Marco Faella

Classi interne, locali ed anonime

Lezione n. 6

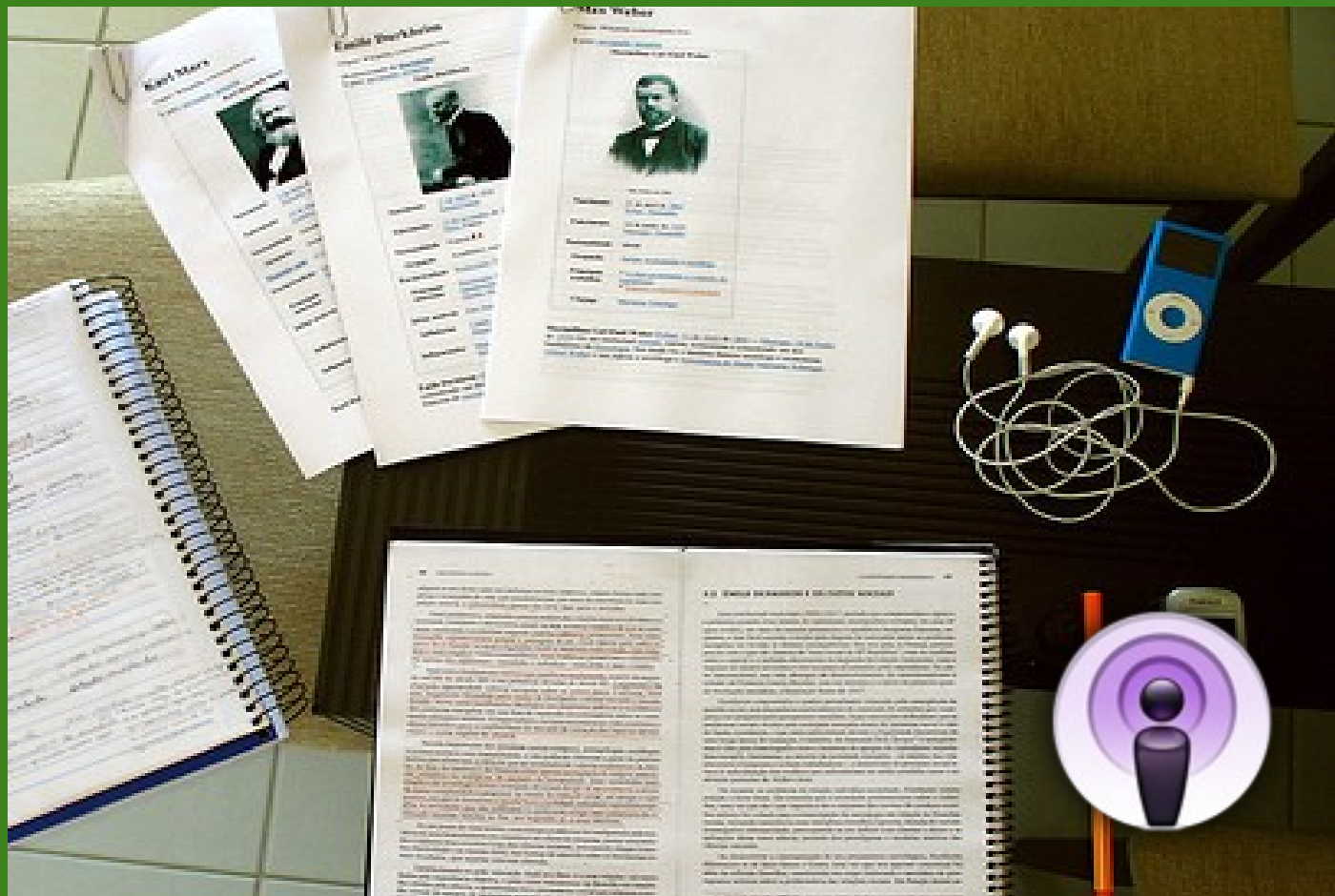
Parole chiave:
Java

Corso di Laurea:
Informatica

Insegnamento:
Linguaggi di Programmazione II

Email Docente:
faella.didattica@gmail.com

A.A. 2009-2010



- Java permette di definire una classe (o interfaccia) all'interno di un'altra
- Queste classi vengono chiamate “interne” o “annidate”
  - in inglese, il termine usato è *nested*
  - In particolare, le classi interne non statiche sono chiamate *inner*

- Tale meccanismo arricchisce le possibilità di relazioni tra classi, introducendo in particolare **nuove regole di visibilità**

- Se la definizione di una classe si trova all'interno di un metodo, tale classe viene chiamata ***locale***

- Infine, una classe locale può anche essere ***anonima***, quando il suo nome non sarebbe rilevante e/o utile

Proprietà delle classi interne

Le classi interne (non statiche) godono delle seguenti proprietà distintive:

1) **Privilegi di visibilità** rispetto alla classe contenitrice e alle altre classi in essa contenute
   - permettono una stretta collaborazione tra queste classi

2) **Restrizioni di visibilità** rispetto alle classi esterne a quella contenitrice
   - permettono di nascondere la classe all'esterno (incapsulamento)

3) Un **riferimento implicito** ad un oggetto della classe contenitrice
   - ogni oggetto della classe interna "conosce" l'oggetto della classe contenitrice che l'ha creato

Restrizioni di visibilità delle classi interne

- Oltre a campi e metodi, una classe può contenere altre classi o interfacce, dette "interne"
- Una classe che non sia interna viene chiamata "top-level"

- A differenza delle classi top-level, le classi interne possono avere **tutte le quattro visibilità** ammesse dal linguaggio
- La visibilità di una classe interna X stabilisce quali classi possono utilizzarla (cioè, istanziarla, estenderla, dichiarare riferimenti o parametri di tipo X, etc.)

- Consideriamo il seguente esempio:

```
public class A {
   private class B {
       ...
   }
   class C {
       ...
   }
}
```

- In quest'esempio, la classe B **non è visibile al di fuori di A**, mentre la classe C è visibile a tutte le classi che si trovano nello stesso pacchetto di A

Privilegi di visibilità delle classi interne

```
public class A {
   private class B {
      ...
   }
   class C {
      ...
   }
}
```

- Dall'esterno di A, i nomi completi delle classi B e C sono A.B e A.C, rispettivamente
- La visibilità di una classe interna non ha alcun effetto sul codice che si trova all'interno della classe che la contiene
- Ad esempio, la classe B dell'esempio è visibile a tutto il codice contenuto in A, compreso il codice contenuto in altre classi interne ad A, come ad esempio C
- Lo stesso discorso si applica per i campi e i metodi di una classe interna
  - i loro attributi di visibilità hanno effetto solo sul codice *esterno* alla classe contenitrice
- In altre parole, **tra classi contenute nella stessa classe non vige alcuna restrizione di visibilità**

- Ciascun oggetto di una classe interna (non statica, come spiegato dopo) possiede un riferimento implicito ad un oggetto della classe contenitrice
- Tale riferimento viene inizializzato automaticamente al momento della creazione dell'oggetto
- Tale riferimento non può essere modificato

- Supponiamo che B sia una classe interna di A
- Se viene creato un oggetto di tipo B in un *contesto non statico* della classe A, il riferimento implicito verrà inizializzato con il valore corrente di this
- In tutti gli altri casi, è necessario utilizzare una **nuova forma dell'operatore "new"**, ovvero:

<riferimento ad oggetto di tipo A>.new B(...)

- All'interno della classe B, la sintassi per denotare questo riferimento implicito è A.this
- L'uso di A.this è facoltativo, come quello di this
  - cioè, si può accedere ad un campo o metodo "f" della classe A sia con "A.f" che con "f"

- Nota: una classe interna non statica non può avere membri (campi o metodi) statici

Classi interne statiche

- Le classi interne possono essere statiche o meno
- Una classe interna dichiarata nello scope di classe (cioè al di fuori di metodi e inizializzatori) è statica se preceduta dal modificatore “static”
- Una classe interna dichiarata all'interno di un metodo (quindi, locale) o di un inizializzatore eredita il proprio essere statica o meno dal metodo in cui è contenuta o dal campo che si sta inizializzando
- Le classi interne statiche non possiedono il riferimento implicito alla classe contenitrice

Proprietà delle classi interne

- Riassumendo, le classi interne non statiche godono delle seguenti proprietà distintive:

  1) **Privilegi di visibilità** rispetto alla classe contenitrice e alle altre classi in essa contenute
     - permettono una stretta collaborazione tra queste classi

  2) **Restrizioni di visibilità** rispetto alle classi esterne a quella contenitrice
     - permettono di nascondere la classe all'esterno (incapsulamento)

  3) Un **riferimento implicito** ad un oggetto della classe contenitrice
     - ogni oggetto della classe interna "conosce" l'oggetto della classe contenitrice che l'ha creato

- Le classi interne *statiche* godono solo delle **prime due** proprietà

Nota: alcuni testi si riferiscono a tutte le classi interne come "annidate" e riservano il termine "interne" solo alle classi interne non statiche

Privilegi di visibilità delle classi interne

Il seguente esempio mostra come le classi interne (anche statiche) abbiano pieno accesso ai membri privati della classe contenitrice

```
public class InternalVisibility
{
   private int n = 42;

   public static class A {
      public void foo(InternalVisibility guest) {
         // This is legal
         guest.n = 0;
     }
   }

   public static class B extends InternalVisibility {
     public void foo() {
         // Wrong syntax (compile-time error)
         n = 0;
         // Wrong syntax (compile-time error)
         this.n = 0;
         // This is legal
         super.n = 0;
     }
   }
}
```

Se le classi A e B fossero top-level, non potrebbero accedere al campo *n*.

Classi locali

- Una classe interna dichiarata all'interno di un metodo viene chiamata classe *locale*

- Una classe locale non ha specificatore di visibilità, in quanto è visibile solo all'interno del metodo in cui è dichiarata
- Una classe locale non può avere il modificatore static, in quanto eredita il suo essere statica o meno dal metodo in cui è dichiarata

- Oltre a godere delle proprietà comuni alle classi interne, le classi locali "vedono" le variabili locali e i parametri formali del metodo in cui sono contenute, a patto che essi siano *effectively final*

- La slide successiva illustra questa, apparentemente arbitraria, restrizione

Classi locali

- Le istanze di una classe locale possono **vivere più a lungo** del metodo in cui la loro classe è visibile
- Tipicamente, questo succede quando un oggetto di una classe locale viene restituito dal metodo, mascherato da una superclasse o super-interfaccia nota all'esterno

- In questi casi, l'accesso alle variabili locali (compresi i parametri formali) di quel metodo può avvenire quando quel metodo è ormai terminato, cancellando di fatto quelle variabili
  - si ricordi che le variabili locali e i parametri formali sono allocati sullo stack e quindi **vengono cancellati al termine di ciascuna invocazione** del metodo

- Pertanto, per dare l'illusione al programmatore di accedere a quelle variabili, il compilatore in realtà inserisce negli oggetti delle classi locali delle *copie* di quelle variabili

- Se il meccanismo funzionasse anche per variabili non final, il "trucco" delle copie delle variabili locali diventerebbe visibile al programmatore, invece che nascosto
  - Infatti, potrebbe accadere che una variabile locale venisse modificata dopo la creazione di un oggetto della classe interna
  - L'oggetto della classe interna, interrogato successivamente, ricorderebbe il vecchio valore di quella variabile, invece del valore modificato

Esempio: Interfaccia ActionListener

Nella libreria AWT (Abstract Windowing Toolkit) per le interfacce grafiche, l'interfaccia ActionListener rappresenta un oggetto in grado di rispondere ad un evento

```
interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

Esempio di classe locale

- Il seguente frammento crea due widget grafici:
  - una scritta statica (cioè non editabile)
  - un pulsante

```
final JLabel label = new JLabel("Hello");
JButton b = new JButton(...);
```

- Alla pressione del pulsante, vogliamo modificare il contenuto della scritta

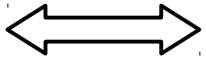
- Usiamo una classe locale per creare un oggetto che implementi ActionListener

```
class MyActionListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      // reagisci alla pressione del tasto b
      label.setText("Goodbye");
   }
}
b.addActionListener(new MyActionListener());
```

Classi anonime

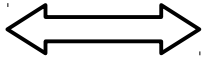
- Talvolta una classe interna viene utilizzata **soltanto una volta**, tipicamente per istanziare un oggetto che poi viene mascherato con una classe o interfaccia nota
- In questi casi, si può utilizzare una classe anonima
- Una classe anonima si definisce a partire da una classe o una interfaccia esistente
- Nella seguente tabella, a sinistra riportiamo la sintassi per la creazione di classi anonime a partire da una classe o interfaccia esistente
- A destra riportiamo il costrutto equivalente utilizzando normali classi con nome

```
new Classe(exp_1, ..., exp_n) {
   <body>
}
```



```
class X extends Classe {
   public X(T_1 x_1, ..., T_n x_n) {
        super(x_1, ..., x_n);
   }
   <body>
}
new X(exp_1, ..., exp_n)
```

```
new Interfaccia() {
   <body>
}
```



```
class X implements Interfaccia {
   <body>
}
new X()
```

Esempio di classe anonima da interfaccia

- Nel seguente frammento viene usata una classe anonima per creare un oggetto che implementi ActionListener

```
JButton b = new JButton(...);

b.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // reagisci alla pressione del tasto b
    }
};
```

Esempio di classe anonima da classe

- Consideriamo la solita classe Employee, dotata di campi nome e salario
- Il seguente metodo usa una classe anonima per creare un Employee dotato di un titolo (Dott., Prof., etc.)
- Il titolo comparirà nella stringa restituita dal metodo toString

```
public static Employee makeSpecialEmployee(String myTitle, String myName, int mySalary)
{
  return new Employee(myName, mySalary) {
       @Override
       public String toString() {
            return myTitle + " " + getName() + ": " + getSalary();
       }
  };
}
```

Il parametro formale "myTitle" è visibile all'interno della classe anonima in quanto *effectively final*

Memory layout di classi interne

- La figura rappresenta il memory layout delle classi definite di seguito
- Notate in particolare la differenza tra classe interna e sottoclasse

```
class A {
    private int x;
    private class B {
            private int y;
    }
    public static class C extends A {
            private int z;
    }
    public Object f(final int n) {
            class Locale {
                    private int m = n;
            }
            return new Locale();
    }
    public static void main(String[] args) {
            A a = new A();
            B b = a.new B();
            C c = new C();
            Object o = a.f(7);
    }
}
```

Memory layout di classi interne

- La figura rappresenta il memory layout delle classi definite di seguito
- Notate in particolare la differenza tra classe interna e sottoclasse

```
class A {
    private int x;
    private class B {
            private int y;
    }
    public static class C extends A {
            private int z;
    }
    public Object f(final int n) {
            class Locale {
                    private int m = n;
            }
            return new Locale();
    }
    public static void main(String[] args) {
            A a = new A();
            B b = a.new B();
            C c = new C();
            Object o = a.f(7);
    }
}
```

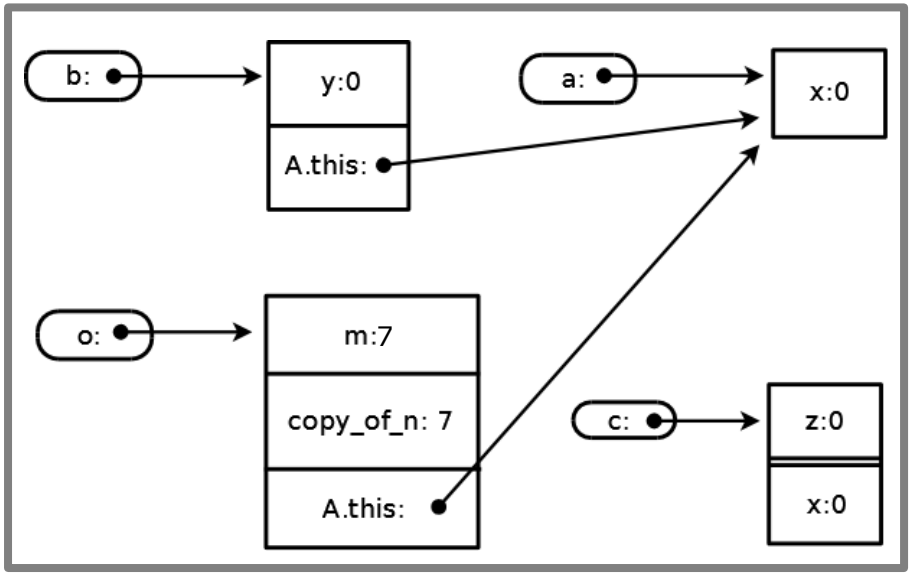


Figura 1: Il memory layout del frammento di programma a sinistra. In particolare, sono mostrati i quattro oggetti creati dal main.

- **C++**
  - [*Come Java*] Prevede classi interne (nested) con restrizioni di visibilità arbitraria (ad es., private)
  - [*Come Java*] La classe interna ha accesso a tutti i membri della contenitrice
  - [*Diverso*] La classe contenitrice *non* ha accesso privilegiato al contenuto di quella interna
  - [*Diverso*] Non prevede riferimento implicito alla classe contenitrice
  - Privilegi di visibilità si ottengono anche tramite il costrutto *friend*

- **C#**
  - Regole analoghe a C++
  - Nessun costrutto friend
  - Le linee guida MS sconsigliano esplicitamente di creare classi interne pubbliche

Esercizio (esame 21/04/2008, #1)

Nell'ambito di un programma di geometria, si implementi la classe Triangolo, il cui costruttore accetta le misure dei tre lati. Se tali misure non danno luogo ad un triangolo, il costruttore deve lanciare un'eccezione. Il metodo getArea restituisce l'area di questo triangolo.

Si implementino anche la classe Triangolo.Rettangolo, il cui costruttore accetta le misure dei due cateti, e la classe Triangolo.Isoscele, il cui costruttore accetta le misure della base e di uno degli altri lati.

Si ricordi che:

- Tre numeri a, b e c possono essere i lati di un triangolo a patto che a<b+c, b<a+c e c<a+b.
- L'area di un triangolo di lati a, b e c è data da: $\sqrt{p(p-a)(p-b)(p-c)}$ (formula di Erone), dove p è il semiperimetro.

Esempio d'uso (fuori dalla classe Triangolo):

```
Triangolo x = new Triangolo(10,20,25);
Triangolo y = new Triangolo.Rettangolo(5,8);
Triangolo z = new Triangolo.Isoscele(6,5);

System.out.println(x.getArea());
System.out.println(y.getArea());
System.out.println(z.getArea());
```

Output dell'esempio d'uso:

```
94.9918
19.9999
12.0
```

Altri esercizi: 29/1/09 (Interval)