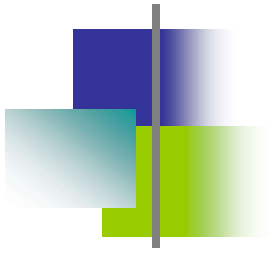
I nuovi stream di java 8



Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Introduzione agli Streams

- Le Collection rappresentano la libreria classica per raggruppare e **rappresentare** dati, ma per eseguire **operazioni** sui dati prevedono un approccio programmatico.

```
List<Dish> lowCaloricDishes = new ArrayList<Dish>();
for(Dish d: menu){
        if(d.getCalories() < 400)
             lowCaloricDishes.add(d);
}
```

- I nuovi **stream di Java 8** offrono uno strumento alternativo per compiere le più comuni **operazioni** sui dati ma con un approccio dichiarativo che ricorda la sintassi SQL

```
Select *

From Dish

Where calories < 400
```

Java8 introduce nel linguaggio (Java SE) il concetto di programmazione dichiarativa. La programmazione dichiarativa è alla base degli EJB e dei framework in genere

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Introduzione agli Streams

- Gli Streams sono un nuova libreria per la manipolazione e la gestione dei dati in una forma dichiarativa da usare con o in alternativa alle collections.
- **Esempio**: dato un menù (collection di piatti), si vogliono recuperare i nomi dei piatti meno calorici, ordinati per numero di calorie. **Vediamo la differenza tra Java 7 e Java 8**

```
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish d: menu){
        if(d.getCalories() < 400)
             lowCaloricDishes.add(d);
}

Collections.sort(lowCaloricDishes,new Comparator(){
    public int compare(Dish d1, Dish d2){
        return Integer.compare(d1.getCalories(),d2.getCalories());
    }
});
List<String> lowCaloricDishesNames = new ArrayList<>();
for(Dish d: lowCaloricDishes){
     lowCaloricDishesNames.add(d.getName());
}
```

Filtriamo i piatti in base alle calorie

ordiniamoli in base ad un criterio

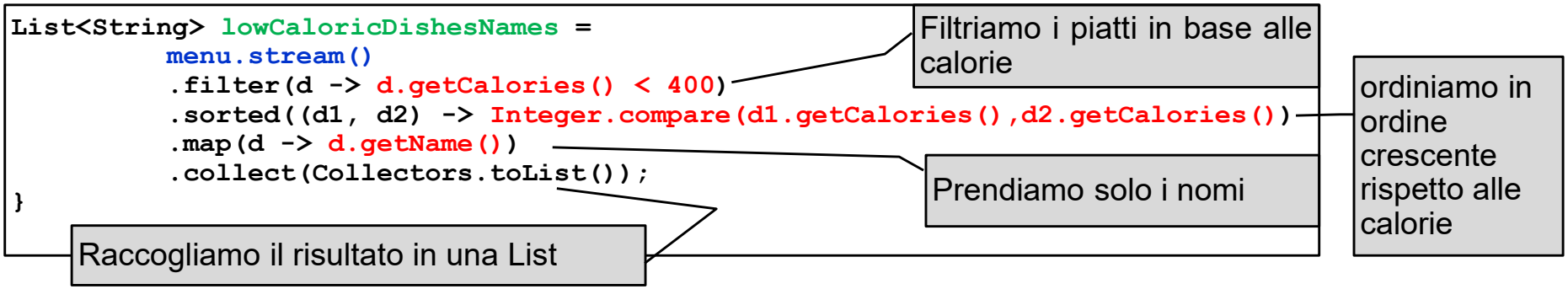
Prendiamo solo i nomi dei piatti

Java 7

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Introduzione agli Streams

- Con Java 8: usando i nuovi Streams e le lambda expression

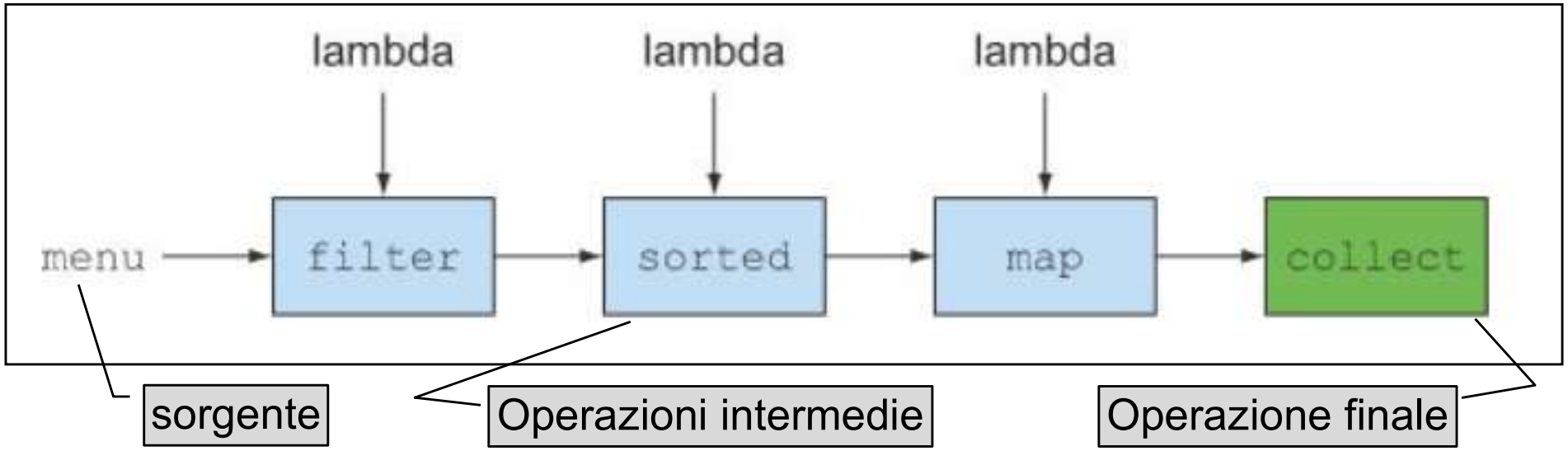


- Si nota
  - Un approccio **dichiarativo**
  - Una **pipenile** (catena di operazioni) → il risultato di una certa operazione è passato direttamente all'operazione successiva (come in una catena di montaggio)

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Introduzione agli Streams

- Schema della pipeline delle operazioni dal precedente esempio:



- Possiamo dire che gli Stream sono:
  - Dichiarativi → usano le espressioni lambda
  - Componibili → si possono eseguire in ordine arbitrario
  - Parallelizzabili → prevedono di impostare (opzionalmente) una gestione multithreading

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Introduzione agli Streams

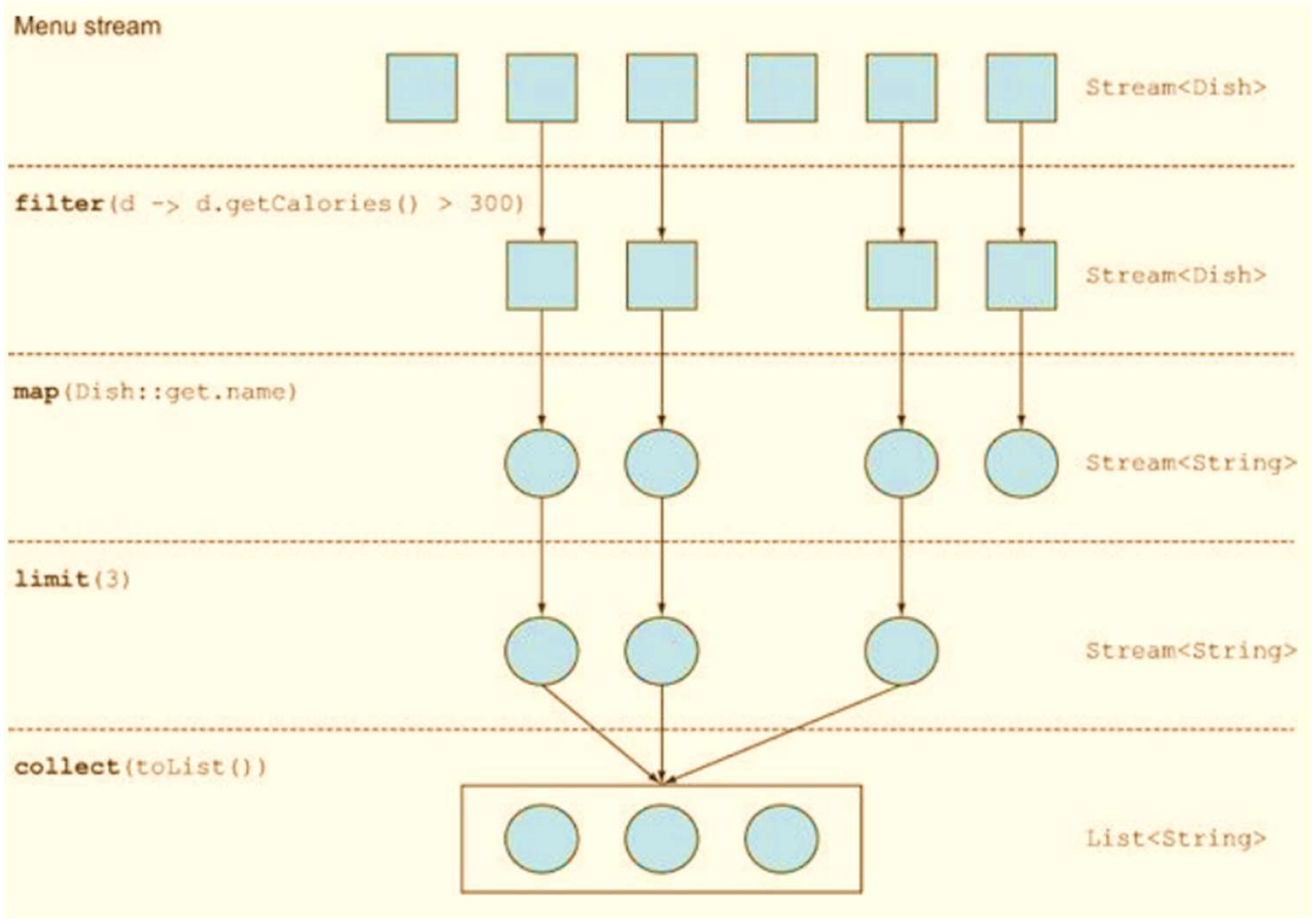
Un altro esempio, proviamo a schematizzare i passi

```
import java.util.stream.Collectors;
import java.util.List;
List<String> threeHighCaloricDishesNames=
         menu.stream()
         .filter(d -> d.getCalories() > 300)
         .map(d -> d.getName())
         .limit(3)
         .collect(Collectors.toList());
}
System.out.println(threeHighCaloricDishesNames);
```

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Introduzione agli Streams

Schema passi

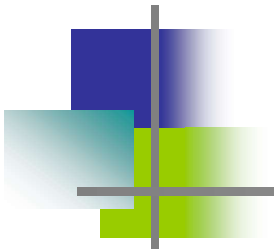


Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Introduzione agli Streams

- Uno Stream è un oggetto che rappresenta una **sequenza di dati** tratti da una **sorgente** su cui si intende fare delle **operazioni**
  - **Sequenza di dati** → come per le collections, anche gli stream vengono definiti attraverso una interfaccia che ne definisce il comportamento generale
    - La differenza fondamentale tra collections e streams è che le collezioni definiscono strutture dati mentre stream definiscono operazioni sui dati
  - **Sorgente** → gli streams non sono strutture dati, bensì sono oggetti che attingono i dati da una sorgente che può essere, ad esempio, una collection, un array o una periferica
    - I dati sono sequenziali nel senso che gli stream non cambiano la sequenza originale dei dati letta dalla sorgente
    - **Qualunque operazione facciamo attraverso gli stream, la sorgente resta inalterata**
  - **Operazioni** → gli streams supportano operazioni (stile SQL) come ordinamento, raggruppamento, ricerca, filtraggio ecc… Queste operazioni possono essere eseguite sequenzialmente o in parallelo

Dott. Romina Fiorenza
fiorenza.romina@gmail.com



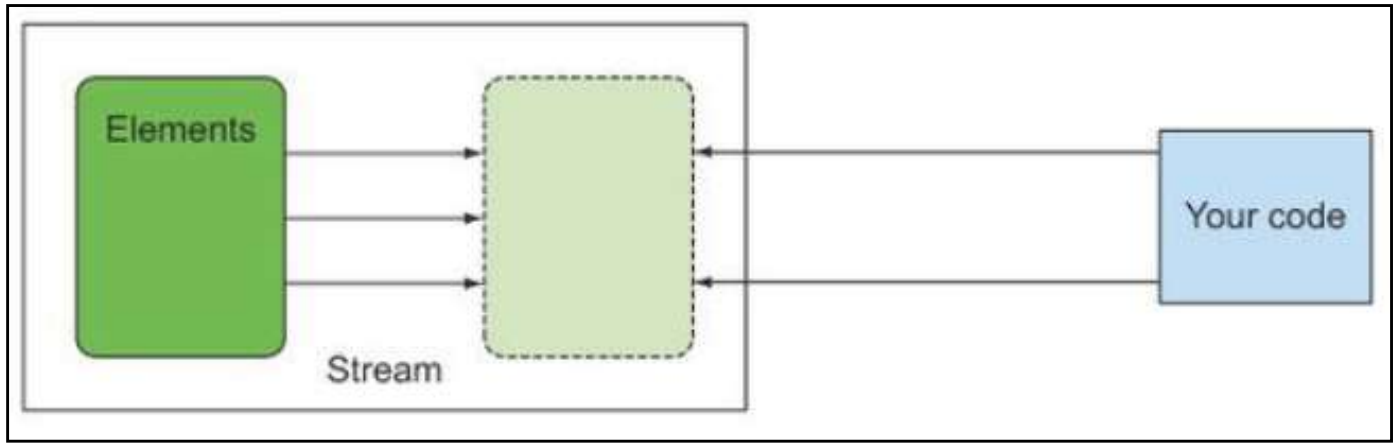
Collection vs Stream

Una collection ed uno stream sono concettualmente cose diverse.

- Una **collection** è una struttura dati che
  - Richiede la presenza di tutti i dati prima di iniziare una elaborazione su di essi
    - Alta occupazione di memoria
  - Prevede una elaborazione ogni volta in cui i dati vengono inseriti/eliminati
    - Ad esempio, un TreeSet ordina i dati al momento del loro inserimento/rimozione
  - Prevede l'uso di un iteratore esterno
- Uno **stream** è un flusso di dati che
  - Non richiede la presenza di tutti i dati ma devono essere disponibili solo quando vengono prelevati/generati dalla sorgente in base al loro consumo ("just-in-time" o "on demand");
    - Bassa occupazione di memoria
  - I dati sono forniti come flusso sequenziale e sono consumabili
    - Se un dato è consumato non può essere recuperato nuovamente
  - Prevede l'uso di un iteratore interno

Iteratore interno vs iteratore esterno

- Gli stream usano un iteratore interno mentre le collection quello esterno

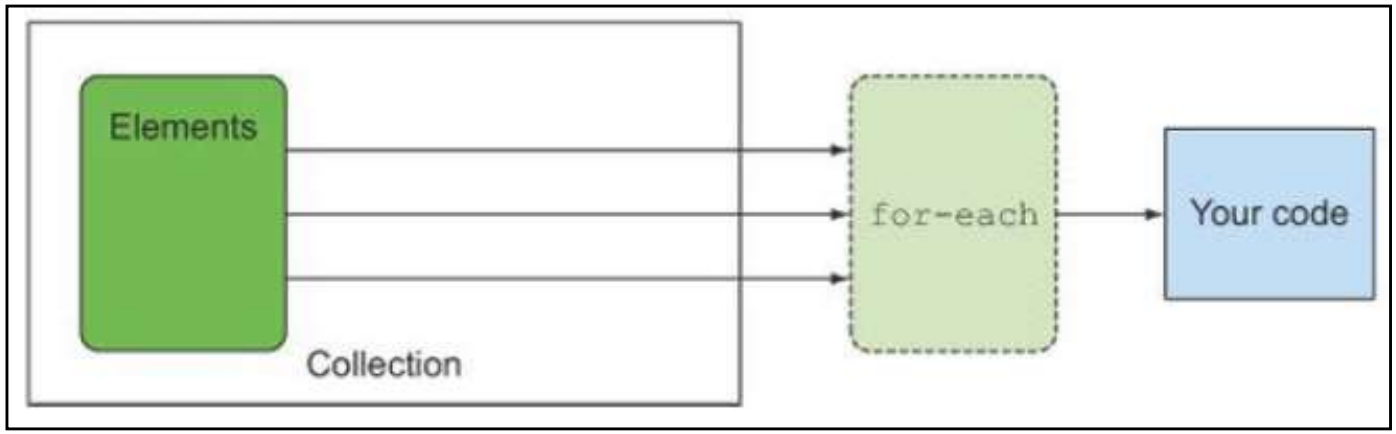


Stream: approccio dichiarativa.

```
List<String> names = menu.stream()
                          .map(Dish::getName)
                          .collect(toList());
```

Collection: approccio programmatico.

```
List<String> names = new ArrayList<>();
for(Dish d: menu){
    names.add(d.getName());
}
```



Dott. Romina Fiorenza
fiorenza.romina@gmail.com

L'architettura

- L'interfaccia ***java.util.Collection*** introduce due nuovi metodi per la creazione di uno stream a partire da una collezione di dati:
  - **`default Stream<E> stream()`** `//single thread`
  - **`default Stream<E> parallelStream()`** `//multiThread`

  Il comportamento di default di questi metodi è ritornare uno stream come semplice sequenza di elementi della collezione chiamante

- E'possibile comunque creare uno stream con i metodi statici della classe **`Stream:`**
  - **`static <T> Stream<T> empty()`** → costruisce uno stream vuoto
  - **`static <T> Stream<T> of(T t)`** → costruisce uno stream con un singolo oggetto T
  - **`static <T> Stream<T> of(T... values)`** → costruisce uno stream con 0… n oggetti di tipo T

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Stream: tipi di operazioni

- Le operazioni su uno stream sono di due tipi
  - **Operazioni intermedie:** filtraggio, raggruppamento, limitazione e selezione
  - **Operazione finale:** stoccaggio, ricerca, riduzione

  Le prime possono essere collegate a formare la pipeline coerente di operazioni sui dati mentre la seconda serve per chiudere la pipeline ed avviare l'effettiva esecuzione
- **Le operazioni intermedie** sono lazy: vengono processate solo al momento dell'operazione finale (in modo da **ottimizzare il processo**)
  - Per il programmatore le operazioni intermedie sono concettualmente separate, ma l'esecutore potrebbe riorganizzarle o fonderle per ottimizzare il processo
  - le operazioni intermedie vengono eseguite **solo se** l'operazione finale esiste e viene eseguita realmente
- **Le operazioni finali** chiudono lo stream generando un risultato.
  - Una volta eseguita l'operazione finale, lo stream è chiuso e non è possibile compiere ulteriori azioni → `java.lang.IllegalStateException: stream has already been operated upon or close`

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Stream: operazioni

- Le operazioni intermedie sono:

Operazione	Tipo di ritorno	Argomenti operazione	Descrizione chiamata lambda
filter	Stream<T>	Predicate<T>	T → boolean
map	Stream<R>	Function<T,R>	T → R
flatMap			
limit	Stream<T>	long	
sorted	Stream<T>	Comparator<T>	(T,T) → int
distinct	Stream<T>		
peek	Stream<T>	Consumer<? Super T>	(T) → void
skip	Stream<T>	long	

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione intermedia: filter

- Un'operazione molto comune è il filtraggio dei dati in base a dei criteri.
- La classe Stream offre i metodi **`filter`**() e **`distinct`**()

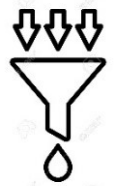
  - **`Stream<T> filter(Predicate<? super T> predicate)`** → ritorna uno stream i cui elementi corrispondono ai criteri definiti dal predicato, scarta i restanti

se ci fossero dei dati duplicati si possono eliminare con distinct()

  - **`Stream<T> distinct()`** → ritorna uno stream i cui elementi sono tutti diversi eliminando eventuali oggetti uguali in accordo con il metodo **`equals`**()

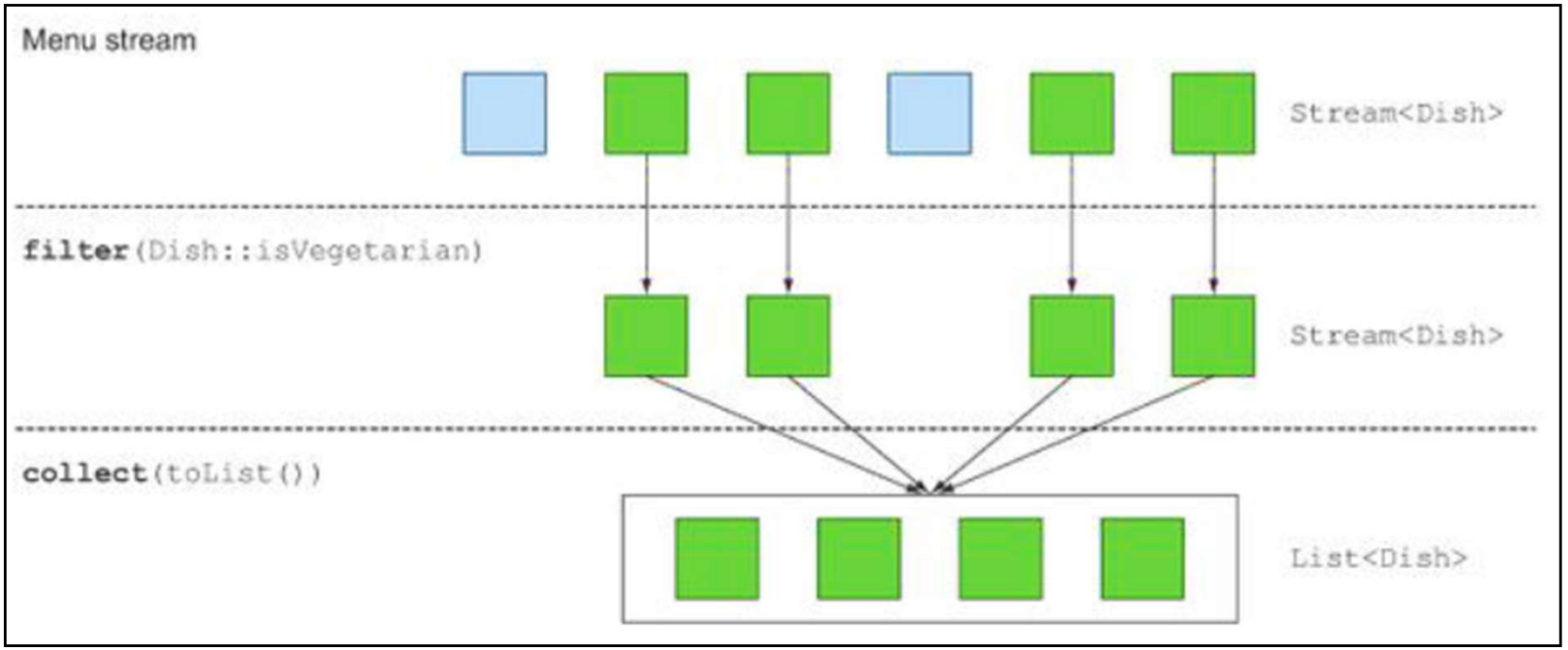
Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione intermedia: filter



- Filtraggio di uno stream attraverso un predicato.
  - Il risultato è un secondo stream che contiene solo gli elementi filtrati

```
List<Dish> vegetarianMenu =
          menu.stream()
          .filter(d -> d.isVegetarian())
          .collect(Collectors.toList());
```



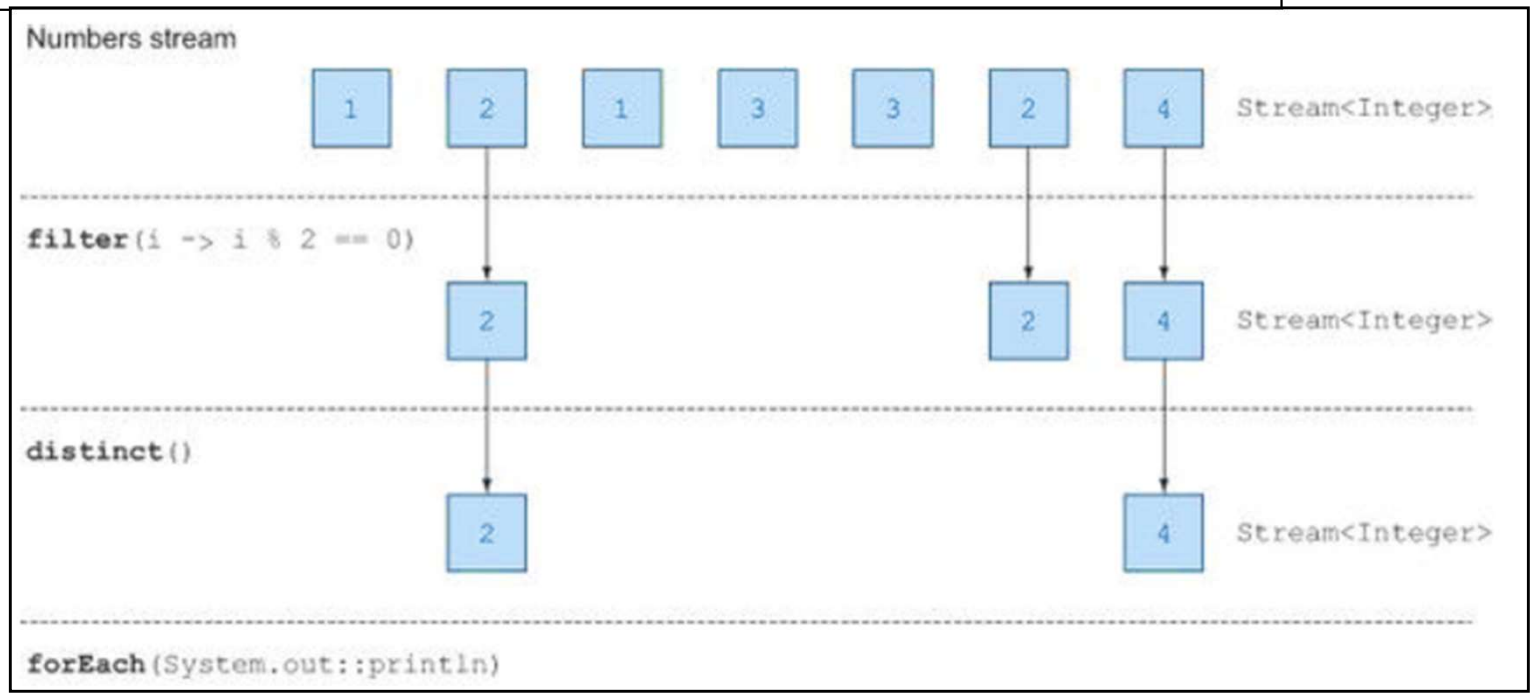
Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione intermedia: distinct

- Eliminazione dei doppioni → metodo **`distinct()`**

In questo caso elimina i doppioni dopo l'operazione di filtraggio

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);
numbers.stream()
            .filter(i -> i % 2 == 0)
            .distinct()
            .forEach(i -> System.out.println(i));
```



Dott. Romina Fiorenza
fiorenza.romina@gmail.com

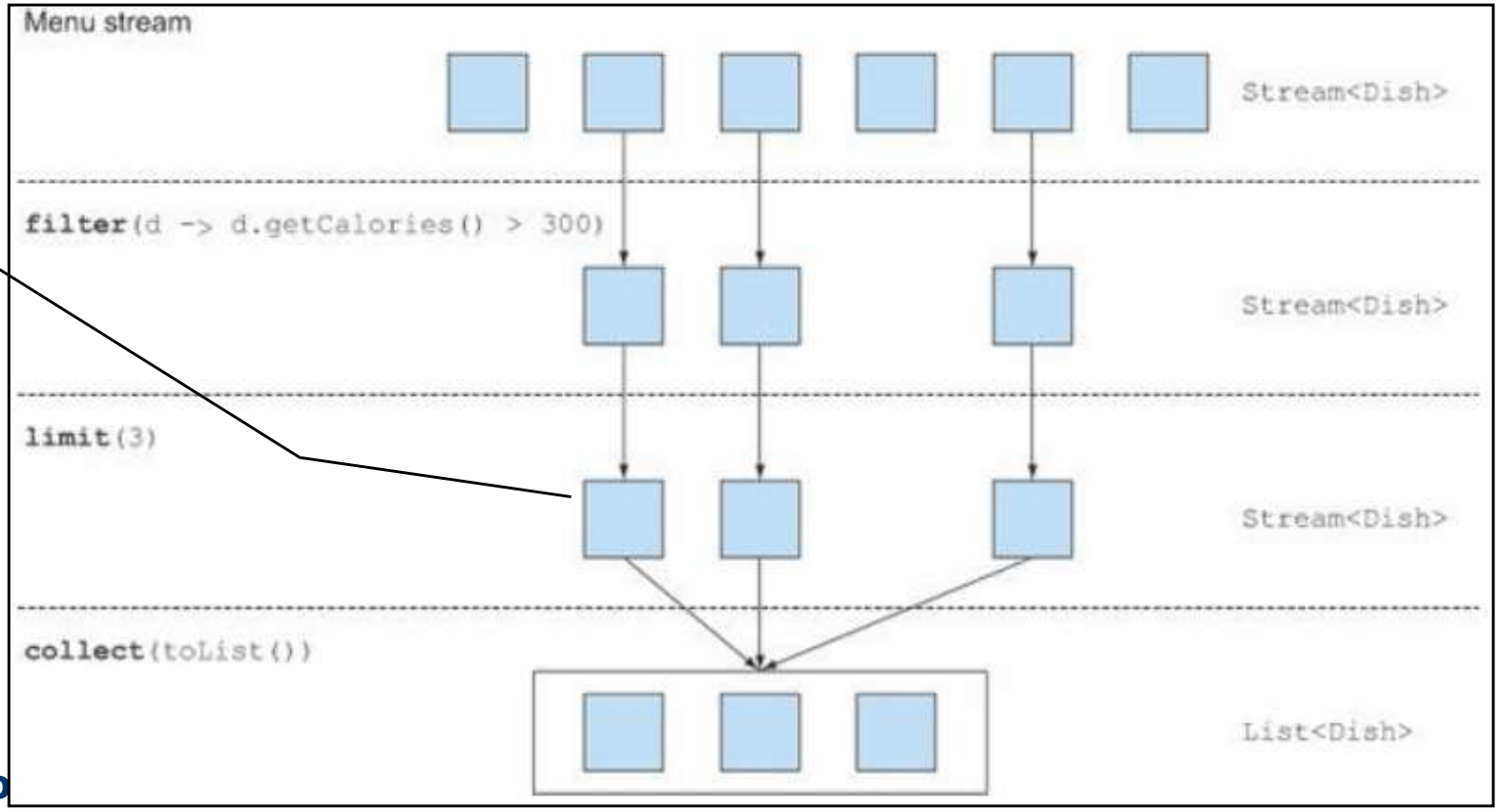
Operazione intermedia: limit

- E' possibile **limitare** il numero di elementi attraverso `limit()`

```
List<Dish> dishes = menu.stream()
          .filter(d -> d.getCalories() > 300)
          .limit(3)
          .collect(toList());
```

limit è performante: non appena 3 elementi sono disponibili, l'operazione termina e si passa allo step successivo (collect)

NB: gli stream rispettano la sequenza definita dalla sorgente.

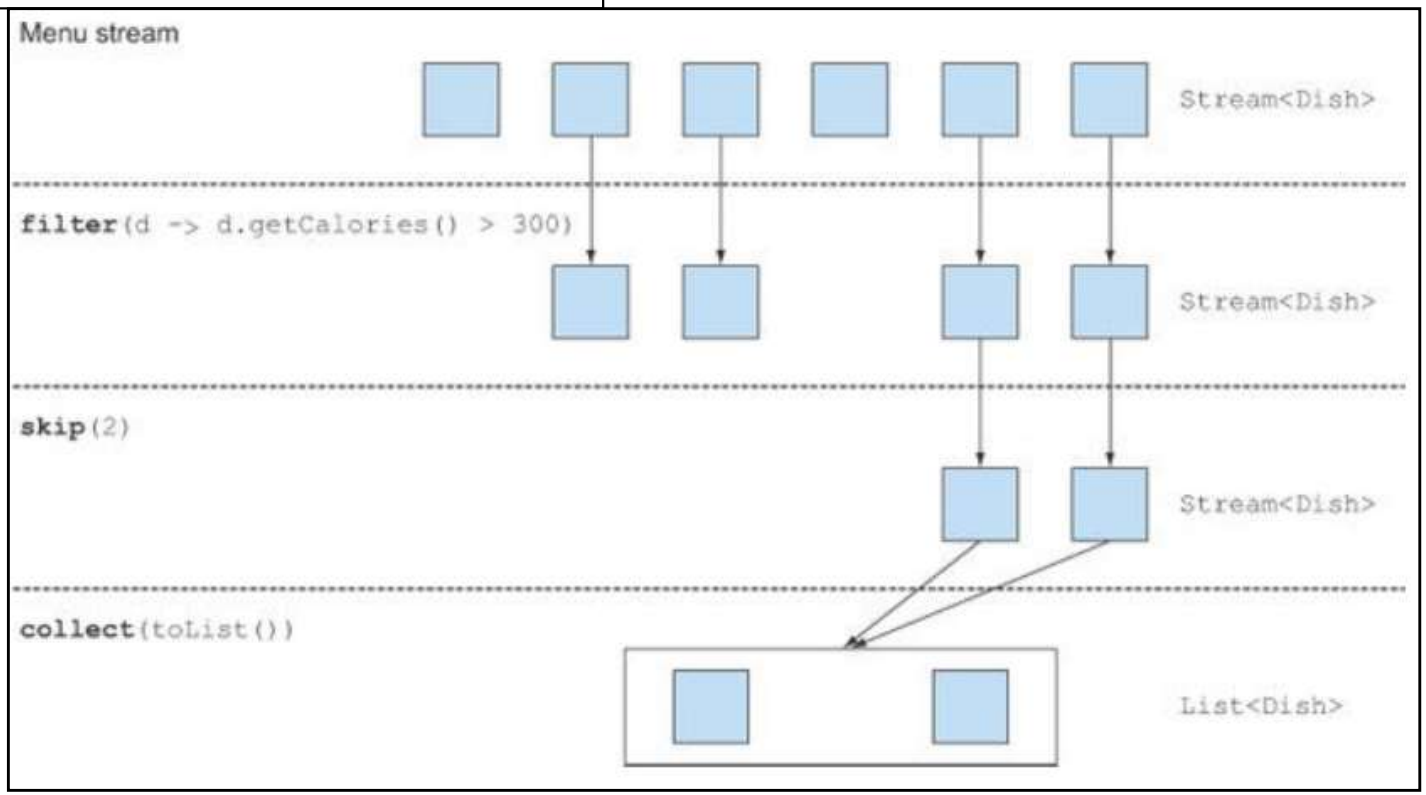


Dott. Romina Fio
fiorenza.romina@gmail.com

Operazione intermedia: skip

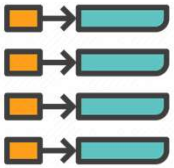
- E' possibile **scartare** un certo numero di elementi (a partire dall'inizio) → **`skip()`**

```
List<Dish> dishes = menu.stream()
          .filter(d -> d.getCalories() > 300)
          .skip(2)
          .collect(toList());
```



Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione intermedia: map



- Un'operazione molto comune sui dati è quella di selezionare solo alcuni campi di un oggetto dell'insieme.
- Gli stream consentono tale selezione tramite i metodi **`map()`** e **`flatMap()`**
  - Corrisponde alla clausola SELECT di una query sql
- Il metodo **`map()`** trasforma gli oggetti originali dello stream in oggetti differenti.
  - La trasformazione è basata sulla funzione passata come argomento al metodo

```
List<String> dishNames = menu.stream()
                             .map(d -> d.getName())
                             .collect(toList());
```

in questo esempio, al metodo **`map`** viene passata una funzione che invoca il metodo **`getName`** della classe **`Dish`**.

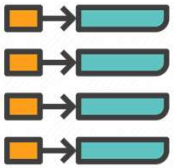
Il metodo **`getName`** viene invocato su ogni elemento dello stream.

Il metodo **`getName`** ritorna il nome del piatto, quindi ogni oggetto Dish dello stream viene trasformato in un oggetto String (mapping)

- Il risultato è una List<String> con i soli nomi dei piatti (Dish)

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione intermedia: map



- Il metodo **`map`** può essere invocato a cascata

secondo la normale catena di invocazione (**pipeline**).

Esempio:

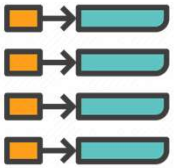
```
List<Integer> dishNamesLengths =
menu.stream()
          .map(d -> d.getName())
          .map(name -> name.length())
          .collect(toList());
```

si ottiene una **`List<Integer>`** con le lunghezze dei nomi di ogni piatto.

- Su ogni elemento dello stream viene invocato il metodo **`getName`** e poi sul risultato il metodo **`length`**

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione intermedia: flatMap



- Piccola variazioni sul tema è l'operazione **`flatMap()`**
  - trasforma uno stream di vettori in uno stream di singoli elementi

- Partendo dall'esempio precedente, data una lista di parole supponiamo di volere un elenco dei diversi caratteri presenti in queste parole, cioè eliminando i duplicati

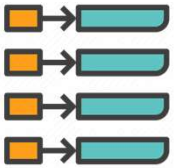
  Usando soltanto distinct non si raggiunge il risultato

```
List<Integer> dishNamesLengths =
         menu.stream()
         .map(word -> word.split(“”))
         .distinct()
         .collect(toList());
```

  si ottiene una **`List<String[]>`** , ogni elemento della lista è un array di tipo String. L'operazione **`distinct`** non porterebbe al risultato sperato perché valuterebbe l'uguaglianza degli array e NON dei suoi singoli elementi

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione intermedia: flatMap



- Il metodo **`flatMap`** esegue la conversione di uno stream

di array in uno stream di elementi di questi array

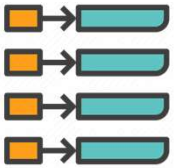
```
List<Integer> dishNameLengths =
menu.stream()
          .map(word -> word.split(“”))
          .flatMap(Arrays::stream)
          .distinct().
          .collect(toList());
```

si ottiene una List<String>

cioè ogni array derivato dall’operazione di split viene “fuso” in un unico stream con i suoi elementi (nell’ordine in cui si presentano)

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione intermedia: flatMap



I metodi sono:

- `<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)`

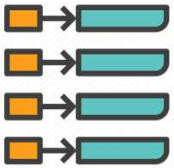
  NOTA: flatMap è una operazione intermedia che ritorna uno stream

  NOTA: fonde strutture come vettori in un unico stream, quindi due invocazioni consecutive portano ad un errore di compilazione

- `IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)`
- `LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)`
- `DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)`

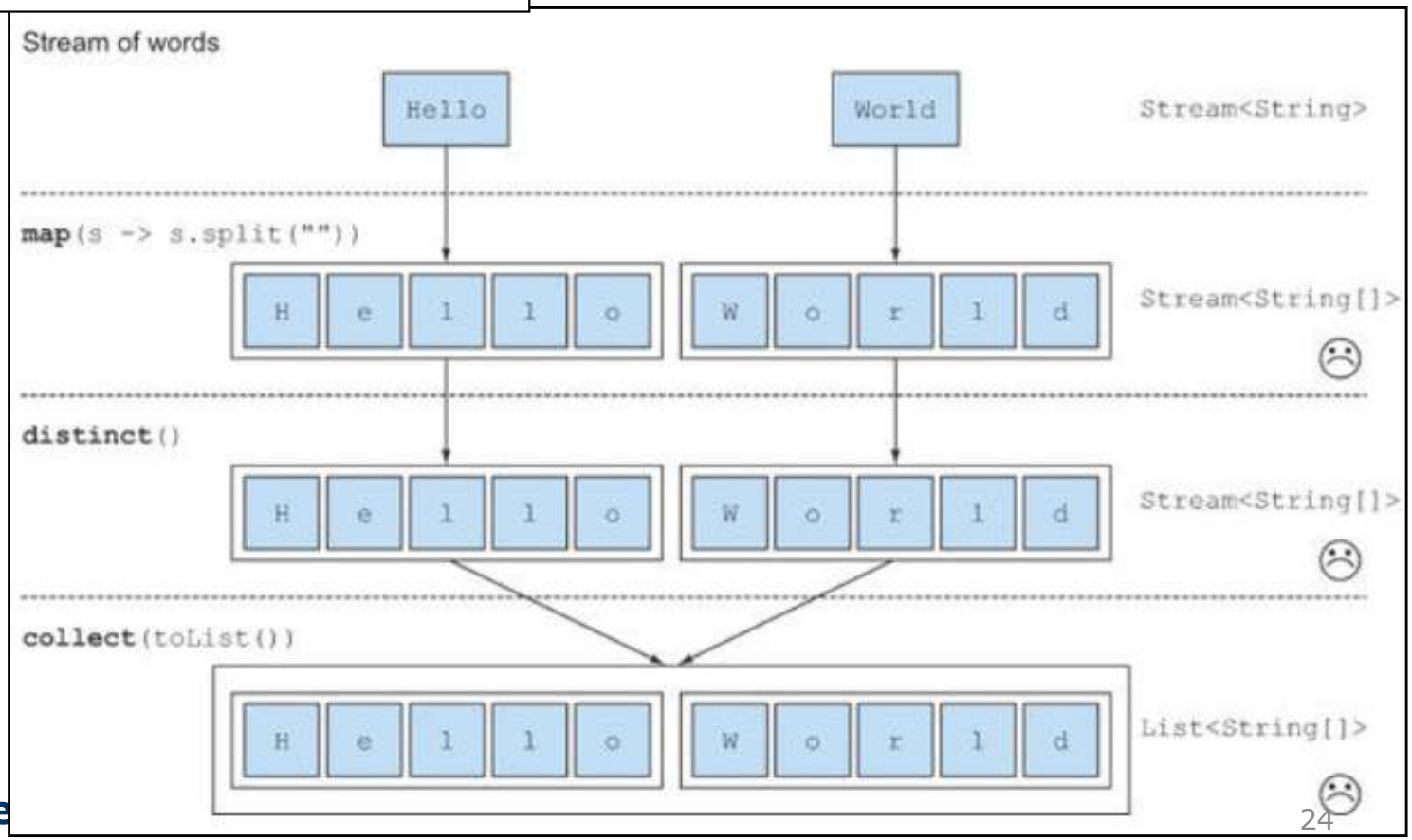
Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione intermedia: flatMap



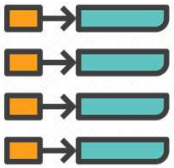
```
List<String[]> dishNameLengths =
        menu.stream()
        .map(word -> word.split(“”))
        .distinct()
        .collect(toList());
```

SENZA **flatMap**



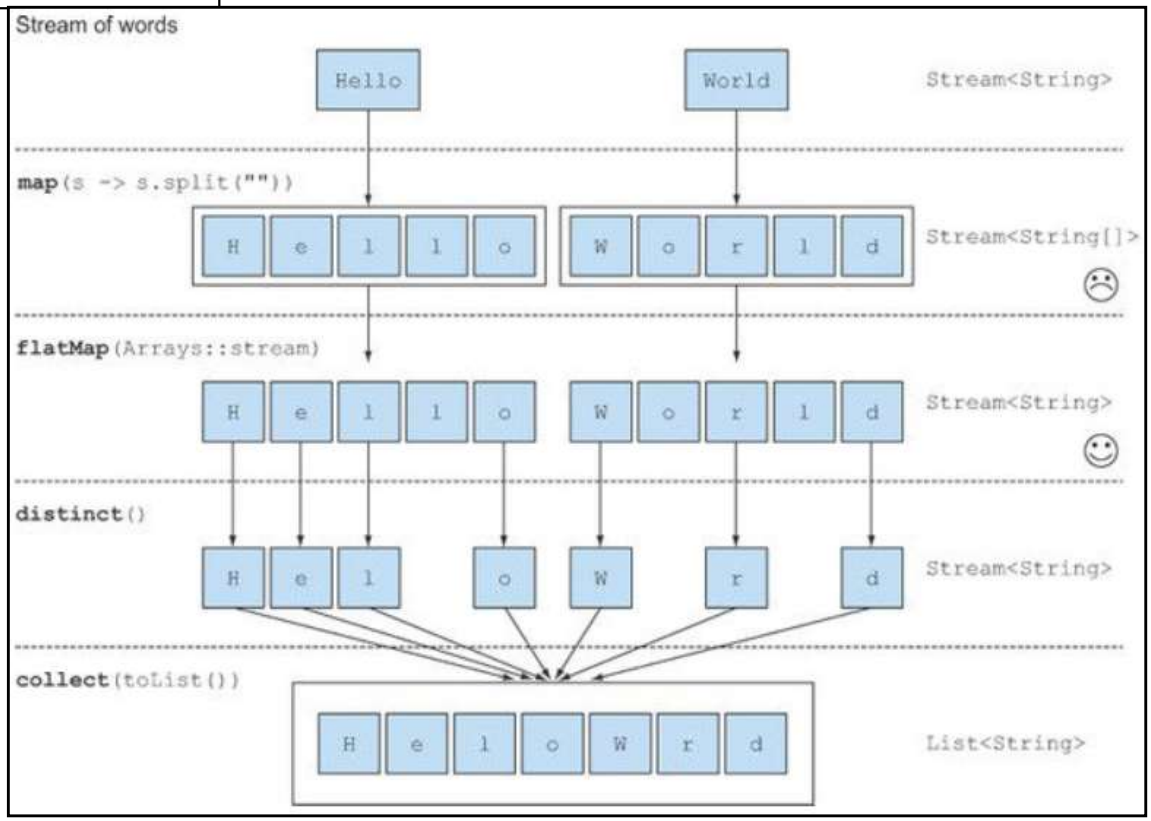
Dott. Romina Fiore
fiorenza.romina@gmail.com

Operazione intermedia: flatMap



```
List<Integer> dishNameLengths =
menu.stream()
        .map(word -> word.split(“”))
        .flatMap(Arrays::stream)
        .distinct().
        .collect(toList());
```

UTILIZZANDO **flatMap**



Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione intermedia: peek

- Metodo di visualizzazione → **`peek()`**

```
Stream<T> peek(Consumer<? super T> action)
```

Molto usato a scopo di test/debug, per osservare lo stato degli oggetti in un certo punto della pipeline.

Applica un certo comportamento su ogni elemento dello stream

```
List<String> list = new ArrayList<>();
list.add("one"); list.add("two"); list.add("three"); list.add("four");

List<String> filterList = list.stream()
              .filter(e -> e.length() > 3)
              .peek(e -> System.out.println("Filtered value: " + e))
              .map(e -> e.toUpperCase())
              .peek(e -> System.out.println("Mapped value: " + e))
              .collect(Collectors.toList());
```

la lista finale contiene [THREE, FOUR] mentre le **`peek`** stampano i valori degli oggetti durante le operazioni intermedie (le stampe appaiono alternate)

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione intermedia: sorted

- Per eseguire l'ordinamento abbiamo **`sorted()`**
- Consente di ordinare gli elementi di uno stream in base ad un criterio
- Esempio

```
List<String> list = new ArrayList<>();
list.add("one"); list.add("two"); list.add("three"); list.add("four");

list.stream()
     .peek(e -> System.out.println("original value: " + e))
     .sorted((s1, s2) -> s1.compareTo(s2))
 //  .sorted() equivalente alla chiamata precedente
     .peek(e -> System.out.println("Mapped value: " + e))
     .collect(Collectors.toList());
```

si ottiene una lista ordinata in ordine alfabetico [four, one, three, two]

- I metodi sono
  - **`Stream<T> sorted()`** → ordina in accordo con l'ordinamento naturale degli elementi
  - **`Stream<T> sorted(Comparator<? super T> comparator)`**

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

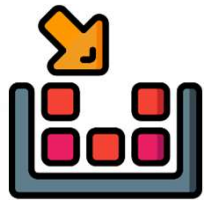
Stream: operazioni finali

Operazione	Tipo di ritorno	Argomenti operazione	Descrizione chiamata lambda
anyMatch	boolean	Predicate<T>	T → boolean
allMatch	boolean	Predicate<T>	T → boolean
noneMatch	boolean	Predicate<T>	T → boolean
findAny	Optional<T>		
findFirst	Optional<T>		
forEach	void	Consumer	T →void
collect	R	Collector<T,A,R>	
reduce	Optional<T>	BinaryOperator<T>	(T, T)→T
count	long		
max	Optional<T>	Comparator<T>	(T, T) → int
min	Optional<T>	Comparator<T>	(T, T) → int
average *	OptionalDouble		

* disponibile solo nelle classi specializzate come IntStream, LongStream e DoubleStream

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione finale: stoccaggio dei dati



- L'operazione finale più semplice è raccogliere i dati a seguito delle operazioni intermedie della pipeline e sistemarli in una Collection.

- A tale scopo, gli Stream mettodo a disposizione il metodo **`collect()`**
  - `<R,A> R collect(Collector<? super T,A,R> collector)`
  - `<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`

- Esempio:

```
List<Integer> dishNameLengths =
menu.stream()
          .map(d -> d.getName())
          .map(s -> s.length())
          .collect(Collectors.toList());
```

  ritorna una collection **`List<Integer>`** con i valori lavorati dalle operazioni intermedie

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione finale: ricerca

- Una delle operazioni più comuni e richieste è la ricerca → metodi **`xxxMatch()`**
- Le operazioni finali di tipo match sono utili per **verificare l'esistenza** di uno o più elementi in uno stream che corrispondono ad un certo criterio.

- Per le semplici verifiche la classe Stream mette a disposizione:
  - **`boolean anyMatch(Predicate<? super T> predicate)`** → restituisce true se lo stream contiene **almeno** una occorrenza che corrisponde al criterio di ricerca impostato
  - **`boolean allMatch(Predicate<? super T> predicate)`** → restituisce true se **tutti** gli elementi dello stream corrispondono al criterio di ricerca impostato
  - **`boolean noneMatch(Predicate<? super T> predicate)`** → restituisce true se **nessun** elemento dello stream corrisponde al criterio di ricerca impostato

NB: questo metodo non consente di sapere quali sono gli elementi che corrispondono al criterio!

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione finale: ricerca

- Esempio di uso di **anyMatch**

```
if(menu.stream().anyMatch(d -> d.isVegetarian()){
        System.out.println("The menu is (somewhat) vegetarian friendly!!");
}
```

restituisce true se almeno una pietanza è marcata come vegetariana

- Esempio di uso di **allMatch**

```
if(menu.stream().allMatch(d -> d.getCalories() < 1000){
        System.out.println("Yeah! All Dishes are for me!");
}
```

restituisce true se tutte le pietanze hanno una quantità di calorie minore di 1000

- Esempio di uso di **noneMatch**

```
if(menu.stream().noneMatch(d -> d.getCalories() >= 1000)){
        System.out.println("Yeah! All Dishes are for me!");

}
```

come il precedente, restituisce true se nessuna pietanza ha calorie pari o superiori a 1000, quindi come l'esempio precedente

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione finale: ricerca

- Ricerche più raffinate si realizzano con **`xxxFind()`**
- Le operazioni precedenti sono utili ma ritornano un boolean.
- Se dobbiamo ottenere uno o più elementi che corrispondono ai criteri impostati, possiamo usare i metodi **`xxxFind()`**
- I metodi sono:
  - **`Optional<T> findFirst()`** → ritorna un elemento (il primo) che corrisponde al criterio di ricerca impostato. **Quindi ha un comportamento deterministico**
  - **`Optional<T> findAny()`** → ritorna un elemento (uno qualsiasi) che corrisponde al criterio di ricerca impostato. **Quindi ha un comportamento non deterministico.**
    - Invocando più volte findAny() si potrebbero avere risultati sempre diversi

NOTA: tutte le operazioni di ricerca sono cortocircuitate.
Se durante l'operazione di match si deduce che il risultato sarà comunque true o false, evita di esaminare i restanti elementi dello stream

Se, ad esempio, durante l'operazione di ricerca con findAny() viene trovato un elemento, i restanti elementi dello stream non vengono esaminati

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione finale: ricerca

- I metodi di tipo **`find`** tornano un oggetto **`Optional<T>`**. Ad esempio:

```
Optional<Dish> o = menu.stream()
                        .filter(d -> d.isVegetarian())
                        .findAny();
```

ritorna una pietanza (una qualsiasi) vegetariana

```
Dish d = menu.stream().findAny().get();     // ritorna una pietanza (una qualsiasi)
Dish d = menu.stream().findFirst().get();   //ritorna la prima pietanza
```

- La classe **`Optional<T>`** è stata introdotta in Java8 per gestire il tipo di ritorno per quei metodi che prevedono la possibilità di non tornare nessun risultato.
  - Normalmente, il metodo tornerebbe null ma il valore nullo si presta a errori

Un oggetto **`Optional`** rappresenta il risultato o la sua assenza.

- Potrebbe "contenere" un valore oppure no
- Se il valore è presente il metodo **`isPresent`**() torna true e **`get`**() torna il valore

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione finale: riduzione

- Un'altra operazione finale è quella di trarre dallo stream un singolo valore come risultato di una operazione (ad esempio di calcolo) → **`reduce()`**
  - In SQL corrisponderebbe alle operazioni di gruppo
- Trarre un singolo valore da uno stream significa applicare una certa funzione sugli elementi dello stream in maniera iterativa
  - L'operazione viene eseguita come in un for
- Esempio, se lo stream contiene i numeri {4,5,3,9} si potrebbe pensare ad una funzione di sommatoria che riduce lo stream ad un numero

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));
```

  - Il metodo sono i seguenti
    - `T reduce(T identity, BinaryOperator<T> accumulator)`
    - `Optional<T> reduce(BinaryOperator<T> accumulator)`
    - `<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`

Non essendoci un valore iniziale, se lo stream è vuoto, non c'è risultato

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione finale: riduzione

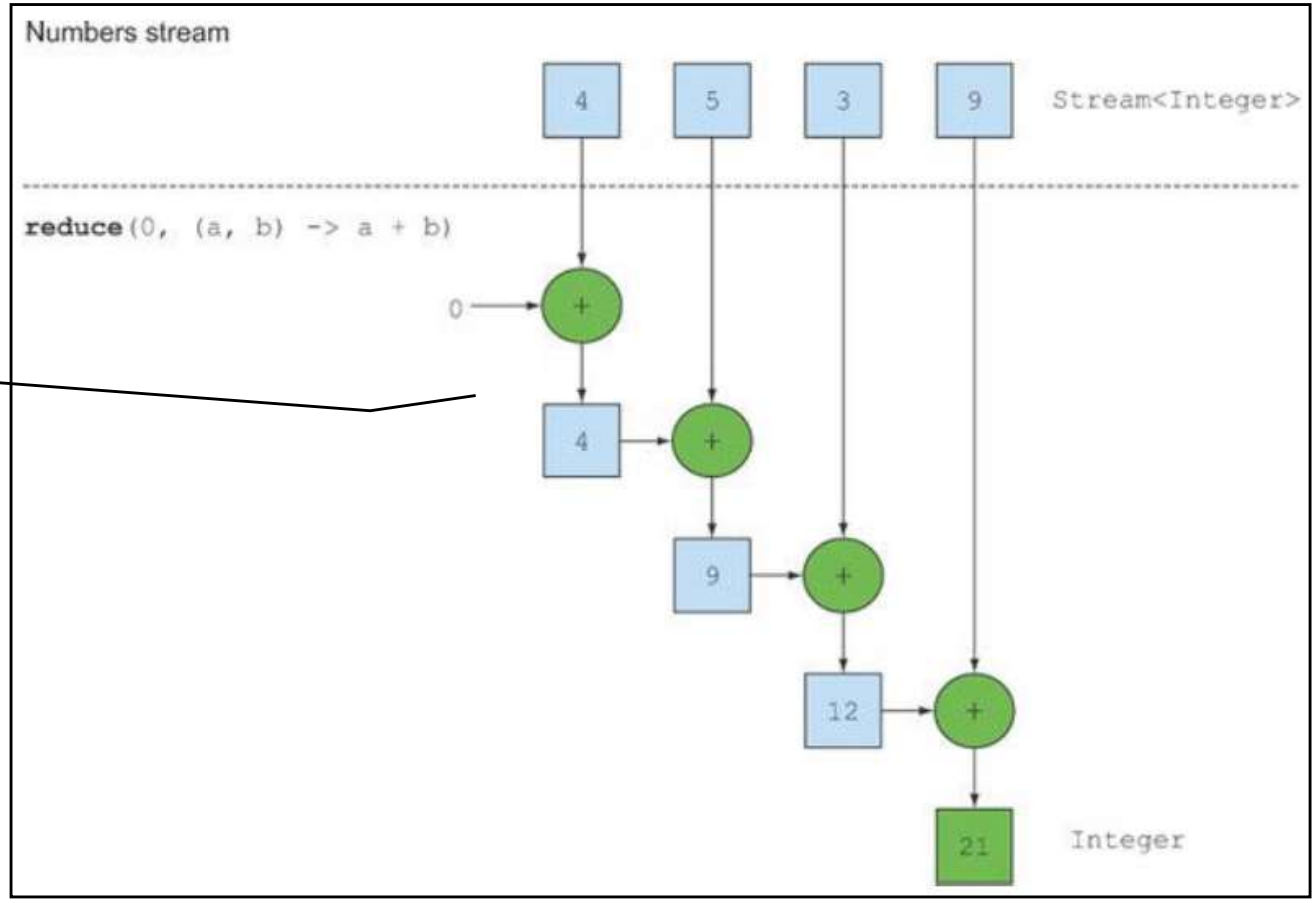
- Esempio di sommatoria:

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
int sum = numbers.stream().reduce(0, (a,b) -> Integer.sum(a,b)); //in alternativa
```

L'operazione Consumer sfrutta il concetto di accumulazione del dato.

I parametri dell' operazione consumer sono il dato accumulato e il prossimo valore dello stream

Il valore accumulato iniziale è il primo parametro del metodo (nell'esempio 0)



Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Nuovi metodi per Number

- Per agevolare l'uso del metodo reduce sono stati introdotti nuovi metodi nelle classi wrapper numeriche (classi di tipo **`Number`**)

- Nelle classi **`Short`**, **`Integer`**, **`Long`**, **`Float`**, e **`Double`**
  - **`static int min(int a, int b)`**
  - **`static int max(int a, int b)`**
  - **`static int sum(int a, int b)`**

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione finale: count, max, min

- Particolari tipi di riduzione sono:
  - **`long count()`** → conteggia il numero di elementi nello stream.
  - **`Optional<T> max(Comparator<? super T> comparator)`** → ritorna il valore massimo in accordo con l'oggetto Comparator passato come parametro
  - **`Optional<T> min(Comparator<? super T> comparator)`** → ritorna il valore minimo in accordo con l'oggetto Comparator passato come parametro

Nota: Se lo stream fosse vuoto, l'optional risulterebbe empty

Nota: esiste anche **`average()`** ma appartiene alle classi Stream specializzate

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Esempio: count()

- Dato il seguente codice:

```
List<String> lista = Arrays.asList("", "Red", "", "Green", "Black");

long val = lista.stream()
          .filter(n -> !n.isEmpty())
          .count();
System.out.println(val);
```

stampa il numero di elementi non vuoti dello stream, cioè 3

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione finale: forEach

- Per applicare un comportamento tutti gli elementi dello stream → **`forEach()`**

```
void forEach(Consumer<? super T> action)
```

NOTE:

- L'ordine non è garantito (nè con **`stream()`** nè con **`parallelStream()`**)
- Particolarmente nel caso di **`parallelStream`**(), l'odine con cui verrà applicata la funzione sugli elementi non è deterministico

- Per superare questo problema:
  - **`void forEachOrdered(Consumer<? super T> action)`**

L'ordine è garantito e le azioni sono fatte una dopo l'altra (senza sovrapporsi) ma è molto meno performante del precedente

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Specializzazioni

- Per facilitare l'uso degli stream con i **tipi primitivi**, Java8 mette a disposizione specializzazioni delle classi Stream per i tipi primitivi
  - `IntStream`
  - `DoubleStream`
  - `LongStream`

  Questi stream offrono metodi specifici per i rispettivi valori primitivi.

- **Esempio**: voglio sommare le calorie dei Dish dello stream:

  Questo codice NON compila perché il metodo **`map`**() ritorna uno Stream<Dish> che non possiede il metodo sum().

```
int calories = menu.stream()
                   .map(d -> d.getCalories())
                   .sum();
```

  Invece usando **`mapToInt`**

  compila e funziona, infatti **`mapToInt`** torna un **`IntStream`** che possiede il metodo **`sum`**()

```
int calories = menu.stream()
                   .maptoInt(d -> d.getCalories())
                   .sum();
```

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione finale: collect

- Di base, il metodo **`collect()`** trasforma i dati presenti in uno stream in una collection (cioè trasforma uno Stream in una Collection o Map)

- L'operazione però va vista come una forma di riduzione (metodo reduct()) perché gli elementi dello stream **vengono processati** prima di essere collocati nella collection finale
  - L'operazione di processing è definita formalmente dall'interfaccia ***Collector***
  - In una semplice trasformazione, come ad esempio collect(toList()), l'operazione di processing non c'è e quindi non vi sono effetti

- E' possibile definire riduzioni complesse che equivalgono alle operazioni di raggruppamento di SQL (clausola Group By di una Select oppure funzioni di gruppo SQL)

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Operazione finale: collect

- La classe **Collectors** possiede diversi metodi statici allo scopo di applicare una logica di processing predefinita. I principali sono:
  - **`toList`**() →nessuna operazione di processing.
  - **`counting`**() → torna il numero di elementi presenti nello stream
  - **`groupingBy`**() → raggruppamento in base ad un valore
  - **`maxBy`**() →tornano l'elemento massimo dello stream in base ad un comparatore
  - **`minBy`**() → tornano l'elemento minimo dello stream in base ad un comparatore
  - **`summingInt`**(), summingDouble(), summingLong()
  - **`summarizingInt`**(), summarizingDouble(), summarizingLong()
  - **`averagingInt`**(), averagingDouble(), averagingLong()
  - **`joining`**() → concatena tutti gli elementi di uno stream in una unica stringa
  - **`toMap`**() → accumula elementi in una mappa le cui chiavi e valori sono definite da altrettante funzioni passate come parametro
  - **`averagingDouble`**(), **`averagingLong`**(), **`averagingInt`** ()→ calcola la media dei valori

Nota: sono tutte varianti convenienti del più generale metodo reduce(), **quindi vanno viste come particolari operazioni di riduzione**

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Costruzione degli Stream

- Si possono costruire stream a partire da una collezione di dati già esistente.
- E' possibile anche costruire stream, compresi quelli specializzati, in diversi modi ed a partire da diverse origini
  - **Stream a partire dai dati** → uno stream costruito a partire da un elenco di dati
    - Metodi statici della classe Stream e specializzazioni in IntStream, DoubleStream e LongStream
  - **Stream a partire da un array**→ uno stream costruito a partire da un elenco di dati presenti in un array
    - Metodi statici della classe Arrays e specializzazioni in overloading per IntStream, DoubleStream e LongStream
  - **Stream a partire da un file**→ uno stream costruito a partire da un file
    - Metodi statici della classe Files
  - **Stream a partire da una funzione** → uno stream di infiniti dati costruito in base ad una funzione (NON mostrato in questo PDF)

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Stream a partire dai dati

- La classe Stream mette a disposizione il metodo statico of() in cui è possibile fornire un elenco (da 0 a n) di valori.
  - `static <T> Stream<T> of(T... values)`
  - `static <T> Stream<T> of(T t)`
- Ad esempio:

```
Stream<String> stream = Stream.of("Java 8 ", "Lambdas ", "In ", "Action");

stream.map(String::toUpperCase)
            .forEach(System.out::println);
```

  stampa l'elenco dei valori passati al metodo **`of()`**.

- E' possibile anche creare uno stream vuoto con l'apposito metodo **`empty()`**
  - `static <T> Stream<T> empty()`

```
Stream<String> emptyStream = Stream.empty();
```

- E' possibile anche ottenere uno stream come concatenzazione di altri 2
  - `static IntStream concat(IntStream a, IntStream b)`

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Stream a partire da un array

- La classe **Arrays** mette a disposizione il metodo statico **`stream()`** in cui è possibile fornire un elenco di valori sotto forma di vettore.

  Ad esempio è possibile convertire un vettore di interi in un **`IntStream`**:

```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();
Sytem.out.println(sum);
```

  stampa la sommatoria dei valori calcolati dal metodo **`sum()`**

Dott. Romina Fiorenza
fiorenza.romina@gmail.com

Stream a partire da un file

- La classe **`Files`** introdotta con Java NIO mette a disposizione diversi metodi statici che costruiscono uno stream a partire da un file
- Questi stream, leggendo dal file system possono sollevare errori di tipo IO

- Ad esempio il metodo **`lines()`** restituisce uno stream di tipo string i cui elementi sono le singole righe di un file di testo

```
try(
Stream<String> lines = Files.lines(Path.get("data.txt"),
Charset.defaultCharset()){
        //uso dello stream
}
```

Dott. Romina Fiorenza
fiorenza.romina@gmail.com