Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Stream di operazioni in Java

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

DAL «COME» AL «COSA»

- Nell'approccio tradizionale, specificare una operazione significa tipicamente dire *come farla istruzione per istruzione*
  - si eseguono le singole sotto-operazioni immediatamente
  - l'ordine delle sotto-operazioni è quello con cui sono scritte
  - chiaro, ma rende di fatto impossibile cambiare successivamente *come* fare una cosa senza riscrivere il codice
  - in particolare, rende impossibile parallelizzarlo senza rifarlo da zero
- Ciò è tipico del *paradigma imperativo*, in cui un programma è appunto una *sequenza di ordini*
  - *ossessione del controllo:* chi scrive il codice deve (vuole) controllare ogni singolo dettaglio di ogni singola operazione – non delega nulla
  - risultato: *si viene sopraffatti da tanto controllo* – difficoltà ad astrarre, impossibilità di prevedere tutto e pianificare tutto… ☹

DAL «COME» AL «COSA»

- Esistono però *altri paradigmi*, non orientati al controllo, in cui il focus è su *cosa si vuole*, lasciando sullo sfondo *come* farlo
  - il paradigma dichiarativo esprime gli *assiomi* e le *regole* che esprimono le *relazioni* vere nel dominio
  - il paradigma funzionale esprime la *trasformazione* dei dati in termini di *funzioni che agiscono* via via su di essi
- Come si computa in tali paradigmi?
  - nel paradigma dichiarativo si computa *cercando di raggiungere un obiettivo (goal)* espresso tipicamente da una domanda (query) tramite un processo di *inferenza logico-deduttiva*
  - nel paradigma funzionale si computa *fornendo i dati iniziali* e *prelevando dall'ultima funzione* i dati di uscita

ESEMPIO (1/3)

- Problema: data una lista di stringhe, produrre la lista delle *corrispondenti lunghezze*
  - ad esempio, data la lista ["pippo", "pluto", "paperino", "zio"], vogliamo ottenere in risposta la lista [5, 5, 8, 3]
- Paradigmi a confronto
  - nel *paradigma imperativo* ci si focalizza sulla *sequenza di ordini*
    - creare preventivamente una lista di interi per il risultato (vuota)
    - scorrere la lista iniziale elemento per elemento → ciclo for (o foreach)
    - per ogni elemento, calcolare la corrispondente lunghezza (un intero)
    - inserire tale intero nella nuova lista di destinazione

```
List<String>  source = List.of("pippo", "pluto", "paperino", "zio");
List<Integer> result = new ArrayList<>();
for (String s: source) result.add(s.length());
System.out.println(result);
```

Controllo!

ESEMPIO (2/3)

- Problema: data una lista di stringhe, produrre la lista delle *corrispondenti lunghezze*
  - ad esempio, data la lista ["pippo", "pluto", "paperino", "zio"], vogliamo ottenere in risposta la lista [5, 5, 8, 3]
- Paradigmi a confronto
  - nel *paradigma dichiarativo* si esprimono i *fatti* e le *relazioni* fra essi
    - a lista iniziale vuota corrisponde lista di uscita vuota
    - alla lista di stringhe [*Testa*|*Coda*] corrisponde la lista di interi [*TestaN* |*CodaN*]
      - *TestaN* è la lunghezza della stringa *Testa*
      - *CodaN* è il risultato (lista) della stessa operazione sulla sottolista *Coda*

Potenzialmente invertibile!

Ad esempio, in Prolog:

```
list_len([], []).
list_len([H|T], [HN|TN]) :- list_len(T,TN), atom_length(H,HN).
```

Non c'è il controllo: solo *le relazioni vere* nel dominio

Query: `list_len([pippo,pluto,paperino,zio], L).`
Reply: `L / [5,5,8,3]`

ESEMPIO (3/3)

- Problema: data una lista di stringhe, produrre la lista delle *corrispondenti lunghezze*
  - ad esempio, data la lista ["pippo", "pluto", "paperino", "zio"], vogliamo ottenere in risposta la lista [5, 5, 8, 3]
- Paradigmi a confronto
  - nel *paradigma funzionale* si esprimono le *trasformazioni* dei dati
    - detta S la sequenza dei dati iniziali
    - ogni elemento va *mappato* nella sua lunghezza: *element* → *length(element)*
    - la sequenza S' dei dati così trasformati costituisce il risultato

Non c'è il controllo: solo *come trasformare i dati*

Schema generale:

["pippo", "pluto", "paperino", "zio"] —— e → length(e) ——→ [5, 5, 8, 3]

```
List<Integer> res = lista.trasforma(s->s.length()).recupera(…);
```

LINGUAGGI MULTI-PARADIGMA

- Può essere molto interessante far coesistere *nello stesso linguaggio* paradigmi di computazione diversi
  - possibilità di sfruttare il più adatto a ogni circostanza
    - concatenare stringhe → Imperativo
    - calcolare le derivate simboliche → Dichiarativo
    - trovare la stringa più corta in una lista → Funzionale
- Ma coniugare più paradigmi non è cosa semplice
  - occorre progettare molto bene, fin dall'inizio, le API delle collections
  - ma soprattutto, occorre un linguaggio «intrinsecamente blended», progettato per essere tale fin dalle fondamenta
- Scala e Kotlin lo sono, Java no
  - Java: troppa base installata per cambiamenti profondi
  - Scala, Koltin: blend di paradigma imperativo + a oggetti + funzionale

JAVA COME LINGUAGGIO MULTI-PARADIGMA

- Per far coesistere *in un linguaggio già esistente* paradigmi di computazione diversi occorre un approccio non invasivo:
  - non si può «rifare tutto»: il know how e la base installata devono essere preservati
  - necessità di *evitare interferenze reciproche* che possano minare alla base il modello (e la correttezza) di funzionamento
- IDEA: *incapsulare il nuovo paradigma in una astrazione ad hoc* che funga da «stargate»
  - una porta verso un «universo parallelo» con altre leggi
    → nel caso OOP, un *oggetto* ad hoc
  - esternamente, un oggetto come un altro, coi suoi metodi
    → usabile secondo il classico paradigma imperativo OOP
  - internamente, un *motore* che opera secondo un *paradigma alternativo*
    → computa secondo sue leggi, nel suo «universo alternativo»

FUNCTIONAL PROGRAMMING IN JAVA: STREAM DI OPERAZIONI

- Gli stream di operazioni sono l'astrazione introdotta da Java per incapsulare il *paradigma di programmazione funzionale*
  - o almeno parte di esso..
- Ispirati al functional programming, gli stream di operazioni promuovono uno stile in cui ragionare *non più* in termini di *controllo*, ma di *come trasformare i dati* via via
  - richiede un pari cambio di *atteggiamento mentale*
- Focus su *cosa si vuole*, lasciando sullo sfondo come farlo

NB: Scala e Kotlin incorporano tale paradigma fin dalle fondamenta, quindi l'astrazione stream non è necessaria

STREAM DI OPERAZIONI

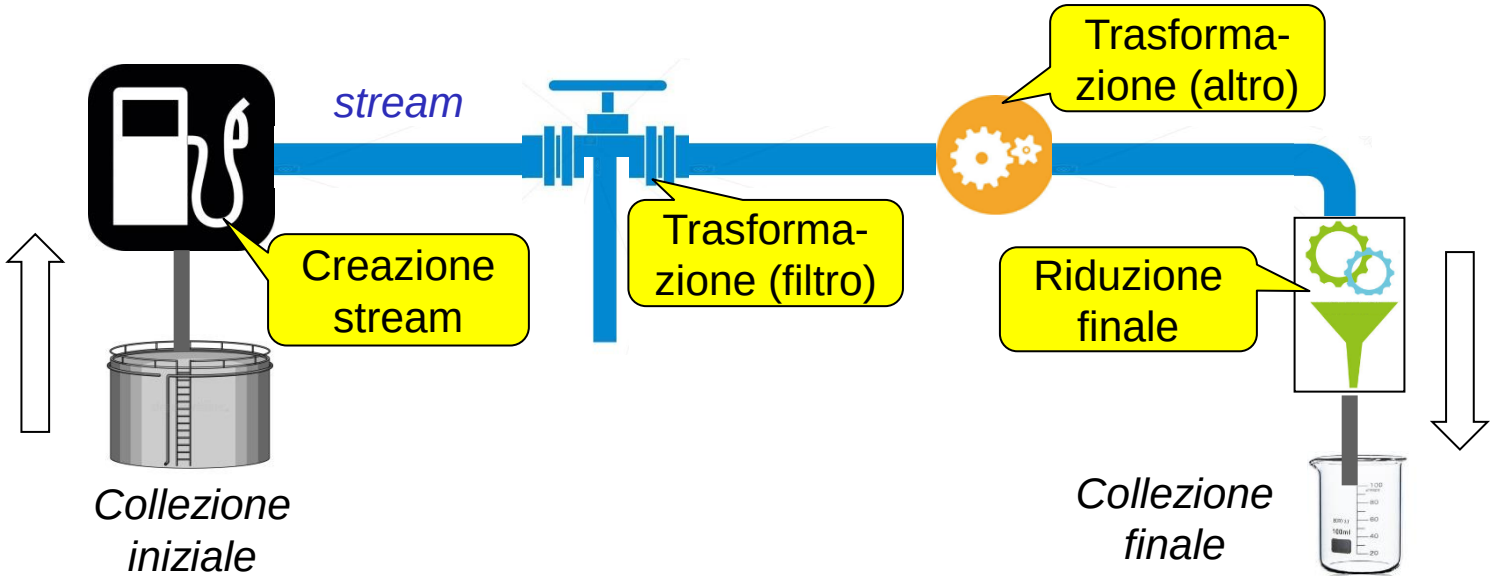
Uno stream di operazioni è un'astrazione per *specificare e concatenare operazioni*

- non è una collection
  - non memorizza elementi: usa una collection come «back end»
- si basa sul pattern cascading → *fluent interface*
  - ogni operazione restituisce un nuovo stream: il risultato dell'operazione precedente, con i dati «trasformati»
- svolge le operazioni in modo *lazy*
  - non le esegue immediatamente, ma *solo* ***quando*** e ***se*** *servono*
  - non le esegue necessariamente nell'ordine in cui sono scritte
  - non facendo ipotesi sull'ordine, rende *facile parallelizzare* l'algoritmo

STREAM DI OPERAZIONI

Tre momenti = tre categorie di funzionalità:

1. creazione dello stream → *operazione di **creazione***
2. uso dello stream → *operazioni di **trasformazione***
3. sintesi del risultato → *operazione di **riduzione***



STREAM DI OPERAZIONI

Tre momenti = tre categorie di funzionalità:

1. creazione dello stream → *operazione di creazione*
   - produce uno stream o *a partire da una sorgente esistente..*
   - *.. o generandone gli elementi uno ad uno*
2. uso dello stream → *operazioni di trasformazione*
   - filtrano, separano, modificano..
   - restituiscono sempre un altro **`Stream<`*`Qualcosa`*`>`**
3. sintesi del risultato → *operazione di riduzione*
   - estrae / computa il «risultato finale»
   - il risultato non è più uno `Stream`: è *qualcos'altro*
   - fatta la riduzione, lo stream è chiuso e non può più essere usato

CREARE UNO STREAM (1)

Uno stream *di oggetti*, **`Stream<Qualcosa>`**, viene creato:

- dal metodo **`stream()`** invocato su qualsiasi collection
- dal factory method **`Stream.of(array)`** nel caso di array
- da altri factory methods di altre classi (es. **`Random`**)

Esistono *versioni ad hoc* per i tre *tipi primitivi più usati*

- **`IntStream`**, **`LongStream`**, **`DoubleStream`**

```
List<String> elencoParole = …;
Stream<String> st = elencoParole.stream();
```

Stream di stringhe

```
String[] array = { "ciao", "ragazzi" };
Stream<String> st = Stream.of(array);
```

```
Random rnd = new Random();
IntStream st = rnd.ints(1,100).limit(500);
```

Stream di int

CREARE UNO STREAM (2)

- Uno stream può essere generato *elemento per elemento*
  - da **`Stream.generate`**, **`Stream.iterate`**
  - da metodi factory di molte classi, come **`Random`**
- Uno stream esegue le operazioni in modo *lazy,* al bisogno
  - ciò permette di generare anche uno stream *potenzialmente infinito*
  - se gli elementi fossero prodotti tutti all'inizio, sarebbe un ciclo infinito; *con la lazy evaluation no,* perché sono prodotti solo *al bisogno*

Stream di oggetti Double

```
Stream<Double> s = Stream.generate(Math::random);
```

```
Stream<Integer> pari = Stream.iterate(2, n->n+2);
```

Stream di oggetti Integer

UN PRIMO ESEMPIO

Una volta creato, sullo stream si invocano *operazioni*

- le trasformazioni intermedie *lavorano sui dati*
  - filtrano, separano, modificano.. tipicamente con una *lambda*
  - restituiscono sempre un altro `Stream<Qualcosa>`
- la riduzione finale *sintetizza il risultato*
  - conta gli elementi, li raccoglie in un *collector*, ..

```
List<String> elencoParole = …;
Stream<String> st = elencoParole.stream();
long numParoleLunghe =
        st.filter( p->p.length()>8 ).count();
List<String> paroleMaiuscole=
        st.map( p->p.toUpperCase() ).collect(…);
```

Trasformazione

Riduzione

STREAM PARALLELI

L'astrazione stream rende anche facile *parallelizzare il codice*

- uno stream parallelo si crea con **`parallelStream`**
  - anche questo è nella interfaccia-base `Collection`
  - si rimpiazza la chiamata `stream()` con `parallelStream()`

```
List<String> elencoParole = …;
long num = elencoParole.stream()
            .filter( p -> p.length()>8 )
            .count();
```

```
List<String> elencoParole = …;
long num = elencoParole.parallelStream()
            .filter( p -> p.length()>8 )
            .count();
```

STREAM PARALLELI: ESEMPIO

Ci si guadagna davvero? Facciamo una prova!

- Generiamo 500 interi ed elaboriamoli in due modi
  - con stream sequenziale / con stream parallelo
- ..e misuriamo i tempi

```
long t0 = System.currentTimeMillis();
List<Integer> list1 = new Random().ints(1,100).parallel()
                      .limit(500)
                      .mapToObj(i->i*i)
                      .collect(Collectors.toList());
long t1= System.currentTimeMillis();
System.out.println(t1-t0);
```

Limita lo stream (potenzialmente infinito) ai primi 500 valori

Calcola i quadrati

Mette i risultati (degli Integer) in una lista

35	32	31
12	10	8

Lo stream parallelo guadagna circa un fattore 3 (in questo caso, su questo pc)

UN ALTRO ESEMPIO

- Nuovo caso di studio
  - costruiamo uno stream (di **`Integer`**, stavolta, *non di* ***`int`***) da un array
  - filtriamo solo i valori positivi
  - mettiamo il risultato in un *nuovo array*
- Per farlo, il collettore generico ***`toArray`*** ha bisogno del *riferimento al costruttore* del «giusto» array da creare
  - se vogliamo un array di **`Integer`**, **`Integer[]::new`**

```
Stream<Integer> st =
  Stream.of(new Integer[]{3,67,1,18,-2,9
        .filter(x->x>0);
Integer[] result = st.toArray(Integer[]::new);
System.out.println(Arrays.toString(result));
```

Method reference al costruttore di «array di Integer»

TRASFORMAZIONI FONDAMENTALI

Ci sono tre trasformazioni fondamentali (con più varianti):

- **`filter(`*`Predicate`*`)`**
  - filtra nel nuovo stream-risultato solo gli elementi dello stream che soddisfano la condizione, ossia solo quelli per cui il predicato è vero
- **`map(`*`Function`*`)`** [e variante **`mapToObj(`*`Function`*`)`**]
  - applica a ogni elemento **x** dello stream la funzione data: **y = f(x)**
  - produce un nuovo stream con i rispettivi risultati **y** (ev. di tipo diverso)
  - la variante **`mapToObj`** si usa per stream di tipi primitivi (il risultato è invece un tipo oggetto, non primitivo)
- **`flatMap(`*`Function`*`)`**
  - applica a ogni elemento dello stream la funzione data, *che si suppone produca come risultato uno stream* (non un singolo elemento)
  - perciò, **`map`** darebbe come risultato uno *stream di stream* (scomodo)
  - **`flatMap`** li unisce in un unico stream, «appiattendo» il risultato

ALTRE TRASFORMAZIONI

È inoltre possibile:

- estrarre un sotto-stream con **`limit(n)`** o **`skip(n)`**
  - **`limit(n)`** estrae solo i primi *n* elementi
    → *importante per limitare gli stream infiniti*
  - **`skip(n)`** fa l'opposto, escludendo i primi *n* elementi
- combinare due stream con **`concat`**

```
Stream<Double> st1 =
      Stream.generate(Math::random).limit(10);
Stream<Double> st2 = st1.skip(5);
Stream<Double> st3 = st1.concat(st2);
```

RIDUZIONI FONDAMENTALI

Riduzioni fondamentali:

- **`count`**, **`min`**, **`max`**, **`average`**
  - conta gli elementi, trova il min, il max, la media..
- **`collect(`*`collector`*`)`**
  - riunisce gli elementi in un *collettore*
  - tipicamente, il collettore è un *array* o una qualche *collection*
- **`reduce(`*`function`*`)`**
  - applica a ogni elemento dello stream la funzione data, *che specifica dove mettere il risultato*
  - il risultato non è più uno stream
- **`forEach(`*`operation`*`)`**
  - iterazione interna dell'operazione data sugli elementi dello stream

La classe **`Collectors`** fornisce molti collettori di uso comune

COLLETTORI FONDAMENTALI

Collettori più comuni:

- **`toCollection`**, **`toList`**, **`toMap(fk,fv)`**, **`toSet`**
  - *restituisce una collection / lista / mappa / set*
  - per la mappa, le due funzioni generano rispettivamente chiavi e valori
- **`groupingBy(`*`criterio`*`)`**, **`partitioningBy(`*`criterio`*`)`**
  - riunisce gli elementi "a gruppi" secondo un dato criterio
  - *restituisce una mappa*, con tante entry quanti i gruppi e per ogni riga la lista degli elementi dello stesso gruppo
- **`summarizingXX`**, **`summingXX`**, **`averagingXX`** (`XX` = `Int`, `Long`, `Double`)
  - *restituisce una mappa,* con tante entry quanti i gruppi e per ogni riga il risultato relativo a tale gruppo
- **`joining`** (per stream di stringhe)
  - *restituisce una stringa* concatenata, col separatore indicato

COLLETTORI FONDAMENTALI

Collettori più comuni:

- **`toCollection`**, **`toList`**, **`toMap(fk,fv)`**, **`toSet`**
  - *restituisce* *llection / lista / mappa / set*

valori

- **`g`**

riga la

- **`s`**

`ouble`)

riga il

risultato relativo a tale gruppo

- **`joining`** (per stream di stringhe)
  - *restituisce una stringa* concatenata, col separatore indicato

- Utile per produrre una collezione *diversa dalle tre standard* **`List`**, **`Map`**, **`Set`** (ad esempio, un **`SortedSet`**)
- Se non c'è un collector standard, si usa **`toCollection`** *passandogli il costruttore della specifica collection da creare*
- Ad esempio, per avere un **`TreeSet`** (unica concretizzazione di **`SortedSet`**), si può scrivere:

```
toCollection(TreeSet::new)
```

ESEMPIO (1)

- Data una lista di Persone
  - si vuole una stringa-risultato ottenuta concatenando le **`toString`** delle varie istanze

Approccio classico:

- scorrere gli elementi della lista → ciclo **`for`** o **`foreach`**
- per ogni elemento, invocare **`toString`** (mettendo il risultato in una variabile temporanea di appoggio)
- concatenare la stringa così ottenuta alle precedenti (serve un accumulatore per il risultato)
- alla fine, il risultato è nell'accumulatore utilizzato

ESEMPIO (2)

- Data una lista di Persone
  - si vuole una stringa-risultato ottenuta concatenando le **`toString`** delle varie istanze

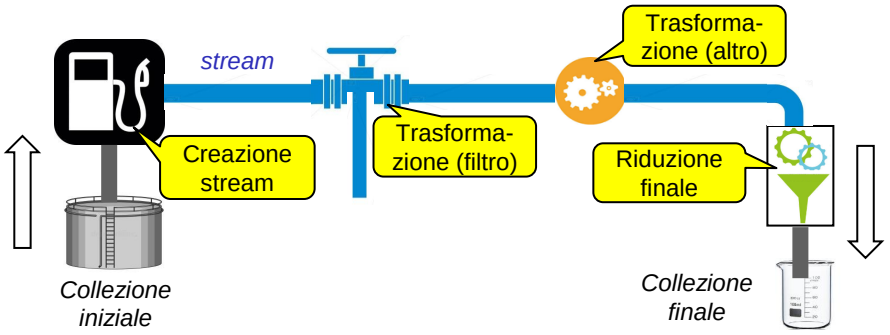
```
String res = "";
for (Persona p : lista){
      res += p.toString();
}
```

Approccio classico:

- scorrere gli elementi della lista → ciclo **`for`** o **`foreach`**
- per ogni elemento, invocare **`toString`** (mettendo il risultato in una variabile temporanea di appoggio)
- concatenare la stringa così ottenuta alle precedenti (serve un accumulatore per il risultato)
- alla fine, il risultato è nell'accumulatore utilizzato

ESEMPIO (3)

- Data una lista di Persone
  - si vuole una stringa-risultato ottenuta concatenando le **`toString`** delle varie istanze

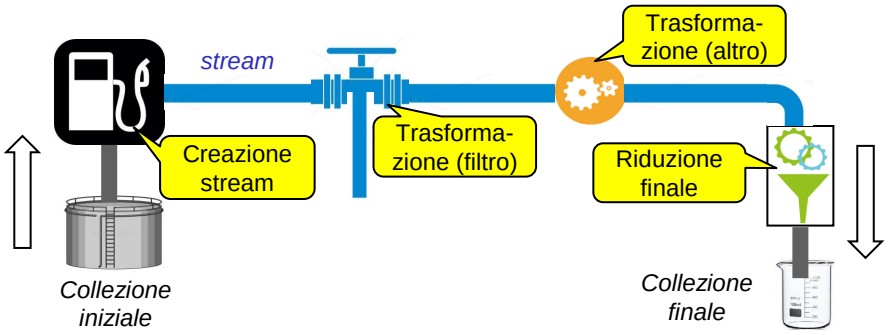


Approccio a stream:

- si crea dalla lista uno stream di persone → **`stream()`**
- per ogni elemento, si mappa la persona corrente nel risultato (stringa) desiderato → **`map(…)`**
- si accumulano tali risultati → **`collect(…)`**
  in un collettore appropriato → **`joining(…)`**
- alla fine, il risultato è nel collettore utilizzato

ESEMPIO (4)

- Data una lista di Persone
  - si vuole una stringa-risultato ottenuta concatenando le **`toString`** delle varie istanze




```
String res =
  lista.stream()
       .map(Persona::toString)
       .collect(Collectors.joining("\n"));
```

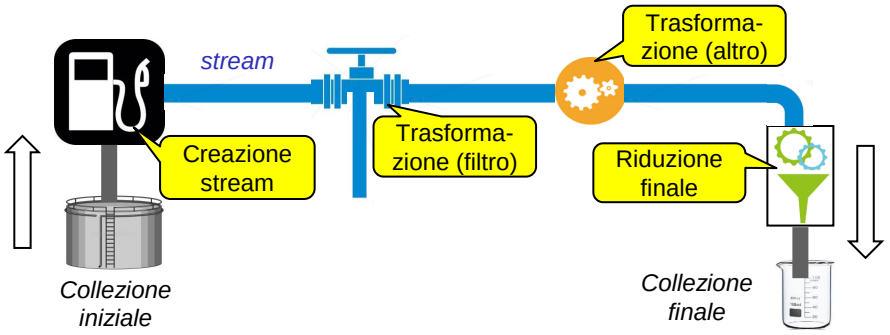
Creazione stream

Trasformazione da Persona a Stringa

Riduzione finale (concatenamento con join)

ESEMPIO (5)

- Variante: solo maggiorenni
  - si vuole una stringa-risultato ottenuta concatenando le **`toString`** delle sole istanze che soddisfano la condizione




```
String res =
  lista.stream()
       .filter(p->p.getEta()>=18)
       .map(Persona::toString)
       .collect(Collectors.joining("\n"));
```

Creazione stream

Trasformazione (filtro)
Solo le persone che…

Trasformazione da Persona a Stringa

Riduzione finale (concatenamento con join)

GENERALIZZAZIONE
PENSARE CON GLI STREAM (1)

- Gli stream costituiscono un totale *cambio di paradigma*
- Richiedono un pari cambio di *atteggiamento mentale*
  - occorre pensare *non più* in termini di controllo
  - ma in termini di *come trasformare i dati via via*

ESEMPIO: data una collezione di elementi, identificare tutti quelli che soddisfano certe caratteristiche, usarli per costruire un altro oggetto di diverso tipo (wrapper) e mettere questi ultimi in un insieme ordinato.

Approccio classico:

- scorrere gli elementi della lista → ciclo **`for`** o **`foreach`**
- per ogni elemento, verificare se ha le caratteristiche → **`if`**
- se **`if`** positivo, costruire il nuovo oggetto richiesto → **`new`**
- inserire tale nuovo oggetto in un **`TreeSet`** (preventivamente creato)

PENSARE CON GLI STREAM (2)

- Gli stream costituiscono un totale *cambio di paradigma*
- Richiedono un pari cambio di *atteggiamento mentale*
  - occorre pensare *non più* in termini di controllo

Approccio fondato sul controllo totale:
- controlliamo tutto
- istante per istante stabiliamo ogni singola istruzione (indici..)

.. così, *al minimo errore esplode tutto!*
- massimo controllo = onere su di noi
- al minimo indice sbagliato, alla minima variabile assegnata male.. *boom!*
- debug faticoso (spesso, un incubo!)

Approccio classico:
- scorrere gli elementi della collezione → ciclo **`for`** o **`foreach`**
- per ogni elemento, verificare se ha le caratteristiche → **`if`**
- se **`if`** positivo, costruire il nuovo oggetto richiesto → **`new`**
- inserire tale nuovo oggetto in un **`TreeSet`** (preventivamente creato)

PENSARE CON GLI STREAM (3)

Approccio classico:

- scorrere gli elementi della collezione → ciclo **`for`** o **`foreach`**
- per ogni elemento, verificare se ha le caratteristiche → **`if`**
- se **`if`** positivo, costruire il nuovo oggetto richiesto → **`new`**
- inserire tale nuovo oggetto in un **`TreeSet`** (preventivamente creato)

```
Collection<Persona> coll = …;
TreeSet<Res> t = new TreeSet<>();
for (Persona q : coll) {
      if ( isGood(q,…) ) {
            Res r = calcRes(q,…);
            t.add(r);
      }
}
```

Esempio: collezione di Persone

Risultato: un albero di Res

se l'elemento rispetta la condizione voluta…

… produciamo l'oggetto richiesto di tipo Res e lo mettiamo in t

PENSARE CON GLI STREAM (4)

- Gli stream costituiscono un cambio di paradigma
- Richiedono un pari cambio di atteggiamento mentale
  - occorre pensare *non più* in termini di controllo
  - ma in termini di *come trasformare i dati via via*

ESEMPIO: data una collezione di elementi, identificare tutti quelli che soddisfano certe caratteristiche, usarli per costruire un altro oggetto e mettere questi ultimi in un insieme ordinato.

Approccio a stream:

- ottenere uno stream di elementi dalla collezione → **`.stream()`**
- filtrare solo gli item con le «giuste» caratteristiche → **`.filter(…)`**
- produrre il risultato mappando il vecchio oggetto nel nuovo → **`.map`**(…)
- raccogliere i risultati nel «giusto» collector → **`.toCollection(…)`**

PENSARE CON GLI STREAM (5)

- Gli stream costituiscono un cambio di paradigma
- Richiedono un pari cambio di atteggiamento mentale
  - occorre pensare *non più* in termini di controllo

Approccio che astrae il controllo:
- non controlliamo niente
- diciamo solo *cosa fare sull'elemento* presente al momento sullo stream

Un altro pianeta:
- nessun onere su di noi
- niente indici, né variabili da assegnare
- maggior chiarezza, debug semplificato

Approccio a stream:
- ottenere uno stream di elementi dalla collezione → `.stream()`
- filtrare solo gli item con le «giuste» caratteristiche → `.filter(…)`
- produrre il risultato mappando il vecchio oggetto nel nuovo → `.map(…)`
- raccogliere i risultati nel «giusto» collector → `.toCollection(…)`

PENSARE CON GLI STREAM (6)

Approccio a stream:

- ottenere uno stream di elementi dalla collezione → **`.stream()`**
- filtrare solo gli item con le «giuste» caratteristiche → **`.filter(…)`**
- produrre il risultato mappando il vecchio oggetto nel nuovo → **`.map`**(…)
- raccogliere i risultati nel «giusto» collector → **`.toCollection(…)`**

```
coll.stream()
    .filter( q->isGood(q,…) )
    .map( q->calcRes(q,…) )
    .collect(
        Collectors.toCollection(TreeSet::new));
```

Generazione stream

Selezione dei soli elementi tali che…

Dato l'oggetto q, produzione dell'altro oggetto di tipo Res

Raccolta di tutti gli oggetti Res nell'opportuno collettore

PENSARE CON GLI STREAM (7)

CONFRONTO

Approccio classico:

- scorrere gli elementi della collezione → ciclo **`for`** o **`foreach`**
- per ogni elemento, verificare se ha le caratteristiche → **`if`**
- se **`if`** positivo, costruire il nuovo oggetto richiesto → **`new`**
- inserire tale nuovo oggetto in un **`TreeSet`** (preventivamente creato)

Approccio a stream:

- ottenere uno stream di elementi dalla collezione → **`.stream()`**
- filtrare solo gli item con le «giuste» caratteristiche → **`.filter(…)`**
- produrre il risultato mappando il vecchio oggetto nel nuovo → **`.map`**(…)
- raccogliere i risultati nel «giusto» collector → **`.toCollection(…)`**

PENSARE CON GLI STREAM (8)

```
Collection<Persona> coll = …;
TreeSet<Res> t = new TreeSet<>();
for (Persona q : coll) {
      if ( isGood(q,…) ) {
            Res r = calcRes(q,…);
            t.add(r);
      }
}
```

Controllo totale (e puntuale)

```
coll.stream()
      .filter( q->isGood(q,…) )
      .map( q->calcRes(q,…) )
      .collect(
            Collectors.toCollection(TreeSet::new));
```

Astrazione del controllo

STREAM DI TIPI PRIMITIVI

Gli stream *di tipi primitivi* **`int`**, **`long`**, **`double`** sono creati:

- dai factory method **`IntStream.of(…)`** nel caso di **`int`**
- dai factory method **`LongStream.of(…)`** nel caso di **`long`**
- dai factory method **`DoubleStream.of(…)`** nel caso **`double`**

Possono essere anche creati da metodi generativi

- es: **`generate`**, **`iterate`**, **`range`**, **`rangeClosed`**, **`concat`**
- a volte da metodi di classi come **`Random`** (e molte altre)

ESEMPI

```
IntStream is = IntStream.iterate(3, x -> 2*x);
DoubleStream ds = DoubleStream.generate(
                                    Math::random);
```

STREAM DI TIPI PRIMITIVI

NON è possibile convertire stream di tipi primitivi in stream di oggetti (wrapper) tramite boxing/unboxing automatici

- per convertire uno stream di tipi primitivi in uno stream di oggetti wrapper, usare il metodo **`boxed()`**
- per mappare uno stream di tipi primitivi in uno stream di oggetti generici, usare il metodo **`mapToObj()`**

ESEMPIO

```
IntStream is = new Random().ints(0,M);
Stream<Integer> iis = is.boxed();
Stream<Frazione> fis = is.mapToObj(mapper);
```

dove *mapper* mappa **`int`** in **`Frazioni`**

ESEMPI (1)

```
List<String> elencoParole = …;
Stream<String> st1 = elencoParole.stream();
Stream<String> st2 = st1.filter(p->p.length()>4);
long num = st2.count();
```

Trasformazione

Predicato

Riduzione

```
Stream<String> st3 =
            st1.map(p->p.toUpperCase());
List<String> listaParoleMaiuscole=
            st3.collect(Collectors.toList());
```

Trasformazione

Funzione

Riduzione

ESEMPI (2)

```
List<String> elencoParole = …;

Stream<String> st1 = list.stream();

Stream<Character> st2 = st1.map(p -> p.charAt(0));
```

Funzione

Trasformazione

Stream di caratteri che contiene *le iniziali di tutte le parole*

```
Stream<List<Integer>> st4 = Stream.of(
   Arrays.asList(1,2),
   Arrays.asList(3,4),
   Arrays.asList(5)
 );
Stream<Integer> st5 =
     st4.flatMap( list -> list.stream() );
```

Stream di liste di interi
[[1,2], [3,4], [5]]

Stream di interi "appiattito"
[1, 2, 3, 4, 5]

ESEMPI (3)

Stampa base

```
Stream<?> st = …;
st.forEach( System.out::println );
```

Stampa un po' più evoluta:

```
Stream<?> st = …;
st.forEach( e -> System.out.print(e +", ") );
System.out.println();
```

Problema: *l'iterazione interna consuma lo stream*

- MOTIVO: è una *riduzione*, ossia una *operazione terminale*
- dunque, a iterazione terminata lo stream è chiuso
  → *non è più possibile svolgere su di esso alcuna operazione*

ESEMPI (4)

Alternativa: l'operazione **`peek`** che *duplica lo stream*

- si itera sugli elementi di un *nuovo stream duplicato*
- lo stream originale non viene toccato

```
st.peek( e -> System.out.print(e +", ") )
  .doSomethingElseOnOriginalStream(..);
```

Esempio completo:

```
Double[] numeri =
Stream.iterate(1.0, x -> 2*x)
      .peek( System.out::println
      .limit(20)
      .toArray(Double[]::new);
```

- Method reference al costruttore di "array di Double"
- Dice a **`toArray`** *che tipo di array* costruire

ESEMPI (5)

Il metodo **`reduce`** esprime la generica riduzione

- 1° argomento: *l'elemento neutro dell'operazione*
- 2° argomento: *l'operazione associativa da svolgere*
- Si applica ripetutamente l'operazione a *coppie di elementi consecutivi*

ESEMPIO: somma degli elementi di uno stream di **`Integer`**

```
Stream<Integer> st
  = Stream.of( new Integer[]{1,2,54,23,12,-4,22} );
Integer sum = st.peek( System.out::println)
                .reduce(0, (res,v) -> res+v );
```

Elemento neutro della somma

Risultato: 110

Operazione da applicare ripetutamente

$Res_1 = 0+v_1$, $Res_2=Res_1+v_2$, ..

$Res_N=Res_{N-1}+v_N$

ESEMPI (5)

ESEMPIO: variante con **`IntStream`**

```
IntStream st
  = Stream.of( new int[]{1,2,54,23,12,-4,22} );
int sum = st.peek( System.out::println)
            .reduce(0, (res,v) -> res+v );
```

Elemento neutro della somma

Operazione da applicare ripetutamente

$Res_1 = 0+v_1$, $Res_2=Res_1+v_2$, ..

$Res_N=Res_{N-1}+v_N$

Risultato: 110

ESEMPI (6)

Esiste anche una versione di **`reduce`** *a singolo argomento*

- l'unico argomento è la funzione: non c'è l'elemento neutro iniziale
- quindi, il risultato può non esistere se lo stream è vuoto

che restituisce un **`Optional`**

- appunto perché il risultato potrebbe non esistere
- occorre quindi verificare che il risultato non sia **`empty`** ed estrarre eventualmente il valore

ESEMPI (7)

ESEMPIO: ulteriore variante con **reduce** a singolo argomento:

```
IntStream st1
 = Stream.of( new int[]{1,2,54,23,12,-4,22} );
IntStream st2 = Stream.of( new int[]{} );
OptionalInt sum1 = st1.reduce((res,v) -> res+v );
OptionalInt sum2 = st2.reduce((res,v) -> res+v );
System.out.println("Somma : " +
  (sum1.isPresent() ? sum1.getAsInt() : "indefinita"));
System.out.println("Somma : " +
  (sum2.isPresent() ? sum2.getAsInt() : "indefinita"));
```

Vuoto

empty

ESEMPI (8)

Esiste anche una **`reduce`** a tre argomenti

- 1° argomento: *l'elemento neutro dell'operazione*
- 2° argomento: *l'operazione da svolgere che produce un risultato in un dominio diverso (accumulatore)*
- 3° argomento: *la funzione che combina risultati parziali*

ESEMPIO: somma *delle lunghezze delle stringhe* di uno stream

```
String[] array = { "ciao", "bravi", "ragazzi" };
Stream<String> stP = Stream.of(array);

int sum = stP.reduce(0,
          (res, parola) -> res + parola.length(),
          (r1,r2)->r1+r2 );
```

Risultato: 16

Operazione che combina i risultati parziali (utile se viene parallelizzata)

Operazione da applicare ripetutamente
$Res_1 = 0+v_1$, $Res_2=Res_1+v_2$, ..
$Res_N=Res_{N-1}+v_N$

ESEMPI (9)

ESEMPIO: variante, uso di **`mapToInt`**

- Si può ottenere lo stesso risultato più semplicemente, così:

```
int sumLengths2 =
      stP.mapToInt( String::length ).sum();
```

Risultato: 16

Mappa ogni parola in un intero..

..che è la sua lunghezza:

la riduzione finale li somma tutti

ESEMPI (10)

Un'altra riduzione utili (fra le tante) è **average**

- fa la media e restituisce un **OptionalDouble**

```
IntStream st1
  = Stream.of( new int[]{1,2,54,23,12,-4,22} );
double media = st2.average().getAsDouble();
```

Risultato: 17.714

ESEMPI (11)

Il metodo **`collect`** accumula elementi trasformati

- l'argomento è un *opportuno Collector*
- i collettori sono ottenibili dalla factory **`Collectors`**

ESEMPIO: concatenare stringhe *separandole con virgole*

- il collettore **`joining`** fa esattamente al caso nostro
- unisce le stringhe dello stream mettendoci in mezzo *(ma non alla fine!)* la stringa specificata

```
Stream<Integer> st1 = Stream.of(
    new Integer[]{1, 2, 54, 23, 12, -4, 22} );
String res =
      st1.map(Object::toString)
         .collect(Collectors.joining(", "));
```

ESEMPI (12)

ESEMPIO: fare la media di un insieme di interi (come prima) *che però non sono già disponibili, ma devono essere prima calcolati*

CASO CONCRETO: *media delle lunghezze di uno stream di parole*

- Il collettore **`averagingInt`** fa esattamente al caso nostro
  → *il suo argomento è l'operazione da svolgere*
  → qui, *ricavare la lunghezza* della stringa corrente
- Alla fine, calcola la media (reale) di un insieme di valori interi

```
String[] array = { "ciao", "bravi", "ragazzi" };
Stream<String> stP = Stream.of(array);
double average =
  stP.collect(Collectors.averagingInt(String::length));
```

Risultato: 5.333

Ricavo int da string

ESEMPI (13)

`collect` si sposa bene con **`toMap`**, **`toList`**, ecc.

ESEMPIO 1: ottenere una mappa stringa/stringa

- Il collettore **`toMap`** fa esattamente al caso nostro
- associa al codice fiscale (chiave) il nome della persona (valore)

```
Persona[] arrayPersone = {
  new Persona("Mario Rossi", "RSSMRA76H12A944I"),
  new Persona("Lucia Verdi", "VRDLCU98T65H223X") };

List<Persona> listaPersone = Arrays.asList(arrayPersone);

Stream<Persona> st = listaPersone.stream();

Map<String, String> mappa1 =
  st.collect( Collectors.toMap(Persona::getCodFisc,
                               Persona::getNome) );
```

```
{ RSSMRA76H12A944I=Mario Rossi, VRDLCU98T65H223X=Lucia Verdi }
```

ESEMPI (14)

ESEMPIO 2: mappa stringa/persona

- Il collettore **`toMap`** fa ancora al caso nostro
- stavolta associa alla chiave la persona stessa (valore)
- Problema: come recuperare tale persona?
  La classe Persona non ha un metodo «**`getThis`**»..
- Provvede a ciò il metodo **`Function.identity`**

```
Map<String, Persona> mappa2 =
  st.collect( Collectors.toMap(Persona::getCodFisc,
                               Function.identity() ) );
```

```
{ RSSMRA76H12A944I=Persona@41629346,
  VRDLCU98T65H223X=Persona@404b9385 }
```

ESEMPI (15)

Il metodo **`groupingBy`** raggruppa gli elementi

- l'argomento è un classificatore *(Classifier)* ossia una funzione che produce risultati di diverse «categorie»
- viene generata una mappa avente per chiave il gruppo e per valore la lista degli elementi di quel gruppo

ESEMPIO: *raggruppare le culture locali per paese*

- per ogni paese, vogliamo le culture locali disponibili

```
Stream<Locale> locales =
  Stream.of( Locale.getAvailableLocales() );

  Map<String, List<Locale>> mappaPaeseLocales =
     locales.collect(
        Collectors.groupingBy( Locale::getCountry ));
```

ESEMPI (16)

Esempio:

```
Stream<Locale> locales =
  Stream.of( Locale.getAvailableLocales() );

  Map<String, List<Locale>> mappaPaeseLocales =
      locales.collect(
        Collectors.groupingBy( Locale::getCountry ));
```

Output:

```
[it_IT]
[fr_CA, en_CA]
[fr_CH, de_CH, it_CH]
```

```
System.out.println( mappaPaeseLocales.get("IT") );
System.out.println( mappaPaeseLocales.get("CA") );
System.out.println( mappaPaeseLocales.get("CH") );
```

ESEMPI (17)

Nel caso specifico di due gruppi (partizione), è opportuno preferire il metodo **`partitioningBy`**

- l'argomento è un *classificatore boolean*

ESEMPIO: *raggruppare le culture locali in due gruppi*

- paesi che parlano inglese / paesi che non lo parlano

```
Stream<Locale> locales =
  Stream.of( Locale.getAvailableLocales() );

  Map<Boolean, List<Locale>> mappaEnglish =
     locales.collect(
       Collectors.partitioningBy(
         loc -> loc.getLanguage().equals("en") ));
```

ESEMPI (18)

Eseguendo, si notano *elementi spuri:*

- le lingue "non istanziate su uno specifico paese"

```
[en_US, en_SG, en_MT, en, en_PH, en_NZ, en_ZA, en_AU, en_IE,
en_CA, en_IN, en_GB]
[ar_AE, ar_JO, ar_SY, hr_HR, fr_BE, es_PA, mt_MT, es_VE, … ]
```

Elementi spuri

```
System.out.println( mappaPaeseLocales.get(true) );
System.out.println( mappaPaeseLocales.get(false));
```

ESEMPI (19)

VARIANTE: evitare elementi spuri

- Togliamo i codici linguaggio generici, che non corrispondono a uno specifico paese
- Rendiamo più selettiva l'espressione di filtro

```
Stream<Locale> locales =
  Stream.of( Locale.getAvailableLocales() );

  Map<Boolean, List<Locale>> mappaEnglish =
     locales.collect(
       Collectors.partitioningBy(
         loc -> loc.getLanguage().equals("en")
         && !loc.getCountry().equals("") ));
```

I codici generici senza Paese associato vengono eliminati

ESEMPI (20)

- Di default, **`groupingBy`** e **`partitioningBy`** raggruppano gli elementi in *liste*
- Si possono ottenere *strutture dati diverse* fornendo un *collettore specifico* *come ulteriore argomento*

ESEMPIO: *raggruppare le culture locali per paese in Set*

```
Stream<Locale> locales =
  Stream.of( Locale.getAvailableLocales() );

  Map<String, Set<Locale>> mappaPaeseLocales =
     locales.collect(
        Collectors.groupingBy( Locale::getCountry,
                               Collectors.toSet()  ));
```

ESEMPI (21)

Più in generale, **`groupingBy` e `partitioningBy`** supportano il *downstream processing,* ossia la possibilità di *ulteriori elaborazioni* sui dati *già raggruppati*.

ESEMPIO: *contare* le culture locali raggruppate

```
Stream<Locale> locales =
  Stream.of( Locale.getAvailableLocales() );

  Map<String, Long> mappaPaeseLocales =
     locales.collect(
       Collectors.groupingBy( Locale::getCountry,
                              Collectors.counting()  ));
```

```
1  System.out.println( mappaPaeseLocales.get("IT") );
2  System.out.println( mappaPaeseLocales.get("CA") );
3  System.out.println( mappaPaeseLocales.get("CH") );
```

ESEMPI (22)

VARIANTE: paesi che parlano inglese o meno

- senza i codici linguaggio generici

```
Stream<Locale> locales =
  Stream.of( Locale.getAvailableLocales() );

  Map<Boolean, Long> mappaContaEnglish =
     locales.collect(
        Collectors.partitioningBy(
          loc -> loc.getLanguage().equals("en")
             && !loc.getCountry().equals(""),
          Collectors.counting()));
```

```
11
149
System.out.println( mappaContaEnglish.get(true) );
System.out.println( mappaContaEnglish.get(false));
```

Esempi da compiti d'esame

(sempre utili…. ☺)

I SEMPRE UTILI (1)

Problema: data una lista di elementi, produrre un set (senza duplicati) dei *soli elementi tali che…*

```
Set<Elemento> scelti =
  lista.stream().filter(e -> taleChe(e)).collect(Collectors.toSet());
```

Problema: data una lista di elementi, produrre *un'altra lista senza duplicati* dei *soli elementi tali che…*

```
List<Elemento> result =
  lista.stream().filter(e -> taleChe(e)).distinct().collect(
                                                  (Collectors.toList());
```

I SEMPRE UTILI (2)

Problema: produrre una *lista di N valori casuali* compresi fra 0 e M (possono esserci duplicati)

```
List<Integer> elementiCasuali =
  new Random().ints(0,M).limit(N).boxed().collect(Collectors.toList());
```

Perché lo stream non sia infinito

Trasforma in **`IntStream`** in uno **`Stream<Integer>`**

(e più in generale uno stream di tipi primitivi in uno di tipi wrapper)

Idem con valori *distinti*:

```
List<Integer> scelti =
  new Random().ints(0,M).distinct().limit(N).boxed().collect(...);
```

Idem con un *set* (elimina i duplicati, quindi sono *al più N* valori)

```
Set<Integer> scelti =
  new Random().ints(0,M).limit(N).boxed().collect(Collectors.toSet());
```

I SEMPRE UTILI (3)

Problema: produrre una *lista di (non più di N)* ***elementi*** *selezionati casualmente* da una lista di M elementi possibili (di tipo Element)

- si genera una lista di N interi casuali (con o senza duplicati) fra 0 e M
- si sfruttano tali indici per estrarre elementi **dall'altra lista (source)**
- si accumulano nella lista-risultato solo gli elementi sorteggiati

```
List<Elemento> scelti =
  new Random().ints(0,M).distinct().limit(N)
        .mapToObj(source::get)
        .collect(Collectors.toList());
```

Come caso particolare, convertire lo stream di **`int`** in stream di **`Integer`** tramite **`mapToObj`** anziché **`boxed`**:

```
List<Integer> elementiCasuali =
  new Random().ints(0,M).limit(N).mapToObj( i -> i ).collect(...);
```

I SEMPRE UTILI (4)

Problema: data una *lista di documenti*

- caratterizzati ciascuno da identificativo univoco (ID) e timestamp

produrre la lista dei soli documenti *modificati entro quel timestamp*

```
List<Document> result =
  lista.stream().filter( doc ->
            doc.getTimeStamp().isBefore(timestamp) ||
            doc.getTimeStamp().equals(timestamp)    )
         .collect(Collectors.toList());
```

Problema: come sopra, ma restituire *solo il documento più recente*

- anziché ammucchiarli in lista, prendiamo solo il max
- il risultato è un Optional perché max potrebbe non trovare il massimo

```
Optional<Document> doc =
  lista.stream().filter( doc ->
            doc.getTimeStamp().isBefore(timestamp) ||
            doc.getTimeStamp().equals(timestamp)    )
         .max(Comparator.comparing(Document::getTimeStamp));
```

I SEMPRE UTILI (5)

Problema: data una *lista di tutti i voli aerei disponibili* (orario stagionale)

- caratterizzati da aeroporti e orari di partenza/arrivo + giorni in cui operano

produrre la lista dei soli voli *fra i due aeroporti dati* che siano *operanti in un ben preciso giorno della settimana* (dato anch'esso)

```
List<FlightSchedule> voliDisponibili =
  orarioStagionale.stream().filter(fs ->
                  fs.getDepartureAirport().equals(departureAirport) &&
                  fs.getArrivalAirport().equals(arrivalAirport) &&
                  fs.getDaysOfWeek().contains(dayOfWeek))
          .collect(Collectors.toList());
```