Programmazione funzionale in Java 8

Prof. Annalisa Appice

Metodi Avanzati di Programmazione Corso A

Problema

- Passaggio di una funzione ad un **metodo come argomento**: *codice come dati*
- Data una collezione di Persone : *List<Persona>*
- Trovare i membri che soddisfano determinati criteri
  - I criteri della ricerca potrebbero cambiare ripetutamente e frequentemente
  - Le azioni da applicare ai membri selezionati potrebbero essere soggetti a cambiamenti.

Class Persona

```
public class Persona {

private String nome;
private int eta;
private  String indEmail;
public Persona(String nome, int eta, String indEmail) {
this.nome=nome;this.eta=eta;this.indEmail=indEmail;
}
public int getEta() { return eta; }
public String getNome(){ return nome; }
public void printPersona() { System.out.println(nome+ " "+ eta+ " "+ indEmail); }

}
```

Ricerca per caratteristica

- Approccio: creare diversi metodi, ognuno per cercare membri con una data caratteristica (per esempio nome o eta)

```
import java.util.LinkedList;
import java.util.List;
class Comunita {
private List<Persona> lista=new LinkedList<>();
public void printPersoneConPiuDi(int eta) {
for (Persona p : list) {
        if (p.getEta() >= eta) {p.printPersona(); }
        }
}
}
```

Limiti

- Approccio inutilmente *restrittivo*:
  - cosa succede se si vogliono stampare i membri con meno di una certa età?
  - Potremmo generalizzare usando un intervallo per la ricerca

```
public void printPersoneConEtaCompresaFra(int inf, int sup) {
        for (Persona p : lista) {
                if (inf <= p.getEta() && p.getEta() < sup) {
                        p.printPersona();
                }
        }
}
```

Limiti

- E se si volesse stampare i membri con il nome che inizia per un dato carattere o una combinazione nome-intervallo d'età?
- E se si volesse cambiare Persona per aggiungere attributi come parentela o locazione geografica?
- Un metodo per ogni possibile query di ricerca renderebbe il codice fragile
- Meglio separare il codice che specificare i criteri di ricerca in una *classe separata*

Classe per la formulazione di Criteri

```
public void printPersone(TestaPersonaI pred) {
        for (Persona p : lista) {
                if (pred.test(p)) {
                        p.printPersona();
                }
        }
}
```

- Definire la Interfaccia TestaPersonaI con la operazione

  boolean test (Persona p)

- Invocare printPersona passandogli una istanza di una classe che implementi TestaPersonaI

Classe anonima come parametro effettivo

- Comunita cs =new Comunita()
- …

```
cs. printPersone(new TestaPersonaI() {
        public boolean test(Persona p) {
                return p.getEta() >= 18 & p.getEta() <= 25;
        }
}
);
```

Classe anonima: osservazioni

- Questa soluzione richiede meno codice (nessuna classe aggiuntiva) ma la sintassi si appesantisce considerando che TestaPersona contiene solo un metodo
- In alternativa si possono usare lambda-espressioni tramite una interfaccia funzionale

Lambda espressione

- In matematica e informatica in generale, un’espressione lambda è una funzione.
- In Java, un’espressione lambda fornisce un modo per creare una funzione anonima, introducendo di fatto un nuovo tipo Java: il tipo *funzione anonima* che può quindi essere passato come argomento o restituito in uscita nei metodi

Lambda espressione : sintassi

- (Lista degli argomenti) -> Espressione

oppure

- (Lista degli argomenti)->{ istruzioni; }

Interfaccia funzionale

- Un'**interfaccia funzionale** contiene solo un metodo astratto
- Dato che è unico, si può *omettere* il suo nome implementandolo; invece d'una classe anonima: lambda espressione

```
printPersone(
        lista,
        (Persona p) ->  p.getEta() >= 18 && p.getEta() <= 25
);
```

Interfaccia funzionale standard

- Si possono usare diverse **interfacce funzionali standard** di java.util.function
- Per esempio,

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

Definizione del metodo di ricerca

```
public void printPersoneConPredicato(Predicate<Persona> pred) {
for (Persona p : lista) {
        if (pred.test(p)) {
                p.printPersona();
        }
}
}
```

Lambda esperessioni in JAVA

```
import java.util.LinkedList;
import java.util.List;
import java.util.function.Predicate;
class Comunita {
private List<Persona> lista=new LinkedList<>();
public void printPersoneConPiuDi(int eta) {
for (Persona p : lista) { if (p.getEta() >= eta)  p.printPersona(); }
}
private void add(Persona p){lista.add(p);}
public void printPersoneConPredicato( Predicate<Persona> pred) {
for (Persona p : lista) { if (pred.test(p)) p.printPersona(); }
}
public static void main(String args[]){
Comunita c=new Comunita();
c.add(new Persona("AA", 22, "a@b")); c.add(new Persona("AB", 12, "a@a"));
c.add(new Persona("CA", 21, "c@b")); c.add(new Persona("AA", 29, "cc@b"));
c.printPersoneConPredicato(  p -> p.getEta() >= 18 && p.getEta() <= 25 );
// Equivalente alla istanzaiazione della classe anonima che implementa l'interfaccia
Predicate<Persona>
/*c.printPersoneConPredicato(new Predicate<Persona>() {
public boolean test(Persona p) { return p.getEta() >= 18 && p.getEta() <= 25;} } );
*/
}
}
```

Lambda espressioni in JAVA

- Si possono anche definire delle azioni alternative da compiere
- L'interfaccia Consumer<T> contiene il metodo **void** accept(T t), che ha sintassi compatibile con printPersona

```
public void elaboraPersone(Predicate<Persona> pred, Consumer<Persona> blocco) {
for (Persona p : lista) {
        if (pred.test(p))
                blocco.accept(p); // p.printPersona();
}
}
```

Lambda espressioni in JAVA

```
public static void main(String args[]){
Comunita c=new Comunita();
c.add(new Persona("AA", 22, "a@b"));
c.add(new Persona("AB", 12, "a@a"));
c.add(new Persona("CA", 21, "c@b"));
c.add(new Persona("AA", 29, "cc@b"));

c.elaboraPersone(p -> p.getEta() >= 18&& p.getEta() <= 25,
                    p -> p.printPersona()
                );
}
```

Lambda espressioni in JAVA

- Se l’azione da compiere deve restituire un valore?
- Si consideri l’interfaccia Function<T,R> contiene l’operazione:

  R apply(T t)

Lambda espressioni in Java

- Cercare il nome con mapper e compiere una azione (stamparlo in minuscolo) su esso specificato da blocco

```
public  void elaboraPersoneConFunction(Predicate<Persona> pred, Function<Persona, String> mapper, Consumer<String> blocco) {
		for (Persona p : lista) {
			if (pred.test(p)) {
				String dati = mapper.apply(p);
				blocco.accept(dati);
			}
		}
}
```

```
public  void elaboraPersoneConFunction(Predicate<Persona> pred, Function<Persona, String> mapper,
Consumer<String> blocco) {
        for (Persona p : lista) {
                if (pred.test(p)) {
                        String dati = mapper.apply(p);
                        blocco.accept(dati);
                }
        }
}
public static void main(String args[]){
        Comunita c=new Comunita();
        c.add(new Persona("AA", 22, "a@b"));
        c.add(new Persona("AB", 12, "a@a"));
        c.add(new Persona("CA", 21, "c@b"));
        c.add(new Persona("AA", 29, "cc@b"));
        c.elaboraPersoneConFunction( p -> p.getEta() >= 18 && p.getEta() <= 25,
        p -> p.getNome(),
        nomex -> System.out.println(nomex.toLowerCase())
        );
}
```

Operazioni aggregate

```
public static void main(String args[]){
Comunita c=new Comunita();
c.add(new Persona("AA", 22, "a@b"));
c.add(new Persona("AB", 12, "a@a"));
c.add(new Persona("CA", 21, "c@b"));
c.add(new Persona("AA", 29, "cc@b"));

c.lista
        .stream() // ottiene lo stream
        .filter(p ->p.getEta() >= 18 && p.getEta() <= 25) // filtra in base a un predicato
        .map(p->p.getNome()) // mappa un oggetto su un valore specifico
        .forEach(nomeX->System.out.println(nomeX)); // esegue la azione su ogni oggetto mappato
}
```

O equivalentemente

```
public  void elaboraPersoneConFunction(Predicate<Persona> pred,Function<Persona, String> mapper,Consumer<String> blocco) {
lista
.stream()
.filter(pred)
.map (mapper)
.forEach(blocco);
}
public static void main(String args[]){
Comunita c=new Comunita();
c.add(new Persona("AA", 22, "a@b"));
c.add(new Persona("AB", 12, "a@a"));
c.add(new Persona("CA", 21, "c@b"));
c.add(new Persona("AA", 29, "cc@b"));
c.elaboraPersoneConFunction(p ->p.getEta() >= 18 && p.getEta() <= 25, p->p.getNome(),nomeX->System.out.println(nomeX));
}
```

Pipeline e stream

- Una **pipeline** è una sequenza di operazioni aggregate (per esempio .filter, .map, .forEach)
- *sorgente della pipeline:* una collezione, array, funzione generatrice o canale I/O (nell'esempio la lista
- *operazioni intermedie della pipleine:* (zero o più) producono uno **stream** - una sequenza di elementi che serve a veicolare valori da una sorgente attraverso una pipeline (e non a conservarli come una collezione), nell'esempio:
  - .stream() crea lo stream dalla lista
  - .filter…) restituisce un nuovo stream costituito dagli elementi di età compresa tra 18 e 25 (quelli che soddisfano il predicatolambda espressione)
- *operazione terminale:* produce risultato finale (che non è uno stream, ma è un tipo primitivo, una collezione, nessun valore - .forEach(…))

Pipeline e stream

- Calcolo età media dei componenti con nome che inizia per «A»

```
double mediaEta(){
double media= lista
.stream()
.filter(p->p.getNome().toUpperCase().charAt(0)=='A')
.mapToInt(p->p.getEta()) // prende in input un mapper e restituisce uno stream di interi
.average() //calcola la media sullo stream di input – lo stream di interi, restituisce una istanza di OptionalDouble
.getAsDouble(); // se lo stream fosse vuoto sollerverebbe una eccezione NoSuchElementException
return media;
}
public static void main(String args[]){
Comunita c=new Comunita();
c.add(new Persona("AA", 22, "a@b")); c.add(new Persona("AB", 12, "a@a"));
c.add(new Persona("CA", 21, "c@b")); c.add(new Persona("AA", 29, "cc@b"));
System.out.println(c.mediaEta());
}
```

Operazioni terminali o di riduzione

- JAVA prevede diverse operazioni terminali o di riduzione (average, sum, min, max e count) che restituiscono un solo valore che combina gli elementi dello stream o anche intere collezioni
- Molte di esse svolgono un compito speci1co
- In aggiutna alle operazioni menzionate ci sono operazioni di riduzione general-purpose, come i metodi
  - Stream.reduce
  - Stream.collect

Stream.reduce

```
int sommaEta(){
        int somma= lista
        .stream()
        .filter(p->p.getNome().toUpperCase().charAt(0)=='A')
        .map(p->p.getEta())
        .reduce( 0, (a,b)->a+b);
// in alternativa
//.mapToInt(p->p.getEta())
//.sum();
        return somma;
}
```

Stream.reduce

- Richiede due argomenti
  - elemento identità rappresenta il valore iniziale della riduzione o il risultato di default in caso di stream vuoto (nell'esempio della somma, la identità è 0)
  - accumulatore che è la funzione binaria che richiede il risultato parziale corrente e il prossimo elemento dello stream e restituisce un nuovo risultato parziale (nell'esempio si usa una lambda espressione per la somma di interi che restituisce un intero: (a, b) -> a + b)

Stream.reduce

- La funzione accumulatore restituisce un nuovo valore ogni volta che elabora un nuovo elemento dello stream
- Riduce lo stream a un elemento complesso
  - Se l'elemento complesso fosse a sua volta una collezione , le prestazioni del reduce sarebbero compromesse
    - Reduce dovrebbe aggiungere elementi a una nuova collezione ogni volta che un elemento è aggiunto allo stream
    - Usare , in alternativa, Stream.collect per aggiornare una collezione esistente

Stream.collect

Calcolare la media – oggetto complesso che collezione numero di valori nello stream e somma dei valori

- Definire la classe Media che implementa l'interfaccia IntConsumer (che include l'operazione void accept(int x))

```
class Media implements IntConsumer
{
private int totale = 0; private int contatore = 0;
public double average() { return contatore > 0 ? ((double) totale)/contatore : 0; }
public void accept(int i) { totale += i; contatore++; }
public void combine(Media altro) {
totale += altro.totale; contatore += altro.contatore;
}
}
```

Stream.collect

```
double mediaEta(){
Media media = lista
.stream()
.filter(p->p.getNome().toUpperCase().charAt(0)=='A')
.map(p->p.getEta())
.collect(Media::new, Media::accept, Media::combine);
return media.average();
}
```

Stream.collect

- Stream.collect ha tre argomenti
- fornitore funzione factory: costruisce nuove istanze del contenitore del risultato (per esempio, nuova istanza di Media)
- accumulatore: funzione che incorpora un elemento dello stream in un contenitore del risultato
- combinatore: funzione che fonde il contenuto di due contenitori di risultato

Computazione parallela

- Suddivisione del problema in sotto-problemi e soluzione di tutti questi in parallelo e combinazione finale dei risultati parziali
- Uso del framework fork-begin
  - La esecuzione in parallelo non è in automatico la esecuzione più veloce (dipende dalla quantità di dati e processori core)

Computazione in parallelo

- Gli stream possono essere eseguiti in serie o in parallelo
  - In parallelo, JRE partiziona lo stream in più sotto-stream
    - le operazioni aggregate iterano su di essi elaborandoli in parallelo e combinando alla fine i risultati
- La creazione di uno stream è seriale se non diversamente specificato
- Per creare uno stream in parallel si può chiamare Collection.parallelStream, o in alternativa, si usa BaseStream.parallel

Computazione in parallelo

- Calcolare l'età media delle persone di età compresa tra 18 e 25 in parallelo

```
double mediaEta(){
Media media = lista
.parallelStream()
.filter(p->p.getNome().toUpperCase().charAt(0)=='A')
.map(p->p.getEta())
.collect(Media::new, Media::accept, Media::combine);
return media.average();
}
```

Riduzione concorrente

- Raggruppamento dei membri per Nome chiamando una collect che riduce lista in una Map

```
Map<String, List<Persona>> perNome(){
        Map<String, List<Persona>> group=
        lista.stream().collect(Collectors.groupingBy(Persona::getNome));
        return group;
}
Map<String, List<Persona>> perNomeConcurrent(){
        Map<String, List<Persona>> group=
        lista.parallelStream().collect(Collectors.groupingByConcurrent(Persona::getNome));
        return group;
}
```

Bibliografia

- David J. Eck: Introduction to Programming Using Java,7th ed. (v.7.0.2) 2015/16
- J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley: The Java® Language Speci1cation – Java SE 9 Edition. 2017 http://docs.oracle.com/javase/specs/