Corso di Programmazione Concorrente

Java 8 Parallel Stream

Valter Crescenzi
http://crescenzi.inf.uniroma3.it

Sommario

- POO vs PF vs PC
- Java 8
  - Lambdas
  - Functional Interface
    - `java.util.function.*`
  - Method Reference
  - Default Methods
- Stream
  - Pipelining, ed operazioni
    - Sorgente
    - Intermedie
    - Terminali
- Decomposizione Parallela di Stream
  - **`SplitIterator`**
  - **`ArrayListSpliterator`**
  - Legami con il Framework Fork/Join

POO vs PF vs PC

***OOP** makes code understandable by **encapsulating** moving parts;*
***FP** makes code understandable by **minimizing** moving parts.*

Michael Feathers

- La PC trova grande giovamento dall'assenza di scritture che potrebbero far violare le condizioni di Bernstein
  - Cfr. Tecnica degli *Oggetti Immutabili* >>
- I legami tra la PC e la PF sono stati rivitalizzati dalla diffusione di massa di architetture multi-core
- La PF favorisce la correttezza del codice concorrente, anche da parte di programmatori non esperti
- La PC ha quindi favorito il recente “riavvicinamento” della POO alla PF
  - Sono nati linguaggi a paradigma *ibrido* come Scala

Programmazione Funzionale

Vedi il corso di *Programmazione Funzionale*

- Programmi espressi come espressioni matematiche
- Effetti collaterali evitati e/o ben “confinati”
  - assenza di scritture: strutture dati *persistenti*
- *Referential Transparency*
  - un’espressione produce lo stesso risultato in ogni contesto
- Le funzioni possono ricevere come parametro, e/o restituire come risultato, altre funzioni

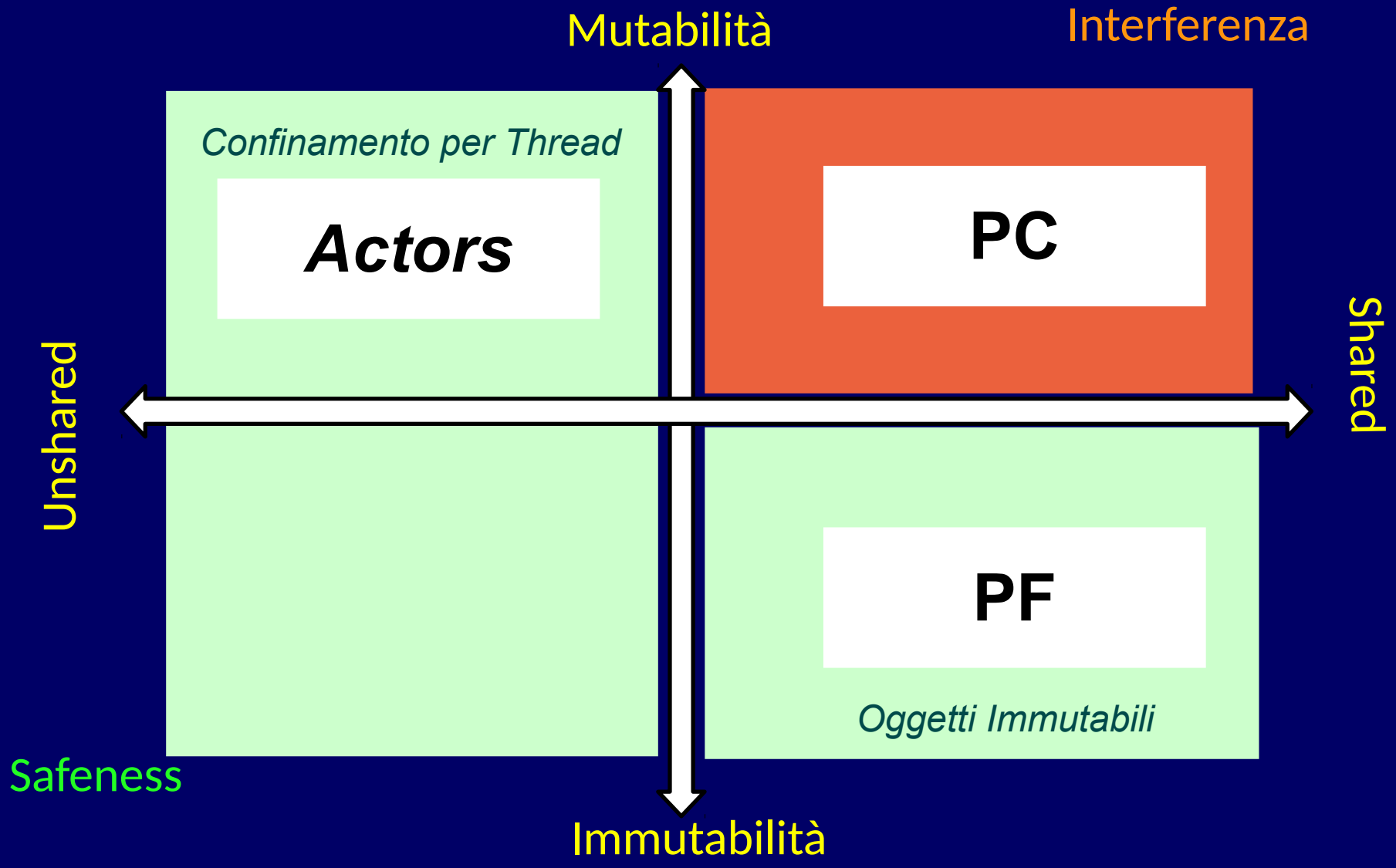
- Molte delle caratteristiche tipiche dei PF sono assolutamente desiderate e desiderabili nella POO
  - è preferibile l'utilizzo di oggetti immutabili, a prescindere da qualsiasi discorso legato alla PC
  - lo stato degli oggetti trattati, se non immutabile, deve seguire una dinamica semplice e facilmente tracciabile
    - la dinamica dello stato di un oggetto è difficile da “debuggare”

Immutabilità e PC

- La tecnica degli *Oggetti Immutabili* per la scrittura di codice thread-safe prevede:
  - di non effettuare mai scritture (tranne che durante la costruzione di oggetti, quando non sono ancora condivisi)
  - la dinamica dello stato di un programma evolve creando nuovi oggetti piuttosto che sovrascrindo lo stato degli oggetti già esistenti

- Strutture dati *persistenti/immutabili* possono essere condivise tra vari f.d.e. senza pericolo di interferenza, in quanto immutabili

La Programmazione e la Thread-Safeness



Java 8 & Lambda Calcolo

- 2006 – Gosling: "We will never have lambdas in Java”
- 2007 – 3 diversi progetti studiano lambda in Java
- 2008 – Reinhold: "We will never have lambdas in Java"
- 2009 – Inizio progetto Lambda (JSR 335)

…

- 2014 – Java 8 introduce Lambda

- Il supporto al lambda calcolo vera novità della release Java 8
- Probabilmente il cambiamento più significativo di sempre del linguaggio. Tardivo… ma ben motivato:
  - diffusione architetture hw multi-core
  - altri linguaggi avevano già “spianato” la strada (Scala, ecc.)

Verbosità Pre-Java 8

- Si definisce un criterio di ordinamento
  - evidente e frequente es. di verbosità del linguaggio

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}

Collections.sort(strings, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
});
```

- Utilizzando l'implementazione Java 8 del lambda calcolo:

```
Collections.sort(strings,(s1, s2) -> s1.compareToIgnoreCase(s2));
```

Lambda Calcolo in Java 8

- Principali difficoltà ingegneristiche legate all'ingombrante lascito delle versioni precedenti
  - *retrocompatibilità* a livello di file binari
  - estendere il linguaggio **senza** rivoluzionarlo

- **NON** è stato introdotto un *tipo funzione* (cfr. Scala)
- *Functional Interface:* semplice ma ingegnosa idea per non sconvolgere il sistema dei tipi Java
  - Associazione tra
    - lambda-function
    - SAM: interfacce a singolo metodo astratto

    creata all'atto dell'utilizzo di una *lambda-expression* e sulla base del *contesto* di utilizzo
- Si “prende in prestito” il tipo di una SAM piuttosto che definire esplicitamente il tipo di una lambda function

Sintassi Java 8 Lambda Function

- *Lambda-Expression*: definisce una lambda function
- come il corpo di un metodo *anonimo* java ma senza la necessità di dichiarare il tipo dei parametri. Due forme

  `(` *parametri* `) -> expression`

  `(` *parametri* `) -> { corpo }`
- Ad es.:
  - `(int x, int y) -> { return x + y; }`
  - `x -> x * x`
  - `( ) -> x`
- Semplici regole:
  - Il tipo dei parametri può essere inferito dal contesto di utilizzo
  - parentesi non nessarie per una lista di un singolo parametro
  - `()` per denotare zero parametri nella lista dei param.
  - Il corpo contiene zero o più istruzioni
  - `{}` non necessarie per corpi di una sola istruzione

Java Lambda Function: Implementazione

- Il compilatore Java 8 converte una lambda expression in un metodo statico
- Ad esempio, semplificando un pochino, a partire da:

```
x -> System.out.println(x)
```

si ottiene:

```
private static synthetic void lambda$0(Integer x) {
    System.out.println(x);
}
```

Ed in quale classe finisce...? ...di che tipo è?

Functional Interface

- Interfaccia SAM (annotata con **`@FunctionalInterface`**)
  - N.B. il nome del metodo non influenza l'associazione con la lambda function nonostante faccia parte della sua segnatura!
  - Meglio usare nomi esplicativi per i metodi, in ogni caso.

    Ad es.:

```
package java.util.function;
@FunctionalInterface
public interface Consumer<T> {
   void accept(T t);
}

@FunctionalInterface
public interface Predicate<T> {
   boolean test(T t);
}
```

Tipo di una Lambda Expression (1)

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

- Definiamo un generico metodo che la utilizzi come parametro:

```
static <T> void forEach(List<T> list, Consumer<T> v) {
    for (T element : list) {
        v.accept(element);
    }
}
```

Tipo di una Lambda Expression (2)

- Possiamo invocare **`forEach()`** utilizzando una lambda expression:

```
forEach( Arrays.asList(1,2,3),
          x -> System.out.println(x)
        );
```

- Oppure, passando per una variabile locale:

```
Consumer<Integer> visitor = x -> System.out.println(x);
forEach(Arrays.asList(1,2,3), visitor);// contesto utilizzo
```

- *target type* di una lambda expression
  - È il tipo “preso in prestito” dalla *functional interface*
  - E' definito dal *contesto di utilizzo* della lambda expression

Tipo di una Lambda Expression (3)

- Il *target type* di una lambda expression è il tipo della functional interface alla quale viene associato sulla base del suo contesto di utilizzo
- La lambda expression definisce il corpo di una implementazione sintetica e concreta dell'interfaccia che il compilatore genera ad *ogni* utilizzo
- Il metodo astratto viene sovrascritto affinché invochi il metodo statico sintetico contenente il corpo della lambda expression
- Conseguenze di questa tecnica
  - il codice che “riceve” una lambda function è costretto ad usare una interfaccia di nome esplicito
  - Sintassi già familiare ma verbosa
    - *inaspettatamente* potrebbe contrastare abusi del lambda calcolo!

Java.util.function.*
@FunctionalInterfaces

Predicate<T>	T → boolean
Consumer<T>	T → void
Supplier<T>	() → T
Function<T,U>	T → U
BiFunction<T,U,V>	(T,U) → V
UnaryOperator<T>	T → T
BinaryOperator<T,T>	(T,T) → T

- Package che fornisce semplici interfacce SAM per fissare un comune riferimento
  - attenzione alla semantica piuttosto *rilassata* dell'associazione tra functional interface e lambda function: il nome del metodo non conta!

Functional Interface di Vecchia Data

- **`Runnable`** e **`Callable<T>`** sono functional interface
  - l'annotazione `@FunctionalInterface` è stata aggiunta in Java 8
- Anche **`Comparable<T>`** e **`Comparator<T>`** ora sono functional interface
- E' perfettamente lecito scrivere:

```
ExecutorService exService = …;
Callable<Double> c = () -> Math.random();
exService.execute(c);
```

- Ma anche:

```
Callable<Double> c = () -> Math.random();
Supplier<Double> s = () -> Math.random();
```

sebbene la seguente assegnazione non compili:

```
c = s ; // NON COMPILA!
```

Metodi vs Funzioni

- I metodi di istanza possono essere facilmente interpretati come funzioni facenti perno su un “oggetto” noto
  - `A obj=…; B b=…;`
  - `C c = obj.m(B b);`
    - una classe di tipo A ospita un metodo `m()` che riceve un parametro di tipo B e restituisce un valore di tipo C
  - L'invocazione `obj.m(…)`, è interpretabile come:

    `b -> obj.m(b)`
  - Ovvero una funzione: B → C
- Esiste una sintassi dedicata alla conversione tra metodi e funzioni...doppi due punti `::`
  - Ragionamenti facilmente estendibili ai metodi statici, ai metodi generici, ai costruttori, specificando o meno l’istanza>>

Method Reference

Tipologia	***Sintassi***	***Esempio***	***Funzione***
Metodo statico	`ClassName::StaticMethodName`	`String::valueOf`	`Object → String`
Costruttore	`ClassName::new`	`ArrayList::new`	`() → ArrayList`
Istanza *specificata;* Metodo di istanza	`objectReference::MethodName`	`x::toString`	`() → String`
Istanza non *specificata;* Metodo di istanza	`ClassName::InstanceMethodName`	`Object::toString`	`Object → String`

- ✔ L'istanza su cui viene invocata una funzione può essere vista come uno dei parametri (il primo) della funzione stessa

Il Java Collection Framework

- Rivisitato. Un semplice ma significativo esempio:

```
Iterable<T>.forEach(Consumer<? super T> action)
```

Consumer: **T → void**

- ✔ Attenzione **`java.lang.Iterable`** esiste da Java 5, ma il metodo **`forEach`**() solo da Java 8
- Utilizzabile così:

```
List<Integer> list = Arrays.asList(1,2,3);
list.forEach(x -> System.out.println(x));
```

- Oppure, usando i *method reference* appena visti:

```
list.forEach(System.out::println);
```

Leggibilità, Riusabilità, Componibilità

- Esempi:

```
Comparator<Person> byAge = Comparators.comparing(p -> p.getAge());
Comparator<Person> byAge = Comparators.comparing(Person::getAge);
```

- `comparing()` è un metodo statico che fabbrica un `Comparator<T>` a partire da una funzione che fornisce una chiave di ordinamento:

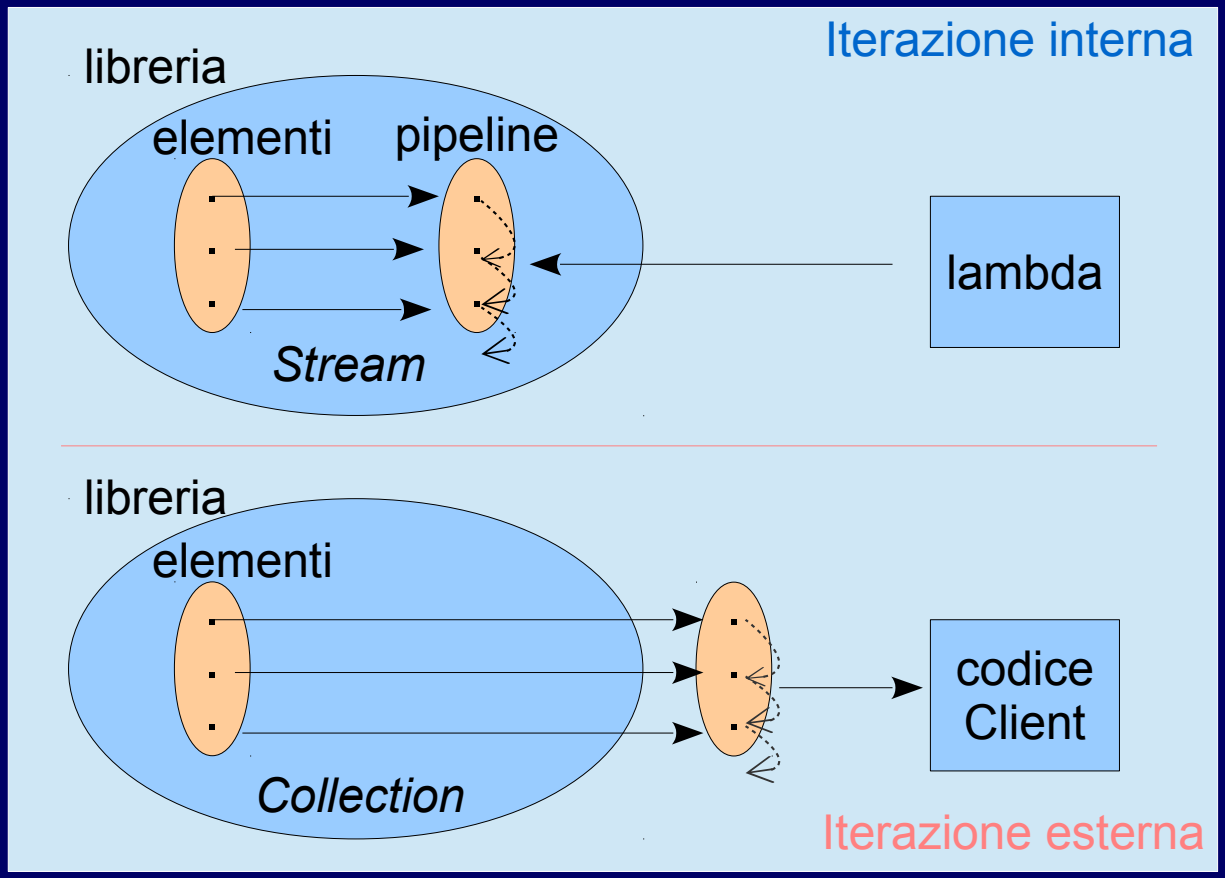
```
Collections.sort(people, comparing(Person::getAge));
```

- `thenComparing()` permette la costruzione di `Comparator<T>` per ordinamenti lessicografici:

```
Collections.sort(people,
        comparing(Person::getLastName).
        thenComparing(comparing(Person::getFirstName)));
```

Lambda Calcolo e Parallelismo

- Risulta agevolata la realizzazione di computazioni *lazy*
- Alcune librerie possono ora eseguire *internamente* processamenti specificati *esternamente* e passati al suo interno
  - In particolare: *iterazioni interne* vs *iterazioni esterne*



Stream

- Supporto al processamento *efficiente*, ed in *stile funzionale,* di sequenze di dati
- Efficiente, ovvero:
  - parallelo per sfruttare le architetture multi-core
  - senza materializzare risultati intermedi
- In stile funzionale, ovvero:
  - supportando la specifica (“monadica”), tipica della PF, di *pipeline* di processamento tramite composizione di operazioni più semplici

- Un modo comune, ma non il solo, di creare nuovi *stream* è partendo da una collezione data:

```
Stream<T> stream = collection.stream();
```

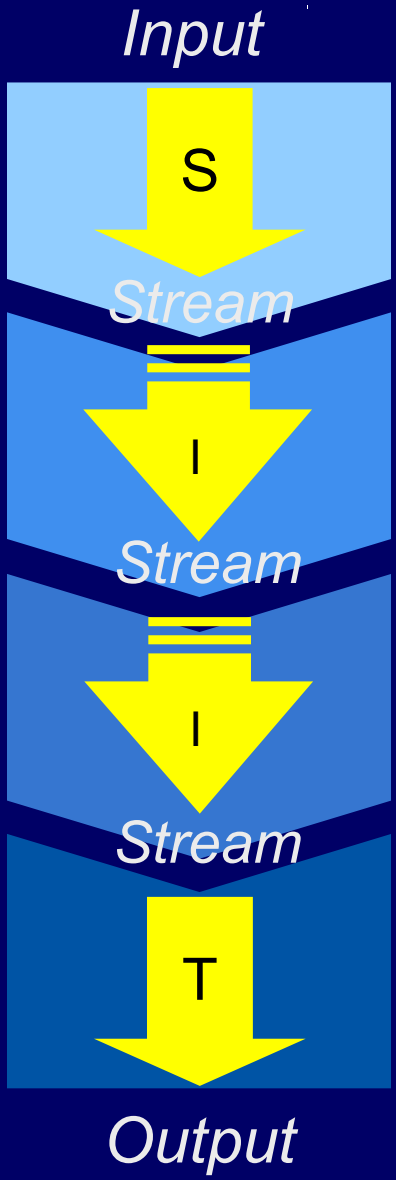
`java.util.stream.`*`Stream`*

- Interfaccia che rappresenta il flusso di dati su cui specificare la pipeline di processamento

- Caratteristiche del processamento
  - lazy>>
  - non materializzato
  - *a singola passata*
- Caratteristiche dello *stream*
  - mono-uso
  - potenzialmente illimitato
  - immutabile[/persistente]

Specifica di una *Stream Pipeline*

- Si ottiene facendo *pipelining* di operazioni di processamento
- L'input da cui generare un primo *Stream* sorgente viene trasformato nell'output, passando per vari *Stream* intermedi
- Componenti di tre tipologie:
  - *Sorgenti* (lazy)
  - *Intermedie* (lazy)
  - *Terminali* (eager)
- *Una sola* sorgente e *max un solo* terminale
- *0-N* operazioni intermedie



Componenti di una *Stream Pipeline*

- Tre tipologie di componenti:
  - *Sorgenti* – generatrice di *Stream*
    - a partire da una sorgente di elementi, come ad es.:
      - Una collezione
      - Una funzione generatrice
      - IO channel
  - *Intermedie* – operazioni di trasformazione
    - produce uno Stream a partire da uno Stream
  - *Terminali* – operazione di formazione dell'output
    - a partire dallo Stream produce qualcos'altro, ad es.:
      - un dato aggregato
      - una collezione (materializzata)

Stream: Esempio di Pipeline Completa

```
int sum = widgets.stream()
                 .filter(w -> w.getColor() == RED)
                 .mapToInt(w -> w.getWeight())
                 .sum();
```

In this example, `widgets` is a `Collection<Widget>`. We create a stream of `Widget` objects via `Collection.stream()`, filter it to produce a stream containing only the red widgets, and then transform it into a stream of `int` values representing the weight of each red widget. Then this stream is summed to produce a total weight.

In addition to `Stream`, which is a stream of object references, there are primitive specializations for `IntStream`, `LongStream`, and `DoubleStream`, all of which are referred to as "streams" and conform to the characteristics and restrictions described here.

Operazioni *Intermedie* e *Terminali*

Intermedie:

- `filter()`
- `map(), flatMap()`
- `limit(), skip()`
- `peek()`
- `sorted()`
- `distinct()`

Terminali:

- `forEach()`
- `min(), max(), count()`
- `reduce()`
- `toList(), toSet()`
- `findAny(), findFirst()`
- `anyMatch(), anyMatch(), noneMatch()`

`map()`: Un'Operazione Intermedia

```
List<Person> list = ... ;
Stream<Person> stream = list.stream() ;
Stream<String> names =
    stream.map(person -> person.getName()) ;
```

```
@FunctionalInterface
public interface Function<T, R> {
      R apply(T t);
}
```

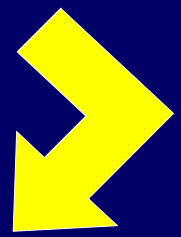
`reduce()`: Un'Operazione Terminale

```
List<String> strings = asList("PC, "POO", "PF");
int length = strings.stream()
                    .map(String::length)
                    .reduce(0, (a, b) -> a + b);
```

```
@FunctionalInterface
public interface BinaryOperator<T>
                    extends BiFunction<T, T, T> {
      T apply(T t, T u);
}
```

Esempio: “I Primi 40 ERROR”

```
List<String> errors = new ArrayList<>();
int errorCount = 0;
File file = new File(fileName);
String line = file.readLine();
while (errorCount < 40 && line != null) {
    if (line.startsWith("ERROR")) {
        errors.add(line);
        errorCount++;
    }
    line = file.readLine();
}
```



```
List<String> errors =
    Files.lines(Paths.get(fileName))
        .filter(l -> l.startsWith("ERROR")
        .limit(40)
        .collect(toList());
```

Raggruppamenti

```
Map<Dish.Type, List<Dish>> dishesByType = new HashMap<>();
for (Dish dish : menu) {
    Dish.Type type = dish.getType();
    List<Dish> dishes = dishesByType.get(type);
    if (dishes == null) {
        dishes = new ArrayList<>();
        dishesByType.put(type, dishes);
    }
    dishes.add(dish);
}
```


```
Map<Dish.Type, List<Dish>> dishesByType =
        menu.stream()
            .collect(groupingBy(Dish::getType));
```

Java 8 Default Methods

- Motivato dalla impellente necessità di mantenere la retrocompatibilità con il *Java Collections Framework*
- Come modificare le vecchie interfacce senza compromettere la base di codice esistente che già vi si basava?
- Si consideri ad es.:
  - un sottotipo concreto di *Collection* creato prima di Java 8
  - se Collection in Java 8 venisse estesa con un nuovo metodo astratto, il codice che usa la vecchia versione smetterebbe di compilare (in Java 8)
- Forma limitata di ereditarietà multipla (cfr. gli S*cala Traits*)
  - di metodi/operazioni
  - ma non di variabili di istanza/stato

Brian Goetz,

Lambdas in Java: A peek under the hood

https://www.youtube.com/watch?v=MLksirK9nnE

Collection.stream()

- Primo e più importante utilizzo dei nuovi "metodi astratti con implementazione di default"
  - Creazione di **`java.util.stream.Stream`** da **`java.util.Collection`**

```
public interface Collection<E> {

   default Stream<E> stream() {
      return StreamSupport.stream(spliterator(), false);
   }

   default Spliterator<E> spliterator() {
      return Spliterators.spliterator(this, 0);
   }
}
```

Costruzione di *Stream*

- Si astrae dai dettagli del singolo *stream*
- Si suppone l'esistenza di un generico strumento di accesso e partizionamento agli elementi
- ***`java.util.Spliterator`***
  - *Split:* per specificare come *spezzare* lo stream
  - *Iterator:* per scandire gli elementi dello stream

```
// Es. ArrayList class (>>)
@Override
public Spliterator<E> spliterator() {
    return new ArrayListSpliterator<>(this, 0, -1, 0);
}
```

- Perché una nuova collezione possa essere utilizzata come *stream* basta implementare un corrispondente *spliterator*

java.util.Spliterator

```
// Spliterator interface
boolean tryAdvance(Consumer<? super T> action);
```

- Consuma il successivo elemento se esiste
- Gli *stream* non assumono che tutti gli elementi siano disponibili a tempo di costruzione
  - ad es. funzioni generatrici di stream
- Notare la scelta ben diversa rispetto alla coppia di metodi **`next()`/`hasNext()`**
  - ✔ per semplificare la gestione concorrente:
  - sarebbero possibili s.e.a. problematiche *tra* le due invocazioni
    - ✔ che normalmente sono *fortemente accoppiate*!

SplitIterator.trySplit()

```
// Spliterator interface
Spliterator<T> trySplit();
```

- Metodo alla base di un processo di decomposizione parallela dello stream da processare
- Serve a generare uno ***Spliterator*** *figlio* che rilevi una parte della collezione da processare
- Il padre conserva a sua volta la restante parte
- Può generare (ricorsivamente) altri ***Spliterator***

✔ Non a caso esattamente ciò che serve per la Decomposizione Parallela (>>)!

`SplitIterator`: Metodi Default

```
// Spliterator interface
long estimateSize();
```

- Sono forniti anche altri metodi (con una implementazione) di default, tra i quali, per chiudere il processamento:

  `forEachRemaining()`

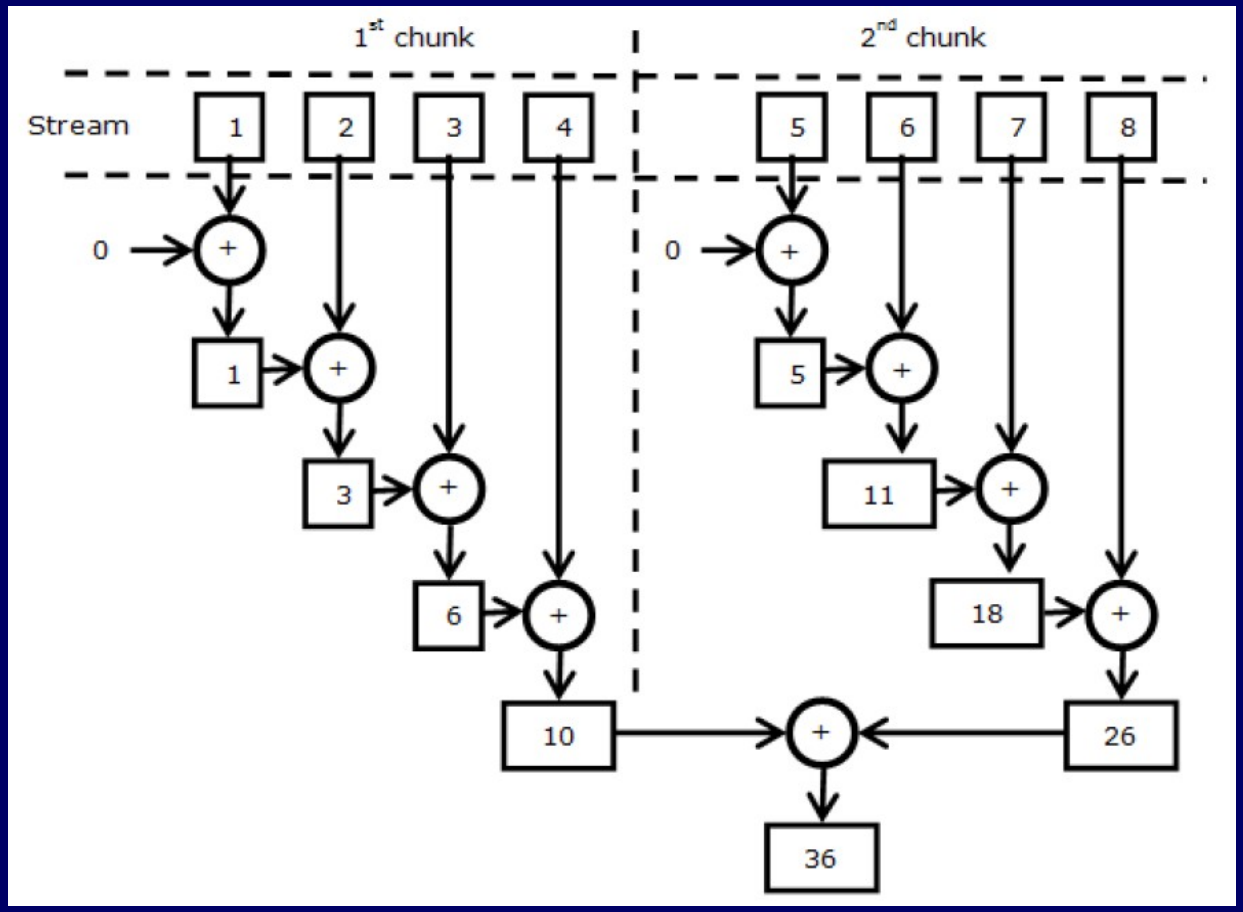
```
// Spliterator interface
@Override
default void forEachRemaining(Consumer<? super T> action) {
    do { } while (tryAdvance(action));
}
```

Processamento Parallelo di Stream

```
employees.parallelStream()
         .filter(e -> e.getRole() == Role.MANAGER)
         .map(Employee::getIncome)
         .reduce(0, (a, b) -> a + b);
```

Riduzione Parallela

- La riduzione deve essere associativa per essere eseguita concorrentemente



Processamento Parallelo di Stream (1)

- Si basa su di un sottostante motore di traduzione che opera in questo modo:
  - *Input*: specifica della *stream pipeline* di processamento
    - usando i metodi offerti da **`java.util.stream.Stream`** e basati sull'uso di lambda expression
  - *Output*: task da sottomettere al Fork/Join Framework
    - usando sottoclassi di **`java.util.concurrent.ForkJoinTask`** perfettamente *nascoste* all'utilizzatore
- Ha indotto a migliorare il framework stesso
  - ora supporta (anche) la *decomposizione esistenziale*
  - vedi ***`java.util.concurrent.CountedCompleter`***

Processamento Parallelo di Stream (2)

- Perché la traduzione sia possibile, si sfrutta la disponibilità di ***`Spliterator`***
  - sia per accedere agli elementi dello stream da processare
    - metodo **`tryAdvance()`**
  - sia per decomporre il problema se ritenuto proficuo
    - metodo **`trySplit()`**
  - ma quando conviene decomporre?
    - Metodo **`estimateSize()`**

ArraySpliterator (1)

```
static final class ArraySpliterator<T> implements Spliterator<T> {

    private final Object[] array;
    private int index;        // current index, modified on advance/split
    private final int fence;  // one past last index
    ...
    public ArraySpliterator(Object[] array, ...) {
        this(array, 0, array.length, ...);
    }


    public ArraySpliterator(Object[] array, int origin, int fence...) {
        this.array = array;
        this.index = origin;
        this.fence = fence;
        ...
    }
```

ArraySpliterator (2)

```
    @Override
    public Spliterator<T> trySplit() {
        int lo = index, mid = (lo + fence) >>> 1;
        return (lo >= mid)
               ? null
               : new ArraySpliterator<>(array, lo, index = mid,...);
    }

    @Override
    public boolean tryAdvance(Consumer<? super T> action) {
        if (action == null)
            throw new NullPointerException();
        if (index >= 0 && index < fence) {
            @SuppressWarnings("unchecked") T e = (T) array[index++];
            action.accept(e);
            return true;
        }
        return false;
    }
    @Override
    public long estimateSize() { return (long)(fence - index); }
}
```

Ancora sul Java Collection Framework, (1) Decomposizione Parallela e Lambda Function

- **`ConcurrentHashMap`** supporta *direttamente* la decomposizione parallela per diverse operazioni

`ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)`

Creates a new, empty map with an initial table size based on the given number of elements (`initialCapacity`), table density (`loadFactor`), and number of concurrently updating threads (`concurrencyLevel`).

  - per il processamento parallelo degli elementi (coppie chiave-valore)
  - la parallelizzazione è gestita *internamente* alla classe
  - esattamente come per gli *stream*
    - processamento parallelo senza conoscere il framework Fork/Join
    - è possibile sfruttare le lambda function per specificare *dall'esterno* quali operazioni svolgere parallelamente
    - internamente sono creati i task da sottomettere al `ForkJoinPool.commonPool()`;
  - a differenza degli *stream* (che sono *immutabili* e monouso)
    - decomp. progettata per lavorare in presenza di aggiornamenti concorrenti

Ancora sul Java Collection Framework, (2) Decomposizione Parallela e Lambda Function

- Perché proprio questa classe è stata estesa in questo modo?
  - ✔ Notare che l'interfaccia ***`ConcurrentMap`*** e l'altra sua implementazione **`ConcurrentSkipListMap`** non sono state cambiate in tal senso
  - ❑ e anche altre collezioni supportano accessi concorrenti
- Perché la natura *non completamente sincronizzata* della classe rende l'utilizzo semplice e proficuo anche in presenza di aggiornamenti…
  - ❑ scalabile rispetto al numero dei f.d.e.

ConcurrentHashMaps support a set of sequential and parallel bulk operations that, unlike most `Stream` methods, are designed to be safely, and often sensibly, applied even with maps that are being concurrently updated by other threads; for example, when computing a snapshot summary of the values in a shared registry. There are three kinds of operation, each with four forms, accepting functions with Keys, Values, Entries, and (Key, Value) arguments and/or return values. Because the elements of a ConcurrentHashMap are not ordered in any particular way, and may be processed in different orders in different parallel executions, the correctness of supplied functions should not depend on any ordering, or on any other objects or values that may transiently change while computation is in progress; and except for forEach actions, should ideally be side-effect-free.

Ancora sul Java Collection Framework, (3) Decomposizione Parallela e Lambda Function

- Es. di operazione *bulk* parallela

forEach

```
public void forEach(long parallelismThreshold,
                    BiConsumer<? super K,? super V> action)
```

Performs the given action for each (key, value).

Parameters:

`parallelismThreshold` - the (estimated) number of elements needed for this operation to be executed in parallel

`action` - the action

- Più specificatamente, per le sole *Map*

search

```
public <U> U search(long parallelismThreshold,
                    BiFunction<? super K,? super V,? extends U> searchFunction)
```

Returns a non-null result from applying the given search function on each (key, value), or null if none. Upon success, further element processing is suppressed and the results of any other parallel invocations of the search function are ignored.

Type Parameters:

`U` - the return type of the search function

Parameters:

`parallelismThreshold` - the (estimated) number of elements needed for this operation to be executed in parallel

`searchFunction` - a function returning a non-null result on success, else null

Returns:

a non-null result from applying the given search function on each (key, value), or null if none

Riferimenti

- Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft. Java 8 in Action: Lambdas, Streams, and functional-style programming – Manning – Parte II, Capitoli 3; 4-7

- *When to use parallel streams?*

  http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html