Metodologie di Programmazione: Gli stream

Roberto Navigli

Dipartimento
di Informatica

Sapienza
Università di Roma

java.util.Optional come contenitore di riferimenti

- java.util.Optional è un contenitore di un riferimento che potrebbe essere o non essere null
- Un metodo può restituire un Optional invece di restituire un riferimento potenzialmente null
- Serve a evitare i NullPointerException

Creare e verificare un java.util.Optional

- Un Optional senza riferimento (contenitore vuoto):

```
Optional.empty()
```

- Un Optional non nullo:

```
Optional.of("bumbumghigno");
```

- Un Optional di un riferimento che può essere nullo:

```
Optional<String> optional = Optional.ofNullable(s);
```

- Controllo della presenza di un valore non null:

```
Optional.empty().isPresent() == false
Optional.of ("bumbumghigno").isPresent() == true
```

Ottenere il valore di un java.util.Optional

- Mediante **orElse**

```
Optional<String> op = Optional.of("eccomi")
op.orElse("fallback"); // "eccomi"
Optional.empty().orElse("fallback"); // "fallback"
```

- Mediante **ifPresent** o **ifPresentOrElse**:

```
op.ifPresent(System.out::println); // "eccomi"
```

- Da non usare (!?#@):

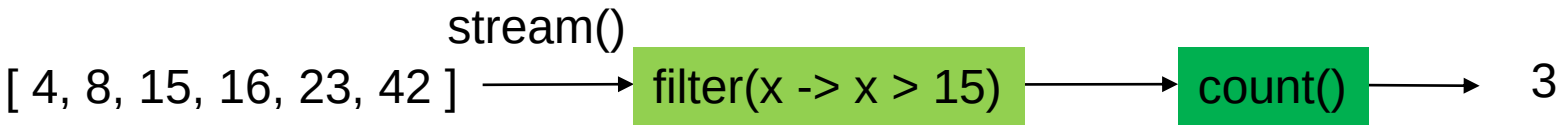
```
optional.get(); // valore o solleva eccezione se non presente
```

Un nuovo meccanismo per le sequenze di elementi: gli Stream

- Una nuova interfaccia java.util.stream.Stream
- Rappresenta una sequenza di elementi su cui possono essere effettuate una o più operazioni
- Supporta operazioni sequenziali e paralelle
- Uno Stream viene creato a partire da una sorgente di dati, ad esempio una java.util.Collection
- Al contrario delle Collection, uno Stream non memorizza né modifica i dati della sorgente, ma opera su di essi

Stream: operazioni intermedie e terminali

- Le operazioni possono essere intermedie o terminali
  - **Intermedie:** restituiscono un altro stream su cui continuare a lavorare
  - **Terminali:** restituiscono il tipo atteso



- Una volta che uno stream è stato consumato (**operazione terminale**), non può essere riutilizzato
- **Builder pattern**: si impostano una serie di operazioni per configurare (op. intermedie) e infine si costruisce l'oggetto (op. terminale)

Metodi principali dell'interfaccia java.util.stream.Stream<T>

Metodo	Tipo	Descrizione
<R, A> R **collect**(Collector<? super T, A, R> *collectorFunction*)	**T**	Raccoglie gli elementi di tipo T in un contenitore di tipo R, accumulando in oggetti di tipo A
long **count**()	**T**	Conta il numero di elementi
Stream<T> **filter**(Predicate<? super T> *predicate*)	**I**	Fornisce uno stream che contiene solo gli elementi che soddisfano il predicato
void **forEach**(Consumer<? super T> *action*)	**T**	Esegue il codice in input su ogni elemento dello stream
<R> Stream<R> **map**(Function<? super T, ? extends R> *mapFunction*)	**I**	Applica la funzione a tutti gli elementi dello stream, fornendo un nuovo stream di elementi mappati
IntStream **mapToInt/ToDouble/ToLong**(ToIntFunction<? super T> *mapFunction*)	**I**	Come sopra, ma la mappatura è su interi/ecc. (**operazione ottimizzata**)
Optional<T> **max/min**(Comparator<? super T> *comparator*)	**T**	Restituisce il massimo/minimo elemento all'interno dello stream
T **reduce**(T *identityVal*, BinaryOperator<T> *accumulator*)	**T**	Effettua l'operazione di riduzione basata sugli elementi dello stream
Stream<T> **sorted**()	**I**	Produce un nuovo stream di elementi ordinati
Stream<T> **limit**(long k)	**I**	Limita lo stream a k elementi

Stream: operazioni intermedie e terminali (2)

- Una volta che uno stream è stato consumato (**operazione terminale**), non può essere riutilizzato
- Comportamento pigro (lazy behavior): Le operazioni intermedie non vengono eseguite immediatamente, ma solo quando si richiede l'esecuzione di un'operazione terminale
- Le operazioni possono essere:
  - **senza stato** (*stateless*): l'elaborazione dei vari elementi può procedere in modo indipendente (es. filter)
  - con **stato** (*stateful*): l'elaborazione di un elemento potrebbe dipendere da quella di altri elementi (es. sorted)

Stream, IntStream, DoubleStream, LongStream

- Poiché Stream opera su oggetti, esistono analoghe versioni ottimizzate per lavorare con 3 tipi primitivi:
  - Su int: IntStream
  - Su double: DoubleStream
  - Su long: LongStream
- Tutte queste interfacce estendono l'interfaccia di base BaseStream

Come ottenere uno stream?

- Direttamente dai dati: con il metodo statico generico Stream.of(elenco di dati di un certo tipo)
- In Java 8 l'interfaccia Collection è stata estesa per includere due nuovi metodi di default:

- default Stream<E> stream() – restituisce un nuovo stream sequenziale
- default Stream<E> parallelStream() – restituisce un nuovo stream parallelo, se possibile (altrimenti restituisce uno stream sequenziale)

- E' possibile ottenere uno stream anche per un array, con il metodo statico Stream<T> Arrays.stream(T[ ] *array*)
- E' possibile ottenere uno stream di righe di testo da BufferedReader.lines() oppure da Files.lines(Path)
- E' possibile ottenere uno stream di righe anche da String.lines

Stream vs. collection

- Lo stream permette di utilizzare uno **stile dichiarativo**
  - Iterazione interna sui dati
- La collection impone l'utilizzo di uno stile imperativo
  - Iterazione esterna sui dati (tranne con forEach)
- Lo stream si focalizza sulle **operazioni di alto livello** da eseguire (mattoncini o building blocks) eventualmente anche in parallelo, senza specificare **come verranno eseguite**
- **Stream:** dichiarativo, componibile, parallelizzabile

Metodi di java.util.stream.Stream: min e max

- I metodi min e max restituiscono rispettivamente il minimo e il massimo di uno stream sotto forma di Optional
- Prendono in input un Comparator sul tipo degli elementi dello stream

```
List<Integer> p = Arrays.asList(2, 3, 4, 5, 6, 7);
Optional<Integer> max = p.stream().max(Integer::compare);
// se c'è, restituisce il massimo; altrimenti restituisce -1
System.out.println(max.orElse(-1));
```

Metodi di java.util.stream.Stream: filter (intermedio), forEach (terminale)

- filter è un metodo di Stream che accetta un predicato (Predicate) per filtrare gli elementi dello stream
  - Operazione intermedia che restituisce lo stream filtrato
- forEach prende in input un Consumer e lo applica a ogni elemento dello stream
  - Operazione terminale

Esempi di Stream (con filter e forEach)

- Filtra gli elementi di una lista per iniziale e stampa ciascun elemento rimanente:

```
List<String> l = Arrays.asList("da", "ab", "ac", "bb");
l.stream()
  .filter(s -> s.startsWith("a"))
  .forEach(System.out::println);
```

- Filtra gli elementi di una lista di interi mantenendo solo quelli dispari e stampa ciascun elemento rimanente:

```
List<Integer> l = Arrays.asList(4, 8, 15, 16, 23, 42);
l.stream()
  .filter(k -> k % 2 == 1)
  .forEach(System.out::println);
```

Altro esempio di Stream (con filter e forEach)

- Filtra gli elementi di una lista per iniziale e lunghezza della stringa e stampa ciascun elemento rimanente:

```
Predicate<String> startsWithJ = s -> s.startsWith("J");
Predicate<String> fourLetterLong = s -> s.length() == 4;

List<String> l = Arrays.asList("Java", "Scala", "Lisp");
l.stream()
  .filter(startsWithJ.and(fourLetterLong))
  .forEach(s -> System.out.println("Inizia con J ed e' lungo
4 caratteri: "+s);
```

Metodi di java.util.stream.Stream: count (terminale)

- count è un'operazione terminale che restituisce il numero long di elementi nello stream
- Esempio:

```
long startsWithA = l.stream().filter(s -> s.startsWith("a")) .count();
System.out.println(startsWithA); // 2
```

- Esempio di conteggio del numero di righe di un file di testo:

```
long numberOfLines = Files.lines(Paths.get("yourFile.txt")).count();
```

Metodi di java.util.stream.Stream: sorted (intermedia)

- sorted è un'operazione intermedia sugli stream che restituisce una vista ordinata dello stream **senza modificare la collezione sottostante**
- Esempio:

```
List<String> l = Arrays.asList("da", "ac", "ab", "bb");
l.stream()
  .sorted()
  .filter(s -> s.startsWith("a"))
  .forEach(System.out::println);
```

- Stampa:

```
ab
ac
```

Metodi di java.util.stream.Stream: map (intermedia)

- map è un'operazione intermedia sugli stream che restituisce un nuovo stream in cui ciascun elemento dello stream di origine è convertito in un altro oggetto attraverso la funzione (Function) passata in input
- Esempio: restituire tutte le stringhe (portate in maiuscolo) ordinate in ordine inverso

```
// equivalente a .map(s -> s.toUpperCase())
l.stream().map(String::toUpperCase).sorted(Comparator.<String>naturalOrder().reversed()) .forEach(System.out::println);
```

- Stampa:

```
DA
BB
AC
AB
```

java.util.stream.Stream.map: esempio

- Si vuole scrivere un metodo che aggiunga l'IVA a ciascun prezzo:

```
List<Integer> ivaEsclusa = Arrays.asList(10, 20, 30);

// In Java 7:
for (int p : ivaEsclusa)
{
    double pIvaInclusa = p*1.22;
    System.out.println(pIvaInclusa);
}
// In Java 8:
ivaEsclusa.stream().map(p -> p*1.22).forEach(System.out::println);
```

Metodi di java.util.stream.Stream: collect (terminale)

- collect è un'operazione terminale che permette di raccogliere gli elementi dello stream in un qualche oggetto (ad es. una collection, una stringa, un intero)
- Ad esempio, per ottenere la lista dei prezzi ivati:

```
List<Integer> ivaEsclusa = Arrays.asList(10, 20, 30);
// In Java 7:
List<Double> l = new ArrayList<>();
for (int p : ivaEsclusa) l.add(p*1.22);
// In Java 8:
List<Double> l = ivaEsclusa.stream().map(p -> p*1.22)
                                    .collect(Collectors.toList());
```

Esempio: creare una stringa che concatena stringhe in un elenco, rese maiuscole e separate da virgola

```
List<String> l = Arrays.asList("RoMa", "milano", "Torino");
String s = "";

// in Java 7:
for (String e : l) s += e.toUpperCase()+", ";
s = s.substring(0, s.length()-2);

// in Java 8:
s = l.stream().map(e -> e.toUpperCase())
              .collect(Collectors.joining(", "));
```

Esempio: trasformare una lista di stringhe in una lista delle lunghezze delle stesse

```
List<String> words = Arrays.asList("Oracle", "Java",
"Magazine");
List<Integer> wordLengths =
        words.stream()
        .map(String::length)
        .collect(toList());
```

java.util.stream.Collectors

- "Ricette" per ridurre gli elementi di uno stream e raccoglierli in qualche modo
- Per rendere più leggibile il codice: import static java.util.stream.Collectors.*
  - In questo modo possiamo scrivere il nome del metodo senza anteporre Collectors. (es. toList() invece di Collectors.toList())

java.util.stream.Collectors: riduzioni a singolo elemento

- counting() – restituisce il numero di elementi nello stream (risultato di tipo long)

```
List<Integer> l = Arrays.asList(2, 3, 5, 6);
// k == 2
long k = l.stream().filter(x -> x < 5).collect(Collectors.counting()));
```

- maxBy/minBy(comparator) – restituisce un Optional con il massimo/minimo valore

```
// max contiene 6
Optional<Integer> max = l.stream().collect(maxBy(Integer::compareTo));
```

- summingInt(lambda che mappa ogni elemento a intero)/averagingInt, summingDouble, averagingDouble

java.util.stream.Collectors: riduzioni a singolo elemento

- joining(), joining(separatore), joining(separatore, prefisso, suffisso) – concatena gli elementi stringa dello stream in un'unica stringa finale

```
List<Integer> l = Arrays.asList(2, 3, 5, 6, 2, 7);
// str.equals("2,3,5,6,2,7")
String str = l.stream().map(x -> ""+x).collect(joining(","));
```

- toList, toSet e toMap – accumulano gli elementi in una lista, insieme o mappa (non c'è garanzia sul tipo di List, Set o Map)

```
Set<String> set = l.stream().map(x -> ""+x).collect(toSet());
```

- toCollection – accumula gli elementi in una collezione scelta

```
ArrayList<String> str = l.stream().map(x -> ""+x)
                                  .collect(toCollection(ArrayList::new));
```

Collectors.toMap: riduzione a una mappa

- toMap prende in input fino a 4 argomenti:
  - la funzione per mappare l'oggetto dello stream nella chiave della mappa
  - la funzione per mappare l'oggetto dello stream nel valore della mappa
  - **opzionale:** la funzione da utilizzare per unire il valore preesistente nella mappa a fronte della chiave con il valore associato all'oggetto dalla seconda funzione (non devono trovarsi due chiavi uguali o si ottiene un'eccezione IllegalStateException)
  - **opzionale:** il Supplier che crea la mappa

```
Map<Integer, String> map = persons
    .stream()
    .collect(Collectors.toMap(
        Person::getAge,
        Person::getName,
        (name1, name2) -> name1 + ";" + name2));
```

java.util.stream.Collectors: raggruppamento di elementi

- groupingBy(lambda che mappa gli elementi di tipo T in bucket rappresentati da oggetti di qualche altro tipo S), restituisce una Map<S, List<T>>
- groupingBy(lambda, downStreamCollector), per raggruppamento multilivello

- Ad esempio, per ottenere una mappa da città a **lista** di persone a partire da una lista di persone (multimappa!):

```
Map<City, List<Person>> peopleByCity = // people è una collection di Person
people.stream().collect(groupingBy(Person::getCity));
```

- La stessa mappa, ma i cui valori siano **insiemi** di persone (multimappa!):

```
Map<City, Set<Person>> peopleByCity =
people.stream().collect(groupingBy(Person::getCity, toSet()));
```

Collectors.mapping

In raccolte multilivello, per esempio usando groupingBy, è utile mappare il valore del raggruppamento a qualche altro tipo:

```
Map<City, Set<String>> peopleSurnamesByCity =
        people.stream().collect(
                groupingBy(Person::getCity,
                        mapping(Person::getLastName, toSet())));
```

Creare il proprio Collector

- Con il metodo statico Collector.of che prende in input 4 argomenti: un supplier per creare la rappresentazione interna, l'accumulator che aggiorna la rappresentazione con il nuovo elemento, combiner che "fonde" due rappresentazione ottenute in modo parallelo e il finisher, chiamato alla fine, che trasforma il tutto nel tipo finale

```
Collector<Person, StringJoiner, String> personNameCollector
  = Collector.of(
        () -> new StringJoiner(" | "),        // supplier
        (j, p) -> j.add(p.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),             // combiner
        StringJoiner::toString);              // finisher
String names = people.stream()
                                .collect(personNameCollector);
System.out.println(names);  // MAX | PETER | PAMELA | DAVID
```

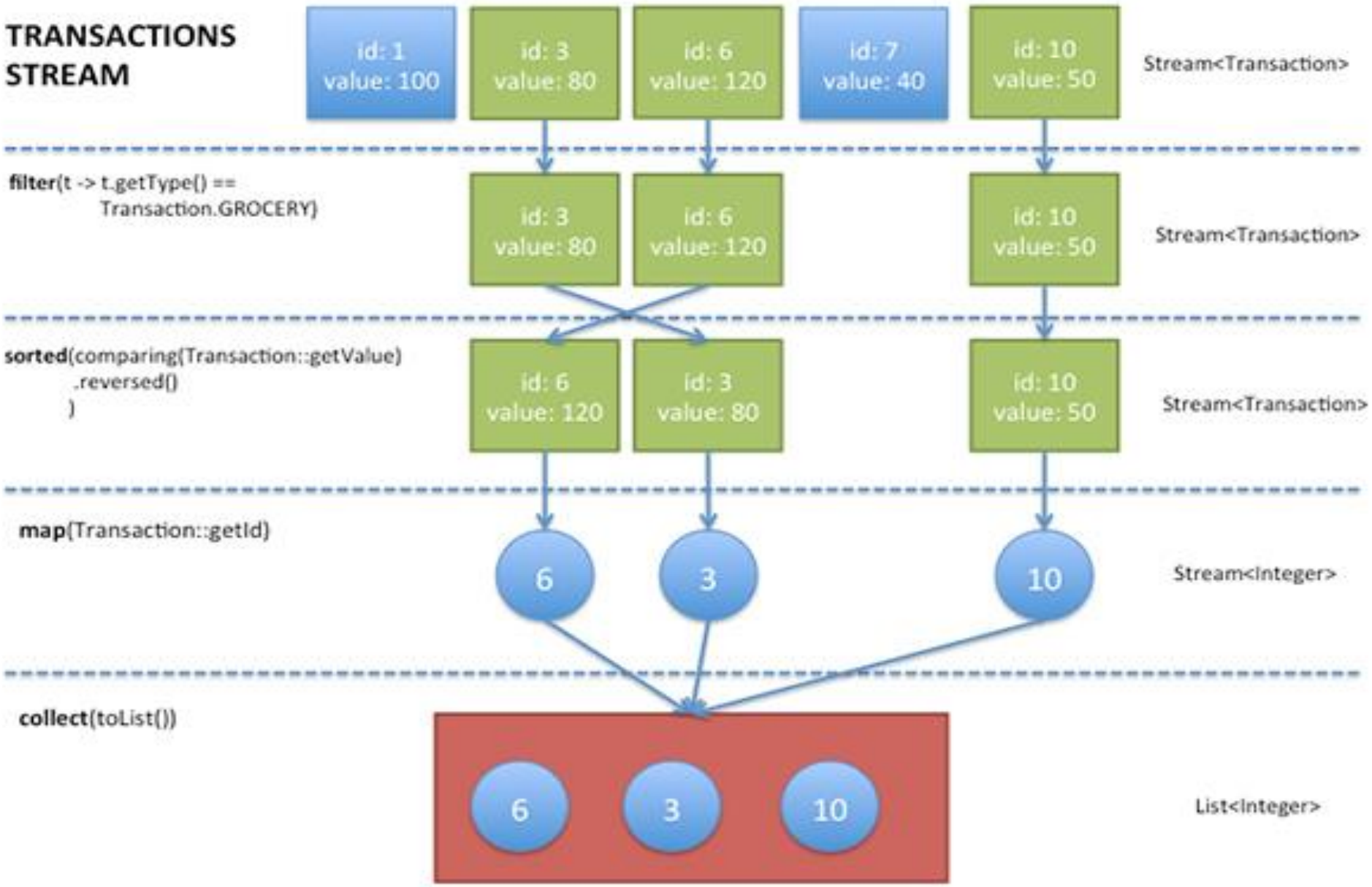
java.util.stream.Collectors: partizionamento di elementi

- partitioniningBy(predicato): raggruppa in una Map<Boolean, List<T>>

// crea una mappa da booleano a lista di interi che soddisfano quel criterio del predicato in input

```
Map<Boolean, List<Integer>> m =
l.stream().collect(Collectors.partitioningBy(x -> x % 2 == 0));
```

Esempio di stream su una classe Transaction



Metodi di java.util.stream.Stream: distinct (intermedia)

- Restituisce un nuovo stream senza ripetizione di elementi (gli elementi sono tutti distinti tra loro):

```
List<Integer> l = List.of(3, 4, 5, 3, 4, 1);
List<Integer> distinti = l.stream().map(x -> x*x)
                          .distinct().collect(Collectors.toList());
```

Metodi di java.util.stream.Stream: reduce (terminale)

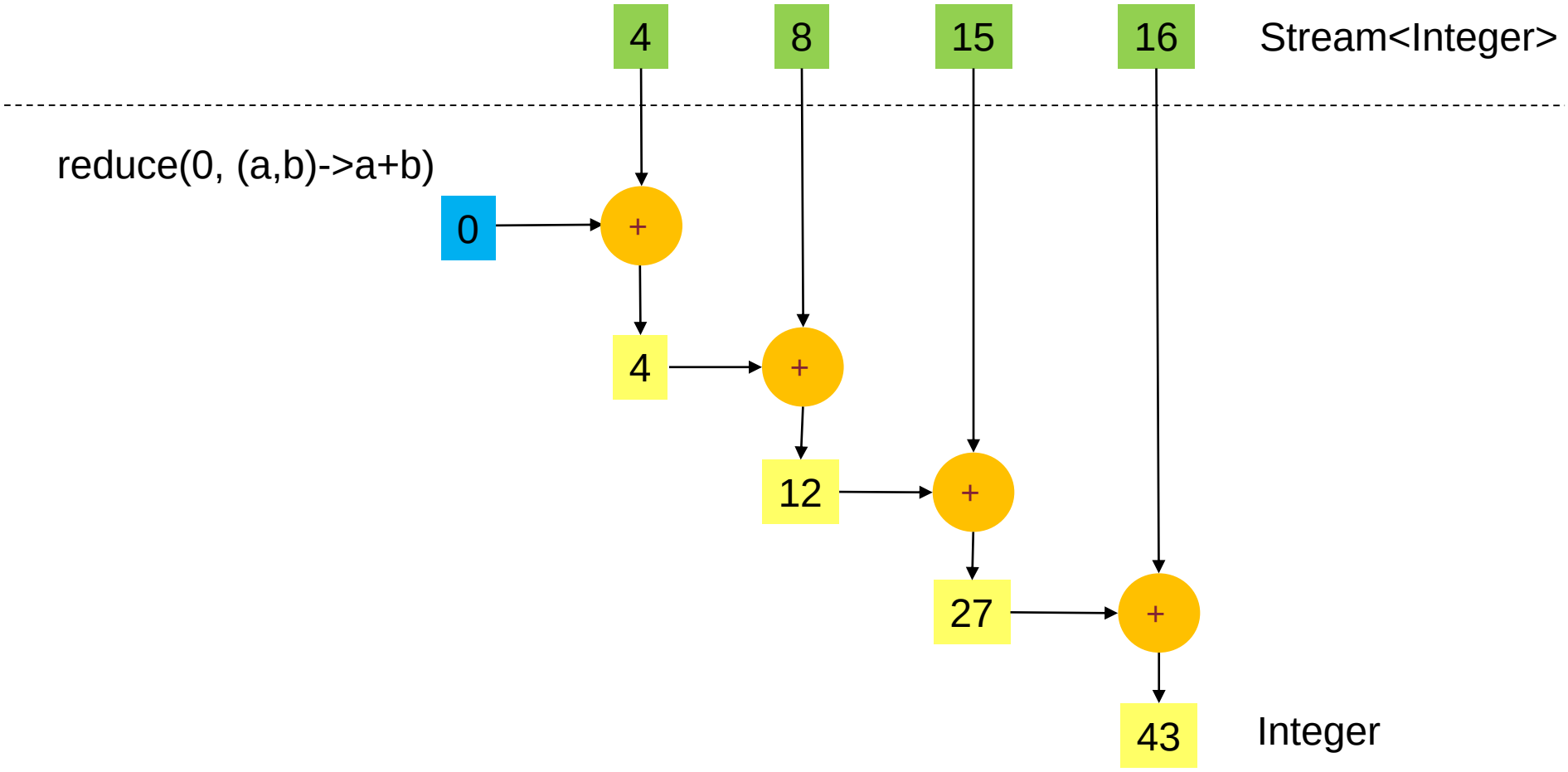
- reduce è un'operazione terminale che effettua una riduzione sugli elementi dello stream utilizzando la funzione data in input
- Vediamo come viene effettuata l'operazione di somma:

```
int somma = 0;
for (int k : lista)
    somma += k;
```

- Questa operazione può essere effettuata su uno stream mediante l'operazione reduce:

```
lista.stream().reduce(0, (a, b) -> a+b); // oppure:
lista.stream().reduce(0, Integer::sum);
```

Esecuzione della somma con reduce



Metodi di java.util.stream.Stream: reduce (terminale)

- Esiste anche una versione di reduce con un solo parametro (senza elemento identità), che restituisce un Optional<T>:

```
lista.stream().reduce(Integer::sum);
```

- Perché Optional<T>? Perché se lo stream è vuoto, non avendo l'elemento identità non si sa quale valore restituire

Metodi di java.util.stream.Stream: reduce (terminale)

- Ad esempio, riduciamo uno stream di String a una stringa costruendola elemento per elemento

```
Optional<String> reduced = l.stream().sorted()
            .reduce((s1, s2) -> s1 + "#" + s2);
reduced.ifPresent(System.out::println); // "ab#ac#bb#da"
```

Metodi di java.util.stream.Stream: reduce (terminale)

- Ad esempio, calcoliamo il prodotto tra interi in una lista:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8);
int product = numbers.stream().reduce(1, (a, b) -> a * b);
// product == 40320
```

- Ad esempio, calcoliamo il massimo tra gli interi di una lista:

```
int max = numbers.stream().reduce(
                                Integer.MIN_VALUE, Integer::max);
// max == 8
```

Esempio: calcolo della somma dei valori iva inclusa

```
List<Integer> ivaEsclusa = Arrays.asList(10, 20, 30);

// In Java 7:
double totIvaInclusa = 0.0;
for (int p : ivaEsclusa)
{
    double pIvaInclusa = p*1.22;
    totIvaInclusa += pIvaInclusa;
}
// In Java 8:
ivaEsclusa.stream().map(p -> p*1.22).reduce((sum, p) -> sum+p).orElse(0);
```

- reduce restituisce un Optional

Esercizio

- Calcolare la somma del doppio dei valori pari di una lista
- Soluzione con filter, map e reduce:

```
l.stream().filter(e -> e%2 == 0)
          .map(e -> e*2)
          .reduce(0, Integer::sum);
```

- Soluzione con filter, mapToInt e IntStream.sum:

```
l.stream().filter(e -> e%2 == 0)
          .mapToInt(e -> e*2)
          .sum();
```

Metodi di java.util.stream.Stream: limit (intermedia)

- Limita lo stream a k elementi (k long passato in input)

- Esempio:

```
List<String> elementi = List.of("uno", "due", "tre");
List<String> reduced = l.stream().limit(2).collect(toList());
// contiene ["uno", "due"]
```

Metodi di java.util.stream.Stream: skip (intermedia)

- Salta k elementi (k long passato in input)

- Esempio:

```
List<String> elementi = List.of("uno", "due", "tre");
List<String> reduced = l.stream().skip(2).collect(toList());
// contiene ["tre"]
```

Metodi di java.util.stream.Stream: takeWhile/dropWhile (intermedie)

- takeWhile prende elementi finché si verifica la condizione del predicato
- dropWhile salta gli elementi finché si verifica la condizione

- Esempio:

```
List<Integer> elementi = List.of(2, 5, 10, 42, 3, 2, 10);
List<Integer> reduced = l.stream().takeWhile(x -> x < 42).collect(toList());
// contiene [2, 5, 10]
```

Metodi di java.util.stream.Stream: anyMatch/allMatch/noneMatch (terminali)

- Gli stream espongono diverse operazioni terminali di matching e restituiscono un booleano relativo all'esito del matching
- Esempio:

```
boolean anyStartsWithA = l.stream().anyMatch(s -> s.startsWith("a"));
System.out.println(anyStartsWithA); // true
boolean allStartsWithA = l.stream().allMatch(s -> s.startsWith("a"));
System.out.println(allStartsWithA); // false
boolean noneStartsWithZ = l.stream().noneMatch(s -> s.startsWith("z"));
System.out.println(noneStartsWithZ); // true
```

Metodi di java.util.stream.Stream: findFirst e findAny (terminali)

- Gli stream espongono due operazioni terminali per ottenere il primo (findFirst) o un qualsiasi elemento (findAny) dello stream
- Esempio:

```
List<String> l2 = Arrays.asList("c", "b", "a");
// v conterrà il riferimento ad "a"
Optional<String> v = l2.stream().sorted().findFirst();
```

Esercizio: restituisci il doppio del primo valore in una lista di interi che sia pari e > 41

- Java 7:

```
int result = 0;
for (int k : numbers)
    if (k > 41 && k%2 == 0)
    {
        result = k*2;
        break;
    }
```

- Qual è il problema?
- Java 8:

```
Optional<Integer> o = numbers.stream()
    .filter(e -> e > 41)
    .filter(e -> e%2 == 0)
    .map(e -> e*2)
    .findFirst();
```

Metodi di stream: mapToInt e IntStream.summaryStatistics

- E’ possibile convertire uno Stream in un IntStream
- IntStream possiede il metodo summaryStatistics che restituisce un oggetto di tipo IntSummaryStatistics con informazioni su: min, max, media, conteggio

```
List<Integer> p = List.of(2, 3, 4, 5, 6, 7);
IntSummaryStatistics stats = p.stream().mapToInt(x -> x)
                                       .summaryStatistics();
System.out.println(stats.getMin());
System.out.println(stats.getMax());
System.out.println(stats.getAverage());
System.out.println(stats.getCount());
```

Metodi di stream: flatMap

- flatMap permette di "unire" gli stream in un unico stream

```
// restituisce uno Stream<String[]>
words.map(w -> w.split(""))
        // restituisce uno Stream<Stream<String>>
        .map(Arrays::stream);
```

- con flatMap gli stream risultanti dalla mappatura vengono sequenziati in un unico stream:

```
Map<String, Long> letterToCount =
        words.map(w -> w.split(""))    // restituisce uno String[]
                .flatMap(Arrays::stream)
                .collect(groupingBy(identity(), counting()));
```

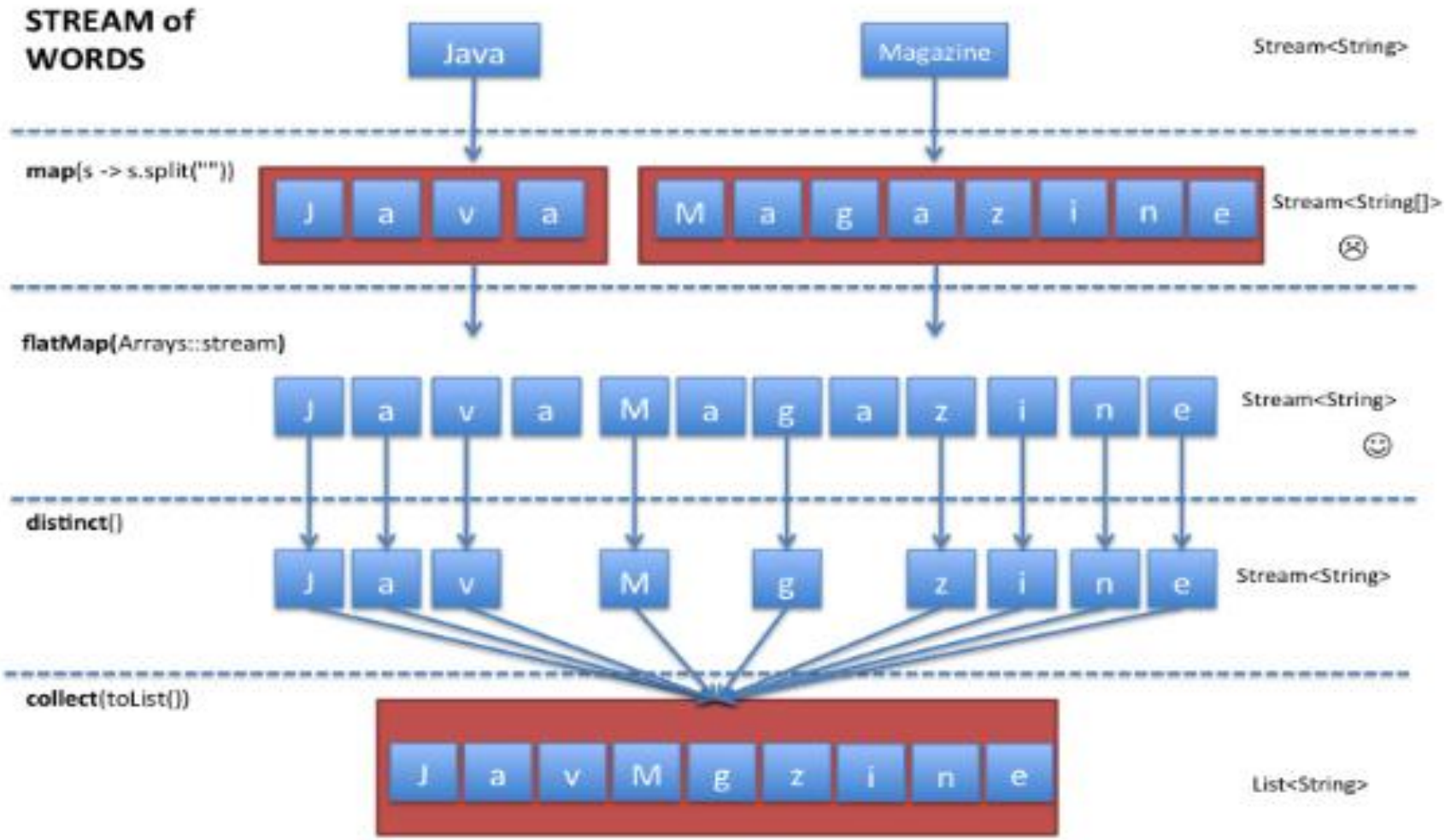
Ancora un esempio con flatMapToInt

- Supponiamo di avere una lista di stringhe
- l.stream().map(String::chars) mappa a uno stream di IntStream (quindi uno stream di stream)
- Per risolvere il problema di avere uno stream di stream, posso utilizzare flatMap (e, in particolare, essendo un IntStream, a flatMapToInt)
- l.stream().flatMapToInt(String::chars) mappa a un unico IntStream

Esempio: stampare i token (distinti) da file

```
Files.lines(Paths.get("stuff.txt"))
        .map(line -> line.split("\\s+")) // Stream<String[]>
        .flatMap(Arrays::stream) // Stream<String>
        .distinct() // Stream<String>
        .forEach(System.out::println);
```

Esempio di stream di lettere senza ripetizioni



java.util.stream.IntStream, DoubleStream e LongStream

- Si ottengono da uno Stream con i metodi mapToInt, mapToLong, mapToDouble
- Analoghi metodi sono disponibili nelle 3 classi (tolto quello del tipo in questione, es. IntStream non ha mapToInt), ma in più hanno mapToObj
- Dispongono di 2 metodi statici:
  - range(inizio, fine), intervallo esclusivo (aperto a destra)
  - rangeClosed(inizio, fine), intervallo inclusivo (chiuso a destra)
- Esempio: stampa i numeri dispari fino a 10:

```
IntStream.range(0, 10).filter(n -> n % 2 == 1)
                .forEach(System.out::println);
```

Un esempio con uno IntStream ottenuto da stream di array di interi

```
Arrays.stream(new int[] {1, 2, 3}) // restituisce un IntStream
    .map(n -> 2 * n + 1)
    .average()  // metodo di IntStream
    .ifPresent(System.out::println);  // 5.0
```

Passaggio da Stream a IntStream, LongStream, DoubleStream e viceversa

- Mediante i metodi Stream.mapToInt|Double|Long
- E, da uno stream di primitivi incapsulati, boxed o mapToObj
- Tra stream di primitivi, as**Double|Long**Stream

```
List<String> s = Arrays.asList("a", "b", "c");
s.stream()                                  // Stream<String>
    .mapToInt(String::length)               // IntStream
    .asLongStream()                         // LongStream
    .mapToDouble(x -> x / 42.0)             // DoubleStream
    .boxed()                                // Stream<Double>
    .mapToLong(x -> 1L)                     // LongStream
    .mapToObj(x -> "");                     // Stream<String>
```

Ottenere uno stream infinito

- L'interfaccia Stream espone un metodo iterate che, partendo dal primo argomento, restituisce uno stream infinito con valori successivi applicando la funzione passata come secondo argomento:

```
Stream<Integer> numbers = Stream.iterate(0, n -> n+10);
```

- E' possibile limitare uno stream infinito con il metodo limit:

```
// 0, 10, 20, 30, 40
numbers.limit(5).forEach(System.out::println);
```

Implementare l'iterazione mediante stream

- Da Java 9, l'interfaccia Stream espone un secondo metodo iterate che, partendo dal primo argomento, restituisce uno stream di valori successivi applicando la funzione passata come terzo argomento e uscendo quando il predicato del secondo argomento è false:

```
// 0, 10, 20, 30, 40
Stream<Integer> numbers = Stream.iterate(
                                    0, n -> n < 50, n -> n+10);
```

L'ordine delle operazioni conta (1)

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

Output:

```
map:  d2
filter:  D2
map:  a2
filter:  A2
forEach: A2
map:  b1
filter:  B1
map:  b3
filter:  B3
map:  c
filter:  C
```

L'ordine delle operazioni conta (2)

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

Output:

```
filter:  d2
filter:  a2
map:  a2
forEach: A2
filter:  b1
filter:  b3
filter:  c
```

- Invertendo l'ordine di filter e map la pipeline è molto più veloce

L'ordine delle operazioni conta (3): sorted è stateful!

- Aggiungiamo un ordinamento dei valori nello stream:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    }).forEach(s -> System.out.println("forEach: " + s));
```

Output:

```
sort:   a2; d2
sort:   b1; a2
sort:   b1; d2
sort:   b1; a2
sort:   b3; b1
sort:   b3; d2
sort:   c; b3
sort:   c; d2
filter:  a2
map:  a2
forEach: A2
filter:  b1
filter:  b3
filter:  c
filter:  d2
```

L'ordine delle operazioni conta (4): sorted è stateful!

- Modifichiamo l'ordine delle operazioni e ottimizziamo:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    }).forEach(s -> System.out.println("forEach: " + s));
```

Output:

```
filter:  d2
filter:  a2
filter:  b1
filter:  b3
filter:  c
map:  a2
forEach: A2
```

L'ordine di esecuzione delle operazioni è ottimizzato

```
Stream.of("domus", "aurea", "bologna", "burro", "charro")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("A");
    });
```

Output:

map: domus

anyMatch: DOMUS

map: aurea

anyMatch: AUREA

L'ordine di esecuzione delle operazioni è ottimizzato

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
List<Integer> twoEvenSquares =
    numbers.stream()
        .filter(n -> {
            System.out.println("filtering " + n);
            return n % 2 == 0;
        })
        .map(n -> {
            System.out.println("mapping " + n);
            return n * n;
        })
        .limit(2)
        .collect(toList());
```

Output:

```
filtering 1
filtering 2
mapping 2
filtering 3
filtering 4
mapping 4
```

Fare copie di stream

- Abbiamo detto che gli stream non sono riutilizzabili
- Tuttavia è possibile creare un builder di stream mediante una lambda
- Ad es:

```
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c")
            .filter(s -> s.startsWith("a"));
streamSupplier.get().anyMatch(s -> true);   // ok
streamSupplier.get().noneMatch(s -> true);  // ok
```

- Ovviamente ha più senso se, invece di un Supplier, abbiamo una funzione che prende in input una Collection e restituisce uno stream su tale collection

Stream paralleli

- Le operazioni su stream sequenziali sono effettuate in un singolo thread
- Le operazioni su stream paralleli, invece, sono effettuate concorrentemente su thread multipli

Perché creare uno stream parallelo?

Esempio:

```
int max = 1000000;
List<String> values = new ArrayList<>(max);
for (int i = 0; i < max; i++)
{
        UUID uuid = UUID.randomUUID();
        values.add(uuid.toString());
}
```

Perché creare uno stream parallelo?

Con **l'ordinamento sequenziale**:

```
long t0 = System.nanoTime();
long count = values.stream().sorted().count();
System.out.println(count);
long t1 = System.nanoTime();
long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.printf("ordinamento sequenziale: %d ms", millis);
// ordinamento sequenziale: 899 ms
```

Perché creare uno stream parallelo?

Con **l'ordinamento parallelo**:

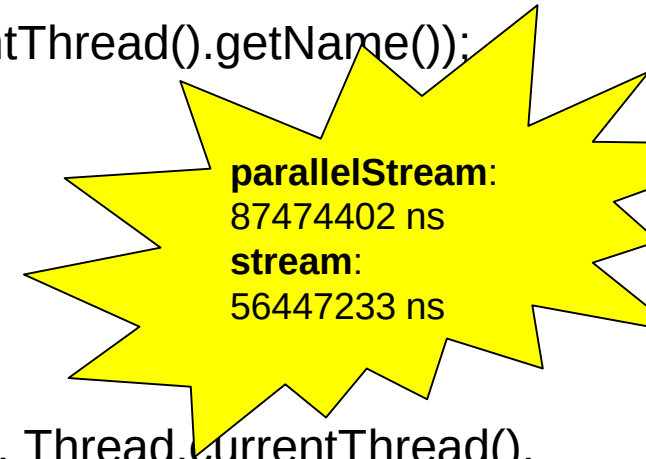
```
long t0 = System.nanoTime();
long count = values.parallelStream().sorted().count();
System.out.println(count);
long t1 = System.nanoTime();
long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.printf("ordinamento parallelo: %d ms", millis);
```

// ordinamento parallelo: 472 ms

Uno stream parallelo può essere MOLTO PIU' LENTO di uno stream sequenziale

- Quando le operazioni sono bloccanti:

```
List<String> ls = IntStream.range(0, 100000).mapToObj(Integer::toString).collect(toList());
l.parallelStream/stream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n", s, Thread.currentThread().getName());
        return true; })
    .map(s -> {
        System.out.format("map: %s [%s]\n", s, Thread.currentThread().getName());
        return s.toUpperCase(); })
    .sorted((s1, s2) -> {
        System.out.format("sort: %s <> %s [%s]\n", s1, s2,
Thread.currentThread().getName());
        return s1.compareTo(s2); })
     .forEach(s -> System.out.format("forEach: %s [%s]\n", s, Thread.currentThread().
                                                              getName()));
```



Quando usare uno stream parallelo?

- Quando il problema è parallelizzabile
- Quando posso permettermi di usare più risorse (es. tutti i core del processore)
- La dimensione del problema è tale da giustificare il sovraccarico dovuto alla parallelizzazione

Le mappe in Java 8 e 9

- Le mappe non supportano gli stream
- Ma forniscono numerose operazioni aggiuntive in Java 8

```
Map<Integer, String> map = new HashMap<>();
for (int i = 0; i < 10; i++) map.putIfAbsent(i, "val" + i);
map.forEach((id, val) -> System.out.println(val));
```

- In Java 9, abbiamo Map.of per creare una mappa costante

Le mappe in Java 8

- Se l'elemento 3 è presente, modifica il valore associatogli utilizzando la BiFunction in input come secondo parametro:

```
map.computeIfPresent(3, (key, val) -> val + key);
map.get(3); // val33
map.computeIfPresent(9, (key, val) -> null);
map.containsKey(9); // false
map.computeIfAbsent(23, key -> "val" + key);
map.containsKey(23); // true
map.computeIfAbsent(3, key -> "bam");
map.get(3); // val33
```

Le mappe in Java 8

```
map.remove(3, "val3");
map.get(3); // val33
map.remove(3, "val33"); // removes the pair
map.get(3); // null
map.getOrDefault(42, "not found"); // not found

map.merge(9, "val9", (value, newValue) ->
value.concat(newValue));
map.get(9); // val9
map.merge(9, "concat", (value, newValue) ->
value.concat(newValue));
map.get(9); // val9concat
```

Nuova API per data e ora

- Package java.time
- ZoneId: rappresenta un fuso orario
- LocalTime: rappresenta un'ora locale
- LocalDate: rappresenta una data locale
- LocalDateTime: rappresenta una data e ora
- Clock: accede all'attuale data e ora

```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();
```

Restituisce l'attuale data e ora in millisecondi, equivalente a System.currentTimeMillis()

Esempio: gestire la data

- LocalDate today = LocalDate.now();
- Alcuni metodi:

```
getDayOfMonth() : int - LocalDate
getDayOfWeek() : DayOfWeek - LocalDate
getDayOfYear() : int - LocalDate
getEra() : Era - LocalDate
getLong(TemporalField arg0) : long - LocalDate
getMonth() : Month - LocalDate
getMonthValue() : int - LocalDate
getYear() : int - LocalDate
```

Esercizio

- Sia data la seguente classe:

```
public class Titolo
{
    public enum Allineamento { CX, SX, DX }
    private Allineamento allineamento;
    private List<Riga> righe;

    public Titolo(Allineamento a) { this(a, new ArrayList<>()); }
    public Titolo(Allineamento a, List<Riga> righe) { allineamento = a; this.righe = righe; }
    public void add(Riga r) { righe.add(r); }
    public boolean isCentered() { return allineamento == Allineamento.CX; }
    @Override public String toString() { return righe.toString(); }
    public Allineamento getAllineamento() { return allineamento; }
    public List<Riga> getRighe() { return new ArrayList<>(righe); }

    static public class Riga
    {
        private String riga;
        private int numero;

        public Riga(String riga, int numero) { this.riga = riga; this.numero = numero; }
        public Riga(String riga) { this(riga, -1); }
        @Override public String toString() { return (numero == -1 ? "" : numero+": ")+riga; }
    }
}
```

Utilizzare gli stream per ottenere i seguenti dati (svolgere ciascun punto sia con riferimenti a metodi che lambda)

- A partire da una List<Titolo>, si calcolino **senza precalcoli di strutture dati intermedie** (ovvero, su una sola riga):
  1. l'insieme dei primi 5 titoli aventi al più una riga
  2. la lista dei soli titoli centrati e in ordine alfabetico
  3. mappa da allineamento a lista di titoli
  4. mappa da allineamento a insieme di titoli
  5. mappa da allineamento alla concatenazione delle stringhe dei titoli
  6. mappa da allineamento alla lista delle stringhe dei titoli
  7. insieme delle righe dei titoli ciascuna rappresentata sotto forma di stringa
  8. mappa dei conteggi delle parole nelle righe di titoli calcolate al punto precedente (questo esercizio richiede l'utilizzo di flatMap)
     - Utilizzando il Collectors.groupingBy
     - Utilizzando il Collectors.toMap