18
Stream funzionali

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2025/2026

Outline

Goal della lezione

- Mostrare la gestione funzionale degli Stream
- Discutere altri aspetti relativi alle novità di Java 8

Outline

Il concetto di Stream

Idee

- Uno Stream rappresenta un flusso sequenziale (anche infinito) di dati omogenei, usabile una volta sola, e dal quale si vuole ottenere una informazione complessiva e/o aggregata
- Assomiglia molto al concetto di Iteratore, ma lo Stream è dichiarativo, perché non indica passo-passo come l'informazione viene processata, e quindi è concettualmente più astratto
- Ove possibile, uno Stream manipola i suoi elementi in modo "lazy" (ritardato): i dati vengono processati mano a mano che servono, non sono memorizzati tutti assieme come nelle Collection
- È possibile creare "catene" di trasformazioni di Stream (implementate con decorazioni successive) in modo funzionale, per ottenere flessibilmente computazioni non banali dei loro elementi, con codice più compatto e leggibile
- Questa modalità di lavoro rende molte computazioni (automaticamente) parallelizzabili, ossia computabili da un set arbitrario di Thread

Computazioni con gli Stream

Struttura a pipeline

- Una sorgente o sink:
  - Una Collection/array, un dispositivo di I/O, una funzione generatrice
- Una sequenza di trasformazioni:
  - mappe e filtri, ma non solo...
- Un terminatore, che aggrega i dati dello Stream:
  - una riduzione ad un valore, una Collection/array, un Iteratore

Esempio: con persone con nome, città e reddito

- Data una `List<Person>` con proprietà reddito e città, ottenere la somma dei redditi di tutte le persone di Cesena
- Come lo realizziamo tramite una pipeline?
  - Sorgente: la lista
  - Trasformazione 1: filtro sulle persone di Cesena
  - Trasformazione 2: si mappa ogni persona sul suo reddito
  - Terminazione: sommo
- Aspetto cruciale: le fasi intermedie (dopo le trasformazioni), non generano collezioni temporanee

Classe Person – equals, hashCode e toString omessi

```java
public class Person {

  private final String name;
  private final Optional<String> city;
  private final double income;
  private final Set<String> jobs;

  public Person(String name, String city, double income, String... jobs) {
    this.name = Objects.requireNonNull(name);
    this.city = Optional.ofNullable(city); // null in ingresso indica città assente
    this.income = income;
    this.jobs = new HashSet<>(List.of(jobs)); // conversione a set
  }

  public String getName() {
    return this.name;
  }

  public Optional<String> getCity() {
    return this.city;
  }

  public double getIncome() {
    return this.income;
  }

  public Set<String> getJobs() {
    return Collections.unmodifiableSet(this.jobs); // copia difensiva
  }
```

Realizzazione dell'esempio in Java 8

```java
public class UseStreamsOnPerson {

  public static void main(String[] args) {

    final List<Person> list = new ArrayList<>();
    list.add(new Person("Mario", "Cesena", 20000, "Teacher"));
    list.add(new Person("Rino", "Forlì", 30000, "Football player"));
    list.add(new Person("Lino", "Cesena", 110000, "Chef", "Artist"));
    list.add(new Person("Ugo", "Cesena", 20000, "Secretary"));
    list.add(new Person("Marco", null, 4000, "Contractor"));

    // formattazione a singolo o doppio indent...
    final double result = list.stream()
        .filter(p -> p.getCity().isPresent())
        .filter(p -> p.getCity().get().equals("Cesena"))
        .map(p -> p.getIncome())
        .reduce((a, b) -> a + b)
        .get();

    System.out.println(result);

    // alternativa con iteratore: qual è la più leggibile?
    double res2 = 0.0;
    for (final Person p : list) {
      if (p.getCity().isPresent() && p.getCity().get().equals("Cesena")) {
        System.out.println(p);
        res2 = res2 + p.getIncome();
      }
    }
    System.out.println(res2);
  }
}
```

La libreria degli Stream

Struttura

- Package `java.util.stream`: interfacce e classi per gli stream
- Interfaccia `Stream<X>`: stream e metodi statici di "factory"
- Interfaccia `BaseStream<X,B>`: sopra-interfaccia di `Stream` con i metodi base
- Interfaccia `DoubleStream`: stream di `double`, con metodi base e specifici
- Interfacce `IntStream`, `LongStream`: simili
- Interfaccia `Collector<T,A,R>`: rappresenta una operazione di rid uzione
- Classe `Collectors`: fornisce una serie di collettori
- . . . altre classi di Java creano degli stream

Le collection generano Stream!

```
1  public interface Collection<E> extends Iterable<E> {
2
3      ..
4
5      Iterator<E> iterator();
6
7      default boolean removeIf(Predicate<? super E> filter) {..}
8
9      default Spliterator<E> spliterator() {..}
10
11     default Stream<E> stream() {..}
12
13     default Stream<E> parallelStream() {..}
14 }
```

L'interfaccia java.util.BaseStream

```
1  public interface BaseStream<T, S extends BaseStream<T, S>> extends ... {
2
3      // Torna un iteratore sugli elementi rimasti dello stream, e lo chiude
4      Iterator<T> iterator();
5
6      // spliterator è un iteratore che supporta parallelismo..
7      Spliterator<T> spliterator();
8
9      // è uno stream gestibili in modalità parallela
10     boolean isParallel();
11
12     // torna una variante sequenziale dello stream
13     S sequential();
14
15     // torna una variante parallela dello stream
16     S parallel();
17
18     // torna una variante non ordinata dello stream
19     S unordered();
20
21     // associa un handler chiamato alla chiusura dello stream
22     S onClose(Runnable closeHandler);
23
24     void close();
25 }
```

Riassunto delle funzionalità di una pipeline per `Stream<X>`

Creazione

- `empty`, `of`, `iterate`, `generate`, `concat`

Trasformazione

- `filter`, `map`, `flatMap`, `distinct`, `sorted`, `peek`, `limit`, `skip`, `mapToXYZ`,...

Terminazione

- `forEach`, `forEachOrdered`, `toArray`, `reduce`, `collect`, `min`, `max`, `count`, `anyMatch`, `allMatch`, `noneMatch`, `findFirst`, `findAny`,...

Una nota sulle classi `DoubleStream` e simili

- sono più specializzate e performanti, non avendo il boxing
- non hanno tutte le funzionalità di cui sopra, se vi servono vi dovete riportare ad un `Stream<X>` con un trasformatore `mapToObj()` o `boxed()`
- ne hanno qualcuna in più specifica, ad esempio `sum`

java.util.Stream: costruzione stream, 1/3

```
1  public interface Stream<T> extends BaseStream<T, Stream<T>> {
2
3      // Static factories
4
5      public static<T> Stream<T> empty() {..}
6      public static<T> Stream<T> of(T t) {..}
7      public static<T> Stream<T> of(T... values) {..}
8      public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f) {..}
9      public static<T> Stream<T> generate(Supplier<T> s) {..}
10     public static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b) {..}
11     // also recall method Collection.stream() and Collection.parallelStream()
12
13     public static<T> Builder<T> builder() {..}
14
15     public interface Builder<T> extends Consumer<T> {
16
17         void accept(T t);
18
19         default Builder<T> add(T t) {
20             accept(t);
21             return this;
22         }
23
24         Stream<T> build();
25     }
```

java.util.Stream: trasformazione stream, 2/3

```
1
2     // Stream transformation
3
4     Stream<T> filter(Predicate<? super T> predicate);
5     <R> Stream<R> map(Function<? super T, ? extends R> mapper);
6     <R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
7     Stream<T> distinct();
8     Stream<T> sorted();
9     Stream<T> sorted(Comparator<? super T> comparator);
10    Stream<T> peek(Consumer<? super T> action);
11    Stream<T> limit(long maxSize);
12    Stream<T> skip(long n);
13
14    IntStream mapToInt(ToIntFunction<? super T> mapper);
15    LongStream mapToLong(ToLongFunction<? super T> mapper);
16    DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper);
17    IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper);
18    LongStream flatMapToLong(Function<? super T, ? extends LongStream> mapper);
19    DoubleStream flatMapToDouble(Function<? super T, ? extends DoubleStream> mapper);
```

java.util.Stream: terminazione stream, 3/3

```
1     // Terminal Operations
2
3     void forEach(Consumer<? super T> action);
4     void forEachOrdered(Consumer<? super T> action);
5     Object[] toArray();
6     <A> A[] toArray(IntFunction<A[]> generator);
7     T reduce(T identity, BinaryOperator<T> accumulator);
8     Optional<T> reduce(BinaryOperator<T> accumulator);
9     <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U>
       combiner);
10    <R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<
       R, R> combiner);
11    <R, A> R collect(Collector<? super T, A, R> collector);
12    Optional<T> min(Comparator<? super T> comparator);
13    Optional<T> max(Comparator<? super T> comparator);
14    long count();
15    boolean anyMatch(Predicate<? super T> predicate);
16    boolean allMatch(Predicate<? super T> predicate);
17    boolean noneMatch(Predicate<? super T> predicate);
18    Optional<T> findFirst();
19    Optional<T> findAny();
20
21 }
```

Trasformazioni di Stream: esempi

```java
public class UseTransformations {

  public static void main(String[] args) {
    final List<Integer> li = List.of(10, 20, 30, 5, 6, 7, 10, 20, 100);

    System.out.print("All\t\t :");
    li.stream()
        .forEach(i -> System.out.print(" " + i));

    System.out.print("\nFilter(>10)\t :");
    li.stream()
        .filter(i -> i > 10) // fa passare solo certi elementi
        .forEach(i -> System.out.print(" " + i));

    System.out.print("\nMap(N:i+1)\t :");
    li.stream()
        .map(i -> "N:" + (i + 1)) // trasforma ogni elemento
        .forEach(i -> System.out.print(" " + i));

    System.out.print("\nflatMap(i,i+1)\t :");
    li.stream()
        .flatMap(i -> List.of(i, i + 1).stream()) // trasforma e appiattisce
        .map(String::valueOf) // invece del forEach..
        .map(" "::concat)
        .forEach(System.out::print);
  }

}
```

Trasformazioni di Stream: esempi pt.2

```java
public class UseTransformations2 {

  public static void main(String[] args) {
    final List<Integer> li = List.of(10, 20, 30, 5, 6, 7, 10, 20, 100);

    System.out.print("\nDistinct\t :");
    li.stream().distinct() // elimina le ripetizioni
        .forEach(i -> System.out.print(" " + i));

    System.out.print("\nSorted(down)\t :");
    li.stream().sorted((i, j) -> j - i) // ordina
        .forEach(i -> System.out.print(" " + i));

    System.out.print("\nPeek(.)\t\t :");
    li.stream().peek(i -> System.out.print(".")) // una azione ognuno
        .forEach(i -> System.out.print(" " + i));

    System.out.print("\nLimit(5)\t :");
    li.stream().limit(5) // solo i primi 5
        .forEach(i -> System.out.print(" " + i));

    System.out.print("\nSkip(5)\t\t :");
    li.stream().skip(5) // salta i primi 5
        .forEach(i -> System.out.print(" " + i));
  }
}
```

Creazione di Stream: esempi

```java
public class UseFactories {

  public static void main(String[] args) {

    final List<Integer> li = List.of(10,20,30,5,6,7,10,20,100);
    System.out.print("Collection: ");
    li.stream()
        .forEach(i->System.out.print(" "+i));

    System.out.print("\nEmpty: ");
    Stream.empty()
        .forEach(i->System.out.print(" "+i));

    System.out.print("\nFromValues: ");
    Stream.of("a","b","c")
        .forEach(i->System.out.print(" "+i));

    System.out.print("\nIterate(+1): ");
    Stream.iterate(0,i->i+1)  // 0,1,2,3,...
        .limit(20)
        .forEach(i->System.out.print(" "+i));
  }
}
```

Creazione di Stream: esempi pt.2

```java
public class UseFactories2 {

  public static void main(String[] args) {

    System.out.print("\nSuppl(random): ");
    Stream.generate(() -> Math.random()) // rand,rand,rand,...
        .limit(5)
        .forEach(i -> System.out.print(" " + i));
    // DoubleStream.generate(()->Math.random())... stream unboxed

    System.out.print("\nConcat: ");
    Stream.concat(Stream.of("a", "b"), Stream.of(1, 2))
        .forEach(i -> System.out.print(" " + i));

    System.out.print("\nBuilder: ");
    Stream.builder()
        .add(1)
        .add(2)
        .build()
        .forEach(i -> System.out.print(" " + i));

    System.out.print("\nRange: ");
    IntStream.range(0, 20) // 0,1,..,19
        .forEach(i -> System.out.print(" " + i));
  }
}
```

Creazione di Stream: file di testo e stringhe

```java
public class UseOtherFactories {

  private final static String aDir = "/home/mirko/aula";
  private final static String aFile = "/home/mirko/aula/oop/17/Counter.java";

  public static void main(String[] args) throws Exception {

    final Path dirPath = FileSystems.getDefault().getPath(aDir);

    System.out.println("Found below "+aDir);
    Files.find(dirPath, 2, (a,b)->true).forEach(System.out::println);

    System.out.println("List directory "+aDir);
    Files.list(dirPath).forEach(System.out::println);

    final Path filePath = FileSystems.getDefault().getPath(aFile);

    System.out.println("Contenuto of "+aFile);
    Files.lines(filePath).forEach(System.out::println);

    System.out.println("Contenuto of "+aFile+" in altra codifica");
    Files.lines(filePath,StandardCharsets.ISO_8859_1).forEach(System.out::println);

    // Si veda il sorgente di BufferedReader.lines() per capire come si realizza
    // uno stream a partire da un iteratore

    System.out.println("Stream da una stringa..");
    "Hellò!".chars().mapToObj(i->(char)i).forEach(System.out::println);
  }
}
```

Terminazioni ad-hoc di Stream: esempi

```java
public class UseTerminations {

  public static void main(String[] args) {

    final List<Integer> li = List.of(10,20,30,5,6,7,10,20,100);
    System.out.print("ForEach:\t ");
    li.stream().forEach(i->System.out.print(" "+i));

    System.out.print("\nForEachOrdered: ");
    li.stream().forEachOrdered(i->System.out.print(" "+i));

    final Integer[] array = li.stream().toArray(i->new Integer[i]);
    System.out.println("\nToArray:\t "+Arrays.toString(array));

    //Integer sum = li.stream().reduce(0,(x,y)->x+y);
    final Integer sum = li.stream().reduce(0,Integer::sum);
    System.out.println("Sum:\t\t "+sum);

    //Optional<Integer> max = li.stream().max((x,y)->x-y);
    final Optional<Integer> max = li.stream().max(Integer::compare);
    System.out.println("Max:\t\t "+max);

    final long count = li.stream().count();
    System.out.println("Count:\t\t "+count);

    final boolean anyMatch = li.stream().anyMatch(x -> x==100);
    System.out.println("AnyMatch:\t "+anyMatch);

    final Optional<Integer> findAny = li.stream().findAny();
    System.out.println("FindAny:\t "+findAny);
  }
}
```

Terminazione generalizzata con Stream.collect

```java
public class UseGeneralizedCollectors {

  public static void main(String[] args) {

    final List<Integer> li = List.of(10,20,30,5,6,7,10,20,100);

    // Uso collect a tre argomenti
    final Set<Integer> set = li.stream().collect(
        ()->new HashSet<>(),     // oggetto collettore
        (h,i)->h.add(i),       // aggiunta di un elemento
        (h,h2)->h.addAll(h2));    // concatenazione due collettori
    System.out.println("Set: "+set);    // un HashSet coi valori dello stream

    // Più frequente: uso collect passandogli un collettore general-purpose
    final Set<Integer> set2 = li.stream().collect(Collector.of(
        HashSet::new,               // oggetto collettore
        HashSet::add,              // aggiunta di un elemento
        (h,h2)->{h.addAll(h2); return h;}));  // concatenazione due collettori
    System.out.println("Set: "+set2);

    // cosa fa questo collettore? (.. un po' complicato)
    final int res=li.stream().collect(Collector.of(
        ()->Arrays.<Integer>asList(0),        // oggetto collettore
        (l,i)->l.set(0,i+l.get(0)),         // aggiunta di un elemento
        (l,l2)->{l.set(0,l.get(0)+l2.get(0)); return l;}))  // concatenazione
        .get(0);                    // estrazione risultato
    System.out.println("Res: "+res);

  }
}
```

Collettori di libreria: la classe Collectors

```
1  class Collectors { // some methods, all public static and generic..
2
3      Collector<T, ?, C> toCollection(Supplier<C> collectionFactory)
4      Collector<T, ?, List<T>> toList()
5      Collector<T, ?, Set<T>> toSet()
6
7      Collector<CharSequence, ?, String> joining(CharSequence delimiter,
8                                                 CharSequence prefix,
9                                                 CharSequence suffix)
10
11     Collector<T, ?, Optional<T>>  minBy(Comparator<? super T> comparator)
12     Collector<T, ?, Optional<T>>  maxBy(Comparator<? super T> comparator)
13
14     Collector<T, ?, Integer> summingInt(ToIntFunction<? super T> mapper)
15     Collector<T, ?, Long> summingLong(ToLongFunction<? super T> mapper)
16
17     Collector<T, ?, Map<K, List<T>>> groupingBy(Function<? super T, ? extends K>
        classifier)
18
19     Collector<T, ?, Map<K, D>> groupingBy(Function<? super T, ? extends K> classifier,
20                                           Collector<? super T, A, D> downstream)
21
22     Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,
23                                     Function<? super T, ? extends U> valueMapper)
24
25     Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,
26                                     Function<? super T, ? extends U> valueMapper,
27                                     BinaryOperator<U> mergeFunction)
28
29     Collector<T, ?, DoubleSummaryStatistics>
30                                 summarizingDouble(ToDoubleFunction<? super T> mapper)
31 }
```

UseCollectors: collettori di base

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

public class UseCollectors {

  public static void main(String[] args) {

    final List<Integer> li = List.of(10,20,30,5,6,7,10,20,100);

    // una List
    System.out.println(li.stream().collect(toList()));
    // un Set
    System.out.println(li.stream().collect(toSet()));
    // un TreeSet
    System.out.println(li.stream().collect(toCollection(TreeSet::new)));

    System.out.println(li.stream().collect(minBy(Integer::compare)));

    System.out.println(li.stream().collect(summingInt(Number::intValue)).
     toString());

    System.out.println(li.stream().map(i->i.toString())
                                  .collect(joining(",","(",")")));
    // (10,20,30,5,6,7,10,20,100)

  }
}
```

UseCollectors: collettori di base pt 2

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

public class UseCollectors2 {

  public static void main(String[] args) {

    final List<Integer> li = List.of(10, 20, 30, 5, 6, 7, 10, 20, 100);

    final Map<Integer, List<Integer>> map = li.stream()
        .collect(groupingBy(x -> x / 10));
    System.out.println(map); // {0=[5,6,7], 1=[10,10], ..}

    final Map<Boolean, Optional<Integer>> map2 = li.stream()
        .collect(groupingBy(x -> x % 2 == 0, minBy(Integer::compare)));
    System.out.println(map2); // minimo dei pari e minimo dei dispari

    final Map<Integer, Integer> map3 = li.stream()
        .distinct()
        .collect(toMap(x -> x, x -> x + 1));
    System.out.println(map3); // mappa ogni x in x+1

    final Map<Integer, Integer> map4 = li.stream()
        .collect(toMap(x -> x / 10, x -> x, (x, y) -> x + y));
    System.out.println(map4); // somma degli elementi in ogni decina

    System.out.println(li.stream()
        .collect(summarizingInt(Number::intValue)).toString());

  }
}
```

Esempi avanzati su Person

```java
public class UseStreamsOnPerson2 {

  public static void main(String[] args) {

    final List<Person> list = new ArrayList<>();
    list.add(new Person("Mario", "Cesena", 20000, "Teacher"));
    list.add(new Person("Rino", "Forlì", 30000, "Football player"));
    list.add(new Person("Lino", "Cesena", 110000, "Chef", "Artist"));
    list.add(new Person("Ugo", "Cesena", 20000, "Secretary"));
    list.add(new Person("Marco", null, 4000, "Contractor"));

    // Jobs of people from Cesena
    final String res = list.stream()
        .filter(p -> p.getCity().filter(x -> x.equals("Cesena")).isPresent())
        .flatMap(p -> p.getJobs().stream())
        .distinct()
        .collect(Collectors.joining("|", "[[", "]]"));
    System.out.println(res);

    // Average income of chefs
    final double avg = list.stream()
        .filter(p -> p.getJobs().contains("Chef"))
        .mapToDouble(Person::getIncome)
        .average().getAsDouble();

    System.out.println(avg);
    System.out.println(list.stream()
        .filter(p -> p.getJobs().contains("Chef"))
        .mapToDouble(Person::getIncome)
        .average());
  }
}
```

Algoritmi funzionali – cosa realizzano?

```java
public class UseStreamsForAlgorithms {

  public static void main(String[] args) {
    System.out.println(LongStream.iterate(2, x -> x + 1)
        .filter((i) -> LongStream.range(2, i / 2 + 1).noneMatch(j -> i % j == 0))
        .limit(1000)
        .mapToObj(String::valueOf)
        .collect(Collectors.joining(",", "[", "]")));

    final Random r = new Random();
    System.out.println(IntStream.range(0, 10000)
        .map(i -> r.nextInt(6) + r.nextInt(6) + 2)
        .boxed() // da int a Integer
        .collect(groupingBy(x -> x,
            collectingAndThen(counting(),
            d -> d / 10000.0))));
    System.out.println(
        "Prova di testo: indovina cosa produce la seguente computazione....."
            .chars()
            .mapToObj(x -> String.valueOf((char) x))
            .collect(groupingBy(x -> x, counting()))
            .entrySet()
            .stream()
            .sorted((e1, e2) -> -Long.compare(e1.getValue(), e2.getValue()))
            .limit(3)
            .map(String::valueOf)
            .collect(Collectors.joining(",", "[", "]")));
  }
}
```

Outline

Stream e parallelismo

Uno dei vantaggi degli stream

- È possibile gestire in modo completamente automatico il parallelismo, e quindi la disponibilità di più core
- Alcuni stream (ossia non tutti) ammettono implementazioni multi-threaded efficienti
  - `range`, collezioni... ma non `iterate`
- Gli effettivi vantaggi (o addirittura penalizzazioni!) dipendono da molti fattori
  - Implementazione corrente dell'API, tipo di stream, tipi di computazioni
  - Sempre meglio verificare se c'è vantaggio prima di usarli

Come si abilita il parallelismo (ove possibile)?

- Basta richiamare il metodo `parallel()` su uno stream...
- ...o il metodo `parallelStream()` da una collection

Test concurrency

```java
public class TestConcurrency {

  private final static int DIM = 1_000_000; // ~6.5 gain on me in 2024!!
  private final static int STEPS = 1_000;

  public static void main(String[] args) {
    long time;
    long time2;
    final List<Double> l = IntStream.range(0, DIM)
        .mapToObj(x -> Math.random())
        .collect(toList());

    time = System.currentTimeMillis();
    IntStream.range(0, STEPS).forEach(
        i -> l.stream().collect(Collectors.averagingDouble(x -> x)));
    time = System.currentTimeMillis() - time;
    System.out.println("Time: " + time);

    time2 = System.currentTimeMillis();
    IntStream.range(0, STEPS).forEach(
        i -> l.stream().parallel().collect(Collectors.averagingDouble(x -> x
    )));
    time2 = System.currentTimeMillis() - time2;
    System.out.println("Time2: " + time2);
    System.out.println("Gain: " + (((double) time) / time2));
  }
}
```

Considerazioni sul guadagno di performance – dati molto variabili...

Nel caso specifico (PC 6 cores), esecuzione ripetuta 1′000 volte

- Con 10′000 elementi nella collection, rapporto circa 0.9 (più lento)
- Con 100′000 elementi nella collection, rapporto circa 3.7
- Con 1′000′000 elementi nella collection, circa 6.5

Indicazioni generali

- Usare parallelismo solo con istanze di dimensione significativa
- È bene se le lambda usate nelle trasformazioni sono semplici
- Verificate sempre prima di usare il parallelismo se conviene

Ma come sono implementati internamente questi stream?

Elementi

- Qualche informazione è deducibile velocemente dal codice sorgente dell'API
- Gli stream sorgente incapsulano uno `Spliterator` – un `Iterator` con funzionalità aggiuntive per supportare il parallelismo
- Gli stream trasformatori sono decoratori degli stream a monte – vedere `java.util.stream.AbstractPipeline`
- Lo stream a valle innesca la chiamata a catena degli elementi negli stream precedenti, uno alla volta (possibilmente in modo lazy).

Una considerazione

- Sarebbe interessante potersi costruire le proprie classi per gli stream, ma la maggior parte delle funzionalità di basso livello utili/necessarie non sono pubbliche

Interfaccia Spliterator

```
1  public interface Spliterator<T> {
2
3      boolean tryAdvance(Consumer<? super T> action);
4
5      default void forEachRemaining(Consumer<? super T> action) {..}
6
7      Spliterator<T> trySplit();
8
9      long estimateSize();
10
11     default long getExactSizeIfKnown() {..}
12
13     int characteristics();
14
15     default boolean hasCharacteristics(int characteristics) {..}
16
17     default Comparator<? super T> getComparator() {
18         throw new IllegalStateException();
19     }
20
21     public static final int ORDERED    = 0x00000010;
22     public static final int DISTINCT   = 0x00000001;
23     public static final int SORTED     = 0x00000004;
24     public static final int SIZED      = 0x00000040;
25     public static final int NONNULL    = 0x00000100;
26     public static final int IMMUTABLE  = 0x00000400;
27     public static final int CONCURRENT = 0x00001000;
28     public static final int SUBSIZED = 0x00004000;
29 }
```

Esempio: Streams.RangeSpliterator

```
1  static final class RangeIntSpliterator implements Spliterator.OfInt {
2
3          private int from;            // initial
4          private final int upTo;      // final
5          private int last;      // current
6          ..
7          public boolean tryAdvance(IntConsumer consumer) {
8              Objects.requireNonNull(consumer);
9              final int i = from;
10             if (i < upTo) {
11                 from++;
12                 consumer.accept(i);
13                 return true;
14             } else if (last > 0) {
15                 last = 0;
16                 consumer.accept(i);
17                 return true;
18             }
19             return false;
20         }
21
22         public long estimateSize() {
23             return ((long) upTo) - from + last;
24         }
25
26         public Spliterator.OfInt trySplit() {
27             long size = estimateSize();
28             return size <= 1
29                     ? null
30                     : new RangeIntSpliterator(from, from = from + splitPoint(size), 0);
31         }
32 }
```