Linguaggi di Programmazione I – Lezione 16

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
http://sette.dnsalias.org

10 giugno 2008

Classi interne

Introduzione

Approccio ad una GUI

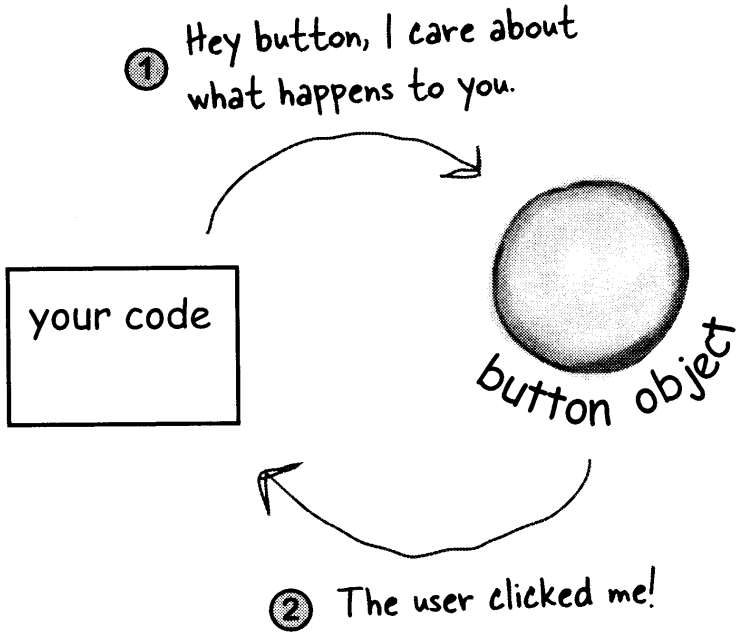
La mia prima GUI

```
import javax.swing.*;

public class ProvaGUI {
  private JFrame frame;
  private JButton pulsante;

  public static void main (String[] args) {
    ProvaGUI gui = new ProvaGUI();
    gui.go();
  }

  public void go() {
    frame = new JFrame();
    pulsante = new JButton("Cliccami");

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(pulsante);
    frame.setSize(300,300);
    frame.setVisible(true);
  }
}
```

Ma non c'è nessuna azione collegata alla pressione del pulsante.

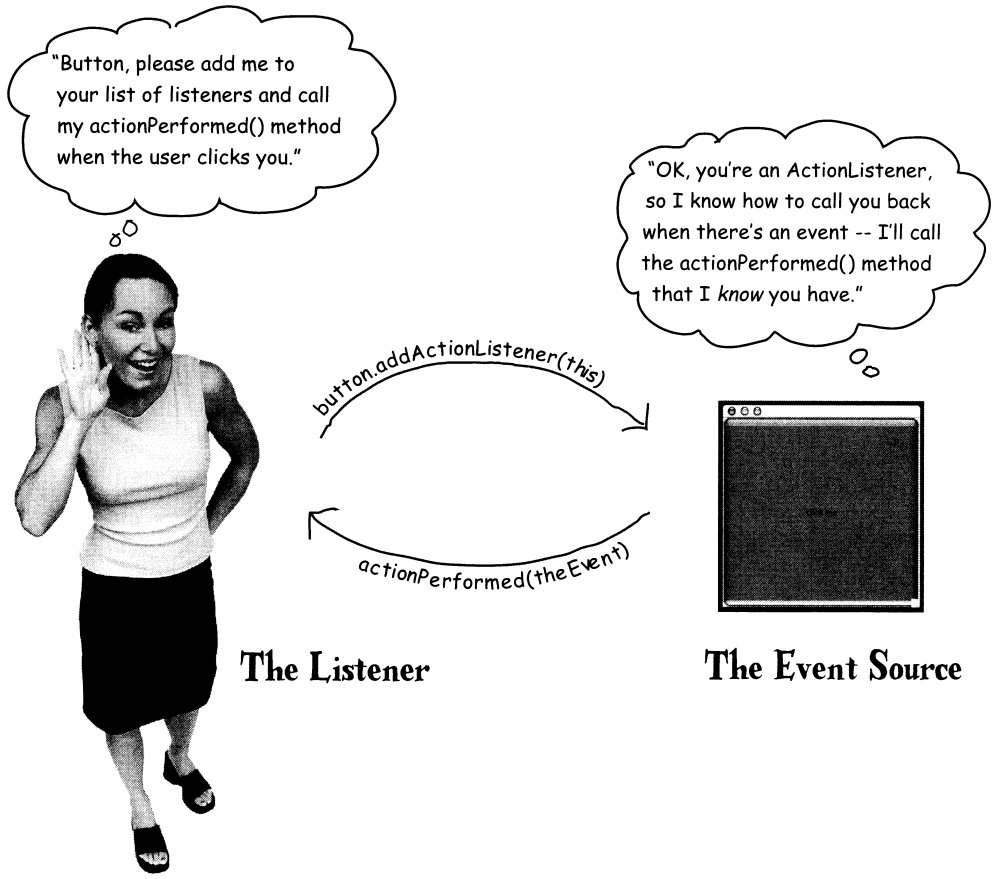
Gestione degli eventi

- Piuttosto che controllare ciclicamente e continuamente lo stato dell'oggetto pulsante, è molto più efficiente lasciare che sia l'oggetto pulsante a comunicarci il proprio cambiamento di stato:



- Il nostro codice diventa un *ascoltatore* e il pulsante è la *sorgente*.

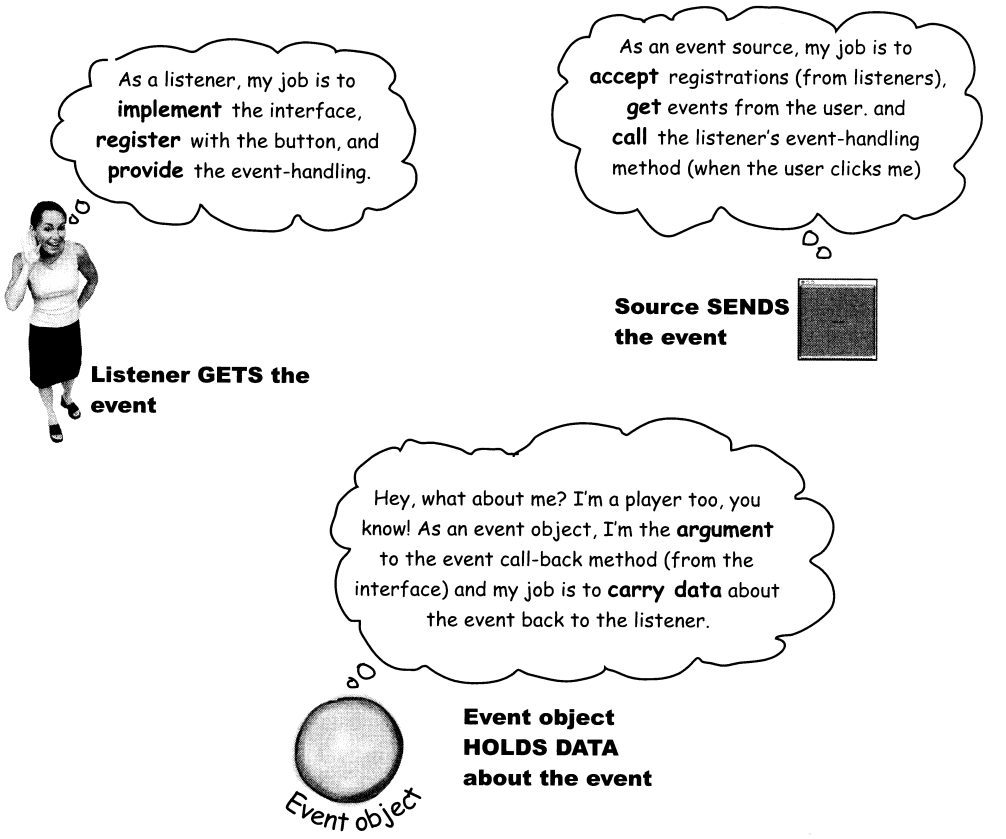
La comunicazione tra sorgente e ascoltatore



GUI con un solo pulsante

```
import javax.swing.*;
import java.awt.event.*;

public class Gui1 implements ActionListener {
  private JFrame frame; private JButton pulsante;

  public static void main (String[] args) {
    new Gui1().go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    pulsante = new JButton("Cliccami");
    pulsante.addActionListener(this);

    frame.getContentPane().add(pulsante);
    frame.setSize(300,300); frame.setVisible(true);
  }

  public void actionPerformed(ActionEvent event) {
    pulsante.setText("Pulsante cliccato!");
  }
}
```


Ascoltatori, sorgenti ed eventi



Proviamo con due pulsanti e due azioni

```
public class Gui2 implements ActionListener {
  private JFrame frame; private JButton pulsante1, pulsante2;
  // qui il solito codice come prima:
  // - generazione dell'oggetto frame e dei due oggetti
  //   pulsanti;
  // - registrazione di this con i due oggetti pulsanti
  //
  // ora devo descrivere le due diverse azioni:

  public void actionPerformed(ActionEvent event) {
    pulsante1.setText("P1: cliccato!");
  }

  public void actionPerformed(ActionEvent event) {
    pulsante2.setText("P2: cliccato!");
  }
}
```

Ma questo è un sovraccaricamento illegale di metodi!

Un primo tentativo

- A questo punto, potrei definire un unico metodo che chieda all'evento ricevuto come parametro informazioni sulla natura della sorgente e che si comporti in modo diverso in base alla risposta,
- ma ciò significherebbe costruire codice non ben separato: una modifica del funzionamento della sorgente comporterebbe la modifica di ogni ascoltatore.

Un altro tentativo

Creazione di due classi ActionListener separate:

```
public class Gui2 {
  private JFrame frame; private JButton pulsante1, pulsante2;
  void go() {
    // codice per istanziare i due oggetti ascoltatori
    // e registrarli con i due oggetti pulsanti
  }
}
```

```
class Ascoltatore1 implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    pulsante1.setText("P1: cliccato!");
  }
}
```

```
class Ascoltatore2 implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    pulsante2.setText("P2: cliccato!");
  }
}
```

Ma `pulsante1` e `pulsante2` non sono accessibili all'esterno di `Gui2`.

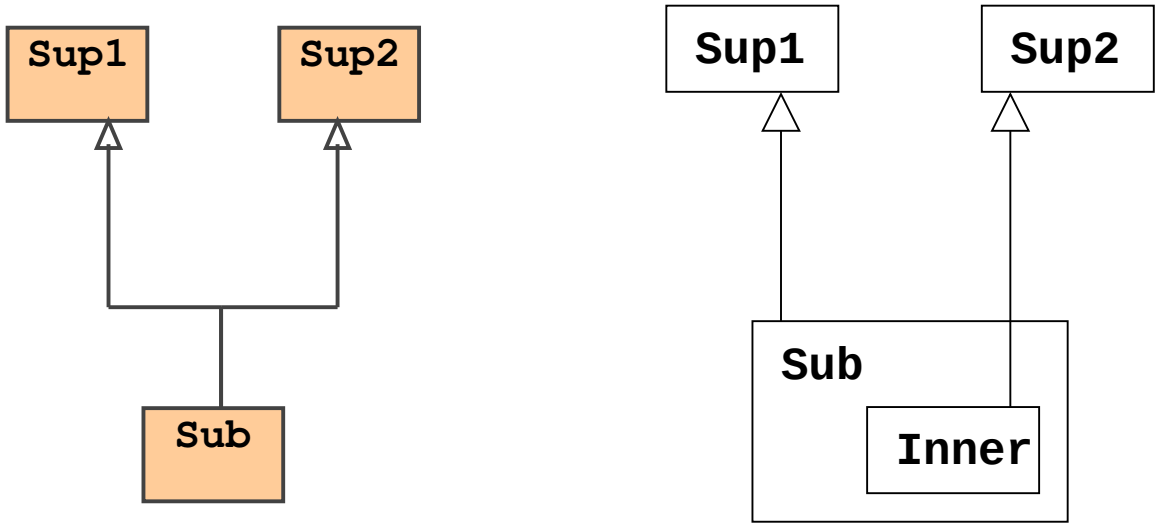
Soluzioni a disposizione

- Rompere l'incapsulazione e rendere pubblici i riferimenti ai pulsanti nella classe `Gui2`,
- oppure rendere più complicate le classi degli ascoltatori, aggiungendo un attributo che possa riferire al pulsante ed un costruttore che consenta alla classe `Gui2` di inizializzare quell'attributo con il valore del proprio attributo (privato) `pulsante1` o `pulsante2`,
- oppure ...

Sarebbe bello se...

Utilità delle classi interne

- Classi *inner* sono state aggiunte con la JDK 1.1.
- Sono a volte chiamate classi annidate.
- Danno al programma maggiore chiarezza e lo rendono più conciso.
- Offrono anche una soluzione elegante al problema realizzativo dell'ereditarietà multipla:



Classe membro di altra classe 16 / 45

Introduzione

- Fondamentalmente, sono come le altre classi, ma sono dichiarate all'interno di altre classi.
- Possono essere poste in qualunque blocco, incluso i blocchi dei metodi.
- Classi definite all'interno di metodi differiscono leggermente dalle altre e saranno trattate meglio in seguito.
- Per ora, con il nome di "classe membro", intenderemo una classe che non è definita in un metodo, ma semplicemente in un'altra classe (come gli altri membri: attributi e metodi).
- La complessità dell'argomento è relativa agli ambiti di validità e di accesso, in particolare all'accesso a variabili nell'ambito includente.

Sintassi base

```
public class Esterno {
  private int x;
  public class Interno {
    private int y;
    public void metodoInterno() {
      System.out.println("y=" + y);
    }
  }
  public void metodoEsterno() {
    System.out.println("x=" + x);
  }
  // altri metodi ...
}
```

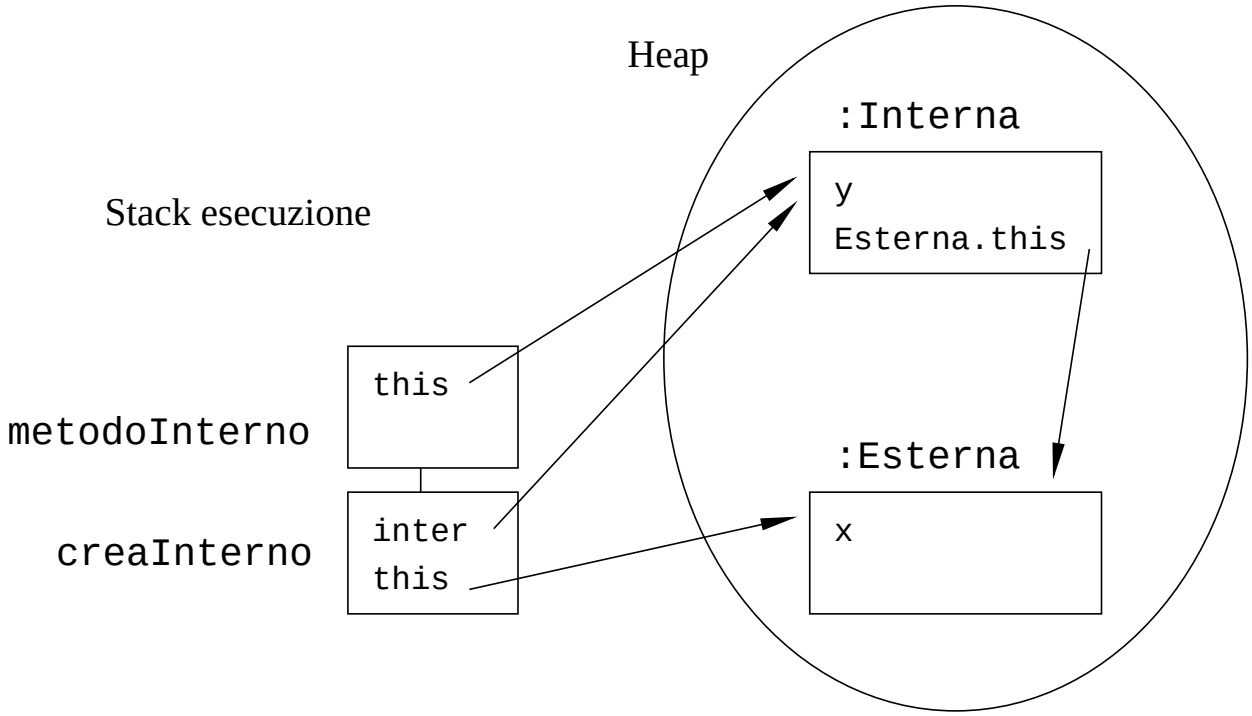
- Il nome della classe includente diventa parte del nome della classe inclusa.
- In questo caso, i nomi completi delle due classi sono `Esterno` ed `Esterno.Interno`.
- Questo formato è reminiscente del modo in cui una classe è nominata all'interno di un pacchetto. Infatti, una classe interna appartiene alla propria classe includente allo stesso modo con cui una classe appartiene al proprio pacchetto.
- È illegale che una classe ed un pacchetto abbiano lo stesso nome, perciò non c'è ambiguità di interpretazione.
- Attenzione: la rappresentazione puntata vale solo all'interno del sorgente Java; essa non riflette il nome del file delle singole classi. Sul disco la classe interna si chiama `Esterno$Interno`.

Costruzione (1)

```
public class Esterna {
  private int x;
  public class Interna {
    private int y;
    public void metodoInterno() {
      System.out.println("x=" + x);
        // attributo esterno
        // accessibile anche
        // con Esterna.this.x
      System.out.println("y=" + y);
        // attributo interno
    }
  }
  public void metodoEsterno() {
    System.out.println("x=" + x);
  }
  public void creaInterno() {
    Interna inter = new Interna();
    inter.metodoInterno();
  }
  // altri metodi ...
}
```

- Quando si deve costruire una istanza di una classe `Interna`, in genere, DEVE esistere già una istanza della classe `Esterna` che agisca da contesto.
- L'oggetto interno può accedere a tutte le componenti dell'oggetto esterno, indipendentemente dalla loro visibilità, attraverso un riferimento implicito.
- L'oggetto esterno, avendo costruito quello interno, ne possiede un riferimento. L'oggetto interno, però, può accedere all'oggetto esterno attraverso il riferimento implicito (tale riferimento può essere anche esplicitato; nel nostro esempio si chiama `Esterna.this`): *oggetti interno ed esterno si appartengono*.

Costruzione (2)



Costruzione (3)

- Nel caso in cui non esista l'oggetto esterno, per esempio perché la JVM sta eseguendo un metodo statico, allora bisogna costruirne uno "al volo":

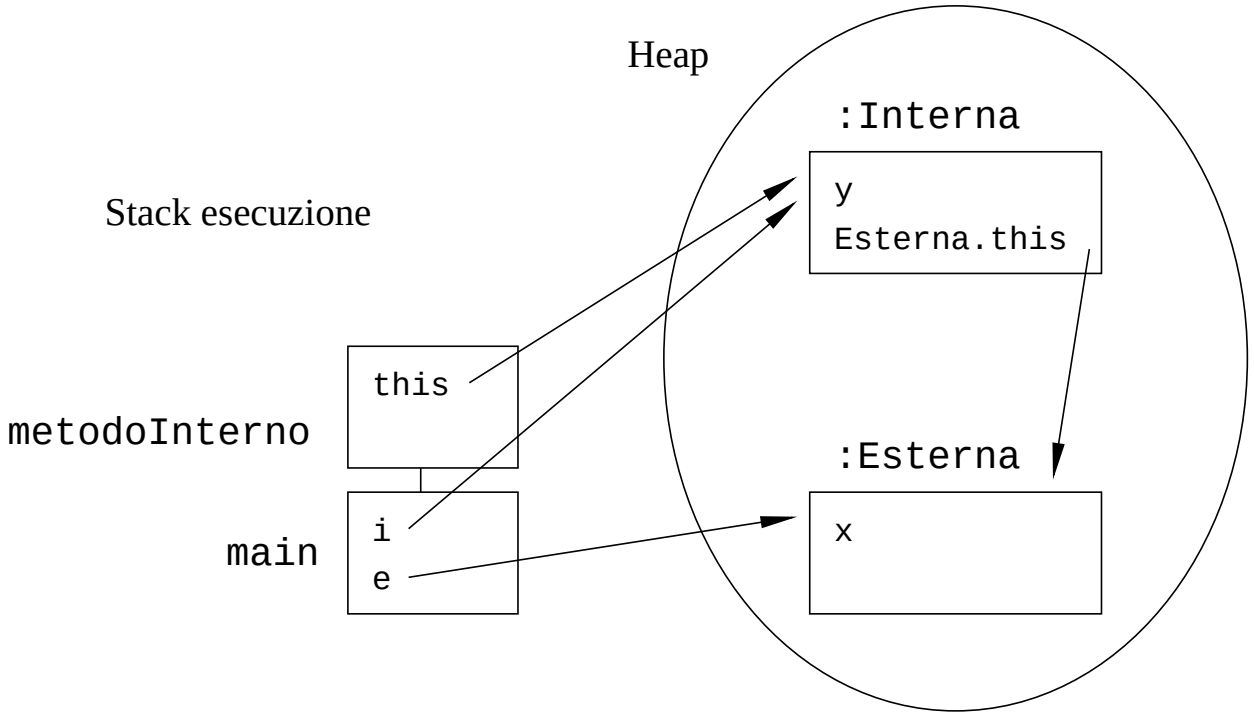
```
public static void main(String args[]) {
  Esterna.Interna i = new Esterna().new Interna();
  i.metodoInterno();
}
```

  oppure, in forma meno compatta:

```
public static void main(String args[]) {
  Esterna e = new Esterna();
  Esterna.Interna i = e.new Interna();
  i.metodoInterno();
}
```

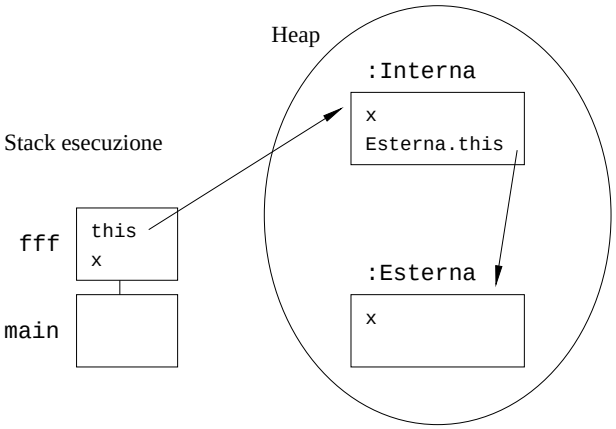
- In entrambi i casi `new` è usato come se fosse un metodo della classe esterna.

Costruzione (4)



Esempio 1

```
public class Esterna {
  private int x;

  public class Interna {
    private int x;

    public void fff(int x) {
      // parametro locale:
      x++;
      // attributo interno:
      this.x++;
      // attributo esterno:
      Esterna.this.x++;
    }
  }
}
```



Esempio 2: GUI a due pulsanti

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Gui2 {
  private JFrame frame;
  private JButton pulsante1, pulsante2;

  public static void main (String[] args) {
    new Gui2().go();
  }

  public void go() {
    frame = new JFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    pulsante1 = new JButton("Pulsante 1");
    pulsante1.addActionListener(new Ascoltatore1());

    pulsante2 = new JButton("Pulsante 2");
    pulsante2.addActionListener(new Ascoltatore2());

    frame.getContentPane().add(BorderLayout.WEST, pulsante1);
    frame.getContentPane().add(BorderLayout.EAST, pulsante2);
    frame.setSize(300,300);
    frame.setVisible(true);
  }

  class Ascoltatore1 implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      pulsante1.setText("P1: cliccato!");
    }
  }

  class Ascoltatore2 implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      pulsante2.setText("P2: cliccato!");
    }
  }
}
```


Modificatori (1)

- Classi membro possono essere opzionalmente marcate con qualunque modificatore di accesso (a differenza delle classi top-level, che possono essere opzionalmente marcate solo `public`).
  Il significato è identico a quello degli altri membri della classe (attributi o metodi). Per esempio, una classe membro `private` può essere vista solo all'interno della classe includente: non si può nominarla al di fuori della classe includente.
- Classi membro `final` non possono essere specializzate.
- Classi membro `abstract` non possono essere istanziate.
- Classi membro `static`:
  - esistono a prescindere dell'oggetto includente;
  - non hanno il riferimento implicito all'oggetto includente;
  - sono praticamente delle classi top-level, con uno schema di nomi modificato;
  - sono istanziate in questo modo:

```
public class Esterna {
  public static class Interna {}
  public static void main
        (String[] args) {
    Interna i =
        new Esterna.Interna();
  }
}
```

Modificatori (2)

Vogliamo esplicitare che esiste una importante restrizione ai modificatori attribuibili ai membri di una classe interna (attenzione, non ai membri di una classe top-level):

- Poiché una istanza di una classe interna non-`static` può solo esistere insieme ad una istanza della classe includente, cioè, in altri termini, poiché una classe interna non-`static` esiste solo per generare istanze e non per fornire servizi, allora
  I MEMBRI DI UNA CLASSE INTERNA NON-`static` NON POSSONO ESSERE MARCATI `static`.
- Ciò implica, per esempio, che:
  - possono esistere classi `static` di livello di annidamento diverso dal primo solo se sono annidate in altre classi `static`;
  - è possibile che esistano interfacce annidate, ma in questo caso esse hanno l'implicito (o esplicito) modificatore `static` e seguono la stessa regola precedente.

Classe definita all'interno di un metodo 27 / 45

Introduzione

Classi definite in un metodo:

1. Come tutte le altre variabili del metodo, sono locali in quel metodo e non possono essere marcate con nessun modificatore tranne che con `abstract` o `final` (non entrambi, perché?).
2. Esse non sono accessibili (come tutte le altre variabili locali) in alcun modo all'esterno del metodo.
3. Hanno accesso limitato alle altre variabili locali del metodo (discuteremo subito).

Accesso a variabili locali

- La regola di accesso è semplice: *ogni variabile locale (anche un parametro) del metodo includente NON può essere utilizzata dalla classe interna, a meno che quella variabile non sia marcata* `final`.
- In realtà un oggetto (sullo heap) non dovrebbe accedere affatto alle variabili dei metodi (di stack), poiché gli oggetti sullo heap possono sopravvivere ai record sullo stack.
- Sveliamo il trucco utilizzato dalla JVM per permettere l'accesso limitatamente alle variabili `final`:

  La variabile `final` è una costante nel metodo. L'oggetto che deve accedervi ne possiede semplicemente una copia (per esempio generata durante la costruzione dell'oggetto). In tal modo, alla terminazione del metodo, la costante sarà ancora utilizzabile, anche se non esiste più il record di attivazione che la contiene.

Esempio 3

```
public class Esterna {
  private int m = (int)(Math.random() * 100);
  public static void main(String[] args) {
    Esterna oggetto = new Esterna();
    oggetto.fai(m, 2*m);
  }

  public void fai(int x, final int y) {
    int a = x + y;
    final int b = x - y;
    class Interna {
      public void metodo() {
        System.out.println("m=" + m);
//        System.out.println("x=" + x); // Illegale
        System.out.println("y=" + y);
//        System.out.println("a=" + a); // Illegale
        System.out.println("b=" + b);
      }
    }
    Interna oggetto = new Interna();
    oggetto.metodo();
  }
}
```


Classi anonime 31 / 45

Introduzione

- Possono essere usate per estendere un'altra classe oppure, in alternativa, per implementare una singola interfaccia.
- La sintassi non concede di fare entrambe le cose, nè di implementare più di una interfaccia: se si dichiara una classe che implementa una singola interfaccia, allora la classe è sottoclasse diretta di `java.lang.Object`.
- Poiché non si conosce il nome di una tale classe, non si può usare `new` nel modo solito per crearne una istanza.
- Infatti, la definizione, la costruzione e il primo uso (spesso in una assegnazione) avvengono nello stesso enunciato.
- L'utilità è nella possibilità di sovrapporre qualche metodo della superclasse (o di implementare metodi di una interfaccia) senza la necessità di scrivere una vera classe che lo faccia.

Esempio 4.a

```
 1. class A {
 2.   public void f() {
 3.     System.out.println("A");
 4.   }
 5. }
 6. class B {
 7.   A a = new A() {
 8.     public void f() {
 9.       System.out.println("B");
10.     }
11.   }; // notare il ';'
12. }
```

- La variabile a si riferisce non ad una istanza di `A`, ma *ad una istanza di una sottoclasse anonima di* `A`.
- La linea 7 comincia con una dichiarazione di variabile-riferimento ad `A`. Ma, invece di essere:

```
A a = new A(); // con ';'
```

  c'è una parentesi graffa aperta alla fine, che si chiude alla linea 11. Qui c'è il punto-e-virgola che manca.
- Quello che c'è tra la linea 7 e la 11 si può parafrasare: "Dichiara una variabile a di tipo `A`; poi dichiara una nuova classe, senza nome, *sottoclasse* di `A`, che contenga solo una sovrapposizione del metodo `f()`; infine, costruisci una istanza di tale sottoclasse ed assegna il suo riferimento ad a".

Esempio 4.b

Il polimorfismo è in azione: noi usiamo un riferimento ad una superclasse per riferirci ad una istanza di una sottoclasse. Ciò implica:

1. non avremo mai la possibilità di accedere a ciò che si aggiunge nella definizione della sottoclasse anonima (con quale nome faremmo un cast?);
2. l'uso di una classe interna anonima è possibile solo per quanto riguarda i metodi che essa sovrappone, e che sono citabili solo perchè presenti nella superclasse;
3. essa non può avere un costruttore esplicito (che nome avrebbe?).

```
class A {
  public void f() {...}
}
class B {
  A a = new A () {
    public void g() {...}
    public void f() {...}
  };
  public void usa() {
    a.f(); // OK, f() e' anche metodo di A
    a.g(); // Illegale, g() non e' un metodo di A
  }
}
```


Questionario

D 1

Quali, tra i seguenti enunciati, sono veri?

A. Una classe interna può essere marcata `private`.
B. Una classe interna può essere marcata `static`.
C. Una classe interna definita in un metodo deve sempre essere anonima.
D. Una classe interna definita in un metodo può accedere tutte le variabili locali di quel metodo.
E. La costruzione di una classe interna può richiedere una istanza della classe esterna.

D 2

Si consideri la seguente definizione:

```
 1. public class Outer {
 2.   public int a = 1;
 3.   private int b = 2;
 4.   public void method(final int c) {
 5.     int d = 3;
 6.     class Inner {
 7.       private void iMethod(int e) {
 8.
 9.       }
10.     }
11.   }
12. }
```

Quali variabili possono essere correttamente utilizzate alla linea 8?

A. `a`
B. `b`
C. `c`
D. `d`
E. `e`

D 3

Quali dei seguenti enunciati sono veri?

A. Sia `Inner` una classe non-`static` dichiarata all'interno di una classe pubblica `Outer`. Una istanza di `Inner` può essere costruita in questo modo:

```
new Outer().new Inner()
```

B. Se una classe anonima all'interno della classe `Outer` è definita in modo da implementare l'interfaccia `Rotante`, essa può essere costruita così:

```
new Outer().new Rotante()
```

C. Sia `Inner` una classe non-`static` dichiarata all'interno di una classe pubblica `Outer`. In un metodo statico, una istanza di `Inner` può essere costruita in questo modo:

```
new Inner()
```

D. Una istanza di classe anonima che implementi l'interfaccia `Inter` può essere costruita e restituita da un metodo come questo:

```
return new Inter(int x) {
  int x;
  public Inter(int x) {
    this.x = x;
  }
};
```

D 4

Qual è il risultato del seguente codice?

```
public class MiaClasse {
  public static void main
                  (String[] args) {
    Esterna e = new Esterna();
    System.out.println(
     e.creaInterna().getSegreto());
  }
}
class Esterna {
  private int segreto;
  Esterna() { segreto=7; }

  class Interna {
    int getSegreto() {
      return segreto; }
  }
  Interna creaInterna() {
    return new Interna(); }
}
```

A. Il codice non viene compilato, poiché la classe `Interna` non può essere dichiarata nella classe `Esterna`.

B. Il codice non viene compilato, poiché il metodo `creaInterna()` non può passare oggetti della classe interna a metodi all'esterno della classe `Esterna`.

C. Il codice non viene compilato, poiché la variabile `segreto` non è accessibile.

D. Il codice non viene compilato, poiché il metodo `getSegreto()` non è visibile dal `main`.

E. Il codice viene compilato correttamente e stampa 7.

D 5

Quali delle seguenti affermazioni riguardanti le classi interne sono vere?

A. Una istanza di una classe interna `static` è associata ad una istanza di una classe esterna.

B. Una istanza di una classe interna `static` può contenere membri non-`static`.

C. Una interfaccia interna `static` può contenere membri non-`static`.

D. Una interfaccia interna `static` è associata ad una istanza di classe esterna.

E. Per ciascuna istanza di classe esterna, possono esistere molte istanze di una classe interna non-`static`.

D 6

```
public class Nesting {
  public static void main
                  (String[] args) {
    B.C o = new B().new C();
  }
}
class A {
  int val;
  A(int v) { val = v; }
}
class B extends A {
  int val = 1;
  B() { super(2); }
  class C extends A {
    int val = 3;
    C() {
      super(4);
      System.out.print(B.this.val);
      System.out.print(C.this.val);
      System.out.print(super.val);
    }
  }
}
```

Qual è l'output del programma precedente?

A. Errore di compilazione

B. 234

C. 142

D. 134

E. 321

D 7

Queli delle seguenti affermazioni sono vere?

A. Classi interne non-`static` devono avere accessibilità `public` o di default.
B. Tutte le classi interne possono contenere classi `static`.
C. I metodi in tutte le classi interne possono essere dichiarati `static`.
D. Tutte le classi interne possono essere dichiarate `static`.
E. Le classi interne `static` possono contenere metodi non-`static`.

D 8

Quali delle seguenti affermazioni sono vere?

A. Non si possono dichiarare membri `static` all'interno di classi interne non-`static`.
B. Se una classe interna non-`static` è posta nella classe `Esterna`, allora i metodi di quella classe devono usare il prefisso `Esterna.this` per accedere ai membri della classe `Esterna`.
C. Tutti le variabili di una classe interna devono essere dichiarate `final`.
D. Classi anonime non possono avere costruttori.
E. Se `objRef` è una istanza di una classe interna nella classe `Esterna`, allora `(objRef instanceof Esterna)` ha valore `true`.

D 9

Quali affermazioni sono vere?

A. Classi membro di pacchetto possono essere dichiarate `static`.
B. Classi membro di classi top-level possono essere dichiarate `static`.
C. Classi locali possono essere dichiarate `static`.
D. Classi anonime possono essere dichiarate `static`.
E. Nessuna classe può essere dichiarata `static`.

D 10

```
interface I { int f(); }
```

Data la dichiarazione precedente, quali dei seguenti metodi sono validi?

A.

```
I makeI(int i) {
  return new I() {
    public f() {
      return i;
    }
  };
}
```

B.

```
I makeI(final int i) {
  return new I {
    public f() {
      return i;
    }
  };
}
```

C.

```
I makeI(int i) {
  class MyI implements I {
    public int f() {
      return i;
    }
  }
  return new MyI();
}
```

D.

```
I makeI(final int i) {
  class MyI implements I {
    public int f() {
      return i;
    }
  }
  return new MyI();
}
```