugli stream

Viviana Bono

Referenze

Capitolo 5 di **Java 8 in action**

API:
https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Running example: classe Dish

```
public class Dish {
  private final String name;
  private final boolean vegetarian;
  private final int calories;
  private final Type type;
  public Dish(String name, boolean vegetarian, int calories, Type type) {
     this.name = name;
     ...
  }
  public String getName() {
    return name;
  }
  public boolean isVegetarian() {
    return vegetarian;

  public int getCalories() {
    return calories;
  }
  public Type getType() {
    return type;
  }
  @Override
  public String toString() {
     return name;
  }
  public enum Type { MEAT, FISH, OTHER }
} // end class Dish
```

Ricapitolando...

```
 // in Java 7:
List<Dish> vegetarianDishes = new ArrayList<>();
for(Dish d: menu)
  if(d.isVegetarian())
      vegetarianDishes.add(d);


// in Java 8:
import static java.util.stream.Collectors.toList;
  List<Dish> vegetarianDishes =
  menu.stream()
      .filter(Dish::isVegetarian)
      .collect(toList());
```

Ricapitolando...

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

// nel main
List<Dish> menu = Arrays.asList(
new Dish("pork", false, 800, Dish.Type.MEAT),
new Dish("beef", false, 700, Dish.Type.MEAT),
...
new Dish("salmon", false, 450, Dish.Type.FISH) );

import static java.util.stream.Collectors.toList;

List<String> threeHighCaloricDishName =
    menu.stream()
        .filter(d -> d.getCalories() > 300)
        .map(Dish::getName) // equivale a map(d -> d.getName())
        .limit(3)
        .collect(toList());
// stream() ottiene uno stream dalla lista
// filter() prende come argomento un Predicate<T>
// map() prende come argomento una Function<T, R>
// limit() prende i primi 3 elementi dello stream
// stream, filter, map, limit: intermediate operations, rest. uno stream
// collect: terminal operation
```

Operazioni sugli stream (1)

```
// eliminare i duplicati con distinct():
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
    .filter(i -> i % 2 == 0)
    .distinct()
    .forEach(System.out::println);


// troncare uno stream con limit(n):
List<Dish> dishes = menu.stream()
                    .filter(d -> d.getCalories() > 300)
                    .limit(3)
                    .collect(toList());
```

Operazioni sugli stream (2)

```
// eliminare i primi n elementi con skip(n):
List<Dish> dishes = menu.stream()
                    .filter(d -> d.getCalories() > 300)
                    .skip(3)
                    .collect(toList());
// se ci sono meno di n elementi, restuituisce lo stream vuoto


// per contare gli elementi:
long count = menu.stream().count();


/*** ESERCIZIO 1: restituire i primi due piatti di carne **/
```

Operazioni sugli stream (3)

```
// applicare una funzione a tutti gli elementi di uno stream con map():
List<String> dishNames = menu.stream()
                         .map(Dish::getName)
                         .collect(toList());

List<String> words = Arrays.asList("Java8", "Lambdas", "In", "Action");
List<Integer> wordLengths = words.stream()
                            .map(String::length)
                            .collect(toList());

List<Integer> dishNameLengths = menu.stream() // ecco uno Stream<Dish>
                    .map(Dish::getName)  // passo a uno Stream<String>
                    .map(String::length) // passo a uno Stream<Integer>
                    .collect(toList());

// map() di Java 8 assomiglia molto a map di Haskell:
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squares =
     numbers.stream()
            .map(n -> n * n)
            .collect(toList());

/*** ESERCIZIO 2: studiare l'operazione su stream flatMap() **/
```

Operazioni sugli stream (4)

```
// ESISTE:  anyMatch()
// (restituisce boolean, non Stream<T> --> terminal operation):
if(menu.stream().anyMatch(Dish::isVegetarian)){
      System.out.println("The menu is (somewhat) vegetarian friendly!!");
}

// PER OGNI:  allMatch()
boolean isHealthy = menu.stream()
             .allMatch(d -> d.getCalories() < 1000);


// NOT(ESISTE):  noneMatch()
boolean isHealthy = menu.stream()
            .noneMatch(d -> d.getCalories() >= 1000);

// Come && e ||, queste tre ultime operazioni sono lazy
// (usano il "short-circuiting")
```

Operazioni sugli stream (5): findAny e classe Optional⟨T⟩

```
// Ottenere un elemento qualsiasi con findAny():
Optional<Dish> dish = menu.stream()
                      .filter(Dish::isVegetarian)
                      .findAny();
```

findAny() restituisce **Optional⟨T⟩**: potrebbe non restituire nessun valore, e **null per rappresentare un non-valore non è una buona scelta**.

In Java 8 è quindi stato introdotto il tipo Optional⟨T⟩. La classe Optional offre i seguenti metodi:
. isPresent() returns true if Optional contains a value, false otherwise.
. ifPresent(Consumer⟨T⟩ block) executes the given block if a value is present.
. T get() returns the value if present; otherwise it throws a NoSuchElement-Exception.
. T orElse(T other) returns the value if present; otherwise it returns a default value (represented by the object 'other').

Optional⟨T⟩

```
// Esempio di uso di ifPresent() di Optional:
Optional<Dish> dish = menu.stream()
                       .filter(Dish::isVegetarian)
                       .findAny()
                       .ifPresent(d -> System.out.println(d.getName()));
```

findAny() $\neq$ findFirst() nel caso di esecuzione parallela delle operazioni su stream

Operazione reduce()

```
int sum1 = numbers.stream().reduce(0, (a, b) -> a + b);
int sum2 = numbers.stream().reduce(0, Integer::sum);
                            // chiama il metodo statico sum di Integer
// reduce() prende un valore iniziale e una lambda (di tipo BinaryOperator<T>)

// versione di reduce() senza valore iniziale:
Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));
// per gestire il caso di stream vuoto, non volendo restituire 0,
// siccome 0 potrebbe anche essere il risultato della somma

Optional<Integer> max = numbers.stream().reduce(Integer::max);
Optional<Integer> min1 = numbers.stream().reduce(Integer::min);
Optional<Integer> min2 = numbers.stream().reduce((x,y)->x<y?x:y);
```

map() e reduce()

```
/*** ESERCIZIO 3: cosa fa il seguente pezzo di codice?
 */
int xxx = menu.stream()
             .map(d -> 1)
             .reduce(0, (a, b) -> a + b);
```

Parellel streams

```
int sum3 = numbers.parallelStream().reduce(0, Integer::sum);
```

ATTENZIONE: in caso di parallelStream(), le lambda espressioni passate (es. nella reduce()) NON devono avere side-effect, ovvero non cambiare lo stato di eventuali oggetti, e le operazioni devono essere ASSOCIATIVE per poter essere eseguite in qualsiasi ordine dalla JVM. **NOTA**: negli stream della teoria non ci sono side effect!

Operazioni:
- stateless (es. filter, map): non hanno stato interno;
- stateful (es. reduce, sum, max): hanno uno stato "accumulatore" interno;
- stateful (es. sorted, distinct): hanno uno stato "buffer" interno.

Numeric streams (1)

```
int calories = menu.stream()
                .map(Dish::getCalories)
                .reduce(0, Integer::sum); // unboxing --> costoso!

int calories = menu.stream()
                .mapToInt(Dish::getCalories)
                .sum(); // senza unboxing

// da un tipo a un altro:
IntStream intStream = menu.stream()
                    .mapToInt(Dish::getCalories)
                  //.sum(); // sbagliato, restituisce int

Stream<Integer> stream = intStream.boxed();

// tipi Optional per i numeri:
OptionalInt maxCalories = menu.stream()
                          .mapToInt(Dish::getCalories)
                          .max();

int max = maxCalories.orElse(1); // cosa fa?
```

Numeric streams (2)

```
// range numerici:
IntStream evenNumbers = IntStream.rangeClosed(1,100)
                        .filter(n -> n%2==0);
System.out.println(evenNumbers.count());

// rangeClosed() --> estremi inclusi
// range() --> estremi esclusi
// ci sono anche per LongStream
```

Come creare stream

```
Stream<String> stream = Stream.of("Java 8 ", "Lambdas ", "In ", "Action");
// of metodo statico di Stream

// converto in tutto-maiuscolo e stampo:
stream.map(String::toUpperCase).forEach(System.out::println);


// creo lo stream vuoto
// (qui, a differenza del modello che stiamo studiando
// gli stream possono essere finiti o infiniti):
Stream<String> emptyStream = Stream.empty();


// da array:
int[] numbers = {2,3,4,5,7,11,13};
IntStream ar = Arrays.stream(numbers);
int sum = ar.sum();
```

È possibile costruire stream a partire da file. Si può vedere un esempio nella Sezione 5.7.3 del libro di testo.

Creare stream infiniti: uso di funzioni (1)

```
/* ITERATE */

// creo i numeri pari a partire da 0, li limito a 10,
// stampo questi 10:
Stream.iterate(0, n -> n + 2) // unbounded stream!
.limit(10)
.forEach(System.out::println);
// iterate() prende un valore iniziale
// e un lambda (di tipo Unary-Operator<T>)


// Cosa fa il seguente pezzo di codice?

Stream.iterate(new int[]{0, 1},
                t -> new int[]{t[1], t[0]+t[1]})
                .limit(20)
                .map(t -> t[0])
                .forEach(System.out::println);
```

Creare stream infiniti: uso di funzioni (2)

```
/* GENERATE */

Stream.generate(Math::random)
              .limit(5)
              .forEach(System.out::println);
// genera valori iterando l'applicazione
// della lambda espressione di tipo Supplier<T>
// che e' il suo parametro

// ATTENZIONE! Se la lambda espressione e' stateful
// (ovvero modifica lo stato della computazione),
// usare generate() in codice parallelo non
// e' safe! Nell'esempio precedente, Math::random
// e' stateless.
```