Programmazione II

15. Functional Programming

https://www.baeldung.com/cs/functional-programming

https://www.baeldung.com/java-functional-programming

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

Dragan Ahmetovic

Programmazione Funzionale

Scelta del paradigma di programmazione

Java è stato creato paradigmaticamente per essere un linguaggio orientato a oggetti

- La scrittura del codice nello **stile object oriented** dipende dallo sviluppatore
  - Se il programmatore non usa correttamente le classi, rimane un codice procedurale o imperativo
  - **Possibili altri stili di programmazione**
- **Programmazione dichiarativa**
  - Non si concepisce codice come **sequenza di istruzioni**
  - Invece l'**astrazione archetipica** dell'approccio è quella di **dichiarare** gli "obiettivi"
- In particolare, è comune e vantaggioso l'approccio di **Programmazione Funzionale**
  - Questo approccio **evita** l'utilizzo dei **tipi di dati mutabili** e di **stato degli oggetti**
  - Invece, vengono usati delle **funzioni simili a quelle matematiche**
  - Possibile creare **comportamenti complessi** per **composizione di funzioni**
- Possibili benefici anche all'interno di un linguaggio a oggetti come Java
  - Sintassi più snella nella creazione di alcuni metodi anonimi
  - Manipolazione di collezioni di elementi più potente, agile e veloce da scrivere

Programmazione Funzionale

Concetti e caratteristiche fondamentali

La Programmazione Funzionale è un paradigma che si basa sui seguenti concetti:

- **Trasparenza Referenziale**
  - Un'espressione è trasparente referenzialmente se **può essere sostituita dal suo valore risultante**
  - **Non può avere altri effetti** tranne quelli catturati dal valore del risultato
- Sono utilizzate `**Funzioni pure**`
  - Nella programmazione funzionale le funzioni **non hanno effetti collaterali** (no `MODIFIES`)
  - Attenzione: i metodi statici in Java comunque possono essere funzioni non pure
- Funzioni sono `**cittadini di prima classe**`
  - Le funzioni **possono essere usate come parametri, valori di ritorno o assegnate a variabili**
  - **Funzioni di ordine superiore** sono funzioni che prendono **come parametri altre funzioni**
- **Immutabilità** dei risultati
  - Non esistono `**variabili**` (si assegna una volta sola)

Programmazione Funzionale

Programmazione Funzionale in Java (da v1.8)

- In Java le funzioni **NON sono cittadini di prima classe**!
- È possibile emularle usando **Interfacce Funzionali**
  - **Qualsiasi interfaccia** con **un metodo solo** (metodi **default** non contano)
  - Esiste un'**interfaccia apposita**: `Function<I,O>` con metodo `O apply(I i)`

Esempio con Classi **Concrete**

```
1  public class Log implements Function<Double,Double>{
2    @Override
3    public Double apply(Double d) {
4      return Math.log(d);
5    }
6  }
7
8  public class Main {
9    public static void main(String[] args) {
10     Function<Double, Double> log = new Log();
11     System.out.println(log.apply(Math.PI));
12   }
13 }
```

- Utilizzo di classi concrete macchinoso, come risolvere?

Programmazione Funzionale

Composizione di Funzioni

- Possibile **combinare** due `Function`, ottenendone **una nuova**, con i metodi default:
  - `a.compose(b)` restituisce una `Function` che prima applica `b` e poi `a`
  - `a.andThen(b)`restituisce una `Function` che prima applica `a` e poi `b`

Esempio con Classi **Anonime**

```
1  public class Main {
2    public static void main(String[] args) {
3      Function<Double, Double> log = new Function<>(){
4        public Double apply(Double d) {
5          return Math.log(d);
6        }
7      };
8
9      Function<Double, Double> sqrt = new Function<>(){
10       public Double apply(Double d) {
11         return Math.sqrt(d);
12       }
13     };
14
15     Function<Double, Double> logThenSqrt = sqrt.compose(log); //prima log poi sqrt
16
17     System.out.println(logThenSqrt.apply(Math.PI));
18   }
19 }
```

Programmazione Funzionale

Lambda Expression

- Sintassi snella per scrivere `Function`
  - espressioni: `(parametro) -> parametro * 2;`
  - blocchi:
```
(parametro) -> {
    return parametro*2;
};
```
- Utile per scrivere classi anonime di **Interfacce Funzionali** (es: `Comparator`)

Esempio con **Lambda Expression**

```
1  public class Main {
2    public static void main(String[] args) {
3      Function<Double, Double> log = (p) -> Math.log(p);
4
5      Function<Integer,Boolean> isEven = (p) -> {
6        if(p%2==0)
7          return true;
8
9        return false;
10     };
11
12     System.out.println(log.apply(Math.PI));
13     System.out.println(isEven.apply(4));
14   }
15 }
```

Programmazione Funzionale

Esempio Comparator con Funzioni Anonime

```
1  public class Main{
2      public static void main(String[] args) {
3          ArrayList<Solido> l = new ArrayList<>();
4
5          ... //popolo la list
6
7          l.sort(new Comparator<Solido>(){ //passo al sort un comparator anonimo
8              @Override
9              public int compare(Solido s1, Solido s2) {
10                 return Double.compare(s1.volume(), s2.volume());
11             }
12         });
13     }
14 }
```

Esempio Comparator con Lambda Expression

```
1  public class Main{
2      public static void main(String[] args) {
3          ArrayList<Solido> l = new ArrayList<>();
4
5          ... //popolo la list
6
7          l.sort((s1, s2) -> s1.volume().compareTo(s2.volume()));
8      }
9  }
```

Programmazione Funzionale

Funzioni a più parametri

- Esiste l'interfaccia `BiFunction<I1,I2,O>`
  - Per fare **Lambda Expression** a 2 parametri `` `(p1, p2) -> espressione(p1, p2)` ``
- Possibile scrivere nuove **interfacce funzionali con più parametri**
  - Complesso doverle scrivere per tutti i casi d'uso (tutti i possibili numeri di parametri)
  - Come risolvere?

Esempio TriFunction

```
1  @FunctionalInterface
2  interface TriFunction<A,B,C,R> {
3
4      R apply(A a, B b, C c);
5
6      default <V> TriFunction<A, B, C, V> andThen(Function<? super R, ? extends V> after) {
7          Objects.requireNonNull(after);
8          return (A a, B b, C c) -> after.apply(apply(a, b, c));
9      }
10 }
```

Programmazione Funzionale

Currying o Applicazione Parziale

- Essendo **cittadini di prima classe**, una `Function` può restituire un'altra `Function`
  - Possibile creare **Funzioni a più parametri** come **applicazioni parziali** di **funzioni a parametri singoli**
  - es: `Function<Double, Function<Double, Double>» = (p1) -> (p2) -> espressione(p1,p2);`

Esempio con **Currying**

```
1  //lambda con due parametri
2  BiFunction<Double, Double, Double> weight1 = (grav, mass) -> grav * mass;
3
4  //lambda con applicazione parziale (currying)
5  Function<Double, Function<Double, Double>> weight2 = (grav) -> (mass) -> grav * mass;
6
7  //applicazione primo parametro
8  Function<Double, Double> weightOnEarth = weight2.apply(9.81);
9
10 //applicazione secondo parametro
11 System.out.println("My weight on Earth: " + weightOnEarth.apply(60.0));
12
13 //lambda col blocco di codice
14 Function<Double, Double> weightOnMars = (mass) -> {
15   final double gravMars = 3.75;
16   return weight2.apply(gravMars).apply(mass);
17 };
18
19 System.out.println("My weight on Mars: " + weightOnMars.apply(60.0));
```

Programmazione Funzionale

Interfaccia `Stream`

- L'interfaccia `Stream` (Java 1.8) facilita l'uso del paradigma funzionale su `Collection`
  - `Collection` ha il metodo `stream()` che genera uno `Stream` dei suoi elementi (stile `Iterator`)
  - Utile per parallelizzare operazioni su collection o fare ragionamenti di insieme

Esempio uso degli `Stream`

```
1  String[] arr = new String[]{"ananas", "banana", "ciliegia", "ananas"};
2  Stream<String> sAr = Arrays.stream(arr); //Stream da array
3
4  ArrayList<Integer> list = new ArrayList<>();
5  list.add(1);
6  list.add(2);
7  list.add(4);
8  Stream<Integer> sLi = list.stream(); //Stream da arraylist
9
10 Stream<Double> sOf = Stream.of(2.2, 3.3, 4.4); //Stream da literal
11
12 System.out.println(sAr.distinct().count()); //trova elementi distinti e li conta
13
14 sAr = Arrays.stream(arr); //Stream non riutilizzabili (come Iterator), serve rifarli
15
16 System.out.println(sAr.allMatch(e -> e.contains("a"))); //true se tutte contengono "a"
17
18 sOf.forEach(e -> System.out.println(e)); //per ogni elemento applica funzione
```

Programmazione II

15. Functional Programming

https://www.baeldung.com/cs/functional-programming

https://www.baeldung.com/java-functional-programming

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

Dragan Ahmetovic