Swing Espresso.

Uno scrittore è tanto più temibile quanto minore è il numero di pagine che gli sono necessarie per annoiare il suo lettore.

Introduzione (breve).

Questo testo non è un trattato esaustivo sulle capacità del framework Swing ma una descrizione volutamente sintetica di alcune caratteristiche e funzioni di Swing. Lo scopo è di concentrare in una lettura di una o due ore le conoscenze necessarie a produrre un'interfaccia a finestre funzionante che faccia uso di alcuni controlli predefiniti – pulsanti, aree di testo, tabelle, alberi, liste eccetera. Il testo fa riferimento alla versione 6 della piattaforma Java SE. All'inizio della stesura il testo aveva un limite di cinquata pagine requisito che, incredibilmente, sono riuscito a rispettare. Va da sé che i volumi da mille e passa pagine non sono scritti da rimbambiti: sono il minimo necessario per un'esposizione completa. Qui si tratta invece di una rapida panoramica con un tanto di utilità.

Swing e concorrenza.

Il framework Swing è a Thread singolo. Significa che gran parte dei metodi delle classi Swing è progettato per essere invocato da un solo Thread. Swing specifica anche quale deva essere questo Thread: l'EDT o Event Dispatcher Thread. L'EDT è preso in prestito da AWT. La morale è che quando si eseguono operazioni su istanze di oggetti Swing occorre fare in modo che tali operazioni siano eseguite dall'EDT. Ci sono due strumenti di base per far sì che l'EDT esegua una sequenza arbitraria di istruzioni: `Runnable` e `EventQueue`. Il metodo è il seguente:

1. si dichiarano le istruzioni che manipolano oggetti Swing nel metodo run di una classe che concretizza `Runnable`
2. si crea un'istanza di quella classe
3. si passa l'istanza creata al metodo statico `invokeLater` di `EventQueue`

Esempio: si vuole creare e aprire una finestra sul desktop. La finestra è un oggetto istanza di `javax.swing.JFrame`. Essendo un oggetto Swing dobbiamo crearlo e gestirlo nel Thread EDT.

```
import javax.swing.JFrame;
import java.awt.EventQueue;

public class Main {

     public static void main(String[] args) {
          class Starter implements Runnable {

               public void run() {
                    JFrame window = new JFrame("Hello world");
                    window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
                    window.setSize(400, 300);
                    window.setVisible(true);
               }
          }
          Runnable task = new Starter();
          EventQueue.invokeLater(task);
     }
```

```
}
```

Usando l'istanziazione di classe interna locale anonima che concretizza un'interfaccia otteniamo la forma più comune:

```
import javax.swing.JFrame;
import java.awt.EventQueue;

public class Main {

      public static void main(String[] args) {
            EventQueue.invokeLater(new Runnable() {

                  public void run() {
                        JFrame window = new JFrame("Hello world");
                        window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
                        window.setSize(400, 300);
                        window.setVisible(true);
                  }
            });
      }
}
```

In un'applicazione con interfaccia Swing la “vita” dell'EDT contribuisce a determinare la persistenza del programma. La JVM resta attiva finché è attivo almeno un Thread non demone. L'EDT è un Thread non demone ergo la JVM, a meno di altri Thread, resta attiva finché l'EDT non ha completato i propri compiti. L'EDT sia avvia automaticamente quando è aggiunto un compito nella coda degli eventi di sistema e persiste finché c'è almeno un compito nella coda o finché almeno una finestra AWT o Swing si trova nello stato “visualizzabile”. Una finestra Swing – `JFrame`, `JDialog`, `JWindow` – diventa visualizzabile dopo l'invocazione di uno tra i metodi `show()` - deprecato – `setVisible(true)` o `pack()` sulla finestra stessa. La finestra permane nello stato visualizzabile fino alla successiva invocazione del suo metodo `dispose()`. L'invocazione di `setVisible(false)` elimina la finestra dal desktop ma non comporta una perdita di visualizzabilità della stessa.

Modello di gestione degli eventi.

L'evento è qualcosa che capita nel corso del programma. Ci sono due specie di eventi in Swing: gli eventi di basso livello e gli eventi semantici. Gli eventi di basso livello sono quelli prodotti dal sistema operativo eventualmente in risposta all'interazione utente – tastiera, touch screen, lettori ottici, mouse e compagnia bella – gli eventi semantici sono quelli prodotti da un componente Swing come conseguenza di un evento di basso livello o in seguito ad una mutazione di stato generata dal framework. Quando l'utente preme un pulsante del mouse nella regione di spazio occupata da una finestra Swing quest'ultimo crea un evento `java.awt.event.MousePressed`. Se nel punto di pressione c'è un pulsante `javax.swing.JButton` e il pulsante premuto è quello sinistro allora il pulsante genera un evento `javax.swing.event.ChangeEvent`. Quando l'utente rilascia il mouse nello stesso punto si produce un evento `java.awt.event.MouseReleased` ed un evento `java.awt.event.ActionEvent`. Il primo è un evento di basso livello, il secondo è un evento semantico. La differenza tra eventi di basso livello ed eventi semantici è che i primi sono vincolati alla rappresentazione del tipo di interazione utente che si è verificata – pressione del mouse, rilascio di un pulsante sulla tastiera, acquisizione del focus da parte di una finestra – i secondi interpretano i primi alla luce dello scopo del componente che subisce l'interazione. Ad esempio un `MousePressed` è generato solo ed esclusivamente in seguito alla pressione, reale o simulata, di un pulsante del mouse e testimonia l'avvenuta pressione di un pulsante del mouse mentre un `ActionEvent` può essere generato in seguito ad un evento del mouse o della tastiera e testimonia l'avvenuta attivazione del controllo che ha prodotto l'ActionEvent. La differenza è importante perché il framework punta più sugli eventi

semantici che su quelli di basso livello quando è ora di capire se un tal pulsante sia stato premuto o un il valore contenuto in una riga della tal tabella sia cambiato o se si sia verificata una selezione in una lista: tutte cose scatenate da eventi di basso livello ma testimoniate da eventi semantici. Posta questa distinzione di base il meccanismo di gestione degli eventi è omogeneo – sempre quello – e tripartito: Evento, Sorgente, Ascoltatore. Riproduciamo il fenomeno con un esempio svincolato da Swing. Da una parte abbiamo la definizione di un Evento:

```
public class Evento {

	private final Object SORGENTE;
	private final String MESSAGGIO;

	public Evento(Object sorgente, String messaggio) {
		MESSAGGIO = messaggio;
		SORGENTE = sorgente;
	}

	public String getMessaggio() {
		return MESSAGGIO;
	}

	public Object getSorgente() {
		return SORGENTE;
	}
}
```

Un Evento è nient'altro che una capsula di informazioni. Segue la definizione di qualcosa che è in grado di ricevere una notifica di avvenuta produzione di un Evento: l'Ascoltatore. Classica interfaccia.

```
public interface AscoltatoreEvento {

	void eventoProdotto(Evento e);
}
```

Infine arriviamo a colui che produce eventi: la Sorgente. Quando e perché la Sorgente produca eventi dipende dalla sorgente e dalla ragione per cui esiste. Prendiamo come fatto tipico che causa la produzione di un Evento il trascorrere di un intervallo di tempo.

```
import java.util.concurrent.*;

public class Sorgente {
	private final CopyOnWriteArrayList<AscoltatoreEvento> registroAscoltatori =
		new CopyOnWriteArrayList<AscoltatoreEvento>();

	public void aggiungiAscoltatore(AscoltatoreEvento e) {
		registroAscoltatori.add(e);
	}

	private void creaENotificaEvento() {
		Evento evento = new Evento(this, "Sono passati cinque secondi circa");
		for(AscoltatoreEvento e : registroAscoltatori) {
			e.eventoProdotto(evento);
		}
	}

	public void avvia() {
		new Thread() {
			@Override public void run() {
				for(int i = 0; i < 3; i++) {
					pausa();
					creaENotificaEvento();
				}
			}
		}.start();
```

```
	}

	private void pausa() {
		try {
			Thread.sleep(5000);
		} catch(InterruptedException ex) {
			//la ignoriamo per brevità ma in genere non si ignora mai
			//una InterruptedException
		}
	}
}
```

La struttura di una sorgente è tipica, ha un modo per aggiungere degli ascoltatori di evento e quando capita qualcosa di interessante altro non fa che incapsulare le informazioni su ciò che è capitato in un evento e rifilare quell'evento ad ogni ascoltatore aggiunto alla sorgente. Ora se noi creiamo una Sorgente, definiamo un AscoltatoreEvento che stampa sulla console l'evento ricevuto attraverso il proprio metodo eventoProdotto, aggiungiamo quell'ascoltatore alla sorgente e avviamo quest'ultima ogni cinque secondi per tre volte otterremo un messaggio sulla console.

```
public class Main {

	public static void main(String[] args) {
		AscoltatoreEvento ascoltatore = new AscoltatoreEvento() {

			public void eventoProdotto(Evento e) {
				System.out.println("Notifica ricevuta: " + e.getMessaggio());
			}
		};

		Sorgente sorgente = new Sorgente();
		sorgente.aggiungiAscoltatore(ascoltatore);
		sorgente.avvia();
	}
}
```

Questo è lo schema di gestione degli eventi in Swing. Quando avete per le mani un componente Swing e volete fare qualcosa in risposta ad un fatto che può capitare a quel componente dovete prendere la documentazione del componente e cercare un metodo addXYZListener. XYZ è il tipo di ascoltatore che deve essere aggiunto al componente per poter intercettare gli eventi XYZ e XYZ deve essere il tipo di evento che volete gestire. Se è un evento del mouse XYZ sarà MouseListener, se è un evento di movimento del mouse allora sarà `MouseMotionListener`, se è un evento della tastiera sarà `KeyListener`, se è un evento di selezione sarà qualcosa tipo `SelectionListener`. Si va a tentativi “orecchiabili” confidando nel buon giudizio dei progettisti di Swing.

Contenitori e contenuti.

Le interfacce Swing si creano a colpi di contenitori e contenuti. Un contenitore è un componente Swing in grado di ospitare altri componenti. Tutti i componenti Swing sono anche contenitori per derivazione da java.awt.Container ma solo alcuni sono tipicamente adatti al contenimento. Il contenitore per eccellenza è `JPanel` a fianco del quale troviamo alcuni contenitore specializzati come `JToolBar`, `JSplitPane` o `JScrollPane`. Dire che un contenitore contiene dei componenti significa dire che i componenti contenuti sono disegnati nella regione di spazio occupata dal loro contenitore. L'aspetto da considerare in questo fatto è che il contenuto si sposta insieme al suo contenitore. Per darvi un'idea, se prendo un contenitore e gli aggiungo un pulsante in basso a destra dopodichè prendo il contenitore e lo aggiungo ad un secondo contenitore in alto a sinistra finisce che il pulsante me lo ritrovo al centro: quando aggiungo il primo contenitore nella parte in alto a sinistra del secondo contenitore il pulsante che appartiene al primo contenitore subisce anch'esso uno spostamento. Questa relatività delle posizioni è fondamentale nella costruzione di interfacce

Swing in virtù della gestione degli spazi attraverso i `LayoutManager`. In linea di principio in Swing non si specifica mai la posizione di un componente nello spazio della finestra. Quello che si fa è stabilire che lo spazio occupato da un contenitore sarà distribuito tra il suo contenuto in un certo modo dopodichè si aggiungono i componenti al contenitore e il risultato è che questi componenti vanno ad occupare certi sottospazi del contenitore. La ragione per cui non si usa un posizionamento diretto per coordinate e dimensioni deriva dalla dipendenza delle dimensioni a video di un componente da parametri che non sono controllabili attraverso la piattaforma Java. I poltergeist in questione sono la risoluzione del desktop, le decorazioni predefinite delle finestre, le dimensioni e il tipo di font eccetera. Se prendiamo un pulsante con la scritta “Ciao Mondo” e stabiliamo che deva essere grande cento pixel per cento pixel può capitare che per dimensioni di font sufficientemente grandi la scritta non si veda affatto. Se stabilisco di disegnare il pulsante sull'asse x = 10 e la barra del titolo della finestra risulta più alta di 10 pixel un pezzo del pulsante sarà coperto dalla barra del titolo. Un certo posizionamento assoluto può funzionare nella versione corrente del mio sistema operativo ma può non dare gli stessi risultati in una versione successiva o precedente o nella stessa versione ma con un tema diverso. Il punto del posizionamento relativo tramite dei gestori di layout risiede proprio nel lasciare che il framework computi durante la costruzione dell'interfaccia posizioni e dimensioni tali da consentire che l'aspetto dell'interfaccia sia proporzionalmente corretto ed ogni componente risulti visibile. Esempio: creo una finestra con un pulsante in alto a destra, un'area di testo al centro e un pulsante in basso a sinistra.

Creo la finestra (senza visualizzarla):

```
JFrame window = new JFrame(“Titolo”);
```

Creo il pulsante che andrà in alto:

```
JButton pulsanteInAlto = new JButton(“Pulsante in Alto”);
```

Creo un contenitore per il pulsante in alto:

```
JPanel contenitoreAlto = new JPanel();
```

Dico a questo contenitore che il suo contenuto dovrà essere collocato per linee orizzontali, partendo dall'angolo in alto a destra:

```
LayoutManager layout = new FlowLayout(FlowLayout.RIGHT);
contenitoreAlto.setLayout(layout);
```

Aggiungo il pulsante al contenitore:

```
contenitoreAlto.add(pulsanteInAlto);
```

Aggiungo il contenitore del pulsante nella parte alta della finestra:

```
window.add(contenitoreAlto, BorderLayout.NORTH);
```

Creo un'area di testo:

```
JTextArea textArea = new JTextArea();
```

E la inserisco nella parte centrale della finestra:

```
window.add(textArea, BorderLayout.CENTER);
```

Creo il pulsante che andrà in basso a sinistra:

```
JButton pulsanteInBasso = new JButton("Pulsante in basso");
```

Come prima creo un contenitore per quel pulsante e uso un LayoutManager che colloca il contenuto per linee orizzontali, partendo dall'alto (del contenitore) stavolta a sinistra:

```
JPanel contenitoreBasso = new JPanel(new FlowLayout(FlowLayout.LEFT));
```

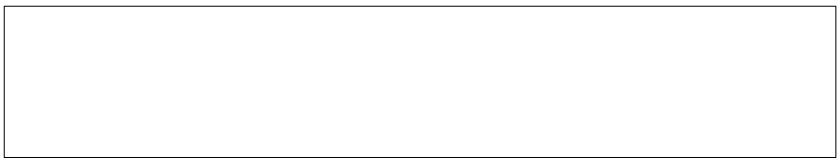
Aggiungo il suo pulsante:

```
contenitoreBasso.add(pulsanteInBasso);
```

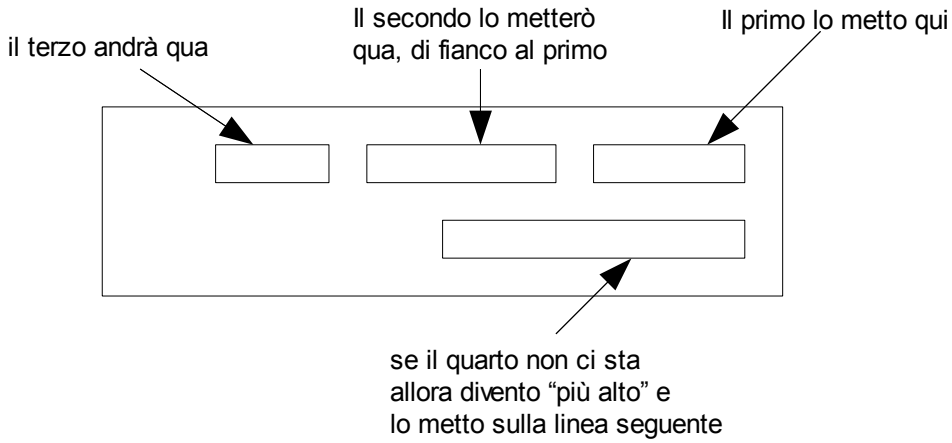
e inserisco questo contenitore nella parte bassa della finestra:

```
window.add(contenitoreBasso, BorderLayout.SOUTH);
```

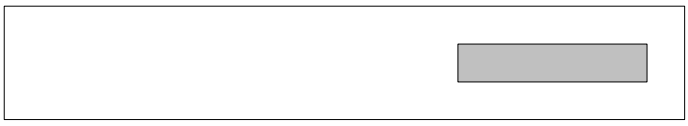
Esaminiamo cosa capita da un punto di vista grafico. Questo è contenitoreAlto quando lo creiamo:



Quando diciamo a contenitoreAlto "setLayout(new FlowLayout(FlowLayout.RIGHT))" idealmente il contenitore si prepara a ospitare dei componenti in questo modo:

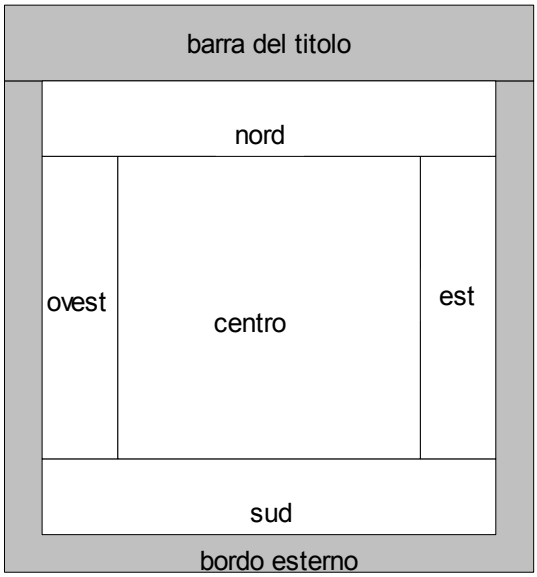


Quindi quando aggiungiamo al contenitore un pulsante quel pulsante in quel contenitore andrà a finire qui:

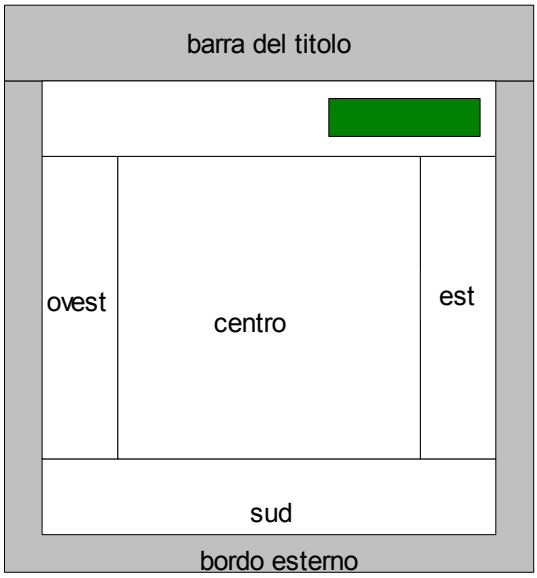


La larghezza e l'altezza del pulsante sono determinate durante l'avvio in base all'etichetta del pulsante, alle sue decorazioni – bordo, eventuali icone eccetera – e al tipo di carattere che sarà usato per disegnare il testo. Quando vado a inserire il pannello nella finestra con `window.add(contenitoreAlto, BorderLayout.NORTH)` sfruttiamo il fatto che una finestra Swing per impostazione predefinita ha un `LayoutManager` – `java.awt.BorderLayout` – che divide la parte

compresa tra la barra del titolo ed il bordo in cinque regioni, nord, sud, est, ovest e centro. In pratica una finestra vergine presenta questa suddivisione dello spazio:



Quando aggiungiamo il pannello in alto nella parte “nord” della finestra per effetto del posizionamento relativo del pulsante rispetto al suo contenitore risulta che il pulsante stesso apparirà nella parte nord della finestra, spostato verso destra.



Ci occuperemo poi dei diversi `LayoutManager` disponibili ma il principio del posizionamento dei componenti in Swing è questo.

Finestre.

Ce ne sono tre: `JWindow`, `JFrame`, `JDialog`. Delle tre se ne usano due: `JFrame` e `JDialog`. La differenza tra `JFrame` e `JDialog` è che un `JFrame` è autonomo, un `JDialog` esiste in rapporto con un'altra finestra. Il rapporto che un `JDialog` ha con un'altra finestra può essere sfruttato per stabilire un predominio della finestra di dialogo rispetto alla finestra da cui dipende. Si tratta della classica finestra che impedisce all'utente di interagire con un'altra finestra finché la prima è visibile. Partiamo da `JFrame`.

JFrame

Tra i costruttori di `JFrame` i più interessanti sono i due che accettano un `GraphicsConfiguration`: l'uso di `GraphicsConfiguration` consente, tra l'altro, di decidere quale schermo ospiterà la finestra nel caso in cui il sistema ne offra più d'uno – reale o virtuale che sia. Salva questa particolarità, il resto è quasi scontato.

```
JFrame frame = new JFrame();//crea una finestra non visibile, non visualizzabile
JFrame frame = new JFrame("titolo");//crea una finestra impostando nel contempo il suo titolo
```

Il metodo `setDefaultCloseOperation` di `JFrame` consente di scegliere uno dei comportamenti predefiniti che la finestra eseguirà quando l'utente premerà il pulsante di chiusura sulla barra del titolo o nel menù di controllo della finestra – a cui si accede attraverso la barra delle applicazioni o attraverso l'icona della finestra. Il metodo in questione richiede come argomento una fra le quattro costanti:

`WindowCostants.DISPOSE_ON_CLOSE`, equivale all'invocazione di `dispose()` ciò che comporta il rilascio delle risorse di basso livello allocate per la gestione della finestra e, in conseguenza, l'irrilevanza del frame ai fini della persistenza dell'EDT;

`WindowCostants.HIDE_ON_CLOSE`, equivale ad un `setVisible(false)`, la finestra scompare dal desktop ma resta visualizzabile;

`WindowCostants.EXIT_ON_CLOSE`, equivale ad un `System.exit(0)`, la JVM termina al volo portandosi dietro tutto il programma, senza possibilità di recupero salvo l'eventuale truccone dei permessi di sicurezza;

`WindowCostants.DO_NOTHING_ON_CLOSE`, scavalca il meccanismo dei comportamenti predefiniti e si usa in tutti i casi in cui al tentativo di chiusura della finestra si voglia ricondurre un comportamento che abbia un minimo di complessità – ad esempio una finestra di conferma "vuoi uscire?".

Dei quattro il meno felice è `EXIT_ON_CLOSE` perché ammazzare la JVM senza uno né due potrebbe non essere una grande idea a meno che non si stia scrivendo giusto un'applicazione d'esempio. Quando vogliamo attivare una procedura in conseguenza della richiesta di chiusura della finestra da parte dell'utente attraverso l'interfaccia grafica – pulsante di chiusura nella barra del titolo o nel menu di controllo che il gestore delle finestre di sistema crea per ogni finestra – impostiamo `DO_NOTHING_ON_CLOSE` come comportamento predefinito in chiusura e aggiungiamo un ascoltatore di eventi WindowListener al frame. Ci sono due versioni di `WindowListener`: uno è l'interfaccia omonima e uno è la classe `WindowAdapter`. La ragione di `WindowAdapter` e degli XYZAdapter in genere è evitare che il programmatore debba fornire un'implementazione vuota per ogni metodo dell'interfaccia listener salvo l'unico che potrebbe interessargli. `WindowListener` ha cinque metodi, magari a noi interessa solo il metodo `windowOpened` ma essendo `WindowListener` un'interfaccia occorre scrivere anche gli altri quattro metodi, lasciandoli vuoti. Per farla più breve esiste `WindowAdapter` che concretizza `WindowListener` dando una definizione "vuota" ai metodi dichiarati nell'interfaccia ciò che permette al programmatore di occuparsi della sovrascrittura del solo metodo che gli interessa. Vale per tutti gli "Adapter" – `MouseAdapter`, `MouseInputAdapter`, `KeyAdapter` eccetera. Dicevamo di voler far qualcosa in seguito alla richiesta di chiusura della finestra.

```
JFrame window = new JFrame("Prova Chiusura");
window.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

```
window.addWindowListener(new WindowAdapter() {

	@Override public void windowClosing(WindowEvent e) {
		//e qui facciamo qualcosa
	}
});
```

Scegliamo `windowClosing` perché la documentazione ci dice che è questo il metodo che riceve un evento prodotto dalla finestra nel momento in cui l'utente richiede la chiusura e a noi interessa proprio quello. L'utente richiede la chiusura: la finestra al momento è ancora aperta e gaudente. Ora che siamo in grado di sapere quando è richiesta la chiusura – ogni volta che il framework causa invocazione del metodo `windowClosing` – reagiamo chiedendo una conferma, giusto per far qualcosa.

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

public class Main {

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		JFrame window = new JFrame("Prova Chiusura");
		window.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
		window.addWindowListener(new WindowAdapter() {

			@Override public void windowClosing(WindowEvent e) {
				int answer = JOptionPane.showConfirmDialog(
					e.getWindow(),
					"Vuoi veramente chiudere la finestra?",
					"Conferma chiusura",
					JOptionPane.YES_NO_OPTION);
				if(answer == JOptionPane.YES_OPTION) {
					e.getWindow().dispose();
				}
			}
		});
		window.setSize(400, 400);
		window.setVisible(true);
	}
}
```

Altro metodo di un certo interesse in `WindowListener` è `windowOpened`. Il metodo `windowOpened` di un `WindowListener` registrato presso un `JFrame` viene invocato dal framework alla prima visualizzazione della finestra. Può essere interessante nel momento in cui il nostro programma voglia innescare un qualche procedimento solo dopo l'avvenuta proiezione della finestra di fronte all'utente. Il caso tipico è quello degli splashscreen. L'applicazione si avvia, sullo schermo appare un'interfaccia non interattiva che attraverso delle animazioni suggerisce all'utente l'idea che il programma impiega degli anni per attivarsi perché sta facendo qualcosa di essenziale per una corretta inizializzazione. L'idea dello splashscreen è quella di far vedere all'utente qualcosa mentre il programma compie delle operazioni che non hanno nulla da far vadere ma sono necessarie. E' intrattenimento puro. La sequenza di attività di un programma che faccia uso di uno splashscreen è la seguente:

1. l'utente avvia il programma

2. il programma inizializza la finestra dello splashscreen
3. il programma proietta lo splashscreen sullo schermo
4. appena lo splashscreen diventa visibile iniziano le operazioni di inizializzazione
5. quando l'inizializzazione termina lo splashscreen viene rimosso dallo schermo
6. quando lo splashscreen è scomparso dallo schermo viene aperta la finestra principale del programma.

Gli eventi di "stato" di un `JFrame` sono alla base del rispetto di questa sequenza. Vediamo il come attraverso un breve programma.

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

public class Main {

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {

		JWindow splashScreen = new JWindow();
		splashScreen.addWindowListener(new WindowAdapter() {
			private boolean closed = false;

			public void windowOpened(WindowEvent e) {
				startBackgroundInitialization(e.getWindow());
			}

			public void windowClosed(WindowEvent e) {
				if(closed == false) {
					closed = true;
					showMainFrame();
				}
			}
		});
		JLabel label = new JLabel("SplashScreen");
		splashScreen.setLayout(new GridBagLayout());
		splashScreen.add(label, new GridBagConstraints());
		splashScreen.setSize(300, 300);
		splashScreen.setLocationRelativeTo(null);
		splashScreen.setVisible(true);
	}

	private static void startBackgroundInitialization(final Window splashScreen) {
		new Thread() {

			@Override public void run() {
				try {
					Thread.sleep(5000);//simula qualcosa da fare...
				} catch(InterruptedException ignore) {
					ignore.printStackTrace();
				} finally {
					disposeWindow(splashScreen);
				}
			}
		}.start();
	}

	private static void showMainFrame() {
		JFrame window = new JFrame("Main Frame");
```

```
            window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
            window.setSize(400, 300);
            window.setLocationRelativeTo(null);
            window.setVisible(true);
      }

      private static void disposeWindow(final Window window) {
            EventQueue.invokeLater(new Runnable() {

                  public void run() {
                        window.dispose();
                  }
            });
      }
}
```

Per il poco appagante splashscreen usiamo un `JWindow` al posto di `JFrame`: è come `JFrame` ma senza titolo, bordo e traccia sulla barra delle applicazioni. Per inserire componenti all'interno della finestra si usano i metodi `add` e `setLayout`. Abbiamo visto come un contenitore gestisca l'inserimento e la proiezione di contenuti attraverso un layout manager. `JFrame` non fa eccezione. Le finestre Swing dispongono inoltre della classica barra dei menu. La barra dei menu è un oggetto `JMenuBar` e, una volta creata, si applica alla finestra tramite il metodo `setJMenuBar`. L'applicazione che segue crea una finestra con una barra dei menu, una barra dei comandi, un'area di testo e un pannello di stato.

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

public class Main {

      public static void main(String[] args) {
            EventQueue.invokeLater(new Runnable() {

                  public void run() {
                        start();
                  }
            });
      }

      private static void start() {
            JMenuBar menuBar = new JMenuBar();
            JMenu fileMenu = new JMenu("File");
            JMenuItem openItem = new JMenuItem("Apri");
            JMenuItem saveItem = new JMenuItem("Salva");
            fileMenu.add(openItem);
            fileMenu.add(saveItem);
            menuBar.add(fileMenu);

            JToolBar toolbar = new JToolBar();
            toolbar.setFloatable(false);
            toolbar.add(new JButton("Apri"));
            toolbar.add(new JButton("Salva"));

            JTextArea textArea = new JTextArea(20, 40);
            JScrollPane textAreaContainer = new JScrollPane(textArea);

            JLabel stateLabel = new JLabel("Status: editing", JLabel.TRAILING);
            JPanel statePanel = new JPanel(new GridLayout(1, 1));
            statePanel.setBorder(BorderFactory.createLoweredBevelBorder());
            statePanel.add(stateLabel);

            JFrame window = new JFrame("Test");
            window.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
            window.addWindowListener(new WindowAdapter() {

                  @Override public void windowClosing(WindowEvent e) {
                        closeApplication(e.getWindow());
                  }
```

```
            });
            window.setJMenuBar(menuBar);
            window.setLayout(new BorderLayout());
            window.add(toolbar, BorderLayout.NORTH);
            window.add(textAreaContainer, BorderLayout.CENTER);
            window.add(statePanel, BorderLayout.SOUTH);
            window.pack();
            window.setVisible(true);
      }

      private static void closeApplication(Window mainWindow) {
            mainWindow.dispose();
      }
}
```

Il metodo `pack()` fa sì che la finestra assuma le dimensioni minime necessarie e sufficienti affinchè ciò che contiene sia visualizzato secondo le sue dimensioni ottimali. Ha un effetto collaterale utile che è quello di realizzare la finestra, cioè attribuisce alla finestra e, indirettamente, al suo contenuto un contesto grafico. L'attribuzione di un contesto grafico segna il momento a partire dal quale un componente può eseguire alcune computazioni tra cui il calcolo della dimensione relativa ad un font, la produzione di un'immagine compatibile con il formato di proiezione del display, la creazione di un `BufferStrategy` per la gestione manuale del disegno.

JDialog

`JDialog` rappresenta una finestra collegata ad un'altra. Il collegamento si traduce nella possibilità della finestra di dialogo di bloccare l'interazione utente con una o più finestre – e quindi col contenuto di quelle finestre. Fatta eccezione per questa caratteristica il comportamento di `JDialog` è identico a quello di `JFrame`. L'impostazione del layout avviene attraverso l'invocazione del metodo `setLayout(LayoutManager)`, i componenti si aggiungono tramite uno dei diversi metodi `add(Component)`, la finestra risponde ai comandi `setVisible(true/false)` e libera le risorse di basso livello associate in seguito all'invocazione di `dispose()`. Java 6 ha introdotto delle costanti ad hoc per il controllo del tipo di “modalità” di una finestra di dialogo. Esistono quattro costruttori che accettano un argomento di tipo `Dialog.ModalityType`, un enumerativo che dispone di quattro valori costanti:

`APPLICATION_MODAL`: la finestra di dialogo impedisce l'interazione con ogni altra finestra dello stesso programma Java;

`DOCUMENT_MODAL`: la finestra di dialogo impedisce l'interazione con ogni altra finestra che abbia come “radice” la stessa radice di questa `JDialog`. In pratica le finestre possono formare degli alberi: creo un JFrame iniziale, poi creo una finestra di dialogo che usa quel `JFrame` come “owner”, aggiungo una seconda finestra di dialogo che usa la precedente come proprio “owner”, risulta che entrambe le finestre di dialogo dipendono dallo stesso `JFrame`, il primo. Il `JFrame` iniziale costituisce il documento rispetto al quale opera la modalità “document”;

`MODELESS`: la finestra di dialogo non impedisce l'interazione con altre finestre;

`TOOLKIT_MODAL`: la finestra di dialogo impedisce l'interazione con tutte le finestre che condividono lo stesso Toolkit. Un `Toolkit` è un oggetto che offre dei servizi necessari alla creazione dei componenti Swing. Se diverse applicazioni Java condividono lo stesso toolkit allora la modalità toolkit interviene rispetto a tutte le finestre proiettate in quell'applicazione. La modalità in questione ha una sua utilità nel caso di applicazioni rappresentate da Applet Java. Si può pensare ad una pagina web che incorpora più di un'applet. Meccanicamente ogni applet rappresenta una diversa applicazione Java ma è possibile che esse rappresentino parti diverse di un unico programma, da cui la possibilità

che sia necessario sovrapporne le diverse finestre. Dal punto di vista del programma la modalità di una finestra si riflette in un'interruzione del flusso di controllo condizionata alla presenza della finestra di dialogo sullo schermo.

```
1: ...dialog = un JDialog modale
2: dialog.setVisible(true);
3: System.out.println("hello");
```

Qui la riga 3 è eseguita solo dopo la scomparsa di dialog dallo schermo, in conseguenza dell'interazione utente o per effetto di un'invocazione programmatica. Ci sono molti modi per gestire una finestra di dialogo. Normalmente le finestre di dialogo si usano o per applicare delle impostazioni ad una parte del programma principale o per richiedere all'utente un input strutturato. In entrambi i casi la gestione della finestra di dialogo può essere fatta creando una classe ad hoc che dichiara non la finestra di dialog ma il contenuto della stessa come proprio campo e attraverso un metodo pubblico permette ad altre parti del programma di visualizzare una finestra di dialogo creata "al volo". Ad esempio:

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

public class MyInputPane implements ActionListener{
     private JPanel contentPane = new JPanel(new BorderLayout());
     private JDialog dialog;

     public MyInputPane() {
          JButton confirm = new JButton("ok");
          confirm.addActionListener(this);
          JPanel buttons = new JPanel(new FlowLayout(FlowLayout.RIGHT));
          buttons.add(confirm);
          contentPane.add(buttons, BorderLayout.SOUTH);
          contentPane.setPreferredSize(new Dimension(300, 300));
     }

     public void actionPerformed(ActionEvent e) {
          dialog.dispose();
     }

     public void show(Window owner) {
          dialog = new JDialog(owner, "Test", Dialog.ModalityType.DOCUMENT_MODAL);
          dialog.setContentPane(contentPane);
          dialog.pack();
          dialog.setVisible(true);
     }
}
```

Qui la finestra di dialogo contiene solo un pulsante, confirm, alla cui pressione corrisponde la distruzione della finestra di dialogo. Per usare una classe del genere si crea un'istanza e si passa al suo metodo show un riferimento alla finestra che dovrà essere bloccata in attesa dell'input utente.

```
MyInputPane ip = new MyInputPane();
ip.show(una finestra);
```

Nel caso frequente in cui la finestra di dialogo sia presentata per la creazione interattiva di un dato il metodo show sarà adattato al fine di restituire il dato costruito.

JOptionPane

A rigore `JOptionPane` non è una finestra ma un componente Swing. La classe `JOptionPane` tuttavia offre un certo numero di metodi statici i quali generano finestre di dialogo. Questi metodi

producono le classi finestrelle di messaggio – errore, notifica, input – ma supportano anche la presentazione di finestre di dialogo più complesse. I metodi statici showABCDialog di `JOptionPane` causano interruzione del flusso di controllo: si apre la finestra di dialogo e finché l'utente non la chiude, direttamente o per interazione con uno dei pulsanti in essa contenuti, l'esecuzione si ferma nel punto del codice in cui appare l'invocazione del metodo show. I diversi metodi showABCDialog della classe `JOptionPane` sono ampiamente trattati nella documentazione standard. Due punti meritano attenzione: il primo è che i metodi showInternal... operano rispetto a finestre contenute in un `JDesktopPane`, il secondo è che quando il parametro "message" è un componente Swing quel componente è inserito all'interno della finestra di dialogo prodotta da `JOptionPane`. Il fatto che il messaggio possa essere un componente Swing permette di creare un contenuto più complesso di una mera stringa di testo. Una possibilità è, ad esempio, quella di presentare un testo su più linee usando un componente `JTextArea`. Il programma che segue genera un'eccezione Java e ne riproduce la traccia in un `JOptionPane`.

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

public class Main {

	public static void main(String[] args) {

		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		final JButton button = new JButton("Eccezione");
		button.addActionListener(new ActionListener() {

			public void actionPerformed(ActionEvent e) {
				try {
					throw new Exception("hello world!");
				} catch(Exception ex) {
					showStack(button, ex);
				}
			}
		});
		JFrame window = new JFrame("JOptionPane test");
		window.add(button);
		window.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
		window.pack();
		window.setVisible(true);
	}

	private static void showStack(Component parent, Throwable t) {
		StringWriter buffer = new StringWriter();
		PrintWriter out = new PrintWriter(buffer);
		t.printStackTrace(out);
		JTextArea textArea = new JTextArea();
		textArea.setText(buffer.toString());
		textArea.setEditable(false);
		textArea.setCaretPosition(0);
		JScrollPane scroller = new JScrollPane(textArea);
		scroller.setPreferredSize(new Dimension(500, 300));
		JOptionPane.showMessageDialog(parent, scroller, "Eccezione",
			JOptionPane.ERROR_MESSAGE);
	}
}
```

Per generare la finestra di dialogo `JOptionPane` richiede un `Component` come primo argomento dei suoi metodi show. La finestra principale da cui dipenderà la finestra di dialogo è la finestra che contiene il componente passato come argomento o l'argomento stesso se risulti essere una finestra. Salva questa particolarità `JOptionPane` offre utilità standard per richiedere all'utente le conferme più variopinte. La tipica finestra di conferma è prodotta dal metodo `showConfirmDialog`.

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

public class Main {

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		JFrame window = new JFrame("Test");
		window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
		window.setSize(400, 400);
		window.setLocationRelativeTo(null);
		window.setVisible(true);

		int risposta = JOptionPane.showConfirmDialog(window,
			"Chiudere il programma?", "Conferma", JOptionPane.YES_NO_OPTION);
		if(risposta == JOptionPane.YES_OPTION) {
			window.dispose();
		}
	}
}
```

Nel codice `JOptionPane.YES_NO_OPTION` rappresenta il tipo di opzioni disponibili all'utente e rappresentate da pulsanti all'interno della finestra di conferma. Quando l'utente preme uno dei pulsanti JOptionPane traduce l'evento in una costante e la restituisce. Se le opzioni sono `YES_NO_OPTION` allora il valore restituito potrà essere `JOptionPane.YES_OPTION`, `JOptionPane.NO_OPTION` o `JOptionPane.CLOSED_OPTION`. La risposta sarà yes o no se l'utente premerà il pulsante yes o no e sarà closed se l'utente chiuderà la finestra di dialogo senza scegliere nulla. Analogamente funzionano le opzioni `OK_CANCEL_OPTION` e `YES_NO_CANCEL_OPTION`.

JFileChooser

Come `JOptionPane`, anche `JFileChooser` è un componente Swing la cui classe offre metodi statici che producono finestre di dialogo. Come suggerito dal nome, `JFileChooser` consente all'utente la selezione di file o cartelle tanto in lettura – tramite `showOpenDialog` – quanto in scrittura – tramite `showSaveDialog`. L'uso è banale:

```
JFileChooser fileChooser = new JFileChooser();
fileChooser.setDialogTitle("Scegli un file");
int selezione = fileChooser.showOpenDialog(unComponent);
```

Il valore di selezione può essere `JFileChooser.APPROVE_OPTION` o `JFileChoser.CANCEL_OPTION`. Nel primo caso significa che l'utente ha confermato un qualche genere di input. Nel secondo caso ha abbandonato nel tentativo di. Per recuperare il file selezionato si invoca successivamente il metodo getSelectedFile.

```
if(selezione == JFileChooser.APPROVE_OPTION) {
	File selectedFile = fileChooser.getSelectedFile();
```

```
}
```

Occorre prestare attenzione al fatto che l'avvenuta selezione di un `File` non comporta necessariamente l'esistenza del file scelto o la validità stessa del percorso che esso rappresenta. `JFileChooser` consente la selezione multipla di file. L'opzione è attivata con l'invocazione del metodo `setMultiSelectionEnabled(true)`. In questo caso i file selezionati dall'utente dovranno essere recuperati dopo l'eventuale accettazione tramite il metodo `getSelectedFiles()`.
E' possibile applicare un filtro personalizzato ai file mostrati da un JFileChooser. Il filtro è rappresentato da un'istanza di javax.swing.filechooser.FileFilter. Il codice che segue mostra un esemprio di filtro per file con estensione “txt”:

```
import javax.swing.filechooser.FileFilter;
import java.io.File;

public class TxtFilter implements FileFilter {

      public boolean accept(File file) {
            return f.isDirectory() || file.getName().toLowerCase().endsWith(“.txt”);
      }

      public String getDescription() {
            return “txt file”;
      }
}
```

Dato un `JFileChooser`, l'applicazione di `TxtFilter` avviene come segue:

```
filechooser.addChoosableFileFiler(new TxtFilter());
```

I filtri aggiunti ad un `JFileChooser` vengono inseriti in una lista e proposti all'utente attraverso l'interfaccia utente. `JFileChooser` supporta l'inserimento di un componente Swing aggiuntivo nell'interfaccia – l'esempio tipico è quello di un pannello che mostra un'anteprima del file attualmente selezionato. Per aggiungere un componente si usa il metodo `setAccessory(JComponent)`. Per adattare il comportamento del componente accessorio agli eventi generati dall'interazione dell'utente con il `JFileChooser` si usa un `PropertyChangeListener`. Ad esempio quando la cartella corrente cambia il `JFileChooser` che subisce questa mutazione notifica a tutti gli ascoltatori `PropertyChangeListener` un evento `PropertyChangeEvent` avente come attributo `propertyName` il valore `JFileChooser.DIRECTORY_CHANGED_PROPERTY`.

Posizione e dimensione dei componenti:

LayoutManager

In Swing la posizione dei componenti in un contenitore è automaticamente determinata in base al tipo di layout usato dal contenitore. Il layout può influire anche sulla dimensione – altezza e larghezza – di un componente. Un componente Swing ha quattro dimensioni: la dimensione attuale – size – la dimensione preferita – preferredSize – la dimensione minima – minimumSize – e la dimensione massima – maximumSize. La dimensione attuale è quella che il componente ha in un certo istante e il suo valore può dipendere dalle altre tre dimensioni. La dimensione preferita è l'altezza e larghezza ideale affinchè il componente possa proiettare il suo contenuto, con tutte le decorazioni i frizzi e i lazzi. La dimensione minima è quella che permette al componente di avere un significato visivamente determinabile pur non essendo pienamente proiettato sullo schermo. La dimension massima è la massima estensione che il componente può assumere. Le informazioni contenute nella dimensione minima, preferita e massima possono essere usate dal layout per

stabilire quanta parte di un contenitore deva essere occupata dallo specifico componente considerando che il contenitore stesso è soggetto ad un ridimensionamento. In pratica la dimensione attuale può essere calcolata usando le altre tre in rapporto alla superficie totale disponibile nel contenitore. La partecipazione delle tre dimensioni al compute della quarta è solo possibile: dipende dal tipo di layout. Ad esempio un `LayoutManager` di tipo `GridLayout` non considera alcuna delle tre dimensioni minima, preferita o massima e si limita ad assegnare al componente una dimensione attuale pari ad una frazione delle dimensioni del contenitore. Un `LayoutManager` di tipo `BorderLayout` usa l'altezza preferita e minima per determinare l'altezza del componente collocato in posizione NORTH o SOUTH, la larghezza preferita e minima per determinare la larghezza del componente in posizione EAST o WEST, le dimensioni preferite del componente al centro per determinare la larghezza del riquadro CENTER a meno che il contenitore stesso non sia troppo grande o troppo piccolo – nel qual caso il componente centrale si vede attribuire una dimensione pari al residuo derivante dalla proiezione dei componenti sulla cornice. Un `LayoutManager` di tipo `FlowLayout` rispetta la dimensioni preferita dei componenti. E così via.

Come funzionano

Per dare un'idea di come faccia un `LayoutManager` a fare...quel che fa creiamo un nuovo `LayoutManager`. Il nostro layout divide il contenitore in linee orizzontali ognuna delle quali contiene uno dei componenti inseriti nel contenitore. Ogni linea ha un'altezza pari all'altezza preferita del componente che contiene.

```
import java.awt.*;

public class RowLayout implements LayoutManager {

      public void removeLayoutComponent(Component c) {}
      public void addLayoutComponent(String name, Component comp) {}
      public Dimension minimumLayoutSize(Container parent) {
            return preferredLayoutSize(parent);
      }

      public void layoutContainer(Container parent) {
            synchronized(parent.getTreeLock()) {
                  Dimension containerSize = parent.getSize();
                  Insets margin = parent.getInsets();
                  int y = margin.top;
                  int x = margin.left;
                  int width = containerSize.width - margin.left - margin.right;
                  int maxY = containerSize.height - margin.bottom;

                  for(Component child : parent.getComponents()) {
                        Dimension childPreferredSize = child.getPreferredSize();
                        int height = childPreferredSize.height;
                        if(y + height <= maxY) {
                              Rectangle childArea = new Rectangle(x,y,width,height);
                              child.setBounds(childArea);
                              child.setVisible(true);
                              y += height;
                        } else {
                              child.setVisible(false);
                        }
                  }
            }
      }

      public Dimension preferredLayoutSize(Container parent) {
            synchronized(parent.getTreeLock()) {
                  Dimension size = new Dimension();
                  for(Component c : parent.getComponents()) {
                        Dimension preferredSize = c.getPreferredSize();
```

```
                    size.width = Math.max(size.width, preferredSize.width);
                    size.height += preferredSize.height;
                }
                Insets margin = parent.getInsets();
                size.width += margin.left + margin.right;
                size.height += margin.top + margin.bottom;
                return size;
            }
        }
}
```

Ogni volta che il framework stabilisce che la disposizione dei componenti in un contenitore deva essere rigenerata – ciò che accade ad esempio quando il contenitore cambia le proprie dimensioni – è invocato il metodo `layoutContainer` del `LayoutManager` associato a quel contenitore. Quando un contenitore non ha una dimensione preferita propria la sua dimensione preferita è calcolata dal `LayoutManager` che, come nel nostro caso, restituisce un valore in tutto o in parte determinato da ciò che il contenitore contiene. I metodi senza corpo nel layout d'esempio sono usati dai `LayoutManager` più raffinati per collegare la disposizione dei componenti ad eventi di inserimento o rimozione di componenti e a particolari istruzioni di posizionamento – margini, allineamenti eccetera.

FlowLayout

Il layout `java.awt.FlowLayout` dispone i componenti di un contenitore per linee orizzontali, affiancando il precedente al successivo, in ordine di inserimento. Ogni componente assume la propria dimensione preferita. Quando la riga corrente, alta quanto il più alto dei componenti sulla riga, non ha più spazio orizzontale disponibile, il layout va a capo. I componenti possono essere allineanti al bordo sinistro o destro del contenitore oppure possono essere centrati nella linea che occupano. I concetti di destro o sinistro sono astratti nelle costanti `LEADING` e `TRAILING` che corrispondono all'inizio della riga o alla fine della stessa secondo la direzione del testo nelle impostazioni di località correnti. `FlowLayout` è particolarmente indicato per costruire barre di pulsanti o di controlli più complessi posto che questi abbiano le stesse dimensioni – altrimenti la linea di componenti risultante assume una forma a montagna russa che non sta particolarmente bene, piuttostochè pannelli in cui accumulare una certa quantità di piccoli oggetti di cui non è noto al momento della compilazione il numero – ad esempio un pannello che mostri delle anteprime di immagini.

GridLayout

Il layout `GridLayout` divide la superficie del contenitore in celle di uguali dimensioni. I costruttori di `GridLayout` richiedono il numero di righe e il numero di colonne della griglia da generare. I componenti vengono associati ad ogni cella in ordine di inserimento nel contenitore, partendo dalla prima cella in alto a sinistra o in alto a destra secondo l'orientamento del contenitore – che dipende dalle impostazioni di localizzazione. Un `GridLayout` così costruito:

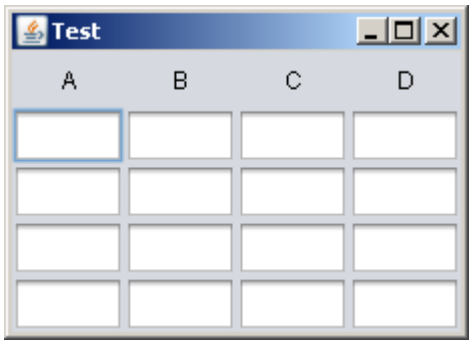
```
GridLayout grid = new GridLayout(3, 5);
```

genera una griglia di tre righe ognuna delle quali contiene cinque celle. Ogni cella ha la stessa altezza e la stessa larghezza quindi se inseriamo un componente largo 50 pixel e alto 100 e quel componente è il più grande tra quelli contenuti allora ogni cella sarà larga 50 pixel e alta 100. Quando il contenitore viene ridimensionato lo spazio eccedente viene equamente diviso tra tutte le celle in modo tale che sia rispettata l'identità delle loro dimensioni. A conti fatti questo comportamento limita l'utilità di `GridLayout` alla creazione di tabelle regolari di componenti. Per

dare un'idea, il codice che segue usa un `GridLayout`:

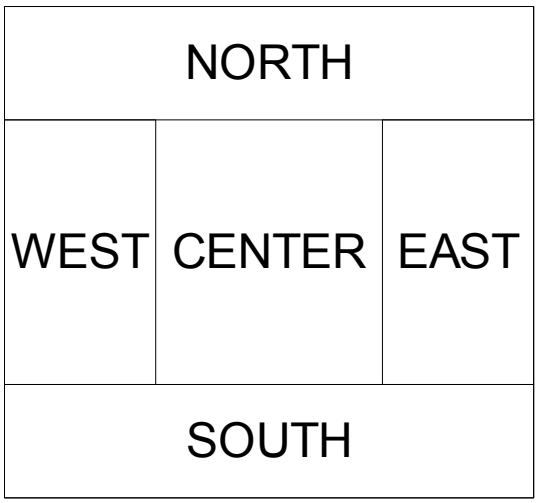
```
import java.awt.*;
import javax.swing.*;

public class Main {

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		JPanel grid = new JPanel(new GridLayout(5, 4));
		grid.add(new JLabel("A", JLabel.CENTER));
		grid.add(new JLabel("B", JLabel.CENTER));
		grid.add(new JLabel("C", JLabel.CENTER));
		grid.add(new JLabel("D", JLabel.CENTER));
		for(int i = 0; i < 16; i++) {
			grid.add(new JTextField(4));
		}

		JFrame window = new JFrame("Test");
		window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
		window.add(grid);
		window.pack();
		window.setLocationRelativeTo(null);
		window.setVisible(true);
	}
}
```

per produrre questa interfaccia:



BorderLayout

`BorderLayout` divide il contenitore in cinque regioni tipicamente idonee a rappresentare la suddivisione di una finestra a contenuto statico: due regioni ad altezza fissa, una in alto e una in basso, due regioni a larghezza fissa, a sinistra e a destra, una regione centrale a dimensioni libere.



Dato un contenitore con layout BorderLayout:

```
JPanel panel = new JPanel(new BorderLayout());
```

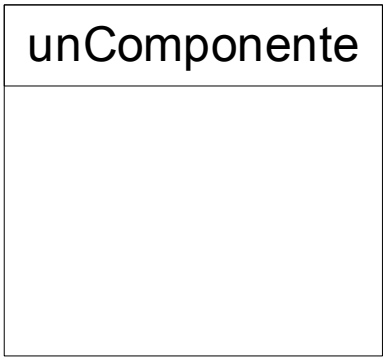
per assegnare un componente alle diverse regioni si possono usare cinque costanti definite in BorderLayout e il metodo `add(Component, Object)` del contenitore.

```
panel.add(unComponent, BorderLayout.NORTH);
panel.add(unComponent, BorderLayout.WEST);
panel.add(unComponent, BorderLayout.CENTER);
panel.add(unComponent, BorderLayout.SOUTH);
panel.add(unComponent, BorderLayout.EAST);
```

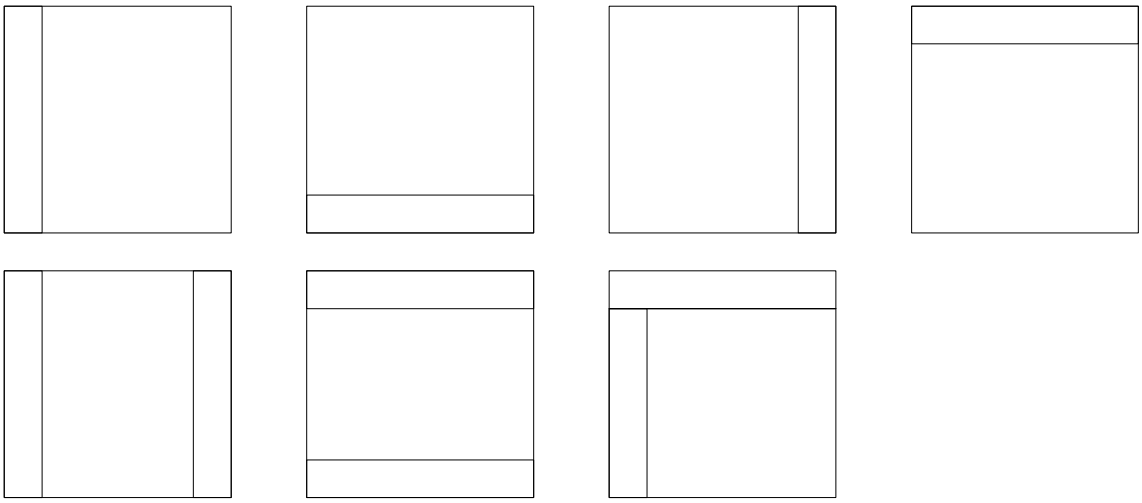
Le dimensioni delle regioni sono parzialmente indipendenti. Le regioni WEST ed EAST assumono la larghezza preferita dei rispettivi componenti associati. Le regioni NORTH e SOUTH assumono l'altezza preferita dei rispettivi componenti associati. La regione CENTER assume dimensioni pari allo spazio eccedente la somma delle altezze di NORTH e SOUTH e la somma delle larghezze di WEST e EAST. In pratica ridmensionando il contenitore il centro si allarga o si restringe. Da notare che il componente centrale non mantiene la propria dimensione preferita se lo spazio a disposizione è maggiore della sua altezza e larghezza preferita quindi BorderLayout non può essere usato per mantenere un elemento al centro dello spazio di un contenitore. Regioni prive di un componente associato partecipano alla distribuzione dello spazio con dimensioni preferite del contenuto pari a zero. E' diverso dal non partecipare affatto. Ad esempio usando la sola regione nord:

```
JPanel panel = new JPanel(new BorderLayout());
panel.add(unComponent, BorderLayout.NORTH);
```

il componente assumerà la sua altezza preferita aderendo al bordo superiore del contenitore.



Questo avviene perché comunque la regione CENTER tende ad occupare tutta la superficie disponibile. La figura che segue mostra alcune delle disposizioni ottenibile con `BorderLayout`.



CardLayout

`CardLayout` funziona come un proiettore di diapositive. I componenti del contenitore vengono proiettati uno alla volta con la possibilità di passare al componente precedente o successivo o saltare ad uno specifico componente. L'ordine di proiezione è determinato dall'ordine di inserimento dei componenti nel contenitore. `CardLayout` ha una sua utilità intrinseca nelle interfacce per procedure guidate, i c.d. wizard. L'applicazione che segue mostra un esempio di passaggio da un componente all'altro all'interno di uno stesso pannello attraverso `CardLayout`.

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

public class Main {

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		JButton next = new JButton("Successivo");
		JButton prev = new JButton("Precedente");

		JPanel first = new JPanel(new BorderLayout());
		JPanel second = new JPanel(new BorderLayout());
		JPanel third = new JPanel(new BorderLayout());

		Dimension cardSize = new Dimension(400, 300);

		first.add(new JLabel("Primo", JLabel.CENTER));
		second.add(new JLabel("Secondo", JLabel.CENTER));
		third.add(new JLabel("Terzo", JLabel.CENTER));

		first.setPreferredSize(cardSize);
		second.setPreferredSize(cardSize);
		third.setPreferredSize(cardSize);
```

```
            final JPanel screen = new JPanel();
            final CardLayout layout = new CardLayout();
            screen.setLayout(layout);

            screen.add(first, "first");
            screen.add(second, "second");
            screen.add(third, "third");

            next.addActionListener(new ActionListener() {

                  public void actionPerformed(ActionEvent e) {
                        layout.next(screen);
                  }
            });
            prev.addActionListener(new ActionListener() {

                  public void actionPerformed(ActionEvent e) {
                        layout.previous(screen);
                  }
            });

            JPanel buttonContainer = new JPanel(new FlowLayout(FlowLayout.RIGHT));
            buttonContainer.add(prev);
            buttonContainer.add(next);

            JPanel container = new JPanel(new BorderLayout());
            container.add(screen, BorderLayout.CENTER);
            container.add(buttonContainer, BorderLayout.SOUTH);

            JFrame window = new JFrame("Test");
            window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
            window.add(container);
            window.pack();
            window.setLocationRelativeTo(null);
            window.setVisible(true);
      }
}
```

Le stringhe usate per aggiungere i componenti al pannello nelle righe evidenziate possono essere usate per imporre al `CardLayout` la proiezione di uno specifico componente. Ad esempo un:

```
layout.show(screen, "second");
```

causerebbe la visualizzazione della `JLabel` con testo “secondo”.

OverlayLayout

Il layout `OverlayLayout` consente la sovrapposizione di più componenti in uno stesso contenitore. La sovrapposizione può essere usata per creare effetti una varietà di effetti, dai pannelli a scomparsa ai fumetti passando per gli sfondi. Il costruttore di `OverlayLayout` richiede come argomento il contenitore a cui il layout stesso sarà applicato dunque la creazione di tale layout avviene come segue:

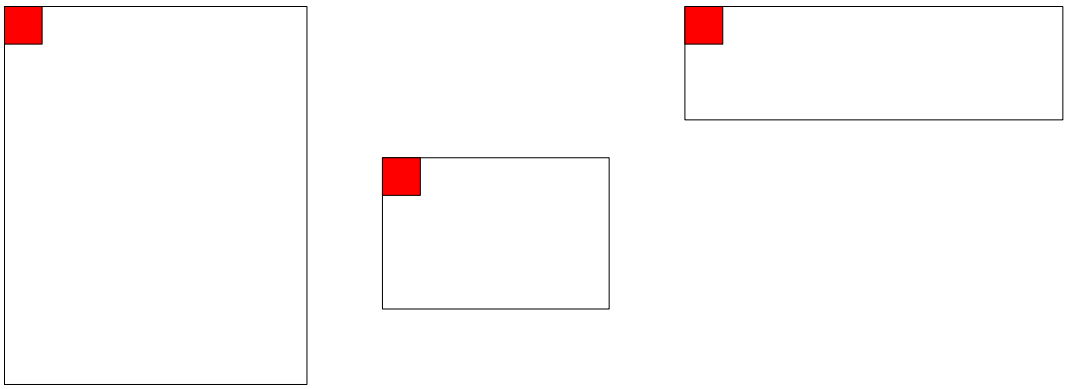
```
JPanel contenitore = new JPanel();
OverlayLayout layout = new OverlayLayout(contenitore);
contenitore.setLayout(layout);
```

L'ordine in cui i componenti sono inseriti nel contenitore determina l'ordine di sovrapposizione: il primo componente è sovrapposto al secondo che è sovrapposto al terzo e così via. Nel codice che segue:

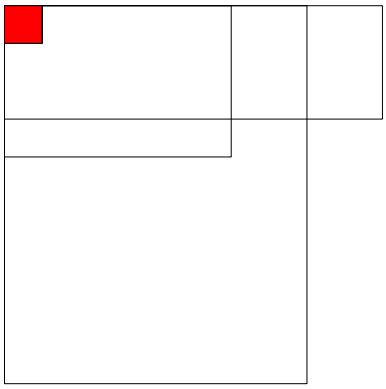
```
contenitore.add(componenteA);
```

```
contenitore.add(componenteB);
```

componenteA appare sopra a componenteB. `OverlayLayout` prende in considerazione il valore dell'allineamento dei singoli componenti (`alignmentX` e `alignmentY`) per calcolare la posizione di ciascuno di essi all'interno del contenitore. I valori `alignmentX` e `alignmentY`, assegnabili ad un `JComponent` tramite i metodi `setAlignmentX` e `setAlignmentY`, individuano un punto nel contenitore. Le coordinate del punto sono calcolate attraverso una proporzione tra la larghezza del contenitore e il valore di `alignmentX` e l'altezza del contenitore e il valore di `alignmentY`. `OverlayLayout` prende i punti di allineamento di ogni componente e fa in modo che si sovrappongano. La morale della favola è che se abbiamo un insieme di componenti con allineamento x = `Component.LEFT_ALIGNMENT` e y = `Component.TOP_ALIGNMENT`:



e li inseriamo in un contenitore con layout `OverlayLayout` essi appariranno come segue:



Le dimensioni complessive del contenitore sono pari alla larghezza e altezza preferite massime tra le altezze e larghezze preferite di tutti i componenti che contiene.

BoxLayout

`BoxLayout` permette la creazione di righe o colonne di componenti. Ogni riga o colonna contiene un componente. L'uso prevede la creazione del contenitore:

```
JPanel panel = new JPanel();
```

La creazione del layout:

```
BoxLayout layout = new BoxLayout(panel, BoxLayout.Y_AXIS); /o X_AXIS
```

L'assegnazione del layout al contenitore:

```
panel.setLayout(layout);
```

E' possibile usare la classe Box per ottenere un contenitore preconfigurato:

```
JComponent contenitore = Box.createVerticalBox();
```

oppure

```
JComponent contenitore = Box.createHorizontalBox();
```

Il programma che segue crea una finestra con tre pulsanti disposti lungo l'asse verticale:

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

public class Main {

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		JPanel panel = new JPanel();
		BoxLayout layout = new BoxLayout(panel, BoxLayout.Y_AXIS);
		panel.setLayout(layout);

		JButton one = new JButton("hello");
		JButton two = new JButton("world");
		JButton three = new JButton("!!!");

		panel.add(one);
		panel.add(two);
		panel.add(three);

		JFrame window = new JFrame("Form");
		window.add(panel);
		window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
		window.pack();
		window.setVisible(true);
	}
}
```

Il risultato è questo:

Come `OverlayLayout` anche `BoxLayout` dispone i componenti tenendo in considerazione il loro allineamento. Se nel codice precedente impostiamo l'allineamento orizzontale dei pulsanti su `Component.CENTER_ALIGNMENT`:

```
one.setAlignmentX(Component.CENTER_ALIGNMENT);
```

```
two.setAlignmentX(Component.CENTER_ALIGNMENT);
three.setAlignmentX(Component.CENTER_ALIGNMENT);
```

otteniamo come risultato:

BoxLayout rispetta le dimensioni minime, massime e preferite dei componenti. Se si vogliono espandere i componenti affinchè occupino tutto lo spazio disponibile è sufficiente impostare la loro dimensione massima ad un valore relativamente alto. Sempre prendendo il codice precedente scrivendo:

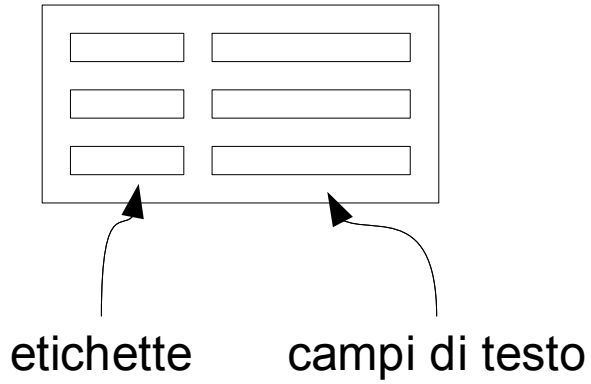
```
three.setMaximumSize(new Dimension(1000, 1000));
```

otteniamo:

Appoggiandosi alla classe `Box` è possibile introdurre dei distanziatori di dimensioni fisse – `createHorizontalStrut`, `createVerticalStrut`, `createRigidArea` – o variabili – `createHorizontalGlue`, `createVerticalGlue`. I distanziatori sono degli oggetti `Component` che devono essere inseriti nel contenitore. I distanziatori fissi occupano sempre lo stesso spazio quelli variabili occupano lo spazio restante dopo aver collocato gli altri componenti.

Combinare i layout

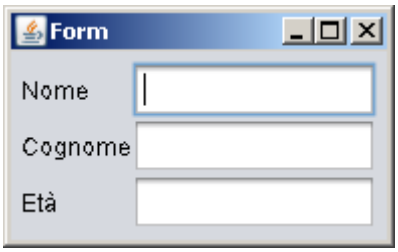
Combinando contenitori dotati di layout diversi è possibile creare una varietà di interfacce senza necessariamente appoggiarsi ai `LayoutManager` più generali, come `GridBagLayout`, o specificamente ideati per l'uso tramite programmi di progettazione assisistita, come `GroupLayout` o `SpringLayout`. Il primo passo è la definizione di una bozza della disposizione dei componenti che si vuole ottenere. Ad esempio:



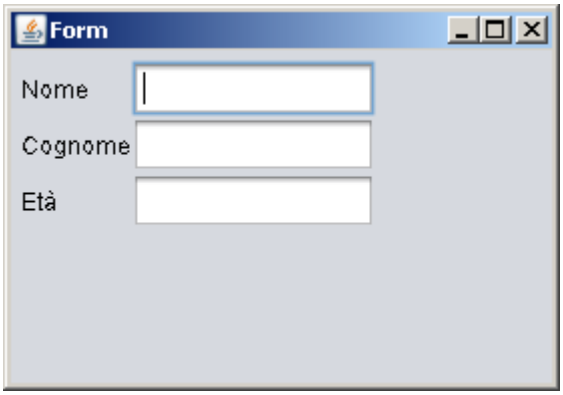
A questo occorre sfruttare le capacità particolari dei singoli layout manager. Si nota che l'interfaccia è divisa in due colonne di dimensione diversa. Ogni colonna è composta di tre righe e i componenti contenuti in queste righe hanno tutti uguale larghezza. Sappiamo che `GridLayout` può dividere lo spazio in righe e che i componenti contenuti in queste righe hanno tutti la stessa larghezza e altezza. Possiamo quindi immaginare di dividere il contenitore in due parti, un contenitore a destra e uno sinistra ognuno dei quali diviso in tre righe da un `GridLayout`. I due contenitori intermedi possono essere inseriti in un terzo contenitore che usi un `BoxLayout`. `BoxLayout` è in grado di affiancare due colonne mantenendo le loro diverse larghezze. Per evitare che il ridimensionamento scompensi l'aspetto generale dell'interfaccia inseriamo il tutto in un contenitore con `FlowLayout` – che è in grado di mantenere le dimensioni preferite del contenuto. Ciò che a parole sembra macchinoso risulta in effetti piuttosto breve:

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

public class Main {

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		JPanel labels = new JPanel(new GridLayout(3, 1));
		labels.add(new JLabel("Nome"));
		labels.add(new JLabel("Cognome"));
		labels.add(new JLabel("Età"));

		JPanel fields = new JPanel(new GridLayout(3, 1));
		fields.add(new JTextField(10));
		fields.add(new JTextField(10));
		fields.add(new JTextField(10));

		Box group = Box.createHorizontalBox();
		group.add(labels);
		group.add(fields);

		JPanel container = new JPanel(new FlowLayout(FlowLayout.LEFT));
		container.add(group);

		JFrame window = new JFrame("Form");
		window.add(container);
		window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
		window.pack();
		window.setVisible(true);
	}
}
```

Con un'eccellente corrispondenza visiva:



ed un comportamento stabile in caso di alterazione delle dimensioni.



Pulsanti, menu e azioni.

Swing offre l'armamentario standard nel campo dei controlli a una o due fasi: pulsanti – `JButton` – interruttori – `JToggleButton` – caselle di spunta – `JCheckBox` – selettori – `JRadioButton`. Accanto a questi troviamo i pulsanti per le barre dei menu (`JMenuBar`) ed i menu contestuali (`JPopupMenu`): `JMenuItem` , `JCheckBoxMenuItem`, `JRadioButtonMenuItem`. Tutti i controlli sono figli o nipoti di `AbstractButton` e condividono pertanto una parte significativa del loro modo d'essere.

ActionListener

Fatte salve le differenze funzionali, il comportamento di questi controlli è omogeneo e deriva da `AbstractButton`: tutti quanti notificano l'avvenuta reazione all'interazione utente attraverso un evento `ActionEvent`, intercettabile con un `ActionListener`, tutti accettano un mix di icone e stringhe di testo come contenuto visivo e possono essere raggruppati. Partiamo da `JButton`. Il primo punto è come catturare l'evento “qualcuno mi ha premuto”. Si può usare l'interfaccia `ActionListener`.

```
JButton pulsante = new JButton("Hello");
pulsante.addActionListener(new ActionListener() {

      public void actionPerformed(ActionEvent e) {
            System.out.println("world");
      }
});
```

Action

Un `Action` è un pacchetto di dati composto dall'operazione che deve essere eseguita quando un controllo che usa l'`Action` subisce l'interazione, dalle etichette del pulsante, dalle combinazioni di tasti che causano l'attivazione dell'azione e da tante altre belle cose. Il punto è che l'`Action` permette di condividere un comportamento e le informazioni utili a rappresentarlo tra più controlli senza dover ripetere ogni volta le stesse cose. L'esempio che segue mostra la condivisione di un'azione tra due controlli – un `JButton` e un `JMenuItem`.

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

public class Main {
```

```
	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		Action salva = new AbstractAction() {

			@Override public void actionPerformed(ActionEvent e) {
				System.out.println("Sto salvando");
			}
		};
		salva.putValue(Action.NAME, "Salva");

		JMenuBar menuBar = new JMenuBar();
		JMenu fileMenu = new JMenu("File");
		fileMenu.add(new JMenuItem(salva));
		menuBar.add(fileMenu);

		JToolBar toolbar = new JToolBar();
		toolbar.setFloatable(false);
		toolbar.add(new JButton(salva));

		JFrame window = new JFrame("Test");
		window.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
		window.addWindowListener(new WindowAdapter() {

			@Override public void windowClosing(WindowEvent e) {
				closeApplication(e.getWindow());
			}
		});
		window.setJMenuBar(menuBar);
		window.setLayout(new BorderLayout());
		window.add(toolbar, BorderLayout.NORTH);
		window.pack();
		window.setVisible(true);
	}

	private static void closeApplication(Window mainWindow) {
		mainWindow.dispose();
	}
}
```

In questo esempio l'azione è associata ai pulsanti attraverso il costruttore di questi ultimi. E' possibile associare un'azione successivamente alla costruzione usando il metodo `setAction`, condiviso da tutti i pulsanti Swing. La descrizione compiuta dei valori associabili ad un `Action` si trova nella javadoc. `JToggleButton`, `JRadioButton` e `JCheckBox` sono controlli a due fasi: premuto e rilasciato. I pulsanti citati rilasciano un evento ActionEvent ad ogni interazione, per verificare se il pulsante si trovi nello stato premuto o rilasciato occorre invocare il metodo `isSelected()` del pulsante stesso.

ButtonGroup

Per raggruppare più pulsanti si usa `ButtonGroup`. Il raggruppamento fa sì che uno solo tra i pulsanti appartenenti ad uno stesso gruppo possa essere selezionato in un certo istante. `ButtonGroup` funziona come una lista, dopo averlo creato di aggiungono i pulsanti da raggruppare usando il metodo add del gruppo. Il metodo `getSelection()` di `ButtonGroup` permette di sapere quale pulsante nel gruppo risulti selezionato, l'identificazione è possibile attraverso il metodo `getActionCommand` del `ButtonModel` restituito. La stringa “action command” è impostabile tramite il metodo

`setActionCommand(String)` del singolo pulsante. Il programma che segue crea un gruppo di pulsanti e identifica il valore selezionato attraverso le stringhe di comando.

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

public class Main {

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		JCheckBox box1 = new JCheckBox("pizza");
		JCheckBox box2 = new JCheckBox("lasagne");
		JCheckBox box3 = new JCheckBox("gelato");
		box1.setActionCommand("pizza");
		box2.setActionCommand("lasagne");
		box3.setActionCommand("gelato");

		final ButtonGroup foodGroup = new ButtonGroup();
		foodGroup.add(box1);
		foodGroup.add(box2);
		foodGroup.add(box3);
		box1.setSelected(true); //valore predefinito

		JPanel foodPanel = new JPanel(new GridLayout(3, 1, 8, 8));
		foodPanel.add(box1);
		foodPanel.add(box2);
		foodPanel.add(box3);
		foodPanel.setBorder(BorderFactory.createTitledBorder("Food"));

		JButton okButton = new JButton("ok");
		okButton.addActionListener(new ActionListener() {

			public void actionPerformed(ActionEvent e) {
				checkChoice(foodGroup, (JComponent)e.getSource());
			}
		});
		JPanel okContainer = new JPanel(new FlowLayout(FlowLayout.RIGHT));
		okContainer.add(okButton);

		JFrame window = new JFrame("Test");
		window.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
		window.add(foodPanel, BorderLayout.CENTER);
		window.add(okContainer, BorderLayout.SOUTH);
		window.pack();
		window.setVisible(true);
	}

	private static void checkChoice(ButtonGroup group, JComponent source) {
		ButtonModel selection = group.getSelection();
		String actionCommand = selection.getActionCommand();
		if("pizza".equals(actionCommand)) {
			JOptionPane.showMessageDialog(source, "PIZZA!!!");
		} else if("lasagne".equals(actionCommand)) {
			JOptionPane.showMessageDialog(source, "LASAGNE!!!");
		} else if("gelato".equals(actionCommand)) {
			JOptionPane.showMessageDialog(source, "GELATO!!!");
		}
	}
}
```

Premendo il pulsante con etichetta “ok” è invocato il metodo `checkChoice` che, tramite il gruppo di pulsanti `buttonGroup`, determina quale tra le caselle di spunta box1, 2 e 3 risulti essere selezionata e apre una finestra di dialogo con una stringa conseguente alla scelta.

ImageIcon

`ImageIcon` è una capsula per immagini idealmente destinate a decorare pulsanti ed etichette. Per creare un'ImageIcon si può passare al suo costruttore il percorso assoluto o relativo di un file:

```
ImageIcon icon = new ImageIcon(“res/icona.png”);
```

un URL:

```
URL iconRes = getClass().getResource(“res/icona.png”);
ImageIcon icon = new ImageIcon(iconRes);
```

o un `java.awt.image.BufferedImage` – tramite `BufferedImage` è possibile creare icone al momento dell'esecuzione.

```
BufferedImage image = new BufferedImage(64, 64, BufferedImage.TYPE_INT_ARGB);
Graphics2D g = image.createGraphics();
//..disegna usando g
ImageIcon icon = new ImageIcon(image);
```

Tutti i pulsanti Swing e le etichette hanno un metodo setIcon che permette di associare un'icona al componente.

Selettori continui.

Swing include i classici selettori a scorrimento, `JSlider` – il pallino che scorre lungo una barra di valori – e `JSpinner` – un valore numerico appartenente ad una successione e la casella combinata – `JComboBox`.

JSlider

I costruttori di `JSlider` permettono di creare un selettore verticale o orizzontale (parametro orientation) e di stabilire il valore minimo ed il valore massimo mostrati nella barra e selezionabili dall'utente. Quando l'utente interagisce con il selettore spostando il nodo di selezione lo slider produce degli eventi `ChangeEvent` quindi per catturare l'interazione utente occorre usare un `javax.swing.event.ChangeListener`. Supponedo che `slider` sia una variabile di tipo `JSlider` vale il seguente esempio:

```
slider.addChangeListener(new ChangeListener() {

     public void stateChanged(ChangeEvent e) {
          JSlider source = (JSlider)e.getSource();
          int value = source.getValue();
          System.out.println(value); //stampa il valore sulla console
     }
});
```

Attraverso lo slider (metodo `getValueIsAdjusting`) è possibile sapere se un evento `ChangeEvent` si è verificato ma l'utente non ha ancora terminato di far scorrere lo slider. Il valore restituito da `getValueIsAdjusting` può quindi essere usato per eseguire un certo compito solo quando l'utente ha effettivamente stabilito un nuovo valore per lo slider e non durante l'interazione.

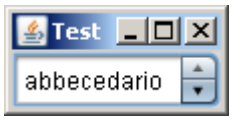
```
slider.addChangeListener(new ChangeListener() {

      public void stateChanged(ChangeEvent e) {
            JSlider source = (JSlider)e.getSource();
            if(!source.getValueIsAdjusting()) {
                  int value = source.getValue();
                  JOptionPane.showMessageDialog(source, "Valore: " + value);
            }
      }
});
```

I metodi `setPaintTick`, `setMinorTickSpacing`, `setMajorTickSpacing`, `setPaintTrack`, setPaintLabels permettono di controllare le decorazioni dello slider. I “tick” controllano le lineette che identificano visivamente le diverse posizioni valide dello slider. Le “labels” sono le etichette associate alle diverse posizioni dello slider. Per creare le etichette si prende un'istanza di `java.util.Hashtable<Integer, JComponent>` – `Dictionary` è vecchio stantio. Nella tabella si associano delle stringhe alle posizioni dello slider da etichettare. Preparato il tutto si passa la tabella al metodo `setLabelTable` e si invoca `setPaintLabels(true)` – a meno che non sia già stato invocato ovviamente. Il risultato è uno slider con delle etichette che identificano i punti chiave. Ad esempio dato uno slider che va da zero a dieci, volendo etichettare i valori zero, cinque e dieci con le stringhe “Minimo”, “Medio” e “Massimo” diremmo:

```
Hashtable<Integer, JComponent> table = new Hashtable<Integer, JComponent>();
table.put(0, new JLabel("Minimo"));
table.put(5, new JLabel("Medio"));
table.put(10, new JLabel("Massimo"));
slider.setLabelTable(table);
slider.setPaintLabels(true);
```

Da notare che `JSlider` usa solo l'aspetto dei componenti nella tabelle delle etichette. Se in quella tabella mettiamo un `JButton` associato al valore zero l'etichetta del valore zero avrà l'aspetto di un pulsante ma non le funzioni del pulsante.

JSpinner



`JSpinner` offre la possibilità di selezionare un valore in un certo range sia attraverso la pressione di due pulsanti di scorrimento sia attraverso l'inserimento diretto. Attraverso il suo `SpinnerModel` `JSpinner` è in grado di visualizzare qualsiasi specie di dati: numeri, orari, etichette, un po' di tutto. Il codice che segue crea un spinner che mostra valori interi compresi tra 0 e 10, permette all'utente di scorrere questi valori con i pulsanti di scorrimento e consente all'utente di immette un qualsiasi valore nella casella di testo associata.

```
SpinnerNumberModel model = new SpinnerNumberModel(0, 0, 10, 1);
JSpinner spinner = new JSpinner(model);
```

Per creare uno spinner che non permetta all'utente di modificare il valore attraverso il campo di

testo associato si recupera il campo di testo dall'editor e si imposta la sua proprietà `editable` a `false`.

```
JSpinner.DefaultEditor editor = (JSpinner.DefaultEditor)spinner.getEditor();
editor.getTextField().setEditable(false);
```

Per catturare l'interazione utente si segue la stessa strada percorsa con `JSlider`: `ChangeListener`.

```
spinner.addChangeListener(new ChangeListener() {

      public void stateChanged(ChangeEvent e) {
            JSpinner source = (JSpinner)e.getSource();
            Object value = source.getValue();
            System.out.println(value);
      }
});
```

Il tipo di valori gestito da uno spinner è arbitrariamente decidibile quindi il valore corrente è restituito in forma di `Object`. Ad esempio se i valori sono numerici allora quell'`Object` sarà un `Number`.

```
SpinnerNumberModel model = new SpinnerNumberModel(0, 0, 10, 1);
JSpinner spinner = new JSpinner(model);
JSpinner.DefaultEditor editor = (JSpinner.DefaultEditor)spinner.getEditor();
editor.getTextField().setEditable(false);

spinner.addChangeListener(new ChangeListener() {

      public void stateChanged(ChangeEvent e) {
            JSpinner source = (JSpinner)e.getSource();
            Number value = (Number)source.getValue();
            int num = value.intValue();
            System.out.println(num + 100);
      }
});
```

Per visualizzare valori di tipo arbitrario si può usare `SpinnerListModel`. Il modello accetta in costruzione una collezione o un array di oggetti e visualizza il prodotto dell'invocazione del metodo `toString` sui valori presenti nella lista o nell'array.

Aree di testo

I componenti di testo Swing derivano da `javax.swing.text.JTextComponent` dunque tutti i metodi là dichiarati sono validi per tutti.

JTextField

`JTextField` è la rappresentazione Swing di un'area di testo a linea singola, detta anche campo di testo. I suoi costruttori permettono di specificare il numero di colonne di testo predefinite per il campo. Il numero di colonne determina la larghezza iniziale del campo di testo ma non limita la quantità di caratteri inseribili dall'utente. Per impostare il testo visualizzato sul campo si usa il metodo `setText(String)`, per ottenere il testo contenuto nel campo si usa il metodo `getText()` che restituisce un oggetto `String`. Un `JTextField` è in grado di produrre un `ActionEvent` in corrispondenza di un evento di interazione, tipicamente collegato alla pressione del tasto invio. Per selezionare tutto il testo contenuto nel campo si usa il metodo `selectAll()`. Il codice che segue crea

e mostra sullo schermo un piccolo form con due campi di testo. Quando l'utente passa da un campo di testo ad un altro il testo contenuto nel campo viene selezionato interamente.

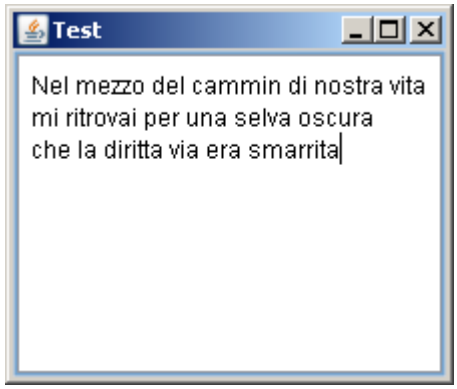
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class Main {

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		FocusListener focusListener = new FocusAdapter() {

			@Override public void focusGained(FocusEvent e) {
				JTextField campo = (JTextField)e.getSource();
				campo.selectAll();
			}
		};
		JTextField campoNome = new JTextField(20);
		JTextField campoCognome = new JTextField(20);
		JLabel etichettaNome = new JLabel("Nome");
		JLabel etichettaCognome = new JLabel("Cognome");

		campoNome.addFocusListener(focusListener);
		campoCognome.addFocusListener(focusListener);

		JPanel contenitore = new JPanel(new GridBagLayout());
		GridBagConstraints lim = new GridBagConstraints();
		lim.gridx = lim.gridy = 0;
		lim.insets = new Insets(4, 4, 4, 4);
		lim.fill = GridBagConstraints.HORIZONTAL;
		contenitore.add(etichettaNome, lim);
		lim.gridy = 1;
		contenitore.add(etichettaCognome, lim);
		lim.gridx = 1; lim.gridy = 0;
		lim.weightx = 1;
		contenitore.add(campoNome, lim);
		lim.gridy = 1;
		contenitore.add(campoCognome, lim);

		JFrame window = new JFrame("Form");
		window.add(contenitore);
		window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
		window.pack();
		window.setVisible(true);

	}
}
```

Il programma usa un `FocusListener` per catturare l'acquisizione del focus da parte di un campo di testo, acquisizione conseguente alla selezione del campo di testo da parte dell'utente, sia attraverso il mouse che attraverso il tasto di tabulazione. All'acquisizione del focus consegue la selezione dell'intero contenuto del campo di testo.

JPasswordField

`JPasswordField` funziona come un `JTextField` con la particolarità che il testo inserito nel campo è visivamente sostiuito da una serie di caratteri di occultamento. Per recuperare la password inserita si usa il metodo `getPassword()`.

JTextArea



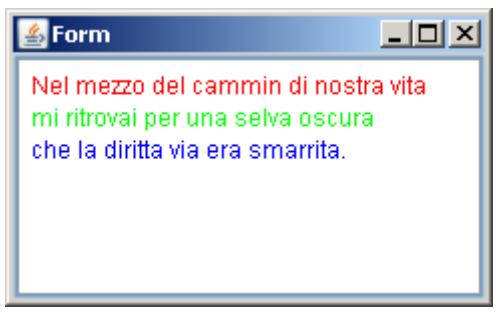
`JTextArea` è un'area di testo a formattazione singola: tutto il testo contenuto nell'area può avere un solo tipo di Font, non ammette immagini eccetera. `JTextArea` offre la possibilità di abilitare la separazione visiva del testo in più linee nel caso in cui la larghezza della linea di testo ecceda la larghezza dell'area di testo attraverso il metodo `setLineWrap(true)`. E' possibile stabilire se la separazione debba mantenere intere le parole – `setWrapStyleWord(true)` – o possa spezzarle – `setWrapStyleWord(false)`. Normalmente un'area di testo è inserita all'interno di un `JScrollPane` per consentire all'utente di visualizzare un testo di dimensioni eccedenti quelle della JTextArea mantenendo inalterata la regione di spazio che l'area occupa nell'interfaccia. JTextArea offre un metodo append(String) per poter accodare del testo a quello già presente. Quando un'area di testo è inserita all'interno di un JScrollPane le barre di scorrimento del JScrollPane seguono la posizione del carrello della JTextArea. Muovendo il carrello con setCaretPosition(int) è possibile quindi controllare la porzione di testo visualizzata quando il testo stesso ecceda l'area di proiezione. L'applicazione che segue crea un'interfaccia con un campo di testo e un'area di testo. Quando l'utente preme invio nel campo di testo il contenuto del campo è inserito in cima all'area di testo.

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

public class Main {

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		final JTextArea textArea = new JTextArea(20, 20);
```

```
            textArea.setEditable(false);
            textArea.setLineWrap(true);
            JScrollPane areaScroller = new JScrollPane(textArea);
            final JTextField textField = new JTextField(20);
            textField.addActionListener(new ActionListener() {

                  public void actionPerformed(ActionEvent e) {
                        String text = textField.getText();
                        if(text.trim().length() > 0) {
                              textArea.setCaretPosition(0);
                              textArea.replaceSelection(text + "\n");
                              textField.selectAll();
                        }
                  }
            });

            JPanel contenitore = new JPanel(new BorderLayout(4, 4));
            contenitore.add(textField, BorderLayout.NORTH);
            contenitore.add(areaScroller, BorderLayout.CENTER);


            JFrame window = new JFrame("Form");
            window.add(contenitore);
            window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
            window.pack();
            window.setVisible(true);

      }
}
```

L'invocazione `setEditable(false)` fa sì che l'area di testo non accetti input dall'utente. Il metodo `setLineWrap(true)` fa sì che il testo venga visivamente separato automaticamente su più linee. Omettendo l'invocazione le linee di testo saranno mantenute integre ed il `JScrollPane` si occuperà di visualizzare la barra di scorrimento orizzontale quando necessario. Quando l'utente preme invio nel campo di testo è eseguito il codice contenuto nell'`actionPerformed` dell'`ActionListener` collegato al campo di testo. Il codice prende il testo dal campo e lo inserisce in cima all'area di testo. Per farlo posiziona il carrello all'inizio del documento con `setCaretPosition(0)`, e rimpiazza la selezione corrente nell'area di testo. Se non esiste una selezione il metodo replaceSelection(String) opera come un inserimento di testo nella posizione in cui si trova attualmente il carrello, effetto puntualmente sfruttato nel programma. Dopo aver inserito il testo il carrello si sposta automaticamente nella posizione seguente l'ultimo carattere inserito. Spostando nuovamente indietro il carrello con `setCaretPosition(0)` ci assicuriamo che il `JScrollPane` mantenga la visibilità sulla prima riga dell'area di testo – necessario considerando il caso in cui il testo inserito sia sufficientemente lungo da creare uno scorrimento verso il basso.

JTextPane



`JTextPane` è un'area di testo formattato. Oltre a formati specifici per paragrafi e caratteri supporta l'inserimento di componenti Swing nel testo e potendo un componente Swing contenere qualsiasi cosa ne deriva una più che ampia flessibilità. Per applicare uno stile di paragrafo si può usare il

metodo `setParagraphAttributes`, per uno stile di carattere il metodo `setCharacterAttributes`. Il paragrafo è una porzione di testo compresa tra due interruzioni di linea. Entrambi i metodi si applicano alla selezione corrente. Se non esiste una selezione lo stile è applicato alla posizione in cui si trova attualmente il carello. Il testo inserito in quella posizione successivamente alla modifica dello stile riflette tale modifica. Gli stili, tanto di paragrafo quanto di carattere, sono istanze di `AttributeSet`. Esistono diverse classi che sono degli `AttributeSet`. Una è `SimpleAttributeSet`. SimpleAttributeSet può essere usata in congiunzione con i metodi statici della classe `javax.swing.text.StyleConstants` per creare rapidamente degli stili da applicare al testo di un `JTextPane`. Ad esempio, per uno stile di carattere grassetto applicato alla posizione corrente del carrello:

```
SimpleAttributeSet grassetto = new SimpleAttributeSet();
StyleConstants.setBold(grassetto, true);
textPane.setCharacterAttributes(grassetto, true);
```

o per uno stile di paragrafo, analogamente:

```
SimpleAttributeSet allineaDestra = new SimpleAttributeSet();
SylteConstants.setAlignment(allineaDestra, StyleConstants.ALIGN_RIGHT);
textPane.setParagraphAttributes(allineaDestra, true);
```

La classe `StyleConstants` può anche essere usata per determinare i valori dello stile applicato ad una posizione in un `JTextPane`. I metodi `getCharacterAttributes` e `getParagraphAttributes` di `JTextPane` restituiscono lo stile di paragrafo e di carattere vigenti nella posizione corrente del carrello. Applicando agli `AttributeSet` così ottenuti i metodi “get” di `StyleConstants` è possibile risalire ai valori di singoli attributi. Ad esempio:

```
AttributeSet stileParagrafo = textPane.getParagraphAttributes();
int allineamento = SylteConstants.getAlignment(stileParagrafo);
if(allineamento == StyleConstants.ALIGN_LEFT) {
     ...
} else if(allineamento == StyleConstants.ALIGN_RIGHT) {
     ...
} else if(allineamento == StyleConstants.ALIGN_CENTER) {
     ...
} else if(allineamento == StyleConstants.ALIGN_JUSTIFIED) {
     ...
}
```

Per inserire del testo in un `JTextPane` programmaticamente si può usare il metodo `replaceSelection` avendo cura di posizionare il carrello nel punto di inserimento desiderato con `setCaretPosition`. Per aggiungere testo in coda si userà un codice tipo:

```
textPane.setCaretPosition(textPane.getDocument().getLength());
textPane.replaceSelection("hello world");
```

Per inserire del testo formattato si aggiuge un'invocazione a `setCharacterAttributes`:

```
textPane.setCaretPosition(textPane.getDocument().getLength());
textPane.setCharacterAttributes(stileCarattere);
textPane.replaceSelection("hello world");
```

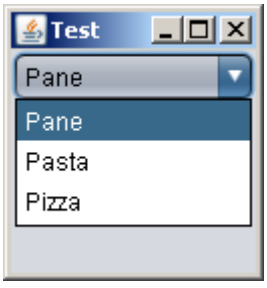
Il programma che segue crea una finestra con un pulsante e un `JTextPane`. Premendo il pulsante l'utente può cambiare il colore del carattere. Il colore proposto dalla finestra di dialogo `JColorChooser` è determinato dal colore attuale del testo nel punto di inserimento.

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

```
import javax.swing.text.*;

public class Main {

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		final JTextPane textPane = new JTextPane();
		JScrollPane areaScroller = new JScrollPane(textPane);
		areaScroller.setPreferredSize(new Dimension(400, 400));

		JButton button = new JButton(new AbstractAction("Colore") {

			@Override public void actionPerformed(ActionEvent e) {
				AttributeSet currentAtt = textPane.getCharacterAttributes();
				Color color = StyleConstants.getForeground(currentAtt);
				color = JColorChooser.showDialog(textPane, "Colore", color);
				if(color != null) {
					SimpleAttributeSet att = new SimpleAttributeSet();
					StyleConstants.setForeground(att, color);
					textPane.setCharacterAttributes(att, true);
					textPane.requestFocusInWindow();
				}
			}
		});
		JPanel buttonContainer = new JPanel(new FlowLayout(FlowLayout.LEFT));
		buttonContainer.add(button);

		JPanel contenitore = new JPanel(new BorderLayout(4, 4));
		contenitore.add(buttonContainer, BorderLayout.NORTH);
		contenitore.add(areaScroller, BorderLayout.CENTER);

		JFrame window = new JFrame("Form");
		window.add(contenitore);
		window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
		window.pack();
		window.setVisible(true);
	}
}
```

Da notare che a differenza di `JTextArea`, `JTextPane` non permette un controllo diretto sulla separazione visiva delle linee. Quando un `JTextPane` è contenuto in un `JScrollPane` tale separazione è automatica.

JComboBox



`JComboBox` è una casella combinata. E' possibile creare una `JComboBox` in abbinamento al suo

modello:

```
DefaultComboBoxModel model = new DefaultComboBoxModel();
model.addElement(“Pane”);
model.addElement(“Pasta”);
model.addElement(“Pizza”);
JComboBox box = new JComboBox(model);
```

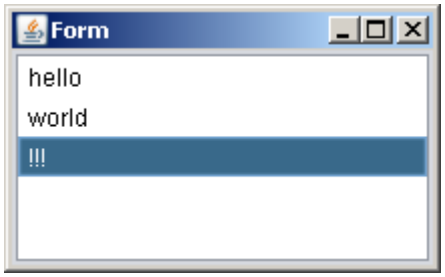
Quando l'utente cambia il valore corrente della casella combinata la casella emette un `ActionEvent`, intercettabile con un `ActionListener`:

```
box.addActionListener(new ActionListener() {

      public void actionPerformed(ActionEvent e) {
            JComboBox source = (JComboBox)e.getSource();
            Object selezione = source.getSelectedItem();
            //fai qualcosa con la selezione...
      }
});
```

Impostando a true la proprietà editable della casella – `box.setEditable(true)` – l'utente può digitare del testo nella casella di selezione. Il testo digitato non viene inserito nella lista di selezione. Per collegare un evento all'interazione dell'utente con la casella di testo si può passare per l'editor della casella combinata. Il codice che segue in cui box è un'ipotetica `JComboBox` con proprietà `editable` impostata a `true` apre una finestra di dialogo con l'input dell'utente ogni volta che egli prema invio nella casella di testo:

```
ComboBoxEditor editor = box.getEditor();
Component component = editor.getEditorComponent();
if(component instanceof JTextField) {
      final JTextField field = (JTextField)component;
      field.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                  JOptionPane.showMessageDialog(field, field.getText());
            }
      });
}
```

JList



`JList` presenta all'utente una lista di dati all'interno della quale è possibile selezionare un valore. Per stabilire i dati presentati dalla lista è possibile appoggiarsi ad un oggetto `DefaultListModel`. Il codice che segue crea un `DefaultListModel`, vi aggiunge tre stringhe, crea una `JList` e associa alla lista il modello precedente.

```
DefaultListModel listData = new DefaultListModel();
listData.addElement("Hello");
listData.addElement("World");
listData.addElement("!!!");
JList list = new JList(listData);
```

Il risultato è che le tre stringhe inserite nel modello appaiono nella lista in ordine di inserimento. La lista può contenere un qualsiasi tipo di oggetti. Ciò che la lista visualizza è il prodotto dell'invocazione del metodo `toString` sul valore contenuto quindi personalizzando il metodo `toString` della classe degli oggetti che si vogliono inserire nella lista è possibile manipolare a piacere il testo che sarà visualizzato nella lista. La lista restituisce il valore correntemente selezionato attraverso il metodo `getSelectedValue` e l'indice dello stesso elemento con il metodo `getSelectedIndex`. E' possibile impostare la selezione corrente con i metodi `setSelectedIndex` e `setSelectedValue`. La lista risponde alla selezione di un valore da parte dell'utente notificando un evento `javax.swing.event.ListSelectionEvent` agli ascoltatori `javax.swing.event.ListSelectionListener`. La sorgente dell'evento è la lista e i valori restituiti dai metodi `getFirstIndex` e `getLastIndex` dell'evento `ListSelectionEvent` possono essere usati per recupare il valore oggetto della selezione. `JList` offre un metodo, `locationToIndex`, utile per definire interazioni personalizzate dell'utente con lista attraverso il mouse. Il metodo citato richiede come argomento un punto nello spazio di coordinate della lista. Il codice che segue usa un MouseListener per visualizzare una finestra di popup quando l'utente preme il pulsante destro del mouse su un elemento della lista.

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

public class Main {

	public static void main(String[] args) {
		EventQueue.invokeLater(new Runnable() {

			public void run() {
				start();
			}
		});
	}

	private static void start() {
		DefaultListModel listData = new DefaultListModel();
		listData.addElement("Hello");
		listData.addElement("World");
		listData.addElement("!!!");
		JList list = new JList(listData);
		JScrollPane container = new JScrollPane(list);

		list.addMouseListener(new MouseAdapter() {

			public void mousePressed(MouseEvent e) {
				if(SwingUtilities.isRightMouseButton(e)) {
					popup((JList)e.getSource(), e.getPoint());
				}
			}
		});


		JFrame window = new JFrame("Form");
		window.add(container);
		window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
		window.pack();
		window.setVisible(true);
	}

	private static void popup(JList list, Point clickPoint) {
		int index = list.locationToIndex(clickPoint);
		if(index >= 0) { //se index == -1 allora il click non è su un elemento
			list.setSelectedIndex(index);
			Object selection = list.getModel().getElementAt(index);
			JPopupMenu popup = new JPopupMenu();
			popup.add("Cancella {" + selection + "}");
			popup.show(list, clickPoint.x, clickPoint.y);
```

```
            }
        }
}
```

`JList` offre due metodi per cambiare completamente il proprio contenuto, entrambi nominati `setListData`. Uno richiede come argomento un array l'altro un `java.util.Vector`. Se gli elementi da inserire nella lista sono contenuti in collezioni diverse è possibile usare il costruttore di `Vector` che accetta un `java.util.Collection` come argomento.

```
LinkedList<String> dati = … una lista di dati ricavati da qualche parte del programma
Vector<String> vector = new Vector<String>(dati);
jlist.setListData(vector);
```

Normalmente una `JList` è sempre inserita in un `JScrollPane` in modo tale da permettere alla lista di avere un numero arbitrario di valori senza occupare più di un certo spazio nell'interfaccia.

JTable

A meno che non si tratti della tabella più banale immaginabile ogni `JTable` parte dalla creazione del suo modello più flessibile, `javax.swing.table.DefaultTableModel`. `JTable` è uno dei componenti normalmente inseriti in `JScrollPane`. La creazione di un `JTable` è molto breve. Prima di dichiara il suo modello:

```
DefaultTableModel data = new DefaultTableModel();
```

A questo punto si creano le colonne usando il metodo `addColumn` di `DefaultTableModel`:

```
data.addColumn(“Nome”);
data.addColumn(“Cognome”);
data.addColumn(“Indirizzo”);
```

Per aggiungere righe si può usare o il metodo `addRow` che accetta un array, ad esempio:

```
data.addRow(new String[] { “Mario”, “Rossi”, “Via Pascoli 2” });
```

oppure si può aggiungere una riga vuota:

```
int riga = data.getRowCount();
data.setRowCount(riga + 1);
```

per poi inserire i valori nelle singole celle con il metodo `setValueAt(valore, riga, colonna)`:

```
data.setValueAt(“Mario”, riga, 0); //prima colonna
data.setValueAt(“Rossi”, riga, 1); //seconda colonna
data.setValueAt(“Via Pascoli 2”, riga, 2); //terza colonna
```

Una volta immessi tutti i dati si crea la tabella, la si inserisce in un `JScrollPane` e poi si aggiungerà quest'ultimo all'interfaccia

```
JTable table = new JTable(data);
JScrollPane scroller = new JScrollPane(table);
```

L'impostazione predefinita della tabella consente all'utente di modificare i valori in essa contenuti. Per impedire tale modifica si può creare una sottoclasse anonima sovrascrivendo il metodo `isCellEditable`.

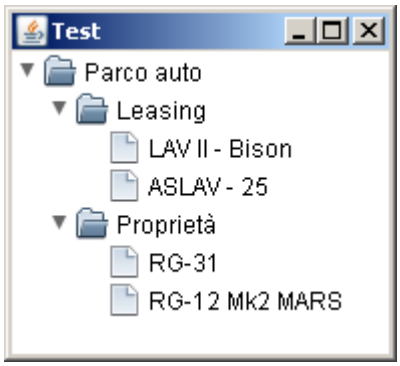
```
JTable table = new JTable(data) {

     @Override public boolean isCellEditable(int row, int col) {
          return false;
     }
};
```

Per ottenere il valore di una cella della tabella si usa il metodo `getValueAt(riga, colonna)` del suo modello. Quando l'utente cambia il valore di una cella della tabella il modello della tabella genera un evento `TableModelEvent` e lo propaga ai suoi `TableModelListener`. Il codice che segue crea una tabella con una riga vuota e attraverso un TableModelListener apre una finestra di notifica che riporta il valore inserito.

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

public class Main {

     public static void main(String[] args) {
          EventQueue.invokeLater(new Runnable() {

               public void run() {
                    start();
               }
          });
     }

     private static void start() {
          DefaultTableModel data = new DefaultTableModel();
          data.addColumn("Nome");
          data.addColumn("Cognome");
          data.addColumn("Indirizzo");
          data.setRowCount(1);
          final JTable table = new JTable(data);

          data.addTableModelListener(new TableModelListener() {

               public void tableChanged(TableModelEvent e) {
                    if(e.getType() == TableModelEvent.UPDATE) {
                         TableModel source = (TableModel)e.getSource();
                         int row = e.getFirstRow();
                         int col = e.getColumn();
                         Object value = source.getValueAt(row, col);
                         JOptionPane.showMessageDialog(table, value);
                    }
               }
          });


          JScrollPane scroller = new JScrollPane(table);

          JFrame window = new JFrame("Form");
          window.add(scroller);
          window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
          window.pack();
          window.setVisible(true);
     }
}
```

Come `JList` anche `JTable` è in grado di convertire un punto del mouse in un indice stavolta di riga – `rowAtPoint` – o di colonna – `columnAtPoint`. Questi metodi possono essere usati per gestire i popup.

JTree



`JTree` mostra dei dati organizzati in un albero. Come `JTable` anche `JTree` può essere inizializzato in diversi modi. Uno di questi modi prevede l'uso di oggetti `javax.swing.tree.DefaultMutableTreeNode`. Gli oggetti `DefaultMutableTreeNode` rappresentano i nodi dell'albero. Il valore contenuto nei nodi è il dato rappresentato dall'albero in corrispondenza del nodo. Come per la tabella anche l'albero invoca il metodo `toString` sui valori e poi proietta la stringa così ottenuta. Un nodo senza figli viene visualizzato con l'icona dei un documento, un nodo con figli appare come una cartella. Si parte della radice:

```
DefaultMutableTreeNode radice = new DefaultMutableTreeNode("radice");
```

Inseriamo dei figli nella radice usando il metodo `add`:

```
DefaultMutableTreeNode figlio = new DefaultMutableTreeNode("figlio");
radice.add(figlio);
```

E' possibile quindi aggiungere altri figli alla radice o figli ai figli della radice e così via fino a ricostruire la gerarchia desiderata. Una volta composta la struttura dati si passa la radice al costruttore di `DefaultTreeModel`:

```
DefaultTreeModel model = new DefaultTreeModel(radice);
```

Quindi si passa il modello al costruttore di `JTree`:

```
JTree albero = new JTree(model);
```

Si inserisce l'albero in un `JScrollPane`:

```
JScrollPane scroller = new JScrollPane(albero);
```

e si procede inserendo il pannello a scorrimento nell'interfaccia. Per ottenere la selezione corrente si usa il metodo `getSelectionPath` di `JTree`. Il metodo restituisce un oggetto `TreePath`. `TreePath` rappresenta un percorso tra la radice ed un nodo dell'albero. Per la selezione corrente il `TreePath` è il percorso tra la radice e il nodo attualmente selezionato. La selezione di un nodo genera un evento TreeSelectionEvent intercettabile con un TreeSelectionListener connesso all'albero. Il codice che segue è un ascoltatore di eventi di selezione collegato ad un albero che stampa sulla console il primo e l'ultimo elemento della selezione:

```
tree.addTreeSelectionListener(new TreeSelectionListener() {
```

```
      public void valueChanged(TreeSelectionEvent e) {
            TreePath selection = e.getPath();
            if(selection != null) {
                  System.out.println(selection.getPathComponent(0));
                  System.out.println(selection.getLastPathComponent());
            }
      }
});
```

Se l'albero è stato costruito come precedentemente indicato allora gli elementi di un `TreePath` sono oggetti di tipo `DefaultMutableTreeNode`. Per inserire dei nodi in un albero visualizzato si usa il metodo `insertNodeInto` del modello. Il metodo richiede il nodo da inserire, un nodo genitore e un indice per la posizione del nuovo nodo nella lista di figli del genitore. Il codice che segue richiede una stringa all'utente e la inserisce in un ipotetico albero `TREE` con modello `TREE_MODEL`:

```
TreePath selection = TREE.getSelectionPath();
if(selection != null) {
      DefaultMutableTreeNode parent = (DefaultMutableTreeNode)
                  selection.getLastPathComponent();
      String name = JOptionPane.showInputDialog(TREE, "Digitare il valore da inserire.");
      if(name != null) {
            DefaultMutableTreeNode newNode = new DefaultMutableTreeNode(name);
            TREE_MODEL.insertNodeInto(newNode, parent, 0);
      }
}
```

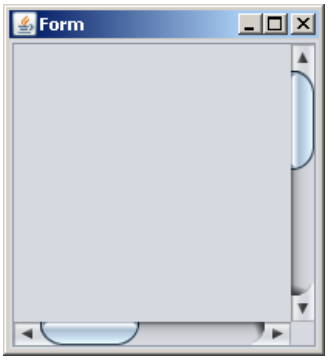
Per rimuovere un nodo da un albero si usa il metodo `removeNodeFromParent` del suo `DefaultTreeModel`. Il metodo richiede solo il nodo da rimuovere e cerca automaticamente il genitore da cui rimuoverlo. Per impostazione prefenita i valori dell'albero non sono modificabili dall'utente. Per cambiare questa impostazione occorre invocare il metodo `setEditable` dell'albero con parametro `true`. Per intercettare le modifiche apportate all'albero tramite l'interfaccia si collega un `TreeModelListener` al modello dell'albero. Si noti che `TreeModelListener` non riceve notifiche solo quando l'utente cambia il valore di un nodo ma più in generale ogni volta che i dati o la struttura dell'albero subiscono una mutazione – anche in via programmatica. Quando il valore di un nodo cambia – ripetiamo sia per effetto dell'interazione utente che per effetto di una manipolazione programmatica del modello – viene invocato il metodo `treeNodesChanged` dei `TreeModelListener` connessi al modello. Il codice che segue collega un `TreeModelListener` al modello `TREE_MODEL` di un ipotetico albero `TREE` e mostra una finestra di dialogo ogni volta che il valore di un nodo cambia.

```
MODEL.addTreeModelListener(new TreeModelListener() {
      public void treeNodesInserted(TreeModelEvent notUsed) {}
      public void treeNodesRemoved(TreeModelEvent notUsed) {}
      public void treeStructureChanged(TreeModelEvent notUsed) {}

      public void treeNodesChanged(TreeModelEvent e) {
            Object[] nodiMutati = e.getChildren();
            if(nodiMutati != null) {
                  for(Object o : nodiMutati) {
                        DefaultMutableTreeNode node = (DefaultMutableTreeNode)o;
                        JOptionPane.showMessageDialog(TREE, "Nuovo valore: " +
                              node.getUserObject());
                  }
            }
      }
});
```

Come liste e tabelle anche gli alberi hanno un metodo che dato un punto, idealmente generato da un evento del mouse sull'albero, restituisce l'elemento visivamente collocato in quella posizione. Il metodo è `getPathForLocation` e restituisce un oggetto `TreePath`, identico a quelli già visti nella selezione.

Contenitori particolari

JScrollPane



`JScrollPane` è un contenitore che visualizza automaticamente delle barre di scorrimento quando il componente che contiene ha dimensioni maggiori dell'area assegnata al `JScrollPane`. Le barre di scorrimento consentono di visualizzare il contenuto del `JScrollPane` un po' per volta. Il componente soggetto a scorrimento è assegnato o passandolo come argomento del costruttore:

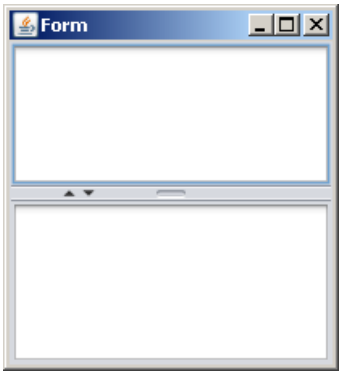
```
JList lista = new JList();
JScrollPane scroller = new JScrollPane(lista);
```

o usando il metodo `setViewportView`:

```
JScrollPane scroller = new JScrollPane();
scroller.setViewportView(lista);
```

Tramite i metodi `setHorizontalScrollBarPolicy` e `setVerticalScrollBarPolicy` è possibile specificare se le barre di scorrimento debbano essere sempre visibili, visibili quando necessario o sempre invisibili.

JSplitPane



`JSplitPane` contiene due componenti e permette all'utente di cambiare la distribuzione dello spazio orizzontale o verticale a loro assegnato. Le due regioni di spazio disponibili sono chiamate left e right. Nel caso in cui la divisione sia verticale left è la parte alta e right la parte bassa. `JSplitPane` ha diversi costruttori. Un modo per inizializzarne uno è usare il costruttore che richiede l'orientamento:

```
JSplitPane splitter = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
```

oppure

```
JSplitPane splitter = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
```

Per far apparire le piccole frecce che permettono all'utente di occultare completamente una delle due aree si usa il metodo `setOneTouchExpandable` con argomento `true`. Per aggiungere i due componenti si usano i metodi `setLeftComponent` e `setRightComponent`. Ad esempio:

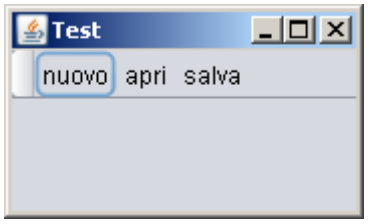
```
JPanel left = new JPanel();
JPanel right = new JPanel();
split.setLeftComponent(left);
split.setRightComponent(right);
```

`JSplitPane`, se ha sufficiente spazio, tiene conto della dimensione preferita dei due componenti quando deve decidere come distribuire lo spazio vale a dire che se il componente a sinistra è largo 200 pixel e quello a destra è largo 400 e lo spazio a disposizione del `JSplitPane` è di poco più di 600 pixel allora il divisore apparirà intorno ai 200 pixel. Per impostare manualmente la posizione del divisore si usano i metodo `setDividerLocation(int)` e `setDividerLocation(double)`. Il primo opera in termini assoluti – il divisore sarà collocato nel punto indicato – il secondo opera in senso propozionale. Il metodo proporzionale richiede che il pannello sia visibile altrimenti è senza effetto. Per condizionare l'esecuzione del metodo `setDividerLocation(double)` alla visibilità del pannello è sufficiente aggiungere al pannello stesso un `ComponentListener`:

```
JSplitPane splitter = …
splitter.addComponentListener(new ComponentAdapter() {

     public void componentShown(ComponentEvent e) {
          JSplitPane source = (JSplitPane)e.getSource();
          source.setDividerLocation(0.5); //a metà
     }
});
```

Sia la posizione proporzionale sia la posizione assoluta sono calcolate a partire dall'alto o a partire da sinistra – quale dei due dipende dall'orientamento.

JToolbar



`JToolBar` è un pannello che rappresenta un barra degli strumenti cioè una fila di pulsanti di dimensioni omogenee solitamente collocata lungo i bordi della finestra. Per impostazione predefinita una `JToolBar` è sganciabile dal contenitore in cui si trova attraverso il trascinamento del mouse e può essere spostata a piacimento dall'utente. Quando è sganciata la `JToolBar` è inserita in una finestrella creata automaticamente. Quando l'utente chiude la finestrella della `JToolBar` quest'ultima torna a occupare la posizione che aveva nell'interfaccia. `JToolBar` perde la capacità di essere separata dall'interfaccia quando si invoca il metodo `setFloatable` con argomento `false`. Per inserire i pulsanti nella barra si usa il metodo add, come un normale contenitore.

```
JToolBar toolbar = new JToolBar();
```

```
toolbar.add(un pulsante);
toolbar.add(un altro pulsante);
```

JToolBar supporta l'inserimento diretto di oggetti Action.

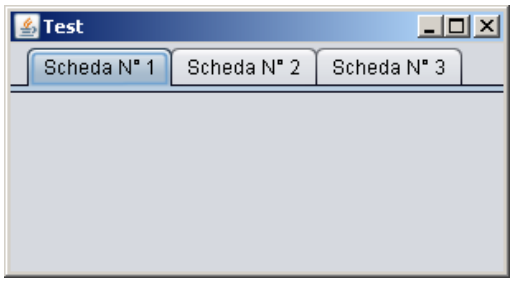
```
toolbar.add(un action);
toolbar.add(un altro action);
```

In questo caso la barra genera automaticamente un controllo sulla base delle proprietà dell'azione – in particolare icona e testo. Il metodo `addSeparator` consente di introdurre un effetto grafico di separazione tra gruppi di pulsanti:

```
toolbar.add(un pulsante);
toolbar.add(un altro pulsante);
toolbar.addSeparator();
toolbar.add(un terzo pulsante);
```

Il metodo `addSeparator(Dimension)` permette di specificare le dimensioni dell'effetto di separazione.

JTabbedPane



`JTabbedPane` è un contenitore a schede. Ogni scheda contiene un componente diverso. Per creare il pannello a schede si può usare il suo costruttore vuoto:

```
JTabbedPane pane = new JTabbedPane();
```

Una volta dichiarato e inizializzato il pannello si possono aggiungere schede usando uno dei metodi `addTab`:

```
pane.add("Scheda N° 1", new JPanel());
pane.add("Scheda N° 2", new JPanel());
pane.add("Scheda N° 3", new JPanel());
```

Quando l'utente seleziona una scheda il pannello genera un evento `ChangeEvent` intercettabile con un ascoltatore `ChangeListener`. Il seguente codice emette un messaggio ogni volta che cambia la scheda attualmente selezionata:

```
pane.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
            JTabbedPane source = (JTabbedPane)e.getSource();
            int tabIndex = source.getSelectedIndex();
            String tabTitle = source.getTitleAt(tabIndex);
            JOptionPane.showMessageDialog(source, "Selezione: " + tabTitle);
      }
});
```

Le intestazioni delle schede possono essere variamente personalizzate con icone – `setIconAt` –

colori – `setBackgroundAt` – o essere interamente rimpiazzate da componenti – `setTabComponentAt`.

JDesktopPane

`JDesktopPane` è un desktop virtuale vale a dire un contenitore di finestre inseribile in una finestra di sistema. Le finestre inseribili in `JDesktopPane` sono oggetti di tipo `JInternalFrame`.

JInternalFrame

`JInternalFrame` è il tipo di finestra inseribile in un `JDesktopPane`. La creazione di un `JInternalFrame` può essere fatta come segue:

```
JInternalFrame frame = new JInternalFrame();
frame.setTitle("JInternalFrame");
frame.setSize(300, 300);
frame.setResizable(true);
frame.setIconifiable(true);
frame.setClosable(true);
frame.setVisible(true);
```

Per aggiungere la finestra al `JDesktopPane` si usa il metodo add di quest'ultimo:

```
desktop.add(frame);
```

Da notare che il `JInternalFrame` aggiunto deve essere visibile – `setVisible(true)` – e deve avere una dimensione – o per via dell'invocazione `setSize` o tramite il metodo `pack`. Fermo restando il fatto che `JInternalFrame` esiste in un `JDesktopPane` per il resto si comporta come una qualsiasi altra finestra Swing.

Finestre di dialogo interne

La classe `JOptionPane` offre dei metodi per creare finestre di dialogo all'interno di un `JDesktopPane`. I metodi per la creazione di finestre di dialogo interne sono contrassegnati dalla parola `Internal` – `showInternalMessageDialog`, `showInternalInputDialog` eccetera. Il funzionamento di questi metodi è analogo ai corrispondenti metodi per la creazione di finestre di dialogo vere e proprie. L'unica differenza sta nel fatto che il componente richiesto come primo argomento dei metodi in parola deve essere contenuto in un `JInternalFrame`. Oltre a bloccare la finestra interna la proiezione di una finestra di dialogo interna causa anche il blocco dell'interazione con la finestra che contiene il `JDesktopPane`.

Altri componenti

JProgressBar

`JProgressBar` è un componente che mostra una decorazione animata variamente collegabile al procedere di una certa operazione. `JProgressBar` ha due modalità operative: una per i casi in cui il procedimento a cui si riferisce non è frazionabile e una per i casi in cui lo è. Se il procedimento è frazionabile allora si userà `JProgressBar` per mostrare l'avanzamento di stato altrimenti si userà `JProgressBar` come semplice segnalatore che il programma sta eseguendo delle operazioni. Si può creare una `JProgressBar` con il costruttore vuoto:

```
JProgressBar bar = new JProgressBar();
```

Se il procedimento non è frazionabile si imposta la barra a indeterminata:

```
bar.setIndeterminate(true);
```

In caso contrario si impostano i valori minimo e massimo:

```
bar.setMinimum(0);
bar.setMaximum(100);
```

I valori dipendono dal procedimento: possono indicare il numero di passaggi nell'inizializzazione di un programma o il numero di byte letti da un file o il numero di frame di un filmato in riproduzione eccetera. E' possibile inserire una stringa di testo all'interno della barra abilitando il disegno del

testo con il metodo `setStringPainted(true)` e impostando la stringa visualizzata con `setString`. Usando il metodo `setValue` si cambia il valore corrente della barra mentre il metodo `getValue` restituisce il valore corrente.

SwingWorker

La classe `SwingWorker` è uno strumento per il coordinamento tra l'Event Dispatcher Thread e altri Thread. `SwingWorker` risponde a due necessità: una è quella di garantire il rispetto del'architettura a Thread singolo di Swing l'altro è quella di evitare il sovraccarico dell'EDT stesso. l'EDT controlla tutta la gestione delle interfacce Swing, dalla propagazione degli eventi di interazione all'aggiornamento visivo. Ogni compito in più che gli si affida rischia di rendere l'interfaccia pesante dal punto di vista dell'interazione utente: ad esempio se premo il pulsante salva e faccio salvare all'EDT un file di una marea di megabyte il risultato è che il pulsante apparirà nello stato premuto e l'interfaccia non risponderà ai comandi finché il file non sia stato salvato. Qualsiasi operazioni onerosa deve essere affidata a Thread diversi dall'EDT. `SwingWorker` è una classe generica e richiede la specificazione del tipo di valore restituito dalla sua operazione parallela e del tipo di valore parziale che emette durante la computazione. Ad esempio se il compito che vogliamo affidargli è quello di leggere interamente un file di testo allora `SwingWorker` restituirà uno String e non emettera valori parziali quindi il nostro `SwingWorker` sarà:

```
//pseudo codice
SwingWorker<String, Void> worker = new SwingWorker<String, Void>() {
      ...
};
```

Se i file da leggere sono più d'uno possiamo decidere di farglieli leggere tutti quanti o permettergli di comunicare il testo letto un file alla volta. Nel primo caso avremo uno `SwingWorker<Collection<String>, Void>`, cioè che restituisce una collezione contenente tante stringhe quante sono i file da leggere e non ha valori parziali. Nel secondo caso avremo uno `SwingWorker<Void, String>`, cioè che non restituisce alcunchè ma comunica il testo di un file non appena ha terminato di leggero, prima di passare alla lettura dei rimanenti. Quando creiamo il nostro `SwingWorker` la prima cosa che facciamo è definire il metodo `doInBackground`. Questo metodo contiene il codice eseguito dal Thread parallelo all'EDT.

```
//pseudo codice
SwingWorker<String, Void> worker = new SwingWorker<String, Void>() {

      @Override protected String doInBackground() {
            File file = new File(...
            ...leggi il file come testo
            ..restituisci il testo come String.
      }
};
```

Se al termine dell'operazione parallela è necessario eseguire un qualche aggiornamento dell'interfaccia che tenga conto del valore prodotto allora ridefiniremo il metodo done tenendo presente che questo metodo è eseguito dall'EDT:

```
//pseudo codice
SwingWorker<String, Void> worker = new SwingWorker<String, Void>() {

      @Override protected String doInBackground() {
            File file = new File(...
            ...leggi il file come testo
            ..restituisci il testo come String.
      }
```

```
      @Override protected void done() {
            String testo = get(); //prendiamo il valore generato parallelamente
            jtextArea.setText(testo); //aggiorniamo un'ipotetica JTextArea
      }
};
```

Se vogliamo emettere dei valori durante l'esecuzione del procedimento parallelo invocheremo da `doInBackground` quando necessario il metodo publish passando come argomento dell'invocazione il valore o i valori parziali. Il tipo di questi valori parziali deve corrispondere al secondo tipo parametrico del nostro `SwingWorker`. Supponendo di voler comunicare il numero di byte letti mentre leggiamo il file di testo useremo uno `SwingWorker` così fatto:

```
//pseudo codice
SwingWorker<String, Long> worker = new SwingWorker<String, Long>() {

      @Override protected String doInBackground() {
            File file = new File(...
            Long byteLetti = 0L;
            while( byte letti != lunghezza file)
                  leggi 64 byte
                  byteLetti += 64;
                  publish(byteLetti);
            ..restituisci il testo come String.
      }

      @Override protected void done() {
            String testo = get(); //prendiamo il valore generato parallelamente
            jtextArea.setText(testo); //aggiorniamo un'ipotetica JTextArea
      }
};
```

Per gestire queste notifiche parziali sovrascriviamo il metodo process. Il metodo process è eseguito dall'EDT.

```
//nella nostra sottoclasse di SwingWorker...
@override protected void process(List<Long> data) {
      jprogressBar.setValue(data.intValue()); //aggiorniamo un'ipotetica JProgressBar
}
```

`SwingWorker` ha tre stati di base che scandiscono il suo ciclo vitale: `PENDING`, `STARTED` e `DONE`, costanti di `SwingWorker.StateValue`. Il primo stato è quello di uno `SwingWorker` non ancora avviato, il secondo è di uno `SwingWorker` che sta eseguendo il procedimento in background, il terzo si raggiunge quando lo `SwingWorker` termina l'esecuzione in bakground. I passaggi da uno stato all'altro sono notificati agli ascoltatori di tipo `PropertyChangeListener` registrati presso lo `SwingWorker`. L'ascoltatore `PropertyChangeListener` ha un metodo `propertyChange` con parametro `PropertyChangeEvent`. Il metodo `propertyChange` di un `PropertyChangeListener` connesso ad uno `SwingWorker` è eseguito dall'EDT dunque è possibile usare quell'ascoltatore di eventi ad esempio per aprire una finestra di dialogo quando lo `SwingWorker` inizia il lavoro in background e chiuderla quando lo ha terminato.

Note.

Corretti alcuni svarioni, un grazie a:

banryu79