Interfacce Grafiche con Java Swing

Progettazione del Software

Giuseppe De Giacomo, Paolo Liberatore, Massimo Mecella

Sapienza Università di Roma
Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Materiale didattico preparato a partire da versioni precedenti di
Massimiliano de Leoni, Claudio Di Ciccio, Fabio Patrizi, Alessandro Russo

Contenuti

Contenuti

Java Foundation Classes (JFC) e Swing

Java Foundation Classes (JFC): framework che include un insieme di componenti, servizi e API (Application Programming Interface) per lo sviluppo di interfacce grafiche (Graphical User Interfaces – GUIs)

Elementi e caratteristiche:

- Componenti Swing: componenti e funzionalità per la realizzazione di interfacce grafiche
- Supporto per diversi Look-and-Feel: possibilità di configurare e personalizzare l'aspetto grafico (look and feel) delle applicazioni
- Supporto per l'Accessibilità: funzionalità a supporto di tecnologie assistive (screen readers, terminali Braille, ecc.) per garantire l'accessibilità a qualsiasi tipologia d'utente
- Java 2D: possibilità di integrare immagini, grafici ed elementi testuali complessi in 2D
- Supporto per l'Internazionalizzazione: possibilità di adattare gli elementi di input ed output in funzione di una specifica area linguistica

Java Swing

Swing: Java GUI toolkit che implementa e fornisce una libreria di componenti grafici (widgets) e funzionalità per lo sviluppo di interfacce grafiche

Caratteristiche della libreria:

- indipendente dalla piattaforma sottostante
  - componenti Swing stessi sono implementati interamente in Java
- estendibile e personalizzabile da parte del programmatore
- configurabile nel look-and-feel e rendering dei componenti grafici
  - aspetto grafico della GUI può essere uguale su tutte le piattaforme o assumere il look and feel del Sistema Operativo sottostante (es. Microsoft Windows, Linux, OS X)
- basata sul pattern Model/View/Controller (MVC) con paradigma di programmazione basato su eventi (event-driven) a singolo thread (single-threaded)
  - ulteriori dettagli più avanti nel corso

Swing e AWT

Libreria Swing è un'estensione dell'Abstract Window Toolkit (AWT) di Java

AWT → componenti heavyweight ("pesanti")

- ogni componente viene visualizzato e gestito da un corrispondente componente nativo del toolkit grafico del sistema operativo (SO) sottostante
- comportamento dei componenti è system-dependent e il rendering grafico è affidato al SO
- alla creazione di ogni componente AWT corrisponde l'allocazione di risorse del toolkit grafico del SO

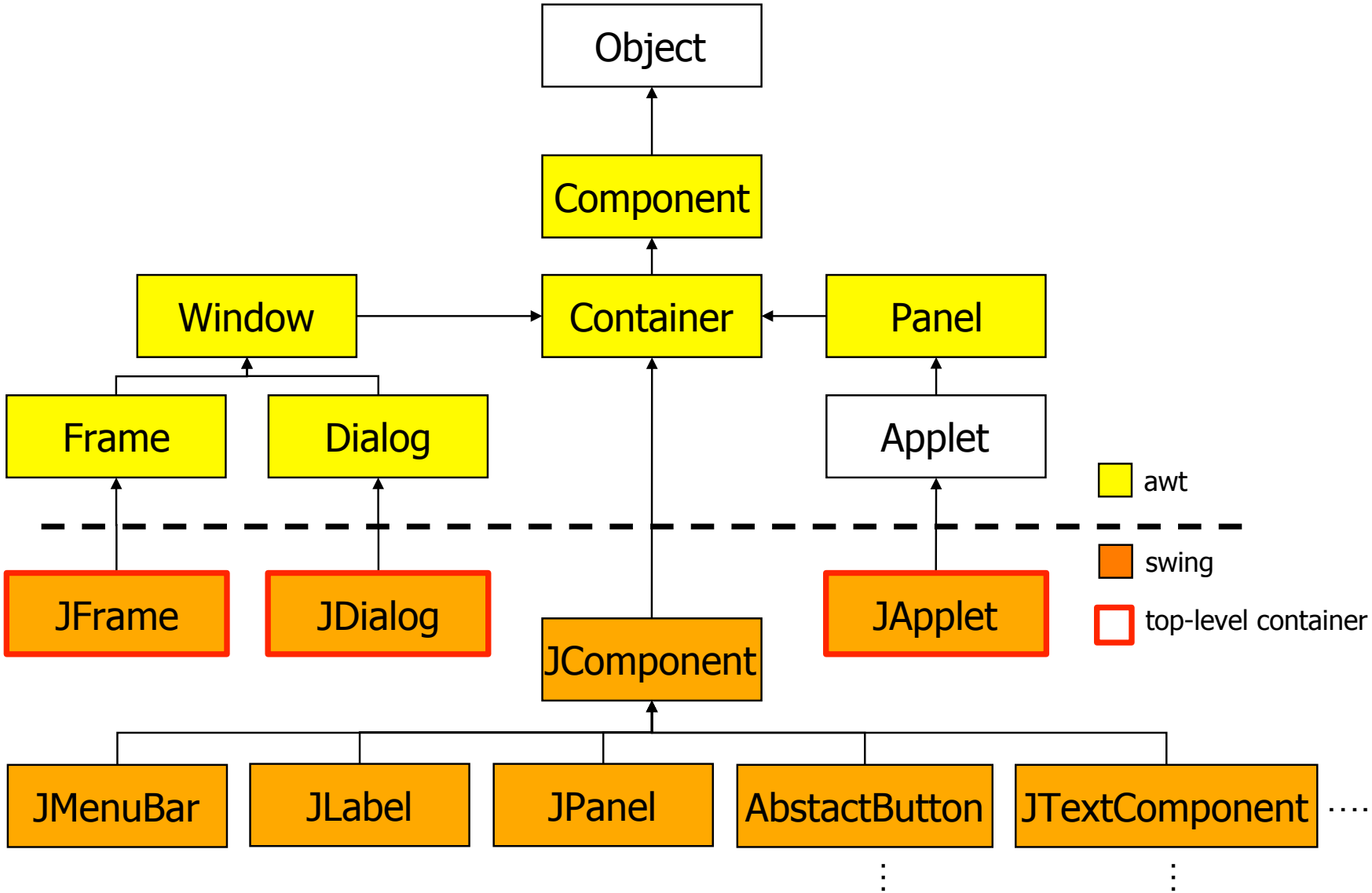
Swing → componenti lightweight ("leggeri")

- componenti implementati direttamente in Java e indipendenti dal toolkit grafico del SO sottostante
- rendering grafico dei componenti è affidato a Java 2D, senza sfruttare il toolkit grafico nativo del SO
- la creazione di componenti Swing non richiede l'allocazione di risorse del toolkit grafico del SO

La libreria Swing

- La libreria è parte della piattaforma Java Standard Edition (Java SE) dalla release 1.2
- Swing include 18 package pubblici
- Package principali
  - `javax.swing.*`
  - `javax.swing.event.*`
- I componenti Swing sono organizzati secondo una gerarchia che estende la struttura di AWT

Swing: la gerarchia di base



I top-level container e le gerarchie di contenimento

Ogni applicazione che utilizza componenti Swing ha almeno un top-level container

- un `Container` è un `Component` che può contenere altri componenti
- Swing definisce tre top-level container: `JFrame`, `JDialog`, e `JApplet`
- il top-level container rappresenta la radice di una gerarchia di contenimento (con struttura ad albero) costituta da tutti i componenti Swing contenuti nel container
- ogni top-level container ha un content pane che contiene (direttamente o indirettamente) i componenti visibili della GUI

Un'applicazione con GUI Swing ha almeno una gerarchia di contenimento con un `JFrame` come radice

- un `JFrame` identifica una finestra (con titolo e bordo) dell'applicazione

Contenuti

La classe `JFrame`: esempio

```
import javax.swing.*;

public class GUIApp {

    public static void main(String[] args) {
        // Crea una finestra con titolo "Hello World"
        JFrame frm = new JFrame("Hello World!");
        // Rende la finestra visibile (di default non lo e')
        frm.setVisible(true);
        System.out.println("Ciao ciao");
    }
}
```

Cosa succede?

1. viene creato l'oggetto `frm` di classe `JFrame` la cui finestra corrispondente è inizialmente *non visibile*;
2. la finestra viene resa visibile;
3. il metodo `main` continua la propria esecuzione e stampa su schermo `Ciao ciao`, poi termina;
   - perché la finestra resta visibile? Dettagli più avanti nel corso...

Chiusura delle finestre

Il metodo `setDefaultCloseOperation()` della classe `JFrame` imposta il comportamento da adottare quando il pulsante di chiusura di un frame viene premuto

Abbiamo 4 possibili alternative, corrispondenti all'assegnazione di una delle seguenti costanti come parametro attuale del metodo:

1. `JFrame.DO_NOTHING_ON_CLOSE`: l'attivazione del pulsante di chiusura della finestra non ha alcun effetto
2. `JFrame.HIDE_ON_CLOSE`: nasconde la finestra, mantenendola in memoria per possibili accessi futuri (e.g., usando `setVisible(true)`)
3. `JFrame.DISPOSE_ON_CLOSE`: elimina la finestra liberando la memoria allocata (non sarà più possibile accedervi)
4. `JFrame.EXIT_ON_CLOSE`: forza la chiusura incondizionata del programma (corrisponde a `System.exit(0)`)

Chiusura di programma con interfaccia grafica

In questo corso identificheremo un'applicazione grafica con il frame principale

- alla chiusura del frame principale vogliamo far corrispondere la chiusura dell'applicazione
- per fare ciò aggiungiamo l'istruzione:

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

  - definiamo che l'applicazione deve terminare *incondizionatamente* quando il pulsante di chiusura del frame principale viene premuto

Esempio

```
import javax.swing.*;

public class GUIApp {

    public static void main(String[] args) {

        // Crea una finestra con titolo "Hello World"
        JFrame frm = new JFrame("Hello World!");

        // Si configura il frame in modo che l'applicazione termini
        // quando il pulsante di chiusura del frame viene premuto
        frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Rende la finestra visibile (di default non lo e')
        frm.setVisible(true);
        System.out.println("Ciao ciao");
    }
}
```

Metodi principali della classe `JFrame` (1/2)

Principali metodi della classe `JFrame`, alcuni dei quali ereditati da classi antenate (`Component`, `Container`, `Window`, `Frame`):

- `JFrame()`: costruttore senza argomenti, crea un oggetto di classe `JFrame` (inizialmente invisibile) senza titolo
- `JFrame(String titolo)`: costruttore ad un argomento, crea un oggetto di classe `JFrame` (inizialmente invisibile) con titolo `titolo`
- `setTitle(String titolo)`: ereditato da `Frame`, assegna il titolo all'oggetto di invocazione
- `void setSize(int lrg, int alt)`: ereditato da `Component`, assegna le dimensioni specificate (misurate in px) alla finestra
- `void setLocation(int x, int y)`: ereditato da `Component`, assegna la posizione specificata (coordinate in px), con origine in alto a sinistra

Metodi principali della classe `JFrame` (2/2)

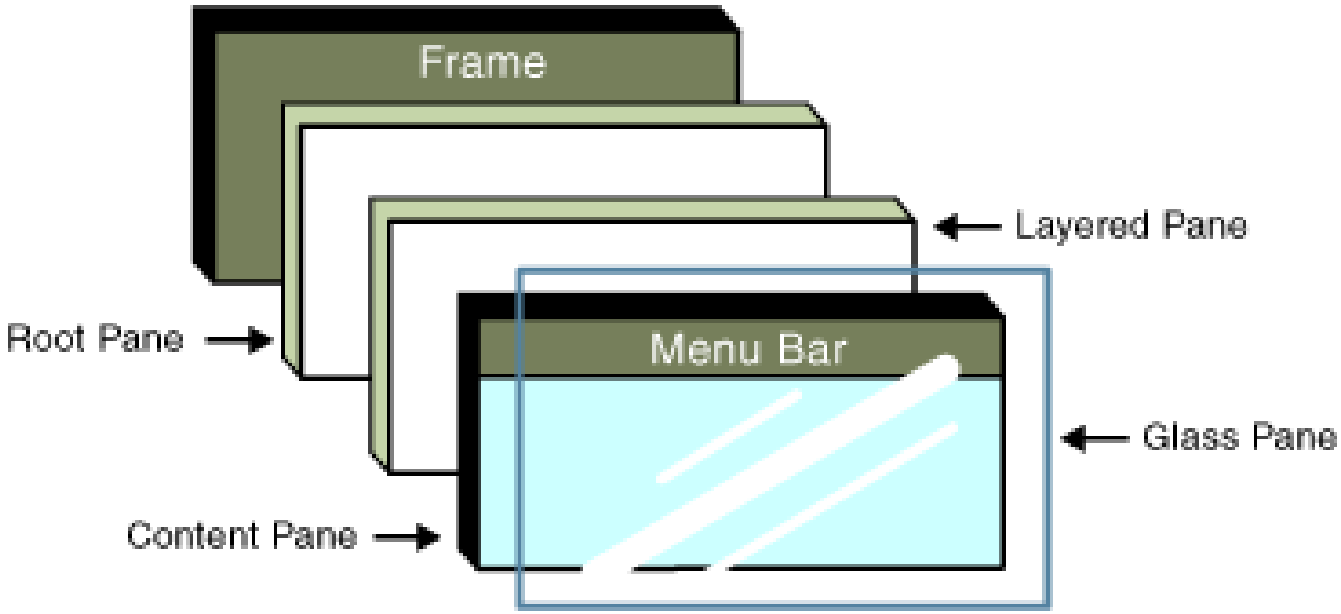
- `void pack()`: ereditato da `Window`, ottimizza le dimensioni della finestra, in base alle dimensioni *preferite* dei componenti in essa contenuti
- `void setVisible(boolean b)`: ereditato da `Component`, visualizza (`b==true`) o nasconde (`b==false`) la finestra
- `void setDefaultCloseOperation(int op)`: imposta il comportamento da adottare all'attivazione del pulsante di chiusura della finestra

Altri metodi importanti illustrati nel seguito

- `Container getContentPane()`
- `void setContentPane(Container contentPane)`
- `void setJMenuBar(JMenuBar menubar)`

Per una lista esaustiva, consultare la documentazione Java
`http://docs.oracle.com/javase/6/docs/api/javax/swing/JFrame.html`

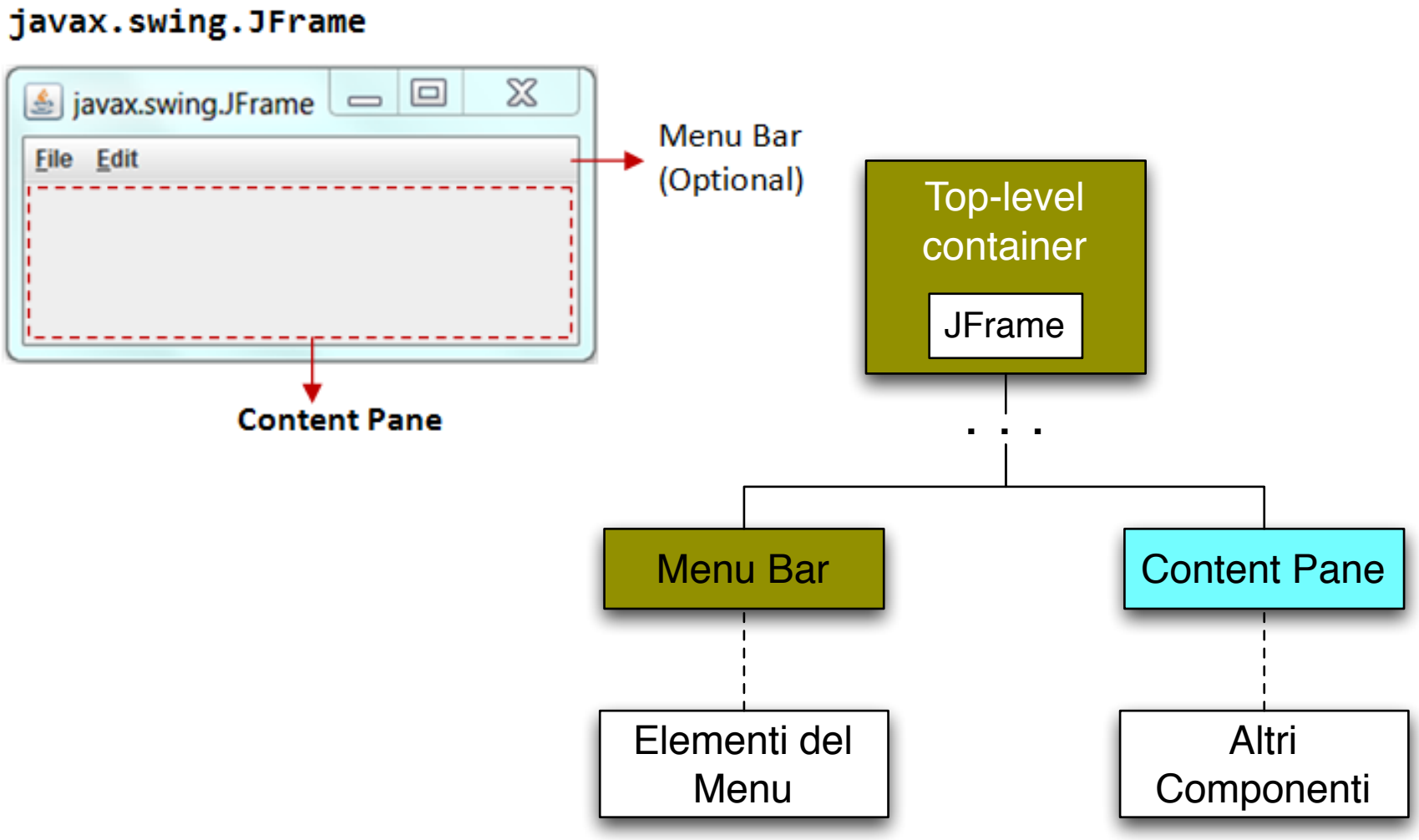
Struttura a livelli di un oggetto `JFrame`



Ciascuno strato svolge una funzione particolare

- ai fini di questo corso è sufficiente considerare il Content Pane
  - per creare finestre più ricche con `JFrame`, che contengano pulsanti, aree di testo, caselle opzione, etc., occorre accedere al content pane dell'oggetto `JFrame`
- gli altri strati permettono di implementare funzionalità avanzate (non usate in questo corso)

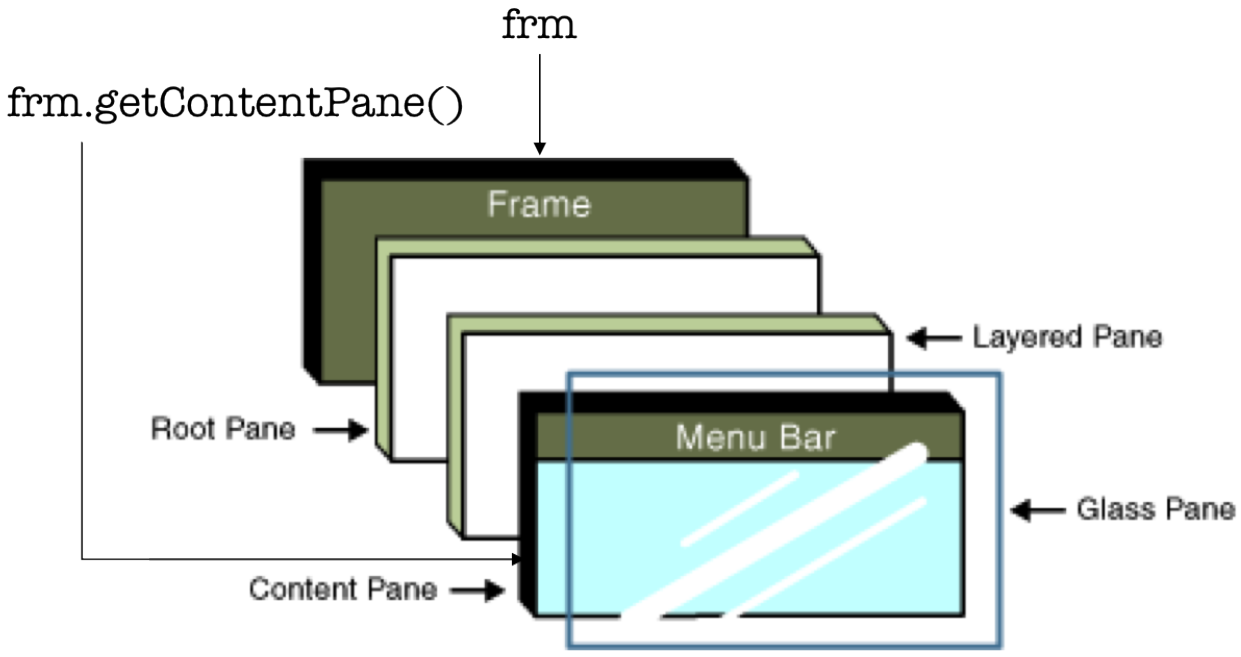
Struttura a livelli e gerarchia



Popolare un `JFrame` (1/2)

- Popolare un oggetto `JFrame` significa aggiungervi nuovi componenti grafici quali bottoni, aree di testo, etc.
  - per essere visibile sullo schermo, ogni componente grafico deve far parte di una gerarchia di contenimento, cioè un albero di componenti che ha come radice un top-level container (come un `JFrame`)
  - ogni top-level container ha un *content pane* che contiene (direttamente o indirettamente) i componenti visibili
- Un oggetto `JFrame` viene popolato aggiungendo componenti grafici al suo content pane
- Per fare ciò dobbiamo innanzitutto ottenere un riferimento al Content Pane del `JFrame`
  - il metodo `getContentPane()` della classe `JFrame` svolge esattamente questa funzione, restituendo un riferimento ad un oggetto, il content pane, appunto, di tipo `Container` (definito nel package AWT)

Popolare un `JFrame` (2/2)



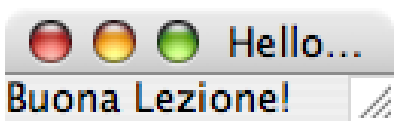
Possiamo ora aggiungere elementi al content pane dell'oggetto `frm` tramite il metodo di classe `Container`:

```
Component add(Component comp)
```

che aggiunge il `Component` passato come argomento al content pane oggetto di invocazione

Esempio 1

```
import javax.swing.*;
// Importa AWT per usare la classe Container
import java.awt.*;

public class GUIApp {

    public static void main(String[] args) {
        JFrame frm = new JFrame("Hello World!");
        frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Ottieni il riferimento al Content Pane
        Container frmContentPane = frm.getContentPane();

        // Usa frmContentPane per aggiungere elementi grafici
        frmContentPane.add(new JLabel("Buona Lezione!"));

        frm.setVisible(true);
    }
}
```



NOTA: le dimensioni sono quelle di default ed il titolo non vi è contenuto per intero

Esempio 2

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

public class GUIApp {

    public static void main(String args[]) {
        JFrame frm = new JFrame("Prima finestra");
        frm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // Imposta la dimensione del frame
        frm.setSize(200,200);
        Container frmContentPane = frm.getContentPane();
        frmContentPane.add(new JLabel("Buona Lezione"));
        frm.setVisible(true);
    }
}
```

Modularità delle interfacce grafiche

Vogliamo evitare di fare gestire al modulo `main()` tutti gli aspetti dell'interfaccia grafica (come avviene invece negli esempi precedenti)

- per fare ciò, decidiamo di costruire una classe per ogni finestra e delegare a ciascuna di esse le operazioni interne che riguardano dettagli implementativi (costruzione dei componenti grafici, creazione delle gerarchie di contenimento, etc.)
- le nuove classi saranno ottenute estendendo `JFrame`, in modo da averne a disposizione tutte le funzionalità

NOTA: La progettazione delle interfacce grafiche è un argomento piuttosto articolato che esula dagli scopi del corso; qui proponiamo un approccio semplificato

Modularità delle interfacce grafiche

Con questo approccio, l'esempio 2 visto prima diventa

File MyFrame.java

```
import javax.swing.*; // AWT non serve (perche'?)

public final class MyFrame extends JFrame {

    private static final String titolo = "Prima Finestra";
    private static final JLabel testo = new JLabel("Buona Lezione");
    private static final int larghezza=200, altezza=200;

    public MyFrame() {
        super(titolo);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(larghezza,altezza);
        this.getContentPane().add(testo);
                  // Abbreviato: evitiamo il Container
        this.setVisible(true);
    }
}
```

Modularità delle interfacce grafiche

File GUIApp.java

```
public class GUIApp {

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

Contenuti

Layout Managers (1/3)

La disposizione di default degli elementi in un container è definita dal Layout Manager del container stesso

Un Layout Manager è un oggetto che implementa l'interfaccia `LayoutManager` e determina il dimensionamento e posizionamento dei componenti all'interno di un `Container`

Esistono numerose implementazioni predefinite di `LayoutManager` ciascuna corrispondente ad una politica di posizionamento dei componenti grafici in un container

- qui ne vedremo solo alcune: `FlowLayout`, `GridLayout`, `BorderLayout`
- vedi `http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html` per info aggiuntive e altri layout manager

Layout Managers (2/3)

Ad ogni `Container` è associata un'implementazione di `LayoutManager` che definisce la disposizione di default dei componenti

In genere, gli unici container per cui può essere necessario impostare un layout manager (diverso da quello di default) sono:

- i content pane, il cui layout manager predefinito è di tipo `BorderLayout`
- i componenti `JPanel`, il cui layout manager predefinito è di tipo `FlowLayout`
  - un `JPanel` è un componente che svolge il ruolo di container per altri componenti grafici (bottoni, aree di testo, etc.)

Layout Managers (3/3)

Per assegnare ad un container (content pane o `JPanel`) un particolare `LayoutManager` (diverso da quello di default), la classe `Container` mette a disposizione il metodo:

```
void setLayout(LayoutManager manager)
```

Ad esempio, se vogliamo disporre i componenti nel content pane di un `JFrame` secondo la politica definita dal `FlowLayout`:

```
// ...
frm = new JFrame(titolo);
Container frmContentPane = frm.getContentPane();
frm.setLayout(new FlowLayout());
// ...
```

Layout Managers: `FlowLayout`

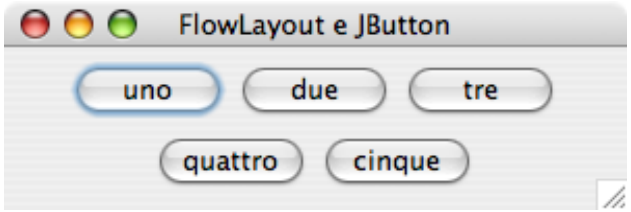
La classe `FlowLayout` implementa un layout manager che dispone i componenti grafici in un container "riga per riga", secondo la seguente politica di posizionamento:

- ogni componente viene inserito in una riga (da sinistra verso destra) finché vi è sufficiente spazio (in base alla larghezza del container)
- il primo componente che non ha spazio nella riga corrente viene posizionato nella riga successiva
- di default, i componenti sono centrati orizzontalmente nel container

Costruttori principali:

- `FlowLayout()`: crea un `FlowLayout` con allineamento centrale
- `FlowLayout(int align)`: crea un `FlowLayout` e consente di specificare (parametro `align`) l'allineamento dei componenti, scegliendo tra `FlowLayout.LEADING` (allineamento a sinistra), `FlowLayout.CENTER` (allineamento al centro), o `FlowLayout.TRAILING` (allineamento a destra)

FlowLayout: esempio




```
public final class MyFrame extends JFrame {

    private static final String titolo = "FlowLayout e JButton";
    private static final int larghezza=300, altezza=100;
    private static final JButton uno = new JButton("uno");
    // ... due, tre, quattro ...
    private static final JButton cinque = new JButton("cinque");

    public MyFrame() {
        super(titolo);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(larghezza,altezza);
        Container frmContentPane = this.getContentPane();
        frmContentPane.setLayout(new FlowLayout());
        frmContentPane.add(uno);
        // ... due, tre, quattro ...
        frmContentPane.add(cinque);
        this.setVisible(true);
} }
```

Layout Managers: `GridLayout`

La classe `GridLayout` implementa un layout manager che dispone i componenti grafici in un container secondo una griglia di celle, procedendo riga per riga da sinistra verso destra e dall'alto verso il basso

- ogni componente riempie interamente lo spazio a disposizione nella propria cella
- tutte le celle hanno la stessa dimensione (determinata dall'elemento di dimensione massima) ed occupano tutto lo spazio disponibile nel container

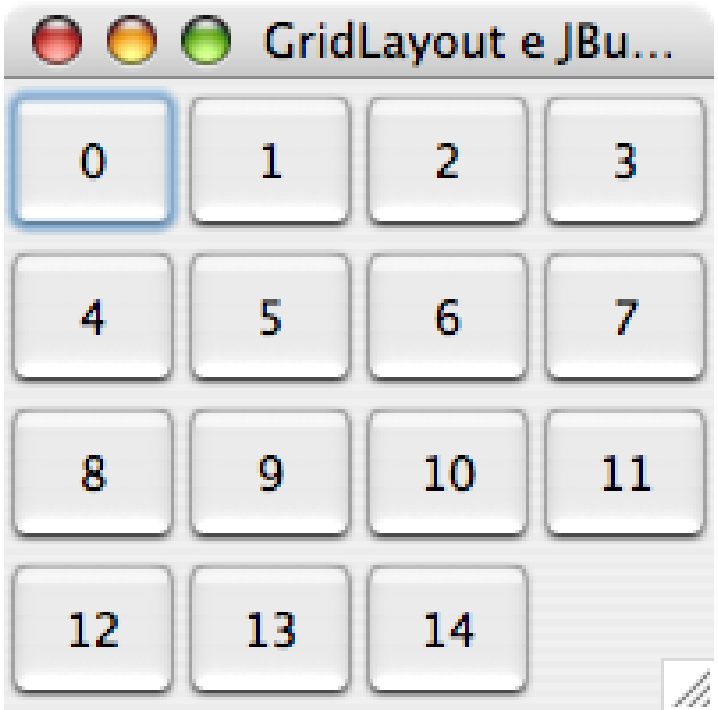
Costruttore principale:

- `GridLayout(int righe, int colonne)`: crea un `GridLayout` con il numero specificato di righe e colonne (n° celle = `righe` x `colonne`)

Layout Managers: `GridLayout`

Ad esempio, in una griglia con 3 righe e 4 colonne (12 celle) i componenti vengono disposti secondo questo schema

1	2	3	4
5	6	7	8
9	10	11	12

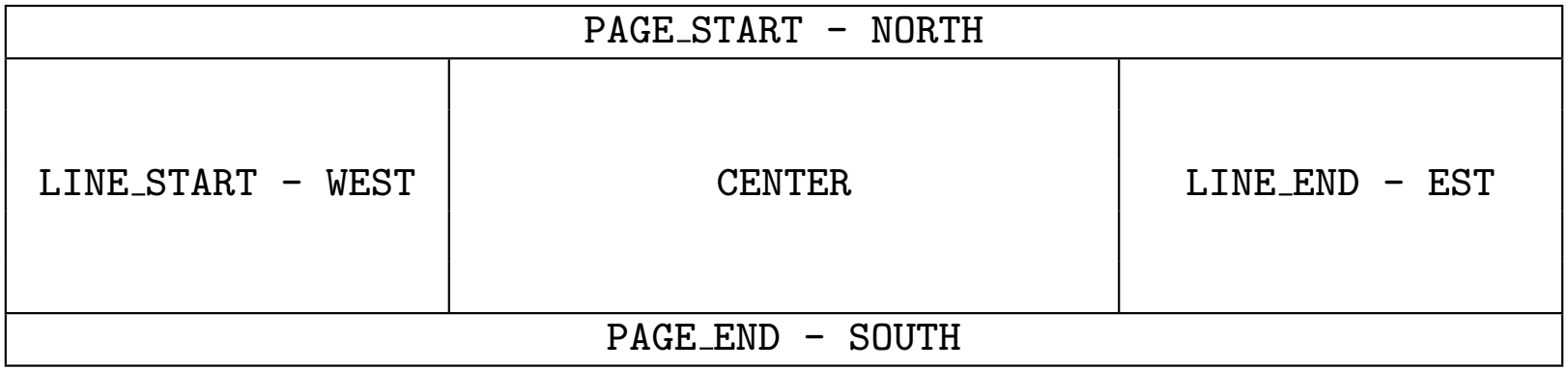


GridLayout: esempio

```
public final class MyFrame extends JFrame {

    private static final String titolo = "GridLayout e JButton";
    private static final int larghezza=200, altezza=200;
    private static final int righe=4, colonne=4;

    public MyFrame() {
        super(titolo);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(larghezza,altezza);
        Container frmContentPane = this.getContentPane();
        frmContentPane.setLayout(new GridLayout(righe, colonne));
        for (int i = 0;  i<15;  i++) {
            frmContentPane.add(new JButton(String.valueOf(i)));
        }
        this.setVisible(true);
    }
}
```

Layout Managers: BorderLayout

La classe `BorderLayout` implementa un layout manager che permette di disporre i componenti suddividendo il container in cinque aree



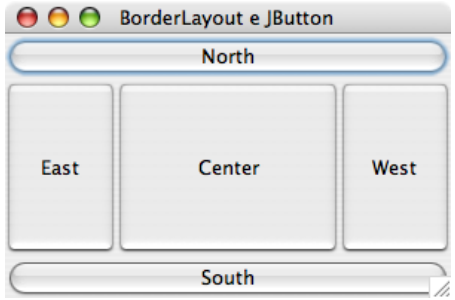
Costruttore principale: `BorderLayout()`

Per un container con layout manager di tipo `BorderLayout` (es. il content pane di un `JFrame`), il metodo `void add(Component c, String pos)` consente di aggiungere componenti specificandone il posizionamento secondo lo schema illustrato sopra

Layout Managers: `BorderLayout`

- il parametro `pos` può assumere i valori: `BorderLayout.CENTER`, `BorderLayout.PAGE_START` (o `BorderLayout.NORTH`), `BorderLayout.PAGE_END` (o `BorderLayout.SOUTH`), `BorderLayout.LINE_START` (o `BorderLayout.WEST`), `BorderLayout.LINE_END` (o `BorderLayout.EAST`)
- aree senza elementi non vengono visualizzate e la finestra viene adattata di conseguenza

BorderLayout: esempio

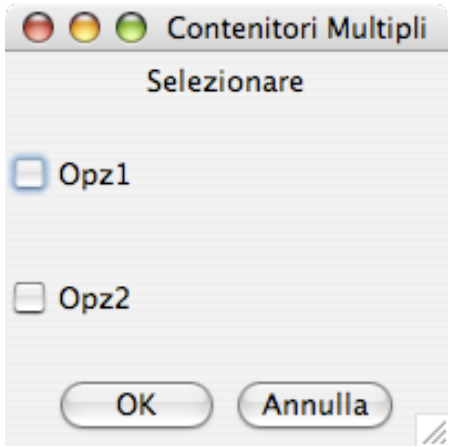



```
public final class MyFrame extends JFrame {
    private static final String titolo = "BorderLayout e JButton";
    private static final int larghezza=300, altezza=200;
    private static final JButton north = new JButton("North");
    // south, east, ...

    public MyFrame() {
        super(titolo);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(larghezza,altezza);
        Container frmContentPane = this.getContentPane();
                                        //ha borderlayout di default
        frmContentPane.add(north, BorderLayout.PAGE_START);
        frmContentPane.add(south, BorderLayout.PAGE_END);
        // south, east, etc...
        this.setVisible(true);
} }
```

Contenuti

Container annidati e JPanel

Per creare interfacce grafiche più complesse si possono comporre container diversi (secondo il principio della gerarchia di contenimento), ciascuno con un proprio `LayoutManager` che definisce la disposizione degli elementi nel container stesso



Per supportare la definizione di una gerarchia di contenimento tra container usiamo oggetti della classe `JPanel`

- un oggetto di classe `JPanel` è un `JComponent` che può essere aggiunto al content pane di un `JFrame`
- essendo anche un oggetto di tipo `Container`, un `JPanel` può a sua volta contenere componenti grafici e altri `JPanel`
- ogni oggetto `JPanel` può avere un proprio `LayoutManager`

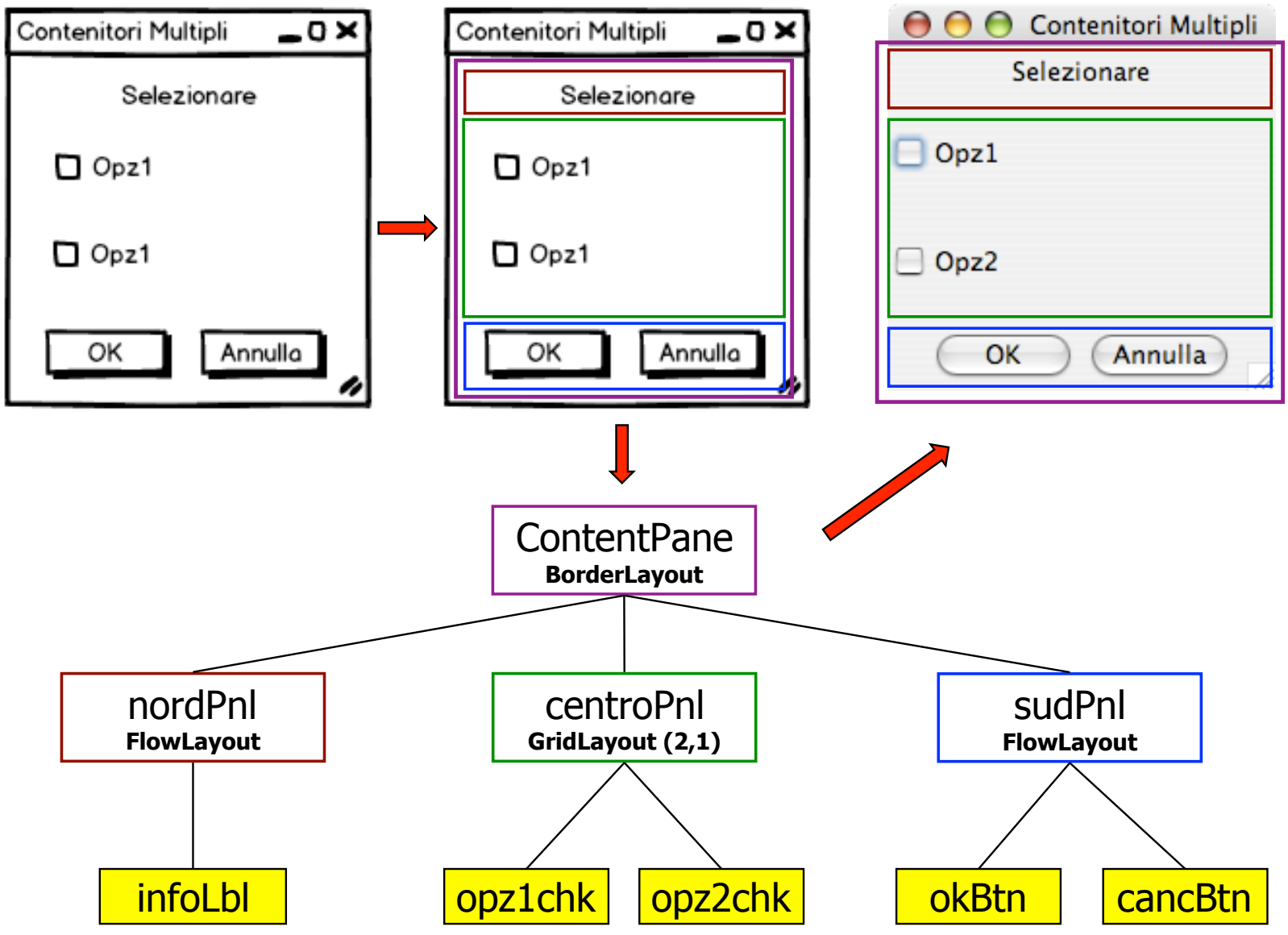
Progettazione della GUI

Adottiamo un approccio top-down alla progettazione: partendo dal contenitore principale, procediamo verso i contenitori e i componenti più interni

1. Si assegna un layout manager al content pane del `JFrame` che rappresenta la finestra da realizzare (di default è `BorderLayout`)
2. Per ciascuna area che si vuole aggiungere viene creato un `JPanel`, assegnando a ciascuno di essi un layout manager (di default è `FlowLayout` con allineamento centrale)
3. Ciascun pannello potrà a sua volta contenere altri pannelli o elementi grafici più semplici (`JButton`, `JLabel`, etc.)

Rappresentiamo il risultato di questa fase di progettazione con un albero della GUI (gerarchia di contenimento) annotato con informazioni utili alla realizzazione (es., layout manager adottato)

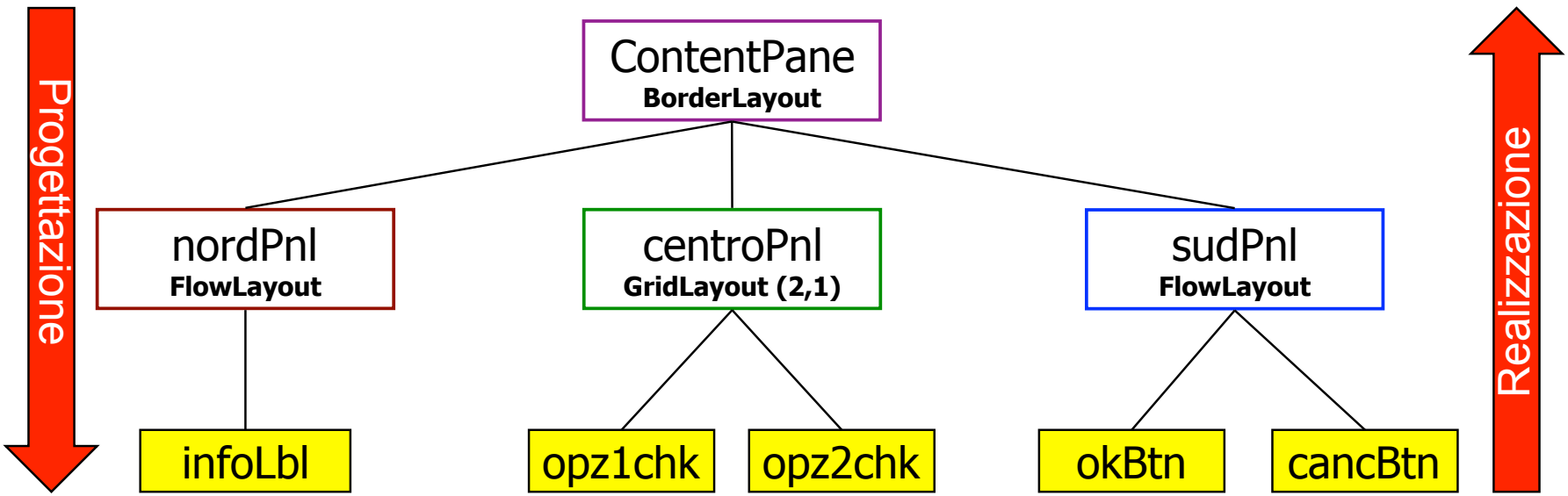
Progettazione della GUI: esempio



Realizzazione della GUI progettata

Nell'implementazione si procede secondo un approccio di tipo bottom-up:

1. costruiamo prima i componenti atomici che rappresentano le foglie dell'albero di contenimento
2. popoliamo i contenitori partendo da quelli più "profondi" nell'albero, fino ad arrivare al content pane radice della gerarchia



Realizzazione: esempio

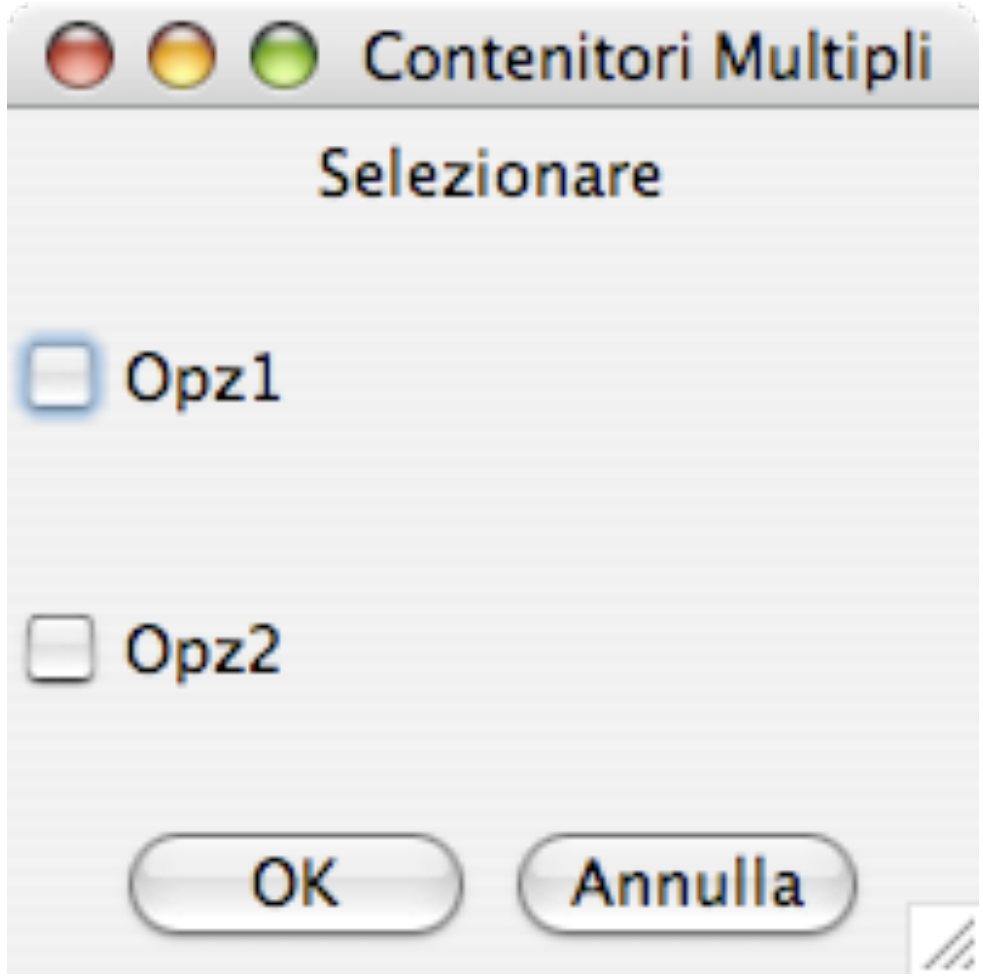
```
public final class MyFrame extends JFrame {

    private static final String titolo = "Contenitori Multipli";

    //Pannello Nord:
    private static final JLabel infoLbl = new JLabel("Selezionare");
    private static final JPanel nordPnl = new JPanel();

    //Pannello Centrale:
    private static final JCheckBox opz1Chk =
                          new JCheckBox("Opz1");
    private static final JCheckBox opz2Chk =
                          new JCheckBox("Opz2");
    private static final JPanel centroPnl =
                          new JPanel(new GridLayout(2,1));

    //Pannello Sud:
    private static final JButton okBtn=new JButton("OK");
    private static final JButton cancBtn=new JButton("Annulla");
    private static final JPanel sudPnl = new JPanel();
```

Realizzazione: esempio

```
//Popoliamo i Container "dal basso verso l'alto" (nel costruttore)

public MyFrame() {

    super(titolo);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Pannello Nord
    nordPnl.add(infoLbl);

    //Pannello Centro
    centroPnl.add(opz1Chk);
    centroPnl.add(opz2Chk);

    //Pannello Sud
    sudPnl.add(okBtn);
    sudPnl.add(cancBtn);
```

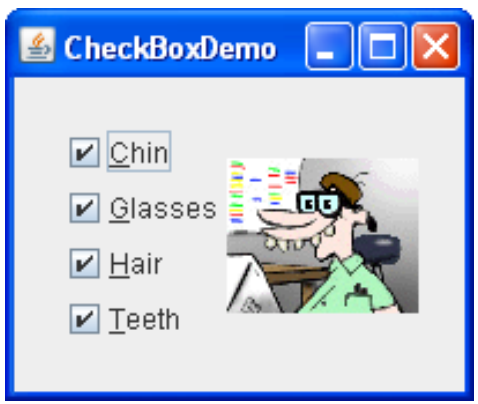
Realizzazione: esempio

```
        //Container Principale
        Container frmContentPane = this.getContentPane();
        frmContentPane.add(nordPnl, BorderLayout.NORTH);
        frmContentPane.add(centroPnl, BorderLayout.CENTER);
        frmContentPane.add(sudPnl, BorderLayout.SOUTH);


        //Impostiamo le proprieta' di visualizzazione
        // Imposta la dimensione minima
        // per visualizzare tutti i componenti
        this.pack();

        // Posizioniamo la finestra al centro dello schermo
        this.setLocationRelativeTo(null);

        // Rendiamo visibile la finestra
        this.setVisible(true);
    }
}
```

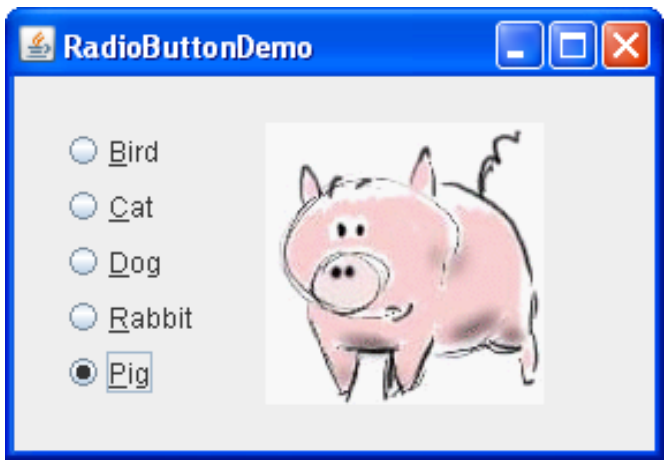

Contenitori Multipli
Selezionare
Opz1
Opz2
OK
Annulla

Alcuni JComponent comuni

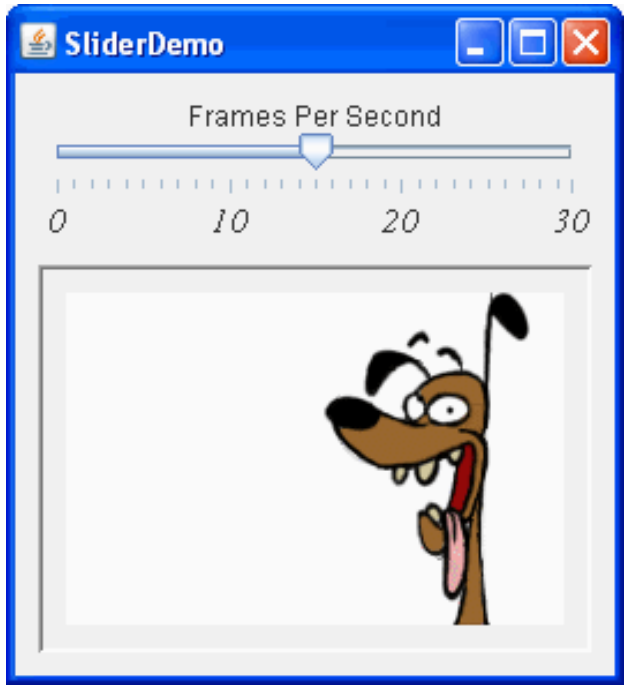
JButton



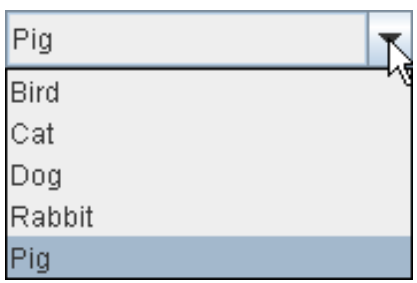
JCheckBox



JRadioButton

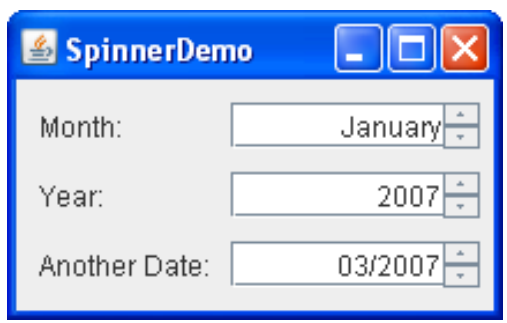


JSlider



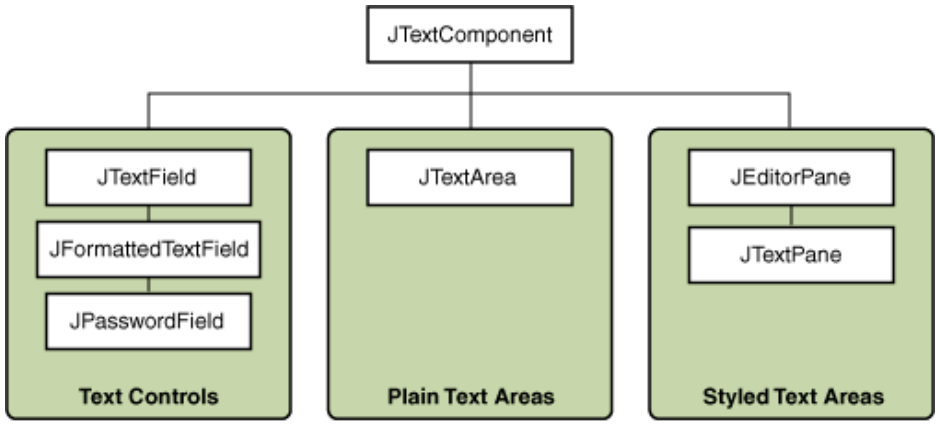
JComboBox

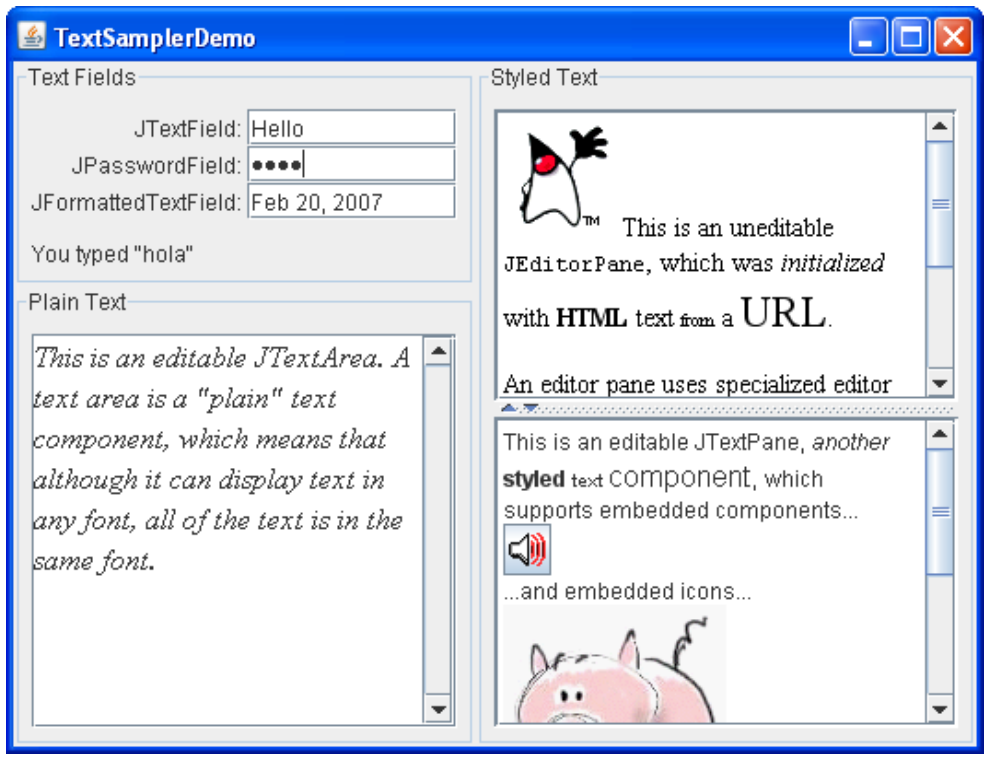
JList



JSpinner

Alcuni JComponent comuni: elementi testuali





Contenuti

I menu e la barra menu (1/3)

- Un menu permette di strutturare gerarchicamente opzioni e funzionalità dell'applicazione
- I menu sono contenuti in una barra menu, posizionata tipicamente nella parte alta di una finestra
- Ciascun `JFrame` può contenere al più una barra menu, identificata da un oggetto `JMenuBar`, che viene creato con il costruttore senza parametri

  `JMenuBar()`

  ed assegnato ad un oggetto `JFrame` tramite il metodo della classe `JFrame`

  `void setJMenuBar(JMenuBar menubar)`

Esempio:

```
JFrame frm = new JFrame("Il mio Frame");
JMenuBar jb = new JMenuBar();
frm.setJMenuBar(jb);
```

I menu e la barra menu (2/3)

- Un oggetto `JMenuBar` può contenere un numero qualsiasi di oggetti `JMenu`, che corrispondono ai menu che compariranno nella barra
- Un oggetto `JMenu` è costruito a partire dal costruttore ad un argomento (il nome del menu)

  ```
  JMenu(String nome)
  ```

  e viene inserito in un oggetto `JMenuBar` tramite il metodo della classe `JMenuBar`

  ```
  JMenu add(JMenu m)
  ```

Esempio:

```
JMenuBar jb = new JMenuBar();
JMenu jm = new JMenu("Edit");
jb.add(jm);
```

I menu e la barra menu (3/3)

- Oggetti di classe `JMenu` possono contenere altri oggetti `JMenu`, in modo da realizzare menu annidati (sottomenu)
- Un sottomenu viene aggiunto ad un menu tramite il metodo della classe `JMenu`

```
Component add(Component)
```

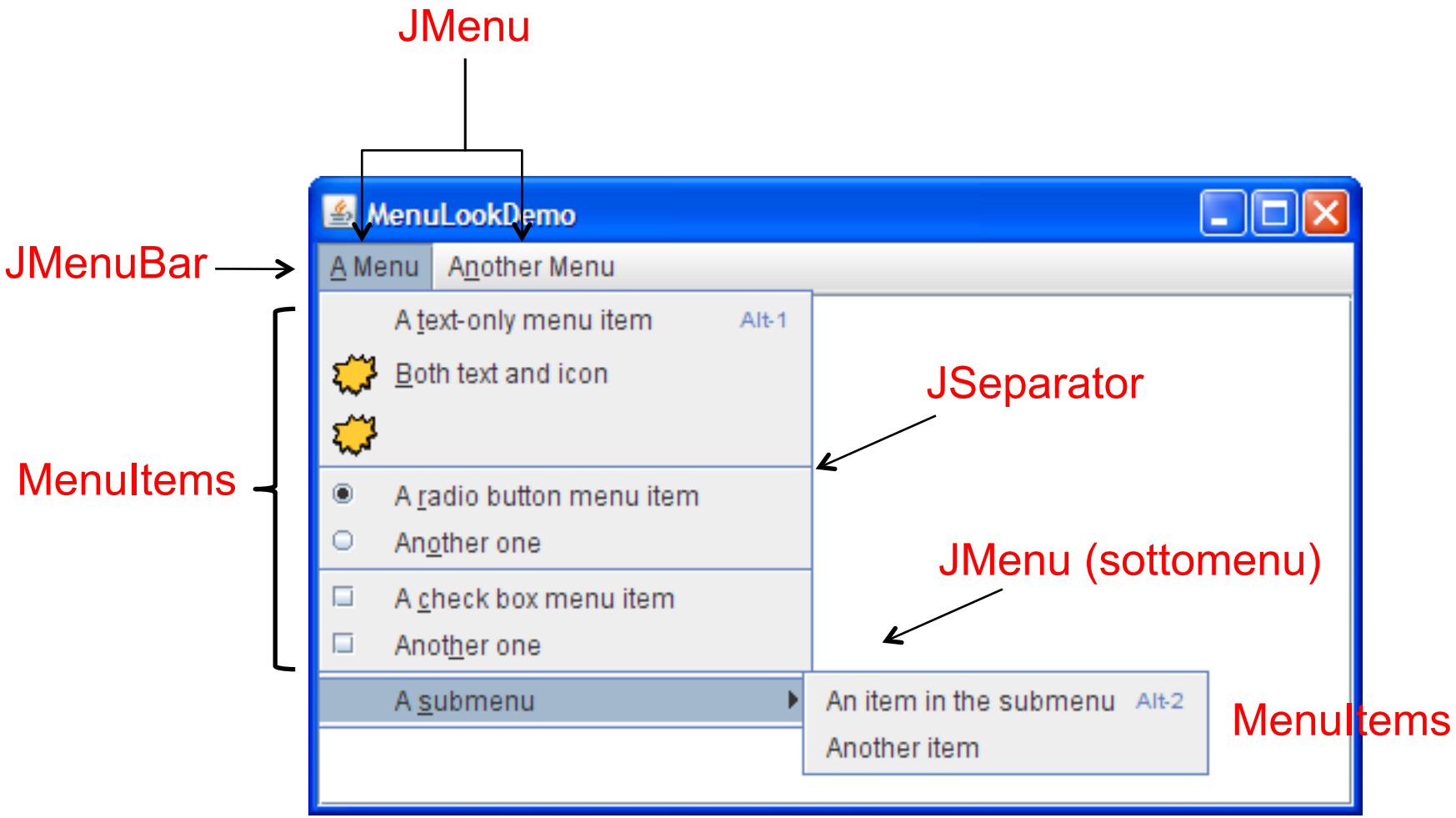
- Inoltre, oggetti di classe `JMenu` possono contenere oggetti di tipo `JMenuItem` che rappresentano le voci del menù
- Un oggetto `JMenuItem` può essere costruito con il costruttore ad un argomento

```
JMenuItem(String voce)
```

  e può essere aggiunto ad un menu con il metodo della classe `JMenu`

```
JMenuItem add(JMenuItem item)
```

Menu ed elementi



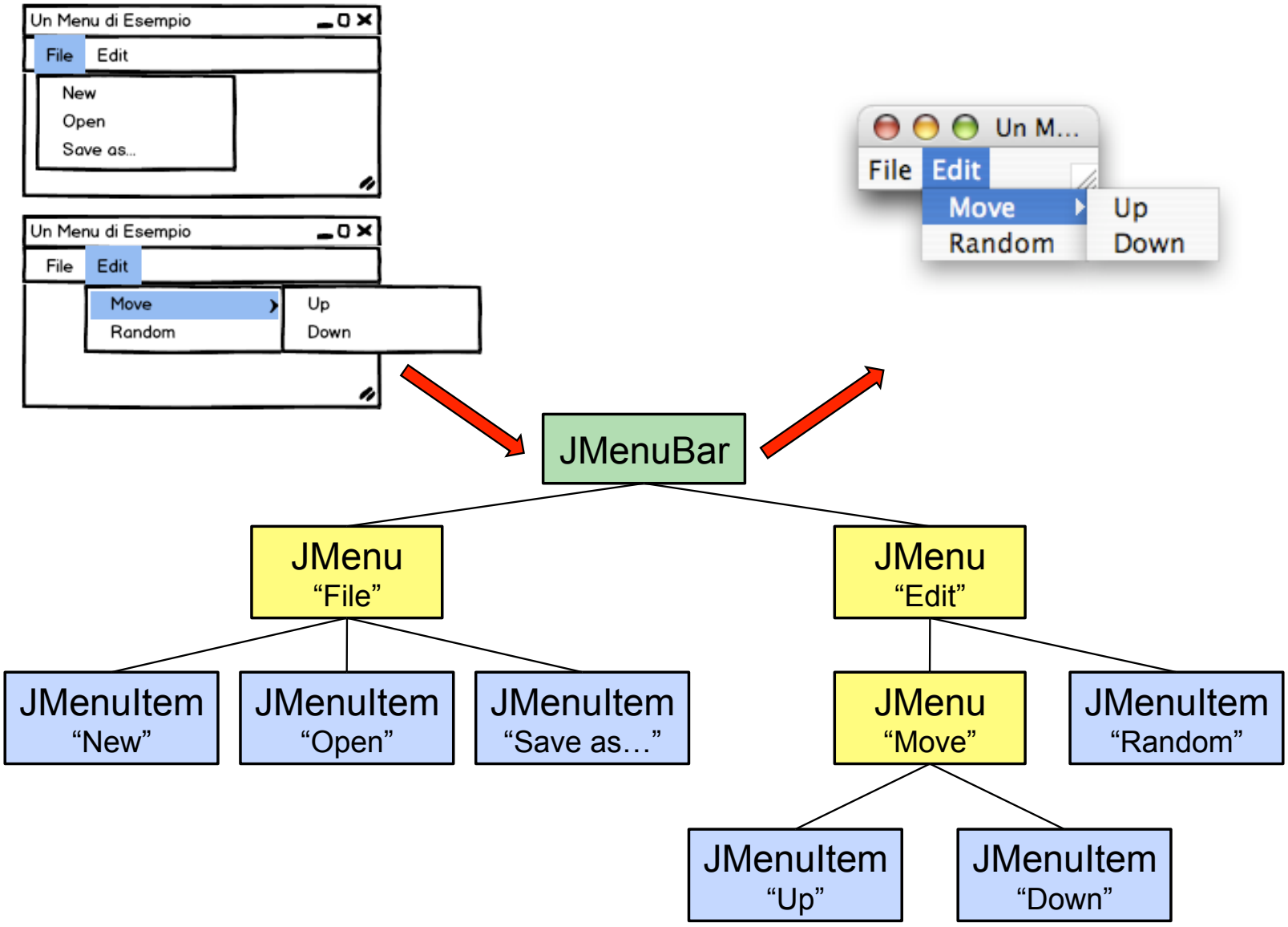
Progettazione dei menu

Anche per i menu adottiamo un approccio top-down alla progettazione: partendo dalla barra menu, definiamo i menu, sottomenu e voci di menu

1. si definiscono tutti i `JMenu` che dovranno apparire nella `JMenuBar`
2. per ciascun `JMenu` si definiscono gli elementi che lo compongono, distinguendo tra sottomenu (`JMenu`) e voci di menu (`JMenuItem`)
3. per ciascun sottomenu si procede come al punto 2, continuando a definire gli elementi di sottomenu fino ad avere solo voci di menu

Rappresentiamo il risultato di questa fase di progettazione con un albero di menu (gerarchia di contenimento) annotato con informazioni utili alla realizzazione (es., nome del (sotto)menu o voce di menu)

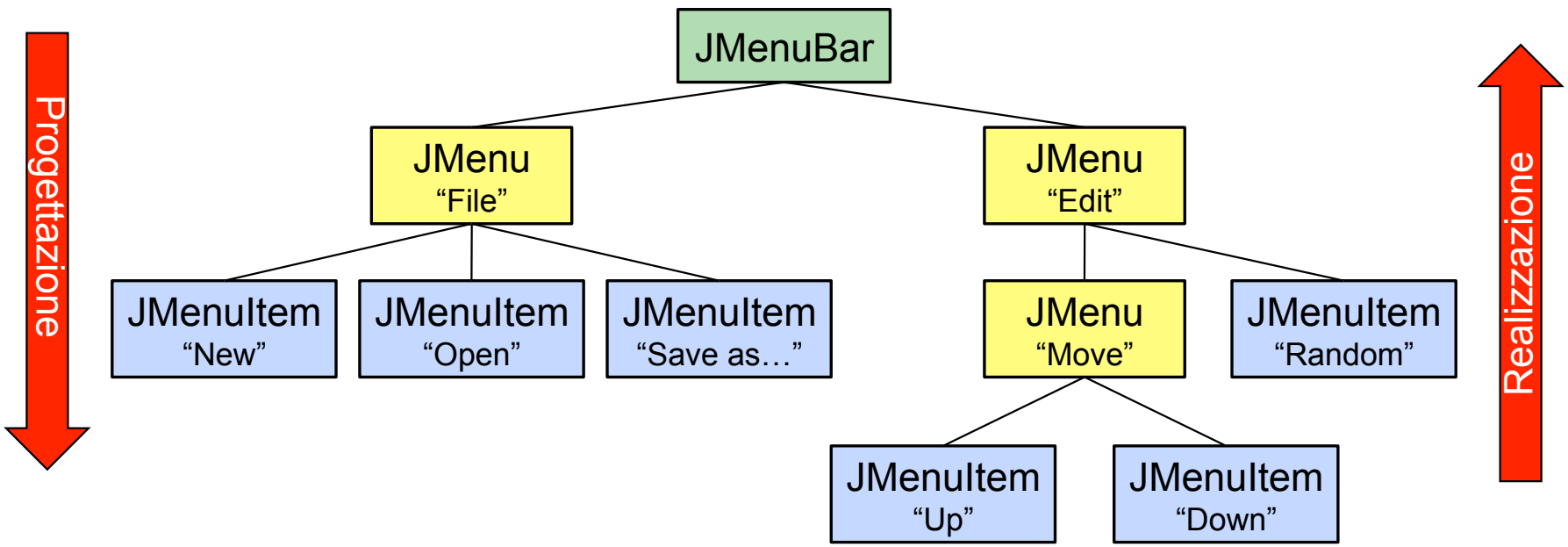
Progettazione dei menu: esempio



Realizzazione dei menu progettati

Nell'implementazione si procede secondo un approccio di tipo bottom-up:

1. costruiamo prima le voci di menu che rappresentano le foglie dell'albero di contenimento
2. costruiamo i (sotto)menu partendo da quelli più "profondi" nell'albero, fino ad arrivare alla barra menu



Realizzazione di menu: esempio

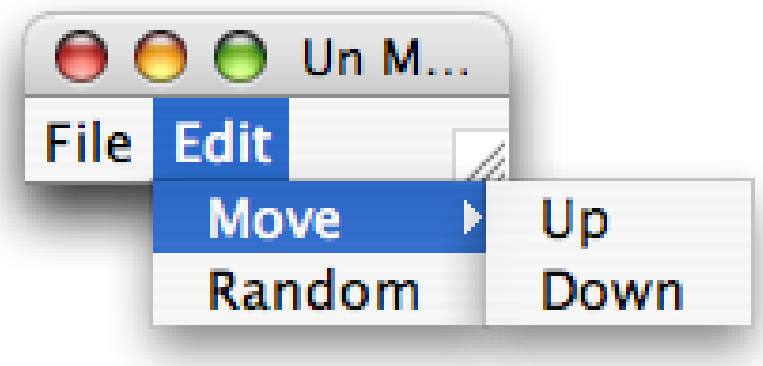
```
public class MyMenu extends JFrame {

    private static final String titolo = "Un Menu di Esempio";

    //Menu Move
    private static final JMenuItem upIt = new JMenuItem("Up");
    private static final JMenuItem dwnIt = new JMenuItem("Down");
    private static final JMenu moveMenu = new JMenu("Move");

    //Menu di Editing
    private static final JMenuItem rndmIt = new JMenuItem("Random");
    private static final JMenu editMenu = new JMenu("Edit");

    //Menu File
    private static final JMenuItem newIt = new JMenuItem("New");
    private static final JMenuItem openIt = new JMenuItem("Open");
    private static final JMenuItem saveIt = new JMenuItem("Save as...");
    private static final JMenu fileMenu = new JMenu("File");

    //Menu Bar
    private final JMenuBar menuBar = new JMenuBar();
```

Realizzazione di menu: esempio

```
// Popoliamo i menu "dal basso verso l'alto" (nel costruttore)

public MyMenu() {

    super(titolo);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Move Menu
    moveMenu.add(upIt);
    moveMenu.add(dwnIt);

    // Edit Menu
    editMenu.add(moveMenu);
    editMenu.add(rndmIt);

    // File Menu
    fileMenu.add(newIt);
    fileMenu.add(openIt);
    fileMenu.add(saveIt);
```

Realizzazione di menu: esempio

```
        // Barra menu
        menuBar.add(fileMenu);
        menuBar.add(editMenu);

        // Imposta la Barra Menu nel frame
        this.setJMenuBar(menuBar);

        // Impostazioni di visualizzazione
        this.pack();
        this.setLocationRelativeTo(null);
        this.setVisible(true);
    }
}
```



Contenuti

Componenti grafici e interazioni

Finora abbiamo visto come progettare e realizzare interfacce grafiche con Swing focalizzandoci sugli aspetti di visualizzazione

- finestre, menu, layout, disposizione dei componenti, ecc...

Molti dei componenti grafici consentono l'interazione con l'utente

- è possibile premere bottoni, posizionare il cursore su una determinata area, premere pulsanti del mouse o tastiera, selezionare elementi o voci di menu, inserire testo, ecc...

Come possiamo collegare i componenti grafici con le possibili interazioni dell'utente e la logica applicativa?

- ad esempio, alla pressione di un bottone da parte dell'utente vogliamo far corrispondere azioni quali l'apertura di una nuova finestra, l'avvio una computazione, l'acquisizione di valori scritti in una casella di testo, ecc...

Il modello ad eventi

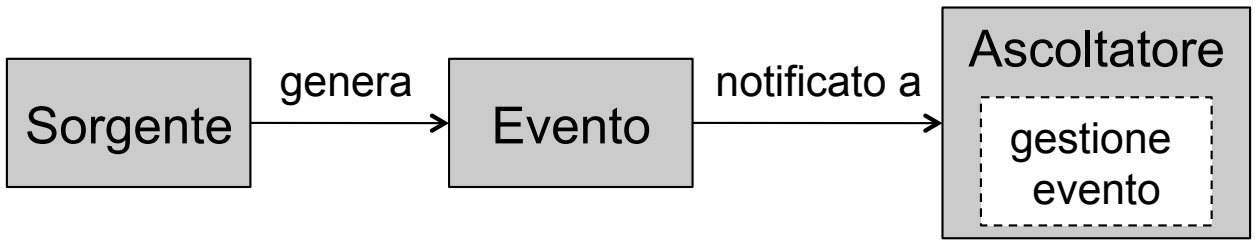
Le fuzionalità ed interazioni con applicazioni dotate di interfaccia grafica Swing sono guidate dall'occorrenza di eventi (*event-driven*)

- l'applicazione reagisce a differenti tipi di eventi generati durante l'esecuzione
- gli eventi possono essere generati dall'utente o da altre sorgenti (es. connessioni di rete, timers, ecc.)

In un modello basato su eventi distinguiamo tra:

1. sorgenti di eventi: gli oggetti il cui stato cambia (es., bottone viene premuto)
2. eventi: oggetti che rappresentano e incapsulano le informazioni relative al cambiamento di stato delle sorgenti
3. ascoltatori di eventi: oggetti cui devono essere notificati gli eventi

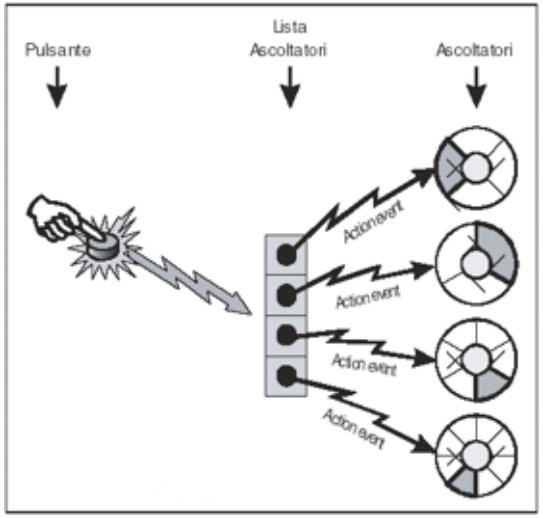
Event delegation (1/2)



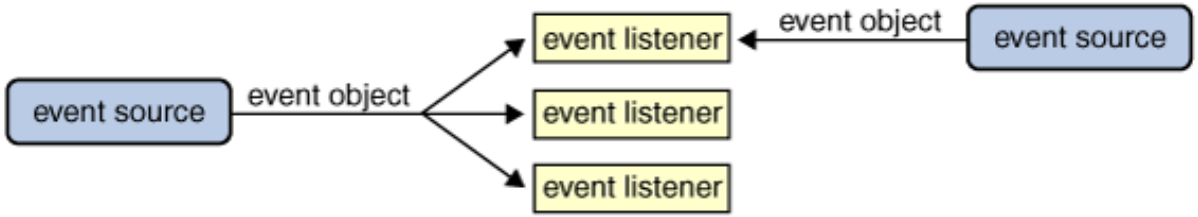
- Ogni volta che una sorgente genera un evento, l'evento generato viene notificato agli ascoltatori interessati
- Gli ascoltatori eseguono opportune azioni in risposta all'evento ricevuto
- La sorgente di fatto delega agli ascoltatori la gestione degli eventi che essa genera

Per implementare questo meccanismo è necessario associare ad ogni componente che può generare eventi (es. bottone) uno o più ascoltatori (event listener) in grado di gestire tali eventi

Event delegation (2/2)



- Ciascun componente (sorgente) può avere più ascoltatori associati ad uno stesso tipo di evento
- Diversi componenti possono avere uno stesso ascoltatore associato ad uno stesso tipo evento



La gestione degli eventi (1/2)

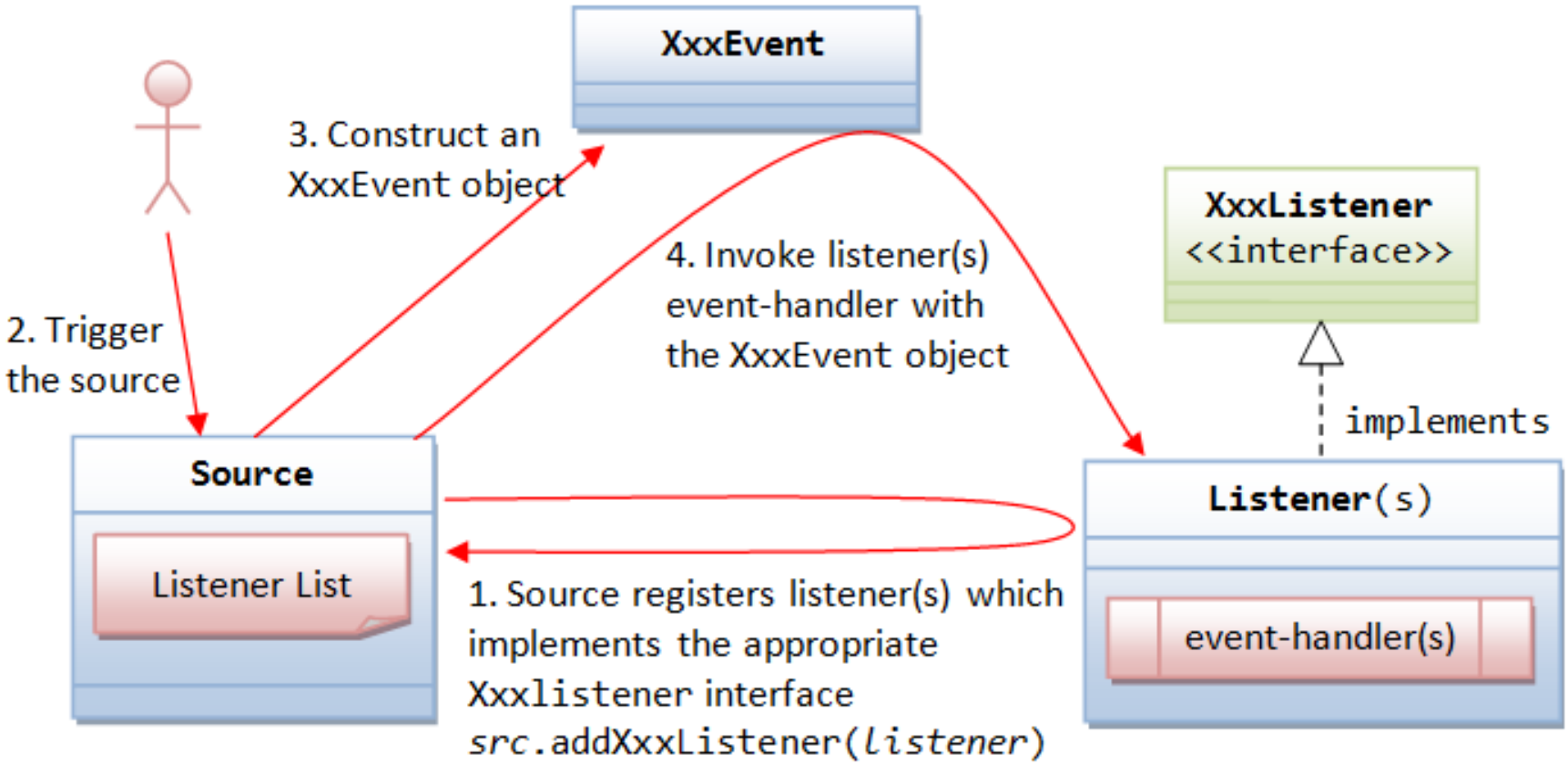
Per realizzare il meccanismo di gestione degli eventi

- identificare il tipo di eventi da gestire
- implementare un'opportuna interfaccia Listener con metodi che realizzino le funzionalità per la ricezione e gestione di eventi (event handlers) di quel tipo (responsabilità del programmatore)

L'interazione tra componenti avviene poi come segue (vedi figura successiva):

1. si registra un'istanza del Listener presso il componente (sorgente) che genera gli eventi di interesse (responsabilità del programmatore)
   - ogni sorgente gestisce una lista di ascoltatori
2. l'utente interagisce con una sorgente di eventi
3. la sorgente crea un oggetto che rappresenta l'evento
4. l'evento viene notificato a tutti gli ascoltatori registrati presso la sorgente
   - ciò determina l'esecuzione della logica di gestione dell'evento (event handler) definita nell'implementazione del Listener

La gestione degli eventi (2/2)



Source: http://www.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html

Esempio: eventi `MouseEvent`

Un oggetto di classe `MouseEvent` (package `java.awt.event`) viene istanziato ogni volta che avviene un evento del mouse (o dispositivo simile, es. touchpad) rispetto ad un certo componente

- il cursore entra o esce dall'area visibile occupata da un componente
- l'utente preme, rilascia o clicca (preme e rilascia) un pulsante del dispositivo

Un oggetto `MouseEvent` contiene informazioni sull'evento generato, accessibili tramite opportuni metodi, ad esempio:

- i metodi `int getX()` e `int getY()` restituiscono le coordinate del mouse relative al componente in cui l'evento è stato generato
- il metodo `int getButton()` permette di conoscere l'eventuale pulsante che ha originato l'evento (`MouseEvent.NOBUTTON`, `MouseEvent.BUTTON1`, `MouseEvent.BUTTON2`, `MouseEvent.BUTTON3`)
- il metodo `int getClickCount()` restituisce il numero di click associati all'evento

Esempio: l'interfaccia `MouseListener`

L'interfaccia `MouseListener` (package `java.awt.event`) definisce metodi per la gestione degli eventi `MouseEvent` generati dal mouse

```
public interface MouseListener {
    // Invocato quando si clicca sul componente associato
    void mouseClicked(MouseEvent e);
    // Invocato quando il puntatore del
    // mouse entra nell'area del componente
    void mouseEntered(MouseEvent e);
    // Invocato quando il puntatore del
    // mouse esce dall'area del componente
    void mouseExited (MouseEvent e);
    // Invocato quando un pulsante del
    // mouse viene premuto sul componente
    void mousePressed(MouseEvent e);
    // Invocato quando un pulsante del
    // mouse viene rilasciato sul componente
    void mouseReleased(MouseEvent e);
}
```

Gestione degli eventi del mouse: esempio

Realizziamo un ascoltatore che implementi l'interfaccia `MouseListener`

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

public class MouseSpy implements MouseListener {

    public void mouseClicked(MouseEvent e) {
        System.out.println("Click su
                    (" + e.getX() + "," + e.getY() + ")");
    }

    public void mousePressed(MouseEvent e) {
        System.out.println("Premuto su
                    (" + e.getX() + "," + e.getY() + ")");
    }

    public void mouseReleased(MouseEvent e) {
        System.out.println("Rilasciato su
                    (" + e.getX() + "," + e.getY() + ")");
    }
    public void mouseEntered(MouseEvent e) {} // evento non gestito
    public void mouseExited(MouseEvent e) {} // evento non gestito
}
```

Gestione degli eventi del mouse: esempio

Associamo un'istanza dell'ascoltatore all'intero `JFrame` della GUI

```
import javax.swing.*;

public class MyFrame extends JFrame {

    public MyFrame() {

        super("MouseTest");

        // registriamo l'ascoltatore presso il frame
        this.addMouseListener(new MouseSpy());

        this.setSize(200,200);
        this.setVisible(true);
    }

    public static void main(String[] args) {
        new MyFrame();
    }
}
```

Principali eventi e listener

Evento	Interfaccia Listener	Descrizione interfaccia
`ActionEvent`	`ActionListener`	Definisce un metodo per gestire eventi di tipo "azione" (*vedi slide successive*)
`ComponentEvent`	`ComponentListener`	Definisce 4 metodi per riconoscere quando un componente viene nascosto, spostato, mostrato o ridimensionato
`FocusEvent`	`FocusListener`	Definisce 2 metodi per riconoscere quando un componente ottiene o perde il focus da tastiera
`KeyEvent`	`KeyListener`	Definisce 3 metodi per riconoscere quando un tasto (della tastiera) viene premuto, rilasciato o battuto
`MouseEvent`	`MouseMotion Listener`	Definisce 2 metodi per riconoscere quando il puntatore del mouse è spostato o trascinato su un componente
`MouseEvent`	`MouseListener`	Gestisce gli eventi del mouse (*vedi slide precedenti*)
`TextEvent`	`TextListener`	Definisce un metodo per riconoscere quando il valore di un campo testuale cambia
`WindowEvent`	`WindowListener`	Definisce 7 metodi per riconoscere quando una finestra viene attivata, eliminata, chiusa, disattivata, ripristinata, ridotta a icona, o resa visibile

Gli eventi `ActionEvent` e l'interfaccia `ActionListener`

Gli eventi di tipo `ActionEvent` vengono generati in corrispondenza di azioni originate dall'interazione dell'utente con componenti Swing

- il tipo di azione dipende dallo specifico componente

Ad esempio, le seguenti azioni generano un `ActionEvent`

- click su un bottone
- selezione di una voce di menu
- pressione del tasto "Invio" in un campo di testo
- selezione di un elemento in un combo box o selezione di un radio button

L'interfaccia `ActionListener` (package `java.awt.event`) definisce un solo metodo per la gestione degli eventi `ActionEvent`

```
public interface ActionListener {

    void actionPerformed(ActionEvent e);

}
```

La gestione degli eventi `ActionEvent`

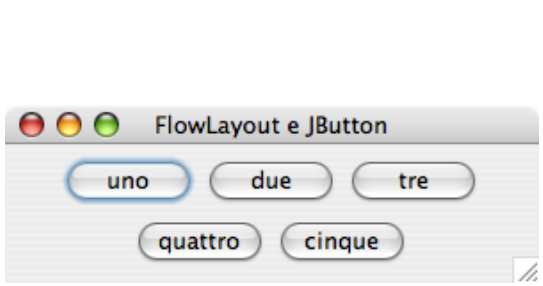
La gestione degli eventi `ActionEvent` segue lo schema generale definito in precedenza ed è basata sull'implementazione di un Action Listener

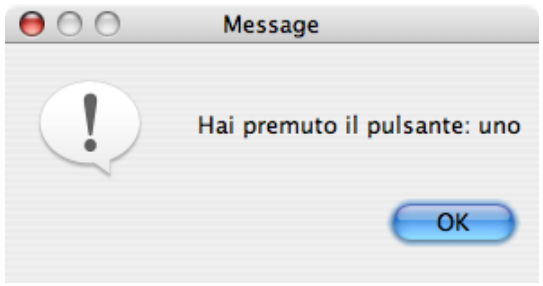
1. definire una classe che implementi l'interfaccia `ActionListener`, specificando la logica di gestione dell'evento nell'implementazione del metodo `void actionPerformed(ActionEvent e)`
2. registrare un'istanza di tale classe come listener per il componente che genera gli eventi di interesse, tramite il metodo

   `void addActionListener(ActionListener l)`
3. quando l'utente interagisce con il componente, viene creato un oggetto `ActionEvent` e viene invocato il metodo `actionPerformed` degli ascoltatori `ActionListener` registrati presso il componente

Implementare un `ActionListener`

Riprendiamo l'esempio introdotto per illustrare il `FlowLayout`





Quando viene premuto un bottone, vogliamo che si apra una finestra informativa con un messaggio (es., che ci dica quale bottone è stato premuto)

Implementare un `ActionListener`

Per prima cosa, implementiamo un `ActionListener` con la logica di gestione degli eventi

- il metodo `actionPerformed()` mostra un messaggio con l'etichetta del pulsante che ha generato l'evento
  - assoceremo il listener a più bottoni che generano eventi

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

public class MyFrameListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
        // gia' sappiamo che assoceremo il Listener ad un bottone...
        // getSource() restituisce un riferimento all'oggetto che ha
        // generato l'evento
        JButton b = (JButton) e.getSource();

        // getText() di JButton restituisce il testo definito come
        // etichetta del bottone
        JOptionPane.showMessageDialog(null,"Hai premuto il pulsante: "
                                                + b.getText());
    }
}
```

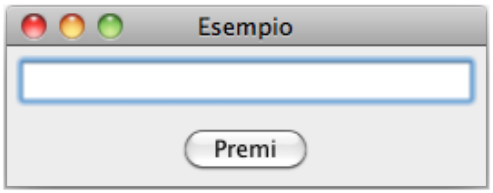
Implementare un ActionListener

Associamo un oggetto di classe `MyFrameListener` a ciascun pulsante definito nell'interfaccia (grafica) originaria; la classe `MyFrame`, pertanto, diventa

```
public final class MyFrame extends JFrame {

    // private static final String titolo, ... larghezza, altezza...
    private static final JButton uno = new JButton("uno");
    // due, ... , cinque

    // Creo un'istanza di MyFrameListener
    private static final MyFrameListener listener = new MyFrameListener();

    public MyFrame() {
        super(titolo);
        //chiusura, dimensioni, layout, content pane ...
        // Associa listener al pulsante uno
        uno.addActionListener(listener);
        // due, tre, quattro
        // Associa listener al pulsante cinque
        cinque.addActionListener(listener);
        frmContentPane.add(uno);
        // due, tre, quattro
        frmContentPane.add(cinque);
        this.setVisible(true);
}  }
```

Implementazione dei Listener e struttura delle classi

Consideriamo la seguente finestra




```
public class EsempioTextField extends JFrame {

    private static final String titolo = "Esempio";
    private static final JPanel centro = new JPanel();
    private static final JPanel sud = new JPanel();
    private static final JTextField campoTesto = new JTextField(20);
    private static final JButton button = new JButton("Premi");
```

Implementazione dei Listener e struttura delle classi

```
    public EsempioTextField() {
        super(titolo);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        centro.add(campoTesto);
        sud.add(button);
        this.getContentPane().add(centro, BorderLayout.CENTER);
        this.getContentPane().add(sud, BorderLayout.SOUTH);
        // Associa un opportuno Listener al bottone
        button.addActionListener(new EsempioTextFieldListener());
        this.pack();
        this.setVisible(true);
    }
}
```

Implementazione dei Listener e struttura delle classi

Vogliamo che, dopo aver riempito la casella di testo, la pressione del pulsante produca una finestra di messaggio che mostri il contenuto della casella

- ma il metodo `getSource()` di `ActionEvent` permette di accedere solo all'oggetto che ha generato l'evento (in questo caso, il bottone...non il campo testuale!)

La realizzazione di Listener può seguire uno dei seguenti schemi implementativi principali:

1. listener come inner class (anche anonima)
2. listener implementato dalla classe che estende `JFrame` ✓
3. listener come classe esterna (vedi anche esempi precedenti) ✓
4. strutturazione dei package per l'interfaccia grafica ✓

Soluzione 1: listener come inner class

Definiamo l'ascoltatore (`EsempioTextFieldListener` nell'esempio) come inner class (classe interna) della classe che estende `JFrame` (`EsempioTextField` nell'esempio)

- la inner class ha accesso a tutti i campi (anche privati) della classe in cui è definita (in particolare al campo `JTextField campoTesto` che rappresenta la casella di testo d'interesse nell'esempio)

```
public class EsempioTextField extends JFrame {

    // Definizione di: centro, sud, campoTesto, button, titolo (come sopra)
    private static final JTextField campoTesto = new JTextField(20);

    public EsempioTextArea() {
        // Creazione e impostazione finestra (come sopra)
        button.addActionListener(new EsempioTextFieldListener());
        this.pack();
        this.setVisible(true);
    }
```

Soluzione 1: listener come inner class

```
    // Inner class
    private class EsempioTextFieldListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            // Accede direttamente al componente "campoTesto"
            JOptionPane.showMessageDialog(null, campoTesto.getText());
        }
    }
}
```

Soluzione 1: listener come inner class (2/2)

L'implementazione con inner class è riportata solo per completezza, poiché è uno stile di programmazione molto diffuso, ma questa soluzione è generalmente da evitare

- *pro*: consente l'accesso diretto ai componenti della GUI e lo stesso listener può essere associato a diversi componenti
- *contro*: accettabile per l'implementazione di listener molto semplici; non c'è chiara separazione tra codice per la realizzazione dell'interfaccia grafica e codice che gestisce gli eventi (soprattutto nel caso di inner class anonima)

Soluzione 2: la finestra implementa il listener

Possiamo garantire l'accesso diretto ai componenti della GUI e il riuso del listener con una realizzazione in cui il listener è implementato dalla classe che estende `JFrame`

- approccio utile per gestire finestre/componenti con poche e semplici funzionalità

```
public class EsempioTextField extends JFrame implements ActionListener {

    // Definizione di: centro, sud, campoTesto, button, titolo (come sopra)
    private static final JTextField campoTesto = new JTextField(20);

    public EsempioTextArea() {
        // Creazione e impostazione finestra (come sopra)

        // il listener e' implementato dalla classe stessa
        button.addActionListener(this);
        this.pack();
        this.setVisible(true);
    }
```

Soluzione 2: la finestra implementa il listener

```
    // Implementazione di actionPerformed per la gestione degli eventi
    public void actionPerformed(ActionEvent e) {
        // Accede direttamente al componente "campoTesto"
        JOptionPane.showMessageDialog(null, campoTesto.getText());
    }
}
```

Soluzione 3: listener come classe esterna

Possiamo implementare un listener come classe esterna che abbia un riferimento alla finestra contenente il componente che genera gli eventi

- in tal modo, il listener avrebbe accesso ai soli campi pubblici della finestra
  - nel nostro esempio, quindi, è necessario dotare la classe `EsempioTextField` di un metodo `JTextField getTextField()` che restituisca un riferimento alla casella di testo `campoTesto`
- questo approccio è utile quando uno stesso listener interagisce con i componenti di finestre diverse

Soluzione 3: listener come classe esterna

Implementazione del listener come classe esterna

```
public class EsempioTextFieldListener implements ActionListener {

    // riferimento alla finestra
    private EsempioTextField frame;

    EsempioTextFieldListener(EsempioTextField frame) {
        this.frame = frame;
    }

    public void actionPerformed(ActionEvent a){
        // Accede al componente "campoTesto" tramite
        // opportuno metodo
        JOptionPane.showMessageDialog
                  (null,frame.getTextField().getText());
    }
}
```

Soluzione 3: listener come classe esterna

```
public class EsempioTextField extends JFrame {

    // Definizione di: centro, sud, campoTesto,
    // button, titolo (come sopra)
    private static final JTextField campoTesto = new JTextField(20);

    public EsempioTextArea() {
        // Creazione e impostazione finestra (come sopra)

        // Passiamo il riferimento alla finestra (this) al costruttore
        // del listener
        button.addActionListener(new EsempioTextFieldListener(this));
        this.pack();
        this.setVisible(true);
    }

    // restituisce riferimento al campo testuale
    public JTextField getTextField() {
        return campoTesto;
    }
}
```

Soluzione 4: package per l'interfaccia grafica

Se non specifichiamo il livello di accesso di un campo dati o di un metodo, esso sarà accessibile (solo) a tutte le classi all'interno dello stesso package

- sfruttiamo questa caratteristica per garantire l'accesso ai componenti di un `JFrame` da parte dei suoi listener
- ovviamente i campi privati rimangono accessibili solo dall'interno della classe cui appartengono

È possibile adottare il seguente approccio implementativo:

- creiamo un package (es., `app.gui` che conterrà tutte le classi relative alla GUI
- aggiungiamo un sotto-package (es., `app.gui.myframe` contenente una classe che implementa la finestra e le classi che implementano i relativi ascoltatori
- se vi sono ascoltatori che interagiscono con più finestre, inseriamo tutto nello stesso sotto-package

Note:

- questa soluzione *riduce l'information hiding* ed *aumenta l'accoppiamento*
- questa soluzione è da preferirsi quando si ha a che fare con listener molto complessi e/o che interagiscono con più di una finestra

Limitare la proliferazione dei listener (1/2)

In generale, è possibile associare un listener ad ogni componente che genera eventi di interesse per le funzionalità dell'applicazione

- al crescere del numero di componenti e tipi di evento da gestire, crescerà il numero di listener implementati
- tipicamente il tempo di avvio di un'applicazione e l'occupazione di memoria sono direttamente proporzionali al numero di classi da caricare
  - all'aumentare del numero di classi aumentano il tempo di avvio e l'occupazione di memoria
  - è necessario bilanciare tra modularità e prestazioni dell'applicazione

Come limitare la proliferazione dei listener?

- possiamo creare listener che offrano più funzionalità, raggruppando funzionalità di gestione degli eventi tra loro omogenee in uno stesso listener

Limitare la proliferazione dei listener (2/2)

Associando uno stesso listener a più componenti è necessario poter riconoscere quale componente è di volta in volta sorgente dell'evento

- il metodo `setActionCommand(String c)` permette di associare un "comando" (una stringa) ad un componente
- quando il componente genera un evento, la stringa "comando" viene inserita nell'`ActionEvent` generato
- il metodo `String getActionCommand()` della classe `ActionEvent` può essere usato dal listener per ottenere la stringa "comando" associata all'evento ricevuto
- la gestione dell'evento da parte del listener può variare in base alla stringa "comando" letta dall'evento

L'uso della stringa "comando" è utile anche nel caso in cui componenti diversi generano eventi che devono essere gestiti nello stesso modo (es, voce di menu "salva" e icona "salva" in una barra degli strumenti)

- la stessa stringa "comando" verrà associata ai diversi componenti

Limitare la proliferazione dei listener: esempio (1/2)

Esempio con classe esterna per il listener (facilmente applicabile agli altri casi)

```
public class Listener implements ActionListener {
    // Costanti usate quando la stringa "comando" viene assegnata
    public final String UPOPT = "up";
    public final String DOWNOPT = "down";
    public final String RANDOMOPT = "random";
    public void actionPerformed(ActionEvent e) {
        // legge il comando dall'evento
        String com = e.getActionCommand();
        // L'evento viene gestito in base al comando
        // (se non e' una stringa valida, nessuna azione viene eseguita)
        if (com == UPOPT)
            upOpt();
        else if (src == DOWNOPT)
            downOpt();
        else if (src == RANDOMOPT)
            randomOpt();
    }
    // Metodi specifici invocati in base alla stringa "comando" ricevuta
    private void upOpt() { ... }
    private void randomOpt() { ... }
    private void downOpt() { ... }
}
```

Limitare la proliferazione dei listener: esempio (2/2)

```
public class MyFrame extends JFrame {
    // ...
    JMenuItem upOpt = new JMenuItem("Up");
    JMenuItem downOpt = new JMenuItem("Down");
    JMenuItem randomOpt = new JMenuItem("Random");
    Listener ascoltatore = new Listener();
    public MyFrame() {
        // costruzione frame e menu ...
        // lo stesso listener viene associato alle diverse voci di menu
        upOpt.addActionListener(ascoltatore);
        // upOpt scrive nell'ActionEvent generato la stringa Listener.UPOPT
        upOpt.setActionCommand(Listener.UPOPT);
        downOpt.addActionListener(ascoltatore);
        // downOpt scrive nell'ActionEvent
        // generato la stringa Listener.DOWNOPT
        downOpt.setActionCommand(Listener.DOWNOPT);
        randomOpt.addActionListener(ascoltatore);
        // randomOpt scrive nell'ActionEvent
        // generato la stringa Listener.RANDOMOPT
        randomOpt.setActionCommand(Listener.RANDOMOPT);
        // ...
    }
}
```

Contenuti

Riferimenti utili

- Java Tutorial – Creating a GUI with Swing
  http://docs.oracle.com/javase/tutorial/uiswing/index.html
- Java API specification
  http://docs.oracle.com/javase/7/docs/api/
- Java Swing tutorial
  http://zetcode.com/tutorials/javaswingtutorial/
- A Swing Architecture Overview
  http://www.oracle.com/technetwork/java/architecture-142923.html
- Sezione "References" in
  http://en.wikipedia.org/wiki/Swing_(Java)
- Slide su "Threads e Concorrenza in Java Swing" (illustrate più avanti nel corso)