Paradigmi di programmazione
(sperimentazioni)
2000-01

JAVA

Alberto Martelli

Programmazione grafica

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, Mac, …)

Nelle ultime versioni di Java è fornita la libreria **SWING**, che fa un uso molto ridotto dei toolkit nativi.

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 Swing.

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

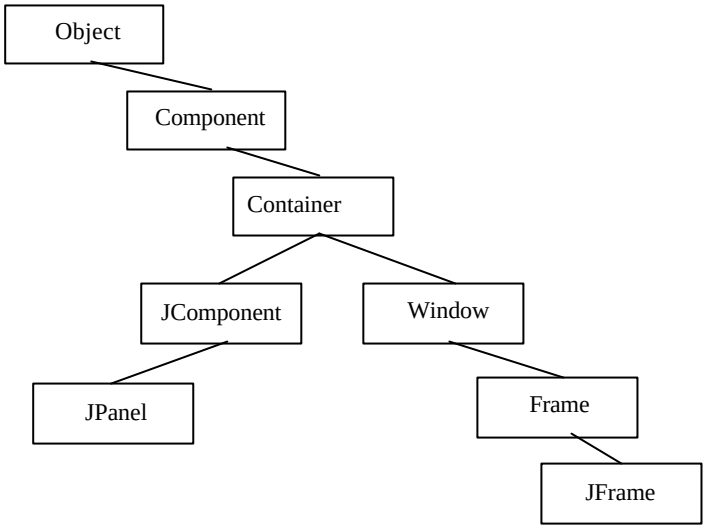
e numerosi strumenti per realizzare grafica.

Vediamo un semplice esempio di una finestra che contiene un pulsante che, quando viene premuto, fa beep.

Come si inserisce il pulsante (**JButton**) nella finestra?

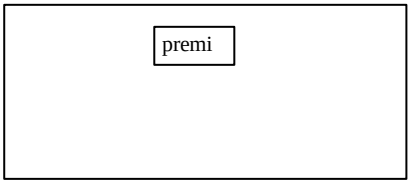
Un **JFrame** ha una struttura complessa. In particolare ha un *pannello del contenuto* (che è un **Container**) in cui si possono inserire i componenti.

Un **JFrame** fornisce il metodo **getContentPane()** per accedere al pannello del contenuto.



Per inserire un componente in un contenitore si usa il metodo **add** del contenitore. Il componente viene inserito nel contenitore secondo un *layout* predefinito. I *layout* possono essere modificati dal programmatore.

```
JPanel panel = new JPanel();
JButton pulsante = new JButton("premi");
panel.add(pulsante);
```



```
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);
       getContentPane().add(panel);
   }

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


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**.

Quando si esegue il **main** di **Beeper**, questo crea la finestra e termina l'esecuzione.

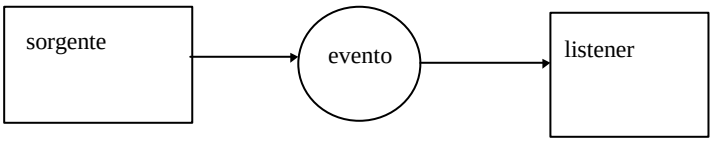
La visualizzazione della finestra attiva un **thread** di interfaccia utente che rimane attivo finché non si chiude la finestra. Se si "clicca" sul pulsante di chiusura della finestra, il thread dell'interfaccia rimane attivo.

L'interazione con l'utente è realizzata con un meccanismo ad **eventi**. Nell'esempio, gli eventi da gestire sono:

- il clic del bottone
- la chiusura della finestra (e disattivazione del thread)

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)
```


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);
       getContentPane().add(panel);
       button.addActionListener(new BeepListener());
   }

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

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


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

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

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

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

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

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`**.

Per gestire l'evento di chiusura della finestra:

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.
```

A noi interessa solo il metodo **`windowClosing`**, ma 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`**.

In generale Java fornisce una classe *Adapter* per tutti i *Listener* che hanno più di un metodo.


```
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
   }}
```

Classi annidate (nested classes)

Java dà la possibilità di definire classi all'interno di altre.

Il meccanismo può essere utile perché:

- le classi annidate possono essere nascoste.
- un oggetto di una classe annidata può accedere alla implementazione dell'oggetto della classe esterna che lo ha creato.

Nel nostro caso i *listener* sono usati solo da **`Beeper`** e possono essere nascosti al suo interno.


```
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
   }}
} //abbiamo spostato solo questa parentesi
  // la classe Beeper termina qui
```

Le classi annidate sono un fenomeno che riguarda il *compilatore*, non l'interprete del bytecode.

Le classi annidate sono tradotte dal compilatore in file **.class** separati, e l'interprete non ha conoscenze specifiche al loro riguardo.

Nel nostro esempio, la compilazione di **`Beeper`** produce tre file:

```
Beeper.class
Beeper$BeepListener.class
Beeper$WL.class
```

Se una classe è usata una sola volta si può evitare di darle un nome, usando la notazione delle **classi anonime**.


```
public class Beeper extends JFrame {
   .....
   Beeper() {
      button = new JButton("Click Me");
      panel = new JPanel();
      panel.add(button);
      getContentPane().add(panel);
      button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                  Toolkit.getDefaultToolkit().beep();}
      });

      addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                  System.exit(0);}
      });
      }
      public static void main(String[] args) {...}
}
```

Per mostrare l'utilità delle classi annidate, estendiamo l'esempio con un campo in cui inserire il numero di volte che si è "cliccato" sul bottone.

Il campo può essere realizzato con un componente **JLabel**, che può contenere una figura o una stringa (nel nostro caso una stringa che rappresenta un numero).

L'**ActionListener** collegato al bottone può modificare direttamente il contenuto di questo componente.

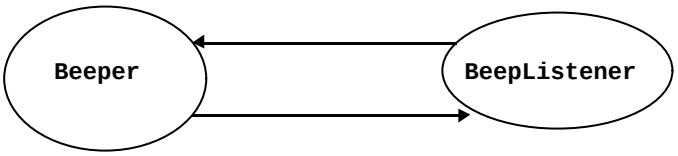

```
public class Beeper extends JFrame {
   private JButton button = new JButton("Click Me");
   private JPanel panel = new JPanel();
   private JLabel display = new JLabel("0");
   private int i = 0;

   Beeper() {
       panel.add(display);
       panel.add(button);
       getContentPane().add(panel);
       button.addActionListener(new BeepListener());
       addWindowListener(new WL());}

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

Il **`BeepListener`** può accedere alle variabili **`display`** e **`i`** del **`Beeper`**.

Se la classe **`BeepListener`** fosse definita esternamente a **`Beeper`**, gli oggetti **`BeepListener`** avrebbero bisogno di usare un riferimento esplicito all'oggetto **`Beeper`** su cui devono operare. Inoltre non potrebbero accedere alle variabili **`display`** e **`i`** perché queste sono dichiarate **`private`**.



Layout

Java fornisce diversi manager di layout predefiniti, fra cui:

- **FlowLayout** - i componenti vengono inseriti uno dopo l'altro e riga per riga.
- **BorderLayout** - i componenti vengono inseriti in posizione nord, sud, est, ovest e centro.
- **GridLayout** - i componenti vengono disposti in una tabella.
- **BoxLayout** - i componenti vengono disposti uno per riga.

Ogni contenitore ha un layout predefinito.

Ad esempio per il *JPanel* è *FlowLayout*.

E' possibile cambiare il layout con il metodo *setLayout*.

```
JPanel p = new JPanel();
p.setLayout(new GridLayout(4,4));
```


Applet

Applet: speciali programmi Java che un browser può scaricare dalla rete ed eseguire. La chiamata dell'*applet* viene inserita in un documento HTML.

(Non sono uno strumento per realizzare pagine web)

Attualmente l'interesse è diminuito perché HTML ed i linguaggi di scripting sono molto potenziati rispetto ai primi tempi di Java.

I principali browser non supportano più le ultime versioni di Java. Occorre installare dei plug-in.

Uso vantaggioso soprattutto in applicazioni *intranet* aziendali.

Come trasformare una applicazione grafica in un'applet

- Sostituire *JFrame* con *JApplet.*
- Eliminare il *main.*
- Eliminare i *listener* della finestra.
- Trasformare il costruttore nel metodo *init*().
- Creare una pagina HTML con un *tag* per caricare il codice dell'*applet.*


```
public class BeeperApplet extends JApplet {
   JButton button;
   JPanel panel;

   public void init() {
      button = new JButton("Click Me");
      panel = new JPanel();
      panel.add(button);
      getContentPane().add(panel);
      button.addActionListener(new BeepListener());
   }
}

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

File HTML:

```
<HTML>
<TITLE>Beeper</TITLE>
<BODY>
<APPLET  CODE = "BeeperApplet.class"
         WIDTH = 100 HEIGHT = 50 >
</APPLET>
</BODY>
</HTML>
```

Purtroppo il browser non riesce ad eseguire direttamente questo file. Occorre trasformarlo in un file html più complesso, usando un traduttore fornito dalla SUN, che inserisce degli script per caricare i plug-in.

Sicurezza delle applet

Le applet (diversamente dalle applicazioni) sono limitate nelle operazioni che possono svolgere.

- Non possono avviare programmi locali.
- Non possono comunicare con host diversi da quello da cui sono state prelevate.
- Non possono accedere al file system locale.
- Non possono recuperare informazioni riguardanti il computer locale.

I controlli sono effettuati dall'interprete della Java Virtual Machine in base ad un modello predefinito di *gestione della sicurezza*.

Le applet possono essere *firmate*.

Se chi firma l'applet è una persona di fiducia, è possibile concedere più privilegi all'applet modificando la politica di gestione della sicurezza. Ad esempio si può concedere ad una applet di una *intranet* aziendale di accedere ai file locali.

I principali metodi delle applet

Per gestire il ciclo di vita di un'applet, la classe JApplet contiene i seguenti quattro metodi (che naturalmente possono essere ridefiniti).

init() - viene chiamato quando viene creata l'applet (come un costruttore)

stop() - viene chiamato tutte le volte che si esce dalla pagina contenente l'applet. Interrompe le attività costose computazionalmente, come animazioni o calcoli complessi.

start() - viene chiamato dopo *init* e tutte le volte che si rientra nella pagina contenente l'applet. Riavvia le attività bloccate dalla *stop*.

destroy() - viene chiamato quando si chiude il browser.