Indice

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 restituisce 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.

Ad esempio, 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' 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 èun'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 (stessa cosa per i parametri della lambda), si dice *esecuzione differita* e se avesse bisogno di più righe allora si usa 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);
```

Coi 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 defailt, basta che abbiamo un singolo metodo astratto e conviene annotarla con @FunctionalInterface, così facendo il compilaotre 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 *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, è libera.
Una lambda expression cattura il valore di queste variabili libere, quando viene eseguita, il body è chiuso rispetto ad esse ed è per questo motivo che a runtime una lambda è detta *chiusura* (closure).
Prima di passare la lambda a repeat. è come se java sostituisse message con "Hello ".

stream