SAPIENZA – Università di Roma
Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Dispensa didattica

Sviluppo di Interfacce Grafiche in Java. Concetti di Base ed Esempi.

M. de Leoni, M. Mecella, S. Saltarelli

Creative Commons License Deed

Attribuzione - Non commerciale - Non opere derivate 2.5 Italia

Tu sei libero:

- di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare questa opera

Alle seguenti condizioni:

Attribuzione. Devi attribuire la paternità dell'opera nei modi indicati dall'autore o da chi ti ha dato l'opera in licenza.

Non commerciale. Non puoi usare questa opera per fini commerciali.

Non opere derivate. Non puoi alterare o trasformare quest'opera, nè usarla per crearne un'altra.

Ogni volta che usi o distribuisci questa opera, devi farlo secondo i termini di questa licenza, che va comunicata con chiarezza.
In ogni caso, puoi concordare col titolare dei diritti d'autore utilizzi di quest'opera non consentiti da questa licenza.
Niente in questa licenza indebolisce o restringe i diritti degli autori.

Le utilizzazioni consentite dalla legge sul diritto d'autore e gli altri diritti non sono in alcun modo limitati da quanto sopra.

Questo è un riassunto in linguaggio accessibile a tutti del Codice Legale (la licenza integrale) disponibile all'indirizzo:
`http://creativecommons.org/licenses/by-nc-nd/2.5/it/legalcode`.

Indice

1 Introduzione 3
2 Il package Swing 3
3 Top Level Container 5
3.1 Uso di JFrame 5
4 Paranoramica di alcuni widget 8
5 L'ereditarietà per personalizzare i frame 10
6 I Layout Manager e la gerarchia di contenimento 13
6.1 Layout Management 13
6.2 Progettazione della GUI con le gerarchie di contenimento . . . 17
6.3 Progettazione top down di interfacce grafiche 19
6.3.1 Esempio di Progettazione Top-Down 20
7 La Gestione degli Eventi 23
7.1 Implementazione dell'*event delegation* 23
7.2 Un esempio: elaborare gli eventi del mouse 25
7.3 Uso di adapter nella definizione degli ascoltatori 27
8 La gestione degli eventi Azione 27
9 Accedere dall'ascoltatore agli oggetti di una finestra 29
10 Condividere gli ascoltatori per più oggetti 33

1 Introduzione

Uno dei problemi più grossi emersi durante la progettazione di Java fu senza dubbio la realizzazione di un toolkit grafico capace di funzionare con prestazioni di buon livello su piattaforme molto differenti tra loro. La soluzione adottata nel 1996 fu **AWT**(Abstract Window Toolkit), un package grafico che mappa i componenti del sistema ospite con apposite classi dette peer, scritte in gran parte in codice nativo. In pratica, ogni volta che il programmatore crea un componente AWT e lo inserisce in un'interfaccia grafica, il sistema AWT posiziona sullo schermo un oggetto grafico della piattaforma ospite, e si occupa di inoltrare ad esso tutte le chiamate a metodo effettuate sull'oggetto Java corrispondente, ricorrendo a procedure scritte in buona parte in codice nativo; nel contempo, ogni volta che l'utente manipola un elemento dell'interfaccia grafica, un'apposita routine (scritta sempre in codice nativo) crea un apposito oggetto *Event* e lo inoltra al corrispondente oggetto Java, in modo da permettere al programmatore di gestire il dialogo con il componente e le azioni dell'utente con una sintassi completamente Object Oriented e indipendente dal sistema sottostante.

A causa di questa scelta progettuale, il set di componenti grafici AWT comprende solamente quel limitato insieme di controlli grafici che costituiscono il minimo comune denominatore tra tutti i sistemi a finestre esistenti: un grosso limite rispetto alle reali esigenze dei programmatori. In secondo luogo, questa architettura presenta un grave inconveniente: i programmi grafici AWT assumono un aspetto ed un comportamento differente a seconda della JVM su cui vengono eseguite, a causa delle macroscopiche differenze implementative esistenti tra le versioni di uno stesso componente presenti nelle diverse piattaforme. Spesso le interfacce grafiche realizzate su una particolare piattaforma mostrano grossi difetti se eseguite su un sistema differente, arrivando in casi estremi a risultare inutilizzabili.

Il motto della Sun per Java era "scrivi (il codice) una volta sola ed eseguilo ovunque"; nel caso di AWT questo si era trasformato in "scrivi una volta sola e correggilo ovunque".

2 Il package Swing

Nel 1998, con l'uscita del JDK 1.2, venne introdotto il package **Swing**, i cui componenti erano stati realizzati completamente in Java, ricorrendo unicamente alle primitive di disegno più semplici, tipo "traccia una linea" o "disegna un cerchio", accessibili attraverso i metodi dell'oggetto *Graphics*, un oggetto AWT utilizzato dai componenti Swing per interfacciarsi con la

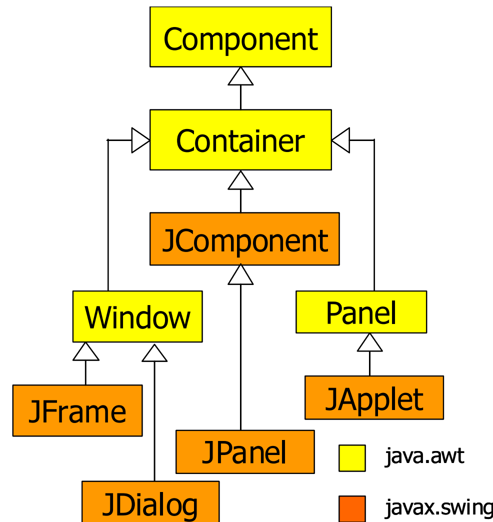


Fig. 1: *Diagramma UML di base del package Swing.*

piattaforma ospite. Le primitive di disegno sono le stesse su tutti i sistemi grafici, e il loro utilizzo non presenta sorprese: il codice java che disegna un pulsante Swing sullo schermo di un PC produrrà lo stesso identico risultato su un Mac o su un sistema Linux. Questa architettura risolve alla radice i problemi di uniformità visuale, visto che la stessa identica libreria viene ora utilizzata, senza alcuna modifca, su qualunque JVM. Liberi dal vincolo del "minimo comune denominatore", i progettisti di Swing hanno scelto di percorrere la via opposta, creando un package ricco di componenti e funzionalità spesso non presenti nella piattaforma ospite. Il procedimento di disegno è ovviamente più lento perché la JVM deve disegnare per proprio conto tutte le linee degli oggetti grafici e gestirne direttamente il comportamento, però è più coerente.

Le classi Swing sono definite nel pacchetto *javax.swing*, il cui nome `javax` indica *estensione standard* a Java. Inizialmente Swing venne rilasciato, infatti, come estensione per poi divenire un omponente standad di Java 2. Per motivi di compatibilita il nome del pacchetto *javax* non venne corretto in *java*. Gli oggetti grafici Swing derivano dai corrispettivi AWT; quindi è possibile utilizzare oggetti Swing, ove erano previsti i oggetti antenati. Swing usa ancora alcuni elementi AWT per disegnare; anche la gestione degli eventi è fatta per la maggior parte da classi AWT.

La Figura 1 riassume molto sinteticamente le classi base del package Swing e come queste derivino da classi AWT. Ogni oggetto grafico (una finestra, un bottone, un campo di testo, . . .) è implementato come classe del package `javax.swing`. Ogni classe che identifica un oggetto Swing deriva

per lo più dalla classe javax.swing.JComponent; si stima che esistono circa 70 o più oggetti diversi. Gli oggetti grafici utilizzati per disegnare le interfacce vengono chiamati anche controlli oppure tecnicamente *widget*. JComponent eredita da java.awt.Container, una sorta di controllo che di default è vuoto e il cui scopo è offrire la possibilità di disporre altri componenti all'interno. Non a caso la classe AWT Window e le sottoclassi Swing JFrame e JDialog, le cui istanze rappresentano finestre, sono sottoclasse di Container. La cosa più sorprendente è che, siccome JComponent deriva da Container, è possibile inserire all'interno di un qualsiasi widget qualsiasi altro. Ad esempio - sebbene poco utile - è possibile aggiungere un campo di testo all'interno di un bottone oppure - molto usato - un Container all'interno di un altro Container. La classe Container (e ogni sottoclasse) definisce un metodo per aggiungere un controllo ad un Container:

```
void add(Component);
```

Il metodo prende come parametro un oggetto Component che è la superclasse di qualsiasi oggetto o container Swing o AWT.

3 Top Level Container

I **top level container** sono i componenti all'interno dei quali si creano le interfacce grafiche: ogni programma grafico ne possiede almeno uno, di solito un **JFrame**, che rappresenta la finestra principale. Ogni top level container possiede un pannello (accessibile tramite il metodo getContentPane()) all'interno del quale vanno disposti i controlli dell'interfaccia grafca. Esistono tre tipi principali di top level Container: JFrame, **JApplet** e **JDialog**. Il primo viene solitamente usato come finestra principale per il programma, il secondo è utilizzato per costruire Applet da visualizzare nella finestra di un web browser mentre il terzo serve a creare le finestre di dialogo con l'utente.

3.1 Uso di JFrame

Un oggetto della classe JFrame può essere creato usando i costruttori:

```
JFrame();
JFrame(String titoloFinestra);
```

Il primo costruisce un JFrame senza titolo; il secondo permette di specificarlo.

È sempre possibile impostare il titolo ricorrendo al metodo setTitle(String s). Due importanti proprietà dell'oggetto sono la dimensione e la posizione, che possono essere impostate sia specificando le singole componenti sia mediante oggetti Dimension e Point del package AWT:

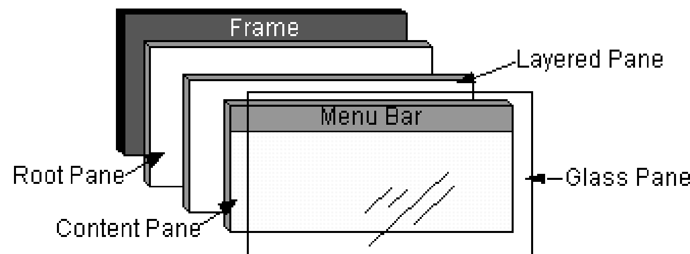


Fig. 2: *Anatomia di un Frame Swing.*

```
public void setSize(Dimension d);
public void setSize(int width,int height);
public void setLocation(Point p);
public void setLocation(int x,int y);
```

Ricorrendo al metodo `setResizable(boolean b)` è possibile stabilire se si vuole permettere all'utente di ridimensionare la finestra manualmente. Infine, vi sono tre metodi piuttosto importanti:

```
public void pack();
public void setVisible(boolean b);
public void setDefaultCloseOperation(int operation);
```

Il primo ridimensiona la finestra tenendo conto delle dimensioni ottimali di ciascuno dei componenti presenti all'interno. Il secondo permette di visualizzare o di nascondere la finestra. Il terzo imposta l'azione da eseguire alla pressione del bottone close, con quattro impostazioni disponibili: `JFrame.DO_NOTHING_ON_CLOSE` (nessun effetto), `JFrame.HIDE_ON_CLOSE` (nasconde la finestra), `JFrame.DISPOSE_ON_CLOSE` (chiude la finestra e libera le risorse di sistema) e `JFrame.EXIT_ON_CLOSE` (chiude la finestra e conclude l'esecuzione del programma).

Per impostazione di default, un JFrame viene costruito non visibile e di dimensione 0 x 0. Per questa ragione, affinchè la finestra sia visibile, è necessario chiamare i metodi `setSize()` *o* `pack()` *per specificare la dimensione e mettere a* TRUE *la proprietà Visible chiamando il metodo:* `setVisible(true)`.

Per poter lavorare con i Frame Swing, è opportuno conoscere il linea generale la struttura della superficie. La superficie di un frame Swing è coperta da quattro lastre:

Glass Pane La lastra di vetro è nascosta di default ed ha il compito di catturare gli eventi di input sulla finestra. Normalmente è completamente trasparente a meno che venga implementato il metodo `paintComponent` del GlassPane. Poichè è davanti a tutte le altre, qualsiasi oggetto disegnato su questa lastra nasconde qualsiasi altro disegnato sulle altre

```
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);
 }
}
```

Listato 1: *La prima finestra*

Content Pane La *lastra dei contenuti* è la più importante perchè è quella che ospita i componenti che volete visualizzare nella finestra e la maggior parte dei programmatori Java lavora solo su questa

Layered Pane Contiene la lastra dei contenuti ed eventualmente i menu. I menu, infatti, non vengono mai aggiunti al Content Pane ma a questa lastra.

Root Pane La lastra radice ospita la lastra di vetro insieme con la lastra dei contenuti e i bordi della finestra.

Per maggiori dettagli si faccia riferimento a [?].

Quindi, un componente, quale un pulsante, un'immagine o altro, non viene aggiunto direttamente alla finestra ma alla lastra dei contenuti. Di conseguenza, occorre per prima cosa procurarsi un riferimento all'oggetto ContentPane, mediante la chiamata al metodo:

```
public Container getContentPane();
```

Come era da aspettarsi, il ContentPane "è un" Container perchè predisposto a contenere altri componenti. A questo punto, come per ogni altro Container, è possibile aggiungere ad esso un componente con il metodo `add` già descritto.

A questo punto è possibile disegnare la prima finestra. Il codice descritto nel Listato 1 mostra la finestra in Figura 3.

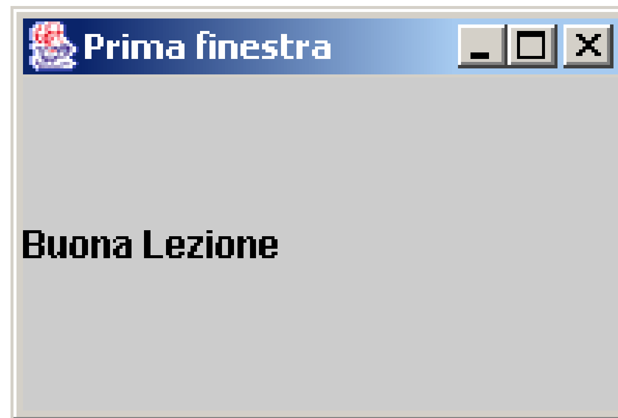


Fig. 3: *La prima finestra.*

4 Paranoramica di alcuni widget

I nomi delle classi per la maggior parte dei componenti dell'interfaccia utente Swing iniziano con la **J**.

JTextField è un campo di testo Swing. Tale classe eredita da TextField, l'obsoleto analogo del package AWT. Per costruire un campo di testo occorre fornirne l'ampiezza, cioè il numero approssimato di caratteri che vi aspettate verranno inseriti dall'utente.

```
JTextField xField=new JTextField(5);
```

Gli utenti possono digitare anche un numero maggiore di caratteri ma sempre 5 contemporaneamente saranno vicini: i 5 attorno alla posizione del cursore nel campo.

Sarebbe opportuno *etichettare* ciascun campo di testo in modo che l'utente sappia cosa scriverci. Ogni etichetta è un oggetto di tipo **JLabel** che viene costruito, sfruttando il costruttore con un parametro stringa; tale parametro rappresenta il testo dell'etichetta:

```
JLabel xField=new JLabel("x␣=␣");
```

Inoltre vorrete dare all'utente la possibilità di inserire informazione in tutti i campi di testo prima di elaborarli per cui avete bisogno di un pulsante che l'utente possa premere per segnalare che i dati sono pronti per essere elaborati. Un pulsante è un oggetto `JButton` che può essere costruito fornendo una stringa che fungerò da etichetta, un'immagine come icona o entrambe

```
JButton moveButton=new JButton("Move");
JButton moveButton=new JButton(new ImageIcon("hand.gif"));
```

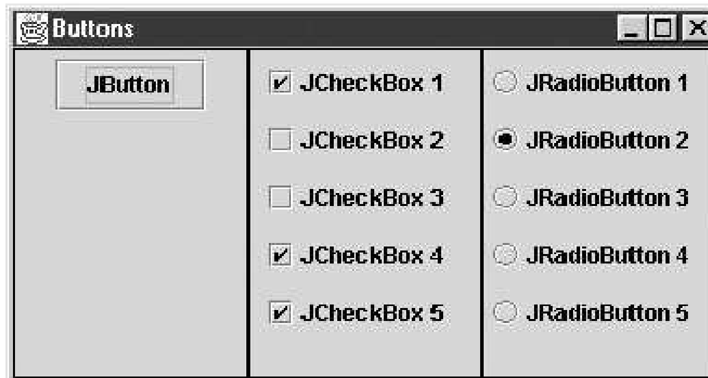


Fig. 4: *Alcuni Bottoni delle Swing.*

```
JButton moveButton=new JButton("Move",new ImageIcon("hand.gif"));
```

JCheckBox è una sottoclasse di JButton che crea caselle di controllo, con un aspetto simile a quello delle caselle di spunta dei questionari. Il suo funzionamento è analogo a quello della superclasse, ma di fatto tende a essere utilizzato in contesti in cui si offre all'utente la possibilità di scegliere una o più opzioni tra un insieme, come avviene per esempio nei pannelli di controllo. I costruttori disponibili sono gli stessi della superclasse, quindi non sarà necessario ripetere quanto è stato già detto.

```
JCheckBox check1=new JCheckBox("JCheck");
```

JRadioButton è una sottoclasse di JButton, dotata dei medesimi costruttori. Questo tipo di controllo, chiamato pulsante di opzione, viene usato tipicamente per fornire all'utente la possibilità di operare una scelta tra un insieme di possibilità, in contesti nei quali un'opzione esclude l'altra. I costruttori disponibili sono gli stessi della superclasse. Per implementare il comportamento di mutua esclusione, è necessario registrare i JRadioButton che costituiscono l'insieme presso un'istanza della classe ButtonGroup, come viene mostrato nelle righe seguenti:

```
JRadioButton radioButton1=new JRadioButton("R1");
JRadioButton radioButton2=new JRadioButton("R2");
JRadioButton radioButton2=new JRadioButton("R3");
ButtonGroup group = new ButtonGroup();
group.add(radioButton1);
group.add(radioButton2);
group.add(radioButton3);
```

Ogni volta che l'utente attiva uno dei pulsanti registrati presso il ButtonGroup, gli altri vengono automaticamente messi a riposo.

I **JComboBox** offrono all'utente la possibilità di effettuare una scelta a partire da un elenco elementi, anche molto lungo. A riposo il componente si presenta come un pulsante, con l'etichetta corrispondente al valore attualmente selezionato. Un clic del mouse provoca la comparsa di un menu provvisto di barra laterale di scorrimento, che mostra le opzioni disponibili. Se si imposta la proprietà editable di un JComboBox a TRUE esso si comporterà a riposo come un JTextField, permettendo all'utente di inserire valori non presenti nella lista. È possibile creare un JComboBox usando i seguenti costruttori:

```
JComboBox();
JComboBox(Object[] items);
```

Il secondo costruttore permette di inizializzare il componente con una lista di elementi di qualsiasi tipo (ad esempio String). Se viene aggiunto al ComboBox un oggetto generico (ad esempio un oggetto Libro), allora il valore corrispondente visualizzato nella lista delle opzioni è quello ottenuto chiamando sull'oggetto il metodo **toString()**. Quindi se si desidera aggiungere oggetti generici (magari definiti all'interno del programma), bisognerà avere la cura di ridefinire tale metodo. Un gruppo di metodi permette di aggiungere, togliere o manipolare gli elementi dell'elenco, così come si fa con un Vector:

```
public void addItem(Object anObject);
public void removeItem(Object anObject)
public void removeItemAt(int anIndex);
public void removeAllItems();
public Object getItemAt(int index);
public int getItemCount();
public void insertItemAt(Object anObject, int index)
```

Per ottenere l'elemento correntemente selezionato, è disponibile il metodo:

```
public Object getSelectedItem();
```

La Figura 5 mostra un esempio di suo uso. Il Listato 2 rappresenta il codice corrispondente:

5 L'ereditarietà per personalizzare i frame

Aggiungendo ad un frame molti componenti dell'interfaccia utente, il frame stesso può diventare abbastanza complesso: per frame che contengono molti componenti è opportuno utilizzare l'ereditarietà. Infatti, se il software che si sta sviluppando contiene molte finestre ricche di componenti ci si troverebbe

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

public class Application {
 public static void main(String args[])
 {
    JFrame win;
    win = new JFrame("Esempio␣di␣JComboBox");
    String lista[]=new String[10];
    for(int i=0;i<lista.length;i++)
       lista[i]="Elemento␣numero␣"+i;
    JComboBox cBox=new JComboBox(lista);
    Container c = win.getContentPane();
    c.add(cBox);
    win.setSize(200,200);
    win.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    win.setVisible(true);
 }
}
```

Listato 2: *Esempio di uso dei ComboBox*

Fig. 5: *Uso di ComboBox.*

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

class MyFrame extends JFrame
{
    JLabel jl = new JLabel("Buona␣Lezione");
    public 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 static void main(String args[])
    {
        MyFrame = new MyFrame();
    }
}
```

Listato 3: *Esempio con l'ereditarietà*

a dover fronteggiare una soluzione il cui caso limite è di un metodo `main` che contiene la definizione di tutte le finestre del programma. Senza arrivare a questa situazione, si otterrebbero moduli sono molto accoppiati e poco coesi; inoltre, gli stessi moduli sarebbero poco leggibili e non sarebbe facile manutenerli. In parole povere, non si sfrutterebbe le potenzialità dell'Object Orientation: non sarebbe possibile permette l'information hiding dei contenuti dei frame e dei controlli in essi contenuti; non sarebbe nemmeno sfruttato l'incapsulamento delle implementazione, mettendo insieme concetti eterogenei e finestre diverse tra loro.

Il Listato 3 produce la stessa finestra in Figura 3 sfruttando l'ereditarietà. La differenza rispetto agli esempi già visti non è da poco: a questo punto ogni finestra diventa una nuova classe che, ereditando da JFrame, è un modello per le finestre. É opportuno definire gli oggetti che mappano i widgets come

variabili di instanza (nel l'esempio l'unico widget è una etichetta (JLabel). La definizione della finestra e dei componenti contenuti viene fatta direttamente nel costruttore (o in metodo chiamati da questo). La prima istruzione chiama il costruttore della superclasse che prende come parametro una stringa; in questo modo è possibile impostare il titolo della finestra. Infatti, se non fosse stato esplicitamente chiamato il costruttore ad un parametro String, sarebbe stato chiamato implicitamente il costruttore della superclasse di default (quello senza parametri) prima di continuare l'esecuzione di quello della classe derivata. Il costruttore senza parametri avrebbe impostato a vuoto il titolo della finestra.[1] Il resto del costruttore è simile alla tecnica di definizione delle finestre degli esempi precedenti con l'unica differenza che i metodi sono chiamati su this (cioè se stesso) perchè a questo punto i metodi da chiamare sono quelli ereditati dalla superclasse JFrame.

6 I Layout Manager e la gerarchia di contenimento

6.1 Layout Management

Quando si dispongono i componenti all'interno di un Container sorge il problema di come gestire il posizionamento: infatti, sebbene sia possibile specificare le coordinate assolute di ogni elemento dell'interfaccia, queste possono cambiare nel corso della vita del programma allorquando la finestra principale venga ridimensionata. In molti Container i controlli sono inseriti da sinistra verso destra con su una ipotetica riga: può non essere sempre la politica per la GUI (Graphic User Interface) desiderata. Per semplificare il lavoro di impaginazione e risolvere questo tipo di problemi possibile ricorrere ai **layout manager**, oggetti che si occupano di gestire la strategia di posizionamento dei componenti all'interno di un contenitore.

Un gestore di layout è una qualsiasi classe che implementa l'interfaccia LayoutManager; ogni container nasce con un certo Layout Manager ma è possibile assegnare il più opportuno per quel Container con il metodo:

```
public void setLayout(LayoutManager m);
```

Il primo layout da essere citato è il *gestore a scorrimento* (**Flow Layout**) che sono inseriti da sinistra verso destra con la loro *Preferred Size*, cioè la dimensione minima necessaria a disegnarlo interamente. Usando il paragone con un editor di testi, ogni controllo rappresenta una "parola" che ha una sua propria dimensione. Come le parole in un editor vengono inseriti da sinistra

[1] A dire il vero, sarebbe stato possibile impostare il titolo della finestra in un secondo momento modificando la proprietà *Title*: `setTitle(‘‘Prima Finestra’’);`

verso destra finchè entrano in una riga, così viene fatto per i controlli inseriti da sinistra verso destra. Quando un componente non entra in una "riga" viene posizionato in quella successiva. I costruttori più importanti di oggetti FlowLayout sono i seguenti:

```
public FlowLayout();
public FlowLayout(int allin);
```

Il secondo costruttore specifica l'allineamento dei controlli su una riga; il parametro può essere una delle seguenti costanti che rispettivamente allineano a sinistra, centro o destra:

```
FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
```

Il costruttore di default imposta l'allineamento centrale. Il Listato 4 mostra un esempio di uso del Flow Layout. L'aspetto del frame risultato è mostrato in Figura 6. É opportuno osservare come l'allineamento dei componenti del frame si adatti durante l'esecuzione quando la stessa finestra viene ridimensionata (vedi situazione 6(b)). Ovviamente se non esiste nessun allineamento per i componenti del frame tale che tutti siano contemporaneamente visibile, allora alcuni di questi risulteranno in parte o del tutto non visibili; per esempio, la finestra è troppo piccola.

Il *gestore a griglia* (**GridLayout**) suddivide il contenitore in una griglia di celle di uguali dimensioni. Le dimensioni della griglia vengono definite mediante il costruttore:

```
public GridLayout(int rows, int columns)
```

in cui i parametri `rows` e `columns` specificano rispettivamente le righe e le colonne della griglia. A differenza di quanto avviene con FlowLayout, i componenti all'interno della griglia assumono automaticamente la stessa dimensione, dividendo equamente lo spazio disponibile. L'esempio descritto nel Listato 5 permette di illustrare il funzionamento di questo pratico layout manager; il risultato è in Figura 7.

Il *gestore a bordi* (**BorderLayout**) suddivide il contenitore esattamente in cinque aree, disposte a croce, come nella Figura 8. Ogni zona può contenere *uno ed un solo widget (o Container)*: un secondo widget inserito in una zona sostituisce il precedente. Se una o più zone non vengono riempite, allora i componenti nelle altre zone sono estesi a riempire le zone vuote. Il BorderLayout è il gestore di layout di default per la Lastra dei Contenuti di un JFrame.

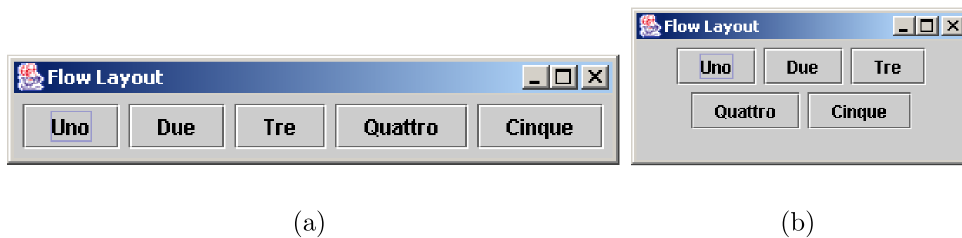
Il programmatore può decidere in quale posizione aggiungere un controllo utilizzando la variante presente nei Container:

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

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

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

Listato 4: *Uso del Flow Layout*

```
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);
    }
}
```

Listato 5: *Uso del Grid Layout*



(a) (b)

Fig. 6: *Aspetto grafico del frame definito nel Listato 4 e comportamento del Flow Layout rispetto al ridimensionamento*

Fig. 7: *Aspetto grafico del frame definito nel Listato 5.*

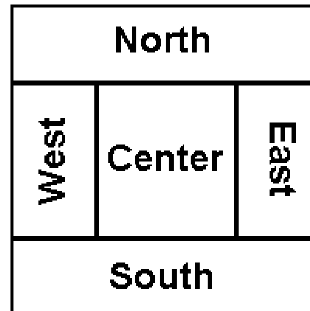


Fig. 8: *Le aree di una disposizione a bordi*

```
public void add(Component c,String s);
```

dove il primo parametro specifica il componente da aggiungere e il secondo indica la posizione. I valori validi per il secondo parametro sono le costanti BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.CENTER, BorderLayout.EAST e BorderLayout.WEST. Si osservi l'esempio nel Listato 6.

6.2 Progettazione della GUI con le gerarchie di contenimento

I gestori di layout permettono maggiore versatilità nell'inserimento dei controlli nelle finestre. Tuttavia nella stragrande maggioranza delle situazioni un'intera finestra non può seguire un unico layout. Si prenda, ad esempio, in considerazione il frame in Figura 9. Non è difficile convincersi che la parte in alto contenente una sorta di "tastierino numerico" segue un GridLayout mentre i bottoni in basso seguono un FlowLayout centrato. Poichè ogni Container può seguire uno ed un unico layout, occorre predisporre più Container, uno per ogni zona che ha un layout differente. Ovviamente ad ogni Container possono essere aggiunti direttamente i controlli oppure altri Container. La lastra dei contenuti è il Container "radice" ed altri si possono aggiungere all'interno. I Container (insieme con i componenti contenuti) aggiunti in un altro Container si comporteranno come ogni altro widget la cui dimensione preferita è quella minima necessaria a visualizzare tutti i componenti all'interno. Per creare nuovi Container conviene utilizzare la classe javax.swing.JPanel che eredita da java.awt.Container.

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 all'interno pannelli interni se ne possono inserire

```
public class MyFrame extends JFrame
{
    JButton nord = new JButton("Nord");
    JButton centro = new JButton("Centro");
    JButton ovest=new JButton("Ovest");

    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(ovest,BorderLayout.WEST);
        setSize(300,300);
        setVisible(true);
    }
}
```

Listato 6: *Uso del Border Layout*

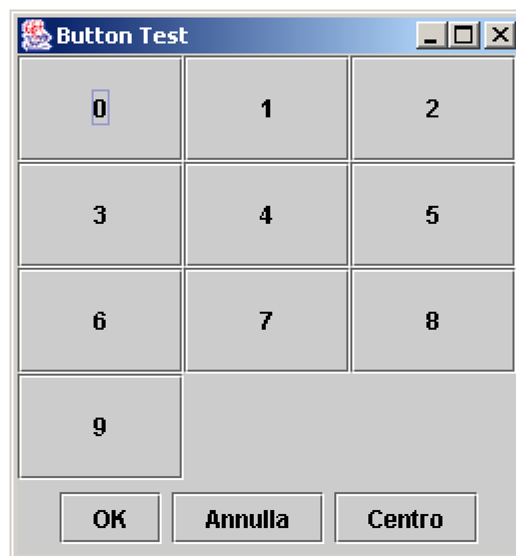


Fig. 9: *Un frame più complesso*

altri con loro layout e così via, come nel gioco delle scatole cinesi. Il numero di soluzioni diverse sono praticamente infinite.

Ad esempio per il frame in Figura 9 è possibile creare due contenitori-pannelli JPanel: uno per contenere il "tastierino numerico" organizzato il GridLayout ed un altro il FlowLayout per i tre bottoni in basso. La lastra dei Contenuti viene lasciata in BorderLayout: al centro viene aggiunto il pannello tastierino ed a sud il pannello con i tre bottoni.

6.3 Progettazione top down di interfacce grafiche

Durante la progettazione delle interfacce grafiche, può essere utile ricorrere a un approccio top down, descrivendo l'insieme dei componenti a partire dal componente più esterno per poi procedere a mano a mano verso quelli più interni. Si può sviluppare una GUI come quella dell'esempio precedente seguendo questa procedura:

1. Si definisce il tipo di top level container su cui si vuole lavorare (tipicamente un JFrame).
2. Si assegna un layout manager al content pane del JFrame, in modo da suddividerne la superficie in aree più piccole.
3. Per ogni area messa a disposizione dal layout manager è possibile definire un nuovo JPanel. Ogni sotto pannello può utilizzare un layout manager differente.
4. Ogni pannello identificato nel terzo passaggio può essere sviluppato ulteriormente, creando al suo interno ulteriori pannelli o disponendo dei controlli.

Il risultato della progettazione può essere rappresentato con un albero della GUI. L'obiettivo di tale albero è mostrare come i Container (a partire dalla *lastra dei contenuti*) sono stati suddivisi per inserire i componenti o altri Container.

Ogni componente (o Container) è rappresentato da un nodo i cui figli sono i componenti (o i Container) contenuti all'interno e il padre è il componente che lo contiene.

Una volta conclusa la fase progettuale, si può passare a scrivere il codice relativo all'interfaccia: in questo secondo momento, è opportuno adottare un approccio bottom up, realizzando dapprima il codice relativo ai componenti atomici, quindi quello dei contenitori e infine quello del JFrame.

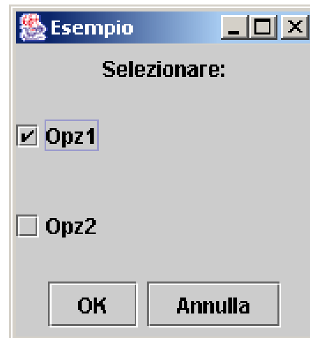


Fig. 10: *Un frame d'esempio*

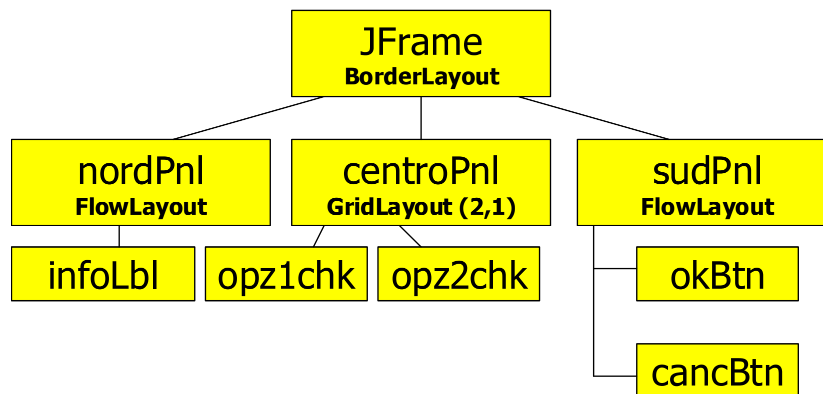


Fig. 11: *Un frame più complesso*

6.3.1 Esempio di Progettazione Top-Down

Un esempio finale viene mostrato per riassumere quanto è stato finora detto. Si inizierà mostrando la progettazione top-down dell'interfaccia e si concluderá mostrandone la realizzazione bottom-up. Si consideri la semplice finestra in Figura 10. L'albero della GUI corrispondente è in Figura 11.

Il codice per mostrare tale finestra è nel Listato 7. Si osservi l'invocazione del metodo `pack()` che sostituisce il metodo `setSize(x,y)` e che imposta la dimensione della finestra alla minima necessaria a visualizzare tutti i controlli. Inoltre si osservi come le due istruzioni successive hanno lo scopo di centrare la finestra sullo schermo. Il metodo `getScreenSize()` chiamato sulla classe *Singleton*[2] Toolkit del package *java.awt* restituisce un riferimento ad un oggetto java.awt.Dimension. Questo possiede due proprietà *Width* e *Height* che, nel caso specifico, conterranno la dimensione dello schermo in pixel. L'istruzione successiva sposta la finestra al centro dello schermo con il

[2] Una classe *Singleton* è tale che di essa esista sempre al più una instanza.

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

Listato 7: *Il codice della figura 10*

metodo `setLocation(int x,int y)` dove `x`, `y` sono le coordinate dell'angolo in alto a sinistra.

7 La Gestione degli Eventi

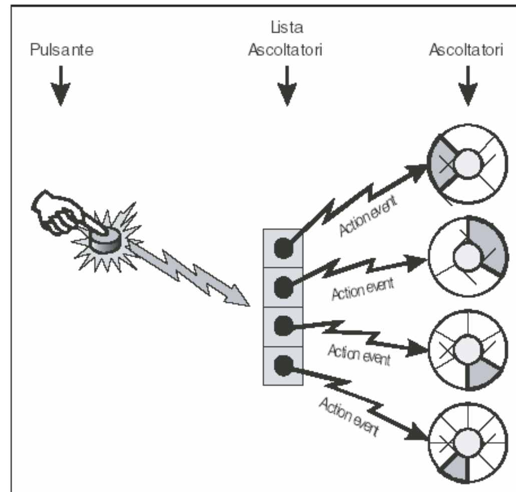


Fig. 12: *La gestione ad* ***event delegation***

La gestione degli eventi grafici in Java segue il paradigma *event delegation* (conosciuto anche come event forwarding). Ogni oggetto grafico è predisposto ad essere sollecitato in qualche modo dall'utente e ad ogni sollecitazione genera eventi che vengono inoltrati ad appositi ascoltatori, che reagiscono agli eventi secondo i desideri del programmatore. L'event delegation presenta il vantaggio di separare la sorgente degli eventi dal comportamento a essi associato: un componente non sa (e non è interessato a sapere) cosa avverrà al momento della sua sollecitazione: esso si limita a notificare ai propri ascoltatori che l'evento che essi attendevano è avvenuto, e questi provvederanno a produrre l'effetto desiderato.

Ogni componente può avere più ascoltatori per un determinato evento o per eventi differenti, come anche è possibile installare uno stesso ascoltatore su più componenti anche diversi a patto che entrambi possono essere sollecitati per generare l'evento. L'operazione di installare lo stesso ascoltatore su più controlli (che quindi si comporteranno nello stesso modo se sollecitati dallo stesso evento) è frequente: si pensi alle voci di un menu che vengono replicate su una toolbar per un più facile accesso ad alcune funzionalità frequenti

7.1 Implementazione dell'event delegation

Il gestore delle finestre può generare un numero enorme di eventi (sono eventi muovere il mouse nella finestra, spostare o ridimensionare una finestra, scri-

vere o abbandonare un campo di testo, ...). Molte volte si è interessati a pochi di questi eventi. Per fortuna, il programmatore può considerare solo gli eventi di cui ha interesse, gestendoli con un ascoltatore opportuno, ignorando completamente tutti gli altri che di default verranno gestiti come eventi vuoti.

Quando accade un evento per il quale esiste un ricevitore, la sorgente dell'evento chiama i metodi da voi forniti nell ricevitore, dando, in un oggetto di una *classe evento*, informazioni più dettagliate sull'evento stesso. In generale sono coinvolte tre classi:

La classe del ricevitore. 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. Grazie al fatto che ogni ascoltatore è caratterizzato da una particolare interfaccia Java, qualsiasi classe può comportarsi come un ascoltatore, a patto che fornisca un'implementazione per i metodi definiti dalla corrispondente interfaccia listener

L'origine dell'evento. É il componente che genera l'evento che si vuole gestire su cui si vuole "installare" l'ascoltatore. Ogni componente dispone per ogni evento supportato di due metodi:

```
void addXXXListener(XXXListener ascoltatore);
void removeXXXListener(XXXListener ascoltatore);
```

La classe evento. 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. Nel momento in cui il componente viene sollecitato, esso chiama gli ascoltatori in modalità *callback*, passando come parametro della chiamata un'apposita sottoclasse di Event. Queste classi contengono tutte le informazioni significative per l'evento stesso e possono essere utilizzate per modificare il comportamento dell'ascoltatore in base alle informazioni sull'evento scatenato

Le classi che permettono di gestire gli eventi generati dai componenti Swing sono in gran parte gli stessi utilizzati dai corrispondenti componenti AWT, e si trovano nel package *java.awt.event*. Alcuni componenti Swing, tuttavia, non hanno un omologo componente AWT: in questo caso le classi necessarie a gestirne gli eventi sono presenti nel package *javax.swing.event*.

7.2 Un esempio: elaborare gli eventi del mouse

Per chiarire il funzionamento, a prima vista complesso, della gestione degli eventi Swing, verra fatto un esempio. In particolare, vogliamo realizzare un meccanismo per "spiare" gli eventi del mouse su una finestra e stamparli.

Una classe che desidera catturare gli eventi del mouse deve implementare l'interfaccia
MouseListener definita nel package *java.awt.event* che è così definita:

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

dove MouseEvent è la corrispondente classe evento il cui scopo è dare informazioni aggiuntive sull'evento del mouse. Infatti, questa definisce due metodi che permettono di conoscere le coordinate del mouse allo scatenarsi dell'evento (i primi due) e un terzo metodo che permette di determinare quale bottone del mouse è stato premuto:

```
int getX();
int getY();
int getModifiers()
```

Nel codice del Listato 9 viene mostrata l'implementazione della "spia". Si noti come la classe ascoltatore *MouseSpy* implementi l'interfaccia MouseListener e come la classe definisca due metodi vuoti: *mouseEntered* e *mouseExited*. La definizione dei due metodi vuoti ha lo scopo di dichiarare che gli eventi legati all'entrata e all'uscita del mouse dall'area della finestra devono essere ignorati. In effetti, ignorare gli eventi è già il comportamento di default e quindi tale dichiarazione è superflua. La ragione per cui è stata comunque dichiarata la gestione di tali eventi è legata al linguaggio Java: affinché una classe implementi una interfaccia, tale classe deve implentare tutti i metodi dell'interfaccia. E questo vale anche in questo caso: la classe *MouseSpy* deve implementare tutti i metodi dell'interfaccia *MouseListener* per poter dichiarare di implementarla.

Nella Tabella 1 sono riassunti gli ascoltatori più importanti. Per ulteriori dettagli si consiglia di consultare la documentazione Java Swing.

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

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) {}
    public void mouseExited(MouseEvent e) {}
}

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

Listato 8: *Primo esempio della gestione degli eventi*

`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 e trascinato o spostato
`MouseListener`	Se ne è 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.

Tab. 1: *Gli ascoltatori più importanti*

7.3 Uso di adapter nella definizione degli ascoltatori

Alcuni componenti grafici generano eventi così articolati da richiedere, per la loro gestione, ascoltatori caratterizzati da più di un metodo. Si è già visto che MouseListener, ne definisce tre. L'interfaccia WindowListener (vedi Listato 9) ne dichiara addirittura sette, ossia uno per ogni possibile cambiamento di stato della finestra. Si è detto che se si desidera creare un ascoltatore interessato a un solo evento (per esempio uno che intervenga quando la finestra viene chiusa), esso dovrà comunque fornire un'implementazione vuota anche dei metodi cui non è interessato. Nei casi come questo è possibile ricorrere agli **adapter**: classi di libreria che forniscono un'implementazione vuota di un determinato ascoltatore. Per esempio, il package *java.awt.event* contiene la classe WindowAdapter, che fornisce un'implementazione vuota di tutti i metodi previsti dall'interfaccia. Una sottoclasse di *WindowAdapter*, pertanto, è un valido *WindowListener*, con in più il vantaggio di una maggior concisione. Si veda l'esempio 10

8 La gestione degli eventi Azione

La maggior parte dei componenti Swing generano eventi, noti come eventi *azione*, che possono essere catturati da un ActionListener. Essa definisce un unico metodo:

```
public interface WindowListener
{
    public void windowOpened(WindowEvent e);
    public void windowClosing(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowActivated(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
}
```

Listato 9: *L'interfaccia MouseListener*

```
class WindowClosedListener extends WindowAdapter {
    public void windowClosed(WindowEvent e)
    {
        // codice da eseguire alla chiusura della finestra
    }
}
```

Listato 10: *Esempio dell'uso degli Adapter*

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

L'oggetto ActionEvent da informazioni aggiuntive sull'evento generato. Ad esempio, definisce il metodo `Object getSource()` che restituisce l'oggetto che ha generato l'evento.

La maggior parte dei widget java sono predisposti per generare eventi azione. Ad esempio quando si clicca su un bottone, si preme INVIO in un campo di testo, si seleziona una voce di un menu oppure si seleziona/deseleziona una voce di una checkBox o un radioButton, viene generato un evento azione. Si potrebbe obiettare che, la gestione del click di un bottone, potrebbe essere fatta usando un mouseListener. Il vantaggio di utilizzare un evento azione risiede nel fatto che è possibile cliccare su un bottone anche con la tastiera (anche se in quel caso non è un vero click!): l'utente con il tasto TAB potrebbe spostarsi sulla schermata, mettere il focus su un bottone e qui premere la barra spaziatrice per voler "cliccare" sul bottone.

É 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

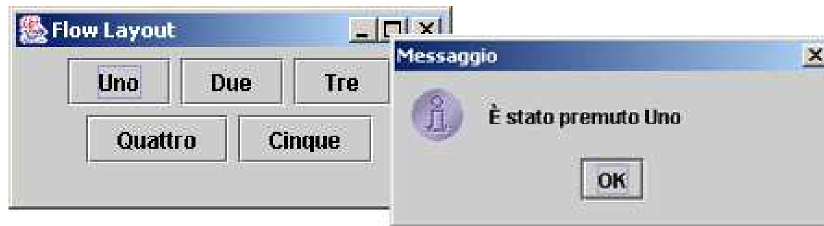


Fig. 13: *La finestra ottenuta dal Listato 11*

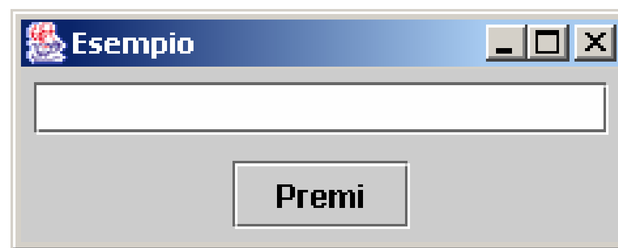


Fig. 14: *La finestra ottenuta dal Listato 12*

del menu che il bottone hanno lo stesso ascoltatore invece che due ascoltatori separati che fanno la stessa cosa.

Un primo esempio di uso degli eventi azione è nel Listato 11. Il comportamento è quello in Figura 13: quando l'utente seleziona uno dei bottoni della finestra `MyFrame` viene aperta una finestra di dialogo con il testo del bottone premuto. Su ogni bottone è installato lo stesso ascoltatore per gli eventi azione. Alla pressione di un bottone viene eseguito il metodo actionPerformed, il quale si fa restituire con il metodo `getSource()` il bottone premuto (si noti il cast giacchè il metodo lo restituisce come riferimento ad Object). A partire dal bottone premuto, con il metodo `getText()` viene restituito il testo visualizzato nel bottone.

9 Accedere dall'ascoltatore agli oggetti di una finestra

L'approccio finora descritto funziona abbastanza bene anche se presenta ancora alcune problematiche. Il primo problema è legato al fatto che la finestra e i suoi ascoltatori sono divisi in classi separate. Supponiamo di avere ascoltatori per gestire gli eventi scatenati da alcuni componenti installati in una data finestra. Essendo le due classi formalmente indipendenti tra loro, per quanto detto, non è possibile accedere dagli ascoltatori a tutti i componenti della finestra (tranne il componente che ha generato l'evento).

Supponiamo di voler realizzare una finestra come quella in Figura 14.

```
public class MyFrame extends JFrame
{
  private JButton uno = new JButton("Uno");
  private JButton due = new JButton("Due");
  private JButton tre = new JButton("Tre");
  private JButton quattro = new JButton("Quattro");
  private 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());
    }
}
```

Listato 11: *Esempio uso di ActionPerformed*

```
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 {
  public void actionPerformed(ActionEvent e) {
    JTextField text = new JTextField();
    JOptionPane.showMessageDialog(
      null,text.getText());
  }
}
```

Listato 12: *Un esempio che non funziona*

Alla pressione del bottone OK il contenuto del campo di testo deve essere visualizzato in un message dialog box. Si consideri il codice nel Listato 12. Nell'ascoltatore definiamo un nuovo JTextField `text`. Quest'approccio non funziona perchè il campo di testo definito è un nuovo campo di testo e non quello della finestra. Quindi la pressione del bottone mostrerà sempre una finestra dal contenuto vuoto perchè di default un campo di testo viene inizializzato con il contenuto vuoto.

Un primo approccio è utilizzare una *classe interna*. Una classe interna è definita all'interno di un'altra classe e può accedere a tutte le variabili e i metodi di istanza della classe esterna (anche a quelli privati). Si consideri il Listato 13 che fa uso della classi interne: l'istruzione `text = txt` nell'ascoltatore fa sì che il riferimento `text` "punti" allo stesso campo di testo riferito da `txt`. Quindi quando nel codice successivo, l'ascoltatore chiede `text.getText()` ottiene effettivamente il codice richiesto.

Questa tecnica è accettabile se l'ascoltatore esegue poche operazioni e

```
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);
    ...
    button.addActionListener(new Listen());
    ...
  }

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

Listato 13: *Una soluzione che funziona con le classi interne*

pochi ascoltatori esistono. Altrimenti la classe `MyFrame` diventa eccessivamente grande. Inoltre il codice ottenuto diventa poco leggibile perchè si tende ad accorpare classi che rappresentano concetti diversi.

La soluzione preferibile è quella in cui l'ascoltatore definisce un costruttore che prende come parametro un riferimento alla finestra con i componenti richiesti. Il costruttore memorizza tale riferimento come variabile di classe. La classe ascoltatore resta comunque esterna: essa accede ai widget della finestra tramite tale riferimento e l'operatore punto. Ovviamente i widget non devono essere memorizzati privati. La soluzione è descritta nel Listato 14. Nel codice viene anche mostrato una grande potenzialità degli eventi azione: l'evento azione del campo di testo è gestito nello stesso modo della finestra. Ne risulta, di conseguenza, che la pressione del bottone e la pressione del tasto ENTER sul campo di testo vengono gestiti dallo stesso ActionListener e quindi gestiti nello stesso modo.

10 Condividere gli ascoltatori per più oggetti

La tecnica più naive per gestire più eventi associati ad un controllo è quello di creare una classe "ascoltatore" per ogni oggetto e per ogni classe di eventi da gestire per quello oggetto. Ciò vorrebbe dire che se dovessimo gestire l'evento di pressione di X bottoni, dovrebbe realizzare X classi che implementano ActionListener.

Ovviamente è nella pratica improponibile prevedere una classe per ogni bottone, voce del menu e così via. Infatti, consideriamo per esempio un'applicazione con 5 finestre, ognuna con 5 bottoni. Supponiamo che una di tali finestre abbia anche 3 voci del menu di 4 opzioni ciascuno. Ciò significherebbe che avremmo 37 classi solo per gestire la pressione dei bottoni o della voci del menu!

L'idea più elegante è raggruppare gli oggetti su cui ascoltare in classi, prevedendo un ascoltatore condiviso per tutti gli oggetti della stessa classe.

Da questo punto in poi consideriamo solo gli eventi *actionListener* senza perdere di generalità. Lo stesso approccio si applica in tutti gli altri casi. I componenti della stessa classe, ovviamente, condivideranno lo stesso metodo `actionPerformed`. A meno che tutti i componenti della stessa classe si devo comportare nello stesso modo allo scatenarsi dell'evento *azione*, occore essere in grado di "capire" quale oggetto ha generato l'evento. Questo al fine di capire quale comportamente adottare.

Esistono due modi per capire "chi" ha generato l'evento:

1. Attraverso il metodo `getSource()` della classe *ActionEvent* che restituisce un riferimento all'oggetto che ha scatenato l'evento. In questo

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

    public Listen(MyFrame aFrame)
    { frame = aFrame; }

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

Listato 14: *La soluzione migliore*

modo è possibile utilizzare un approccio di tipo switch/case: se il riferimento "punta" a X, allora chiama un metodo privato che corrisponde alla gestione di pressione del bottone X; se il riferimento "punta" a Y, allora chiama il metodo per Y. E cosi via. 'E ovviamente chiaro se questa approccio è fattibile se è possibile confrontare i riferimenti ottenuti con il metodo `getSource()` con i riferimenti memorizzati internamente alla classe che estende *JFrame* e che disegna la finestra con i bottoni X, Y e gli altri. Questo approccio è generalmente fattibile, quindi, con le classi interne.

2. Utilizzando la proprietà *actionCommand*, implementata per ogni componente, che permette di associare una stringa identificativa univoca ad ogni componente che scatena un evento azione. A questo punto è possibile definire all'interno del metodo `actionPerformed` un approccio di tipo switch/case sugli actionCommand. Se lo actionCommand dell'oggetto è "ACTX" (che è stato associato ad X) allora l'ascoltatore esegue una certa porzione di codice, se lo actionCommand è "ACTY" fa un'altra cosa. E così via. In questo nuovo contesto non è più necessario per l'ascoltatore di avere accesso ai riferimenti.

Si consideri, ad esempio, la porzione di condice nel Listato 15 dove ci sono tre voci di menu `UpOpt`, `DownOpt`, `RandomOpt`. Le tre voci del menu hanno associati lo stesso ascoltatore con lo stesso metodo *actionPerformed*. All'interno del metodo la discriminazione su come l'ascoltatore si deve comportare in funzione del metodo invocato è fatto analizzando i riferimenti. L'istruzione `src=e.getSource()` restituisce un riferimento all'oggetto che ha generato l'evento. Se tale riferimento è uguale a `UpOpt` (entrambi puntano allo stesso oggetto corrispondente alla voce del menu "Up", allora la voce del menu scelta è *Up*. Quindi viene eseguita la porzione di codice relativa a tale voce. Lo stesso per le altre voci. L'ascoltatore è realizzato come classe interna in modo tale da poter realizzare l'uguaglianza tra i puntato e verificare se coincidono.

Come si è detto, l'approccio con le classi interne ha molti svantaggi. la metodologia che vogliamo qui introdurre per capire "chi" ha generato l'evento non deve assumere classi interne e, quindi, non può basarsi sull'uguaglianza tra puntatori.

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 nel Listato 16 i possibili valori per le proprietà *actionCommand* sono memorizzate come costanti stringa, cioè `public final static`, della classe ascoltatore `Listener`. Ad ogni oggetto da predisporre per gestire

```
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 }
     }
  }
}
```

Listato 15: *Ascoltatore condiviso come classe Interna*

```
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.setActionCommand(ascolt.RANDOMOPT)
      ...
  }
}
```

Listato 16: *Il JFrame per usare ascoltatori esterni*

gli eventi azione viene associato un valore per la *actionCommand* presi tra tali costanti stringa. Questo viene fatto grazie al metodo

```
void setActionCommand(String);
```

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. Eventualmente è possibile associare lo stesso actionCommand a componenti gestiti dallo stesso ascoltatore se si desidera che questi gestiscano l'evento azione nello stesso modo. Questo cosa è molto utile quando si desidera replicare la voce di un dato menu con un bottone da inserire nella toolbar della finestra: per fare sì che sia la voce del menu che il bottone nella toolbar gestiscano l'evento nello stesso modo è sufficiente associare ad entrambi lo stesso actionCommand.

Concludiamo il capitolo con l'ascoltatore relativo all'esempio nel Listato 16. Il suo scheletro è nel Listato 17. Si possono osservare sempre tre parti in questo tipo di ascoltatori:

- La parte che definisce le costanti stringa (le tre dichiarazioni di oggetti pubblici, costanti[3] e statici)

[3] In Java è possibile definire che il valore di un riferimento non può cambiare dichiaran-

```
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()
     { ... }
}
```

Listato 17: *Ascoltatore condiviso come classe Interna*

- Il metodo *actionPerformed* che gestisce l'evento. Questo metodo in questi casi non fa nient'altro che leggere il valore della proprietà *actionCommand* dell'oggetto che ha generato l'evento. In funzione del valore della proprietà (e quindi dell'oggetto scatenante), viene attivata una gestione separata che consiste nell'invocare metodi diversi.
- I diversi metodi privati che eseguono effettivamente le operazioni associate all'attivazione dei diversi controlli/widget.

---

dolo `final`. Questo ovviamente non è sufficiente per dichiarare un oggetto costante. Tuttavia nel caso di oggetti `String` le due cose coincidono dato che gli oggetti stringa sono immutabili: non può cambiare il riferimento, nè può cambiare il valore dell'oggetto puntato