

Java

Lambda Expressions

Metodi Avanzati di Programmazione
(Corso B)

Dipartimento di Informatica
Università degli Studi di Bari Aldo Moro

docente: dott. Pierpaolo Basile

Note e riferimenti

Note

Gli esempi di queste slide sono riportati nel package `di.uniba.map.b.lab.lambda` del repository github del corso

Riferimenti

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

Lambda Expressions

Per comprendere il funzionamento delle lambda expressions partiremo da un esempio pratico legato all'utilizzo delle **classi anonime**.

Supponiamo di avere una classe Person che memorizza delle informazioni relative ad una persona

```
public class Person {  
    public enum Gender {  
        MALE, FEMALE  
    }  
  
    private String name;  
  
    private String surname;  
  
    private int age;  
  
    private Gender gender;  
  
    public Person(String name, String surname, int age, Gender gender) {  
        this.name = name;  
        this.surname = surname;  
        this.age = age;  
        this.gender = gender;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

La classe Person

La classe Person avrà i relativi metodi get e set per gli attributi privati.

Inoltre ha un metodo (printPerson) che stampa a video le informazioni sulla persona.

```
public void setGender(Gender gender) {  
    this.gender = gender;  
}  
  
public void printPerson() {  
    System.out.println("Person{" + "name=" + name + ", surname=" + surname + ", age=" + age + ", gender=" + gender + '}');  
}
```

Ricerca

Un'operazione comune potrebbe essere quella di ricercare delle istanze (persone) della classe `Person` che rispettano alcune caratteristiche.

Ad esempio, vogliamo tutte le persone che superano una certa età. Potremmo definire un metodo di questo tipo:

```
//Create Methods That Search for Members That Match One Characteristic
public static void printPersonsOlderThan(List<Person> roster, int age) {
    for (Person p : roster) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

Questo tipo di codice è poco “elegante” poiché è fortemente connesso alle caratteristiche della classe `Person` e al modo con cui l'età è memorizzata nella classe. In caso di cambiamenti a `Person` dovremo riscrivere il codice.

Ricerca

Una soluzione più generica potrebbe essere:

```
//Create More Generalized Search Methods
public static void printPersonsWithinAgeRange(List<Person> roster, int low, int high) {
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

Questo metodo è ancora legato a come l'età (attributo age) è memorizzato in Person e soprattutto non è generico poiché mi permette di filtrare/cercare solo sull'attributo età.

Ricerca con interfaccia

Potremmo definire un'interfaccia `CheckPerson` che ha un unico metodo che verifica se una istanza di `Person` è valida o meno

```
//  
public interface CheckPerson {  
  
    boolean test(Person p);  
}
```

e un metodo che utilizza questa interfaccia per verificare se una persona rispetta o meno dei criteri che saranno definiti nell'implementazione del metodo `test`.

```
//Specify Search Criteria Code in a Local Class  
public static void printPersons(List<Person> roster, CheckPerson tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```


Ricerca, utilizzo interfaccia

Forniamo un'implementazione dell'interfaccia CheckPerson

```
public class CheckPersonEligibleForSelectiveService implements CheckPerson {  
  
    @Override  
    public boolean test(Person p) {  
        return p.getGender() == Person.Gender.MALE  
            && p.getAge() >= 18  
            && p.getAge() <= 25;  
    }  
  
}
```

Utilizzo della classe CheckPersonEligibleForSelectiveService

```
//create a new instance of CheckPerson  
printPersons(persons, new CheckPersonEligibleForSelectiveService());  
//use of an anonymous class
```



In questo caso creiamo una nuova istanza e la passiamo come parametro al metodo printPersons

Ricerca (classe anonima)

```
//use of an anonymous class
printPersons(persons, new CheckPerson() {
    @Override
    public boolean test(Person p) {
        return p.getGender() == Person.Gender.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25;
    }
});
```

Classe anonima: dichiarare e contemporaneamente istanziare una classe, per questo motivo la classe è priva di un nome.

Possono essere definite nel corpo di un metodo, come nell'esempio riportato nella slide. Le abbiamo utilizzate spesso con SWING per creare le ActionListener.

Interfacce funzionali

L'interfaccia `CheckPerson` è un'interfaccia funzionale. Un'interfaccia funzionale è qualsiasi interfaccia che:

- contiene solo un metodo astratto (*un'interfaccia funzionale può contenere altri metodi statici con implementazione*)
- poiché un'interfaccia funzionale contiene solo un metodo astratto, è possibile omettere il nome di quel metodo quando lo si implementa

Invece di utilizzare un'espressione di classe anonima, è possibile utilizzare un'espressione lambda.

Espressioni lambda

```
public static void printPersonsWithPredicate(List<Person> roster, Predicate<Person> tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```

Predicate<T> è un'interfaccia funzionale definita in `java.util.function` e funziona in modo simile a `CheckPerson`, ma è **generica** e prevede il solo metodo: `boolean test(T t)`. Poiché ha un solo metodo astratto non è necessario specificare il suo nome e può essere utilizzata in un'espressione lambda nel seguente modo:

```
printPersons(persons.  
    (Person p) -> p.getGender() == Person.Gender.MALE // Predicate function  
    && p.getAge() >= 18  
    && p.getAge() <= 25  
);
```

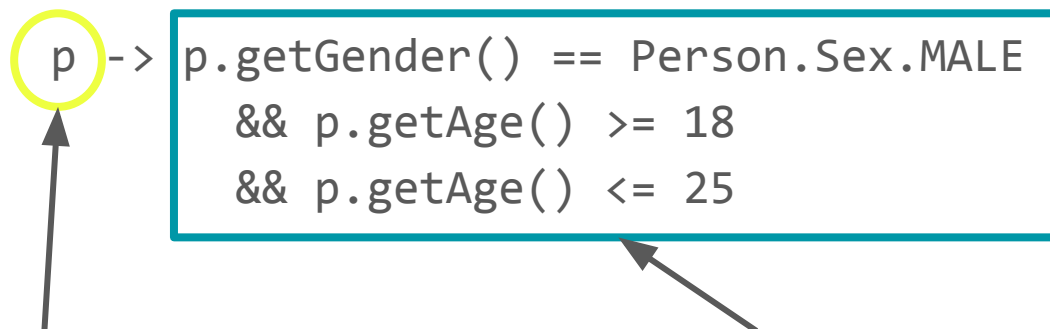
lambda expression

Sintassi delle espressioni lambda

Un'espressione lambda è composta:

- da una **lista di parametri formali** separati da virgole e racchiusi tra parentesi tonde: nell'esempio precedente c'è un solo parametro formale p
***NB:** è possibile omettere il tipo di dati dei parametri in un'espressione lambda. Inoltre, è possibile omettere le parentesi se esiste un solo parametro*
- il token **->**
- un **corpo** che contiene una singola espressione o un blocco di istruzioni. Il blocco di istruzioni è racchiuso nelle parentesi graffe {}, se il blocco di istruzioni contiene un'invocazione ad un metodo che non restituisce nessun valore (void) si possono omettere le parentesi

Sintassi delle espressioni lambda



parametri formali

espressione

p -> {return p.getGender() == Person.Sex.MALE
&& p.getAge() >= 18
&& p.getAge() <= 25;}

Sintassi con blocco di istruzioni

email -> System.out.println(email): nel caso venga invocato un metodo che restituisce un valore void si può omettere il blocco di istruzioni {}

Sintassi delle espressioni lambda

Come detto in precedenza possiamo avere più di un parametro formale

```
public class Calculator {  
  
    interface IntegerMath {  
        int operation(int a, int b);  
    }  
  
    public int operateBinary(int a, int b, IntegerMath op) {  
        return op.operation(a, b);  
    }  
  
    public static void main(String... args) {  
  
        Calculator myApp = new Calculator();  
        IntegerMath addition = (a, b) -> a + b;  
        IntegerMath subtraction = (a, b) -> a - b;  
        System.out.println("40 + 2 = " +  
            myApp.operateBinary(40, 2, addition));  
        System.out.println("20 - 10 = " +  
            myApp.operateBinary(20, 10, subtraction));  
    }  
}
```

Perché espressioni lambda?

Le espressioni lambda sono esempi di **programmazione funzionale**:

- il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni
- mancanza di *side-effect*
- la programmazione funzionale pone il focus sulla definizione di funzioni
 - contrariamente, i paradigmi procedurali e imperativi prediligono la specifica di una sequenza di comandi da eseguire e i valori vengono calcolati cambiando lo stato del programma attraverso l'operazione di assegnazione
 - un programma funzionale è immutabile: i valori non vengono calcolati cambiando lo stato del programma, ma costruendo nuovi stati a partire dai precedenti
- la programmazione funzionale ha le sue radici nel lambda calcolo
 - calcolo basato sulle funzioni composto da un linguaggio formale utilizzato per esprimere le funzioni e un sistema di riscrittura per stabilire come i termini possano essere ridotti e semplificati

Consumer

Riprendiamo il metodo `printPersonWithPredicate`. Questo metodo filtra le persone in base ad una funzione `Predicate` passata come parametro e chiama il metodo `printPerson` di `Person`.

E se volessimo **generalizzare anche sull'operazione** da applicare alle istanze per cui `test` restituisce `true`?

```
public static void printPersonsWithPredicate(List<Person> roster, Predicate<Person> tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```


Consumer

```
//Consumer<T> is a predefined function in java.util.function that works as printPerson but with generics
public static void processPersons(List<Person> roster, Predicate<Person> tester, Consumer<Person> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            block.accept(p);
        }
    }
}
```

L'interfaccia `Consumer<T>` definita in `java.util.function`, funziona esattamente come l'istruzione `printPerson`, ovvero esegue un'operazione sull'istanza `p` di `Person` senza restituire alcun valore. `Consumer` ha un solo metodo:

```
void accept(T t)
```

Consumer

```
//Consumer<T> is a predefined function in java.util.function that works as printPerson but with generics
public static void processPersons(List<Person> roster, Predicate<Person> tester, Consumer<Person> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            block.accept(p);
        }
    }
}
```

```
processPersons(people,
    p -> p.getGender() == Person.Gender.MALE // Predicate function
    && p.getAge() >= 18
    && p.getAge() <= 25,
    p -> p.printPerson() // Consumer function
);
```

E se prima di *consumare* un'istanza volessimo **effettuare un'operazione** su di essa, ad esempio recuperare il cognome e stampare solo esso?

Function

```
//Function<T, R> is a predefined function in java.util.function that works as R apply(T t).  
//Apply performs an action on T and return R  
public static void processPersonsWithFunction(List<Person> roster, Predicate<Person> tester,  
    Function<Person, String> mapper, Consumer<String> block) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            String data = mapper.apply(p);  
            block.accept(data);  
        }  
    }  
}
```

L'interfaccia `Function<T, R>` definita in `java.util.function`, **esegue** un'operazione sull'istanza di tipo `T` restituendo un'istanza di tipo `R`.

Function ha un solo metodo:

`R apply(T t)`

Function

```
//Function<T, R> is a predefined function in java.util.function that works as R apply(T t).  
//Apply performs an action on T and return R  
public static void processPersonsWithFunction(List<Person> roster, Predicate<Person> tester,  
    Function<Person, String> mapper, Consumer<String> block) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            String data = mapper.apply(p);  
            block.accept(data);  
        }  
    }  
}
```

```
processPersonsWithFunction(people,  
    p -> p.getGender() == Person.Gender.MALE // Predicate function  
    && p.getAge() >= 18  
    && p.getAge() <= 25,  
    p -> p.getSurname(), // Function  
    surname -> System.out.println(surname) //Consumer  
);
```

Visibilità delle variabili

Un'espressione lambda non introduce un nuovo livello di scopo, infatti sono *lexically scoped*. Possono accedere a variabili definite nello scopo locale nel quale è scritta l'espressione lambda, esattamente come avviene per le classi anonime

```
import java.util.function.Consumer;

public class LambdaScopeTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {

            // The following statement causes the compiler to generate
            // the error "local variables referenced from a lambda expression
            // must be final or effectively final" in statement A:
            //
            // x = 99;

            Consumer<Integer> myConsumer = (y) ->
            {
                System.out.println("x = " + x); // Statement A
                System.out.println("y = " + y);
                System.out.println("this.x = " + this.x);
                System.out.println("LambdaScopeTest.this.x = " +
                LambdaScopeTest.this.x);
            };

            myConsumer.accept(x);
        }
    }
}
```

Generalizziamo

Possiamo rendere il metodo `checkPersonsWithFunction` generico, in modo da poterlo applicare a qualsiasi tipo di classe e non solo a `Person`? E renderlo contemporaneamente generico anche rispetto al tipo restituito dall'implementazione di `Function`?

```
//Function<T, R> is a predefined function in java.util.function that works as R apply(T t).  
//Apply performs an action on T and return R  
public static void processPersonsWithFunction(List<Person> roster, Predicate<Person> tester,  
    Function<Person, String> mapper, Consumer<String> block) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            String data = mapper.apply(p);  
            block.accept(data);  
        }  
    }  
}
```

Generalizziamo

Possiamo rendere il metodo `checkPersonsWithFunction` generico, in modo da poterlo applicare a qualsiasi tipo di classe e non solo a `Person`? E renderlo contemporaneamente generico anche rispetto al tipo restituito dall'implementazione di `Function`?

```
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);
        }
    }
}
```

- `X` è il generics che fa riferimento agli oggetti che voglio processare
- `Iterable<X>` è il contenitore degli oggetti di tipo `X` sul quale posso effettuare un'operazione di iterazione sugli oggetti contenuti
- `Y` è il generics relativo al risultato della funzione `mapper` (di `Function`)

Generalizziamo

```
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);
        }
    }
}
```

```
processElements(persons,
    p -> p.getGender() == Person.Gender.MALE // Predicate function
    && p.getAge() >= 18
    && p.getAge() <= 25,
    p -> p.getSurname(), // Function
    surname -> System.out.println(surname) //Consumer
);
```

```
processElements(persons,
    p -> p.getGender() == Person.Gender.MALE // Predicate function
    && p.getAge() >= 18
    && p.getAge() <= 25,
    p -> p.getAge(), // Function
    age -> System.out.println(age) //Consumer
);
```


Generalizziamo

Nel dettaglio il metodo `processElements` lavora in questo modo:

- **Ottiene** una sorgente di oggetti da un iteratore. Nell'esempio gli oggetti sono di tipo `Person` e la sorgente è una `List` che implementa l'interfaccia `Iterable`
- **Filtra** gli oggetti in base all'oggetto `Predicate`. Nel nostro caso definito dall'espressione `lambda` che controlla il genere e l'età
- **Mappa** tutti gli oggetti filtrati su un altro valore definito da `Function`. In questo caso un'altra espressione `lambda` che restituisce il cognome
- **Esegue** un'azione sull'oggetto mappato definita dall'oggetto `Consumer`. In questo caso stampa una stringa che è il cognome dell'oggetto restituito da `Function`

Operazioni aggregate

Quello che fa il metodo `processElements` si può ottenere attraverso l'utilizzo degli **stream** e delle **operazioni aggregate**.

```
//processElements can be replaced by a pipeline applied to a stream
persons
    .stream()
    .filter(p -> p.getGender() == Person.Gender.MALE // Predicate function
    && p.getAge() >= 18
    && p.getAge() <= 25)
    .map(p -> p.getSurname())
    .forEach(surname -> System.out.println(surname));
```

Le operazioni `filter`, `map` e `forEach` sono delle **operazioni aggregate**. Esse processano degli elementi a partire da uno **stream** (che non è una collection). Uno stream è una **sequenza di elementi** e diversamente dalle collection non è una struttura dati che contiene elementi. Invece, uno stream preleva i valori da una sorgente, come ad esempio una collection, attraverso una **pipeline**. Una pipeline è una **sequenza di operazioni** (es. `filter-map-foreach`), i parametri di queste operazioni in genere sono espressioni lambda e ciò permette facilmente di personalizzare il loro comportamento.

Pipeline e stream

Una pipeline è una sequenza di operazioni aggregate applicate ad uno stream.

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

In questo caso abbiamo due operazioni, una filtra (`filter`) gli oggetti, l'altra esegue un'operazione su tutti gli elementi (`forEach`) dello stream. La pipeline in questo caso equivale a:

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

Pipeline

Una pipeline è composta da:

- **una sorgente**: che potrebbe essere una collection, un array, una *funzione generatore* o un canale di I/O
- zero o più **operazioni intermedie**: ogni operazione intermedia crea un nuovo stream, ad esempio `filter`
- un'**operazione terminale**: produce un risultato (finale) che non è più uno stream. Ad esempio, il `forEach` semplicemente esegue un'operazione su ogni elemento dello stream

Esempi di pipeline

Calcola la media delle persone di genere maschile. L'operazione average è un'operazione terminale

```
double avgAge = persons
    .stream()
    .filter(p -> p.getGender() == Person.Gender.MALE)
    .mapToInt(p -> p.getAge())
    .average().getAsDouble();
System.out.println(avgAge);
```

Conta il numero di persone maggiorenni, anche in questo caso count è un'operazione terminale

```
long m = persons
    .stream()
    .filter(p->p.getAge()>17)
    .mapToInt(p->p.getAge())
    .count();
System.out.println(m);
```

Operazioni terminali

La lista delle operazioni terminali è presente nella Javadoc dell'interfaccia Stream

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>

e nella Javadoc delle sue diverse implementazioni:

- IntStream
- LongStream
- DoubleStream

Funzioni generatrici

Stream ha un metodo `generate` che permette di generare uno stream infinito di valori la cui generazione è regolata dal parametro di tipo `Supplier`.

```
generate(Supplier<T> s)
```

L'interfaccia `Supplier` ha un solo metodo `get` che restituisce un valore di tipo `T`.

L'altro metodo di `Stream` che genera valori è `iterate` che genera valori infiniti a partire da un `seed` al quale viene applicato iterativamente una funzione `f`:

```
seed, f(seed), f(f(seed), f(f(f(seed), ...
```

Ogni implementazione di `Stream` (`IntStream`, `DoubleStream`, ...) ha altri metodi che funzionano come generatori, ad esempio `IntStream` ha dei generatori che creano interi all'interno di un intervallo specifico.

Funzioni generatrici (esempi)

Stampa cento numeri pari a partire da 2.

```
IntStream.iterate(2, i -> i+2)
    .limit(100)
    .forEach(p -> System.out.println(p));
```

Stampa le prime 11 potenze di 2.

```
IntStream.rangeClosed(0, 10)
    .map(i -> (int) Math.pow(2, i))
    .forEach(p -> System.out.println(p));
```

Calcola e stampa la media dei numeri interi nell'intervallo [0, 10)

```
System.out.println(IntStream.range(0, 10)
    .average().getAsDouble());
```


Funzioni generatrici (esempi)

Crea uno stream infinito di numeri random nell'intervallo [0, 1), filtra i primi 100 maggiori o uguali a 0.5 e li stampa.

```
DoubleStream.generate(() -> Math.random())  
    .filter(p -> p >= 0.5)  
    .limit(100)  
    .forEach(p -> System.out.println(p));
```

La funzione Supplier è definita attraverso un'espressione lambda senza parametri che esegue un'istruzione che genera un double, in questo caso il metodo statico random di Math

Genera 100 numeri random nell'intervallo [0, 1), filtra quelli maggiori a 0.5 e conta quanti sono. Il risultato finale viene stampato.

```
System.out.println(DoubleStream.generate(() -> Math.random())  
    .limit(100)  
    .filter(p -> p > 0.5)  
    .count());
```

Riduzione (reduction)

- Le operazioni terminali come average e count si chiamano **riduzioni**. Queste restituiscono un unico valore combinando gli oggetti presenti nello stream
- Esistono anche operazioni di riduzione che restituiscono una collezione e non un singolo valore
- Esempi di operazioni di riduzione che restituiscono un singolo valore sono: average, sum, min, max e count
- Oltre alle operazioni di riduzione predefinite è possibile definirne di nuove utilizzando i metodi reduce e collect di Stream



Stream.reduce

```
Integer totalAge = persons
    .stream()
    .mapToInt(p -> p.getAge())
    .sum();
System.out.println(totalAge);
```

Nell'esempio precedente utilizziamo la funzione di riduzione sum per calcolare la somma di tutte le età, la stessa funzione potrebbe essere espressa utilizzando il metodo reduce di Stream

```
totalAge = persons
    .stream()
    .map(p -> p.getAge())
    .reduce(0,
        (a,b) -> a+b);
System.out.println(totalAge);
```

Stream.reduce

```
totalAge = persons
    .stream()
    .map(p -> p.getAge())
    .reduce(0, 
        (a,b) -> a+b); 
System.out.println(totalAge);
```

reduce prende in input due parametri:

1. **identity**: il valore iniziale della riduzione o il valore che viene restituito nel caso non ci siano elementi nello stream
2. **accumulator**: la funzione accumulatore accetta due parametri dello stesso tipo e restituisce un risultato. In questo esempio, la funzione accumulatore è un'espressione lambda che aggiunge due valori Integer e restituisce un valore Integer. I due parametri della funzione accumulatore sono: il risultato parziale della riduzione (in questo esempio, la somma di tutti gli interi elaborati finora) e l'elemento successivo dello stream (in questo esempio, un numero intero, l'età)

Stream.collect

- La funzione `reduce` restituisce ogni volta un nuovo valore.
Ricordiamo che la programmazione funzionale è priva di *side-effect*
- Nel caso precedente il nuovo valore è di tipo `Integer` quindi non abbiamo grossi problemi di performance, ma se la funzione `reduce` lavorasse con oggetti più complessi, ad esempio `Collection`, ogni volta che viene chiamata creerebbe una nuova `Collection`
- In questi casi è opportuno utilizzare il metodo `collect` che effettua un update dell'oggetto che si sta ottenendo dalla riduzione dello stream
- Il metodo `collect` contrariamente al metodo `reduce`, modifica un valore preesistente

Stream.collect

Il metodo `collect` accetta tre parametri:

1. `supplier`: è il costruttore dell'oggetto che verrà restituito dal metodo `collect` e che si presuppone venga modificato durante l'operazione di riduzione
2. `accumulator`: questa funzione serve ad inglobare il valore attuale dello stream nell'oggetto che verrà restituito
3. `combiner`: questa funzione combina due contenitori di risultati e unisce il loro contenuto

Supponiamo di voler implementare l'operatore terminale che calcola la media attraverso il metodo `collect`. Abbiamo bisogno di definire una classe `Average` che manterrà la somma corrente e il numero totale degli elementi esaminati nello stream. Questa classe verrà poi utilizzata nel metodo `collect`

Stream.collect

```
public class Averager implements IntConsumer {  
    private int sum = 0;  
    private int count = 0;  
  
    public double average() {  
        return count > 0 ? ((double) sum) / count : 0;  
    }  
  
    @Override  
    public void accept(int value) {  
        sum += value;  
        count++;  
    }  
  
    public void combine(Averager other) {  
        sum += other.sum;  
        count += other.count;  
    }  
}
```

La class Averager implementa l'interfaccia IntConsumer che prevede il solo metodo accept che verrà utilizzato come **accumulator**. Il costruttore sarà il metodo utilizzato come **supplier**, invece il metodo combine come **combiner**.

Il metodo average ci restituirà il risultato finale.

```
Averager avgAgeC = persons  
    .stream()  
    .map(p -> p.getAge())  
    .collect(Averager::new, Averager::accept, Averager::combine);  
System.out.println(avgAgeC.average());
```

Stream.collect

```
Averager avgAgeC = persons
    .stream()
    .map(p -> p.getAge())
    .collect(Averager::new, Averager::accept, Averager::combine);
System.out.println(avgAgeC.average());
```

L'operatore (`::`) si chiama **method reference operator** e ci permette di chiamare un metodo facendo riferimento ad esso direttamente con il nome della sua classe. Sono equivalenti alle espressioni lambda che invocano direttamente il metodo, ma più immediati nell'utilizzo perché basati solo sul nome del metodo.

L'istruzione precedente corrisponde a:

```
Averager avgAgeC = persons
    .stream()
    .map(p -> p.getAge())
    // .collect(Averager::new, Averager::accept, Averager::combine);
    .collect(() -> new Averager(), (a, b) -> a.accept(b), (a, b) -> a.combine(b));
System.out.println(avgAgeC.average());
```


Stream.collect

- in `collect` il `supplier` è un'espressione `lambda` (o un riferimento al metodo) al contrario in `reduce` è un valore (l'elemento identità nell'operazione di riduzione)
- le funzioni `accumulator` e `combiner` non restituiscono un valore
- È possibile utilizzare le operazioni di `collect` con flussi paralleli (in esecuzione concorrente): la JDK crea un nuovo thread ogni volta che la funzione combinator crea un nuovo oggetto, come un oggetto `Averager` nell'esempio precedente (non è necessario preoccuparsi della sincronizzazione). Per maggiori dettagli:
<https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html>

Stream.collect

Esiste un'altra implementazione del metodo `collect` che prende come parametro un `Collector` che è una classe che incapsula al suo interno `supplier`, `accumulator` e `combiner`.

La classe `Collectors`¹ contiene numerosi metodi statici che restituiscono svariati implementazioni di `Collector`.

```
List<String> namesOfMaleMembersCollect = persons
    .stream()
    .filter(p -> p.getGender() == Person.Gender.FEMALE)
    .map(p -> p.getName())
    .collect(Collectors.toList());
System.out.println(namesOfMaleMembersCollect);
```

Il `Collector` `toList` trasforma i risultati ottenuti dallo stream in una `List`. In questo caso il tipo di oggetti contenuto in `List` è determinato dall'operazione `map`.

¹ <https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html>

Group by

Il collector `groupBy` mi permette di raggruppare gli oggetti in base ad una funzione di classificazione. Nell'esempio seguente filtriamo tutte le persone con età maggiore a 17 e li raggruppiamo in base al genere. Questo collector restituisce una Map

```
Map<Person.Gender, List<Person>> collect = persons
    .stream()
    .filter(p -> p.getAge() > 17)
    .collect(Collectors.groupingBy(Person::getGender));
```

Esiste anche un'implementazione di `groupBy` con due parametri: il primo è la funzione di classificazione, il secondo detto *downstream collector* è un collector che viene applicato al risultato di un altro collector

Group by con downstream collector

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

Il Collector **mapping** che utilizziamo qui come *downstream collector* produce una List dove il tipo degli elementi dipende dalla funzione passata come primo argomento a mapping. In questo caso raggruppiamo per genere e inseriamo nella lista solo il nome

Group by con downstream collector

```
Map<Person.Gender, Integer> totalAgeByGender = persons
    .stream()
    .collect(
        Collectors.groupingBy(
            Person::getGender,
            Collectors.reducing(
                0,
                Person::getAge,
                Integer::sum)));
```

In questo caso il *downstream collector* è un'operazione di riduzione che effettua la somma delle età, nell'esempio raggruppate per genere

Esecuzione parallela

Verranno forniti alcuni esempi di esecuzione parallela sugli stream.

Calcolo della media:

```
double average = persons
    .parallelStream()
    .filter(p -> p.getGender() == Person.Gender.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

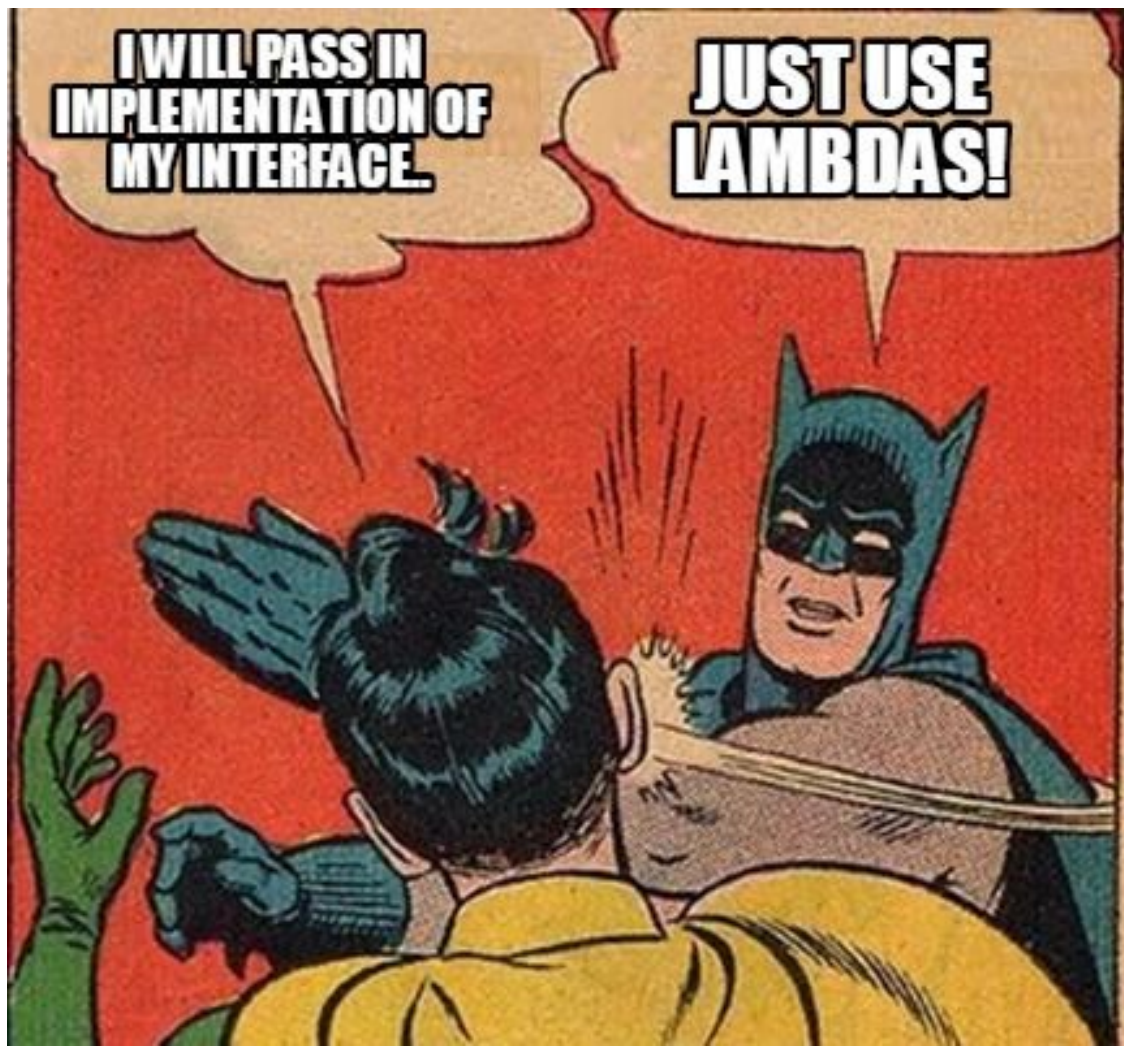
Raggruppamento delle persone in base al genere:

```
ConcurrentMap<Person.Gender, List<Person>> byGender = persons
    .parallelStream()
    .collect(
        Collectors.groupingByConcurrent(Person::getGender));
```

Per maggiori dettagli: <https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html>

**I WILL PASS IN
IMPLEMENTATION OF
MY INTERFACE.**

**JUST USE
LAMBDA!**



Esercizi

Esercizi

Negli esercizi successivi lavoreremo su un dataset chiamato Movielens 1M. Questo dataset contiene le informazioni sulle preferenze espresse dagli utenti su un insieme di film.

Il dataset è contenuto nella cartella: [resources/movielens1M](#), le classi per il caricamento del dataset e per le relative strutture dati (User, Movie, Rating) sono contenute nel package `di.uniba.map.b.lab.dataset.movielens`.

Nello stesso package è contenuta la classe `Movielens` che si occupa di caricare i dati del dataset in tre `List`: una degli utenti, una dei film e una delle preferenze.

Nella cartella del dataset è contenuto anche un file README con le informazioni sul contenuto del dataset.

Leggere attentamente il codice delle strutture dati per capire come gli attributi sono memorizzati!!!

Esercizi

Alcune note:

- la classe `Movielens` si occupa di caricare i dati attraverso i tre metodi `load` ognuno per ogni tipo di dato
 - in particolare il metodo `loadRatings` si occupa di caricare i dati da un file compresso `gzip`, potete notare l'utilizzo della classe `GZIPInputStream`
- l'età dell'utente è divisa in classi di età, guardare il file `README`
- l'occupazione dell'utente è definita attraverso un codice intero, per ottenere la stringa associata all'occupazione utilizzare il metodo statico di `User` `occupationToString` (vedere `README`)
- il rating (preferenza) è in una scala da 1 a 5
- per ordinare uno stream si può utilizzare l'operazione (metodo) `sorted` di `Stream` che restituisce un altro stream
- cercare di non utilizzare gli aiuti dell'IDE per trasformare le iterazioni in operazioni sugli stream e in caso di dubbi avvalersi della `javadoc`

Esercizi

Effettuare le seguenti operazioni utilizzando il più possibile gli stream e le espressioni lambda

1. raggruppare tutti gli utenti di genere femminile con età maggiore o uguale a 25 anni in base all'occupazione
2. raggruppare tutti gli utenti di genere maschile che hanno come occupazione "unemployed" e raggrupparli in base allo zipcode
3. raggruppare tutti gli utenti che hanno come occupazione "unemployed" e contare quanti utenti ci sono per ogni classe di età
4. contare quanti movie hanno un rating >3
5. calcolare per ogni movie il rating medio dato dagli utenti
6. calcolare il rating medio dei movie di genere "Comedy"
7. calcolare il rating medio raggruppando per genere del movie