Programmazione III

Interfacce grafiche (prima parte)

Alberto Martelli (con aggiornamenti da parte
di Liliana Ardissono)

PROGRAMMAZIONE GRAFICA

Molti programmi interagiscono con l'utente attraverso una interfaccia grafica

GUI - Graphical User Interface

Java fornisce diverse librerie di classi per realizzare GUI.

Nelle prime versioni di Java (1.0, 1.1) era fornita la libreria

AWT (Abstract Window Toolkit)

per realizzare la portabilità, la gestione dei componenti grafici era delegata ai toolkit nativi delle varie piattaforme (Windows, Solaris, IoS, …)

Successivamente è stata fornita la libreria **SWING**, che fa un uso molto ridotto dei toolkit nativi.

I componenti sono *dipinti* in finestre vuote.

In ogni caso, programmi Java che usano SWING, devono spesso usare anche classi AWT.

Il componente di più alto livello di una interfaccia grafica è una finestra, realizzata dalla classe **JFrame**.

Tutte le classi i cui nomi iniziano con J appartengono alla libreria javax.swing.

Gli altri componenti al livello top sono:

JApplet e **JDialog**

I *frame* sono dei contenitori, in cui si possono inserire altri componenti (pulsanti, testo, …) o in cui si può disegnare.

Altri contenitori sono:

JPanel

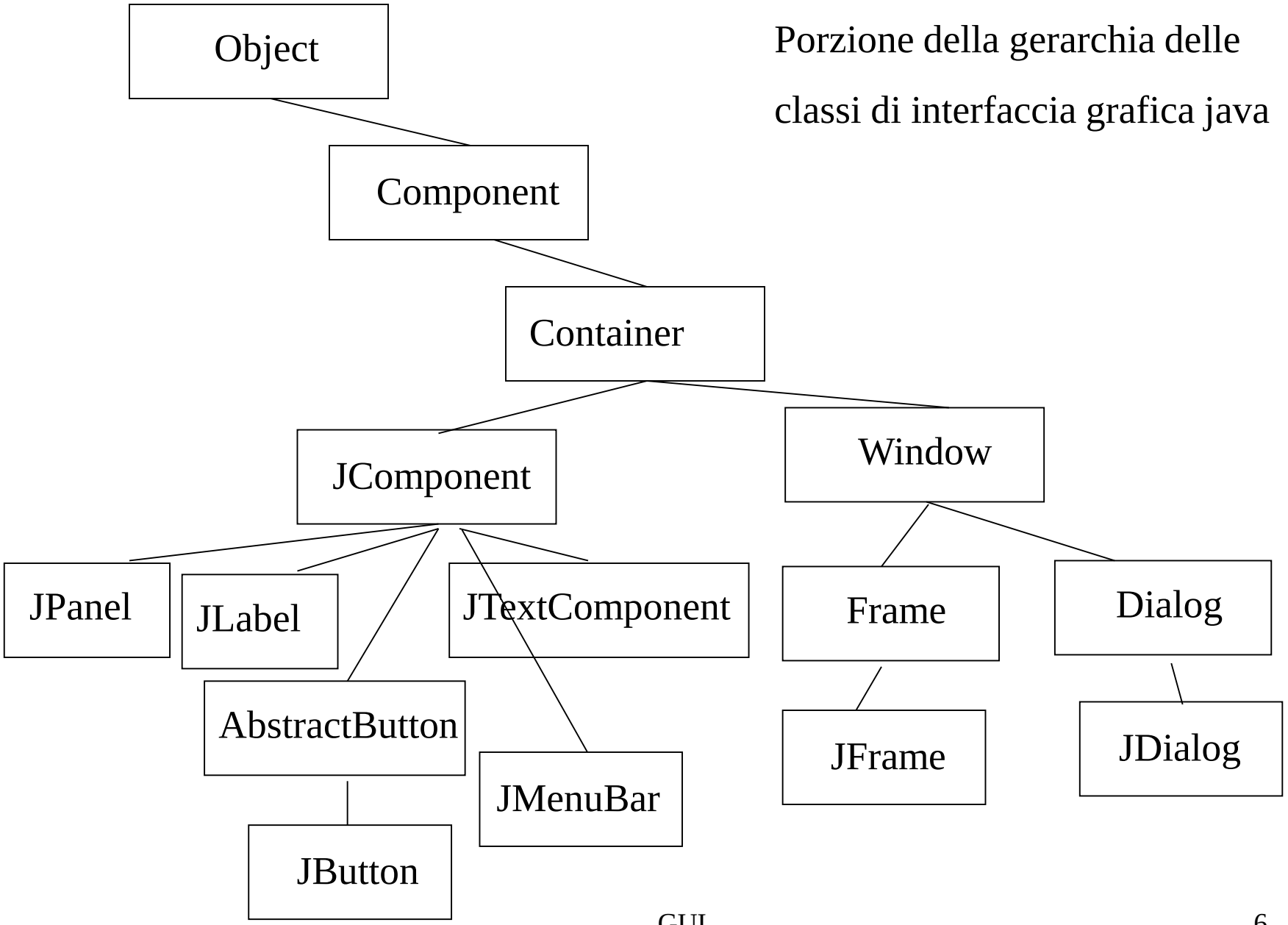
Container

Nella libreria Swing sono disponibili numerosi componenti:

- pulsanti
- check box
- menu
- barre di scorrimento
- liste
- finestre di dialogo
- file chooser
- campi di testo
- alberi

e numerosi strumenti per realizzare grafica.

Porzione della gerarchia delle classi di interfaccia grafica java



Vediamo un semplice esempio di una finestra che contiene la scritta *Hello World*.

Un **JFrame** ha una struttura complessa. In particolare ha un *pannello del contenuto* (che è un **Container**) in cui si possono inserire i componenti.

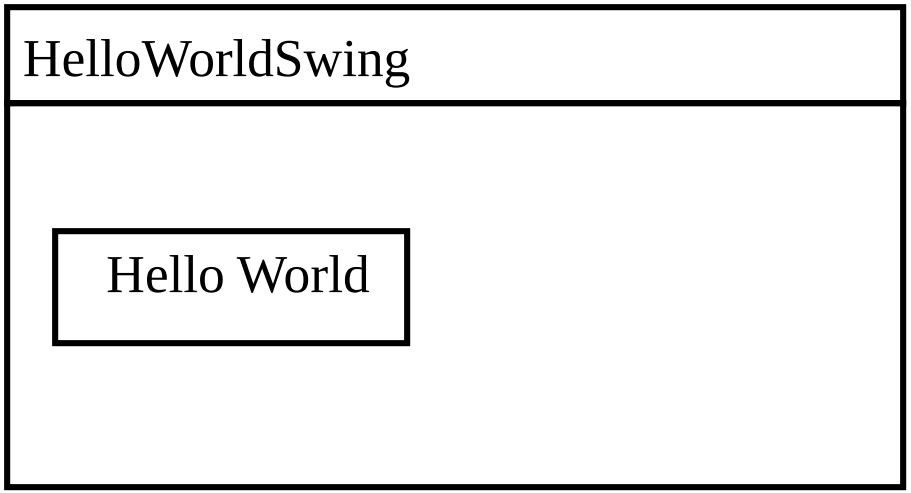
La scritta *Hello World* viene messa in un componente **JLabel**, etichetta con testo, che viene inserito nel "content pane" del **JFrame**.

Per inserire un componente in un contenitore si usa il metodo **add** del contenitore.

Un componente viene inserito nel contenitore secondo un *layout* (disposizione) predefinito, che dipende dal tipo del contenitore.

I *layout* possono essere modificati dal programmatore.

```
JFrame frame = new JFrame("HelloWorldSwing");
JLabel label = new JLabel("Hello World");
frame.add(label);
```



```
import javax.swing.*;

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

    JFrame frame = new JFrame("HelloWorldSwing");

    final JLabel label = new JLabel("Hello World");

    frame.add(label);

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
           // quando si chiude la finestra
           // termina l'esecuzione dell'applicazione

    frame.pack();  // fissa la dimensione ottimale
                   // della finestra in base al contenuto
    frame.setVisible(true); // mette la finestra visibile
  }
}
```

Quando si crea un frame, questo ha dimensione 0×0 pixel.

Si può stabilire la dimensione con il metodo **`setSize`**.

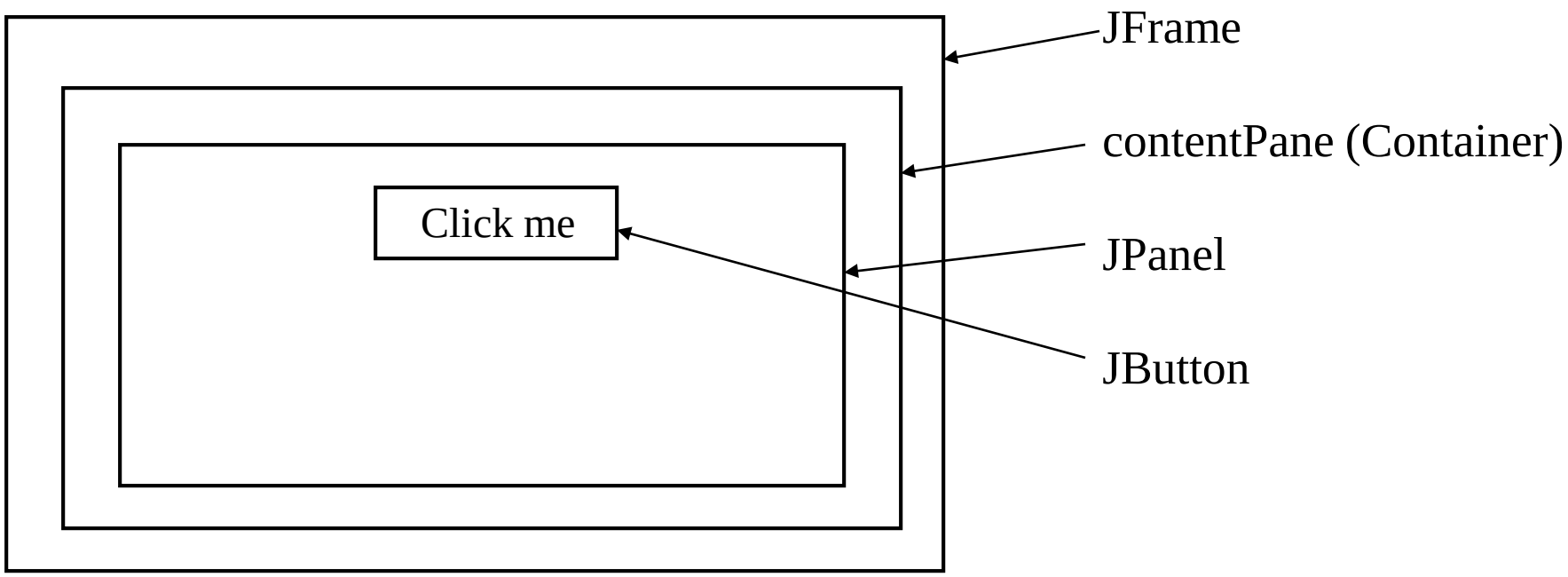
Inizialmente le finestra è invisibile: occorre chiamare **`setVisible(true)`**.

Inoltre, si può creare una classe che estende JFrame, e che nel suo costruttore ha tutti gli elementi del tipo di finestra desiderato:

```
class MyFrame extends JFrame {
        public MyFrame(String s) {
                super(s);
                setSize(400, 200);
                add(new JLabel("ciao"));
                ...
        }
}
```

Vediamo un altro esempio di una finestra che contiene un pulsante che, quando viene premuto, fa beep. Il *layout* può essere realizzato in questo modo:

```
button = new JButton("Click Me");
panel = new JPanel();
panel.add(button);
frame.add(panel);
```



NOTA Si potrebbe evitare di usare il **JPanel** e inserire direttamente il pulsante nel *pannello del contenuto*.

Tuttavia l'aspetto sarebbe diverso perché il pannello del contenuto, che è un **Container**, ha un *layout* di default diverso da quello del **JPanel**.

Inoltre, aggiungere pannelli è utile quando si vuole suddividere una finestra (JFrame) in più aree (contenitori), all'interno di ciascuna si possono inserire i componenti.

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

public class Beeper extends JFrame {
   JButton button;
   JPanel panel;

   Beeper() {
      button = new JButton("Click Me");
      panel = new JPanel();
      panel.add(button);
      add(panel);
   }

   public static void main(String[] args) {
      Beeper beep = new Beeper();
      beep.pack();
      beep.setVisible(true);
   }
}
```

Il programma precedente contiene solo il layout.

Come far sì che, quando si preme il bottone, si senta beep?

Quando si fa clic con il mouse sul bottone, viene generato un **evento** "bottone premuto". Questo evento deve attivare l'azione di fare beep.

Per fare questo si usa una tecnica di programmazione diversa da quella tradizionale: la **programmazione guidata dagli eventi**.

Vediamo le caratteristiche principali di questa tecnica di programmazione, adottata da molti linguaggi usati per realizzare interfacce o per programmare browser (Visual Basic, JavaScript, ...)

Programmazione guidata dagli eventi (event-driven)

I programmi tradizionali, ad esempio quelli che implementano algoritmi, hanno un comportamento funzionale: ricevono un input, eseguono la propria computazione e restituiscono un risultato.

Normalmente questi programmi seguono il proprio flusso di controllo e solo raramente possono contenere punti di diramazione che si basano su input dell'utente.

In molti casi invece, es. interfacce grafiche, un programma deve avere un comportamento **reattivo**: ogni volta che l'utente genera un evento, il programma deve reagire all'evento eseguendo una azione opportuna.

Programmazione guidata dagli eventi (event-driven)

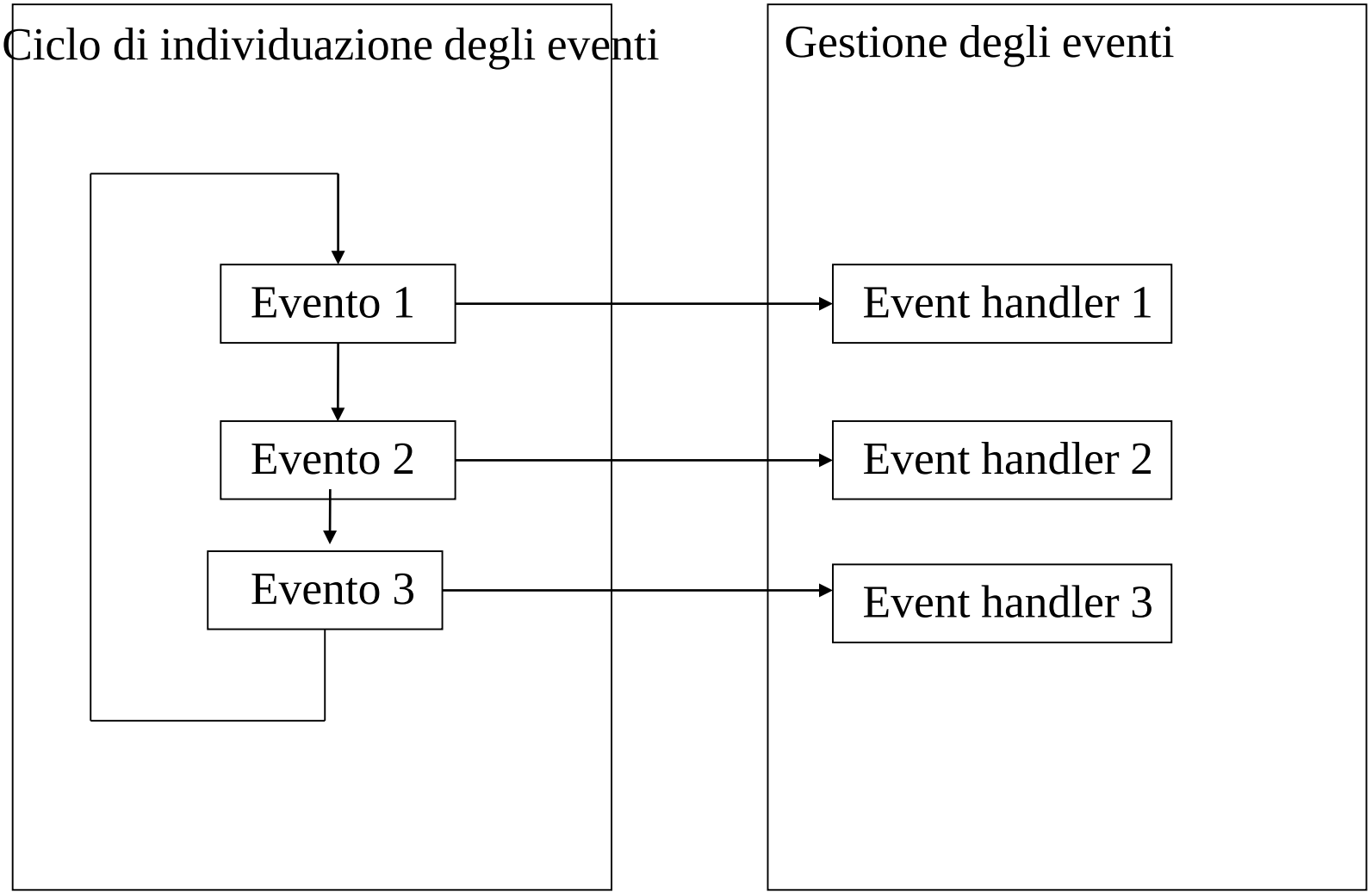
Un programma basato su questa metodologia consiste di un insieme di procedure (**event handlers**), ciascuna delle quali specifica cosa fare quando si verifica un certo evento.

Il programma contiene un event-handler per ogni evento a cui è interessato.

Quando l'evento si verifica, verrà eseguito l'event-handler associato.

Il flusso di controllo con cui il programma viene eseguito non è determinato a priori, ma dipende dall'ordine con cui gli eventi si verificano. Il programma termina quando si verifica un evento che ne richiede la terminazione.

Schema di programma guidato dagli eventi



L'applicazione deve inizialmente creare gli *event-handlers*.

Successivamente deve registrare gli *event handlers* presso la sorgente degli eventi, ossia legare ogni *event handler* a un evento che riguarda la specifica sorgente (componente della GUI).

La sorgente degli eventi esegue un ciclo degli eventi, per scoprire se qualche evento si verifica.

Quando si verifica un evento, la sorgente degli eventi invoca l'*event handler* (*listener*) associato, se esiste.

Il ciclo degli eventi è un concetto astratto. In realtà gli eventi potrebbero essere segnalati da un meccanismo di *interrupt*, senza bisogno che il programma esegua continuamente il ciclo.

In Java gli **eventi** sono oggetti derivati dalla classe **EventObject**.

Si può distinguere fra

eventi semantici, che fanno riferimento a quello che l'utente fa su componenti "virtuali" dell'interfaccia (premere un pulsante, selezionare la voce di un menu, ...) e

eventi low-level, ossia eventi fisici relativi al mouse o alla tastiera (tasto premuto, tasto rilasciato, mouse trascinato, ...)

Le **sorgenti** degli eventi sono i diversi componenti dell'interfaccia, come JButton, JTextField, Component, Window, ...

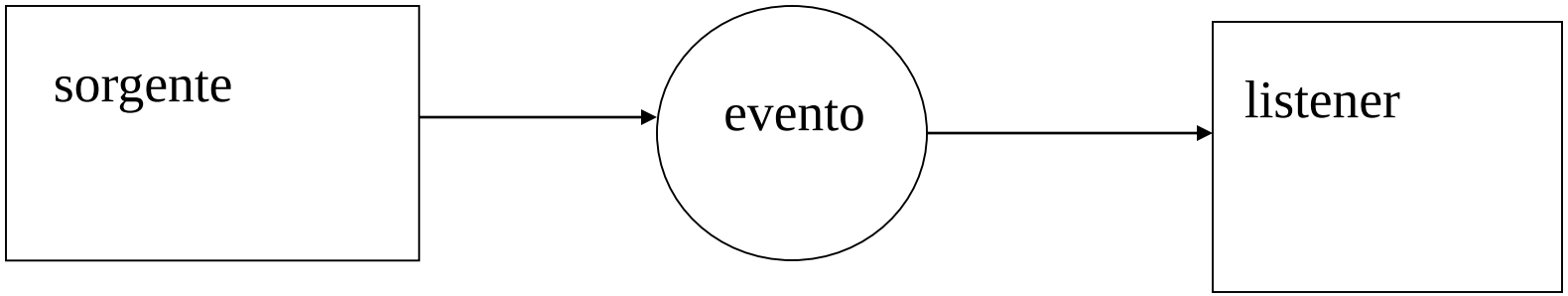
In Java un *event-handler*, chiamato **listener**, è un'istanza di una classe che contiene dei metodi per gestire gli eventi.

Per ogni tipo di evento è definita una interfaccia che il *listener* relativo deve implementare (ogni *listener* può gestire eventi di un certo tipo). Es:

- ActionListener (per eventi da bottoni)
- MouseListener (eventi del mouse)
- MouseMotionListener (spostamenti del mouse)
- WindowListener (eventi dovuti ad azioni su finestra - JFrame)
- ...

Eventi

Gli eventi sono gestiti con un meccanismo di *delega*.



La sorgente,quando genera un evento, passa un **oggetto** che descrive l'evento ad un "listener" che gestisce l'evento.

Il *listener* deve essere "registrato" presso la sorgente.

Il passaggio dell'evento causa l'invocazione di un metodo del *listener*.

Ad es. i bottoni causano un solo tipo di evento: **`ActionEvent`**.

La classe **`ActionEvent`** fornisce (fra l'altro) i metodi:

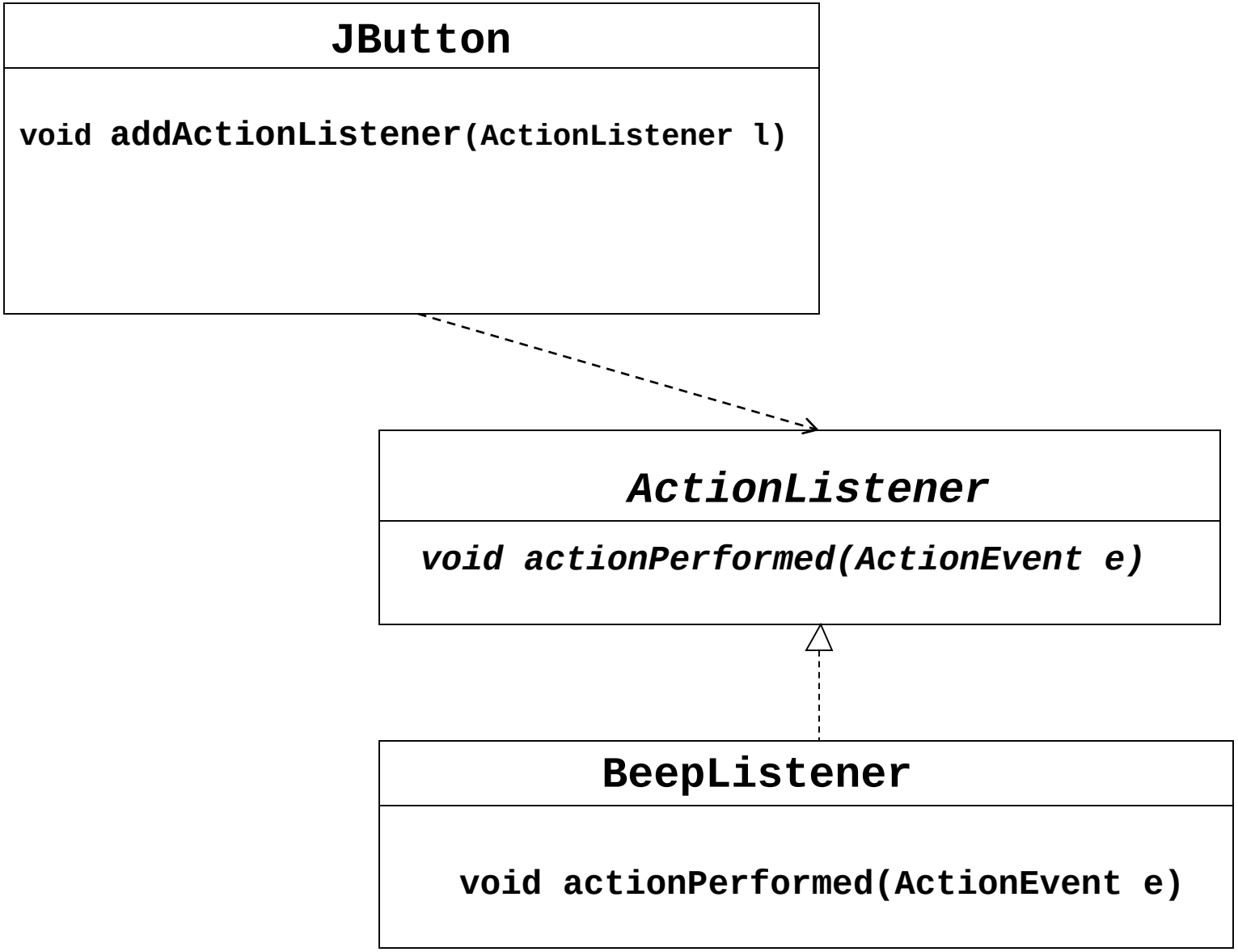
```
String getActionCommand()
Object getSource()
```

Il rispettivo *listener* deve implementare l'interfaccia

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

Per registrare l'**`ActionListener`** nel bottone, si usa il metodo della classe **`JButton`**

```
void addActionListener(ActionListener l)
```


JButton
void addActionListener(ActionListener l)
ActionListener
void actionPerformed(ActionEvent e)
BeepListener
void actionPerformed(ActionEvent e)

Per gestire un **`ActionEvent`** generato da un bottone, si deve:

- definire una classe che implementa l'interfaccia **`ActionListener`**, con il relativo metodo **`actionPerformed`**;
- creare un'istanza di questa classe;
- *registrarla* presso il bottone, eseguendo il metodo **`addActionListener`** del bottone stesso.

Ogni volta che si preme il bottone, questo chiama automaticamente il metodo **`actionPerformed`** del listener inviandogli l'evento.

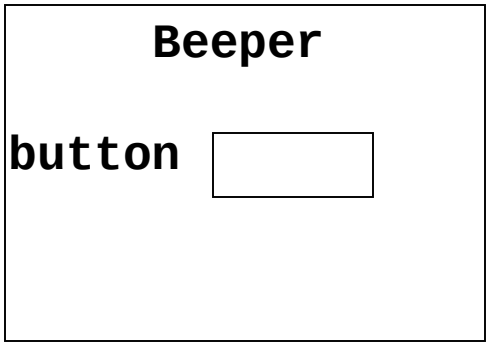
E' possibile registrare più listener nello stesso componente.

```
public class Beeper extends JFrame {
   JButton button;
   JPanel panel;

   Beeper() {
      button = new JButton("Click Me");
      panel = new JPanel();
      panel.add(button);
      add(panel);
      button.addActionListener(new BeepListener());
   }

   public static void main(String[] args) {...}
}

class BeepListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      Toolkit.getDefaultToolkit().beep();
    }
}
```

A runtime

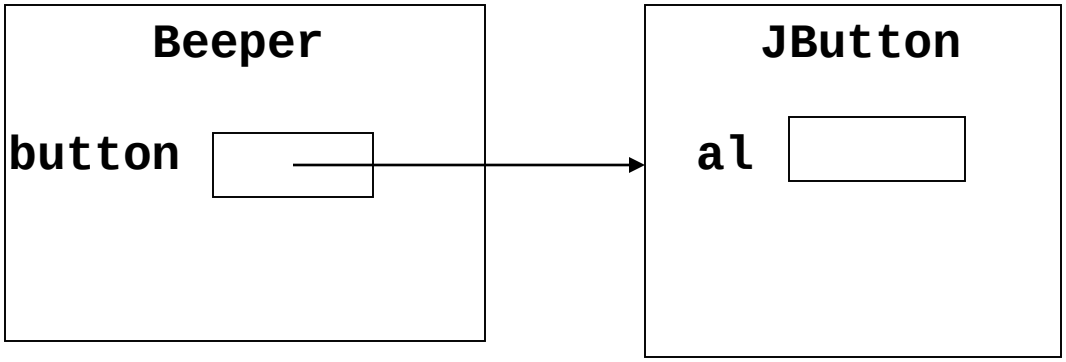
`Beeper beep = new Beeper(); (nel main)`

Oggetti creati nello HEAP



A runtime

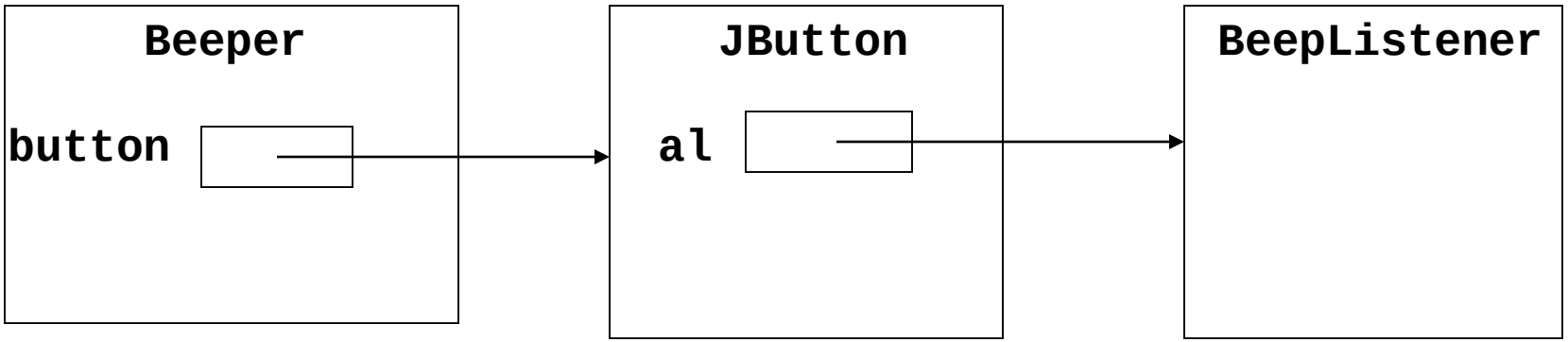
```
Beeper beep = new Beeper(); (nel main)
button = new JButton("Click Me"); (nel costruttore
                                    di Beeper)
```



Il campo **al** di **JButton** contiene un riferimento alla lista degli **ActionListener** del bottone.

A runtime

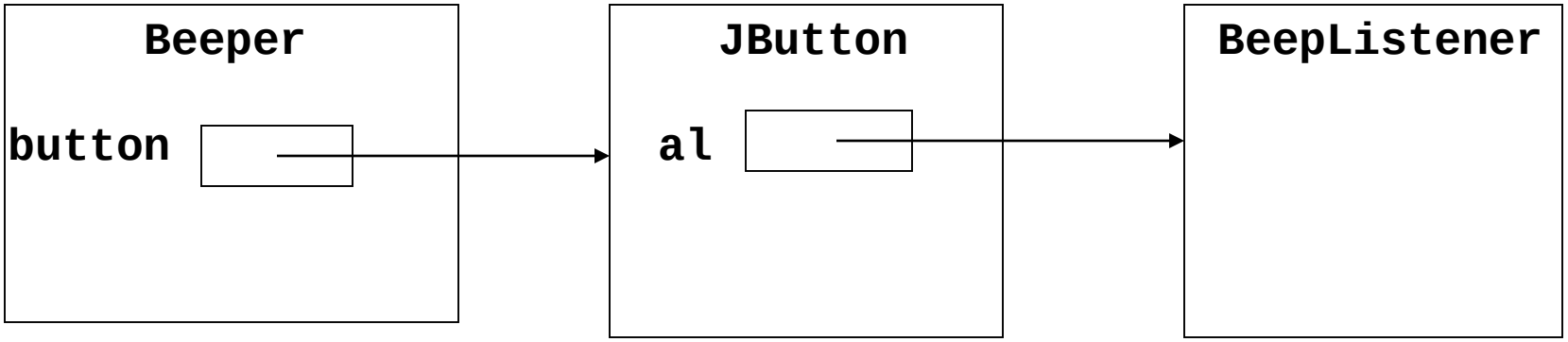
```
Beeper beep = new Beeper(); (nel main)
button = new JButton("Click Me");
                                  (nel costruttore di Beeper)
button.addActionListener(new BeepListener());
                                  (nel costruttore di Beeper)
```



Il campo **al** di **JButton** contiene un riferimento alla lista degli **ActionListener** del bottone.

A runtime

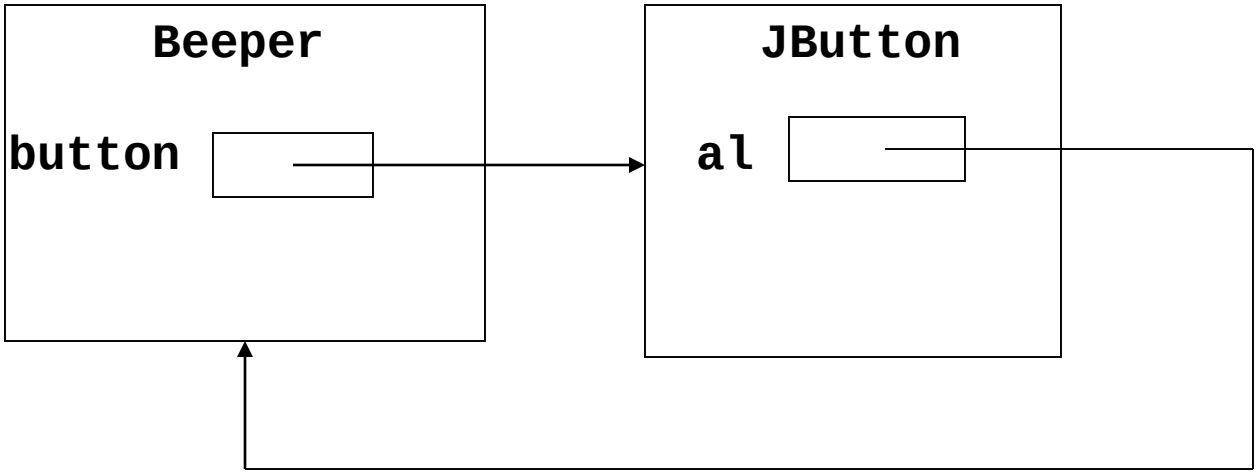
```
Beeper beep = new Beeper(); (nel main)
button = new JButton("Click Me");
                                  (nel costruttore di Beeper)
button.addActionListener(new BeepListener());
                                  (nel costruttore di Beeper)
```



Il bottone, quando viene premuto, crea un oggetto **ActionEvent** e lo passa al **BeepListener** chiamandone il metodo **actionPerformed**

```
al.actionPerformed(new ActionEvent());
                                  (in JButton)
```

NOTA L'*ActionListener* potrebbe essere implementato direttamente dal *JFrame*



Beeper estende **JFrame** e implementa **ActionListener**

```
public class Beeper extends JFrame
                        implements ActionListener {
   JButton button;
   JPanel panel;

   Beeper() {
      button = new JButton("Click Me");
      panel = new JPanel();
      panel.add(button);
      add(panel);
      button.addActionListener(this);
   }

    public void actionPerformed(ActionEvent e) {
      Toolkit.getDefaultToolkit().beep();
    }

   public static void main(String[] args) {...}
}
```

Tipi di eventi, eventHanlders e loro metodi

In generale, i nomi delle classi e delle operazioni relative agli eventi seguono un *pattern* comune.

Se **`C`** è una classe (bottone, finestra, ...), i cui oggetti possono generare eventi di tipo **`XXX`**, ci sarà:

una classe **`XXXEvent`** che implementa gli eventi;

una *interface* **`XXXListener`** con uno o più metodi per gestire l'evento;

i metodi **`addXXXListener`** o **`removeXXXListener`** nella classe **`C`**.

Classi filtro (Adapters) - I

Le interfacce di molti tipi di listener specificano un lungo elenco di metodi per gestire i vari tipi di evento che possono essere lanciati dal corrispondente tipo di sorgente. Es:

- **MouseListener:** mouseExited(MouseEvent), mousePressed(MouseEvent), mouseReleased(MouseEvent), mouseEntered(MouseEvent)

- **WindowListener:** windowClosing(WindowEvent), windowOpened(WindowEvent), windowIconified(WindowEvent), windowDeiconified(WindowEvent), windowClosed(WindowEvent), ...

Classi filtro (Adapters) - II

Il pattern di implementazione dell'interfaccia richiederebbe che il listener che voi sviluppate implementi tutti i suoi metodi, cosa che potrebbe non essere rilevante per voi (magari vi interessa gestire un solo tipo di evento)

→ Introdotte le classi filtro, o adapters, che offrono le implementazioni di default delle interfacce dei listener (con metodi che non fanno nulla).

→ Invece di implementare l'interfaccia del listener, quando non si è interessati a gestire tutti i suoi eventi si può estendere la classe adapter del listener e fare overriding dei soli metodi di gestione di eventi che ci servono

Esempio: gestione di eventi delle finestre (JFrame)

Quando si preme il pulsante di chiusura di una finestra, viene generato un **WindowEvent** che deve essere opportunamente gestito.

Ad esempio, con l'istruzione:

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

quando si specifica che quando si chiude la finestra deve terminare l'esecuzione del programma.

Però, se in chiusura di finestra volessimo fare anche altre operazioni, questa istruzione non sarebbe sufficiente. → Servirebbe un listener con opportuno metodo di gestione dell'evento.

L'evento di chiusura della finestra può essere gestito come qualunque altro evento.

Un **`JFrame`** genera un **`WindowEvent`** ogni volta che la finestra cambia stato: aperta, chiusa, ridotta ad icona, …

L'interfaccia **`WindowListener`** deve gestire tutti i possibili cambiamenti di stato della finestra, e per questo contiene sette metodi:

```
windowActivated(WindowEvent e)
windowClosing(WindowEvent e)
ecc.
```

Se a noi interessa solo il metodo **`windowClosing`**, per implementare correttamente l'interfaccia dovremmo comunque definire anche gli altri sei metodi.

Per risparmiarci la fatica, Java fornisce la classe **`WindowAdapter`**, che implementa l'interfaccia **`WindowListener`** con i sette metodi che non fanno nulla.

Noi dovremo solo estendere questa classe ridefinendo i metodi che ci interessano. Nel nostro caso solo **`windowClosing`**.

→ Overriding dei metodi che vogliamo personalizzare

→ Il polimorfismo fa eseguire i metodi da noi scritti anziché quelli vuoti dell'implementazione di default

```
public class Beeper extends JFrame {
   ...
   Beeper() {
     ...
     button.addActionListener(new BeepListener())
     addWindowListener(new WL());
   }

   public static void main(String[] args) {...}
}
class BeepListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
     Toolkit.getDefaultToolkit().beep();
   }}
class WL extends WindowAdapter {
   public void windowClosing(WindowEvent e) {
     System.exit(0); //termina l'esecuzione
   }}
```