UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA
DIPARTIMENTO DI ECCELLENZA

Università di degli Studi di Salerno
Dipartimento di Informatica

Programmazione ad Oggetti

a.a. 2023-2024

Stream in Java

Docente: Prof. Massimo Ficco

E-mail: mficco@unisa.it

Gli Stream

Una interessante caratteristica di Java 8 è la possibilità di processare gli oggetti contenuti in una collection attraverso l'utilizzo di una nuovo strumento: gli Stream, utilizzando un approccio dichiarativo (simile all'SQL).

Uno stream rappresenta una sequenza di oggetti ottenuti da una specifica sorgente ai quali possiamo applicare una sequenza di operazioni.

Più formalmente uno stream ha le seguenti caratteristiche:

- Consente l'accesso in modo sequenziale ad un insieme di elementi di un tipo specifico;
- Gli elementi dello stream possono essere recuperati da una collezione, da un array o da una operazione di I/O;
- Lo stream supporta operazioni di aggregazione;
- Molte operazioni sugli stream restituiscono stream, quindi possono essere concatenate.

Caratteristiche degli Stram

- Gli Stream non sono correlati a InputStreams, OutputStreams, ecc.
- Gli Stream NON sono strutture di dati ma sono wrapper attorno alle Collection che **trasportano valori da un'origine attraverso una pipeline di operazioni**.
- Gli stream sono più potenti, più veloci e più efficienti in termini di memoria rispetto alle List.
- Gli Stream sono progettati per lambda espressioni.
- Gli Stream possono essere l'output di array o List
- Gli Stream utilizzano una valutazione lazy
- Gli Stream sono parallelizzabili
- Il flussi possono "on-the-fly"

Creare uno Stream

Per poter operare sugli stream è innanzitutto necessario instanziarne uno.
Allo scopo è stato introdotto un metodo specifico a secondo dell'origine dei dati:

- A partire da valori individuali:
  - Stream.of(val1, val2, …)
- A partire da array
  - Stream.of(someArray)
  - Arrays.stream(someArray)
- A partire da List (e alte Collections)
  - someList.stream()
  - someOtherCollection.stream()
- Uno Stream vuoto
  - Stream<String> emptyStream = Stream.empty()

Creare uno Stream

Un metodo specifico nell'interfaccia Collection:

```
List<String> items = new ArrayList<String>();
items.add("uno");
items.add("tre");
items.add("otto");
items.add("undici");

Stream<String> stream = items.stream();
```

Tipi primitivi

Non essendo possibile tipizzare lo Stream con tipi primitivi (*int*, *long* e *double*) Java 8 mette a disposizione 3 interfacce IntStream, LongStream e DoubleStream per la creazione di questi tipi dato.

```
IntStream intStream = IntStream.range(1, 10);
LongStream longStream = LongStream.range(1, 10);
DoubleStream longStream = intStream.asDoubleStream()
```

Tipi primitivi

Java 8 non fornisce un'interfaccia CharStream per la gestione dei *char* che possono essere invece gestiti com *IntStream*. La stessa class *String* possiede in metodo *chars()* che restituisce uno Stream dei caratteri presenti

```
IntStream charsStream = "rosso".chars();
```

Stream Processing

- Uno Stream viene elaborato attraverso una pipeline di operazioni
- Uno Stream inizia con una struttura di dati di origine
- I metodi intermedi vengono eseguiti sugli elementi Stream. Questi metodi producono flussi e non vengono elaborati finché non viene chiamato il metodo terminale.
- Uno Stream viene considerato consumato quando viene richiamata un'operazione terminale. Successivamente non è possibile eseguire altre operazioni sugli elementi Stream
- Una pipeline Stream contiene alcuni metodi short-circuit (che potrebbero essere metodi intermedi o terminali) che causano l'elaborazione dei metodi intermedi precedenti solo fino a quando non è possibile valutare il metodo short-circuit.

Anatomia di uno Stram

- **Intermediate Methods**

  map, filter, distinct, sorted, peek, limit, parallel

- **Terminal Methods**

  forEach, toArray, reduce, collect, min,

  max, count, anyMatch, allMatch, noneMatch, findFirst, findAny, iterator

- **Short-circuit Methods**

  anyMatch, allMatch, noneMatch, findFirst, findAny,limit

Anatomia di uno Stram

Tutte le operazioni disponibili si suddividono i 2 categorie:

- **Intermediate operation**, cioè tutte quelle operazioni che danno come risultato uno Stream<T>.

  E' molto importante ricordare che tutte queste operazioni sono **lazy**, vengono quindi eseguite solo se sono necessarie per eseguire una terminal operation. Altro aspetto importante da tenere in considerazione è che, restituendo uno Stream<T>, sono **concatenabili**, potranno quindi a loro volta eseguire un'altra intermediate operation.

- **Terminal operation,** ovvero tutte quelle operazioni che restituiscono un tipo definito (tipo primitivo, una collection, un void).

Stream Processing

Una volta ottenuto lo stream dalla Collection è possibile procedere all'elaborazione dei suoi elementi attraverso un processo a due fasi:

1. **Configurazione**: consiste in un processo di filtraggio e/o mappatura degli elementi;
2. **Elaborazione**: esegue l'operazione specificata sugli elementi filtrati o mappati.

Le operazioni di configurazione non vengono eseguite fino a quando non è avviata l'elaborazione.

Operazioni di Configurazione

Filtraggio

Per filtrare lo stream è sufficiente utilizzare il **metodo filter()** che riceve in ingresso un oggetto che implementa l'interfaccia **java.util.function.Predicate** che definisce un solo metodo con firma **boolean test(T t),** che riceve in ingresso un item della collezione e determina se filtrarlo o meno. Solo gli oggetti che passano il test vengono processati nelle successive fasi di elaborazione.

```
Stream<String> filteredStream = stream.filter( new Predicate<String>() {
   @Override
   public boolean test(String str) {
      return str.startsWith( "u" );
   }
});
```

oppure

```
Stream<String> filteredStream = stream.filter(str -> str.startsWith( "u" ));
```

Operazioni di Configurazione

Mapping

L'operazione di mapping consiste nel mappare gli oggetti della collezione in altri oggetti da essi derivati. Allo scopo l'oggetto Stream espone il metodo **map()** che riceve in ingresso un oggetto che implementa l'interfaccia java.util.function.Function. Tale interfaccia **Function <T,R>** contiene il metodo **R apply(T t),** che esegue il lavoro di mapping vero e proprio.

```
Stream<Integer> mappedStream = filteredStream.map(new Function<String,Integer>() {
    @Override
    public Integer apply(String str) {
        return str.length();
    }
});
```

oppure

```
Stream<Integer> mappedStream = filteredStream.map(str ->  {str.length()});
```

Operazioni di Elaborazione

Alcuni metodi:

Limit

Consente di ridurre la dimensione dello stream ad un valore massimo specificato eliminando gli item in eccesso.

```
Stream<String> limitedStream = mappedStream.limit( 2 );
```

Sorted

Il metodo consiste di ordinare lo stream. Può essere utilizzato senza parametri e quindi l'ordinamento sarà quello naturale associato al tipo di elementi dello stream, oppure può accettare un oggetto di tipo Comparator.

```
Stream<String> sordedStream = stream.sorted( new Comparator<String>() {
   @Override
   public int compare(String item1, String item2) {
      return Integer.valueOf( item1.length() ).compareTo( Integer.valueOf( item2.length() ) );
   }
});
```

Operazioni di Elaborazione

- **Void forEach (Consumer)**
  - Modo semplice per eseguire il loop sugli elementi Stream
  - Come argomento ha una lambda che viene chiamato su ogni elemento dello Stream
  - Il metodo peek correlato fa esattamente la stessa cosa, ma restituisce lo Stream originale

```
List<Employee> employees = getEmployees();
for(Employee e: employees) {
        e.setSalary(e.getSalary() * 11/10);
}

Employees.forEach(e -> e.setSalary(e.getSalary() * 11/10))
```

Operazioni di Elaborazione

Alcuni medodi:

- **Min e Max**

```
mappedStream.max( new Comparator<Integer>() {
  @Override
  public int compare(Integer o1, Integer o2) {
    return o1.compareTo( o2 );
  }
});
```

- **Count**

```
filteredStream.count();
```

- **Reduce**
- **Collect**
- **…..**

Funzioni Aggregate

- Nell'esempio del metodo processPersonsWithFunction() con l'uso dei generics più estensivo abbiamo visto la seguente implementazione:

```
public static <X, Y> void processElements( Iterable<X> source,
        Predicate<X> tester, Function <X, Y> mapper, Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

Funzioni Aggregate

- Nell'esempio del metodo processPersonsWithFunction() con l'uso dei generics più estensivo abbiamo visto la seguente implementazione:

```
public static <X, Y> void processElements( Iterable<X> source,
        Predicate<X> tester, Function <X, Y> mapper, Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

Questo metodo può essere invocato nel seguente modo:

```
processElements(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Funzioni Aggregate

- Nell'esempio del metodo processPersonsWithFunction() con l'uso dei generics più

Passiamo al metodo una raccolta di oggetti, di tipo Person, rappresentata da un oggetto di tipo List, che è anche di tipo Iterable. Tale interfaccia rappresenta un generico oggetto che può essere oggetto dell'istruzione "for-each", impiegata da processElements() per scorrere la raccolta elemento per elemento.

```
urce,
sumer<Y> block) {
```

```
            if (tester.test(p)) {
                Y data = mapper.apply(p);
                block.accept(data);
            }
        }
    }
```

```
do può essere invocato nel seguente modo:
essElements(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Funzioni Aggregate

- Nell'esempio del metodo processPersonsWithFunction() con l'uso dei generics più estensivo abbiamo visto la seguente implementazione:

```
public static <X, Y> void processElements( Iterable<X> source,
        Predicate<X> tester, Function <X, Y> mapper, Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

Ogni elemento della raccolta viene filtrato, e considerato solo se soddisfa una data condizione.

Questo me… …e invocato nel seguente modo:

```
processEle…
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Funzioni Aggregate

- Nell'esempio del metodo processPersonsWithFunction() con l'uso dei generics più estensivo abbiamo visto la seguente implementazione:

```
public static <X, Y> void processElements( Iterable<X> source,
        Predicate<X> tester, Function <X, Y> mapper, Consumer<Y> block) {
    for (X p : source) {
        block.accept(data);
    }
}
```

Questo metodo può essere invocato nel seguente modo:

```
.getGender() == Person.Sex.MALE
& p.getAge() >= 18
&& p.getAge() <= 25,
p -> p.getEmailAddress(),
email -> System.out.println(email)
);
```

Da ogni elemento filtrato viene estratto un dato, in questo campo l'indirizzo e-mail.

Funzioni Aggregate

- Nell'esempio del metodo processPersonsWithFunction() con l'uso dei generics più estensivo abbiamo visto la seguente implementazione:

```
public static <X, Y> void processElements( Iterable<X> source,
        Predicate<X> tester, Function <X, Y> mapper, Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

Al dato estratto, viene applicata un'azione, specificata dal metodo accept di Consumer<T>, in questo caso la stampa a video.

re invocato nel seguente modo:

```
== Person.Sex.MALE
p.getAge() >= 18
& p.getAge() <= 25,
> p.getEmailAddress(),
email -> System.out.println(email)
);
```

Funzioni Aggregate

- Nell'eser… generic… estensiv…

Posso rendere il codice ancora meno verbose e più comprensibile???.

```
public static <X, Y> void processElements( Iterable<X> source,
        Predicate<X> tester, Function <X, Y> mapper, Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

Questo metodo può essere invocato nel seguente modo:

```
processElements(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Funzioni Aggregate

- È possibile realizzare la stessa azione adottando una scrittura più compatta e leggibile, impiegando le operazioni aggregate:

```
roster.stream().filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18 && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```

Funzioni Aggregate

- È possibile realizzare la stessa azione adottando una scrittura più compatta e leggibile, impiegando le operazioni aggregate:

```
roster.stream().filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18 && p.getAge() <= 25)
```

Operazione aggregata – Stream<E> stream()
data una raccolta di oggetti restituisce un flusso di oggetti per eseguire operazioni di tipo funzionale.

Funzioni Aggregate

- È possibile realizzare la stessa azione adottando una scrittura più compatta e leggibile, impiegando le operazioni aggregate:

```
roster.stream().filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18 && p.getAge() <= 25)
```

Cosa è uno stream? Sussistono delle similarità con i flussi di IO?

Operazione aggregata – Stream<E> stream()
data una raccolta di oggetti restituisce un flusso di oggetti per eseguire operazioni di tipo funzionale.

Funzioni Aggregate

- È possibile realizzare la stessa azione adottando una scrittura più compatta e leggibile, impiegando le operazioni aggregate:

Cosa è uno stream? Sussistono delle similarità con i flussi di IO?

```
roster.stream().filter(
    p -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18 && p.getAge() <= 25)
```

Operazione aggregata – Stream<E> stream()
data una raccolta di oggetti restituisce un flusso di oggetti per eseguire operazioni di tipo funzionale.

Nessuna parentela con InputStream o OutputStream. Uno stream è un'interfaccia che restituisce un flusso di dati finito o infinito su cui è possibile fare operazioni di filtro, mappa e riduzione. Non è una raccolta di oggetti, ma un modo per manipolare i dati in maniera dichiarativa e compositiva.

Funzioni Aggregate

- È possibile realizzare la stessa azione adottando una scrittura più compatta e leggibile, impiegando le operazioni aggregate:

```
roster.stream().filter(
        p -> p.getG    er() == Person.Sex.MALE
            &&     >= 18 && p.getAge() <= 25)
```

Operazione aggregata – Stream<T> filter(Predicate<? super T> predicate) restituisce un flusso filtrato con oggetti dato un predicato.

Funzioni Aggregate

- È possibile realizzare la stessa azione adottando una scrittura più compatta e leggibile, impiegando le operazioni aggregate:

```
roster.stream().filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18 && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```

Operazione aggregata – <R> Stream<R> map(Function<? super T, ? extends R> mapper)
mappa gli oggetti di un filtro in altri valori restituiti come filtri, secondo quanto indicato nell'oggetto Function<K,V>.

Funzioni Aggregate

- È possibile realizzare la stessa azione adottando una scrittura più compatta e leggibile, impiegando le operazioni aggregate:

```
roster.stream().filter(
    .map(p                    mailAddress())
    .forEach(email -> System.out.println(email));
```

Operazione aggregata – void forEach(Consumer<? super T> action) esegue un'azione, come indicato nell'oggetto Consumer<T> su ogni oggetto del flusso su cui è applicato.

Funzioni Aggregate

- È possibile realizzare la stessa azione adottando una scrittura più compatta e leggibile, impiegando le operazioni aggregate:

```
roster.stream().filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18 && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```

Una serie di operazioni aggregate prende il nome di **pipeline**.

Esempio

- Il vantaggio della soluzione con operatori aggregati è di evitare l'effetto a farfalla. Nel caso in cui il problema cambi anche di poco, senza gli stream c'è il rischio di modificare molte righe, invece con gli stream c'è meno codice da riscrivere.
- Consideriamo il codice che, data una lista di album, se l'album è pubblicato prima del 2000, aggiunge tutte le canzoni alla lista. Ordina la lista e poi stampa i primi 10 elementi. Ecco il codice senza flussi e operazioni aggregate:

```
int limit = 10; List<String> songs = new ArrayList<>();
for (Album album : albums) {
  if (album.getYear() < 2000)
    songs.addAll(album.getSongs());
}
Collections.sort(songs);
for (int i = 0; i < limit; i++)
      System.out.println(songs.get(i));
```

Esempio

- Il vantaggio della soluzione con operatori aggregati è di evitare l'effetto a farfalla. Nel caso in cui il problema cambi anche di poco, senza gli stream c'è il rischio di modificare molte righe, invece con gli stream c'è meno codice da riscrivere.
- Consideriamo il codice che, data una lista di album, se l'album è pubblicato prima del 2000, aggiunge tutte le canzoni alla lista. Ordina la lista e poi stampa i primi 10 elementi. Ecco il codice senza flussi e operazioni aggregate:

```
int limit = 10; List<String> songs = new ArrayList<>();
for (Album album : albums) {
  if (album.getYear() < 2000)
    songs.addAll(album.getSongs());
}
Collections.sort(songs);
for (int i = 0; i < limit; i++)
    System.out.println(songs.get(i));
```

Ecco il codice equivalente con flussi e operatori aggregati:

```
albums.stream()
  .filter(album -> album.getYear() < 2000)
  .flatMap(album -> album.getSongs().stream())
  .sorted()
  .limit(10)
  .forEach(System.out::println);
```

Esempio

- Il vantaggio della soluzione con operatori aggregati è di evitare l'effetto a farfalla. Nel caso in cui il problema cambi anche di poco, senza gli stream c'è il rischio di modificare molte righe, invece con gli stream c'è meno codice da riscrivere.
- Consideriamo il codice che, data una lista di album, se l'album è pubblicato prima del 2000, aggiunge tutte le canzoni alla lista. Ordina la lista e poi stampa i primi 10 elementi. Ecco il codice senza flussi e operazioni aggregate:

```
int limit = 10; List<String> songs = new Arra
for (Album album : albums) {
  if (album.getYear() < 2000)
    songs.addAll(album.getSongs());
}
Collections.sort(songs);
for (int i = 0; i < limit; i++)
    System.out.println(songs.get(i));
```

Prende uno elemento dell stream come argomento e restituisce uno strem

Ecco il codice con flussi e operatori aggregati:

```
albu  eam()
  .filt (album -> album.getYear() < 2000)
  .flatMap(album -> album.getSongs().stream())
  .sorted()
  .limit(10)
  .forEach(System.out::println);
```

Esempio

- Il vantaggio della soluzione con operatori aggregati è di evitare l'effetto a farfalla. Nel caso in cui il problema cambi anche di poco, senza gli stream c'è il rischio di modificare molte righe, invece con gli stream c'è meno codice da riscrivere.
- Consideriamo il codice che, data una lista di album, se l'album è pubblicato prima del 2000, aggiunge tutte le canzoni alla lista. Ordina la lista e poi stampa i primi 10 elementi. Ecco il codice senza flussi e operazioni aggregate:

```
int limit = 10; List<String> songs = new ArrayList<>();
for (Album album : albums) {
  if (album.getYear() < 2000)
    songs.addAll(album.getSongs());
}
Collections.sort(songs);
for (int i = 0; i < limit; i++)
    System.out.println(songs.get(i));
```

Ecco il co

```
.flatM...album -> album.getSongs().stream())
.sorted()
.limit(10)
.forEach(System.out::println);
```

Troncamento dello stream fino a 10 elementi, gli altri sono scartati. Complementare è skip(n) che elimina i primi n elementi.

Complichiamo l'esempio

```
int limit = 10;
List<String> songs = new ArrayList<>();

for (Album album : albums) {
  if (album.getYear() < 2000)
    songs.addAll(album.getSongs());
}

Collections.sort(songs);

for (int i = 0; i < limit; i++)
  System.out.println(songs.get(i));
```

```
albums.stream()
   .filter(album -> album.getYear() < 2000)
   .flatMap(album -> album.getSongs().stream())
   .sorted()
   .limit(10)
   .forEach(System.out::println);
```

Codice con flussi e operazioni aggregate

Ipotizziamo di voler modificare leggermente il codice, che deve restituire le prime 10 canzoni che iniziano con la lettera "D", in ordine alfabetico, pubblicate prima del 2000 ma senza ripetizioni.

Complichiamo l'esempio

```
 int limit = 10;
Set<String> songs = new TreeSet<>();
for (Album album : albums) {
  if (album.getYear() < 2000) {
    for (String song : album.getSongs()) {
      if (song.startsWith("D"))
        songs.add(song);
    }
  }
}
Collections.sort(songs);

int temp = 0;
for (String s : songs) {
  System.out.println(s);
  temp++;
  if (temp == limit)
    break;
}
```

```
albums.stream()
    .filter(album -> album.getYear() < 2000)
    .flatMap(album ->
      album.getSongs().stream().filter(
          s -> s.startsWith("D")))
    .distinct()
    .sorted()
    .limit(10)
    .forEach(System.out::println);
```

Codice con flussi e operazioni aggregate

Compl[illegible]mpio

Metodo della classe String

```
int limit = 10;
Set<String> songs = new TreeSet<>
for (Album album : albums) {
  if (album.getYear() < 2000) {
    for (String song : album.getSongs()) {
      if (song.startsWith("D"))
        songs.add(song);
    }
  }
}
Collections.sort(songs);

int temp = 0;
for (String s : songs) {
  System.out.println(s);
  temp++;
  if (temp == limit)
    break;
}
```

```
albums.stream()
    .filter(album -> album.getYear() < 2000)
    .flatMap(album ->
      album.getSongs().stream().filter(
          s -> s.startsWith("D")))
    .distinct()
    .sorted()
    .limit(10)
    .forEach(System.out::println);
```

Codice con flussi e operazioni aggregate

Complichiamo l'esempio

```
 int limit = 10;
Set<String> songs = new TreeSet<>();
for (Album album : albums) {
  if (album.getYear() < 2000) {
    for (String song : album.getSongs()) {
      if (song.startsWith("D"))
        songs.add(song);
    }
  }
}


int temp = 0;
for (String s : songs) {
  System.out.println(s);
  temp++;
  if (temp == limit)
    break;
}
```

```
albums.stream()
   .filter(album -> album.getYear() < 2000)
   .flatMap(album ->
     album.getSongs().stream().filter(
        s -> s.startsWith("D")))
   .distinct()
   .sorted()
   .limit(10)
   .forEach(System.out::println);
```

Codice con flussi e operazioni aggregate

Vediamo le righe modificate.

Complichiamo l'esempio

```
int limit = 10;
Set<String> songs = new TreeSet<>();
for (Album album : albums) {
  if (album.getYear() < 2000) {
    for (String song : album.getSongs()) {
      if (song.startsWith("D"))
        songs.add(song);
    }
  }
}
```

```
int temp = 0;
for (String s : songs) {
  System.out.println(s);
  temp++;
  if (temp == limit)
    break;
}
```

```
albums.stream()
    .filter(album -> album.getYear() < 2000)
    .flatMap(album ->
      album.getSongs().stream().filter(
          s -> s.startsWith("D")))
    .distinct()
    .sorted()
    .limit(10)
    .forEach(System.out::println);
```

Codice con flussi e operazioni aggregate

Vediamo le righe modificate.

Complichiamo l'esempio

```
 int limit = 10;
Set<String> songs = new TreeSet<>();
for (Album album : albums) {
  if (album.getYear() < 2000) {
    for (String song : album.getSongs()) {
      if (song.startsWith("D"))
        songs.add(song);
    }
  }
}
```

```
int temp = 0;
for (String s : songs) {
  System.out.println(s);
  temp++;
  if (temp == limit)
    break;
}
```

```
albums.stream()
   .filter(album -> album.getYear() < 2000)
   .flatMap(album ->
     album.getSongs().stream().filter(
        s -> s.startsWith("D")))
   .distinct()
   .sorted()
   .limit(10)
   .forEach(System.out::println);
```

Codice con flussi e operazioni aggregate

Vediamo le righe modificate.

Esempio

- Consideriamo il caso di dover implementare una funzione che stampa i membri maschili contenuti nella raccolta di tipo List<Person>.
  - **Soluzione con ciclo for-each**

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());    }
}
```

  - **Soluzione con operazioni aggregate**

```
roster.stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName()));
```

Esempio

- Consideriamo il caso di dover implementare una funzione che stampa i membri maschili contenuti nella raccolta di tipo List<Person>.
  - **Soluzione con ciclo for-each**

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());    }
}
```

Il codice indica allo stream cosa fare, non come farlo, di conseguenza si ha qualcosa più simile al problema di partenza, piuttosto che a una procedura per risolverlo. È più semplice da leggere perché non ci sono variabili intermedie, nè cicli.

  - **Soluzione con operazioni aggregate**

```
roster.stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName()));
```

Pipeline e Stream

- Consideriamo il caso di dover implementare una funzione che stampa i membri maschili contenuti nella raccolta di tipo List<Person>.
  - **Soluzione con ciclo for-each**

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());    }
}
```

Possiamo distinguere le diverse entità che compongono una pipeline.

  - **Soluzione con operazioni aggregate**

```
roster.stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName()));
```

Pipeline e Stream

- Consideriamo il caso di dover implementare una funzione che stampa i membri maschili contenuti nella raccolta di tipo List<Person>.
  - **Soluzione con ciclo for-each**

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());    }
}
```

  - **Soluzione c**

Può essere una raccolta, un array, una funzione generatore o un canale I/O.

```
roster.stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName()));
```

Pipeline e Stream

- Consideriamo il caso di dover implementare una funzione che stampa i membri maschili contenuti nella raccolta di tipo List<Person>.
  - **Soluzione con ciclo for-each**

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());    }
}
```

2. **zero o più funzioni intermedie**

  - **Soluzion**

Sono operazioni che a partire da una sorgente o un flusso originano un nuovo flusso.

```
roster.stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName()));
```

Pipeline e Stream

- Consideriamo il caso di dover implementare una funzione che stampa i membri maschili contenuti nella raccolta di tipo List<Person>.
  - **Soluzione con ciclo for-each**

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());    }
}
```

  - **Soluzione con operazioni aggregate**

```
roster.
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName()));
```

Sono operazioni che producono in uscita un valore non di flusso, come un tipo primitivo, una raccolta, o void.

Esempio

- Consideriamo un altro esempio di un metodo che calcola l'età media dei membri maschili in una raccolta:

```
double average = roster.stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

Esempio

- Consideriamo un altro esempio di un metodo che calcola l'età media dei membri maschili in una raccolta:

```
double average = roster.stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

Operazione intermedia che estrae un flusso di interi dagli oggetti di flusso applicando un'apposita funzione, passata come riferimento a metodo della classe Person. In alternativa l'espressione lambda sarebbe: p -> p.getAge().

Esempio

- Consideriamo un altro esempio di un metodo che calcola l'età media dei membri maschili in una raccolta:

```
double average = roster.stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

Questa operazione terminale calcola la media in un flusso di numeri di tipo IntStream e restituisce un oggetto di tipo OptionalDouble. Se il flusso non ha elementi, si ha un'istanza vuota di OptionalDouble.

Esempio

- Consideriamo un altro esempio di un metodo che calcola l'età media dei membri maschili in una raccolta:

```
double average = roster.stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

Questa operazione terminale opera il cast di un oggetto di tipo OptionalDouble in uno Double. Se si ha un'istanza vuota di OptionalDouble, viene sollevata un'eccezione di tipo NoSuchElementException.

Esempio

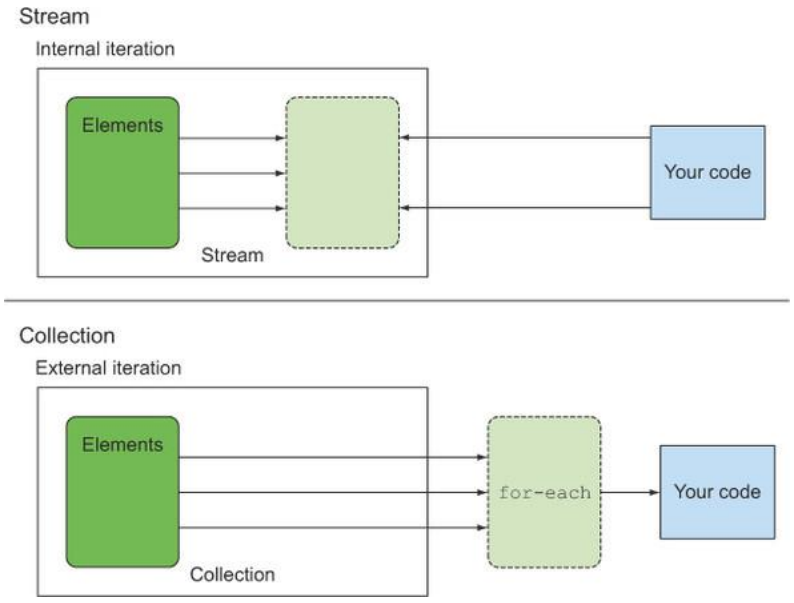
- Consideriamo un altro esempio di un metodo che calcola l'età media dei membri maschili in una raccolta:

```
double average = roster.stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

- La JDK contiene molte operazioni terminali (come average(), sum(), min() o max()) che restituiscono un valore a fronte di un'operazione sul contenuto di un flusso, oppure terminali che restituiscono una collezione o void. Tali operazioni che realizzano un compito ben preciso spesso prendono il nome operazioni di riduzione.

*ForEach vs Interazione Interna

- Operazioni aggregate, come **forEach**, sembrano simili agli iteratori, ma in realtà presentano varie differenze:
- Usano iterazioni interne – le operazioni aggregate non contengono un metodo tipo next() per istruirle su come elaborare il prossimo elemento della raccolta.



- Con **l'iterazione interna**, l'applicazione determina quale collezione bisogna iterare, ma la JDK determina come bisogna iterarla.
- Con **l'iterazione esterna**, l'applicazione determina sia quale collezione iterare e sia come l'iterazione deve avvenire.

*ForEach vs Interazione Interna

- D'altra parte, l'iterazione esterna consente solo un accesso sequenziale agli elementi della raccolta, mentre quella interna non presenta questa limitazione. Può sfruttare i vantaggi dell'elaborazione parallela, che consiste nel dividere il problema in sotto-problemi da risolvere parallelamente, e poi si combinano i risultati dei sotto-problemi.

- Operazioni aggregate elaborano gli elementi presi da un flusso – Tali operazioni operano sugli elementi di un flusso, non direttamente su una raccolta.
- Supportano dei pattern di comportamento come parametri – è possibile indicare espressioni lambda come parametri per la maggior parte delle operazioni, così da particolareggiarne il comportamento.

Pipeline e Stream

- Uno **stream** non contiene dati a parte una manciata di flag che consentono di controllare e ottimizzare il flusso. Ha bisogno, quindi, di una sorgente da cui pescare informazioni.

- L'esempio più comune consiste nel partire da una collection, ma uno stream può essere generato anche in altri modi come riportato qui sotto.

```
IntStream intStream = IntStream.range(1, 10);
DoubleStream doubleStream = new Random().doubles();
Stream<Path> pathStream = Files.list(Paths.get("c:\\"));
Stream<T> stringStream = Stream<T>.of(X extends T...vars)
```

Pipeline e Stream

- Uno **stream** non contiene dati a parte una manciata di flag che consentono di controllare e ottimizzare il flusso. Ha bisogno, quindi, di una sorgente da cui pescare informazioni.
- L'esempio … no stream può esser …

Si costruisce un IntStream a partire da un intervallo di interi. IntStream non è altro che una specializzazione di Stream specifica per gli interi; allo stesso modo Java 8 fornisce DoubleStream e LongStream.

```
IntStream intStream = IntStream.range(1, 10);
DoubleStream doubleStream = new Random().doubles();
Stream<Path> pathStream = Files.list(Paths.get("c:\\"));
Stream<T> stringStream = Stream<T>.of(X extends T...vars)
```

Pipeline e Stream

- Uno **stream** non contiene dati a parte una manciata di flag che consentono di controllare e ottimizzare il flusso. Ha bisogno, quindi, di una sorgente da cui pescare informazioni.

- L'esempio più comune consiste nel partire da una collection, ma uno stream può essere generato anche in altri modi come riportato qui sotto.

Abbiamo uno stream infinito di double pseudocasuali compresi tra 0 e 1.

```
IntStre
DoubleStream doubleStream = new Random().doubles();
Stream<Path> pathStream = Files.list(Paths.get("c:\\"));
Stream<T> stringStream = Stream<T>.of(X extends T...vars)
```

Pipeline e Stream

- Uno **stream** non contiene dati a parte una manciata di flag che consentono di controllare e ottimizzare il flusso. Ha bisogno, quindi, di una sorgente da cui pescare informazioni.

- L'esempio più comune consiste nel partire da una collection, ma uno stream può essere generato anche in altri modi come riportato qui sotto.

IntStr

Doubl

A partire da un percorso su file system (stiamo usando java.nio), restituiamo uno stream di Path.

```
Stream<Path> pathStream = Files.list(Paths.get("c:\\"));
```

```
Stream<T> stringStream = Stream<T>.of(X extends T...vars)
```

Pipeline e Stream

- Uno **stream** non contiene dati a parte una manciata di flag che consentono di controllare e ottimizzare il flusso. Ha bisogno, quindi, di una sorgente da cui pescare informazioni.

- L'esempio più comune consiste nel partire da una collection, ma uno stream può essere generato anche in altri modi come riportato qui sotto.

```
IntStream intStream = IntStream.range(1, 10);
Double
Stream
Stream<T> stringStream = Stream<T>.of(X extends T...vars)
```

Costruisce uno stream con i valori contenuti nel metodo of. Novità di Java 9!!!

*Pipeline e Stream Collection Immutabili

- Le Collections sono strumenti molto utili per i programmatori perchè permettono di raggruppare oggetti in maniera opportuna in modo tale da poter essere poi agevolmente manipolati.
- Fino alla versione 9 di Java non esisteva un metodo semplice per creare una Collection con dati predefiniti o un modo rendere la collezione immutabile (cioè non è possibile aggiungere, togliere o modificare gli elementi) senza scrivere ancora altro codice.

```
List<Point> myList = new ArrayList<>();
myList.add(new Point(1, 1));
myList.add(new Point(2, 2));
myList.add(new Point(3, 3));
myList.add(new Point(4, 4));
myList = Collections.unmodifiableList(myList);
```

*Pipeline e Stream Collection Immutabili

- Le Collections sono strumenti molto utili per i programmatori perchè permettono di raggruppare oggetti in maniera opportuna in modo tale da poter essere poi agevolmente manipolati.
- Fino alla versione 9 di Java non esisteva un metodo semplice per creare una Collection con dati predefiniti o un modo rendere la collezione immutabile (cioè non è possibile aggiungere, togliere o modificare gli elementi) senza scrivere ancora altro codice.

```
List<Point> list = List.of(new Point(1, 1), new Point(2, 2),
new Point(3, 3), new Point(4, 4));
```

Questi metodi generano collections immutabili. Se si prova ad aggiungere, eliminare o aggiornarne gli elementi, viene generata un'eccezione UnsupportedOperationException. Non consentono di inserire valori nulli, altrimenti viene generata un'eccezione NullPointerException.

*Stream.empty

- È possibile creare uno stream senza elementi con il metodo empty():

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

- In Java 9, un nuovo metodo è stato aggiunto per creare uno stream da un oggetto che può essere nullo:

  String homeValue = System.getProperty("home");

  Stream<String> homeValueStream =

  homeVale == null ? Stream.empty() : Stream.of(value);

*Pipeline e Stream

- È possibile creare uno stream senza elementi con il metodo empty():

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

- In Java 9, un nuovo metodo è stato aggiunto per creare uno stream da un oggetto che può essere nullo:

```
String homeValue = System.getProperty("home");
Stream<String> homeValueStream =
        homeVale == null ? Stream.empty() : Stream.of(value);

Stream<String> homeValueStream = Stream.ofNullable(
                                    System.getProperty("home"));
```

Pipeline e Stream - Libreria NIO

- La libreria NIO è stata aggiornata per sfruttare lo Stream API, e molti dei suoi metodi ritornano uno stream.
  - Un metodo utile è **Files.lines**, che ritorna uno stream di linee come stringhe lette da un file.

```
long uniqueWords = 0;
try(Stream<String> lines = Files.lines(Paths.get("data.txt"),
        Chrset.defaultCharset())){
    uniqueWords = lines.flatMap(line -> Arrays.stream(line.split(" ")))
        .distint().count();
} catch(IOException e) {…}
```

Pipeline e Stream - Libreria NIO

- La libreria NIO è stata aggiornata per sfruttare lo Stream API, e molti dei suoi metodi ritornano uno stream.
  - Un metodo utile è **Files.lines**, che ritorna uno stream di linee come stringhe lette da un file.

```
long uniqueWords = 0;
try(Stream<String> lines = Files.lines(Paths.get("data.txt"),
          Chrset.defaultCharset())){
    uniqueWords = lines.flatM      ine -> Arrays.stream(line.split(" ")))
          .distint().count();
} catch(IOExceptic
```

Gli stream sono autocloseable e non c'è bisogno del blocco finally.

Pipeline e Stream

- Lo Stream API dispone di due metodi statici per la creazione di stream a partire da una funzione e che sono caratterizzati da un insieme infinito di elementi.
  - Il metodo iterate() assume un valore iniziale e una espressione lambda per generare i valori successivi a partire dai precedenti.

```
Stream.iterate(0, n -> n+2).limit(10).forEach(System.out::println);
```

In Java 9, questo metodo è stato esteso con il supporto di un predicato per determinare quando continuare la generazione.

```
IntStream.iterate(0, n -> n < 100, n -> n + 4)
        .forEach(System.out::println);
```

Pipeline e Stream

- Il metodo generate() produce uno stream infinito come iterate() ma non è eseguito in maniera sequenziale.

```
Stream.generate(Math::random).limit(5).forEach(System.out::println);
```

*Optional

- Optional è un tipo monadico a cui è possibile associare funzioni per trasformare il valore al suo interno. Ecco un semplice esempio: Immaginate di avere una chiamata ad una API che potrebbe restituire un valore o un null, tale valore deve essere poi elaborato mediante la chiamata al metodo transform(); Vediamo come trattare il caso con e senza l'utilizzo degli Optional:

Senza l'uso degli Optional :

```
User user = getUser(name); // può ritornare null
Location location = null;
if(user != null)
  location = getLocation(user);
```

Con l'uso degli Optional :

```
Optional<User> user = Optional.ofNullable(getUser(user));
Optional<Location> location = user.map(u -> getLocation(user));
```

- La soluzione con l'uso degli Optional risulta più elegante e ci permette di non "inquinare" il codice con i controlli dei valori null. L'utilizzo nel codice è sempre possibile in quanto gli Optional posso essere usati con qualsiasi funzione.

*Optional

- È bene tenere presente alcune regole empiriche per il loro utilizzo:
  - I campi di istanza delle classi – usare valori semplici. Gli Optional non sono stati creati per l'utilizzo nei campi. Non sono serializzabili e aggiungono un sovraccarico per il wrapping che non è necessario. Vanno usati invece nei metodi quando si elaborano i dati dei campi.
  - Parametri dei metodi – usare anche qui valori semplici. Usare gli Optional nei parametri dei metodi "inquina" le firme dei metodi e rende il codice più difficile da leggere e mantenere.
  - Valori di ritorno dei metodi – va considerato l'utilizzo degli Optional. Invece di restituire valori nulli, il tipo Optional potrebbe essere migliore. Chiunque utilizza il codice sarà costretto a gestire i casi null e con l'uso degli Optional il codice risulta più pulito.

Riduzioni

- In aggiunta ad un insieme di operazioni di riduzione che restituiscono un valore o una raccolta di valori dato un flusso, il cui comportamento è già implementato, la JDK offre due operazioni che hanno un comportamento general-purpose:
  - Stream.reduce();
  - Stream.collect().

Stream.reduce()

- Il metodo reduce() di Stream rappresenta un'operazione di riduzione general-purpose che restituisce un valore ed è caratterizzata da due elementi di ingresso:
  - identity –rappresenta sia l'elemento di partenza che il risultato di default se non ci sono elementi nel flusso;
  - accumulator – funzione con due parametri, il primo è il risultato parziale della riduzione in iterazioni precedenti, mentre il secondo è l'elemento corrente del flusso e restituisce un nuovo valore parziale della riduzione.

Stream.reduce()

- Consideriamo la seguente pipeline che calcola la somma delle età dei membri maschi nella collezione roster, usando l'operazione sum() di riduzione:

```
Integer totalAge = roster.stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge).sum();
```

Stream.reduce()

- Consideriamo la seguente pipeline che calcola la somma delle età dei membri maschi nella collezione roster, usando l'operazione sum() di riduzione:

```
Integer totalAge = roster.stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge).sum();
```

- Confrontiamo questa pipeline con una che impiega l'operazione Stream.reduce() per ottenere lo stesso risultato:

```
Integer totalAgeReduce = roster.stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .map(Person::getAge)
    .reduce(0, (a, b) -> a + b);
```

Stream.reduce()

- Consideriamo la seguente pipeline che calcola la somma delle età dei membri maschi nella collezione roster, usando l'operazione sum() di riduzione:

```
Integer totalAge = roster.stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge).sum();
```

- Confrontiamo questa pipeline con una che impiega l'operazione Stream.reduce() per ottenere lo stesso risultato:

```
Integer totalAgeReduce = roster.stream()
    .filter(e -> e.g
    .map(Person
    .reduce(0, (a, b) -> a + b);
```

L'identity è pari a 0, ovvero il valore di partenza della somma, ma anche il valore restituito se il flusso è nullo.

Stream.reduce()

- Consideriamo la seguente pipeline che calcola la somma delle età dei membri maschi nella collezione roster, usando l'operazione sum() di riduzione:

```
Integer totalAge = roster.stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge).sum();
```

- Confrontiamo questa pipeline con una che impiega l'operazione Stream.reduce() per ottenere lo stesso risultato:

```
Integer totalAg
    .filter(e -> e.g
    .map(Person
    .reduce(0, (a, b) -> a + b);
```

La funzione accumulator è un'espressione lambda che somma due interi in oggetto Integer e restituisce un valore intero.come oggetto di Integer.

Stream.collect()

- L'operazione collect() è stata pensata per le raccolte, e non è disponibile solo una versione di questa operazione. Consideriamo una pipeline per racchiudere in una lista i nomi di membri maschi:

```
List<String> namesOfMaleMembersCollect = roster
    .stream()
    .filter(p ->
    .map(p ->
    .collect(Collectors.toList());
```

Si ha un solo argomento di tipo Collector, che incapsula le funzioni usate come argomenti in collect(). La classe Collectors racchiude delle utili operazioni di riduzione restituite come istanze di Collector.

Stream.collect()

- L'operazione collect() è stata pensata per le raccolte, e non è disponibile solo una versione di questa operazione. Consideriamo una pipeline per racchiudere in una lista i nomi di membri maschi:

```
List<String> namesOfMaleMembersCollect = roster
    .stream()
    .filter(p ->
    .map(p ->
    .collect(Collectors.toList());
```

Si ha un solo argomento di tipo Collector, che incapsula le funzioni usate come argomenti in collect(). La classe Collectors racchiude delle utili operazioni di riduzione restituite come istanze di Collector.

Collectors.toList() è un'operazione che accumula gli elementi del flusso in una nuova istanza di List.

Stream.collect()

- Un'altra utile operazione offerta da Collections è **groupingBy**, che restituisce una mappa le cui chiavi sono i valori dell'espressione lambda passata come argomento (detta funzione di classificazione). Vediamo un esempio di classificazione degli elementi di roster per genere:

```
Map<Person.Sex, List<Person>> byGender =
    roster.stream()
        .collect(Collectors.groupingBy(Person::getGender));
```

Stream.collect()

- Un'altra utile operazione offerta da Collections è **groupingBy**, che restituisce una mappa le cui chiavi sono i valori dell'espressione lambda passata come argomento (detta funzione di classificazione). Vediamo un esempio di classificazione degli elementi di roster per genere:

```
Map<Person.Sex, List<Person>> byGender =
    roster.stream()
        .collect(Collectors.groupingBy(Person::getGender));
```

In questo esempio la mappa restituita ha due chiavi: Person.Sex.MALE e Person.Sex.FEMALE, mentre i corrispondenti valori sono oggetti di tipo List che contengono gli elementi di flusso che, quando processati dalla funzione di classificazione, assumono il valore della chiave. Alla chiave Person.Sex.MALE corrisponde una lista di tutti i membri maschi.

Stream.collect()

- Consideriamo un altro esempio di una pipeline che recupera i nomi di ogni membro nella raccolta roster e li raggruppa per genere:

```
Map<Person.Sex, List<String>> namesByGender =
    roster.stream()
        .collect(
            Collectors.groupingBy(
                Person::getGender,  Collectors.mapping(
                    Person::getName, Collectors.toList())));
```

Stream.collect()

- Consideriamo un altro esempio di una pipeline che recupera i nomi di ogni membro nella raccolta roster e li raggrupp

```
Map<Person.Sex, List<String>> namesI
    roster.stream()
        .collect(
            Collectors.groupingBy(
                Person::getGender,  Collectors.mapping(
                    Person::getName, Collectors.toList())));
```

In questo caso l'operazione groupingBy prende due parametri: una funzione di classificazione e un'istanza di Collector, che è chiamato downstream collector.

*Stream.collect()

- Consideriamo un altro esempio di una pipeline che recupera i nomi di ogni membro nella raccolta roster e li raggruppa per genere:

```
Map<Person.Sex, List<String>> namesByGender =
    roster.stream()
        .collect(
            Collectors.groupingBy(
                Person::getGender,  Collectors.mapping(
                    Person::getName, Collectors.toList())));
```

- Il downstream collector viene applicato ai risultati di un altro collector. In questo uso, si ha che il metodo collect() viene applicato ai valori List creati dall'operazione groupingBy(). Al flusso originatosi da roster, viene prima applicata l'operazione di raggruppamento e poi l'estrazione dei nomi. Quando+line sono applicati vari downstream collector, otteniamo una riduzione multi-livello.

*Stream.collect()

- Partitioning è un caso speciale del raggruppamento, e il predicato usato restituisce un valore booleano che viene usato come chiave, mentre come valore troviamo la lista degli elementi del flusso che soddisfano o meno tale predicato.

```
Map<Boolean, List<Dish> > partitionedMenu = menu.stream()
                        .collect(partitiongBy(Dish::isVegetarian));
```

Tale istruzione ritorna una istanza di Map del tipo:

```
{false=[pork, beef, salmon],
 true=[french fries, rise, fruit] }
```

È possibile ottenere i piatti vegetariani come

```
List<Dish> vegetariaDishes = partitionedMenu.get(true);
```

*Stream.collect()

- Lo stesso risultato è ottenibile anche con filter:

```
List<Dish> vegetariaDishes = menu.stream()
                .filter(Dish::isVegetarian).collect(toList());
```

- Il vantaggio del partitioning però è di disporre di entrambe le liste e non solo di quella con gli elementi che soddisfano la condizione.

Slicing

- Java 9 ha aggiunto due nuovi metodi per selezionare elementi in uno stream. Consideriamo un esempio di una lista di piatti di cui si vogliono selezionare i piatti le cui calorie sono inferiori a 320.
- Esempio di utilizzo con filter():

```
List<Dish> filteredMenu = specialMenu.stream()
                              .filter(dish -> dish.getCalories() < 320)
                              .collect(toList());
```

Slicing

- Java 9 ha aggiunto due nuovi metodi per selezionare elementi in uno stream. Consideriamo un esempio di una lista di piatti di cui si vogliono selezionare i piatti le cui calorie sono inferiori a 320.
- Esempio di utilizzo con filter():

```
List<Dish> filteredMenu = specialMenu.stream()
                          .filter(dish -> dish.getCalories() < 320)
                          .collect(toList());
```

- Lo svantaggio di tale approccio è che si ha la necessità di iterare tutto lo stream, e computare il predicato per ogni elemento. Invece, sarebbe meglio fermarsi non appena trovato un elemento che soddisfa il predicato.

Slicing

- La funzione takeWhile aiuta a effettuare una slice di uno stream sulla base di un predicato.

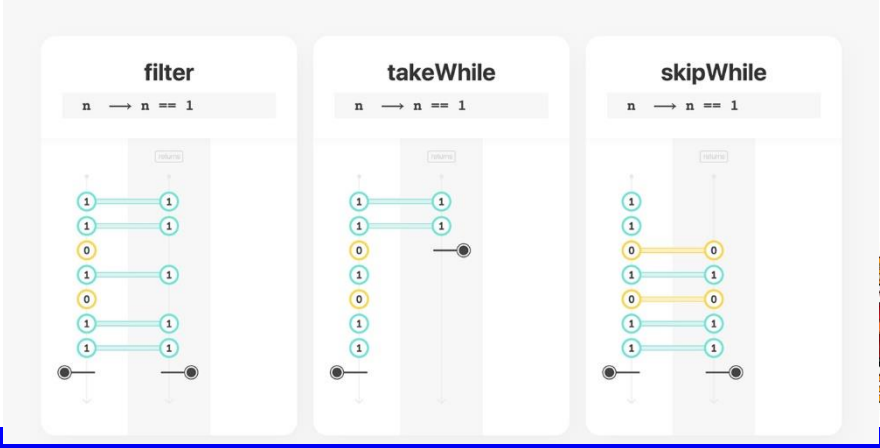
```
List<Dish> filteredMenu = specialMenu.stream()
                              .takeWhile(dish -> dish.getCalories() < 320)
                              .collect(toList());
```

Slicing

```
List<Dish> filteredMenu = specialMenu.stream()
                          .takeWhile(dish -> dish.getCalories() < 320)
                          .collect(toList());
```

- Se si vuole rimuovere tutti gli elementi fino a quando il predicato non è soddisfatto?

```
List<Dish> filteredMenu = specialMenu.stream()
                          .dropWhile(dish -> dish.getCalories() < 320)
                          .collect(toList());
```



Ricerca

- Un'altra comune operazione è la ricerca di alcuni elementi in un insieme, pertanto l'API Stream offre varie funzioni di ricerca e verifica su stream, che ritornano un valore booleano e che quindi sono funzioni terminali.
  - Il metodo **anyMatch** ritorna vero se almeno un elemento del flusso soddisfa il predicato.
  - Il metodo **allMatch** ritorna vero se tutti gli elementi di uno stream soddisfano un predicato.
  - Il metodo **noneMatch** è il negato del precedente e ritorna vero se nessun elemento dello stream soddisfa il predicato.
  - Il metodo **findFirst** restituisce il primo elemento dello stream;
  - Il metodo **findAny** ritorna un elemento arbitrario del flusso, come un Optional, e può essere usato in combinazione con altri metodi come filter().

```
Optional<Dish> dish = menu.stream().filter(Dish::isVegeterian)
                                   .findAny();
```

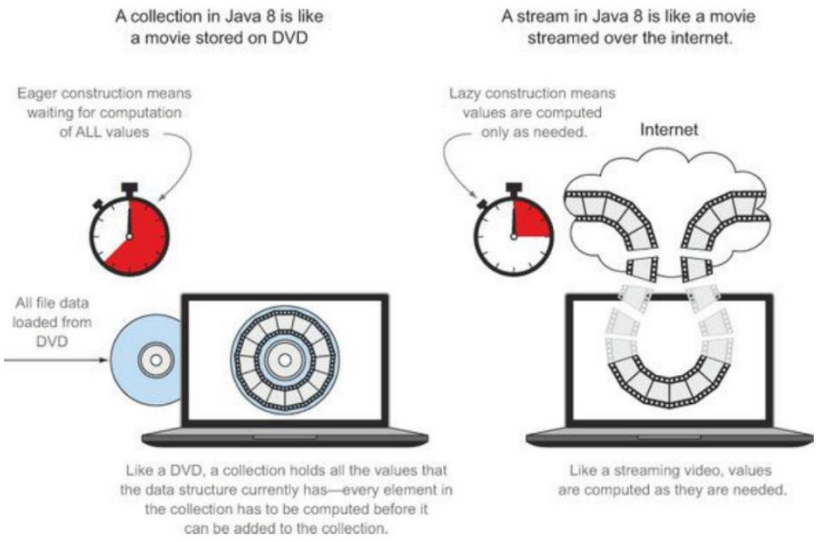
Ricerca

- La classe Optional<T> dispone di vari metodi che forzano il controllo esplicito della presenza di un valore o di gestire la situazione di una sua assenza:
  - **isPresent()** ritorna vero se è presente un valore;
  - **ifPresent(Consumer<T> block)** esegue la lambda espressione passata some input se un valore è presente;
  - **T get()** restituisce il valore se presente altrimenti solleva una eccezione di tipo NoSuchElementException;
  - **T orElse(T other)** restituisce il valore se presente oppure un valore di default passato come input.

```
Menu.stream().filter(Dish::isVegeterian).findAny().
        .ifPresent(dish -> System.out.println(dish.getName());
```

Stream vs Collection

- La differenza consiste nel quando gli elementi sono elaborati.
  - Una Collection è una struttura dati in memoria che mantiene tutti i suoi valori.
  - Uno Stream è una struttura dati concettualmente fissa (a cui non è possibile aggiungere/rimuovere elementi), i cui elementi sono computati su richiesta.
    - La valutazione eager consiste nel risolvere un'espressione non appena essa viene legata a una variabile.



- Tipicamente, il termine è usato in contrasto con la valutazione lazy, in cui un'espressione viene valutata solo quando si richiede il suo valore.
- Le collection sono costruite in maniera eager, lo stream in maniera lazy.
- Lo stream può essere attraversato una sola volta, e lo si dice consumato, una collection una multitudine di volte.