Stream pt.2

Metodo filter()

Prende in un espressione lambda e ritorna un booleano: se è true, allora l'elemento viene messo nello stream di ouput che contiene gli elementi valutati dall'espressione lambda.

- Filter non è terminale e restituisce uno *Stream*
- il metodo `count()` serve per contare gli elementi in un determinato Stream

```
List<String> nomi = "Pippo", "Pappa", "Poppo";

long c = nomi.stream().filter(s -> s.lenght == 5).count(); // 3
```

Tipo Predicate<>

Predicate è un tipoc che contiene un'espressione lambda che poi viene passata ad una Filter.

```
Predicate<Integer> positive = x -> x >= 0;

Stream<Integer> result = Stream.of(2, -1, -5, 34, 3).filter(positive); // result = [2,34,3]
```

Ne segue che Predicate e Stream devono corrispondere ai tipi ***NON PRIMITIVI*** (*Integer, Long, Double*, ...) passati in input alla funzione `filter()`.

Metodo reduce()

```
recude(T identity, BinaryOperator<T> accumulator); // T è uguale al tipo di oggetti dello Stream

reduce(0, (accum,v) -> accum + v); // parte da 0 e vengono ad accum si sommano gli elementi "v" dello Stream
```

Alla prima iterazione:

- `accum = 0` perchè è il valore di partenza specificato
- `v` corrisponde primo valore dello stream
- L'espressione lambda in `reduce()` prende *due parametri in input* e ne *restituisce un solo valore (*accum*).
- `reduce()` **è un'operazione terminale** e si usa quando si vuole passare da un insieme di valori ad un singolo valore.

Riferimenti a metodi

Ci si può riferire a metodi dall'interno di una classe.
Il codice si troverà all'interno dentro un metodo specifico di una determinata classe. Il metodo non viene chiamato direttamente ma attraverso il richiamo della classe.

```
reduce(0, Integer::sum); // -> restituisce T
reduce(Integer::sum) // -> restituisce Optional<T>. Invocare isPresent()
```

All'interno di Integer ci sarà il metodo `sum()` e rappresenta un'espressione lambda visto che reduce() accetta solo espressioni lambda.

- Ovviamente il risultato non esisterà se lo Stream sarà vuoto. In questo caso, la reduce con 2 parametri restituisce 0.
- Se, invece, la reduce ha un solo parametro, quindi `reduce(Integer::sum)`, allora restituisce un tipo `Optional<T>` che è un contenitore che **può avere/non avere** un risultato.
  - *Se lo Stream è vuoto allora esso sarà vuoto*
  - Su `Optional<T>` si può invocare `isPresent()` o `isEmpty()` per vedere se il contenitore è pieno o no, oppure se la reduce ha restituito un risultato oppure no.
- **Bisogna accertarsi, prima di andare avanti, che la `reduce()` abbia restituito un valore** *per evitare errori*
- `reduce()` può prendere in ***ingresso solo l'espressiona lambda***, quindi senza il parametro di inizializzazione della variabile. In questo caso, il metodo prende come parametri iniziali (*run-time*) il primo elemento e il secondo dello Stream

```
// Esempio uso reduce()
public class Pagamenti {
        private List<Float> importi = new ArrayList<>();

        // in stile imperativo
        public float calcolaSommaImper() {
                float risultato = 0f;
                for (float v : importi)
                        risultato += v;
                return risultato;
        }

        // in stile funzionale
        public float calcolaSomma() {
                return importi.stream().reduce(0f, Float::sum);
        }
}
```

Metodo map()

Si applica ad uno Stream e prende in input un'espressione lambda. Produce uno Stream in uscita e li ***TRASFORMA*** lo Stream.

- Per trasformazione si intende: cambiare il tipo degli elementi di partenza, cambiarne i valori raddoppiandoli ecc...
- Il tipo in uscita di map() può, quindi, essere diverso dal tipo di elementi nello Stream iniziale

```
map(Function<T,R> mapper); //T è il tipo in input. R è il tipo in output

// Esempio
List<Integer> l = List.of(1,2,5);
Stream<Integer> s1 = l.stream().map (x -> x * 2); // risultato = [2,4,10]
List<Integer> s2 = s1.toList(); //converte in List uno Stream
```

```
//----------------------------
List<Persona> l = List.of(new Persona("Pippo", 46), new Persona("Alessio", 18));
Stream<Integer> result = l.stream().map(Persona::getEta); // result = [46,18]
// l.stream().map(p -> p.getEta()); // chiamata equivalente

map(Persona::getEta); // Per ogni elemento dello Stream si chiama il metodo getEta presente in Persona.
```

- Calcoliamo la somma delle età delle istanze di Persona

```
public class Persona {
   private String nome;
   private int eta;
   public Persona(String n, int e) {
      nome = n;
      eta = e;
   }
   public String getNome() {
      return nome;
   }
   public int getEta() {
      return eta;
   }
}
```

```
// in stile imperativo
int somma = 0;
for (Persona x : amici)
      somma += x.getEta();
```

```
// alternativa in stile funzionale
int somma =
   amici.stream()
        .map(ps -> ps.getEta())
        .reduce(0, (s, e) -> s + e);
```

```
List<Persona> amici = Arrays.asList(
  new Persona("Saro", 24), new Persona("Taro", 21),
  new Persona("Ian", 19), new Persona("Al", 16));

// somma calcolata in stile funzionale con i riferimenti ai metodi
int somma = amici.stream()
             .map(Persona::getEta)
             .reduce(0, Integer::sum);
```

- La funzione passata a `map()` è il metodo `getEta()` di Persona

Stile dichiarativo vs funzionale

```
// Data una lista contenente valori *String*:

List<String> nomi = Arrays.asList("Saro", "Taro", "Ian", "Al");

// Per determinare se la lista contiene un certo valore, in stile dichiarativo:

if (nomi.contains("Saro")) System.out.println("Saro trovato");

// Data una lista contenente istanze di Persona:
List<Persona> amici = Arrays.asList(new Persona("Saro", 24), new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 21));
//Lo stile funzionale consente di estrarre il campo nome, e inoltre permette di valutare una funzione ad-hoc, quindi è molto più flessibile e potente

long c2 = amici.stream().filter(s -> s.getNome().equals(“Taro”)).filter(s -> s.getEta() == 21).count();
```

- count() conta quanti elementi ha lo stream prodotto da filter
  - Data la lista di istanze di Persona, trovare il nome della persona che è più grande (di età) fra quelli che hanno meno di 20 anni

```
List<Persona> amici = Arrays.asList(new Persona("Saro", 24),
  new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));
```

  - In versione imperativa, se volessimo scorrere la lista solo una volta

```
Persona pmax = null;
for (Persona ps : amici)
  if (ps.getEta() < 20) {
     if (pmax == null) pmax = ps;
     if (pmax.getEta() < ps.getEta()) pmax = ps;
  }
if (pmax != null) System.out.println("persona: " + pmax.getNome());
```

  - Il corpo del ciclo ha varie condizioni, queste rendono il codice più difficile da comprendere

Ricerca funzionale, v1.0

- Aggiungendo su Persona il metodo getMax()

```
/** restituisce l'istanza con il valore massimo di eta' */
public static Persona getMax(Persona p1, Persona p2) {
    if (p1.getEta() > p2.getEta())
        return p1;
    return p2;
}
```

- Possiamo implementare la ricerca in modo funzionale

```
Optional<Persona> pmax = amici.stream()
                          .filter(x -> x.getEta() < 20)
                          .reduce(Persona::getMax);

if (pmax.isPresent())
        System.out.println("persona: " + pmax.get().getNome());
```

- **filter()** è usata per separare gli elementi che soddisfano la condizione sull'età, prende in ingresso la funzione (predicato) da eseguire su ciascun elemento
- **reduce()** è usata per selezionare un elemento: invoca **getMax()** che confronta a due a due

Ricerca funzionale, v2.0

```
Optional<Persona> pmax = amici.stream()
                      .filter(x -> x.getEta() < 20)
                      .max(Comparator.comparing(Persona::getEta));
if (pmax.isPresent())
     System.out.println("persona: " + pmax.get().getNome());
```

- **filter()** è usata per separare gli elementi che soddisfano la condizione sull'età, prende in ingresso la funzione (predicato) da eseguire su ciascun elemento
- **max()** trova il valore massimo, è *un'operazione terminale*, prende un **Comparator**, restituisce un **Optional**, **max()** opera in modo simile a **reduce()**
- **Comparator.comparing()** prende una funzione che estrae una chiave e restituisce un **Comparator**
- **Comparator** implementa una funzione di confronto (**compare()**) che controlla l'ordinamento di una collezione di oggetti
- **max()** al suo interno chiama il metodo **compare()** del **Comparator** in input

- `Comparator.comparintg(Persona::getEta)` : il parametro per fare il confronto in `comparing()` è l'età della Persona.

- `comparing()`, al suo interno, chiama un altro metodo implementato, chiamato `compare()` e fa un confronto fra 2 parametri

Tipo Comparator

- `Comparator<T>` è una interfaccia funzionale, una sua implementazione permette di stabilire un ordine su una collezione di oggetti. Il metodo che definisce è `compare(T o1, T o2)`
- Se o1 == o2 restituisce 0
- Se o1 > o2 restituisce un valore positivo
- Se o1 < o2 restituisce un valore negativo
- Quindi posso implementare il `Comparator` nel seguente modo

```
Comparator<Persona> myComp = (p1, p2) -> p1.getEta() - p2.getEta();
Optional<Persona> pmax = amici.stream()
                .filter(x -> x.getEta() < 20)
                .max(myComp);
```

- Ovvero

```
                .max((p1, p2) -> p1.getEta() - p2.getEta());
```

Visto che mi occorre fare il confronto fra età, allora implemento un Comparator apposito.

Estrarre il massimo (come in questi esempi). Bisogna ricordare almeno un metodo per farlo (***ESAME***)

Metodo collect()

```
List<Persona> amici = List.of(new Persona("Saro", 24),
 new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));
```

- Ricaviamo la lista delle età

```
List<Integer> e = amici.stream()
                    .map(x -> x.getEta())
                    .collect(Collectors.toList());
```

- Come prima, `map()` restituisce uno stream con i valori delle età, e la funzione passata dice come trasformare ciascun elemento dello stream
- `collect()` permette di raggruppare i risultati e prende in ingresso un `Collector`
- La classe `Collectors` implementa metodi utili per raggruppamenti, il metodo `toList()` restituisce un `Collector` che accumula elementi in una `List`
- Java 16 dà il metodo `Stream.toList()`

```
// in stile imperativo
List<Persona> p; // come sopra
List<Integer> e = new ArrayList<>();
for (Persona x : p)
       e.add(x.getEta());
```

- `.toList()` trasforma in List e basta
- `.collect()` permette di trasformare in List o qualsiasi altra cosa, quindi risulta più **FLESSIBILE**

Programamzione Parallela

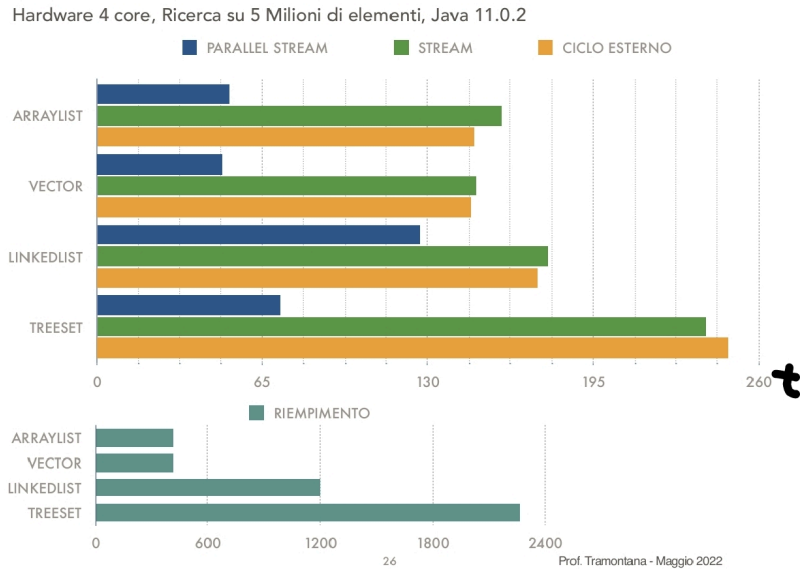
Quando si opera con gli Stream, è più utile andare in parallelo
Le istruzioni viste in precedenza si applicano sequenzialmente nel tempo agli elementi iniziali ma, se si vuole usare il parallelismo, si può fare:

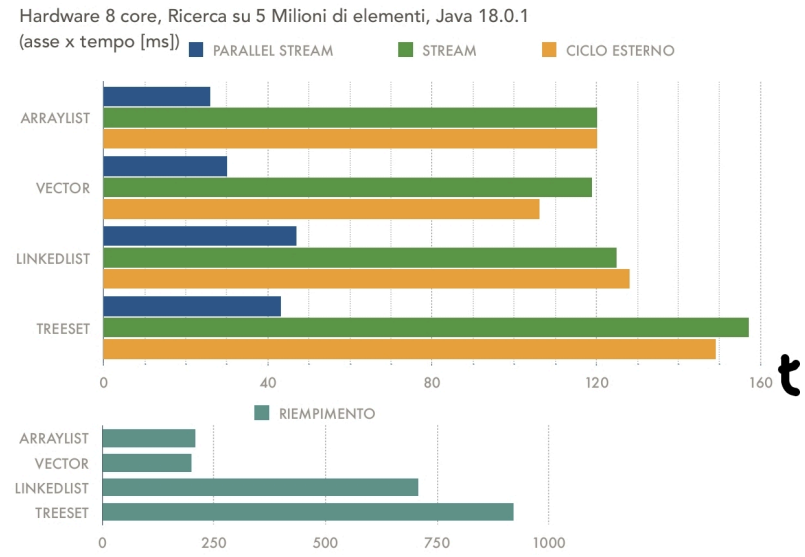
- piuttosto che invocare Stream si usa parallelStream() e dà uno Stream in parallelo
- Esiste un altro metodo, `parallel()` invocato sullo Stream

Le operazioni fatte in parallelo, dal punto di vista della correttezza, funzionano bene in tutti i casi se il programmatore sta attento a quello che fa.

- In generale, si produce sempre uno Stream (output) differente dallo Stream in input
- Tutto quello che avviene nello Stream iniziale può avvenire separatemente rispetto a quello che si produce in output e ciò **garantisce la CORRETTEZZA** visto che l'input non viene modificato in alcun modo (in particolar modo quando uso il parallelismo)
- I programmatori potrebbero sbagliare nello scrivere il codice:
  - l'esecuzione in parallelo non prevede di avere uno ***stato globale*** e quindi non si aggiorna uno stato globale e in questo caso il parallelismo va benissimo
  - In caso contrario, ***quindi si modifica uno stato globale***, ci si deve chiedere se *è necessario farlo*. Se serve farlo, bisogna *valutare e controllare di farlo correttamente*, controllando anhce l'*ordine delle operazioni* (*diventa complicato*).
  - Per ***STATO GLOBALE*** si intende una qualsiasi ***variabile/classe condivisa*** con le altre parti di codice

La programmazione parallela consente di avere un notevole guadagno nelle prestazioni (*ricerca di elementi*):





- La barra (*blu*) più "corta" indica tempi migliori con programmazione parallela

Considerazioni

- Le prestazioni migliorano in caso di parallelismo (basta inserire `.parallel()`)
- Se uso un LinkedList ci perdo rispetto all'uso di un ArrayList
- Se si usano gli Stream di default si possono avere dei guadagni in termini di tempo e prestazioni man mano che l'hardware si aggiorna
- Il metodo `parallelStream()` ***POSSIBILMENTE*** da uno Stream parallelo perchè si deve valutare se il parallelismo è supportato dall'hardware prima di avviare tale procedura. Viene valutata anche la dimensione lo Stream e la sua dimensione: se è molto poca allora viene scandita in maniera sequenziale perchè ci vuole meno tempo rispetto all'avvio del parallelismo stesso

Esempio 1

- Data una lista di istanze di Persona trovare i nomi delle persone che sono giovani ed hanno ruolo Programmer, e ordinare i risultati

```
List<Persona> team = List.of(new Persona("Kent", 29, “CTO"), new Persona("Luigi", 25,
"Programmer"), new Persona("Andrea", 26, "GrLeader"), new Persona("Sofia", 26,
"Programmer"));

team.stream()
    .filter(p -> p.giovane())
    .filter(p -> p.isRuolo(“Programmer”))
    .sorted(Comparator.comparing(Persona::getNome))
    .forEach(p -> System.out.print(p.getNome() + " "));

// Output: Luigi Sofia
```

- `filter()` operazione intermedia che restituisce gli elementi che soddisfano il predicato passato
- `sorted()` operazione intermedia **stateful** che restituisce uno stream che ha gli elementi ordinati in base al Comparator passato
- `comparing()` permette di estrarre la chiave per il confronto
- `forEach()` operazione terminale che esegue un'azione su ciascun elemento dello stream (su uno stream parallelo l'ordine non è garantito)

Prof. Tramontana - Maggio 2019

- `.sorted()` ordina i risultati secondo una caratteristica passata in input tramite `Comparator.comparing(Persona::getNome)` e quindi si ordina secondo il nome della persona
  - è un'**operazione intermedia**, detta ***STATEFUL***, cioè hanno visione di tutti gli elementi dello Stream per lavorare
  - Generalizzando, i metodi `min()`, `max()` e `sorted()` vogliono in input un *Comparator* dello stesso formato
- `.forEach()` è **terminale** e permette di ***eseguire un'operazione per ogni elemento*** dello Stream ed ***ELIMINA LO STREAM***. Quindi non ha un valore di ritorno.
  - In questo caso, lo ***Stream viene eliminato*** e viene stampato solo l'informazione richiesta con `System.out.println()`
  - (**Attenzione!**) Segue che, **DOPO** `.forEach()`, **NON** si possono invocare altri metodi di Stream visto che esso viene ***ELIMINATO***. Tutte le operazioni che si vogliono invocare devono essere chiamate prima di di invocare tale metodo, appunto
- E' sempre meglio fare 2 filter piuttosto che due condizioni in AND.
- I `filter()` ***concatenati*** sono ***AND LOGICI*** e quindi vengono selezionati gli elementi dello Stream che ***soddisfano ENTRAMBI I FILTER*** espressi.

Esempio 2

- Data una lista di istanze di Persona trovare i diversi ruoli

```
team.stream()
    .map(p -> p.getRuolo())
    .distinct()
    .forEach(s -> System.out.print(s+" "));

// Output: CTO Programmer GrLeader
```

- `distinct()` operazione intermedia **stateful** che restituisce uno stream di elementi distinti
- Data una lista di istanze di Persona trovare il nome di una che ha il ruolo Programmer

```
Optional<Persona> r = team.stream()
                          .filter(p -> p.isRuolo("Programmer"))
                          .findAny();
if (r.isPresent()) System.out.println(r.get().getNome());

// Output: Luigi
```

- `findAny()` (simile a `findFirst()`) operazione terminale che restituisce un `Optional`, per valutarla non è necessario esaminare tutto lo stream, si dice short-circuiting (può far sì che alcune parti non vengano eseguite)

Prof. Tramontana - Maggio 2019

- `distinct()` permette di avere dei risultati ***NON DUPLICATI*** all'interno di uno Stream
  - è un'***operazione intermedia*** ed è *stateful*

Si può usare parallel() per l'operazione `sorted()` e ci penseranno le varie operazioni e le librerie a parallelizzare

Stateless vs Stateful

- Le operazioni map() e filter() sono stateless, ovvero non hanno uno stato interno, prendono un elemento dello stream e danno zero o un risultato
- Le operazioni come reduce(), max() accumulano un risultato. Quest'ultimo ha una dimensione limitata, indipendente da quanti elementi vi sono nello stream. Il risultato in una passata viene dato in ingresso alla passata successiva
- Le operazioni come sorted() e distinct() devono conoscere gli altri elementi dello stream per poter eseguire, si dicono stateful

Generare Stream: `iterate()`

- iterate() produce uno ***Stream infinito***
- Per questo motivo si chiama `limit()` che dice quante volte deve essere chiamata iterate(), e quindi il numero di elementi dello Stream
- `iterate(elemento_iniziale, espressione_lambda)` è la firma della funzione e lo Stream è riempito con gli elementi nel modo seguente:
  - valore iniziale seguito dal risultato dell'espressione lambda applicata all'elemento stesso e, chiaramente, deve essere compatibile con il tipo degli elementi
- Si conosce già il seme e il risultato dell'espressione lambda viene ***applicata al risultato dell'applicazione precedente***

```
Stream.iterate(2, n -> n * 2)
        .limit(10)
        .forEach(System.out::println);
```

Generare Stream: `generate()`

- Non prende parametri in ingresso ma applica la funzione in ingresso infinite volte (va aggiunto anche `limit()`)
- Il valore viene fornito in maniera indipendente rispetto agli altri valori dello Stream
  - Il metodo `generate()` permette di produrre uno stream infinito di valori, tramite una funzione di tipo `Supplier`, ovvero che fornisce un valore
    - `generate()` non applica una funzione ad ogni nuovo valore prodotto, come invece fa `iterate()`

```
Stream.generate(() -> Math.round(Math.random()*10))
      .limit(5)
      .forEach(System.out::println);
```

  - Il codice sopra genera uno stream di 5 numeri casuali, ciascuno fra 0 e 10

- `limit()` si può usare anche con filter() per indicare di selezionare solo un determinato numero massimo di elementi che "passano" il controllo
  - E' obbligatoria per `generate()` e `iterate()` e facoltativa per il resto

Tipo IntStream

Rappresenta uno Stream di valore Intero.

- `Stream<Integer>` e `IntStream` ***NON*** sono compatibili
- IntStream può avere ***SOLO*** valori interi
- *Mette a disposizione*:
  - `rangeClosed(1,6)` che produce un IntStream che comprende valori interi compresi fra gli estremi indicati (***inclusi***)
  - `.sum()` somma gli elementi di uno IntStream e restituisce, appunto, un valore intero.
    - Questa funzione ***NON E' DISPONIBILE*** per gli Stream normali

Conversioni Stream -> IntStream

```
int result =
 Stream.of("truth", "flows", "to", "them", "sweetly", "by", "nature")
       .mapToInt(x -> x.length()).sum();
// Output: result = 31
```

- `mapToInt()` esegue la funzione passata e restituisce un IntStream

```
Stream<Integer> s = IntStream.rangeClosed(1, 10).boxed();
```

- `boxed()` restituisce uno `Stream` di `Integer` a partire da un `IntStream`
- Si abbia la lista che contiene istanze di Persona, si generi uno stream contenente i primi 4 elementi

```
List<Persona> lista;

Stream<Persona> p =
   IntStream.rangeClosed(0, 3)
            .mapToObj(i -> lista.get(i));
```

- `mapToObj()` restituisce uno stream di oggetti a partire da un `IntStream`

- `mapToInt()` è un metodo di Stream e produce un IntStream
- Seguendo la serie di invocazioni, `sum()` è invocato su un tipo IntStream
- Da un IntStream posso generare uno `Stream<Integer>` invocando `.boxed()`
- `mapToObj()` è un metodo IntStream e produce Oggetti da mettere all'interno di uno Stream