Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

Interfacce Utente Grafiche (GUI) – parte 1
(basi con Java SWING)

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, Unix, 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**

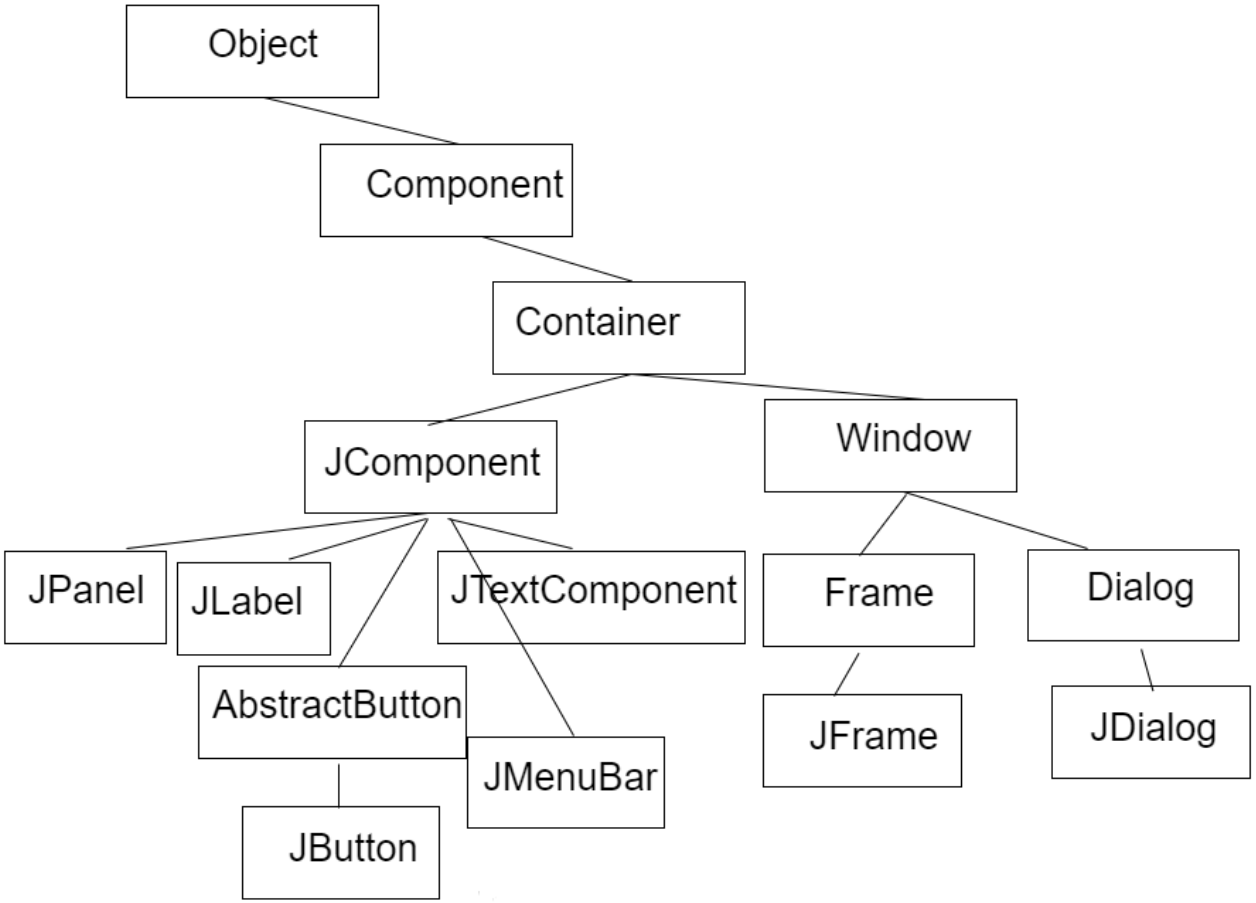
I *frame* sono dei contenitori, in cui si possono inserire altri componenti (pulsanti, testo, …) o in cui si può disegnare.

Altri contenitori sono:

JPanel

Container

Porzione della gerarchia delle classi di interfaccia grafica java



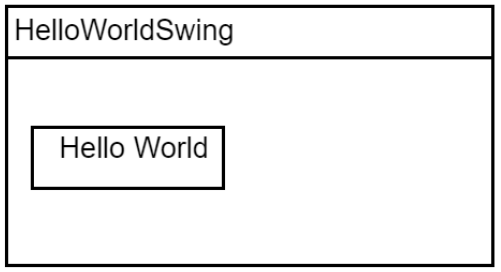
Esempio: HelloWorldSwing - I

Un **JFrame** contiene un *pannello del contenuto* (che è un **Container**) in cui si possono inserire i componenti grafici.

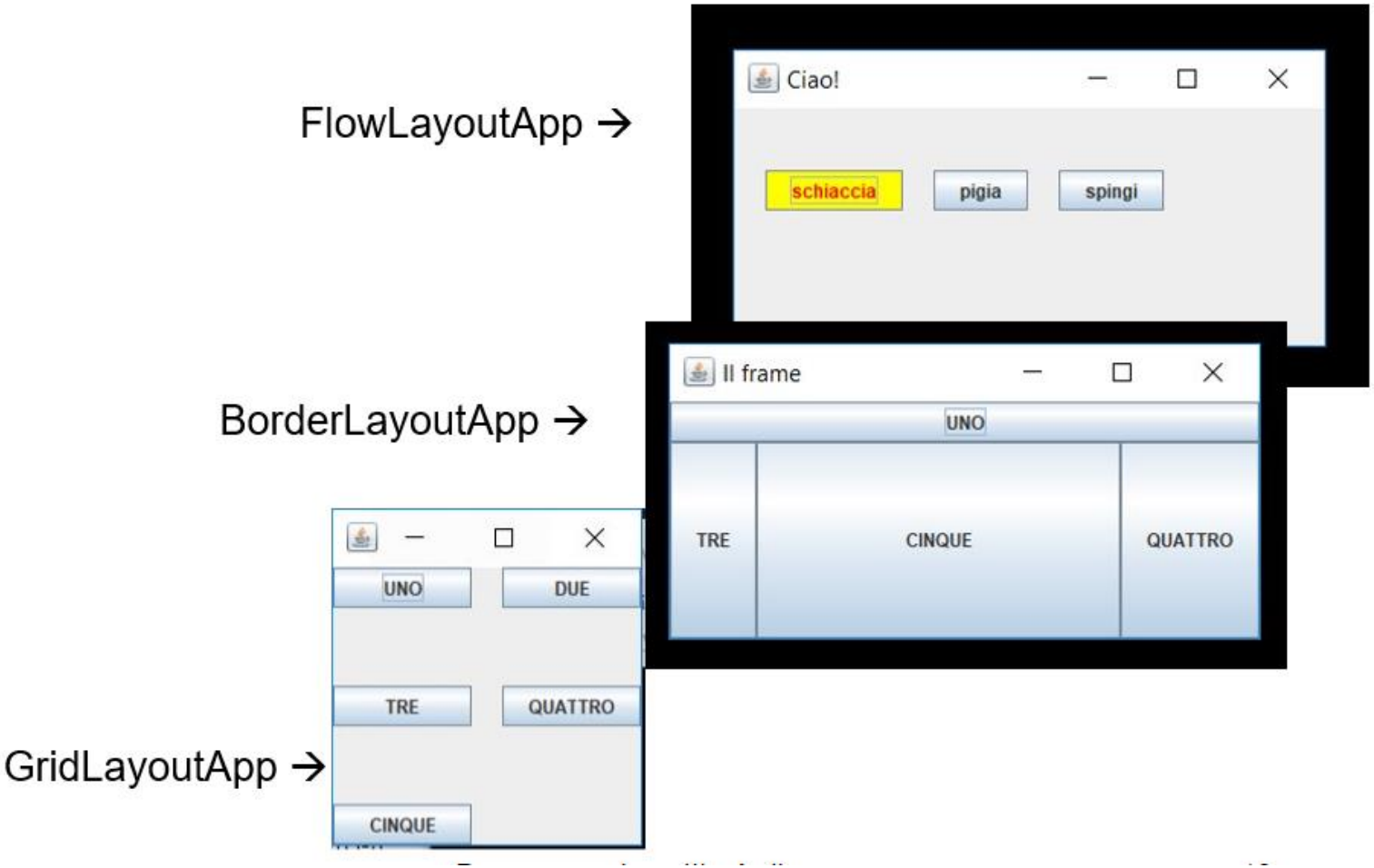
I componenti grafici vengono inseriti nel contenitore secondo un *layout* (disposizione) predefinito, che dipende dal tipo del contenitore.

Il programmatore può impostare il *layout* da utilizzare tramite opportuni comandi, che trascuriamo.

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

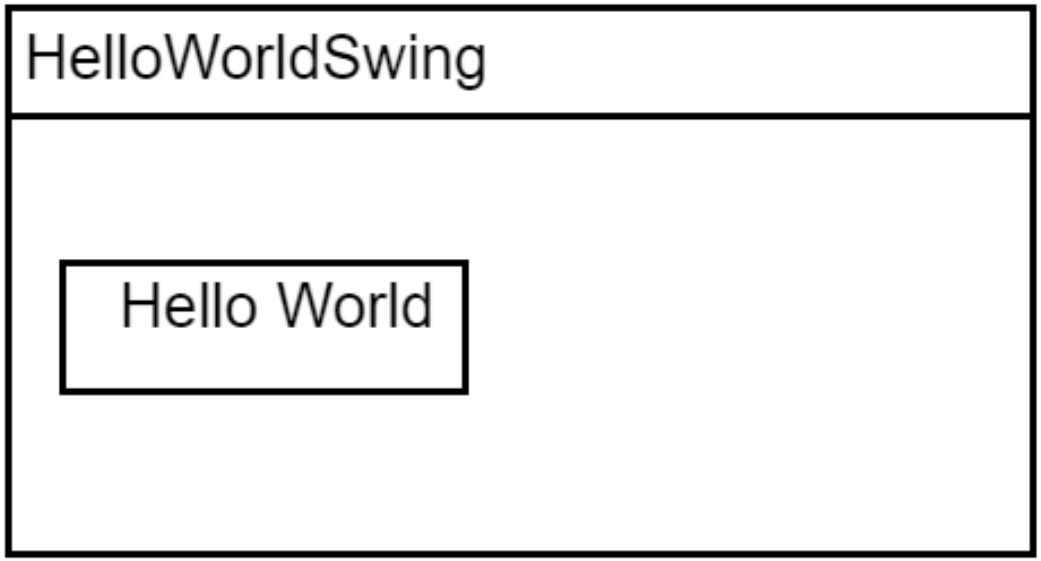


Esempi di applicazioni con layout differenti (su I-Learn):
gli elementi sono visualizzati nel pannello in modi diversi.



Esempio: HelloWorldSwing - II

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



Esempio: HelloWorldSwing - III

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

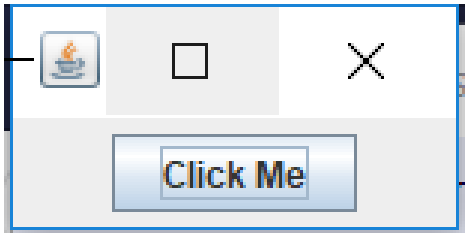
Definizione di classi di finestre specifiche

Per permettere la creazione di finestre multiple, e per organizzare il codice in modo modulare, 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"));
                ...
        }
}
```

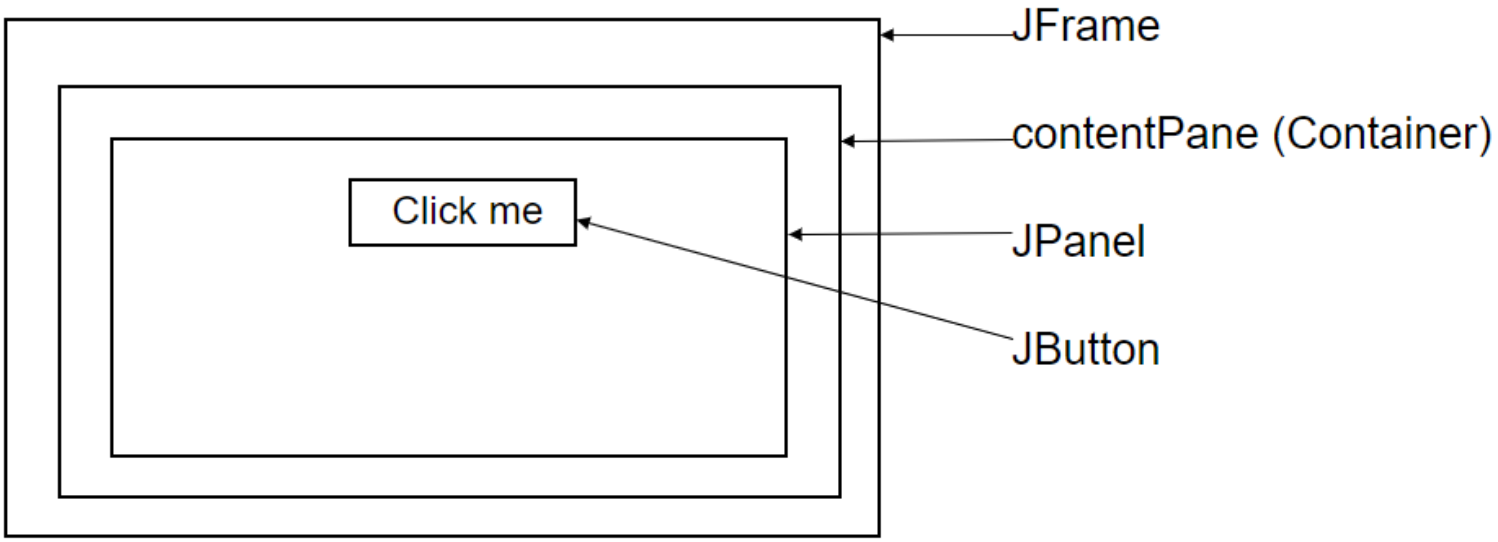
Esempio: Beeper - I

```
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);
   }
}
```



Beeper – struttura della finestra

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



Event-driven programming

Come far sì che, quando si preme il bottone «Click me» nella finestra di Beeper, si senta *beep*?

Quando si fa click 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 la programmazione guidata dagli eventi: **event-driven programming**.

Questa tecnica è adottata da molti linguaggi usati per realizzare interfacce o per programmare browser (JavaScript, ...).

Programmazione guidata dagli eventi

(event-driven programming)

I programmi tradizionali 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 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 programming)

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 tipo di evento.

Il programma deve contenere un event-handler per ogni tipo di evento da gestire. 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.

Gli eventi in un’interfaccia grafica

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

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

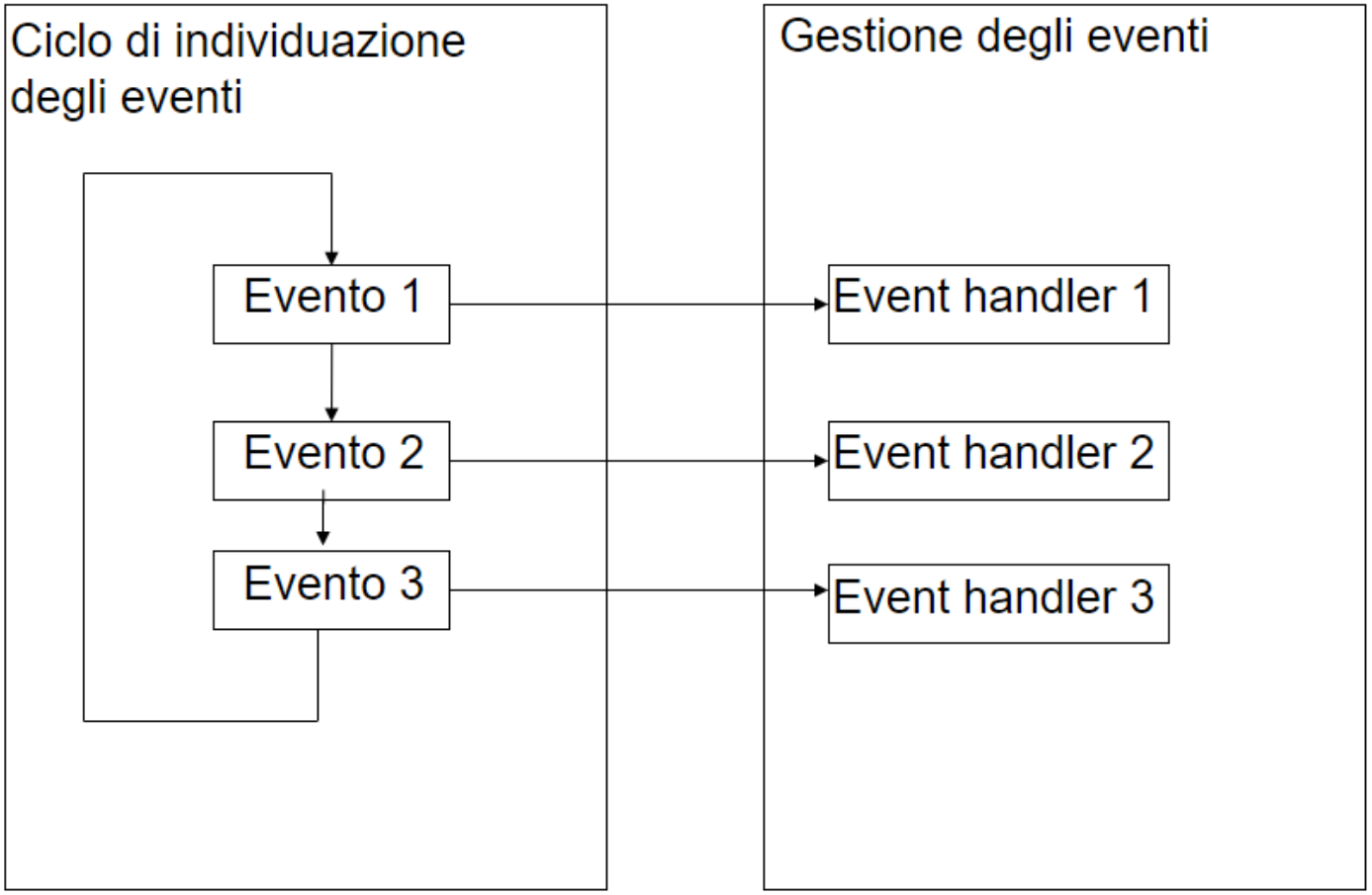
Le **sorgenti** degli eventi sono i diversi componenti dell'interfaccia, come JButton, JTextField, Component, Window, ...

Organizzazione di un’applicazione event-driven

1. L'applicazione crea gli *event-handlers*.
2. Poi, l’applicazione deve registrare gli *event handlers* presso la sorgente degli eventi. Questo significa legare ogni *event handler* a un tipo di evento che riguarda la specifica sorgente (componente della GUI).

Durante l’esecuzione dell’applicazione, 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.

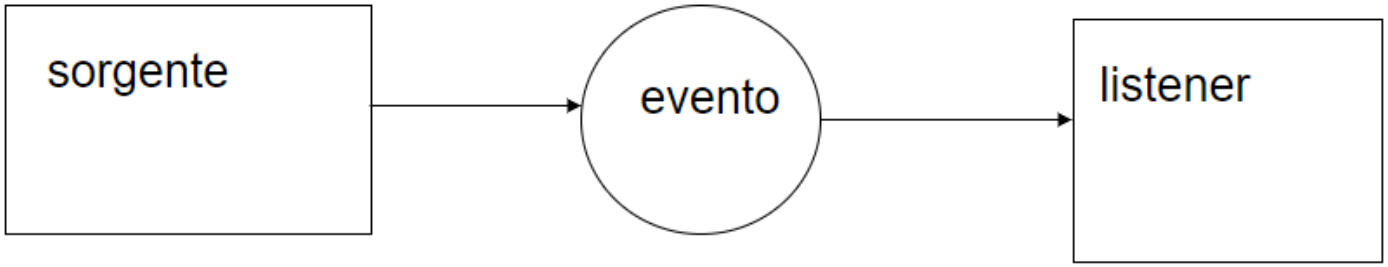
Schema di programma guidato dagli eventi



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

Gestione degli 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 per poter ricevere l’evento.

Il passaggio dell'evento causa l'invocazione di un metodo del *listener* associato a quel tipo di evento.

Event handlers (listeners)

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)
- ...

Esempio: eventi generati dai bottoni (Jbutton)

I bottoni generano un solo tipo di evento, quando vengono schiacciati: l'**ActionEvent**.

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**

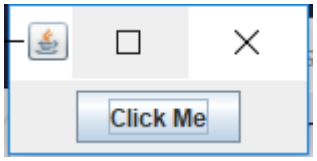
```
void addActionListener(ActionListener l)
```

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 invoca automaticamente il metodo **actionPerformed()** del listener inviandogli (passando come parametro attuale) l'evento.

È possibile registrare più listener nello stesso componente grafico.



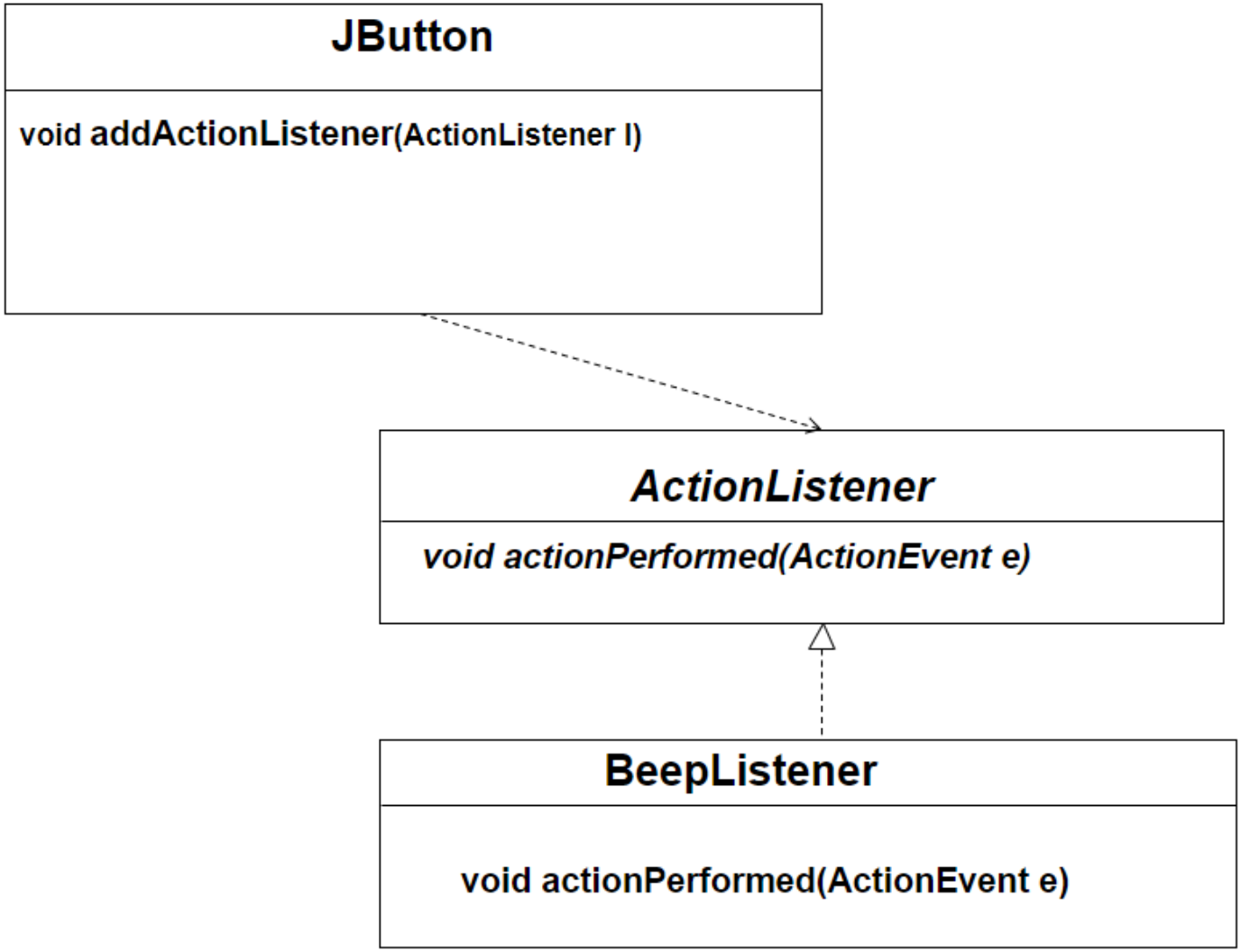
Esempio: Beeper - II

```
public class Beeper extends JFrame { //con classe interna non anonima
   JButton button;
   JPanel panel;

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

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

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

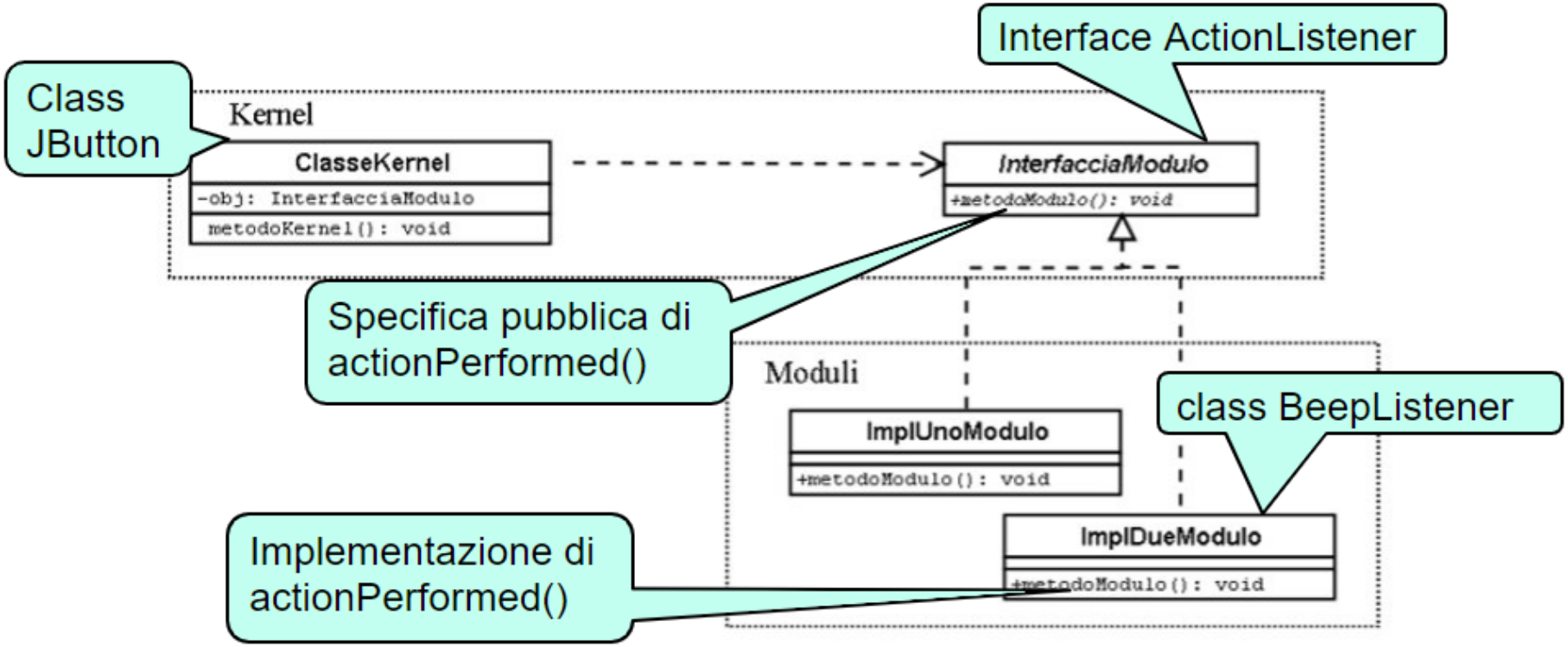
Versione alternativa, con classe anonima o lambda expression

```
public class Beeper6Lambda extends JFrame
   private JButton button; private JPanel panel;

   public Beeper6Lambda() {
      panel = new JPanel(); button = new JButton("Click Me");
      ActionListener listener = event -> {Toolkit.getDefaultToolkit().beep();
                                          System.out.println("BEEP!");};
      button.addActionListener(listener);  // con lambda expression
     /*button.addActionListener(new ActionListener() { // classe anonima
            public void actionPerformed(ActionEvent e) {
                  Toolkit.getDefaultToolkit().beep();
                  System.out.println("BEEP!"); } });*/
      panel.add(button); add(panel); pack();
   }
public static void main(String[] args) {
      Beeper6Lambda beep = new Beeper6Lambda(); beep.setVisible(true); }
}
```

Gerarchia delle classi



Ma è lo schema Kernel-Modulo!

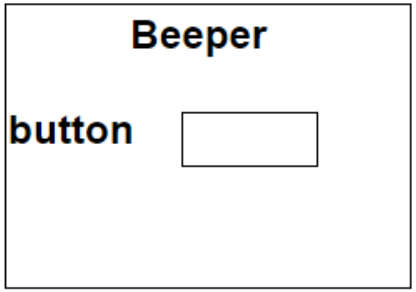


Simulazione della memoria JVM

A runtime

Beeper beep = new Beeper(); \\ istruzione del main()

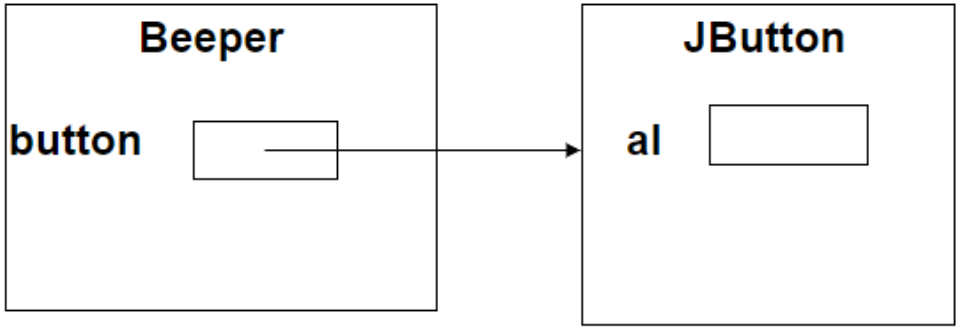
Oggetti creati nello HEAP



A runtime

JButton button = new JButton("Click Me"); \\ nel costruttore di Beeper

Oggetti creati nello HEAP

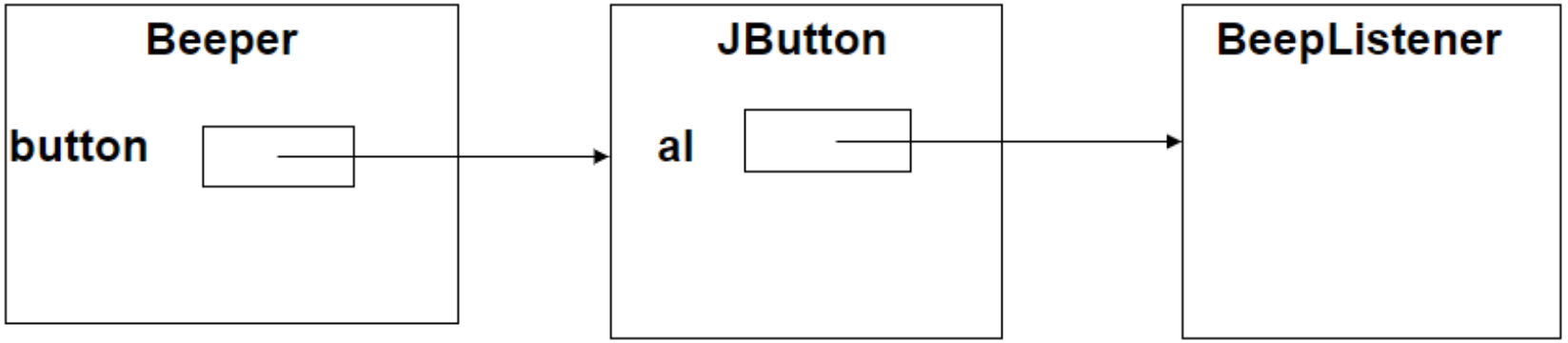


Il campo **al** di **JButton** contiene un riferimento alla lista degli **ActionListener** del bottone.

A runtime

```
JButton button = new JButton("Click Me");
button.addActionListener(new BeepListener());
```

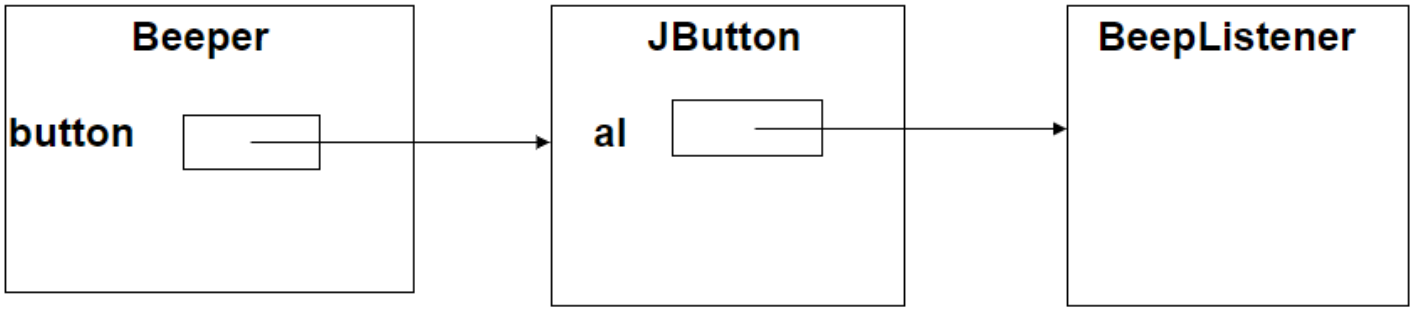
Oggetti creati nello HEAP



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

A runtime

```
JButton button = new JButton("Click Me");
button.addActionListener(new BeepListener());
```



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

al.actionPerformed(new ActionEvent());

(in JButton)

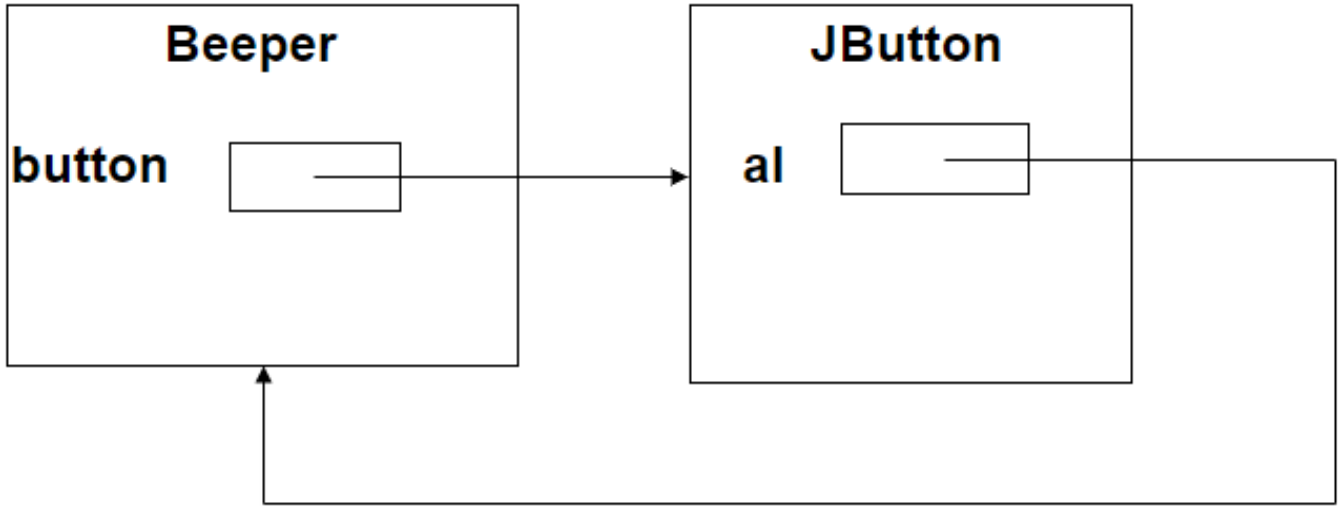
Beeper: implementazione alternativa - I

```
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); pack();
          button.addActionListener(this);
   }

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

Beeper: implementazione alternativa - II

L'*ActionListener* è implementato direttamente dal *JFrame:*

public class Beeper extends JFrame implements ActionListener {



button.addActionListener(this);

Tipi di eventi, eventHandlers 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. Questo potrebbe non essere rilevante per voi (magari vi interessa gestire un solo tipo di evento).

→ Sono state 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 a icona, …

L'interfaccia **WindowListener** deve gestire tutti i possibili cambiamenti di stato della finestra. Per questo specifica 7 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 6 metodi.

Implementazioni di default delle interface!

Java fornisce la classe **WindowAdapter**, che implementa l'interfaccia **WindowListener** con i 7 metodi che non fanno nulla (body vuoto).

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

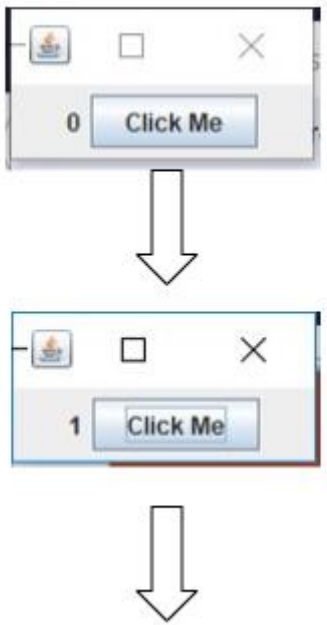
→ Overriding dei metodi che vogliamo personalizzare.

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

```
public class Beeper extends JFrame {
   ...
   Beeper() {
         JButton button ...
         button.addActionListener(new ActionListener {
                  public void actionPerformed(ActionEvent e) {
                        Toolkit.getDefaultToolkit().beep();
                  }
            });
         }
         addWindowListener(new WindowAdapter {
                  public void windowClosing(WindowEvent e) {
                        System.exit(0);
                        //termina l'esecuzione del programma
                  }
            });
   }
}
```

Beeper 1: non solo BEEP, ma anche accesso a variabili del pannello contenitore!

Supponiamo di voler contare il numero di volte che l'utente clicca il bottone Click Me → **il Listener del bottone deve gestire un contatore, da visualizzare nell'interfaccia utente**.

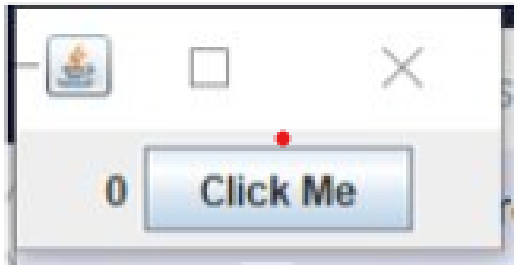


Esempio1: Beeper2 (il panel implementa l'ActionListener)

```
public class Beeper2 extends JFrame implements ActionListener {
   private JButton button = new JButton("Click Me");
   private JPanel panel = new JPanel();
   private JLabel display = new JLabel("0");
   private int i = 0;  // i è il contatore dei click

   Beeper2() {
      panel.add(display); panel.add(button); add(panel);
      button.addActionListener(this);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }

   public void actionPerformed(ActionEvent e) {
      Toolkit.getDefaultToolkit().beep();
      i++;
      display.setText(Integer.toString(i));
   }

   public static void main(String[] args) {
      Beeper2 beep = new Beeper2(); beep.pack(); beep.setVisible(true);
   }
}
```

Esempio2: ActionListener come nested classa anonima

```
public class BeeperNestedAnonima extends JFrame {

   private JButton button = new JButton("Click Me");
   private JPanel panel = new JPanel();
   private JLabel display = new JLabel("0");
   private int i = 0;

   BeeperNestedAnonima() {
         panel.add(display); panel.add(button); add(panel);
         button.addActionListener(new ActionListener() {
                                    public void actionPerformed(ActionEvent e) {
                                          Toolkit.getDefaultToolkit().beep();
                                          i++; // accede a i perché nested class
                                          display.setText(Integer.toString(i)); }
                                    });
         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         pack(); setVisible(true);
   }

   public static void main(String[] args) {
         BeeperNestedAnonima beep = new BeeperNestedAnonima();
   }
}
```



Esempio3: ActionListener come lambda expression

```
public class BeeperLambda extends JFrame {

   private JButton button = new JButton("Click Me");
   private JPanel panel = new JPanel();
   private JLabel display = new JLabel("0");
   private int i = 0;

   BeeperLambda() {
            panel.add(display); panel.add(button); add(panel);
            ActionListener listener = event -> { Toolkit.getDefaultToolkit().beep();
                                                 i++;
                                                 display.setText(Integer.toString(i)); };
            button.addActionListener(listener);
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            pack(); setVisible(true);
   }

  public static void main(String[] args) {
            BeeperLambda beep = new BeeperLambda();
 }
}
```

Ringraziamenti

Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell’Università di Torino per aver redatto la prima versione di queste slides.