Classi interne, espressioni lambda

Classi interne

- Classi definite all'interno di altre classi
  - fuori dai metodi: visibile in tutti i metodi (anche fuori dalla classe, dipende da specificatore d'accesso)
  - all'interno di un metodo: visibile solo nel metodo

- I metodi della classe interna
  - hanno accesso alle variabili e ai metodi a cui possono accedere i metodi della classe in cui sono definite (accesso all'ambiente in cui è definita)
    - se definite in un metodo statico accedono solo alle variabili statiche non alle variabili di istanza
  - possono accedere a variabili locali solo se sono effettivamente final, cioè:
    1. state dichiarate final
    2. oppure il loro valore non viene modificato

    (seconda opzione introdotta a partire da Java 8)

Esempio

```
import java.awt.Rectangle;

public class DataSetTest {
   public static void main(String[] args){
         //classe interna
     class RectangleMeasurer
                              implements
     Measurer<Rectangle>{
     public double measure(Rectangle aRectangle){
              double area = aRectangle.getWidth()
                          * aRectangle.getHeight();
           return area;
          }
     }
```

```
     Measurer m = new RectangleMeasurer();
     DataSetMeasurer<Rectangle> data = new
       DataSetMeasurer<Rectangle>(m);
       data.add(new Rectangle(5, 10, 20, 30));
       data.add(new Rectangle(10, 20, 30, 40));
       data.add(new Rectangle(20, 30, 5, 10));

       System.out.println("La media delle aree è = "
                + data.getAverage());
       Rectangle max =
                       data.getMaximum();

      System.out.println("L'area maggiore è = " +
                                        m.measure(max));
   }
}
```

Eventi di temporizzazione

- La classe Timer in javax.swing genera una sequenza di eventi ad intervalli di tempo prefissati
  - Utile per la programmazione di animazioni
- Un evento di temporizzazione deve essere notificato ad un ricevitore di eventi
- Per creare un ricevitore bisogna definire una classe che implementa l'interfaccia ActionListener in java.awt.event

Esempio

```
class MioRicevitore implements ActionListener
   {
     public void actionPerformed(ActionEvent event)
     {
   //azione da eseguire ad ogni evento di
   //        temporizzazione
     }
}

ActionListener listener = new MioRicevitore();
Timer t = new Timer(interval, listener);
t.start();
```

Eventi di temporizzazione

- Un temporizzatore invoca il metodo actionPerformed dell'oggetto listener ad intervalli regolari
- Il parametro interval indica il lasso di tempo tra due eventi in millisecondi
- Vediamo un programma che conta all'indietro fino a zero con un secondo di ritardo tra un valore e l'altro

Programma CountDown

```
import java.awt.event.ActionEvent;     import java.awt.event.ActionListener;
import javax.swing.JOptionPane;       import javax.swing.Timer;

public class TimerTest{   //   Questo programma collauda la classe Timer
   public static void main(String[] args){
      class CountDown implements ActionListener {
         public CountDown(int initialCount){ count = initialCount;}
         public void actionPerformed(ActionEvent event){
                if (count >= 0) System.out.println(count);
                count--;
          }
          private int count;
      }
      CountDown listener = new CountDown(10);
      Timer t = new Timer(1000, listener);      t.start();

      JOptionPane.showMessageDialog(null, "Quit?");     System.exit(0);
   }
}
```

Eventi di temporizzazione

- Implementare un ricevitore come classe non interna
  - Il ricevitore di eventi può aver bisogno di modificare lo stato di oggetti nel metodo actionPerformed
  - Occorre memorizzare questi oggetti nelle variabili di istanza della classe che implementa ActionListener
- In genere preferibile definire ricevitore come classi interne
  - Può accedere alle variabili dell'ambiente in cui è implementata la classe
  - Un ricevitore ha solitamente un uso locale (funzionalità specifiche dell'applicazione in cui viene usato)

Esempio

```
import java.awt.event.ActionEvent;     // import come esempio precedente

/** Uso di un temporizzatore per aggiungere interessi ad un conto bancario una volta al
    secondo */

public class TimerTest {
  public static void main(String[] args){

    final BankAccount account =  new BankAccount(1000);

    class InterestAdder implements ActionListener{
       public void actionPerformed(ActionEvent event){
         double interest = account.getBalance() * RATE / 100;
         account.deposit(interest);
         System.out.println("Balance = “ + account.getBalance());
       }
    }

    InterestAdder listener = new InterestAdder();
    ……..……  // uso Timer e finestra JOptionPane come esempio precedente
  }
  private static final double RATE = 5;
}
```

account non è modificato, possiamo non usare final (Java 8)

```
import java.awt.event.ActionEvent;     // import come esempio precedente

/** Uso di un temporizzatore per aggiungere interessi ad un conto bancario una volta al
    secondo */

public class TimerTest {
  public static void main(String[] args){
    BankAccount account =  new BankAccount(1000);
    class InterestAdder implements ActionListener{
      public void actionPerformed(ActionEvent event){
        double interest = account.getBalance() * RATE / 100;
        account.deposit(interest);
        System.out.println("Balance = " + account.getBalance());
      }
    }

    InterestAdder listener = new InterestAdder();
    ……..……  // uso Timer e finestra JOptionPane come esempio precedente
  }
  private static final double RATE = 5;
}
```

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 alcune operazioni (ad es. inviare una email) nei confronti di tutti i membri che soddisfano determinati criteri

- Ci concentriamo sui criteri di selezione dei membri

Alcuni dettagli del 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

Metodo per selezione: soluzione 1

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 desiderato 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 (stesso nome interfaccia)
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
  - compilazione non genera bytecode in file separato
  - 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));
    }
}
```