Lambda Expressions

Espressioni lambda

- Costituiscono una delle principali novità di Java 8
- Permettono di descrivere un metodo nel punto in viene utilizzato
- Hanno un tipo definito da un'interfaccia funzionale
  - essenzialmente un'interfaccia con un solo metodo astratto

Problema

- Vogliamo realizzare un'applicazione tipo per un social network

- In particolare, vogliamo consentire ad un amministratore di eseguire ogni tipo di azione (ad es. inviare una email) nei confronti di tutti i membri che soddisfano determinati criteri

Scheda applicazione

- attore: amministratore
- pre-condizione: login effettuata con successo
- post-condizione: azione eseguita solo su membri che soddisfano il criterio
- passi esecuzione:
  1. amministratore specifica criterio
  2. amministratore specifica azione da eseguire
  3. amministratore seleziona pulsante **Submit**
  4. sistema trova tutti i membri che soddisfano criterio
  5. sistema esegue azione su membri individuati

Alcuni dettagli codice

astrazione per membri:

```
public class Person {
    public enum Sex { MALE, FEMALE };
    private String name;
    private LocalDate birthday;
    private Sex gender;
    private String emailAddress;

    public int getAge() { // ... }
    public String getPerson() { // ... }
}
```

Si assuma che gli oggetti Person sono mantenuti nel sistema con un'ArrayList di Person

Soluzione 1: metodo per selezione

Criterio selezione: membri in base ad un'età minima

```
public static String getPersonsOlderThan(ArrayList<Person> roster, int age) {
    String selection="";
    for (Person p : roster) {
        if (p.getAge() >= age) { selection+=(p.getPerson()+'\n'; }
    }
    return selection;
}
```

Problema: L'applicazione è fragile

se vogliamo selezionare i membri "più giovani di.."?

Soluzione 2: estensione criterio

```
public static String getPersonsWithinRange(ArrayList<Person> roster,
                                                    int low, int high) {
    String selection="";
    for (Person p : roster) {
        if (low <= p.getAge() &&  p.getAge() <= high) {
                                selection+=(p.getPerson()+'\n'; }
    }
    return selection;
}
```

Problema: Criterio solo legato ad età, criteri più generali?

Soluzione 3: uso polimorfismo

Definiamo criterio attraverso una Java interface:

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

public static String getPersons(ArrayList<Person> roster,
                                CheckPerson tester) {
    String selection="";
    for (Person p : roster) {
        if (tester.test(p)) { selection+= p.getPerson()+'\n'; }
    }
    return selection;
}
```

Bisogna implementare ogni criterio in una classe che implementa CheckPerson

Implementazione CheckPerson

```
public class CheckPersonEligibleForSelectiveService
                    implements CheckPerson {
          public boolean test(Person p) {
                    return p.gender == Person.Sex.FEMALE
                              && p.getAge() >= 18 && p.getAge() <= 25;
          }
}
```

Può essere implementata anche come classe interna se astrazione non serve altrove

Soluzione 4: classe anonima

- Se il criterio serve solo per l'invocazione del metodo possiamo usare una classe anonima (senza nome) per implementare la Java interface CheckPerson

- Il metodo getPersons viene invocato in questo modo:

```
getPersons( roster, new CheckPerson() {
                        public boolean test(Person p){
                            return p.getGender() == Person.Sex.FEMALE
                                && p.getAge() >= 18
                                && p.getAge() <= 25; } }
                );
```

1. La definizione della classe viene fornita al momento dell'invocazione del costruttore
2. E' una classe interna: stesse regole delle classi interne

Soluzione 5: espressione lambda (Java 8)

- CheckPerson è un'interfaccia funzionale
  - un solo metodo astratto (non statico)
- Per dare un'implementazione di un'interfaccia funzionale possiamo usare una espressione lambda invece di un'espressione contenente una classe anonima:

```
getPersons( roster,
                    p -> p.getGender() == Person.Sex.FEMALE
                        && p.getAge() >= 18
                        && p.getAge() <= 25;
            );
```

Sintassi di una espressione lambda

lista di parametri -> istruzione

- lista di parametri: lista di identificatori separati da virgole racchiusa tra parentesi tonde
  - es. due parametri: (x, y)
  - le parentesi possono essere omesse se parametro è singolo
  - se non c'è alcun parametro si usa lista vuota ( )
- istruzione può essere istruzione semplice, istruzione composta, o blocco di istruzioni

Altro esempio

```
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 class CalculatorTester{
   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));
   }
}
```

Regole di scoping (visibilità variabili)

- Come per le classi interne e anonime
  - variabili dichiarate nell'ambiente esterno sono visibili nel corpo dell'espressione lambda
  - le variabili locali dell'ambiente esterno utilizzate nell'espressione lambda devono essere effettivamente final

    (dichiarate final oppure il loro valore effettivamente non viene modificato)
- Differentemente da classi interne e anonime
  - non introduce un nuovo ambiente di scoping

  (non si può dichiarare una variabile con un nome già definito nel metodo in cui viene scritta)

Esempio

```
public class LambdaScope {
    public int x = 0;
    public class FirstLevel {
        public int x = 1;
        String methodInFirstLevel(int x) {
            PrintFormatter<Integer> myF = (y) ->
                { String s = "x = " + x+ "\n";        s+="y = " + y+ "\n";
                  s+="this.x = " + this.x +"\n";
                  s+="LambdaScope.this.x = " + LambdaScope.this.x + "\n";
                  return s;
                };
            return myF.format(x);
        }
    }
}
```

```
public interface PrintFormatter<T>{
        String format(T t);
}
```

```
public class LambdaScopeTester {
   public static void main(String[] args) {
        LambdaScope st = new LambdaScope();
        LambdaScope.FirstLevel fl = st.new FirstLevel();
        System.out.println(fl.methodInFirstLevel(23));
    }
}
```

Esempio

```
public class LambdaScope {
    public int x = 0;
    public class FirstLevel {
        public int x = 1;
        String methodInFirstLevel(int x) {
            PrintFormatter<Integer> myF = (y) ->
                { String s = "x = " + x+
                  s+="this.x =
                  s+="LambdaSco
                  return s;
                };
            x++;
            return myF.format(x);
        }
    }
}
```

```
public interface PrintFormatter<T>{
        String format(T t);
}
```

ERRORE la variabile locale x è modificata

```
er {
    void main(String[] args) {
    LambdaScope st = new LambdaScope();
    LambdaScope.FirstLevel fl = st.new FirstLevel();
    System.out.println(fl.methodInFirstLevel(23));
    }
}
```

Esempio

```
public class LambdaScope {
    public int x = 0;
    public class FirstLevel {
        public int x = 1;
        String methodInFirstLevel(int x) {
            PrintFormatter<Integer> myF = (x) ->
                { String s = "x = " + x+ "
                  s+="this.x = " + this.x +
                  s+="Lambda
                  return s;
                };
            return myF.format(x);
        }
    }
}
```

```
public interface PrintFormatter<T>{
        String format(T t);
}
```

ERRORE x è il nome di una variabile locale di methodInFirstLevel

```
                                  args) {
                    new LambdaScope();
        LambdaScope.FirstLevel fl = st.new FirstLevel();
        System.out.println(fl.methodInFirstLevel(23));
    }
}
```

Tipi e espressioni lambda

- Tipo di una lambda espressione è il tipo dell'interfaccia che implementa (target type)

- Il compilatore Java determina il tipo di una espressione lambda dal contesto in cui viene utilizzata
  - possiamo utilizzare una espressione lambda dovunque questo è possibile
  - ad es. assegnamenti, dichiarazione di variabili, istruzioni di return, argomenti di metodi, etc.

Tipi e espressioni lambda

- Considera l'espressione lambda

```
p -> p.getGender() == Person.Sex.FEMALE
                   && p.getAge() >= 18
                   && p.getAge() <= 25;
```

- In getPersons(List<Person> roster, CheckPerson tester) è di tipo CheckPerson

- Ma potrebbe anche essere di tipo Predicate<Person> dove:

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

Target type e metodi overloaded

- Consideriamo le interfacce funzionali

```
public interface Runnable {  void run();  }
public interface Callable<V> {  V call();  }
```

- Supponiamo che un metodo invoke è overloaded:

```
void invoke(Runnable r) {   r.run();   }
<T> T invoke(Callable<T> c) {   return c.call();  }
```

- Qual'è il metodo invocato nell'istruzione seguente?

```
String s = invoke( ( ) -> "done");
```

  - il metodo invoke(Callable<T> c) in quanto restituisce un valore
  - in questo caso ( ) -> "done" è di tipo Callable<V>

Target type e metodi overloaded

- Se ci fosse ambiguità sul tipo di return il target type non potrebbe essere determinato
- Ad es:

```
int  invoke(Measurable r) {   return r.getMeasure();   }
<T> T invoke(Callable<T> c) {   return c.call();  }
```

- Con

```
int x = invoke( ( ) -> 3);
```

- In questo caso il compilatore dà errore

Serializzazione di espressioni lambda

- Un’espressione lambda è serializzabile se
  - gli argomenti sono serializzabili
  - il suo target type è serializzabile
- La serializzazione delle lambda espressioni è fortemente sconsigliata