Indice

1 Lambda expression **2**
1.1 Intro . . . 2
1.2 Lambda expression . . . 3
1.3 Method references . . . 4
1.4 Scope di una lambda . . . 4

2 Stream **5**
2.1 Creazione di uno stream . . . 5
2.1.1 Stream infiniti . . . 5
2.2 Operazioni intermedie . . . 6
2.3 Operazione di riduzione di uno stream . . . 6
2.3.1 Optional . . . 6
2.4 Operazione di raccolta di uno stream . . . 6

Lambda expression

1.1 Intro

Per comprendere ed usare le lambda, bisogna partire dalle *interfacce*.
Un'interfaccia è un meccanismo per definire un *contratto* tra due parti, il fornitore del servizio (l'interfaccia stessa) e le classi che vogliono che i loro oggetti siano utilizzabili con quel servizio.
Prendiamo ad esempio l'interfaccia Comparable<T>, avente un metodo, compareTo(T o), che restiruisce un intero e assicura di confrontare solo oggetti dello stesso tipo.
Se una classe decidesse di implementare questa interfaccia, quindi fornire un'implementazione del metodo, allora i suoi oggetti potrebbero essere ordinati da java.
Il contratto è che x.compareTo(y) deve restituire:

- un intero positivo se x viene dopo di y;
- un intero negativo nel caso contrario;
- 0 altrimenti.

N.B. La classe String implementa di suo questa interfaccia e implementa compareTo con il confronto lessicografico.
Questo spezzone di codice funziona in quanto *Arrays.sort* riesce a ordinare oggetti la cui classe implementa Comparable e, come detto prima, String la implementa.

```
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends); // friends is now ["Mary", "Paul", "Peter"]
System.out.println(Arrays.toString(friends));
```

Così come potremmo ordinare oggetti *Employee* in base al loro nome

```
public class Employee implements Comparable<Employee> {
    private String name;

    @Override
    public int compareTo(Employee other) {
        return name.compareTo(other.getName());
    }
}
```

dove, in questo caso, il compareTo di Employee delega il confronto al compareTo di String.
Se volessimo ordinare Employee/String con un altro criterio, non potremmo farlo in quanto non sarebbe possibile definire due metodi compareTo e non sarebbe possibile modificare la classe java.
Esiste una variante di Arrays.sort che oltre ad accettare una lista da ordinare, accetta un'altra interfaccia, Comparator<T>, avente un metodo compare(T o1, T o2) che restituisce un intero.
Quindi per definire un nuovo criterio, dovremo definire una nuova classe, che implementa Comparator, e passarla al metodo Arrays.sort.

```
public class SortDemo {
    public static void main(String[] args) {
        String[] friends = new String[] { "Peter", "Paul", "Mary" };
        Arrays.sort(friends, new LengthComparator());
        // [Paul, Mary, Peter]
        System.out.println(Arrays.toString(friends));
    }
}
...
class LengthComparator implements Comparator<String> {
    public int compare(String first, String second) {
        return first.length() - second.length();
    }
}
```

Se il metodo di confronto ci servisse solo in quel punto di SortDemo, saremmo comunque costretti a definire una classe e istanziarla.
Per questo motivo ci sono le *classi anonime*, un meccanismo che riduce la verbosità del codice

- che permette di dichiarare e istanziare una classe allo stesso tempo;

- sono simile alle classi locali, solo che non hanno un nome;
- la loro invocazione avviene come quella di un costruttore, solo che al suo interno c'è una classe vera e propria.

```
public class SortDemo {
    public static void main(String[] args) {
        String[] friends = new String[] { "Peter", "Paul", "Mary" };
        Arrays.sort(friends, new Comparator<String>() {
            public int compare(String first, String second) {
                return first.length() - second.length();
            }
        });
        // [Paul, Mary, Peter]
        System.out.println(Arrays.toString(friends));
    }
}
```

Quindi, fino ad ora, abbiamo visto due interfacce, avente un singolo metodo, il contratto che stiamo usando dipende dal quel singolo metodo e se non ci fosse bisogno di mantenersi uno stato per implementare quel metodo, allora sarebbe più comodo poter specificare solo quel singolo blocco di codice invece di creare una classe che implementa l'interfaccia e istanziarla o creare una classe anonima.

1.2 Lambda expression

E' una *funzione anonima*, un blocco di codice che può essere passato, assegnato, restituito in modo da essere eseguito in un secondo momento, una o più volte.
I valori gestiti sono *funzioni* e non oggetti, in java una funzione è un'istanza di un oggetto che implementa una certa interfaccia.
Nell'esempio di LengthComparator, a noi basterebbe dire che, per confrontare due stringhe, bisogna usare il blocco di codice di compare, specificando che first e second sono oggetti di tipo String.
Quindi dovremmo passare ad Arrays.sort una funzione che, dati due oggetti String, restituisce
first.length() − second.length().
In java, la sintassi per definire questa funzione è

```
(String first, String second) -> first.length() - second.length()
```

che risulta essere la nostra lambda expression.
Quindi, nel metodo di Arrays.sort, invece di passare un'istaza di una classe che implementa Comparator o una classe anonima, gli passiamo la lambda.

```
public class SortDemo {
    public static void main(String[] args) {
        String[] friends = new String[] { "Peter", "Paul", "Mary" };
        Arrays.sort(friends, new Comparator<String>() {
            public int compare(String first, String second) {
                return first.length() - second.length();
            }
        });
        // [Paul, Mary, Peter]
        System.out.println(Arrays.toString(friends));
    }
}
...
public class SortDemo {
    public static void main(String[] args) {
        String[] friends = new String[] { "Peter", "Paul", "Mary" };
        Arrays.sort(friends, (String first, String second) -> first.length() - second.length());
        // [Paul, Mary, Peter]
        System.out.println(Arrays.toString(friends));
    }
}
```

Il body di una lambda viene eseguito non quando viene passata al metodo sort ma quando bisogna effetivamente confrontare gli oggetto (*esecuzione differita*) e se il body dovesse aver bisogno di più righe, allora si userebbero le parentesi graffe e il return.

Java può inferire il tipo dei parametri della lambda dal contesto, in tal caso si possono omettere i tipi, stessa cosa per il tipo di ritorno anche se qui java fa un controllo che sia utilizzabile nel contesto in cui viene usata la lambda.
Si può assegnare/passare una lambda quando ci si aspetta un oggetto dichiarato di tipo interfaccia

- che ha un singolo metodo astratto;
- purché la lambda sia compatibile con tale metodo, considerando il tipo dei parametri della lambda, che devono essere compatibili coi parametri del metodo, e del tipo inferito del body della lambda che deve essere compatibile col tipo di ritorno del metodo.

Una tale interfaccia è detta *interfaccia funzionale* o *SAM* (Single Abstract Method).

1.3 Method references

Il codice che si scrive in una lambda expression richiama semplicemente un metodo che è già implementato, in questi casi, invece di passare/assegnare una lambda che chiama semplicemente quel metodo passandogli i parametri della lambda, si passa/assegna un riferimento a quel metodo (*method reference*), attraverso la notazione '::'.
Abbiamo tre tipi di method reference

- Class::instanceMethod dove il primo parametro diventa il ricevente del metodo, gli altri sono passati al metodo

```
Arrays.sort(strings, (x, y) -> x.compareToIgnoreCase(y));

Arrays.sort(strings, String::compareToIgnoreCase);
```

- Class::staticMethod dove tutti parametri sono passati al metodo statico

```
list.removeIf(x -> Objects.isNull(x));

list.removeIf(Objects::isNull);
```

- Class:instanceMethod dove il metodo viene richiamato sull'object specificato prima dei :: mentre gli altri parametri sono passati al metodo

```
strings.forEach(x -> System.out.println(x));

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

Con i method reference si usa uno stile più dichiarativo, si scrive meno codice e lo si legge meglio.

1.4 Scope di una lambda

Un'interfaccia funzionale può avere tanti metodi statici e di default ma basta che abbiamo un singolo metodo astratto.
Un'interfaccia conviene annotarla con il tag @FunctionalInterface, così facendo il compilatore controllerà che il vincolo sia rispettato e gli altri utenti sapranno che quell'interfaccia è pensata come funzionale.
Non possiamo dichiarare variabili locali in una lambda o parametri di una lambda con lo stesso nome di variabili già definite nei blocchi esterni.
Una lambda può riferirsi a variabili definite NON dentro la lambda purchè dichiarate final o effectively final, si dice che ha un *ambito di visibilità circostante* (enclosing scope), per esempio

```
String message = "Hello ";
repeat(10, (x) -> System.out.println(message + x));
```

La lambda si riferisce alla variabile 'message' ma il body della lambda sarà effettivamente eseguito da dentro il metodo repeat e, da dentro il metodo repeat, la variabile 'message' non è visibile, eppure il codice è lecito e tutto funziona.
Una lambda ha tre ingredienti, parametri, body e i valori delle *variabili libere*, ovvero parametri che non fanno parte dei parametri della lambda e che non fanno parte delle variabili dichiarate nel blocco di codice della lambda.
Nell'esempio di prima, 'x' non è una variabile libera, è legata al parametro della lambda, mentre 'message' si.
Una lambda expression cattura il valore di queste variabili libere e, quando viene eseguita, il body è chiuso rispetto ad esse ed è per questo motivo che a runtime una lambda è detta *chiusura* (closure).
Ovvero prima di passare la lambda a repeat, è come se java sostituisse message direttamente con "Hello ".

Stream

Una “vista” dei dati per specificare computazioni a un più alto livello concettuale rispetto alle collezioni e gli iteratori.
Supponiamo di voler calcolare la media dei salari di una collezione di oggetti Employee.

Se decidessimo di usare gli iteratori

- dovremmo dichiarare una variabile per accumulare i risultati intermedi;
- iterare sulla sorgente dati, dove, ad ogni iterazione, aggiorniamo i risultati intermedi;
- alla fine, calcolare la media.

Se usassimo gli stream, basterebbe

- specificare la sorgente dati;
- la proprietà di interesse;
- cosa vogliamo fare con quella proprietà.

La libreria stream farà tutto il resto ottimizzando il calcolo.

Rispettano il principio ”“what, not how”, ovvero si specifica cosa si vuole fare e non il come deve essere fatto.
Stream e collezioni, superficialmente, sembrano simili, entrambi permettono di trasformare e recuperare dati ma

- uno stream non *memorizza* i dati, sono memorizzati nella collezione originale o generati su richiesta;
- uno stream non *modifica* i dati originali, ma genera un nuovo stream dove alcuni elementi dello stream precendete non sono presenti nello stream corrente;
- le operazioni sono ”*lazy*” quanto il più possibile, ovvero non vengono eseguite finché non serve il risultato (si possono avere anche stream infiniti).

Gli stream si basano sul *method chaining*, ovvero chiamano un metodo sul risultato di un altro metodo, senza memorizzare i risultati intermedi.
Il workflow tipico di uno stream consiste nel creare una “pipeline” di operazioni in 3 fasi:

- creazione dello stream;
- specifica delle operazioni intermedie;
- applicazione di un’operazione finale di riduzione per produrre un risultato oppure un’operazione di raccolta.

2.1 Creazione di uno stream

Gli stream si creano da collezioni e array o usando generatori o iteratori.
Per le collezioni abbiamo i metodi stream() e parallelStream(), per gli array o un numero arbitrario di argomenti (vararg) abbiamo il metodo statico Stream.of() e poi abbiamo i metodi Stream.generate(Supplier <T>) e Stream.iterate(T seed, UnaryOperator<T> f).
Gli ultimi due metodi possono genera stream infiniti.

2.1.1 Stream infiniti

Stream.generate(Supplier <T>) prende una lambda senza argomenti e viene chiamato solo quando viene richiesto allo stream il prossimo elemento

```
Stream<String> echos = Stream.generate(() -> "Echo");
Stream<Double> randoms = Stream.generate(Math::random);
```

Stream.iterate(T seed, UnaryOperator<T> f) dove seed è il ”seme” iniziale mentre f è una funzione che sarà applicata al valore precendete

```
Stream<BigInteger> integers = Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE))
```

2.2 Operazioni intermedie

Le operazioni intermedie di uno stream sono metodi che trasformano lo stream in un altro stream.
Tra questi metodi troviamo filter() che si occupa di filtrare uno stream, map() che si occupa di trasformare gli elementi di uno stream, limit() che limita ad un certo numero di elementi dello stream (occhio a dove lo mettiamo), skip() che salta determinati elementi dello stream, distinct() che scarta i duplicati, etc...
Con gli stream si scrive meno codice, lo si rende più leggibile ma allo stesso tempo è molto facile commettere errori, i due prossimi stream forniscono un risultato differente

```
#SI LIMITA ALLE PRIME 5 STRINGHE CHE RISPETTANO IL FILTRO
long count = words.stream()
                  .filter(w -> w.length() > 12)
                  .limit(5)
                  .count();
#SI LIMITA ALLE PRIME 5 STRINGHE
long count = words.stream()
                  .limit(5)
                  .filter(w -> w.length() > 12)
                  .count();
```

2.3 Operazione di riduzione di uno stream

Una volta applicate tutte le operazioni di trasformazione, alla fine si deve applicare un'operazione di riduzione.
Alcuni metodi di riduzione sono min() e max() che restituiscono, rispettivamente, il minimo ed il massimo di uno stream in base ad un criterio oppure il metodo reduce(...) che prende una funziona binaria e continua ad applicarla partendo dai primi due elementi.

```
List<Integer> values = ...;
Optional<Integer> sum = values.stream().reduce((x, y) -> x + y);
```

Le operazioni di raccolta/riduzione forzeranno l'esecuzione di tutte le operazioni, lazy, precedenti e dopo lo stream non potrà più essere usato.
Alcuni di queste operazioni, come reduce(...), restituiscono un oggetto di classe Optional, un'alternativa più sicura della gestione dei valori null come ad esempio findFirst(), ifPresent(), etc...

2.3.1 Optional

Un oggetto Optional<T> è un wrapper di un oggetto T oppure nessun oggetto.
L'idea di base sull'utilizzo di un oggetto Optional è quella di usare i suoi metodi che permettono di produrre un'alternativa se il valore non è presente o consumarlo.
Questo oggetto deve essere usato in modo appropriato, altrimenti si hanno gli stessi problemi che si hanno con null, ad esempio

```
Optional<T> optionalValue = ...;
optionalValue.get().someMethod();

T value = ...;
value.someMethod();
```

Questi due blocchi sono uguali, se optionalValue non contenesse alcun valore, allora avremo una NPE, stessa cosa per quest'altro esempio

```
if (optionalValue.isPresent())
    optionalValue.get().someMethod();

if (value != null)
    value.someMethod();
```

2.4 Operazione di raccolta di uno stream

Un'alternativa alla riduzione di uno stream è la raccolta.

Ci sono molti metodi tra cui il metodo collect() che prende in input oggetti che implementano l'interfaccia Collector, tra cui Collectors che fornisce metodi per creare istanze delle collezioni più comuni come ad esempio toList() o toSet(), iterator() che ritorna un iteratore o anche toArray().
Spesso si vuole una mappa che associa ad una chiave, una collezione di oggetti e qui entrano in gioco i metodi di raggruppamento o partizionamento.
Si usa il primo specificando la lambda che calcola la chiave, detta *classifier function* mentre si usa il secondo quando quest'ultima è booleana

```
#per ogni stringa del nome abbiamo la lista delle relative persone
Map<String, List<Person>> sameName = people.collect(Collectors.groupingBy(Person::getName));

#abbiamo due liste, una che contiene il nome e una che non lo contiene
Map<Boolean, List<Person>> emptyNames = people.collect(Collectors.partitioningBy(p ->
    p.getName().isEmpty()));
```

Di default groupingBy raggruppa in una List ma ci sono altre versioni che permettono di specificare un downstream collector

```
#visto prima
Map<String, List<Person>> sameName = people.collect(Collectors.groupingBy(Person::getName));

#nuovo tipo
Map<String, Long> sameName = people.collect(Collectors.groupingBy(Person::getName, Collectors.counting()));
```