Tecniche di Programmazione avanzata

Corso di Laurea Specialistica in Ingegneria Telematica
Università Kore – Enna – A.A. 2008-2009

Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it

Programmazione GUI

Programmazione GUI

- Nella programmazione di un'interfaccia grafica, numerosi sono gli aspetti da prendere in considerazione:
  - Quali elementi grafici sono disponibili
  - Gerarchie di contenimento degli elementi grafici
  - Modelli degli elementi grafici
  - Bitmap vs vettoriale
  - Thread a supporto della GUI
  - Layout

Java e la grafica

- L'architettura Java è graphics-ready:
- Package java.awt
  - il primo package grafico (Java 1.0)
  - indipendente dalla piattaforma... o quasi!
- Package javax.swing
  - Il nuovo package grafico (Java 2; preliminare da Java 1.1.6)
  - scritto esso stesso in Java, realmente indipendente dalla piattaforma
- Swing definisce una gerarchia di classi che forniscono ogni tipo di componente grafico: finestre, pannelli, frame, bottoni, aree di testo, checkbox, liste a discesa, ecc.
- Programmazione "event-driven":
  - non più algoritmi stile input/elaborazione/output, ma reazione agli eventi che l'utente, in modo interattivo, genera sui componenti grafici
  - ascoltatore degli eventi
- Si può considerare un paradigma di programmazione a sé stante

Java e la grafica: JFC

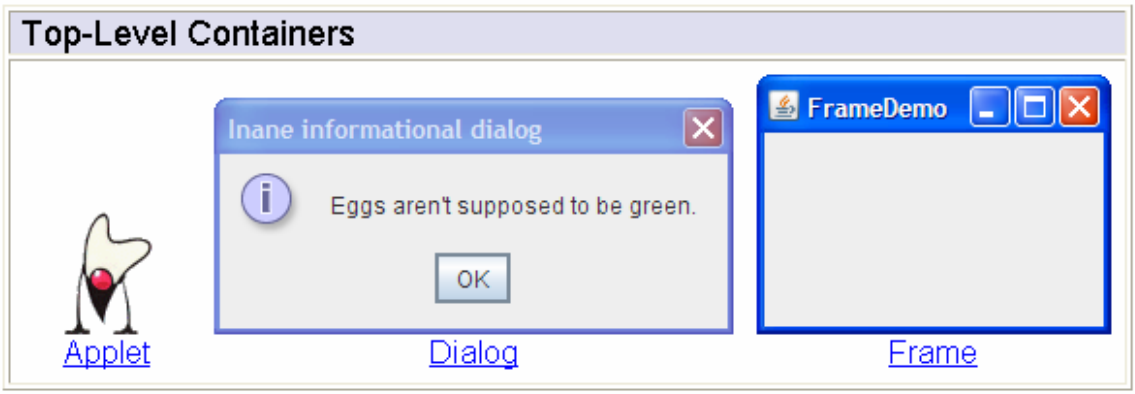
Features of the Java Foundation Classes	
Swing GUI Components	Includes everything from buttons to split panes to tables.
Pluggable Look-and-Feel Support	Gives any program that uses Swing components a choice of look and feel. For example, the same program can use either the Java or the Windows look and feel. As of v1.4.2, the Java platform supports the GTK+ look and feel, which makes hundreds of existing look and feels available to Swing programs.
Accessibility API	Enables assistive technologies, such as screen readers and Braille displays, to get information from the user interface.
Java 2D API	Enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and applets. Java 2D includes extensive APIs for generating and sending high-quality output to printing devices.

Java e la grafica: JFC

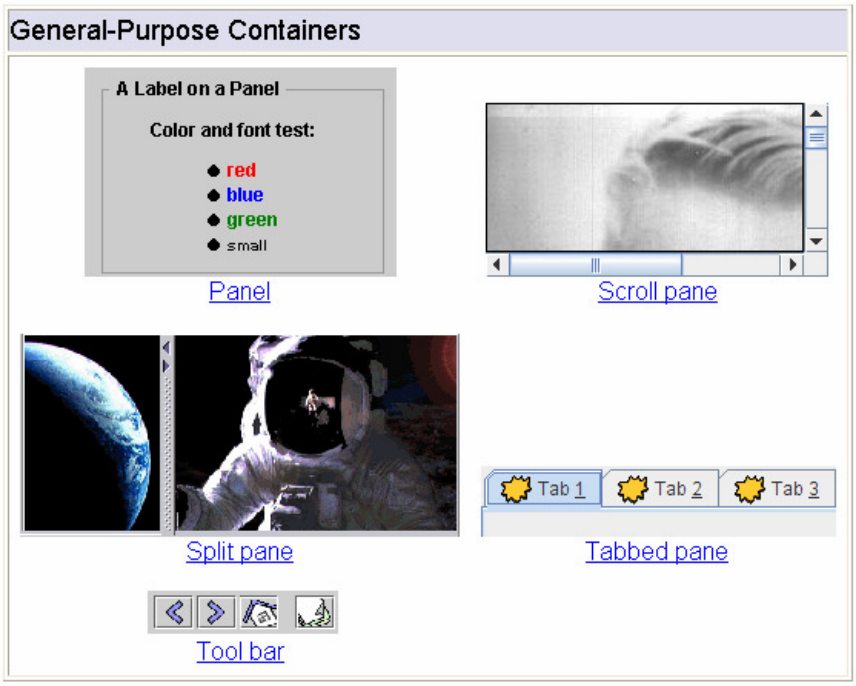
Features of the Java Foundation Classes	
Drag-and-Drop Support	Provides the ability to drag and drop between Java applications and native applications.
I18N	Allows developers to build applications that can interact with users worldwide in their own languages and cultural conventions. With the input method framework developers can build applications that accept text in languages that use thousands of different characters, such as Japanese, Chinese, or Korean.

- In release 1.4 of the Java platform, the Swing API has 17 public packages
- Fortunately, most programs use only a small subset of the API:
- javax.swing
- javax.swing.event (not always required)

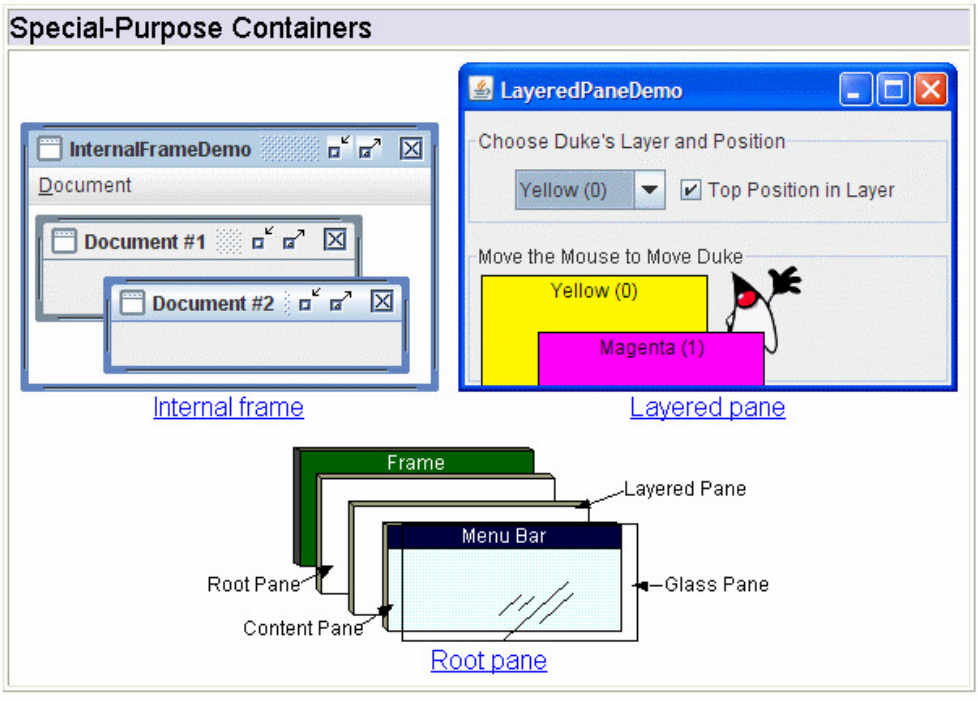
JFC Swing: panoramica



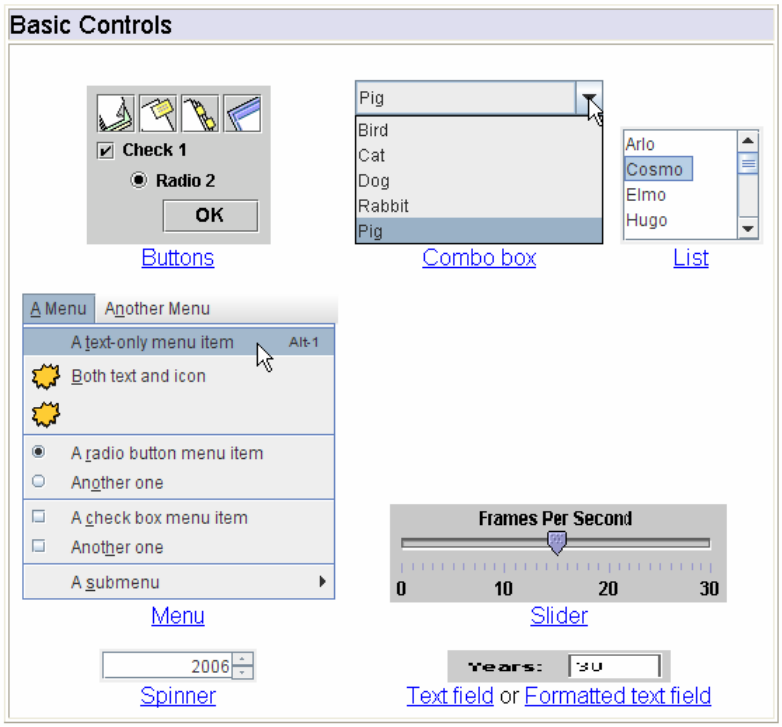
JFC Swing: panoramica



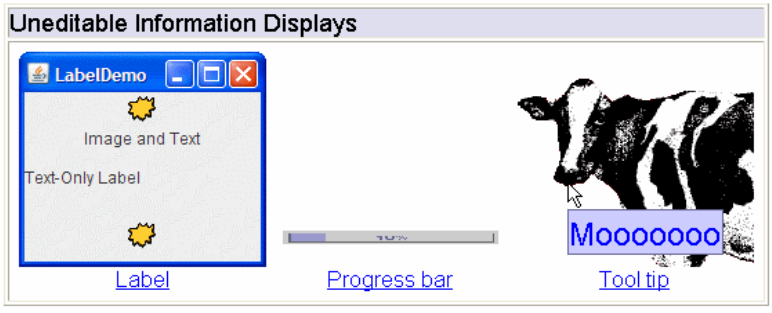
JFC Swing: panoramica

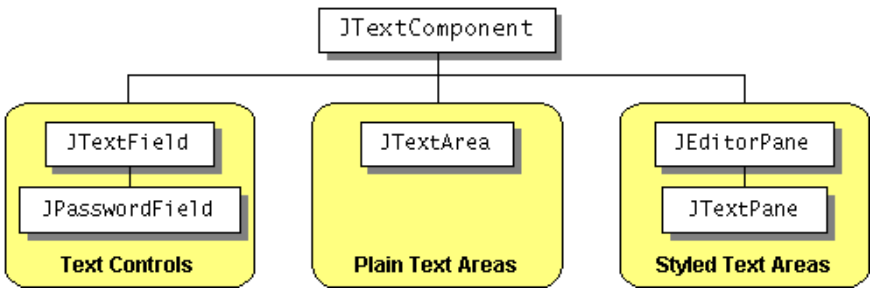


JFC Swing: panoramica

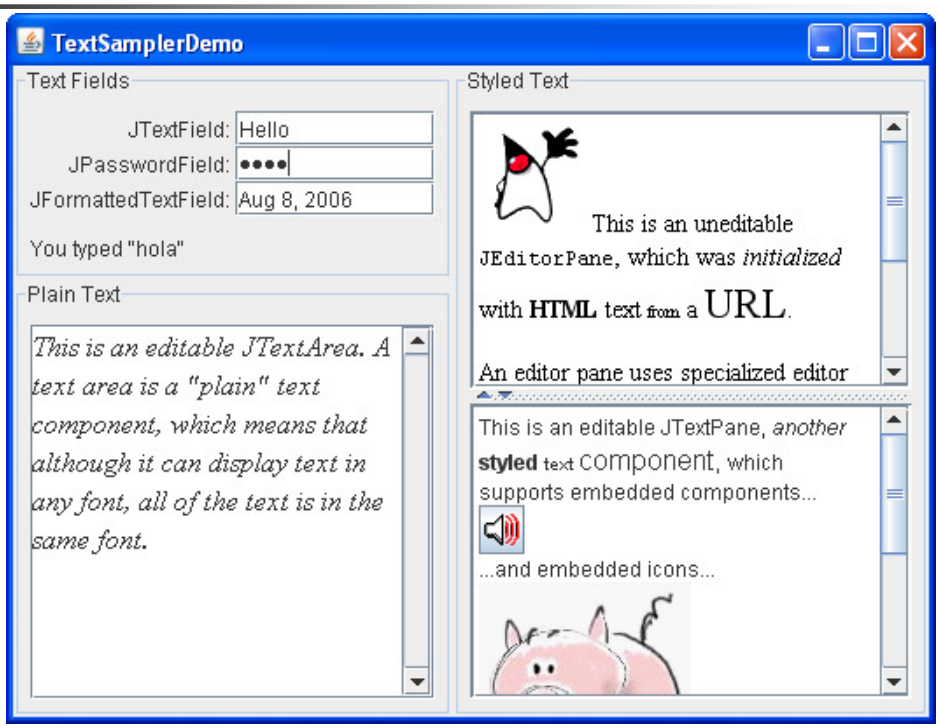


JFC Swing: panoramica

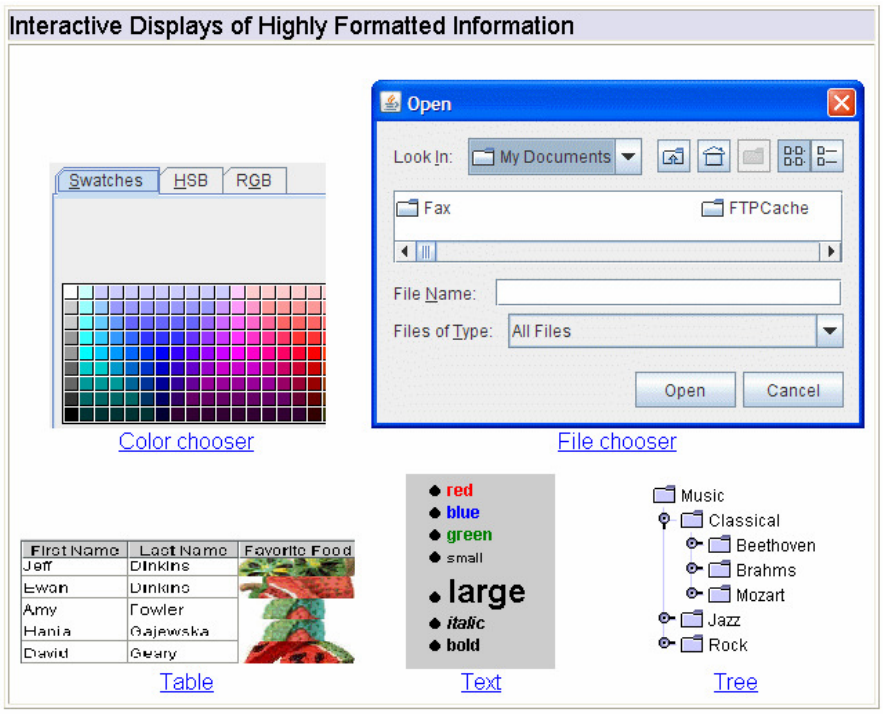




JFC Swing: panoramica



JFC Swing: panoramica

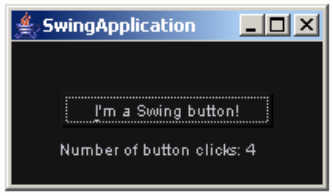


JFC Swing: panoramica

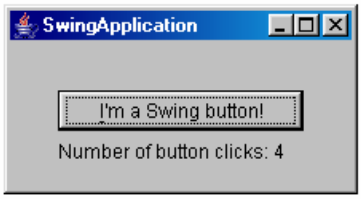
Look and Feel

The following screenshots show the GUI of the `SwingApplication`, each one with a different look and feel.

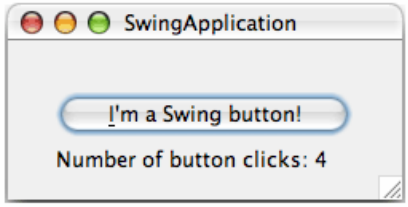
Java look and feel



GTK+ look and feel



Windows look and feel



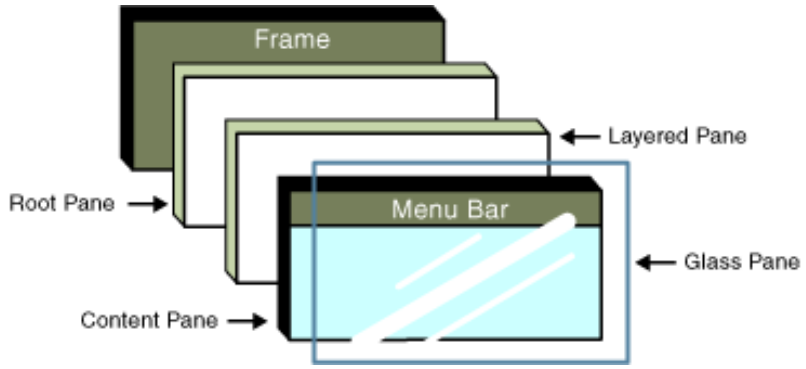
Mac OS look and feel

JFC Swing: panoramica

- Swing provides three generally useful top-level container classes: JFrame, JDialog, and JApplet. When using these classes, you should **keep these facts in mind**:
- To appear on the screen, every GUI component must be part of a **containment hierarchy**. A containment hierarchy is a tree of components that has a top-level container as its root.
- **Each GUI component can be contained only once**. If a component is already in a container and you try to add it to one another, it is removed from the first and added to the second one.
- Each top-level container has a **content pane** that contains the visible components in that container's GUI.
- You can optionally add a **menu bar** to a top-level container. The menu bar is by convention positioned within the top-level container, but outside the content pane. Some look and feels, such as the Mac OS look and feel, give you the option of placing the menu bar in another place more appropriate for the look and feel, such as at the top of the screen.

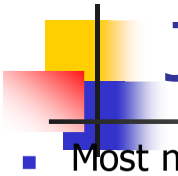
JFC Swing: panoramica

- Each **top-level container** relies on a reclusive intermediate container called the **root pane**. The root pane manages the content pane and the menu bar, along with a couple of other containers. You generally don't need to know about root panes to use Swing components. However, if you ever need to intercept mouse clicks or paint over multiple components, you should get acquainted with root panes. A root pane consists of:



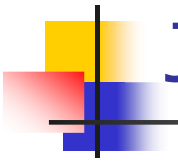
JFC Swing: panoramica

- **The glass pane**
  - Hidden, by default. If you make the glass pane visible, then it's like a sheet of glass over all the other parts of the root pane. It's completely transparent unless you implement the glass pane's paintComponent method so that it does something, and it intercepts input events for the root pane.
- **The layered pane**
  - Serves to position its contents, which consist of the content pane and the optional menu bar. Can also hold other components in a specified Z order.
- **The content pane**
  - The container of the root pane's visible components, excluding the menu bar.
- **The optional menu bar**
  - The home for the root pane's container's menus. If the container has a menu bar, you generally use the container's setJMenuBar method to put the menu bar in the appropriate place.
- Although the example uses a JFrame, the same concepts apply to JApplets and JDialogs.



JFC Swing: panoramica

- Most noncontainer Swing components have **models**. A button (JButton), for example, has a model (a ButtonModel object) that stores the button's state — what its keyboard mnemonic is, whether it's enabled, selected, or pressed, and so on. Some components have multiple models. A list (JList), for example, uses a ListModel to hold the list's contents, and a ListSelectionModel to track the list's current selection. You often don't need to know about the models that a component uses. For example, programs that use buttons usually deal directly with the JButton object, and don't deal at all with the ButtonModel object.
- Why then do models exist? The biggest reason is that **they give you flexibility in determining how data is stored and retrieved**. For example, if you're designing a spreadsheet application that displays data in a sparsely populated table, you can create your own table model that is optimized for such use.
- Models have other benefits, too. **Models automatically propagate changes to all interested listeners, making it easy for the GUI to stay in sync with the data.** For example, to add items to a list you can invoke methods on the list model. When the model's data changes, the model fires events to the JList and any other listeners, and the GUI is updated accordingly.



JFC Swing: panoramica

- Except for top-level containers, all Swing components whose names begin with "J" descend from the **JComponent class**.
- The JComponent class extends the Container class, which itself extends Component. The Component class includes everything from providing layout hints to supporting painting and events. The Container class has support for adding components to the container and laying them out.
- The JComponent class provides the following functionality :
  - Tool tips (string for mouse over component)
  - Painting inside a component and borders management
  - Application-wide pluggable look and feel
  - Custom properties can be defined
  - Support for layout
  - Support for accessibility
  - Support for drag and drop
  - Double buffering (provide smooths on-screen painting)
  - Key bindings (provide reactions when keyboard is used)

Frame

- La più **semplice applicazione grafica** consiste in una classe il cui main crea un JFrame e lo rende visibile col metodo show():

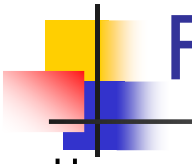
```
import java.awt.*; import javax.swing.*;
public class EsSwing1 {
    public static void main(String[] v){
    JFrame f = new JFrame("Esempio 1");
    f.show(); //mostra il JFrame    } }
```

- I comandi standard delle finestre sono già attivi
- la chiusura nasconde soltanto il frame. Per chiuderlo serve Ctrl+C
- Per impostare le dimensioni di un qualunque contenitore si usa setSize(), che ha come parametro un opportuno oggetto di classe Dimension:
- f.setSize(new Dimension(300,150)); // le misure x,y sono in pixel
- Inoltre, la finestra viene visualizzata nell'angolo superiore sinistro dello schermo Per impostare la posizione di un qualunque contenitore si usa setLocation():
- f.setLocation(200,100)); // (0,0) = angolo superiore sinistro

Frame

- Posizione e dimensioni si possono anche fissare insieme, col metodo setBounds()
- esempio di finestra già dimensionata e collocata nel punto previsto:

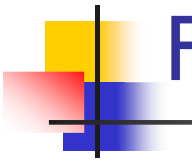
```
import java.awt.*;
import javax.swing.*;
public class EsSwing1 {
    public static void main(String[] v){
      JFrame f = new JFrame("Esempio 1");
      f.setBounds(200,100, 300,150)
              f.show();
    }
}
```



Frame

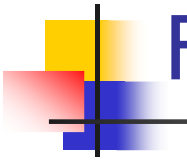
- Un approccio efficace consiste nell'**estendere JFrame**, definendo una nuova classe:

```
import java.awt.*;
import javax.swing.*;
public class MyFrame extends JFrame {
   public MyFrame(String titolo){
     super(titolo);
     setBounds(200,100,300,150);
   }
}
   import java.awt.*; import javax.swing.*;
   public class EsSwing2 {
   public static void main(String[] v){
     MyFrame f = new MyFrame("Esempio 2");
     f.show();
     }       }
```

Frame

- In Swing non si possono aggiungere nuovi componenti direttamente al Jframe
- Però dentro ogni JFrame c'è un Container, recuperabile col metodo getContentPane(): è a lui che vanno aggiunti i nuovi componenti
- Tipicamente, si aggiunge un pannello (un JPanel o una nostra versione più specifica), tramite il metodo add()
- sul pannello si può disegnare (forme, immagini...) o aggiungere pulsanti, etichette, icone, (cioè aggiungere altri componenti)

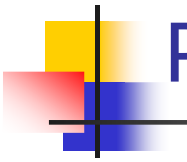


Frame & Panel

- **Aggiunta di un pannello** al Container di un frame, tramite l'uso di getContentPane():

```
import java.awt.*;
import javax.swing.*;
public class EsSwing3 {
        public static void main(String[] v){
                MyFrame f = new MyFrame("Esempio 3");
                Container c = f.getContentPane();
                JPanel panel = new JPanel();
                c.add(panel);
                f.show();
        }
}
```

- NOTA: non abbiamo disegnato niente, né aggiunto componenti, sul pannello! Però, avendo, il pannello, potremmo usarlo per disegnare e inserire altri componenti!

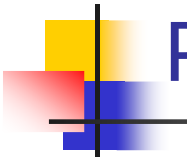


Panel

- Per **disegnare su un pannello** occorre:
  - definire una propria classe (MyPanel) che estenda JPanel
  - ridefinire paintComponent(), che è il metodo (ereditato da JComponent) che si occupa di disegnare il componente
- ATTENZIONE: il nuovo paintComponent() deve sempre richiamare il metodo paintComponent() originale, tramite super
- Il nostro pannello personalizzato:

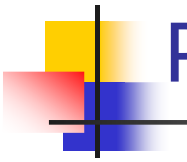
```
public class MyPanel extends JPanel {
    // nessun costruttore, va bene il default
    public void paintComponent(Graphics g){
            super.paintComponent(g);
            // qui aggiungeremo le nostre istruzioni di disegno…
    }
}
```

- Graphics g, di cui non ci dobbiamo occupare esplicitamente, è l'oggetto del sistema che effettivamente disegna ciò che gli ordiniamo



Panel

- Quali **metodi per disegnare**?
  - drawImage(), drawLine(), drawRect(), drawRoundRect(), draw3DRect(), drawOval(), drawArc(), drawString(), drawPolygon(), drawPolyLine()
  - fillRect(), fillRoundRect(), fill3DRect(), fillOval(), fillArc(), fillPolygon(), fillPolyLine()
  - getColor(), getFont(), setColor(), setFont(), copyArea(), clearRect()



Panel

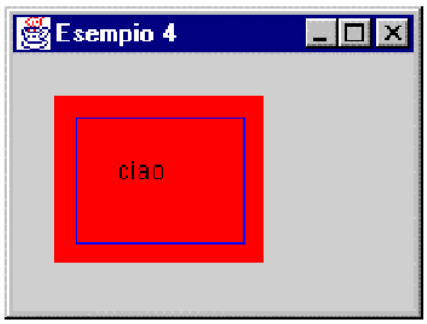
- Il pannello personalizzato con il disegno:

```
import java.awt.*;
import javax.swing.*;
public class MyPanel extends JPanel  {
   public void paintComponent(Graphics g){
     super.paintComponent(g);
     g.setColor(Color.red);      // white, gray, lightGray
     g.fillRect(20,20, 100,80);
     g.setColor(Color.blue);
     g.drawRect(30,30, 80,60);
     g.setColor(Color.black);
     g.drawString("ciao",50,60);
   }
}
```

Panel

- Il main che lo crea e lo inserisce nel frame:

```
import java.awt.*;
import javax.swing.*;
public class EsSwing4 {
   public static void main(String[] v){
     MyFrame f = new MyFrame("Esempio 4");
     // potremmo anche usare un JFrame standard…
     Container c = f.getContentPane();
     MyPanel panel = new MyPanel();
     c.add(panel);
     f.show();
   }
}
```



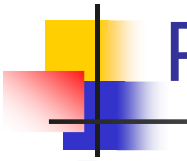
Panel

- Per cambiare font, si crea un oggetto Font appropriato, lo si imposta come font predefinito usando il metodo setFont()

```
Font f1 = new Font("Times", Font.BOLD, 20);
// nome del font, stile, dimensione in punti
// stili possibili: Font.PLAIN, Font.ITALIC
g.setFont(f1);
```

- Recuperare le proprietà di un font: il font corrente si recupera con getFont(), mentre le sue proprietà si recuperano con getName(), getStyle(), getSize() e si verificano con i predicati isPlain(), isBold(), isItalic():

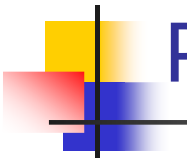
```
Font f1 = g.getFont();
int size = f1.getSize();
int style = f1.getStyle();
String name = f1.getName();
```



Panel

- **Esempio di grafico di una funzione**:
- occorre creare un'apposita classe FunctionPanel che estenda JPanel, ridefinendo il metodo paintComponent() come appropriato, ad esempio:
  - sfondo bianco, cornice nera
  - assi cartesiani rossi, con estremi indicati
  - funzione disegnata in blu
  - creare, nel main, un oggetto di tipo FunctionPanel
- Definizione del solito main:

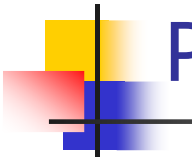
```
import java.awt.*;
import javax.swing.*;
public class EsSwing5 {
   public static void main(String[] v){
     JFrame f = new JFrame("Grafico f(x)");
     Container c = f.getContentPane();
     FunctionPanel p = new FunctionPanel(); c.add(p);
     f.setBounds(100,100,500,400);        f.show();        } }
```

Panel

- Definizione del pannello apposito:
- class FunctionPanel extends JPanel {

```
int xMin=-7, xMax=7, yMin=-1, yMax=1;
   // gli intervalli in cui vogliamo graficare
int larghezza=500, altezza=400;
   // corrispondono alla grandezza del Jframe
float fattoreScalaX, fattoreScalaY;
public void paintComponent(Graphics g){
     super.paintComponent(g);
     setBackground(Color.white);
     fattoreScalaX=larghezza/((float)xMax-xMin);
     fattoreScalaY=altezza/((float)yMax-yMin);
     g.setColor(Color.black);
```



Panel

```
g.drawRect(0,0,larghezza-1,altezza-1);
g.setColor(Color.red);
g.drawLine(0,altezza/2, larghezza-1,altezza/2);
g.drawLine(larghezza/2,0,larghezza/2,altezza-1);
g.drawString(""+xMin, 5,altezza/2-5);
g.drawString(""+xMax, larghezza-10,altezza/2-5);
g.drawString(""+yMax, larghezza/2+5,15);
g.drawString(""+yMin, larghezza/2+5,altezza-5);
```

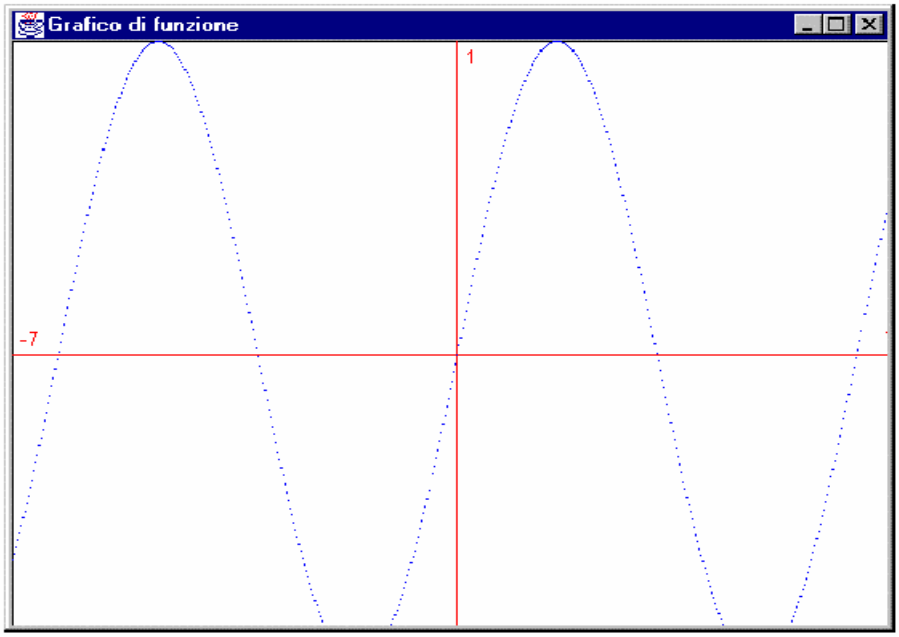
- Continua…..

Panel

```
// disegna il grafico della funzione in blu
g.setColor(Color.blue); setPixel(g,xMin,f(xMin));
 // punto iniziale
 for (int ix=1; ix<larghezza; ix++){ // per ognuno dei pixel della finestra
    float x = xMin+((float)ix)/fattoreScalaX; setPixel(g,x,f(x));
 }
}
 // definizione della funzione, statica, da graficare
 static float f(float x){ return (float)Math.sin(x);        }
// questa serve per riportare i valori della funzione sui valori della finestra
 void setPixel(Graphics g, float x, float y){
    if (x<xMin || x>xMax || y<yMin || y>yMax ) return;
    int ix = Math.round((x-xMin)*fattoreScalaX);
    int iy = altezza-Math.round((y-yMin)*fattoreScalaY);
    g.drawLine(ix,iy,ix,iy);
 // disegna in effetti un singolo punto
 } }
```

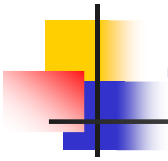
Panel

- Risultato:



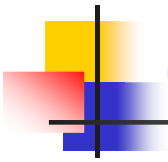
Swing & thread

- Careful use of **concurrency** is particularly important to the Swing programmer. A well-written Swing program uses concurrency to create a user interface that never "freezes" — the program is always responsive to user interaction, no matter what it's doing.
- A Swing programmer deals with **three kinds of threads**:
- **Initial threads**, the threads that execute initial application code.
- The **event dispatch thread**, where all event-handling code is executed. Most code that interacts with the Swing framework must also execute on this thread.
- **Worker threads**, also known as background threads, where time-consuming background tasks are executed.
- The programmer does not need to provide code that explicitly creates these threads: they are provided by the runtime or the Swing framework. The programmer's job is to utilize these threads to create a responsive, maintainable Swing program.
- Like any other program running on the Java platform, a Swing program can create additional threads But for basic Swing programs the threads described here are sufficient.



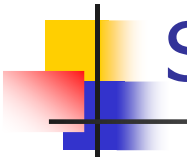
Swing & thread

- Every program has a set of threads where the application logic begins.
  - In standard programs, there's only one such thread: the thread that invokes the main method of the program class.
  - In applets the initial threads are the ones that construct the applet object and invoke its init and start methods; these actions may occur on a single thread, or on two or three different threads, depending on the Java platform implementation.
- These threads are called the **initial threads**. In Swing programs, the initial threads don't have a lot to do. Their most essential job is to create a Runnable object that initializes the GUI and schedule that object for execution on the event dispatch thread.
- Once the GUI is created, the program is primarily driven by GUI events, each of which causes the execution of a short task on the event dispatch thread.
- Application code can schedule additionals tasks on the event dispatch thread (if they complete quickly, so as not to interfere with event processing) or a worker thread (for long-running tasks).



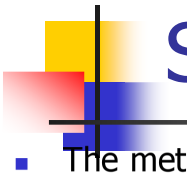
Swing & thread

- An **initial thread** schedules the GUI creation task by invoking *javax.swing.SwingUtilities.invokeLater* or *javax.swing.SwingUtilities.invokeAndWait*. Both of these methods take a single argument: the Runnable that defines the new task. Their only difference is indicated by their names: invokeLater simply schedules the task and returns; invokeAndWait waits for the task to finish before returning.
- In **an applet**, the GUI-creation task must be launched from the init method using invokeAndWait; otherwise, init may return before the GUI is created, which may cause problems for a browser launching an applet.
- In **an application**, scheduling the GUI-creation task is usually the last thing the initial thread does, so it doesn't matter whether it uses invokeLater or invokeAndWait.
- Why doesn't the initial thread simply create the GUI itself? Because almost all code that creates or interacts with Swing components must run on the event dispatch thread. This is discussed later.



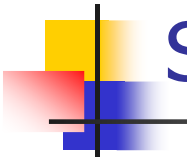
Swing & thread

- Swing event handling code runs on a special thread known as the **event dispatch thread**. Most code that invokes Swing methods also runs on this thread. This is necessary because **most Swing object methods are not "thread safe"**: invoking them from multiple threads risks thread interference or memory consistency errors. Some Swing component methods are labelled "thread safe" in the API; these can be safely invoked from any thread. All other Swing component methods must be invoked from the event dispatch thread. Programs that ignore this rule may function correctly most of the time, but are subject to unpredictable errors that are difficult to reproduce.
- It's useful to think of the code running on the event dispatch thread as a series of short tasks.
  - Most tasks are invocations of event-handling methods, such as *ActionListener.actionPerformed*.
  - Other tasks can be scheduled by application code, using invokeLater or invokeAndWait.
- Tasks on the event dispatch thread must finish quickly; if they don't, unhandled events back up and the user interface becomes unresponsive.



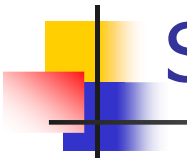
Swing & thread

- The method invokeLater(Runnable) is intended to execute a new runnable thread from a Swing application **without disturbing the normal sequence of event dispatching** from the GUI.
- The method places the runnable object in the queue of Swing events that are due to be processed and returns immediately. The runnable object's run() method is only called when it reaches the front of the queue.
- The deferred effect of the invokeLater(Runnable) method **ensures that any necessary updates to the user interface can occur immediately**, and the runnable work will begin as soon as those high priority events are dealt with.
- The invoke later method might be used to start work in response to a button click that also requires a significant change to the user interface, perhaps to restrict other activities, while the runnable thread executes.
- The Swing Library is not thread safe, which means that only one thread may create or modify swing objects. This is the AWT Event thread. **Whenever Swing calls a listener method, it is executed on the AWT Event thread**, so you may modify Swing objects here. InvokeLater causes Swing to run a given runnable when it can on the same AWT Event thread, ensuring correctness. **You should use invokeLater any time you want to modify a Swing object outside of a listener method.** Quindi invokelater NON è obbligatorio per la creazione della GUI, né deve essere esclusivamente utilizzato per tale scopo, ma può essere sfruttato quando occorre.



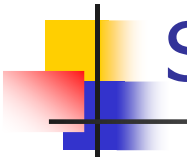
Swing & thread

- When a Swing program needs to execute a long-running task, it usually uses one of the **worker threads**, also known as the background threads. Each task running on a worker thread is represented by an instance of javax.swing.SwingWorker. SwingWorker itself is an abstract class; you must define a subclass in order to create a SwingWorker object. SwingWorker provides the following features:
- SwingWorker can define a method, *done*, which is automatically invoked on the event dispatch thread when the background task is finished.
- SwingWorker implements *java.util.concurrent.Future*. This interface allows the background task to provide a return value to the other thread. Other methods in this interface allow cancellation of the background task and discovering whether the background task has finished or cancelled.



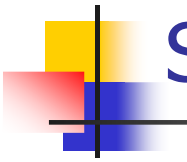
Swing & thread

- The background task can provide **intermediate results** by invoking *SwingWorker.publish*, causing *SwingWorker.process* to be invoked from the event dispatch thread.
- The background task can define bound properties. Changes to these properties trigger events, causing event-handling methods to be invoked on the event dispatch thread.
- The javax.swing.SwingWorker class was added to the Java platform in Java SE 6. Prior to this, another class, also called SwingWorker, was widely used for some of the same purposes. The old SwingWorker was not part of the Java platform specification, and was not provided as part of the JDK.



Swing & thread

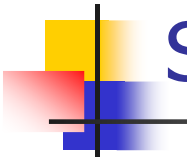
- Let's start with an example of a very simple but time-consuming task. The *TumbleItem* applet loads a set of graphic files used in an animation.
  - If the graphic files are loaded from an initial thread, there may be a delay before the GUI appears.
  - If the graphic files are loaded from the event dispatch thread, the GUI may be temporarily unresponsive (invokelater() is not feasible).
  - To avoid these problems, TumbleItem executes an **instance of SwingWorker** from its initial threads.
- The object's **doInBackground method, executing in a worker thread**, loads the images into an ImageIcon array, and returns a reference to it.
- Then the **done method**, **executing in the event dispatch thread**, invokes get to retrieve this reference, which it assigns to an applet class field named imgs. This allows ThumbleItem to construct the GUI immediately, without waiting for images to finish loading.



Swing & thread

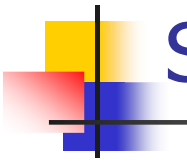
```
SwingWorker worker = new SwingWorker() {
@Override
public ImageIcon[] doInBackground() {
    final ImageIcon[] innerImgs = new ImageIcon[nimgs];
    for (int i = 0; i < nimgs; i++) { innerImgs[i] = loadImage(i+1); }
    return innerImgs; }
@Override
public void done() {
    try { imgs = get(); }
    catch (java.util.concurrent.ExecutionException e) { … }
     }
};
```

- All concrete subclasses of SwingWorker implement doInBackground; implementation of done is optional



Swing & thread

- You may wonder if the code that sets imgs is unnecessarily complicated. Why make doInBackground return an object and use done to retrieve it? Why not just have doInBackground set imgs directly?
- The problem is that the object imgs refers to is created in the worker thread and used in the event dispatch thread. When **objects are shared between threads** in this way, you must make sure that changes made in one thread are visible to the other. Using get guarantees this, because it creates a **happens before relationship** between the code that creates imgs and the code that uses it.
- There are actually two ways to retrieve the object returned by doInBackground: invoke *SwingWorker.get* with no arguments, i.e. if the background task is not finished get blocks until it is; otherwise, a timeout can be specified, i.e. if the timeout expires first, get throws java.util.concurrent.TimeoutException.
- Be careful when invoking either overload of get from the event dispatch thread; **until get returns, no GUI events are being processed, and the GUI is "frozen".** Don't invoke get without arguments unless you are sure that the background task is finishing quickly.



Swing & thread

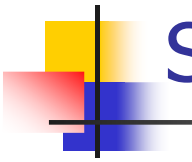
- It is often useful for a background task to **provide interim results while it is still working**. The task can do this by invoking **SwingWorker.publish method**, which accepts a set of arguments.
- To collect results provided by publish, override SwingWorker.process, which will be invoked from the event dispatch thread. Results from multiple invocations of publish are often accumulated for a single invocation of process.
- Let's look at the way the Flipper example uses publish to provide interim results. This program generates a series of random boolean values in a background task. This is equivalent to flipping a coin; hence the name Flipper. To report its results, the background task uses an object of type FlipPair

Swing & thread

```
private static class FlipPair {
private final long heads, total;
FlipPair(long heads, long total) {
this.heads = heads;
this.total = total; }
}
```

- The heads field is the number of times the random value has been true; the total field is the total number of random values. The background task is represented by an instance of FlipTask:

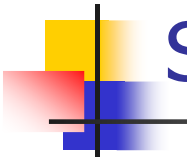
```
private class FlipTask extends SwingWorker {
@Override
protected void doInBackground() {
long heads = 0; long total = 0;
Random random = new Random();
while (!isCancelled()) {
total++;
if (random.nextBoolean()) { heads++; }
publish(new FlipPair(heads, total)); }
}
```

Swing & thread

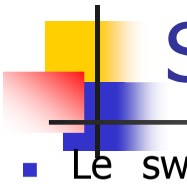
```
protected void process(List pairs) {
FlipPair pair = pairs.get(pairs.size() - 1);
headsText.setText(String.format("%d", pair.heads));
totalText.setText(String.format("%d", pair.total));
```

- Because publish is invoked very frequently, a lot of FlipPair values will probably be accumulated before *process* is invoked in the event dispatch thread; process is only interested in the last value reported each time, using it to update the GUI
- **Each publish invocation does not always generate a corresponding process invocation**: publish(...) joins events and only calls process(...) "occasionally". This is a really nice feature because it helps avoid blocking the EDT. WE cannot control when process is called
- Moreover, process(...) is sometimes called after done()! You don't really have any control over this and cannot predict when it will happen. This is not a bug, since Java only guarantees that done "is Executed on the Event Dispatch Thread after the doInBackground() method is finished", so **simply don't rely on the order of process(...) and done().**



Swing & thread

- To **cancel a running background task**, invoke SwingWorker.cancel. The task must cooperate with its own cancellation in two ways:
  - By terminating when it receives an interrupt.
  - By invoking SwingWorker.isCanceled at short intervals. This method returns true if cancel has been invoked for this SwingWorker.
- The cancel method takes a single boolean argument. If the argument is true, cancel sends the background task an interrupt. Whether the argument is true or false, invoking cancel changes the cancellation status of the object to true. This is the value returned by isCanceled. Once changed, the cancellation status cannot be changed back.
- The Flipper example from the previous section uses the status-only idiom. The main loop in doInBackground exits when isCancelled returns true. This will occur when the user clicks the "Cancel" button, triggering code that invokes cancel with an argument of false (no interrupt).
- The status-only approach makes sense for Flipper because its implementation of SwingWorker.doInBackground does not include any code that might throw InterruptedException.

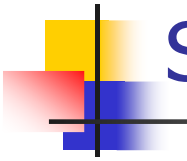


Swing & eventi

- Le swing si basano su **eventi** associati alle azioni eseguite dall'utente. Consideriamo un esempio di un bottone che, se cliccato, deve fare emettere un suono:

```
public class Beeper ... implements ActionListener {
 ...
 //where initialization occurs:
 button.addActionListener(this);
 ...
 public void actionPerformed(ActionEvent e) {
 ...//Make a beep sound... }
 }
```

- The Beeper class implements the ActionListener interface, which contains one method: actionPerformed. Since Beeper implements ActionListener, a Beeper object can register as a listener for the action events that buttons fire. Once the Beeper has been registered using the Button addActionListener method, the Beeper's actionPerformed method is called every time the button is clicked.

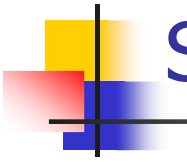


Swing & eventi

- You can tell what kinds of events a component can fire by looking at the **kinds of event listeners** you can register on it. For example, the JComboBox class defines these listener reg methods:
- addActionListener
- addItemListener
- addPopupMenuListener
- Thus, a combo box supports action, item, and popup menu listeners in addition to the listener methods it inherits from JComponent. **A component fires only those events for which listeners have registered on it**. For example, if an action listener is registered on a particular combo box, but the combo box has no other listeners, then the combo box will fire only action events — no item or popup menu events.
- Listeners supported by Swing components fall into two categories:
  - listeners that all swing components support
  - other specific listeners that swing components support

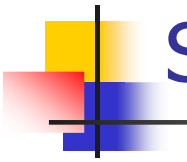
Swing & eventi

- Because all Swing components descend from the AWT Component class, you can register the **following common listeners** on any component:
- **component listener**: Listens for changes in the component's size, position, or visibility.
- **focus listener**: Listens for whether the component gained or lost the ability to receive keyboard input.
- **key listener**: Listens for key presses; key events are fired only by the component that has the current keyboard focus.
- **mouse listener**: Listens for mouse clicks and mouse movement into or out of the component's drawing area.
- **mouse-motion listener**: Listens for changes in the cursor's position over the component.
- **mouse-wheel listener** (introduced in 1.4): Listens for mouse wheel movement over the component.
- All Swing components descend from the AWT Container class, but many of them aren't used as containers. So any Swing component can fire container events, which notify listeners that a component has been added or removed. However, only containers (such as panels and frames) and compound components (such as combo boxes) typically fire container events.



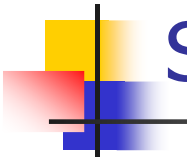
Swing & eventi

- **Any number of event listener objects** can listen for **all kinds** of events from **any number of event source objects**. For example, a program might create one listener per event source, or it might have a single listener for all events from all sources. A program can even have more than one listener for a single kind of event from a single event source.
- Multiple listeners can register to be notified of events of a particular type from a particular source. Also, the same listener can listen to notifications from different objects.
- Each event is represented by an object that gives information about the event and identifies the event source. **Event sources are often components or models**, but other (any…) kinds of objects can also be event sources.
- Whenever you want to detect events from a particular component, first check the how-to section for that component.



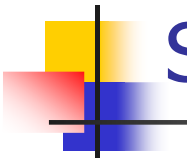
Swing & eventi

- The most important rule is that **event listeners should execute very quickly**. Because all drawing and event-listening methods are executed in the same thread, a slow event-listener method can make **the program seem unresponsive** and slow to repaint itself. If you need to perform some lengthy operation as the result of an event, **do it by starting up another thread** (or somehow sending a request to another thread) to perform the operation (e.g. using Swingworker).
- You have **many choices on how to implement an event listener.** You might choose to implement separate classes for different kinds of event listeners. This can be an easy architecture to maintain, but many classes can also mean reduced performance.
- When designing your program, you might want to implement your **event listeners in a class that is not public**, but somewhere more hidden. A private (**e.g. anonymous inner class**) implementation is a more secure implementation.



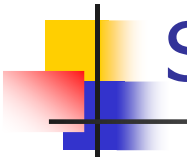
Swing & eventi

- **Every event-listener method has a single argument** — an object that inherits from the EventObject class. Although the argument always descends from EventObject, its type is generally specified more precisely. For example, the argument for methods that handle mouse events is an instance of MouseEvent, where MouseEvent is an indirect subclass of EventObject.
- The EventObject class defines one very useful method Object **getSource()**, that returns the object that fired the event.
- Often, an event class defines methods that return information about the event. For example, you can query a MouseEvent object for information about where the event occurred, how many clicks the user made, which modifier keys were pressed, and so on.



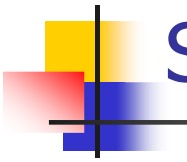
Swing & eventi

- Events can be divided into **two groups: low-level events and semantic events**:
- Low-level events represent window-system occurrences or low-level input. Everything else is a semantic event.
- Examples of low-level events include mouse and key events — both of which result directly from user input. Examples of semantic events include action and item events. A semantic event might be triggered by user input; for example, a button customarily fires an action event when the user clicks it, and a text field fires an action event when the user presses Enter. However, **some semantic events aren't triggered by low-level events, at all**. For example, a table-model event might be fired when a table model receives new data from a database.



Swing & eventi

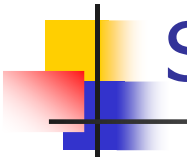
- Whenever possible, **you should listen for semantic events** rather than low-level events. so your code is robust and portable as possible. For example, listening for action events on buttons, rather than mouse events, means that the button will react appropriately when the user tries to activate the button using a keyboard alternative or a look-and-feel-specific gesture.
- When dealing with a **compound component** such as a combo box, it's imperative that you stick to semantic events, since you have no reliable way of registering listeners on all the look-and-feel-specific components that might be used to form the compound component.



Swing & eventi

- Some listener interfaces contain **more than one method**. For example, the MouseListener interface contains five methods: mousePressed, mouseReleased, mouseEntered, mouseExited, and mouseClicked. Even if you care only about mouse clicks, if your class directly implements MouseListener, then you must implement all five MouseListener methods. Methods for those events you don't care about can have empty bodies.

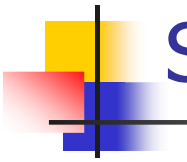
```
//An example that implements a listener interface directly.
public class MyClass implements MouseListener {
... someObject.addMouseListener(this); ...
public void mousePressed(MouseEvent e) { }
public void mouseReleased(MouseEvent e) { }
public void mouseEntered(MouseEvent e) { }
public void mouseExited(MouseEvent e) { }
public void mouseClicked(MouseEvent e) {
...//Event listener implementation goes here... }
}
```



Swing & eventi

- The resulting collection of empty method bodies can make code harder to read and maintain. To help you avoid implementing empty method bodies, the API generally includes an **adapter** class for each listener interface with more than one method. For example, the MouseAdapter class implements the MouseListener interface. An adapter class implements empty versions of all its interface's methods. To use an adapter, you create a subclass of it and override only the methods of interest, rather than directly implementing all methods of the listener interface.

```
/* * An example of extending an adapter class instead of * directly
   implementing a listener interface. */
public class MyClass extends MouseAdapter {
... someObject.addMouseListener(this); ...
public void mouseClicked(MouseEvent e) {
...//Event listener implementation goes here... }
}
```

Swing & eventi

- Generalmente, i listener sono realizzati all'interno di un'applicazione come classe innestata (eventualmente anonima):

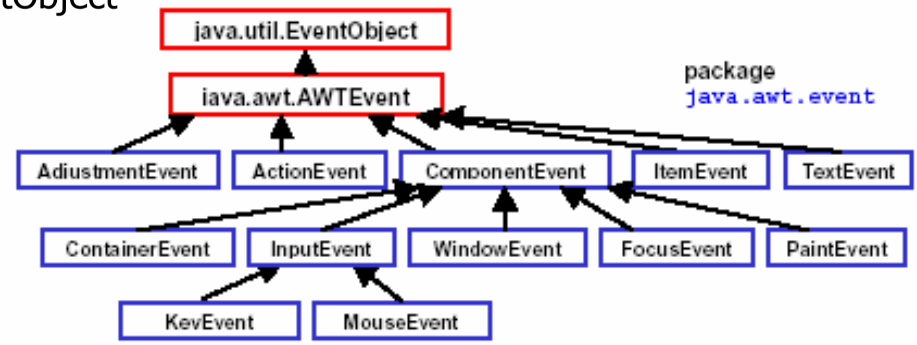
```
//An example of using an inner class.
public class MyClass extends Applet {
... someObject.addMouseListener(new MyAdapter()); ...
class MyAdapter extends MouseAdapter {
public void mouseClicked(MouseEvent e) {
...//Event listener implementation goes here... } } }
```

- Tramite classe anonima:

```
//An example of using an anonymous inner class.
public class MyClass extends Applet {
... someObject.addMouseListener(new MouseAdapter() {
public void mouseClicked(MouseEvent e) {
...//Event listener implementation goes here... } });
... } }
```

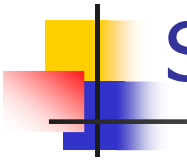
Swing & eventi

- Ogni componente grafico, quando si opera su di esso, genera un evento che descrive cosa è accaduto (attenzione: il concetto di evento non si applica necessariamente solo agli oggetti grafici, ma è generalmente con la grafica che esso assume rilevanza e comprensione immediata)
- Tipicamente, ogni componente può generare molti tipi diversi di eventi, in relazione a ciò che sta accadendo
- In Java, un evento è un oggetto, istanza di (una sottoclasse di) java.util.EventObject



Swing & eventi

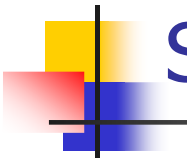
- Quando si interagisce con un componente "attivo" si genera un evento, che è un oggetto Event della (sotto)classe opportuna; l'oggetto Event contiene tutte le informazioni sull'evento (chi l'ha creato, cosa è successo, ecc); il sistema invia tale oggetto Event all'oggetto Listener (ascoltatore degli eventi) preventivamente registrato, che gestisce l'evento.
- L'attività non è più algoritmica (input/computazione/output), è interattiva e reattiva con opportune azioni
- Quando viene ad esempio premuto un bottone, esso genera un evento di classe ActionEvent; questo evento viene inviato dal sistema allo specifico ascoltatore degli eventi per quel bottone.
- L'ascoltatore degli eventi deve implementare la interfaccia ActionListener, e può essere un oggetto di un'altra classe al di fuori del pannello o può essere anche il pannello stesso (this)
- Tale ascoltatore degli eventi deve implementare il metodo che gestisce l'evento definito nella interfaccia actionListener, in particolare actionPerformed(ActionEvent ev)



Swing & eventi

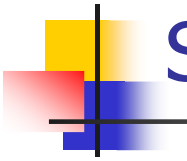
- Esempio: un'applicazione costituita da un'etichetta (JLabel) e un pulsante (JButton); l'etichetta può valere "Tizio" o "Caio" (all'inizio vale "Tizio"). Premendo il bottone, l'etichetta deve commutare, diventando "Caio" se era "Tizio", o "Tizio" se era "Caio":

```
public class Es8Panel extends JPanel implements ActionListener{
   private JLabel l;
   public Es8Panel(){
     super();
     l = new JLabel("Tizio");
     add(l);
     JButton b = new JButton("Tizio/Caio");
                        // Tizio/Caio è l'etichetta del pulsante
     b.addActionListener(this);
                        // registra l'oggetto panel stesso come
                        // ascoltatore degli eventi
     add(b); } …
```

Swing & eventi

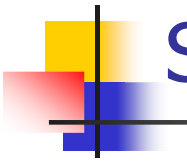
```
…
public void actionPerformed(ActionEvent e){ if
   (l.getText().equals("Tizio"))
     l.setText("Caio");
   else l.setText("Tizio");
   }
}

import java.awt.*; import javax.swing.*;
import java.awt.event.*;
public class EsSwing8 {
    public static void main(String[] v){
   JFrame f = new JFrame("Esempio 7");
   Container c = f.getContentPane();
   Es8Panel p = new Es8Panel();
   c.add(p); f.pack(); f.show();    }}
```



Swing & eventi

- Altro esempio: Cambiare il colore di sfondo tramite due pulsanti: uno lo rende rossa, l'altro azzurro
- Architettura dell'applicazione: Un pannello che contiene i due pulsanti creati dal costruttore del pannello, ed un unico ascoltatore degli eventi per entrambi i pulsanti, quindi necessità di capire, in actionPerformed(), quale pulsante è stato premuto

```
public class Es9Panel extends JPanel implements
ActionListener {
   JButton b1, b2;
   public Es9Panel(){
     super();
     b1 = new JButton("Rosso");      b2 = new JButton("Azzurro");
     b1.addActionListener(this);     b2.addActionListener(this);
     // il pannello fa da ascoltatore degli eventi per entrambi i pulsanti
     add(b1);          add(b2);}
...
```

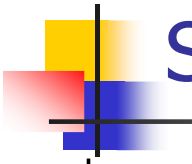



Swing & eventi

```
...
public void actionPerformed(ActionEvent e){
      Object pulsantePremuto = e.getSource();
      // si recupera il riferimento all'oggetto
      // che ha generato l'evento
      if (pulsantePremuto==b1)
      // e si confronta questa con i riferimenti
      // agli oggetti bottoni b1 e b2
      setBackground(Color.red);
      if (pulsantePremuto==b2)
      setBackground(Color.cyan);
   }
}
```

- Un modo alternativo per capire chi aveva generato l'evento poteva essere quello di guardare l'etichetta associata al pulsante:

```
String nome = e.getActionCommand();
if nome.equals("Rosso") …
```



Swing & eventi

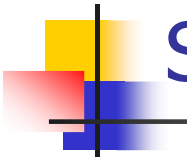
- Le operazioni sulle finestre (finestra chiusa, aperta, minimizzata, ingrandita…) generano un WindowEvent
- Gli eventi di finestra sono gestiti dai metodi dichiaratidall'interfaccia WindowListener

```
public void windowClosed(WindowEvent e);
public void windowClosing(WindowEvent e);
public void windowOpened(WindowEvent e);
public void windowIconified(WindowEvent e);
public void windowDeiconified(WindowEvent e);
public void windowActivated(WindowEvent e);
public void windowDeactivated(WindowEvent e);
```

- ogni metodo viene invocato dall'evento appropriato;
- Il comportamento predefinito di questi metodi va già bene tranne windowClosing(), che non fa uscire l'applicazione ma nasconde solo la finestra; per far sì che chiudendo la finestra del frame l'applicazione venga chiusa, il frame deve implementare l'interfaccia WindowListener, e ridefinire WindowClosing in modo che invochi System.exit()
- Gli altri metodi devono essere implementati, ma, non dovendo svolgere compiti precisi, possono essere definiti con un corpo vuoto

Swing & eventi

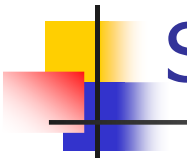
```
public class EsSwing9 {
    public static void main(String[] v){
        JFrame f = new JFrame("Esempio 9");
        Container c = f.getContentPane();
        Es9Panel p = new Es9Panel();          c.add(p);
        f.addWindowListener( new Terminator() );
            // Terminator è la classe che implementa
            // l'interfaccia WindowListener
        f.pack();          f.show();
    }
}
class Terminator implements WindowListener {
    public void windowClosed(WindowEvent e){}
    public void windowClosing(WindowEvent e){
        System.exit(0);
        // così chiudendo la finestra si esce dalla applicazione    }}
```



Swing & eventi

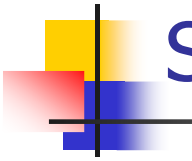
- Il JTextField è un componente "campo di testo", usabile per scrivere e visualizzare una riga di testo
  - il campo di testo può essere editabile o no
  - il testo è accessibile con getText() / setText()
- Il campo di testo è parte di un oggetto Document
- Ogni volta che il testo in esso contenuto cambia si genera un DocumentEvent nel documento che contiene il campo di testo
- Se però è sufficiente registrare i cambiamenti solo quando si preme INVIO, basta gestire semplicemente il solito ActionEvent

- ESEMPIO
- Un'applicazione comprendente un pulsante e due campi di testo
  - uno per scrivere testo, l'altro per visualizzarlo
  - Quando si preme il pulsante, il testo del secondo campo (non modificabile dall'utente) viene cambiato, e reso uguale a quello scritto nel primo
  - L'unico evento è ancora il pulsante premuto: ancora non usiamo il DocumentEvent



Swing & eventi

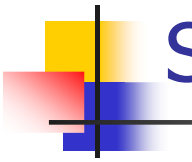
```
public class EsSwing {
    public static void main(String[] v){
            JFrame f = new JFrame("Esempio");
            Container c = f.getContentPane();
            Es10Panel p = new Es10Panel();
            c.add(p);
            f.addWindowListener( new Terminator() );
            f.setSize(300,120);
            f.show();
    }
}
```



Swing & eventi

Il pannello:

```
class Es10Panel extends JPanel implements ActionListener {
        JButton b;
        JTextField txt1, txt2;
        public Es10Panel(){
                super();
                b = new JButton("Aggiorna");
                txt1=new JTextField("Scrivere qui il testo", 25);
                txt2 = new JTextField(25); // larghezza in caratt.
                txt2.setEditable(false); // non modificabile
                b.addActionListener(this);
                add(txt1);      add(txt2);      add(b);
   }
   public void actionPerformed(ActionEvent e){
        txt2.setText( txt1.getText() );    }}
```

Swing & eventi

- Sfruttiamo il concetto di **documento** che sta dietro a ogni campo di testo
- A ogni modifica del contenuto, il documento di cui il campo di testo fa parte genera un DocumentEvent per segnalare l'avvenuto cambiamento
- Tale evento dev'essere gestito da un opportuno DocumentListener cioè da un oggetto di una classe che implementi l'interfaccia DocumentListener
- **L'interfaccia DocumentListener** dichiara tre metodi:
  - void insertUpdate(DocumentEvent e);
  - void removeUpdate(DocumentEvent e);
  - void changedUpdate(DocumentEvent e);
- Il terzo non è mai chiamato da JTextField, serve solo per altri componenti
- L'oggetto-evento DocumentEvent passato come parametro in realtà esiste solo per uniformità, in quanto cosa sia accaduto è già implicito nel metodo chiamato.
- Nel nostro caso: l'azione da svolgere in caso di inserimento o rimozione di caratteri è identica, quindi i due metodi insertUpdate e removeUpdate saranno identici, ma vanno comunque implementati, analogamente al metodo changedUpdate(DocumentEvent e)

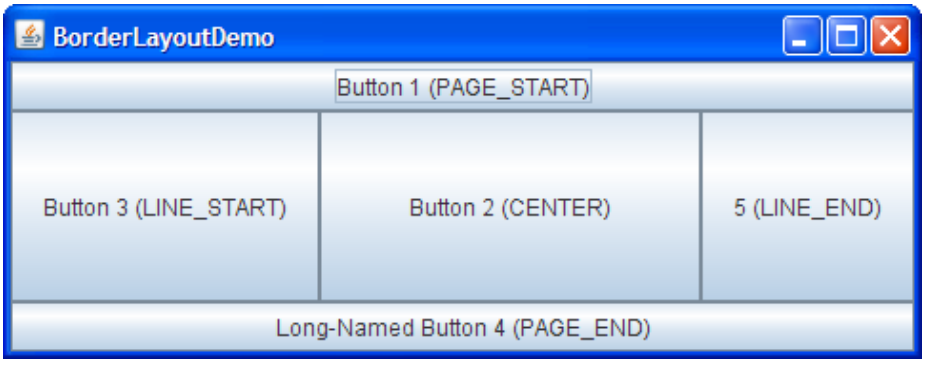
Swing & eventi

```
import javax.swing.event.*;
class Es12Panel extends JPanel implements DocumentListener {
        JTextField txt1, txt2;
        public Es12Panel(){
                super();
                txt1= new JTextField("Scrivere qui il testo", 25);
                txt2 = new JTextField(25);
                txt2.setEditable(false);
                txt1.getDocument().addDocumentListener(this);
                // ricava il documento di cui il campo
                // di test txt1 fa parte e gli associa il
                // pannello come listener
                add(txt1);        add(txt2);
        }
Public   void insertUpdate(DocumentEvent e){txt2.setText(txt1.getText()); }
public void removeUpdate(DocumentEvent e){txt2.setText(txt1.getText()); }
public void changedUpdate(DocumentEvent e){}
```

A ogni inserimento o cancellazione di caratteri l'aggiornamento è automatico

Layout

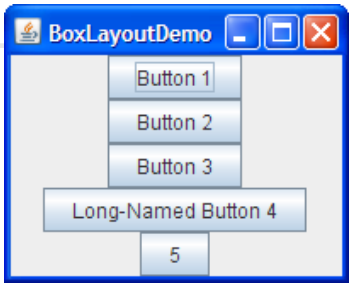
- E' possibile dare una struttura (layout) all'applicazione basata su swing
- Esistono diversi layout predefiniti in Java, il primo è **BorderLayout**:

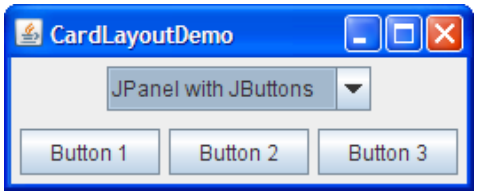


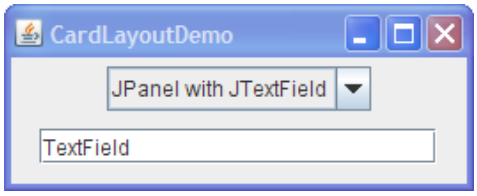
- Every content pane is initialized to use a BorderLayout. A BorderLayout places components in up to five areas: top, bottom, left, right, and center. All extra space is placed in the center area, eventually resizing its dimensions, consequently enlarging the entire frame.

Layout

- The **BoxLayout** class puts components in a single row or column. It respects the components' requested maximum sizes and also lets you align components.

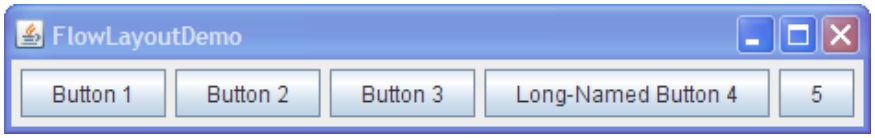






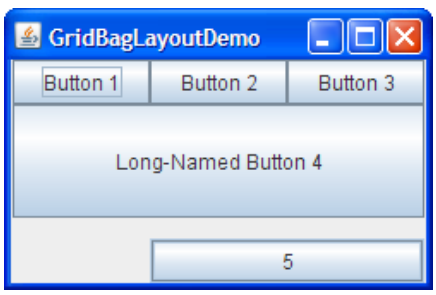
- The **CardLayout** class lets you implement an area that contains different components at different times. A CardLayout is often controlled by a combo box, with the state of the combo box determining which panel (group of components) the CardLayout displays. An alternative to using CardLayout is using a tabbed pane, which provides similar functionality but with a pre-defined GUI.

Layout



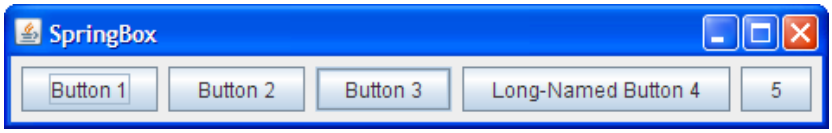
- **FlowLayout** is the default layout manager for every JPanel. It simply lays out components in a single row, starting a new row if its container isn't sufficiently wide. Both panels in CardLayoutDemo, shown previously, use FlowLayout.

- **GridBagLayout** is a sophisticated, flexible layout manager. It aligns components by placing them within a grid of cells, allowing some components to span more than one cell. The rows in the grid can have different heights, and grid columns can have different widths.

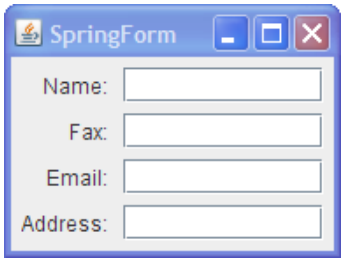


Layout

- **GridLayout** simply makes a bunch of components equal in size and displays them in the requested number of rows and columns.



- **SpringLayout** is a flexible layout manager designed for use by GUI builders. It lets you specify precise relationships between the edges of components under its control. For example, you might define that the left edge of one component is a certain distance (which can be dynamically calculated) from the right edge of a second component.



Layout

- A layout manager is an object that implements the LayoutManager interface and determines the size and position of the components within a container. Although components can provide size and alignment hints, a container's **layout manager has the final say on the size and position** of the components within the container.
- As a rule, the only containers whose layout managers you need to worry about are JPanels and content panes. Each JPanel object is initialized to use a FlowLayout, unless you specify differently when creating the JPanel. Content panes use BorderLayout by default. If you don't like the default layout manager that a panel or content pane uses, you're free to change it to a different one:
- You can set a panel's layout manager using the JPanel constructor:
  JPanel panel = new JPanel(new BorderLayout());
- In the second case,after a container has been created, you can set its layout manager using the setLayout method:
  Container contentPane = frame.getContentPane();
  contentPane.setLayout(new FlowLayout());

Layout

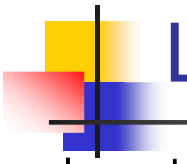
- Although we recommend that you use layout managers, you can perform layout without them. By setting a container's layout property to null, you make the container use no layout manager. With this strategy, **absolute positioning**, you must specify the size and position of every component within that container. One drawback of absolute positioning is that it doesn't adjust well when the top-level container is resized. It also doesn't adjust well to differences between users and systems, such as different font sizes and locales.
- When you add components to a panel or content pane, the arguments you specify to the add method depend on the layout manager that the panel or content pane is using. For example, BorderLayout requires that you specify the area to which the component should be added, using code like this: pane.add(aComponent, BorderLayout.PAGE_START); The how-to section for each layout manager has details on what, if any, arguments you need to specify to the add method. Some layout managers, such as GridBagLayout and SpringLayout, require elaborate setup procedures. Many layout managers, however, simply place components based on the order they were added to their container.

Layout

- Sometimes you need to customize the size hints that a component provides to its container's layout manager, so that the component will be laid out well. **You can do this by specifying one or more of the minimum, preferred, and maximum sizes** of the component. You can invoke the component's methods for setting size hints — setMinimumSize, setPreferredSize, and setMaximumSize.
- Besides providing size hints, **you can also provide alignment hints**. For example, you can specify that the top edges of two components should be aligned. You set alignment hints either by invoking the component's setAlignmentX and setAlignmentY methods
- Attenzione perché le richieste espresse nei due punti precedenti sono nella maggior parte dei casi, **ignorate**

Layout

- Three factors influence the amount of space between visible components in a container:
  - **The layout manager**; Some layout managers automatically put space between components; others don't. Some let you specify the amount of space between components. See the how-to page for each layout manager for information about spacing support.
  - **Invisible components**; You can create lightweight components that perform no painting, but that can take up space in the GUI. Often, you use invisible components in containers controlled by BoxLayout.
  - **Empty borders**; No matter what the layout manager, you can affect the apparent amount of space between components by adding empty borders to components. The best candidates for empty borders are components with no default border, such as panels and labels. Some other components might not work well with borders in some look-and-feel implementations, because of the way their painting code is implemented.



Layout

- Layout managers have different strengths and weaknesses. Flexible layout managers such as GridBagLayout and SpringLayout can fulfill many layout needs, however the right layout depends on the **scenario** you face with:
- You need to display a component in as much space as it can get; If it's the only component in its container, use GridLayout or BorderLayout. Otherwise, BorderLayout or GridBagLayout might be a good match. If you use BorderLayout, you'll need to put the space-hungry component in the center. With GridBagLayout, you'll need to set the constraints for the component so that fill=GridBagConstraints.BOTH. Another possibility is to use BoxLayout, making the space-hungry component specify very large preferred and maximum sizes.
- You need to display a few components in a compact row at their natural size. Consider using a JPanel to group the components and using either the JPanel's default FlowLayout manager or the BoxLayout manager. SpringLayout is also good for this.

Layout

- If you need to display a few components of the same size in rows and columns, GridLayout is the right choice
- If you need to display a few components in a row or column, possibly with varying amounts of space between them, custom alignment, or custom component sizes, BoxLayout is the right choice
- You need to display aligned columns, as in a form-like interface where a column of labels is used to describe text fields in an adjacent column. SpringLayout is a natural choice for this. The SpringUtilities class used by several Tutorial examples defines a makeCompactGrid method that lets you easily align multiple rows and columns of components.
- You have a complex layout with many components. Consider either using a very flexible layout manager such as GridBagLayout or SpringLayout, or grouping the components into one or more JPanels to simplify layout. If you take the latter approach, each JPanel might use a different layout manager.