

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. 2022/2023

Goal della lezione

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

1 Stream

2 Implementazione Stream e Concorrenza

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 Iterator

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

```
1 public class Person {
2
3     private final String name;
4     private final Optional<String> city;
5     private final double income;
6     private final Set<String> jobs;
7
8     public Person(String name, String city, double income, String... jobs) {
9         this.name = Objects.requireNonNull(name);
10        this.city = Optional.ofNullable(city); // null in ingresso indica città assente
11        this.income = income;
12        this.jobs = new HashSet<>(List.of(jobs)); // conversione a set
13    }
14
15    public String getName() {
16        return this.name;
17    }
18
19    public Optional<String> getCity() {
20        return this.city;
21    }
22
23    public double getIncome() {
24        return this.income;
25    }
26
27    public Set<String> getJobs() {
28        return Collections.unmodifiableSet(this.jobs); // copia difensiva
29    }
```

Realizzazione dell'esempio in Java 8

```
1 public class UseStreamsOnPerson {
2
3     public static void main(String[] args) {
4
5         final List<Person> list = new ArrayList<>();
6         list.add(new Person("Mario", "Cesena", 20000, "Teacher"));
7         list.add(new Person("Rino", "Forli", 30000, "Football player"));
8         list.add(new Person("Lino", "Cesena", 110000, "Chef", "Artist"));
9         list.add(new Person("Ugo", "Cesena", 20000, "Secretary"));
10        list.add(new Person("Marco", null, 4000, "Contractor"));
11
12        // formattazione a singolo o doppio indent...
13        final double result = list.stream()
14            .filter(p -> p.getCity().isPresent())
15            .filter(p -> p.getCity().get().equals("Cesena"))
16            .map(p -> p.getIncome())
17            .reduce((a, b) -> a + b)
18            .get();
19
20        System.out.println(result);
21
22        // alternativa con iteratore: qual è la più leggibile?
23        double res2 = 0.0;
24        for (final Person p : list) {
25            if (p.getCity().isPresent() && p.getCity().get().equals("Cesena")) {
26                System.out.println(p);
27                res2 = res2 + p.getIncome();
28            }
29        }
30        System.out.println(res2);
31    }
32 }
```

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 riduzione
- 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 // Stream transformation
2
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

```
1 public class UseTransformations {
2
3     public static void main(String[] args) {
4         final List<Integer> li = List.of(10, 20, 30, 5, 6, 7, 10, 20, 100);
5
6         System.out.print("All\t\t :");
7         li.stream()
8             .forEach(i -> System.out.print(" " + i));
9
10        System.out.print("\nFilter(>10)\t :");
11        li.stream()
12            .filter(i -> i > 10) // fa passare solo certi elementi
13            .forEach(i -> System.out.print(" " + i));
14
15        System.out.print("\nMap(N:i+1)\t :");
16        li.stream()
17            .map(i -> "N:" + (i + 1)) // trasforma ogni elemento
18            .forEach(i -> System.out.print(" " + i));
19
20        System.out.print("\nflatMap(i,i+1)\t :");
21        li.stream()
22            .flatMap(i -> List.of(i, i + 1).stream()) // trasforma e appiattisce
23            .map(String::valueOf) // invece del forEach..
24            .map(" " :: concat)
25            .forEach(System.out::print);
26    }
27
28 }
```

Trasformazioni di Stream: esempi pt.2

```
1 public class UseTransformations2 {
2
3     public static void main(String[] args) {
4         final List<Integer> li = List.of(10, 20, 30, 5, 6, 7, 10, 20, 100);
5
6         System.out.print("\nDistinct\t :");
7         li.stream().distinct() // elimina le ripetizioni
8             .forEach(i -> System.out.print(" " + i));
9
10        System.out.print("\nSorted(down)\t :");
11        li.stream().sorted((i, j) -> j - i) // ordina
12            .forEach(i -> System.out.print(" " + i));
13
14        System.out.print("\nPeek(.)\t\t :");
15        li.stream().peek(i -> System.out.print(".")) // una azione ognuno
16            .forEach(i -> System.out.print(" " + i));
17
18        System.out.print("\nLimit(5)\t :");
19        li.stream().limit(5) // solo i primi 5
20            .forEach(i -> System.out.print(" " + i));
21
22        System.out.print("\nSkip(5)\t\t :");
23        li.stream().skip(5) // salta i primi 5
24            .forEach(i -> System.out.print(" " + i));
25    }
26 }
```


Creazione di Stream: esempi

```
1 public class UseFactories {
2
3     public static void main(String[] args) {
4
5         final List<Integer> li = List.of(10,20,30,5,6,7,10,20,100);
6         System.out.print("Collection: ");
7         li.stream()
8             .forEach(i->System.out.print(" "+i));
9
10        System.out.print("\nEmpty: ");
11        Stream.empty()
12            .forEach(i->System.out.print(" "+i));
13
14        System.out.print("\nFromValues: ");
15        Stream.of("a","b","c")
16            .forEach(i->System.out.print(" "+i));
17
18        System.out.print("\nIterate(+1): ");
19        Stream.iterate(0,i->i+1) // 0,1,2,3,...
20            .limit(20)
21            .forEach(i->System.out.print(" "+i));
22    }
23 }
```

Creazione di Stream: esempi pt.2

```
1 public class UseFactories2 {
2
3     public static void main(String[] args) {
4
5         System.out.print("\nSuppl(random): ");
6         Stream.generate(() -> Math.random()) // rand,rand,rand,...
7             .limit(5)
8             .forEach(i -> System.out.print(" " + i));
9         // DoubleStream.generate(()->Math.random())... stream unboxed
10
11        System.out.print("\nConcat: ");
12        Stream.concat(Stream.of("a", "b"), Stream.of(1, 2))
13            .forEach(i -> System.out.print(" " + i));
14
15        System.out.print("\nBuilder: ");
16        Stream.builder()
17            .add(1)
18            .add(2)
19            .build()
20            .forEach(i -> System.out.print(" " + i));
21
22        System.out.print("\nRange: ");
23        IntStream.range(0, 20) // 0,1,..,19
24            .forEach(i -> System.out.print(" " + i));
25    }
26 }
```

Creazione di Stream: file di testo e stringhe

```
1 public class UseOtherFactories {
2
3     private final static String aDir = "/home/mirko/aula";
4     private final static String aFile = "/home/mirko/aula/oop/17/Counter.java";
5
6     public static void main(String[] args) throws Exception {
7
8         final Path dirPath = FileSystems.getDefault().getPath(aDir);
9
10        System.out.println("Found below "+aDir);
11        Files.find(dirPath, 2, (a,b)->true).forEach(System.out::println);
12
13        System.out.println("List directory "+aDir);
14        Files.list(dirPath).forEach(System.out::println);
15
16        final Path filePath = FileSystems.getDefault().getPath(aFile);
17
18        System.out.println("Contenuto of "+aFile);
19        Files.lines(filePath).forEach(System.out::println);
20
21        System.out.println("Contenuto of "+aFile+" in altra codifica");
22        Files.lines(filePath, StandardCharsets.ISO_8859_1).forEach(System.out::println);
23
24        // Si veda il sorgente di BufferedReader.lines() per capire come si realizza
25        // uno stream a partire da un iteratore
26
27        System.out.println("Stream da una stringa..");
28        "Hellò!".chars().mapToObj(i->(char)i).forEach(System.out::println);
29    }
30 }
```

Terminazioni ad-hoc di Stream: esempi

```
1 public class UseTerminations {
2
3     public static void main(String[] args) {
4
5         final List<Integer> li = List.of(10,20,30,5,6,7,10,20,100);
6         System.out.print("ForEach:\t ");
7         li.stream().forEach(i->System.out.print(" "+i));
8
9         System.out.print("\nForEachOrdered: ");
10        li.stream().forEachOrdered(i->System.out.print(" "+i));
11
12        final Integer[] array = li.stream().toArray(i->new Integer[i]);
13        System.out.println("\nToArray:\t "+Arrays.toString(array));
14
15        //Integer sum = li.stream().reduce(0,(x,y)->x+y);
16        final Integer sum = li.stream().reduce(0,Integer::sum);
17        System.out.println("Sum:\t\t "+sum);
18
19        //Optional<Integer> max = li.stream().max((x,y)->x-y);
20        final Optional<Integer> max = li.stream().max(Integer::compare);
21        System.out.println("Max:\t\t "+max);
22
23        final long count = li.stream().count();
24        System.out.println("Count:\t\t "+count);
25
26        final boolean anyMatch = li.stream().anyMatch(x -> x==100);
27        System.out.println("AnyMatch:\t "+anyMatch);
28
29        final Optional<Integer> findAny = li.stream().findAny();
30        System.out.println("FindAny:\t "+findAny);
31    }
32 }
```

Terminazione generalizzata con Stream.collect

```
1 public class UseGeneralizedCollectors {
2
3     public static void main(String[] args) {
4
5         final List<Integer> li = List.of(10,20,30,5,6,7,10,20,100);
6
7         // Uso collect a tre argomenti
8         final Set<Integer> set = li.stream().collect(
9             ()->new HashSet<>(),           // oggetto collettore
10            (h,i)->h.add(i),               // aggiunta di un elemento
11            (h,h2)->h.addAll(h2));         // concatenazione due collettori
12         System.out.println("Set: "+set);   // un HashSet coi valori dello stream
13
14         // Più frequente: uso collect passandogli un collettore general-purpose
15         final Set<Integer> set2 = li.stream().collect(Collector.of(
16             HashSet::new,                  // oggetto collettore
17             HashSet::add,                  // aggiunta di un elemento
18             (h,h2)->{h.addAll(h2); return h;}); // concatenazione due collettori
19         System.out.println("Set: "+set2);
20
21         // cosa fa questo collettore? (.. un po' complicato)
22         final int res=li.stream().collect(Collector.of(
23             ()->Arrays.<Integer>asList(0), // oggetto collettore
24             (l,i)->l.set(0,i+1.get(0)),     // aggiunta di un elemento
25             (l,l2)->{l.set(0,l.get(0)+l2.get(0)); return l;})) // concatenazione
26             .get(0);                         // estrazione risultato
27         System.out.println("Res: "+res);
28     }
29 }
30 }
```

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>
18                                             classifier)
19
20     Collector<T, ?, Map<K, D>> groupingBy(Function<? super T, ? extends K> classifier,
21                                             Collector<? super T, A, D> downstream)
22
23     Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,
24                                     Function<? super T, ? extends U> valueMapper)
25
26     Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,
27                                     Function<? super T, ? extends U> valueMapper,
28                                     BinaryOperator<U> mergeFunction)
29
30     Collector<T, ?, DoubleSummaryStatistics>
31         summarizingDouble(ToDoubleFunction<? super T> mapper)
```

UseCollectors: collettori di base

```
1 import static java.util.stream.Collectors.*;
2
3 public class UseCollectors {
4
5     public static void main(String[] args) {
6
7         final List<Integer> li = List.of(10,20,30,5,6,7,10,20,100);
8
9         // una List
10        System.out.println(li.stream().collect(toList()));
11        // un Set
12        System.out.println(li.stream().collect(toSet()));
13        // un TreeSet
14        System.out.println(li.stream().collect(toCollection(TreeSet::new)));
15
16        System.out.println(li.stream().collect(minBy(Integer::compare)));
17
18        System.out.println(li.stream().collect(summingInt(Number::intValue)).
19            toString());
20
21        System.out.println(li.stream().map(i->i.toString())
22            .collect(joining(",","(",")")));
23        // (10,20,30,5,6,7,10,20,100)
24    }
25 }
```

UseCollectors: collettori di base pt 2

```
1 import static java.util.stream.Collectors.*;
2
3 public class UseCollectors2 {
4
5     public static void main(String[] args) {
6
7         final List<Integer> li = List.of(10, 20, 30, 5, 6, 7, 10, 20, 100);
8
9         final Map<Integer, List<Integer>> map = li.stream()
10             .collect(groupingBy(x -> x / 10));
11         System.out.println(map); // {0=[5,6,7], 1=[10,10], ..}
12
13         final Map<Boolean, Optional<Integer>> map2 = li.stream()
14             .collect(groupingBy(x -> x % 2 == 0, minBy(Integer::compare)));
15         System.out.println(map2); // minimo dei pari e minimo dei dispari
16
17         final Map<Integer, Integer> map3 = li.stream()
18             .distinct()
19             .collect(toMap(x -> x, x -> x + 1));
20         System.out.println(map3); // mappa ogni x in x+1
21
22         final Map<Integer, Integer> map4 = li.stream()
23             .collect(toMap(x -> x / 10, x -> x, (x, y) -> x + y));
24         System.out.println(map4); // somma degli elementi in ogni decina
25
26         System.out.println(li.stream()
27             .collect(summarizingInt(Number::intValue)).toString());
28
29     }
30 }
```


Esempi avanzati su Person

```
1 public class UseStreamsOnPerson2 {
2
3     public static void main(String[] args) {
4
5         final List<Person> list = new ArrayList<>();
6         list.add(new Person("Mario", "Cesena", 20000, "Teacher"));
7         list.add(new Person("Rino", "Forli", 30000, "Football player"));
8         list.add(new Person("Lino", "Cesena", 110000, "Chef", "Artist"));
9         list.add(new Person("Ugo", "Cesena", 20000, "Secretary"));
10        list.add(new Person("Marco", null, 4000, "Contractor"));
11
12        // Jobs of people from Cesena
13        final String res = list.stream()
14            .filter(p -> p.getCity().filter(x -> x.equals("Cesena")).isPresent())
15            .flatMap(p -> p.getJobs().stream())
16            .distinct()
17            .collect(Collectors.joining("|", "[", "]"));
18        System.out.println(res);
19
20        // Average income of chefs
21        final double avg = list.stream()
22            .filter(p -> p.getJobs().contains("Chef"))
23            .mapToDouble(Person::getIncome)
24            .average().getAsDouble();
25
26        System.out.println(avg);
27        System.out.println(list.stream()
28            .filter(p -> p.getJobs().contains("Chef"))
29            .mapToDouble(Person::getIncome)
30            .average());
31    }
32 }
```

Algoritmi funzionali – cosa realizzano?

```
1 public class UseStreamsForAlgorithms {
2
3     public static void main(String[] args) {
4         System.out.println(LongStream.iterate(2, x -> x + 1)
5             .filter((i) -> LongStream.range(2, i / 2 + 1).noneMatch(j -> i % j == 0))
6             .limit(1000)
7             .mapToObj(String::valueOf)
8             .collect(Collectors.joining(",", "[", "]")));
9
10        final Random r = new Random();
11        System.out.println(IntStream.range(0, 10000)
12            .map(i -> r.nextInt(6) + r.nextInt(6) + 2)
13            .boxed() // da int a Integer
14            .collect(groupingBy(x -> x,
15                collectingAndThen(counting(),
16                    d -> d / 10000.0))));
17        System.out.println(
18            "Prova di testo: indovina cosa produce la seguente computazione....."
19            .chars()
20            .mapToObj(x -> String.valueOf((char) x))
21            .collect(groupingBy(x -> x, counting()))
22            .entrySet()
23            .stream()
24            .sorted((e1, e2) -> -Long.compare(e1.getValue(), e2.getValue()))
25            .limit(3)
26            .map(String::valueOf)
27            .collect(Collectors.joining(",", "[", "]")));
28    }
29 }
```

1 Stream

2 Implementazione Stream e Concorrenza

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

```
1 public class TestConcurrency {
2
3     private final static int DIM = 1_000_000; // 5.72 gain on me in 2022!!
4     private final static int STEPS = 1_000;
5
6     public static void main(String[] args) {
7         long time;
8         long time2;
9         final List<Double> l = IntStream.range(0, DIM)
10             .mapToObj(x -> Math.random())
11             .collect(toList());
12
13         time = System.currentTimeMillis();
14         IntStream.range(0, STEPS).forEach(
15             i -> l.stream().collect(Collectors.averagingDouble(x -> x)));
16         time = System.currentTimeMillis() - time;
17         System.out.println("Time: " + time);
18
19         time2 = System.currentTimeMillis();
20         IntStream.range(0, STEPS).forEach(
21             i -> l.stream().parallel().collect(Collectors.averagingDouble(x -> x
22             )));
23         time2 = System.currentTimeMillis() - time2;
24         System.out.println("Time2: " + time2);
25         System.out.println("Gain: " + (((double) time) / time2));
26     }
27 }
```

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 `Splitter` – 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.RangeSplitter

```
1 static final class RangeIntSplitter implements Splitter.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 Splitter.OfInt trySplit() {
27        long size = estimateSize();
28        return size <= 1
29            ? null
30            : new RangeIntSplitter(from, from = from + splitPoint(size), 0);
31    }
32 }
```