



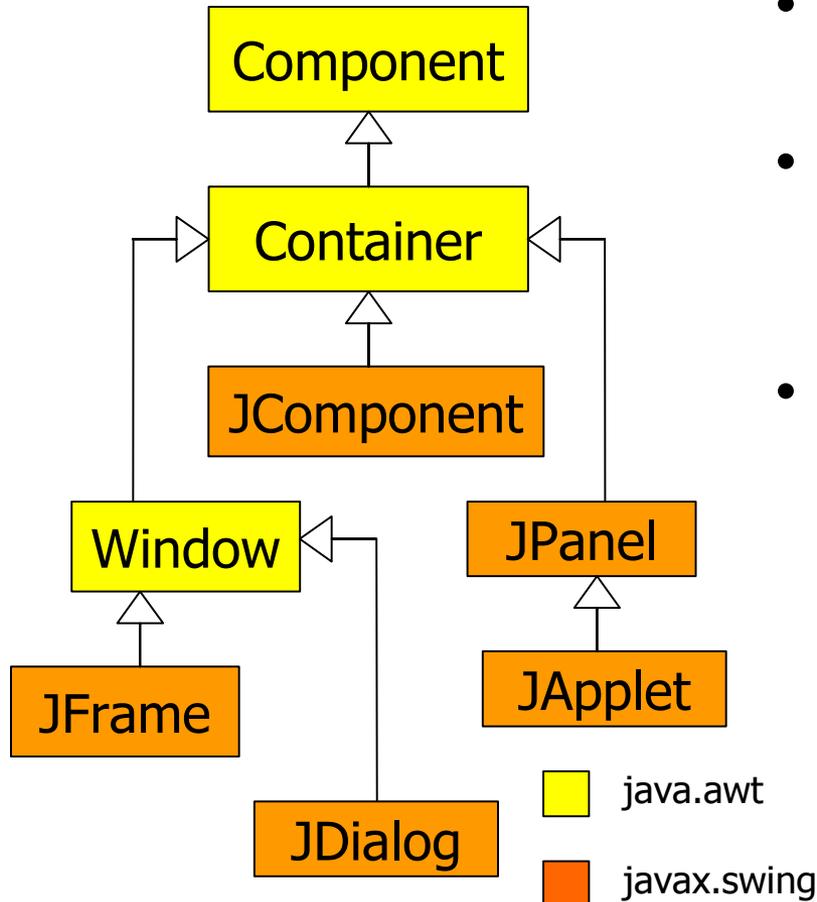
# **Sviluppo di Interfacce Grafiche in Java**

Università di Roma “La Sapienza”

# Dalle AWT alle Swing

- Nel 1996 Sun introduce nel Java il primo package grafico: le **AWT** (Abstract Windows Toolkit)
  - Le classi che mappano i componenti grafici sono scritti in codice nativo e si appoggiavano alle chiamate del sistema operativo
  - Difetti:
    - Occorre limitarsi a considerare *controlli* (tecnicamente *widget*) che sono il “minimo comun denominatore” di tutti i sistemi operativi
    - Appoggiandosi al sistema operativo, l’aspetto grafico può variare sensibilmente in funzione della piattaforma su cui gira la JVM
    - Conclusione: poca portabilità
- Per correggere questi difetti la Sun ha successivamente introdotto le **Swing** in cui i componenti vengono direttamente disegnati dalla JVM
  - Visualizzazione uniforme su ogni piattaforma, ricchezza di componenti

# Package



- Gli oggetti grafici Swing derivano da quelli AWT
- I 70 e più controlli Swing derivano per lo più da **JComponent**
- JComponent eredita da **Container**, una sorta di controllo contenitore che offre la possibilità di disporre altri componenti all'interno mediante il metodo:
  - `void add(Component)`

# I *top-level* Container

- I *top-level* Container sono i componenti all'interno dei quali si creano le interfacce
  - Noi useremo **JFrame** ma esistono anche JDialog, JApplet, ecc.
- Si può creare un oggetto JFrame usando il costruttore:
  - `JFrame([String titoloFinestra])`
- JFrame ha una struttura a strati piuttosto complessa. Noi useremo solo il “**pannello** di contenimento”. I componenti, infatti, non si aggiungono direttamente al JFrame ma a tale pannello. Il riferimento al “pannello di contenimento” di un JFrame si ottiene attraverso il metodo:
  - `Container getContentPane()`

Come era da aspettarsi il `ContentPane` è una istanza di **Container**

# Esempio



Aggiunge una etichetta di testo al ContentPane

Imposta la dim. della finestra (di default 0 x 0)

Imposta il comportamento alla chiusura della finestra

Rende visibile la finestra (di default è nascosta)

```
import javax.swing.*;  
import java.awt.*;  
  
public class Application  
{  
    public static void main(String args[])  
    {  
        JFrame win;  
        win = new JFrame("Prima finestra");  
        Container c = win.getContentPane();  
        c.add(new JLabel("Buona Lezione"));  
        win.setSize(200,200);  
        win.setDefaultCloseOperation  
            (JFrame.EXIT_ON_CLOSE);  
        win.setVisible(true);  
    }  
}
```

Inizialmente le swing erano **eXtension** delle librerie di Java: per questo si importa **javax.swing.\*** (per *backward-compatibility*)

**EXIT\_ON\_CLOSE** indica che il processo termina alla chiusura della finestra (di default la finestra non viene distrutta ma semplicemente nascosta e il processo non viene terminato)

# Estendere JFrame

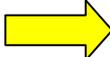
```
import javax.swing.*;  
import java.awt.*;
```

```
class MyFrame extends JFrame  
{  
    JLabel jl = new JLabel("Buona  
        Lezione");  
    MyFrame() {  
        super("Prima finestra");  
        Container c = this.getContentPane();  
        c.add(jl);  
        this.setSize(200,200);  
        this.setDefaultCloseOperation  
            (JFrame.EXIT_ON_CLOSE);  
        this.setVisible(true);  
    }  
}  
public class Application  
{  
    public ... main(String args[])  
    {  
        ... = new MyFrame();  
    }  
}
```

- La soluzione usata nell'esempio precedente non è preferibile perché non sfrutta le potenzialità dell'OO:
  - Non permette l'information hiding dei controlli.
  - Non sfrutta l'incapsulamento mettendo insieme concetti eterogenei e finestre diverse tra loro
    - Nel caso limite il *main* contiene le definizioni di tutte le finestre del programma

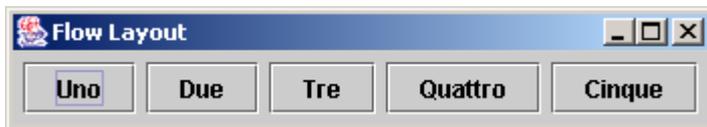
Meglio una classe per ogni finestra

# I Layout Manager

- In molti Container i controlli sono inseriti da sinistra verso destra come su una riga ideale  può non essere sempre la politica della GUI desiderata
  - Per superare questo problema è nato il concetto di Gestore di Layout
- Un **Layout Manager** è una politica di posizionamento dei componenti in un *Container*. Ogni Container ha un *Layout Manager*
  - Un gestore di layout è una qualsiasi classe che implementa *LayoutManager* (**interfaccia** definita nel package **java.awt**)
  - Ogni qualvolta occorre ridimensionare o dimensionare la prima volta un Container viene richiamato il Gestore di Layout.
- Per impostare un certo Layout per un Container si usa il metodo
  - `void setLayout(LayoutManager)`

# FlowLayout

```
public class MyFrame extends JFrame {
    JButton uno=new JButton("Uno");
    ...
    JButton cinque = new JButton("Cinque");
    public MyFrame() {
        super("Flow Layout");
        Container c = this.getContentPane();
        c.setLayout(new FlowLayout());
        c.add(uno);
        ...
        c.add(cinque);
        setSize(300,100);
        setVisible(true);
    }
}
```



- I componenti sono inseriti in ipotetiche “righe” da sinistra verso destra.
  - Quando un componente non entra in una riga viene posizionato nella riga successiva
- Costruttore:  
FlowLayout([int allin])

Specifica l'allineamento dei controlli su una riga:  
**FlowLayout.LEFT**  
**FlowLayout.CENTER** (Default)  
**FlowLayout.RIGHT**

# FlowLayout

```
public class MyFrame extends JFrame {
    JButton uno=new JButton("Uno");
    ...
    JButton cinque = new JButton("Cinque");
    public MyFrame() {
        super("Flow Layout");
        Container c = this.getContentPane();
        c.setLayout(new FlowLayout());
        c.add(uno);
        ...
        c.add(cinque);
        setSize(300,100);
        setVisible(true);
    }
}
```



- I componenti sono inseriti in ipotetiche “righe” da sinistra verso destra.
  - Quando un componente non entra in una riga viene posizionato nella riga successiva
- Costruttore:  
`FlowLayout([int allin])`

Specifica l'allineamento dei controlli su una riga:  
**FlowLayout.LEFT**  
**FlowLayout.CENTER** (Default)  
**FlowLayout.RIGHT**

# GridLayout

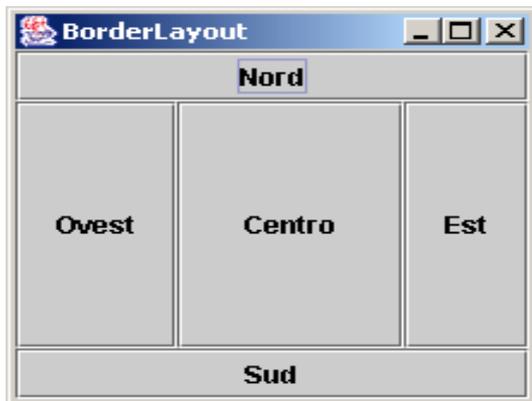
```
public class MyFrame extends JFrame
{
    public MyFrame() {
        super("Grid Layout");
        Container c = this.getContentPane();
        c.setLayout(new GridLayout(4,4));
        for(int i = 0; i<15; i++)
            c.add(new JButton(String.valueOf(i)));
        setSize(300,300);
        setVisible(true);
    }
}
```



- Il Container viene suddiviso in una griglia di celle di uguali dimensioni
  - Diversamente dal FlowLayout, i componenti all'interno della griglia assumono tutti la stessa dimensione
- La dimensione della griglia viene impostata dal costruttore:
  - `GridLayout(int r, int c)`

# BorderLayout

```
public class MyFrame extends JFrame {
    JButton nord = new JButton("Nord");
    ...
    public MyFrame() {
        super("Border Layout");
        Container c = this.getContentPane();
        c.setLayout(new BorderLayout());
        c.add(nord, BorderLayout.NORTH);
        c.add(centro, BorderLayout.CENTER);
        c.add(est, BorderLayout.EAST);
        c.add(ovest, BorderLayout.WEST);
        c.add(sud, BorderLayout.SOUTH);
        setSize(300, 300); setVisible(true);
    }
}
```



- Il Container è suddiviso in cinque aree a croce
  - Se un'area non viene riempita, le altre si estendono fino a riempirla
  - Ogni area può contenere un solo componente
    - Un secondo componente sostituisce il primo
- L'aggiunta dei componenti è realizzata con il metodo:
  - `void add(Component, String)`

Specifica la posizione in cui si intende aggiungere:  
**BorderLayout.CENTER, BorderLayout.NORTH,**  
**BorderLayout.SUD, BorderLayout.EAST,**  
**BorderLayout.WEST**

# BorderLayout

```
public class MyFrame extends JFrame {
    JButton nord = new JButton("Nord");
    ...
    public MyFrame() {
        super("Border Layout");
        Container c = this.getContentPane();
        c.setLayout(new BorderLayout());
        c.add(nord, BorderLayout.NORTH);
        c.add(centro, BorderLayout.CENTER);
        c.add(est, BorderLayout.EAST);
        c.add(ovest, BorderLayout.WEST);
        c.add(sud, BorderLayout.SOUTH);
        setSize(300, 300); setVisible(true);
    }
}
```



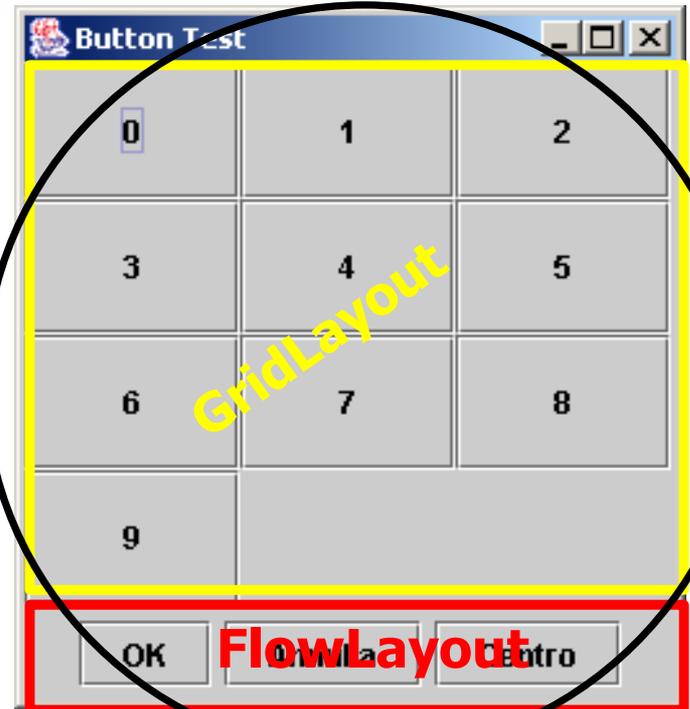
- Il Container è suddiviso in cinque aree a croce
  - Se un'area non viene riempita, le altre si estendono fino a riempirla
  - Ogni area può contenere un solo componente
    - Un secondo componente sostituisce il primo
- L'aggiunta dei componenti è realizzata con il metodo:
  - `void add(Component, String)`

Specifica la posizione in cui si intende aggiungere:  
**BorderLayout.CENTER, BorderLayout.NORTH,**  
**BorderLayout.SUD, BorderLayout.EAST,**  
**BorderLayout.WEST**

# Gerarchia di Contenimento

## BorderLayout

- La maggior parte delle finestre non possono essere realizzate con un solo Layout (e Container)
  - In casi come quello in figura occorre predisporre più Container, uno per ogni zona che ha un layout differente
  - La lastra dei contenuti è il Container principale ed altri Container si possono aggiungere come **JPanel** da usare come contenitori per componenti.
  - I JPanel, essendo *JComponent*, possono essere aggiunti a *ContentPane*



La forza di questa soluzione è data dall'alta modularità: è possibile usare un layout per il pannello interno e un altro layout per il *ContentPane*. Il pannello interno verrà inserito nella finestra coerentemente con il layout della lastra dei contenuti. Inoltre nei pannelli interni se ne possono inserire altri con loro layout e così via, come nel gioco delle scatole cinesi. Il numero di soluzioni diverse sono praticamente infinite.

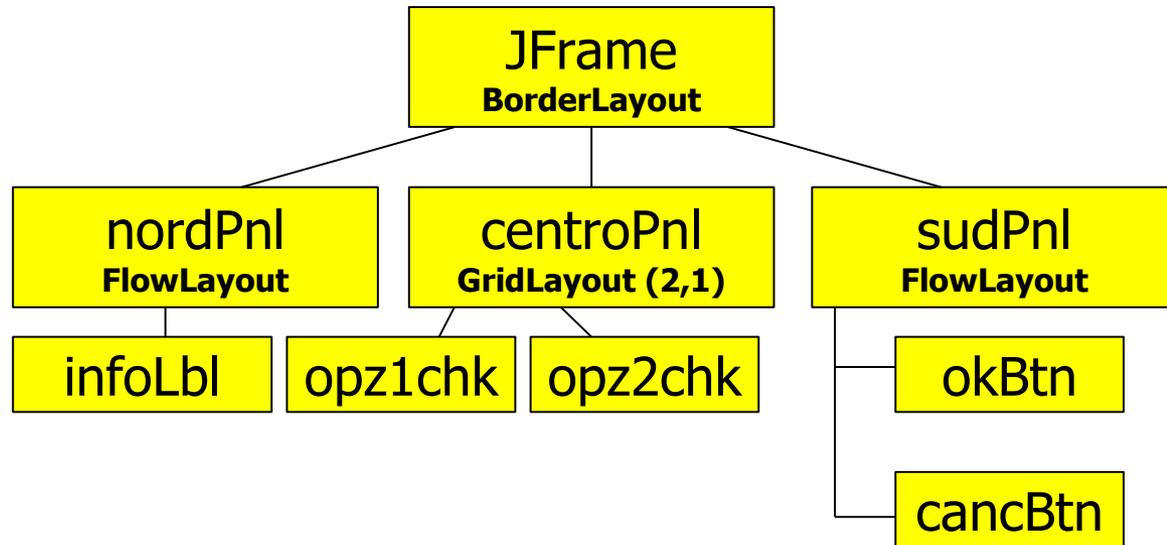
# Progettazione della GUI

- Approccio Top-Down: si parte dai componenti più esterni procedendo via via verso quelli più interni
  1. Si assegna un Layout Manager al *JFrame* per suddividere l'area in zone più piccole
  2. Per ogni area a disposizione si crea un *JPanel*
    - Ogni pannello può ricorrere ad un Layout differente
  3. Ogni pannello può essere sviluppato creando all'interno altri pannelli e/o inserendo direttamente i controlli

Il risultato della progettazione può essere rappresentato con un albero della GUI

- Ogni componente è rappresentato da un nodo i cui figli sono i componenti contenuti all'interno e il padre è il componente che lo contiene
- Realizzazione Bottom-Up: si realizzano prima i componenti atomici, risalendo attraverso i Container fino al “pannello dei contenuti” del *JFrame*

# Esempio/1



Come default, il “Pannello dei Contenuti” di un JFrame ha un BorderLayout, mentre un JPanel ha un FlowLayout con allineamento centrale

➔ Occorre modificare sono il  
LayoutManager del *centroPnl*

# Esempio/2

```
public class MyFrame extends JFrame
{
    JPanel nordPnl = new JPanel();
    JPanel centroPnl = new JPanel();
    JPanel sudPnl = new JPanel();
    JLabel infoLbl = new JLabel("Selezionare:");
    JCheckBox opz1Chk = new JCheckBox("Opz1");
    JCheckBox opz2Chk = new JCheckBox("Opz2");
    JButton okBtn=new JButton("OK");
    JButton cancBtn=new JButton("Annulla");
```

# Esempio/3

```
public MyFrame() {
    super("Esempio");
    centroPnl.setLayout(new GridLayout(2,1));
    centroPnl.add(opz1Chk);
    centroPnl.add(opz2Chk);
    nordPnl.add(infoLbl);
    sudPnl.add(okBtn);
    sudPnl.add(cancBtn);
    getContentPane().add(nordPnl, BorderLayout.NORTH);
    getContentPane().add(centroPnl, BorderLayout.CENTER);
    getContentPane().add(sudPnl, BorderLayout.SOUTH);
    pack();
    Dimension dim =
        Toolkit.getDefaultToolkit().getScreenSize();
    setLocation((dim.getWidth()-this.getWidth())/2,
        (dim.getHeight()-this.getHeight())/2);
    setVisible(true);
}
```

La classe `java.awt.Dimension` possiede due proprietà **Width** e **Height**. Nel caso specifico conterranno la dimensione dello schermo in pixel

La dimensione è impostata come la minima necessaria a visualizzare tutti i controlli

Sposta la finestra al centro dello schermo con il metodo `setLocation(int x, int y)` dove `x, y` sono le coordinate dell'angolo in alto a sinistra

# Differenze di layout / 1

Tendenzialmente, **GridLayout** estende i widget su tutta l'area disponibile

```
public class SwingFrame extends JFrame
{
    public SwingFrame()
    {
        super();
        Container c = this.getContentPane();

        c.setLayout(new GridLayout(1, 2));

        c.add(new JButton("OK"));
        c.add(new JButton("Annulla"));

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

Imposto il layout

Aggiungo i pulsanti

Risultato...  
"OK" e "Annulla" delle stesse dimensioni,  
"splittati" sul campo



# Differenze di layout / 2

Se associamo ad ogni *widget* un proprio *pannello*, non vengono alterate le sue dimensioni per adattarle al *Container* più esterno!

```
public class BetterSwingFrame extends JFrame {  
    public BetterSwingFrame() {  
        super();  
        Container c = this.getContentPane();  
  
        c.setLayout(new GridLayout(1, 2));  
  
        JPanel pOk = new JPanel();  
        JPanel pUndo = new JPanel();  
        pOk.add(new JButton("OK"));  
        pUndo.add(new JButton("Annulla"));  
        c.add(pOk);  
        c.add(pUndo);  
  
        this.pack();  
        ...  
    }  
}
```

Imposto il layout

Creo dei pannelli appositi

Aggiungo i pulsanti ai pannelli

Risultato...  
"OK" e "Annulla" non più delle stesse  
dimensioni, hanno il loro spazio!



# Caselle di Controllo

- Una checkbox (o casella di controllo) è un controllo grafico che alterna due stati: selezionato/non selezionato
  - In Java una checkbox è un oggetto della classe `JCheckBox`
  - Si costruisce fornendo il nome nel costruttore ed eventualmente lo stato iniziale: `JCheckBox(String nome, [boolean stato])`
  - È possibile conoscerne lo stato invocando il metodo della classe: `boolean isSelected()` che restituisce *true* se la casella di controllo è selezionata, *false* altrimenti
- È utilizzato in contesti in cui si vuole lasciare l'utente nella possibilità di scegliere una o più opzioni in un insieme (eventualmente formato da un solo elemento)
  - Le caselle di controllo quando selezionate contengono tipicamente un segno di spunta

# I Menu/1

- Permettono di accedere ad un gran numero di azioni in uno spazio ridotto, organizzato gerarchicamente
- Un menu è caratterizzato da 3 elementi:
  1. Una sola barra dei menu che contiene gli elementi dei menu
    - In Java questo componente è un oggetto della classe *JMenuBar*
    - Una *JMenuBar* si crea a partire dal costruttore senza parametri
    - Una *JMenuBar* per una finestra si imposta attraverso il metodo della classe *JFrame*: `void setJMenuBar(JMenuBar)`
  2. Uno o più menu. Ogni menu può a sua volta contenere un sottomenu
    - In Java questo componente è un oggetto della classe *JMenu*
    - Un *JMenu* si può creare a partire dal costruttore:  
`JMenu(String nomeMenu)`
    - È possibile aggiungerne uno alla *MenuBar* con il metodo della classe *JMenuBar*:  
`void add(JMenuBar)`
    - È possibile aggiungere un sottomenu ad un menu con il metodo della classe *JMenu*: `void add(JMenu)`

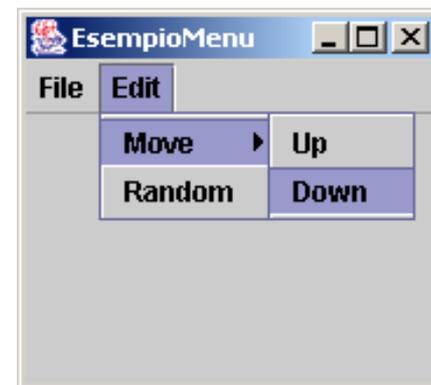
# I Menu/2

```
public class MyFrame extends JFrame {
    JMenuBar bar = new JMenuBar();
    JMenu fileMnu = new JMenu("File");
    JMenu editMnu = new JMenu("Edit");
    JMenu moveMnu = new JMenu("Move");

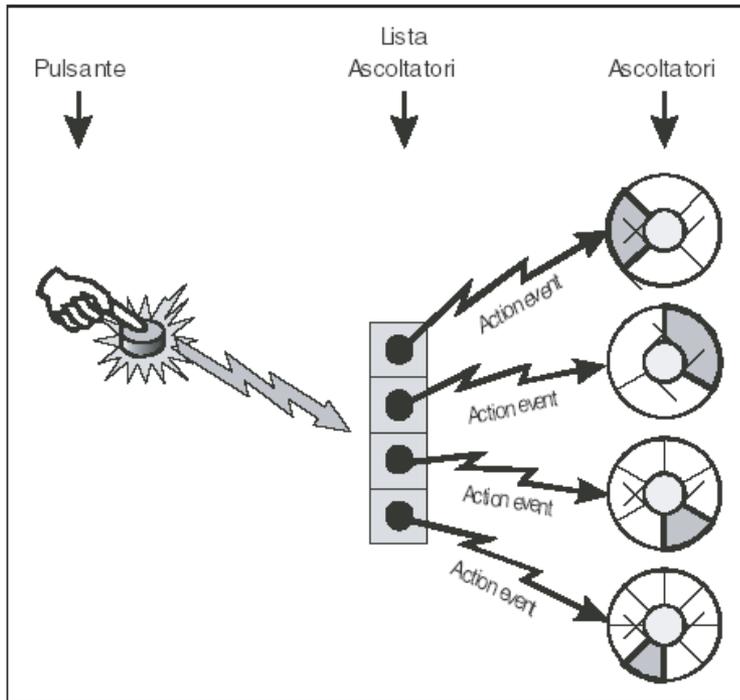
    public MyFrame() {
        ...
        setJMenuBar(bar);
        bar.add(fileMnu);
        bar.add(editMnu);
        editMnu.add(moveMnu);
        editMnu.add(new JMenuItem("Random"));
        moveMnu.add(new JMenuItem("Up"));
        moveMnu.add(new JMenuItem("Down"));
        ...
    }
}
```

## 1. Un certo numero di voci contenute nei Menu

- In Java ogni voce del menu è una istanza della classe *JMenuItem*
- Un *JMenuItem* si può creare a partire dal costruttore:  
`JMenuItem(String nomeVoce)`
- È possibile aggiungere una voce ad un menu con il metodo della classe *JMenu*:  
`void add(JMenuItem)`



# Gestione degli Eventi/1



- In Java ogni componente è predisposto ad essere sollecitato dall'utente
  - Le interazioni generano *eventi* su un controllo il quale si limita a *delegare* a un *ascoltatore* l'azione da svolgere (*Event Delegation*)
    - Per esempio un pulsante *non sa* cosa avverrà alla sua pressione: si limita a *notificare* ai propri ascoltatori che l'evento che attendevano è avvenuto. Saranno questi a provvedere a produrre un effetto
  - Ogni componente può avere più ascoltatori per un determinato evento o per eventi differenti
- 
- È possibile *installare* uno stesso ascoltatore su più componenti anche diversi
    - Questo ovviamente se entrambi i componenti possono gestire l'evento
    - Questo operazione è frequente: si pensi alle voci di un menu che vengono replicate su una toolbar per un più facile accesso ad alcune funzionalità frequenti

# Gestione degli Eventi/2

Implementare un ascoltatore coinvolge 3 classi:

1. *La classe dell'ascoltatore* che implementa una particolare interfaccia del tipo `XXXListener` tipica degli eventi di una certa classe
  - I metodi dell'interfaccia che la classe dell'ascoltatore implementa contengono il codice eseguito allo scatenarsi degli eventi di una classe che l'ascoltatore intercetta
2. *La classe evento* che contiene le informazioni riguardanti le caratteristiche dell'evento generato.
  - Gli oggetti di questa classe sono istanziati direttamente dai componenti che notificano eventi agli ascoltatori
  - Formalmente sono parametri di input dei metodi dell'interfaccia implementata dall'ascoltatore
  - Possono essere utilizzati per modificare il comportamento dell'ascoltatore in base alle informazioni sull'evento scatenato
3. *L'origine dell'evento* cioè il componente che scatena l'evento per cui si vuole "installare" l'ascoltatore
  - È possibile installare un ascoltatore per un componente col metodo:  
`addXXXListener (XXXListener)`

# Esempio: MouseListener / 1

L'interfaccia `MouseListener` si occupa di intercettare gli eventi associati al mouse

```
public interface MouseListener
{
    void mouseClicked(MouseEvent e);
    void mouseEntered(MouseEvent e);
    void mouseExited(MouseEvent e);
    void mousePressed(MouseEvent e);
    void mouseReleased(MouseEvent e);
}
```

Metodo invocato quando si clicca sul componente dove il listener è installato

Metodo invocato quando il mouse entra sul comp.

Metodo invocato quando si preme il mouse

Metodo invocato quando si rilascia il mouse

- I metodi `int getX()` e `int getY()` di `MouseEvent` permettono di ottenere le coordinate del mouse allo scatenarsi dell'evento
- Il metodo `int getModifiers()` permette di determinare quale bottone è stato premuto

# Esempio: MouseListener /2

```
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) {
        ...out.println("Premuto su ("+e.getX()+", "+e.getY()+")");
    }
    public void mouseReleased(MouseEvent e) {
        ...out.println("Rilasciato su ("+e.getX()+", "+e.getY()+")")
    }
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}

public class MyFrame extends JFrame
{
    public MyFrame()
    {
        super("MouseTest");
        this.addMouseListener(new MouseSpy());
        setSize(200,200);
        setVisible(true);
    }
}
```

Il package `java.awt.event` si occupa degli eventi

Quando il mouse entra o nella finestra viene eseguito questo metodo vuoto. Quindi è come se l'evento non venisse gestito

# Ascoltatori più Importanti

Interfaccia	Descrizione
ActionListener	Definisce 1 metodo per ricevere eventi-azione
ComponentListener	Definisce 4 metodi per riconoscere quando un componente viene nascosto, spostato, mostrato o ridimensionato
FocusListener	Definisce 2 metodi per riconoscere quando un componente ottiene o perde il focus
KeyListener	Definisce 3 metodi per riconoscere quando viene premuto, rilasciato o battuto un tasto
MouseMotionListener	Definisce 2 metodi per riconoscere quando il mouse è trascinato o spostato
MouseListener	Definisce 5 metodi (se ne già parlato)
TextListener	Definisce 1 metodo per riconoscere quando cambia il valore di un campo testo
WindowListener	Definisce 7 metodi per riconoscere quando un finestra viene attivata, chiusa, disattivata, ripristinata, ridotta a icona, ecc.

# Gli Eventi Azione

- Tutti i componenti Swing generano eventi che possono essere catturati da un *actionListener*
  - I seguenti eventi ad esempio notificano un evento azione:
    - Quando si preme **INVIO** in un campo di testo
    - Quando si seleziona una checkbox, un radioButton o una voce in un ComboBox
    - Quando si seleziona una voce di un menu o si clicca su un bottone
  - È possibile installare uno stesso ascoltatore, ad esempio, per un bottone della toolbar e una voce di menu
    - Il vantaggio risiede nel fatto che sia la voce del menu che il bottone hanno lo stesso ascoltatore (invece che due ascoltatori separati che fanno la stessa cosa)
- L'interfaccia di un *actionListener* è definita:

```
public interface ActionListener
{
    public void actionPerformed(ActionEvent ae);
}
```

# Esempio

```
public class MyFrame extends JFrame {
    JButton uno = new JButton("Uno");
    ...
    JButton cinque = new JButton("Cinque");
    Ascoltatore listener = new Ascoltatore();
    public MyFrame() {
        ...
        Container c = this.getContentPane();
        c.add(uno);
        uno.addActionListener(listener);
        ...
        c.add(cinque);
        cinque.addActionListener(listener);
    }
}
```

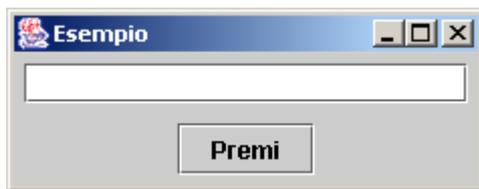
```
public class Ascoltatore implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        JButton b = (JButton)event.getSource();
        JOptionPane.showMessageDialog(null,
            "È stato premuto"+b.getText());
    }
}
```



Il metodo `Object.getSource()` restituisce il riferimento all'oggetto che ha generato l'evento

# Esempio

- Scrivere un listener che alla pressione del bottone mostri il contenuto della JTextField
- L'ascoltatore deve aver accesso al riferimento a quel JTextField di quella particolare istanza di *MyFrame*
  - Tuttavia la classe `ActionEvent` permette di accedere solo all'oggetto che ha scatenato l'evento



**Errore! Questo è un altro JTextField!**

```
public class MyFrame extends JFrame {
    JPanel centro = new JPanel();
    JPanel sud = new JPanel();
    JTextField txt = new JTextField(20);
    JButton button = new JButton("Premi");
    public MyFrame() {
        super("Esempio");
        centro.add(txt);
        sud.add(button);
        getContentPane().add
            (centro, BorderLayout.CENTER);
        getContentPane().add
            (sud, BorderLayout.SOUTH);
        button.addActionListener(new
            Listen());
        ...
    }
}

class Listen implements ActionListener
{
    void actionPerformed(ActionEvent e)
    {
        JTextField text = new JTextField();

        JOptionPane.showMessageDialog(
            null, text.getText());
    }
}
```

# Soluzione/1

```
public class MyFrame extends JFrame {
    JPanel centro = new JPanel();
    JPanel sud = new JPanel();
    JTextField txt = new JTextField(20);
    JButton button = new JButton("Premi");
    public MyFrame() {
        super("Esempio");
        centro.add(txt);
        sud.add(button);
        getContentPane().add
            (centro, BorderLayout.CENTER);
        getContentPane().add
            (sud, BorderLayout.SOUTH);
        button.addActionListener(new
            Listen());
        ...
    }

    class Listen implements ActionListener
    { void actionPerformed(ActionEvent e) {
        JTextField text = txt;
        JOptionPane.showMessageDialog(
            null, text.getText());
        }
    }
}
```

Unica classe esterna

- Si può utilizzare una classe interna come *ActionListener*
  - Una classe interna è definita all'interno di un'altra classe
  - I metodi della classe interna possono accedere alle variabili d'istanza della classe esterna (eventualmente anche a quelli privati!)
  - Questa tecnica è accettabile se l'ascoltatore "fa poco" altrimenti la classe cresce sensibilmente in dimensioni
    - Si accorpano due classi che rappresentano due aspetti concettualmente diversi

# Soluzione/2

```
public class MyFrame extends JFrame {
    JPanel centro = new JPanel();
    JPanel sud = new JPanel();
    JTextField txt = new JTextField(20);
    JButton button = new JButton("Premi");
    public MyFrame() {
        ...
        sud.add(button);
        button.addActionListener(new
            Listen(this));
        ...
    }
}

class Listen implements ActionListener
{
    MyFrame finestra;

    public Listen(MyFrame frame) {
        finestra=frame;
    }
    void actionPerformed(ActionEvent e)
    {
        JTextField text = finestra.txt;
        JOptionPane.showMessageDialog(
            null, text.getText());
    }
}
```

- Si può progettare l'ascoltatore prevedendo un costruttore che prende in ingresso il riferimento alla finestra contenente il bottone
  - Il riferimento alla finestra, parametro del costruttore, viene memorizzato con una variabile d'istanza
  - La classe comunque resta esterna ma può accedere a tutti gli oggetti della finestra (a patto che questi non siano privati)

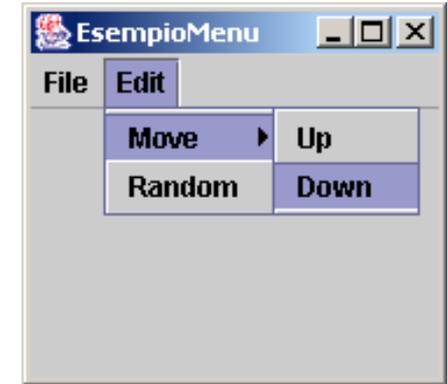
# Un Listener per N Controlli/1

- Nella pratica è improponibile prevedere una classe *actionListener* per ogni bottone, voce del menu, ecc...
  - Un'applicazione, ad esempio, con 5 finestre, ognuna con 5 bottoni ed una anche con 3 menu di 4 opzioni ciascuno, avrebbe 37 classi solo per gestire la pressione dei bottoni o della voci del menu!
- Conviene prevedere, per quanto sia possibile, pochi ascoltatori *actionListener*, ognuno di essi installato su molti bottoni del JFrame o su molte voci del menu
  - I componenti condividono lo stesso metodo `actionPerformed` che, quindi, deve essere in grado di “capire” quale controllo ha generato l'evento per poter scegliere che “comportamento” adottare
  - “Capire chi” ha generato l'evento può essere fatto in due modi:
    1. Utilizzando il metodo `getSource` e le classi interne
    2. Utilizzando la proprietà stringa `actionCommand`, implementata per ogni componente, che permette di associare una stringa identificativa ad ogni componente che scatena un evento azione

# Un Listener per N Controlli/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() {
        ...
        UpOpt.addActionListener(ascoltatore);
        DownOpt.addActionListener(ascoltatore);
        RandomOpt.addActionListener(ascoltatore);
        ... }
    class Listener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            Object src = e.getSource();
            if (src == UpOpt)
                { codice della voce del menu Up }

            else if (src == DownOpt)
                { codice della voce del menu Down }
            else if (src == RandomOpt)
                { codice della voce del menu Random }
        }
    }
}
```



Se **src==UpOpt** significa che è il componente che ha scatenato l'evento è la voce del menu **Up**

Se **src==RandomOpt** significa che è il componente che ha scatenato l'evento è la voce del menu **Random**

# Un Listener per N Controlli/3

```
public class MyFrame extends JFrame {
    ...
    JMenuItem UpOpt = new JMenuItem("Up");
    JMenuItem DownOpt = new JMenuItem("Down");
    JMenuItem RandomOpt = new JMenuItem("Random");
    Listener ascolt = new Listener();
    public MyFrame() {
        ...
        UpOpt.addActionListener(ascolt);
        UpOpt.setActionCommand(ascolt.UPOPT);
        DownOpt.addActionListener(ascolt);
        DownOpt.setActionCommand(ascolt.DOWNOPT);
        RandomOpt.addActionListener(ascolt);
    }
    RandomOpt.setActionListener(ascolt.RANDOMOPT)
    ...
}
```

- Quando si usano classi esterne, è possibile “capire” il componente che ha notificato l’evento associando ai diversi componenti un diverso valore della proprietà *actionCommand*
  - Nell’esempio i possibili diversi valori sono memorizzati come costanti della classe ascoltatore `Listener`

- Il metodo `actionPerformed` dell’*actionListener* legge il valore della proprietà *actionCommand* del componente che ha notificato l’evento e, in funzione del valore letto, sceglie la porzione di codice da eseguire
  - Associando lo stesso valore all’*actionCommand* di due componenti gestiti dallo stesso ascoltatore, questi gestiranno l’evento azione nello stesso modo

# Un Listener per N Controlli/4

```
public class Listener implements ActionListener
{
    public final static String UPOPT = "up";
    public final static String DOWNOPT = "down";
    public final static String
        RANDOMOPT = "random";

    public void actionPerformed(ActionEvent e)
    {
        String com = e.getActionCommand();
        if (com == UPOPT)
            upOpt();
        else if (src == DOWNOPT)
            downOpt();
        else if (src == RANDOMOPT)
            randomOpt();
    }

    private void upOpt()
    { ... }
    private void randomOpt()
    { ... }
    private void downOpt()
    { ... }
}
```

Le costanti  
*actionCommand*

Il metodo comune che legge il valore dell'*actionCommand* del componente notificatore e, in funzione del valore letto, esegue un metodo diverso specifico del componente

I metodi privati specifici di ogni componente

# Per saperne di più...

Le **Swing** sono un ottimo motivo per cominciare ad usare a fondo la **documentazione** e i **tutorial** di Java forniti dalla **Sun**!

Esistono svariati **costruttori** per JFrame, JPanel, JButton... con *side-effect* sul widget o sulla finestra differenti, con possibilità di *customizzazione* sempre crescente!

Inoltre, esistono anche molti altri widget:

- **JTextArea**: aree di inserimento testo
- **JFileChooser**: tipica finestra di selezione file (“Apri”, “Salva”...)
- **JSlider**: cursori di selezione
- ...

come molti altri **Layout Manager** e **Pannelli** (es. **JTabbedPane**).

Trattare il tutto in una qualunque dispensa o libro sarebbe lungo, macchinoso e... non terrebbe conto della continua evoluzione del campo! Dunque? Buona lettura!!

<http://java.sun.com/docs/books/tutorial/uiswing/TOC.html#components>