

## UD11. ENTORN GRÀFIC

- 11.1 Interfície gràfica d'usuari ( GUI )
- 11.2 El paquet Java Swing
- 11.3 Components Swing. Els controls i contenidors
- 11.4 El disseny en un contenidor. Layouts
  - 11.4.1 FlowLayout
  - 11.4.2 BorderLayout
  - 11.4.3 GridLayout
  - 11.4.4 BoxLayout
  - 11.4.5 CardLayout
  - 11.4.6 GridBagLayout
  - 11.4.7 GroupLayout
  - 11.4.8 Creació interfícies complexes
- 11.5 La barra de menús
- 11.6 Eclipse WindowBuilder
- 11.7 Esdeveniments. Concepte i controladors
  - 11.7.1 El patró Model-Vista-Controlador
  - 11.7.2 Control d'esdeveniments
  - 11.7.3 Captura d'esdeveniments
- 11.6 Esdeveniments. Concepte i controladors
  - 11.6.1 El patró Model-Vista-Controlador
  - 11.6.2 Control d'esdeveniments
  - 11.6.3 Captura d'esdeveniments

## 11.1 Interfície gràfica d'usuari ( GUI )

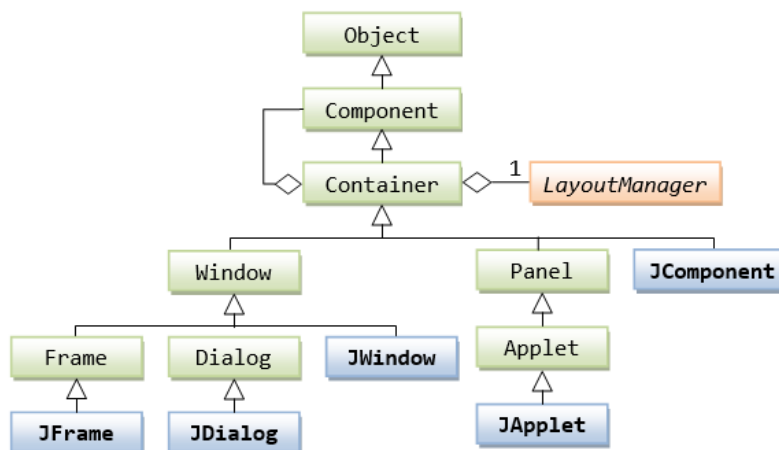
Una GUI (Graphical User Interface) és un tipus d'interface en què els mecanismes que utilitza l'usuari per donar ordres al programa o visualitzar qualsevol informació es basa en la manipulació d'icones en lloc de l'entrada d'ordres textuais.

## 11.2 El paquet Java Swing

El conjunt de classes vinculades a l'entorn gràfic del Java pertanyen a la jerarquia de paquets `javax.swing`. Familiarment, es coneixen com la biblioteca Java Swing o, simplement, Swing. Aquestes són una extensió d'una biblioteca més antiga anomenada AWT (Abstract Windows Toolkit, joc d'eines abstracte de finestres). A l'actualitat, pràcticament cap aplicació usa AWT com a biblioteca gràfica, sempre s'utilitza la biblioteca Swing. Tot i així, AWT sempre està present en el rerefons de qualsevol aplicació basada en Swing.

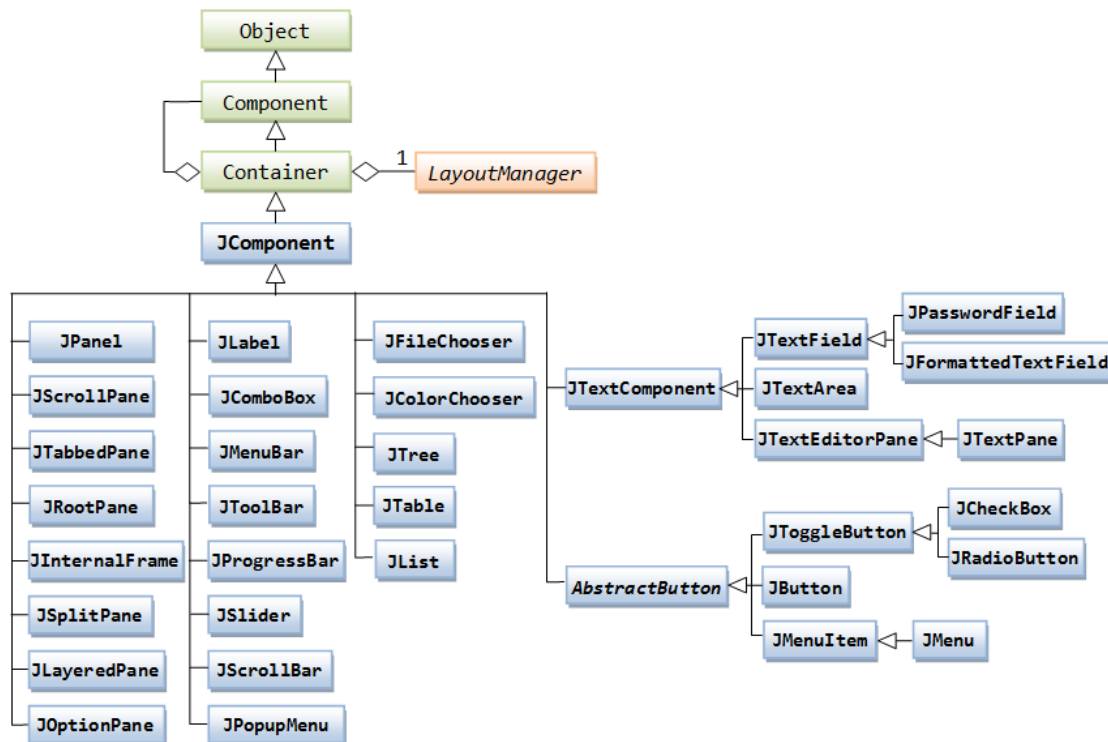
La biblioteca Swing pren la forma d'una jerarquia de classes de mida considerable. Cadascuna de les classes que en formen part representa un element típic d'un entorn gràfic: finestres, botons, formularis, menús, etc. Si es vol incloure algun d'aquests elements en la interface de l'aplicació, caldrà instanciar la classe pertinent.

La jerarquia de nivell superior (JFrame, JDialog, JApplet) és així:



Les classes amb noms que comencen per J en la figura són les incloses a Swing, mentre que la resta pertanyen a AWT o a la biblioteca estàndard de Java. Com es pot veure, una part de la biblioteca original AWT continua present dins Swing en forma de les superclasses `java.awt.Component` i `java.awt.Container` (entre d'altres), ja que Swing és una extensió d'aquesta. Aquestes dues classes especifiquen tots els aspectes genèrics del comportament dels elements d'un entorn gràfic. Per fer-ho, defineixen mètodes que les subclasses poden sobreescrivre d'acord amb les seves particularitats. Això permet al motor gràfic del Java garantir que cada component sempre té implementat tot el conjunt de mètodes necessaris per visualitzar-los correctament i cridar-los polimòrficament.

I la jerarquia dels components es:



Qualsevol element gràfic dins la interface gràfica s'anomena un **component**, ja que tots són un objecte java.awt.Component. Específicament, la classe abstracta **java.awt.Component** defineix totes les **característiques referents a l'aspecte d'un element gràfic**, com la mida, la font del text que conté, si està habilitat o la ubicació en pantalla, com també tot el conjunt d'interaccions que l'element pot rebre (ser pitjat, seleccionat, posar el punter del ratolí a sobre, etc.). En canvi, la classe **java.awt.Container** especifica tot el comportament relatiu a la capacitat d'un element gràfic de **contenir-ne d'altres**.

### 11.3 Components Swing. Els controls i contenidors

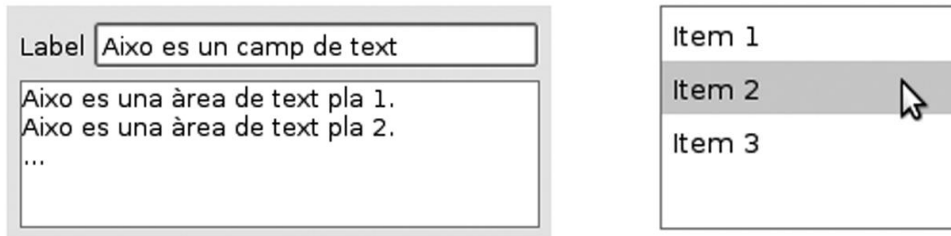
Podem diferenciar dos tipus d'elements gràfics: els controls i els contenidors. Per una banda, els **controls Swing**, que són aquells components amb els quals l'usuari interactua directament.

Exemples de controls són:

- **JButton**: Correspon al botó típic que es pot pitjar per donar ordres a l'aplicació.
- **JToggleButton**: Es tracta d'un botó especial amb ressort, que alterna entre un estat de pitjat o no. Cada cop que es pitja canvia d'estat.
- **JCheckBox** i **JRadioButton**: Representen un selector d'opció amb un text associat. En el primer cas, de forma quadrada tipus checked / unchecked (marcat / no marcat), i en el segon de forma rodona. En el fons, és un cas especial d'un botó amb ressort, però amb una representació gràfica diferent.
- **JLabel**: Correspon a una etiqueta en què es pot mostrar text o una imatge.
- **JTextField**: Correspon a un camp de text, en què l'usuari pot escriure. Només es pot escriure una única línia. No permet variar l'estil dins del text (mida o tipus de la font).
- **JTextArea**: Component similar a l'anterior, per en aquest cas especifica una àrea de text en què és possible escriure lliurement, sense limitació a una única línia. La figura mostra

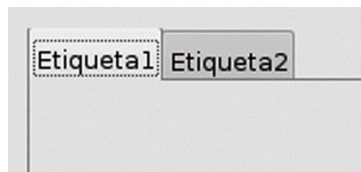
un exemple de JLabel (dalt a l'esquerra), JTextField (dalt a la dreta) i JTextArea (a baix). Mentre que en la primera no es pot escriure, a la resta sí que es pot.

- **JTextPane:** Component de funcionalitat pràcticament idèntica a l'àrea de text, però amb la capacitat afegida de poder editar text amb estil diferent (cursiva, negreta, etc.).
- **JList:** Una llista d'elements o ítems, normalment cadenes de text, d'entre les quals es pot seleccionar un conjunt.



D'altra banda, els **contenidors Swing**, que són aquells components que no tenen una funció directa d'interacció amb l'usuari, que serveixen exclusivament per encabir i organitzar a dintre qualsevol component, tant controls com altres contenidors. Aquests normalment corresponen a components com finestres, panells o barres de menú. Si bé és possible interactuar-hi (per exemple, redimensionar una finestra o obrir un menú), els resultats de l'acció solen quedar dins l'àmbit de l'entorn gràfic, i no se solen usar per donar ordres a la lògica interna del programa. Exemples de contenidors poden ser:

- **JFrame:** La finestra principal de la interface gràfica en una aplicació d'escriptori.
- **JDialog:** És un quadre d'alerta o de diàleg amb l'usuari.
- **JPanel:** Representa una àrea específica de la finestra, amb unes propietats concretes diferenciades: color de fons, vora, etc.
- **JScrollPane:** Idèntic a l'anterior, però si en algun moment la mida de la finestra és massa petita per visualitzar tot el contingut d'aquesta àrea, apareix una barra de desplaçament (scroll).
- **JTabbedPane:** Un conjunt de panells accessible mitjançant etiquetes (tabs) amb una cadena de text, de manera semblant a fitxes de biblioteca. Cada panell està associat a una etiqueta, de manera que quan es pitja, només es visualitza aquest panell.



La relació entre aquests dos tipus de components dins de qualsevol interface gràfica és la següent. Dins un contenidor poden haver tant components com d'altres contenidors. En canvi, un component no pot contenir res, és un element final a l'estructura de la interfície Swing.

Per incloure qualsevol component dins un contenidor, cal cridar el mètode, **add(Component comp)** sobre el contenidor, passant com a paràmetre el component a afegir-hi. Aquest es troba definit en la classe `java.awt.Container` i totes les classes de la biblioteca Swing l'hereten. Addicionalment, hi ha un subtipus especial de contenidors Swing, els anomenats **contenidors d'alt nivell** (top-level containers). Aquests es consideren els contenidors principals de la interfície d'usuari, de manera que l'objecte arrel de l'arbre que conforma el mapa d'objectes de la interfície

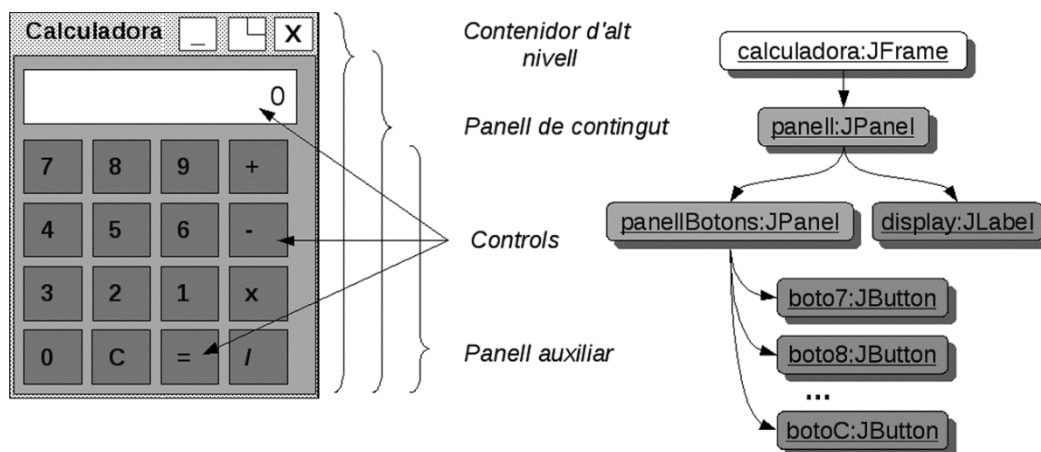
gràfica sempre és un contenidor d'aquest subtipus. Swing en defineix tres, tots subclasses de `java.awt.Window`: **JFrame**, **JApplet** i **JDialog**.

En el cas dels contenidors d'alt nivell, no és possible afegir-hi directament components cridant el mètode `add`. Només és possible la interacció mitjançant el seu **panell de contingut** (content pane), que es pot obtenir amb el mètode `getContentPane()`. Aquest ja és un contenidor normal sobre el qual sí que es pot cridar el mètode `add`.

**Exemple** d'una interface gràfica mínima:

```
import javax.swing.*;
public class GUI {
    public static void main (String args[]) {
        // Crea contenidor principal
        JFrame frame= new JFrame("Nom finestra principal");
        // Crea panell de contingut
        JPanel panell= (JPanel)frame.getContentPane();
        // Crea un control tipus etiqueta
        JLabel display= new JLabel("Hola món");
        // Afegeix el control al panell
        panell.add(display);
        // Estableix una grandària adequada al frame
        frame.setSize(400, 200);
        // Centra el frame a la pantalla
        frame.setLocationRelativeTo(null);
        // Fa visible la finestra
        frame.setVisible(true);
    }
}
```

En un exemple un poc més complex, la següent figura mostra una interface gràfica concreta en forma de mapa d'objectes i la seva classificació per tipus.



**Exemple** de codi per representar la calculadora. En aquest exemple tenim un `JFrame` que serà la finestra del programa i dins un `JPanel` principal (panell) que contindrà `JLabel` (display) que seria el display de la calculadora i un altra `JPanel` (panellBotons) amb tots els botons numèrics i de les operacions.

```
//Contenidor d'alt nivell: finestra principal
JFrame calculadora = new JFrame("Calculadora");
```

```

//Panell de contingut
Container panell = calculadora.getContentPane();
...
//Display de la calculadora
JLabel display = new JLabel();
display.setHorizontalAlignment(SwingConstants.RIGHT);
panell.add(display);
...
//Panell auxiliar on posar els botons
JPanel panellBotons = new JPanel();
JButton boto7 = new JButton("7");
JButton boto8 = new JButton("8");
...
JButton botoC = new JButton("C");
//Afegir botons a panell auxiliar
panellBotons.add(boto7);
panellBotons.add(boto8);
...
panellBotons.add(botoC);
//Afegir panell auxiliar de botons a interface
panell.add(panellBotons);
calculadora.setVisible();

```

**Exercici 11.1:** Completar l'exemple de la calculadora amb tots els botons fins que surti el mateix disseny que la imatge anterior. Què passa si canviem la mida de la finestra de l'aplicació?

## 11.4 El disseny en un contenidor. Layouts

<https://docs.oracle.com/javase/tutorial/uiswing/layout/index.html>

El mètode usat per afegir controls a un contenidor, `add (Component comp)`, i les seves sobrecàrregues no disposen de cap paràmetre vinculat a les **coordenades** en què es pugui ubicar el component, només s'indica el component que s'ha d'afegir. El motiu és el mecanisme que el Java aporta per solucionar la circumstància que és impossible establir per endavant els paràmetres sota els quals s'executa el navegador. En una aplicació Swing (o AWT), en realitat, no és possible establir la ubicació i mides exactes de cada component dins la interface gràfica. El que es fa és, per a cada contenidor Swing (principalment, els `JPanel` i `JFrame`), **especificar una política d'ubicació de components**, de manera que el motor gràfic del Java escull automàticament la millor opció d'acord amb les dimensions reals de la finestra principal.

Cada cop que la finestra principal canvia de dimensions, els components es **reubiquen i redimensionen dinàmicament**. Aquestes polítiques no les ha de generar el desenvolupador -el Swing ja defineix un conjunt predeterminat disponible-, entre les quals tan sols cal triar-ne una per a cada contenidor en la interface. Cada una és el que s'anomena un layout.

Un **layout** és una política d'ubicació i dimensionament de components, de manera que el motor gràfic del Java escull automàticament on s'ha de visualitzar i quina ha de ser la seva mida.

Concretament, els layouts disponibles a Swing són totes les classes que implementen la interface **java.awt.LayoutManager**:

- BorderLayout
- BoxLayout

- CardLayout
- FlowLayout
- GridBagLayout
- GridLayout
- GroupLayout
- SpringLayout

Sota el sistema de layouts, tots els components tenen com a mida per defecte la mínima necessària per encabir tot el contingut. En principi, les seves dimensions no es poden establir de manera estàtica.

Per aplicar un layout a un contenidor es pot fer de dues maneres:

- Amb el constructor: `JPanel panell= new JPanel( new FlowLayout( ) );`
- Amb un mètode: `panell.setLayout( new FlowLayout( ) );`

#### 11.4.1 FlowLayout

El FlowLayout és el layout que hi ha per defecte en tots els contenidors si no se n'assigna cap altre mitjançant el mètode setLayout. S'anomena així perquè es considera que els elements "flueixen de manera natural" dins el contenidor. La política que defineix és que tots els components es mantenen en la seva mida per defecte i es van ubicant per línies, **d'esquerra a dreta i de dalt a baix, centrats horitzontalment**. Si en un moment donat un component no cap en la línia actual, s'ubica en la línia immediatament inferior. **L'ordre** en què s'afegeixen al contenidor és el mateix en què s'ha cridat el mètode add.

El seu constructor és public FlowLayout().



Codi:

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

public class FlowLayoutDemo {

    public static void main(String[] args) {

        JFrame frame = new JFrame("FlowLayoutDemo");
        JPanel pane = (JPanel) frame.getContentPane();
        pane.setLayout(new FlowLayout());

        JButton button = new JButton("Button 1");
        pane.add(button);
        button = new JButton("Button 2");
        pane.add(button);
        button = new JButton("Button 3");
        pane.add(button);
    }
}
```

```

        button = new JButton("Longer Button 4");
        pane.add(button);
        button = new JButton("Button 5");
        pane.add(button);

        frame.pack();
        frame.setVisible(true);
    }
}

```

**Exercici 11.2:** Reproduir el disseny de la finestra següent utilitzant FlowLayout. Els elements que apareixen son JLabel, JTextField, JRadioButton i una JComboBox. Els dos radio buttons han d'estar inclosos dins un ButtonGroup per a que només se'n pugui seleccionar un dels dos.



#### 11.4.2 BorderLayout

En un component en què s'aplica el BorderLayout es defineixen cinc zones diferenciades: **nord, sud, est, oest i centre**, que corresponen als punts cardinals del contenidor. En cada zona només es pot ubicar **un component**, que l'ocupa totalment i mai no varia de posició. Els components de les zones nord i sud ocupen el màxim espai possible horitzontal i el mínim indispensable en vertical. Per a les zones est i oest es dona el cas invers. La zona central és l'única que varia de mida quan es redimensiona el contenidor.

El seu constructor és public BorderLayout().

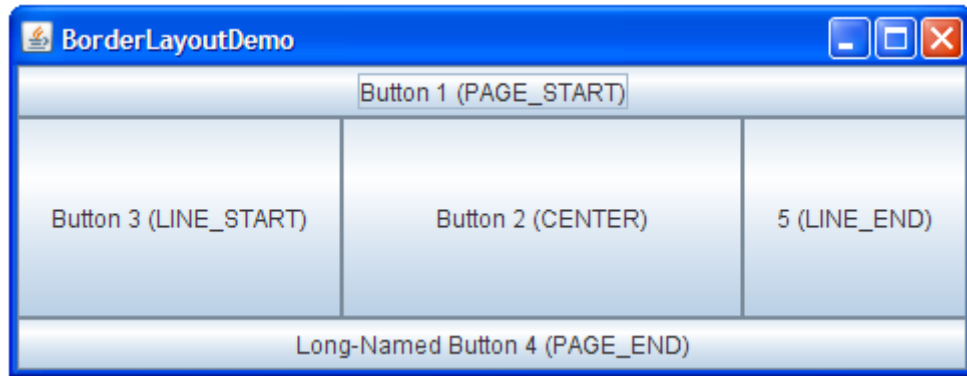
Aquest layout és una excepció respecte a la resta, ja que sobre un contenidor que l'usa es pot cridar una sobrecàrrega del mètode add en què es passa un paràmetre addicional que indica la zona en què es pot ubicar el component:

```
public void add(Component comp, Object constraints)
```

La classe BorderLayout defineix diverses constants que serveixen per indicar cada zona en el paràmetre constraints:

- BorderLayout.NORTH o BorderLayout.PAGE\_START
- BorderLayout.SOUTH o BorderLayout.PAGE\_END
- BorderLayout.EAST o BorderLayout.LINE\_END
- BorderLayout.WEST o BorderLayout.LINE\_START
- BorderLayout.CENTER





Codi:

```
import javax.swing.*;
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.FlowLayout;

public class BorderLayoutDemo {

    public static void main(String[] args) {

        JFrame frame = new JFrame("BorderLayoutDemo");
        JPanel pane = (JPanel) frame.getContentPane();
        pane.setLayout(new BorderLayout());

        JButton button = new JButton("Button 1 (PAGE_START)");
        pane.add(button, BorderLayout.PAGE_START);

        //Make the center component big, since that's the
        //typical usage of BorderLayout.
        button = new JButton("Button 2 (CENTER)");
        button.setPreferredSize(new Dimension(200, 100));
        pane.add(button, BorderLayout.CENTER);

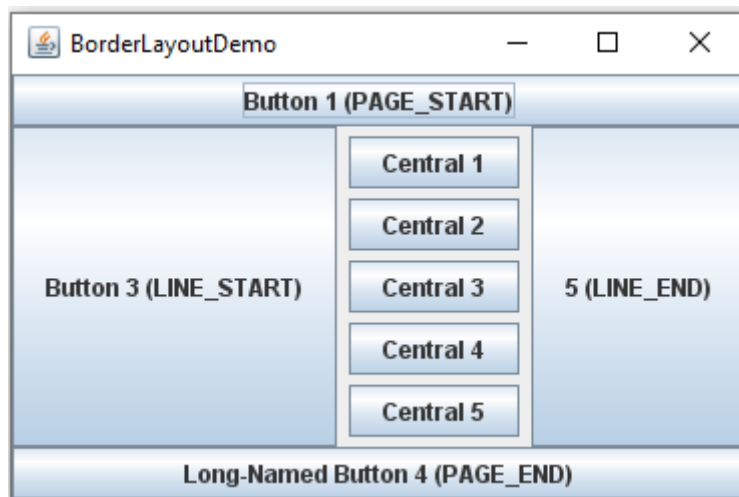
        button = new JButton("Button 3 (LINE_START)");
        pane.add(button, BorderLayout.LINE_START);

        button = new JButton("Long-Named Button 4 (PAGE_END)");
        pane.add(button, BorderLayout.PAGE_END);

        button = new JButton("5 (LINE_END)");
        pane.add(button, BorderLayout.LINE_END);

        frame.pack();
        frame.setVisible(true);
    }
}
```

**Exercici 11.3:** Dins de cadascuna de les zones d'un BorderLayout hi pot ver un o més elements agrupats dins un contenidor. Modificar el contingut de la zona central de l'exemple anterior per a que apareguin 5 botons diferents com es mostra a la imatge següent:



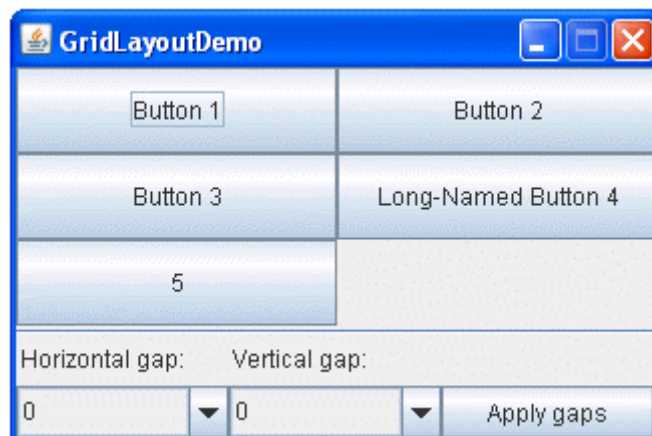
### 11.4.3 GridLayout

El GridLayout organitza el component com una **matriu** amb cel·les de mida idèntica. En cada cel·la només hi pot haver **un component**, que ocupa tot l'espai disponible independentment de quina en seria la mida per defecte. Els components s'ubiquen per files, d'esquerra a dreta en el mateix ordre en què es crida el mètode add.

No es poden saltar cel·les en anar cridant add. Si es vol deixar una cel·la en blanc, n'hi ha prou d'afegir un panell buit.

El seu constructor és public GridLayout(int rows, int cols).

El paràmetre rows indica el nombre de files i cols el nombre de columnes.



Codi:

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

public class GridLayoutDemo {

    public static void main(String[] args) {

        JFrame frame = new JFrame("FlowLayoutDemo");
```

```

JPanel pane = (JPanel) frame.getContentPane();
pane.setLayout(new GridLayout(0,2));

JButton button = new JButton("Button 1");
pane.add(button);
button = new JButton("Button 2");
pane.add(button);
button = new JButton("Button 3");
pane.add(button);
button = new JButton("Longer Button 4");
pane.add(button);
button = new JButton("Button 5");
pane.add(button);

frame.pack();
frame.setVisible(true);
}
}

```

#### 11.4.4 BoxLayout

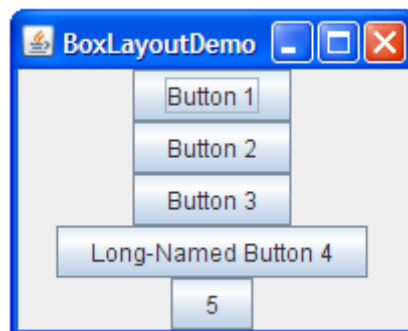
El **BoxLayout** s'acostuma a aplicar sobre un contenidor específic, el **Box**, que ja porta incorporat aquest layout per defecte en ser instanciat i no es pot canviar. Els components que conté mantenen la mida per defecte i s'ubiquen en una sola línia horitzontal o vertical, centrada.

Les instàncies de la classe **Box** no es generen mitjançant un mètode constructor, en el seu lloc s'utilitza un dels dos mètodes estàtics definits en la mateixa classe **Box**. El mètode a usar depèn de si es volen alinear els components horitzontalment o verticalment:

```

Box alineavel = Box.createVerticalBox();
Box alineahoritzontal = Box.createHorizontalBox();

```

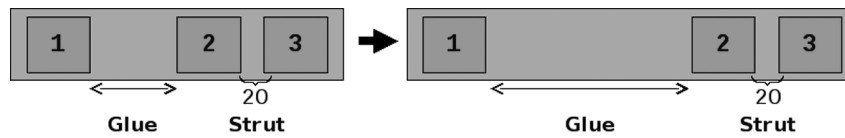


Una altra de les particularitats dels contenidors **Box** és la seva capacitat d'incloure, usant el mètode **add**, dos components especials que cap altre tipus de layout pot usar: les **Struts** i les **Glues**, verticals i horitzontals. Donada una instància de **Box**, només es poden afegir **Glues** o **Struts** de la seva mateixa alineació (vertical o horitzontal). Novament, aquests components s'instancien mitjançant mètodes estàtics definits en la classe **Box**:

- Component **Box.createVerticalGlue()**
- Component **Box.createVerticalStrut(int height)**
- Component **Box.createHorizontalGlue()**

- Component `Box.createHorizontalStrut(int width)`

Les Struts són espais en blanc d'un nombre concret de píxels (indicat amb els paràmetres `height` o `width`). Independentment de les dimensions de la Box, aquest espai es manté sempre invariable. Les Glue són exactament el contrari al que podria donar a entendre la seva traducció, "cola". Dos components separats per una Glue sempre s'ubiquen el més separat possible segons l'espai que hi ha. Es pot considerar que es comporta com una molla en expansió.



Codi:

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

public class BoxLayoutDemo {

    public static void main(String[] args) {

        JFrame frame = new JFrame("BoxLayoutDemo");
        JPanel pane = (JPanel) frame.getContentPane();

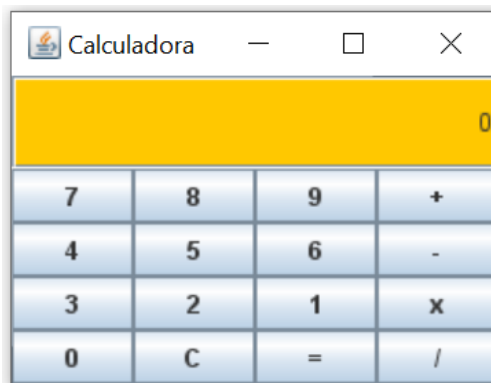
        Box boxContainer = Box.createVerticalBox();

        JButton button = new JButton("Button 1");
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        boxContainer.add(button);
        button = new JButton("Button 2");
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        boxContainer.add(button);
        boxContainer.add(Box.createVerticalGlue());
        button = new JButton("Button 3");
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        boxContainer.add(button);
        button = new JButton("Longer Button 4");
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        boxContainer.add(Box.createVerticalStrut(20));
        boxContainer.add(button);
        button = new JButton("Button 5");
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        boxContainer.add(button);

        pane.add(boxContainer);

        frame.pack();
        frame.setVisible(true);
    }
}
```

**Exercici 11.4:** Aprofitar els diferents Layouts que ja coneixeu per reproduir la calculadora i que es mantengui la mateixa distribució dels components quan es redimensioni la finestra. Canviau també el color del fons del display.



#### 11.4.5 CardLayout

Aquest layout organitza els components com una pila de cartes, on tots estan ubicats però a cada moment només se'n pot visualitzar un, el qual ocupa el màxim espai possible. L'ordre amb què s'afegeixen a la pila és el mateix en què es crida el mètode `add` sobre el contenidor.

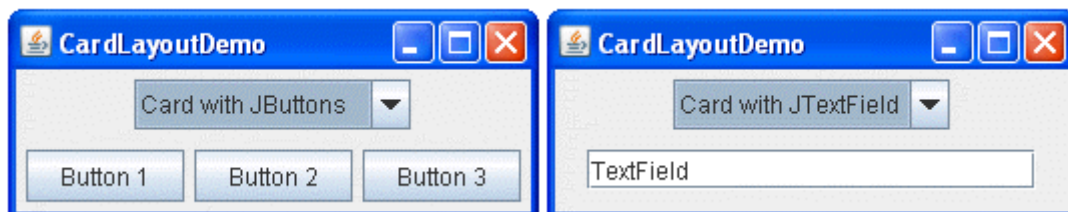
El seu constructor és `public CardLayout()`.

El `CardLayout` se sol usar per sobreposar panells.

Per anar canviant entre els diferents components de la pila, la classe `CardLayout` disposa d'un seguit de mètodes. En tots el paràmetre es refereix al contenidor en què s'ha aplicat el layout:

- **`public void first(Container parent)`**: Salta al primer component afegit.
- **`public void last(Container parent)`**: Salta al darrer component afegit.
- **`public void next(Container parent)`**: Salta al component afegit a continuació del visualitzat actualment.
- **`public void previous(Container parent)`**: Salta al component afegit tot just abans del visualitzat actualment.

La classe `CardLayout` permet implementar una àrea que conté diferents components en diferents moments. Sovint, un `CardLayout` està controlat per un quadre combinat, amb l'estat del quadre combinat que determina quin panell (grup de components) mostra el `CardLayout`.



Les passes a realitzar per crear una aplicació amb aquest layout son les següents:

1. Crear un panell amb `JPanel` i establir el layout a `CardLayout`

```
JPanel cards = new JPanel(new CardLayout());
```

2. Afegir tots els panells que volguem posar dins aquest card layout amb una etiqueta identificadora.

```
cards.add (panell1, "primer panell");  
cards.add (panell2, "segon panell");  
cards.add (panell3, "tercer panell");
```

3. Incorporar el panell amb card layout al panell principal del frame.

```
frame.getContentPane().add(cards);
```

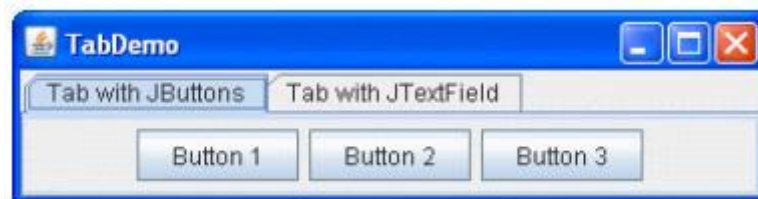
4. Crear un objecte CardLayout per poder manejar quin panell es mostra en primer pla.

```
CardLayout cl = (CardLayout)(cards.getLayout());
```

5. La manera de canviar d'un panell a l'altre és utilitzant el mètode show d'aquest objecte.

```
cl.show(cards,"primer panell");
```

Una alternativa a utilitzar CardLayout és utilitzar un panell amb pestanyes, que ofereix una funcionalitat similar però amb una GUI predefinida.



#### 11.4.6 GridBagLayout

El GridBagLayout és el més versàtil i complex de tots els layouts. Justament per la seva complexitat, ens limitarem a donar una idea general del seu funcionament, ja que una explicació detallada seria molt extensa.

Aquest layout és conceptualment similar al GridLayout, i divideix el contenidor en una **matriu de cel·les**. La particularitat és que en aquest cas les diferents files i columnes **poden ser de mida desigual** i els components inclosos poden ocupar **diverses cel·les contigües**, tant en diferents files com columnes. Els components sempre ocupen tot l'espai possible de les cel·les assignades.

El seu constructor és públic GridBagLayout().

Totes les propietats especials de les cel·les es defineixen amb l'ajut de la classe auxiliar GridBagConstraints. Les crides al mètode add al contenidor que usa aquest layout contenen tant el component a afegir com una instància d'aquesta classe:

```
add(Component comp, GridBagConstraints constraints)
```

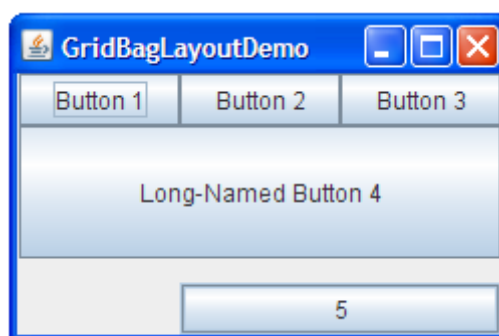
La classe GridBagConstraints té un conjunt de propietats amb les quals definim **com s'ubica** l'element afegit dins la graella i **quantes caselles ocupa** per cada eix. Algunes de les més significatives són:

- **gridx, gridy.** Especifica la fila i columna en l'extrem superior dret del component. La primera fila i columna de la graella són les posicions zero. Si no s'especifica, l'element s'ubicarà tot just després de l'afegit tot just abans.
- **gridwidth, gridheight.** Indica el nombre de cel·les horitzontals o verticals que ocupa el component. Per defecte és 1. Si s'usa la constant GridBagConstraints.REMAINDER, significa que es volen ocupar totes les files o columnes restants fins al final.
- **ipadx, ipady.** Especifica un farcit extra al voltant del component, internament, en píxels. Per tant, el component mai serà més petit que aquest valor. Per defecte és zero.
- **insets.** Similar al cas anterior, però el farcit és extern, per la qual cosa es crea un cert espai de separació entre aquest component i la resta que l'envolten. Per defecte és zero.
- **anchor.** Usat quan el component és més petit que la cel·la, de manera que indica a quin extrem d'aquesta s'ha d'ajustar. GridBagConstraints especifica un seguit de constants per a aquest valor: CENTER (per defecte), PAGE\_START, PAGE\_END, LINE\_START, LINE\_END, FIRST\_LINE\_START, FIRST\_LINE\_END, LAST\_LINE\_END, i LAST\_LINE\_START. En la següent imatge es veu com s'interpreten aquests valor dins d'un contenidor:

FIRST_LINE_START	PAGE_START	FIRST_LINE_END
LINE_START	CENTER	LINE_END
LAST_LINE_START	PAGE_END	LAST_LINE_END

Tot i que res no impedeix reusar objectes GridBagConstraints en crides successives del mètode add per als casos de components amb característiques idèntiques, és millor usar instàncies diferents per evitar confusions.

La següent imatge mostra una finestra que empra GridBagLayout.



En aquesta imatge es veu la divisió en caselles de la finestra anterior. Com es pot veure hi ha tres fileres i tres columnes. El botó de la segona fila ocupa les tres columnes, mentre que el botó de la tercera fila ocupa les dues columnes de la dreta. Aquest botó 5 és un exemple d'un element més petit que les cel·les que ocupa, i per tant ha estat afegit amb un anchor de LAST\_LINE\_END



Codi:

```
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class GridBagLayoutDemo {

    public static void main(String[] args) {
        JFrame frame = new JFrame("GridBagLayoutDemo");
        frame.setSize(300,300);
        JPanel pane = (JPanel) frame.getContentPane();

        pane.setLayout(new GridBagLayout());

        GridBagConstraints constraints = new GridBagConstraints();

        JButton button = new JButton("Button 1");
        constraints.fill = GridBagConstraints.HORIZONTAL;
        constraints.weightx=0.5;
        constraints.gridx = 0;
        constraints.gridy = 0;
        pane.add(button,constraints);

        button = new JButton("Button 2");
        constraints.fill = GridBagConstraints.HORIZONTAL;
        constraints.weightx = 0.5;
        constraints.gridx = 1;
        constraints.gridy = 0;
        pane.add(button, constraints);

        button = new JButton("Button 3");
        constraints.fill = GridBagConstraints.HORIZONTAL;
        constraints.weightx = 0.5;
        constraints.gridx = 2;
        constraints.gridy = 0;
        pane.add(button, constraints);

        button = new JButton("Long-Named Button 4");
        constraints.fill = GridBagConstraints.HORIZONTAL;
```



```

constraints.ipady = 40;           //make this component tall
constraints.weightx = 0.0;
constraints.gridwidth = 3;
constraints.gridx = 0;
constraints.gridy = 1;
pane.add(button, constraints);

button = new JButton("5");
constraints.fill = GridBagConstraints.HORIZONTAL;
constraints.ipady = 0;           //reset to default
constraints.weighty = 1.0;       //request any extra vertical
space
constraints.anchor = GridBagConstraints.PAGE_END; //bottom of
space

constraints.insets = new Insets(10,0,0,0); //top padding
constraints.gridx = 1;           //aligned with button 2
constraints.gridwidth = 2;       //2 columns wide
constraints.gridy = 2;           //third row
pane.add(button, constraints);

frame.setVisible(true);
}
}

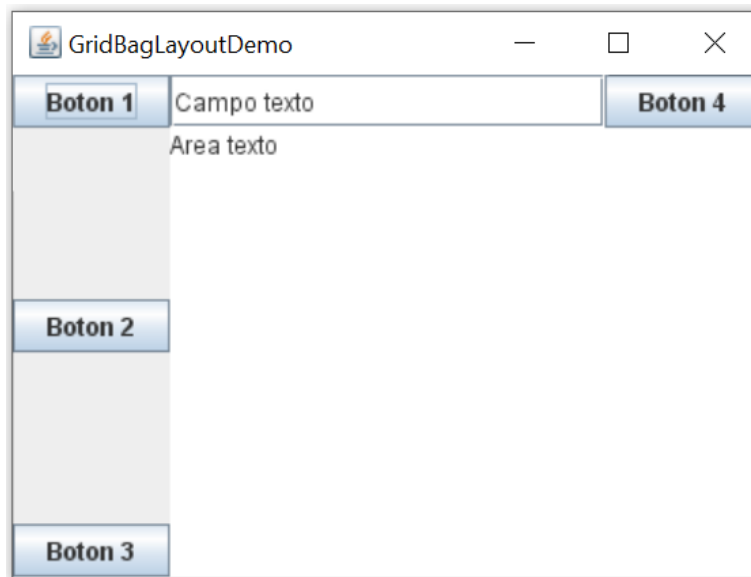
```

**Exercici 11.5:** Reproduir la finestra següent emprant GridBagLayout i seguint els passos que trobareu en aquest enllaç:

<https://old.chuidiang.org/java/layout/GridBagLayout/GridBagLayout.php>



En base a aquesta finestra modifiqui els components i reubicar-los d'aquesta manera:



#### 11.4.7 GroupLayout

Des de la versió 1.6, Java incorpora el layout GroupLayout, molt orientat al desenvolupament automàtic d'interfícies gràfiques mitjançant eines auxiliars. Res no impedeix usar-lo manualment, tot i que, com el GridBagLayout, té un cert grau de complicació.

La filosofia d'aquest layout és desplegar els elements al llarg dels dos eixos de coordenades (vertical i horitzontal). Al llarg d'aquests eixos, els elements s'agrupen mitjançant **grups jeràrquics**, de manera que donat un grup, hi pot haver continguts, components, altres grups, o espais en blanc. L'addició de grups és el que permet fer una organització jeràrquica, mentre que els espais són espais en blanc, són separacions buides entre elements. Els grups estan representats per la classe GroupLayout.Group, en la qual hi ha un conjunt de mètodes que permeten afegir-hi elements:

- addComponent (Component comp)
- addGroup(GroupLayout.Group group)
- addGap(int size)

Al mateix temps, hi ha dos tipus de grups: seqüencials i paral·lels. En el primer cas els elements s'ubiquen un darrera l'altre al llarg de l'eix de coordenades corresponent, mentre que en el segon cas s'ubiquen en paral·lel. De manera similar a les Box, hi ha una classe concreta per a cada subtipus: SequentialGroup i ParallelGroup. Per crear algun d'aquest grups cal cridar els mètodes definits en la classe GroupLayout:

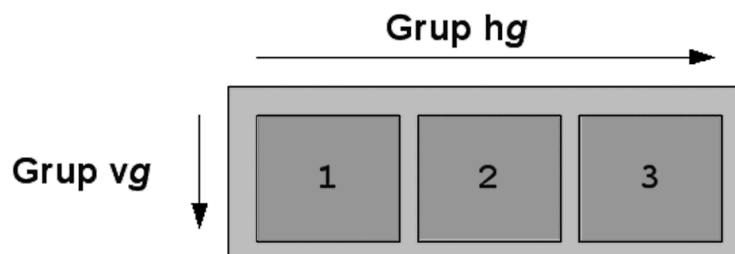
- GroupLayout.Group createSequentialGroup()
- GroupLayout.Group createParallelGroup()

La clau d'aquest layout està en el fet que almenys hi ha d'haver **un grup associat a l'eix vertical i un altre a l'horitzontal**, i tots els components **han de pertànyer a dos grups**, un de l'eix vertical i un de l'horitzontal. A partir del grup on estan associats, se'n pot calcular la ubicació.

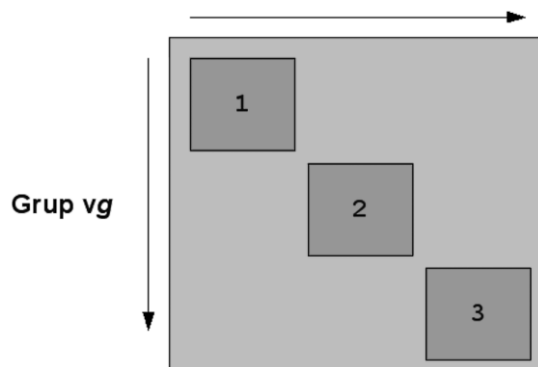
La millor manera d'entendre com es fa aquest càlcul d'ubicació és mitjançant un exemple. La figura mostra un GroupLayout en què hi ha un únic grup paral·lel associat a l'eix vertical i un de seqüencial a l'eix horitzontal. Cada component visualitzat (1, 2 i 3) pertany a tots dos grups i s'ha

afegit al grup en ordre de numeració incremental. Els espais entre elements en realitat no existiren, s'han posat en la imatge per fer-la més clara.

```
GroupLayout layout = new GroupLayout(panel)
GroupLayout.SequentialLayout hg = layout.createSequentialGroup();
GroupLayout.ParallelLayout vg = layout.createParallelGroup();
JLabel l1 = new JLabel("1");
JLabel l2 = new JLabel("2");
JLabel l3 = new JLabel("3");
hg.addComponent(l1);
hg.addComponent(l2);
hg.addComponent(l3);
vg.addComponent(l1);
vg.addComponent(l2);
vg.addComponent(l3);
layout.setHorizontalGroup(hg);
```

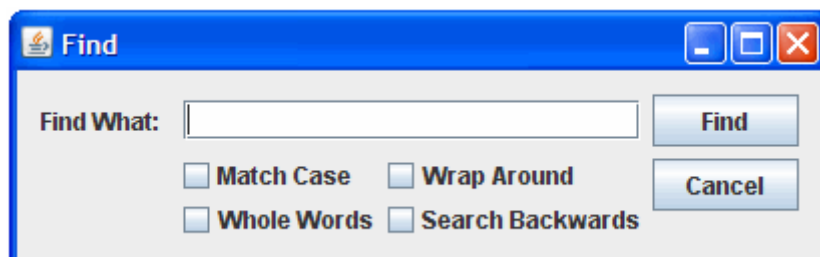


En contraposició, en la figura anterior es mostra com s'ubicarien els components si els dos grups fossin seqüencials en els dos eixos. Els elements s'han afegit als grups igual que en el cas de la figura, però com es pot apreciar, en lloc d'ubicar-se paral·lelament en l'eix vertical avancen una posició perquè són un grup seqüencial.

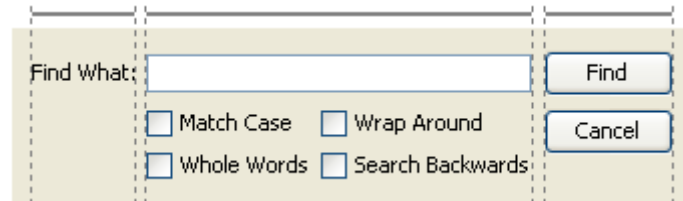


Una de les particularitats més importants del GroupLayout és que permet crear composicions d'elements sense haver de dependre de subpanells auxiliars. Els grups ja fan aquest paper.

Vegem com podríem construir una finestra com aquesta utilitzant GroupLayout:



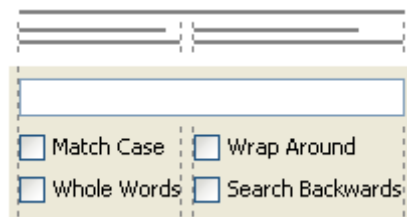
Examinant la **dimensió horitzontal** d'esquerra a dreta veiem tres grups en seqüència. El primer de fet no és un grup, només una etiqueta (label). El segon és un grup que conté el camp de text (textfield) i les checkboxes. El tercer grup conté els dos botons.



Podem començar a descriure en codi aquest grup seqüencial. Fixau-vos que GroupLayout.Alignment.LEADING correspon a un alineament a l'esquerra en la dimensió horitzontal. Fixau-vos també que no especificarem gaps, assumint que gap auto-insertion està activat.

```
layout.setHorizontalGroup(layout.createSequentialGroup()  
    .addComponent(label)  
  
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING))  
  
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING))  
);
```

Estudiem ara el grup central. Hi ha un camp de text en seqüència amb dos grups que cadascun conté dos check boxes.



Això en codi es pot representar així:

```
layout.setHorizontalGroup(layout.createSequentialGroup()  
    .addComponent(label)  
  
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)  
        .addComponent(textField)  
        .addGroup(layout.createSequentialGroup()  
  
            .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)  
                .addComponent(caseCheckBox)  
                .addComponent(wholeCheckBox))  
  
            .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)  
                .addComponent(wrapCheckBox)  
                .addComponent(backCheckBox)))  
  
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING))  
);
```

El text field s'ajusta automàticament sense necessitat de fer res més.

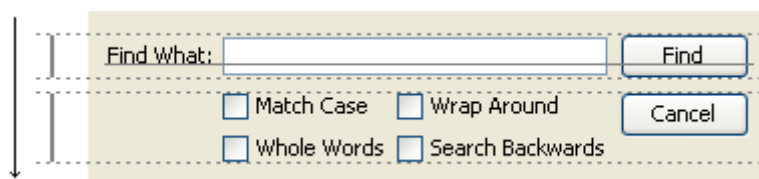
El darrer grup és trivial. Consta només dels dos botons:

```
layout.setHorizontalGroup(layout.createSequentialGroup()  
    .addComponent(label)  
  
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)  
        .addComponent(textField)  
        .addGroup(layout.createSequentialGroup()  
  
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)  
        .addComponent(caseCheckBox)  
        .addComponent(wholeCheckBox))  
  
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)  
        .addComponent(wrapCheckBox)  
        .addComponent(backCheckBox)))  
  
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)  
        .addComponent(findButton)  
        .addComponent(cancelButton))  
  
    );
```

Finalment volem que els dos botons siguin sempre de la mateixa mida independentment de la mida de la finestra:

```
layout.linkSize(SwingConstants.HORIZONTAL, findButton, cancelButton);
```

En la **dimensió vertical** examinem el layout de dalt a baix. Volem tots els components de la primera fila alineats a la baseline. Així, en l'eix vertical hi ha una seqüència entre el primer grup de la baseline seguit d'un grup amb la resta de components



Anam a començar a escriure el codi amb l'estructura bàsica:

```
layout.setVerticalGroup(layout.createSequentialGroup()  
  
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE))  
  
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING))  
  
    );
```

Podem omplir la baseline tot seguit:

```
layout.setVerticalGroup(layout.createSequentialGroup()  
  
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE))  
  
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING))  
  
    );
```

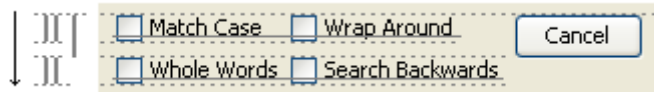
```

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
            .addComponent(label)
            .addComponent(textField)
            .addComponent(findButton))

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
            );

```

Si ens fixem ara en el grup de baix, el botó cancel no comparteix la baseline amb les check boxes. Així doncs el segon grup paral·lel compren el botó i un grup seqüencial de dos grups de baseline amb check boxes:



```

layout.setVerticalGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
            .addComponent(label)
            .addComponent(textField)
            .addComponent(findButton))

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                    .addComponent(caseCheckBox)
                    .addComponent(wrapCheckBox))

.addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                    .addComponent(wholeCheckBox)
                    .addComponent(backCheckBox)))
            .addComponent(cancelButton))
            );

```

Ja tenim complet el layout, incloent el comportament quan es modifiqui la mida de la finestra.

Aquí hi ha el codi sencer:

```

import javax.swing.*;
import static javax.swing.GroupLayout.Alignment.*;

public class Find {

    public static void main(String[] args) {

        JFrame frame = new JFrame ("Find");
        JPanel pane = (JPanel) frame.getContentPane();

        JLabel label = new JLabel("Find What:");

```

```

JTextField textField = new JTextField();
JCheckBox caseCheckBox = new JCheckBox("Match Case");
JCheckBox wrapCheckBox = new JCheckBox("Wrap Around");
JCheckBox wholeCheckBox = new JCheckBox("Whole Words");
JCheckBox backCheckBox = new JCheckBox("Search Backwards");
JButton findButton = new JButton("Find");
JButton cancelButton = new JButton("Cancel");

GroupLayout layout = new GroupLayout(pane);
pane.setLayout(layout);
layout.setAutoCreateGaps(true);
layout.setAutoCreateContainerGaps(true);

layout.setHorizontalGroup(layout.createSequentialGroup()
    .addComponent(label)
    .addGroup(layout.createParallelGroup(LEADING)
        .addComponent(textField)
        .addGroup(layout.createSequentialGroup()
            .addGroup(layout.createParallelGroup(LEADING)
                .addComponent(caseCheckBox)
                .addComponent(wholeCheckBox))
            .addGroup(layout.createParallelGroup(LEADING)
                .addComponent(wrapCheckBox)
                .addComponent(backCheckBox))))
    .addGroup(layout.createParallelGroup(LEADING)
        .addComponent(findButton)
        .addComponent(cancelButton))
    );

layout.linkSize(SwingConstants.HORIZONTAL, findButton,
cancelButton);

layout.setVerticalGroup(layout.createSequentialGroup()
    .addGroup(layout.createParallelGroup(BASELINE)
        .addComponent(label)
        .addComponent(textField)
        .addComponent(findButton))
    .addGroup(layout.createParallelGroup(LEADING)
        .addGroup(layout.createSequentialGroup()
            .addGroup(layout.createParallelGroup(BASELINE)
                .addComponent(caseCheckBox)
                .addComponent(wrapCheckBox))
            .addGroup(layout.createParallelGroup(BASELINE)
                .addComponent(wholeCheckBox)
                .addComponent(backCheckBox)))
        .addComponent(cancelButton))
    );

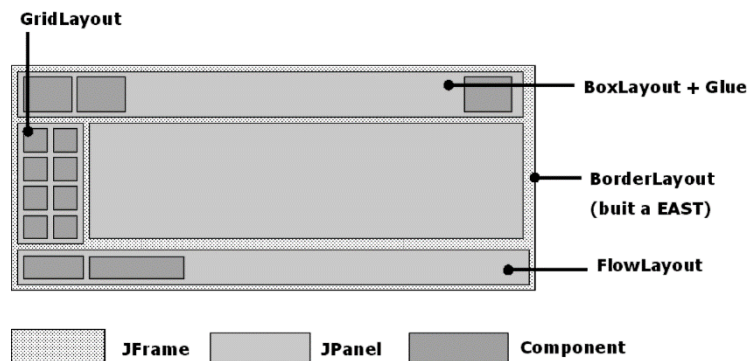
frame.pack();
frame.setVisible(true);
}
}

```

### 11.4.8 Creació interfícies complexes

A partir de l'estudi dels diferents layouts existents, es pot arribar a la conclusió que molts, per si mateixos, no són suficients per generar una interface gràfica d'una certa complexitat. Aquest fet és especialment evident en casos com el BorderLayout, en què en cada zona només hi pot haver un component i, per tant, només hi podria haver cinc components en tota la interface.

Això es deu al fet que, normalment, un entorn gràfic no es pot resoldre amb un únic layout, sinó que cal la combinació de diferents layouts usats en diversos panells dins la finestra principal, tal com mostra la figura. De fet, els layouts és el que dóna sentit a l'existència de la classe JPanel, un contenidor que defineix àrees dins la finestra principal.



El fet d'haver d'estudiar com es combinen diferents layouts per assolir el resultat desitjat és un dels factors que donen una certa complexitat al disseny d'una interface gràfica i el que fa recomanable l'ús d'un IDE.

A continuació es mostra un exemple de com es poden combinar diferents layouts per generar la calculadora que es mostra en la figura. Concretament, es combina un BorderLayout amb un GridLayout per aconseguir l'efecte final.

```
//Contenidor d'alt nivell: finestra principal
JFrame frame = new JFrame("Calculadora");
JPanel panell = calculadora.getContentPane();
panell.setLayout(new BorderLayout());
...
JLabel display = new JLabel();
display.setHorizontalAlignment(SwingConstants.RIGHT);
...
//Display de la calculadora a la zona superior
panell.add(display, BorderLayout.NORTH);
...
JPanel panellBotons = new JPanel();
panellBotons.setLayout(new GridLayout(4, 4));
JButton boto7 = new JButton("7");
...
panell.add(boto7);
...
//Panell de botons a la zona central
panell.add(panellBotons, BorderLayout.CENTER);
...
```

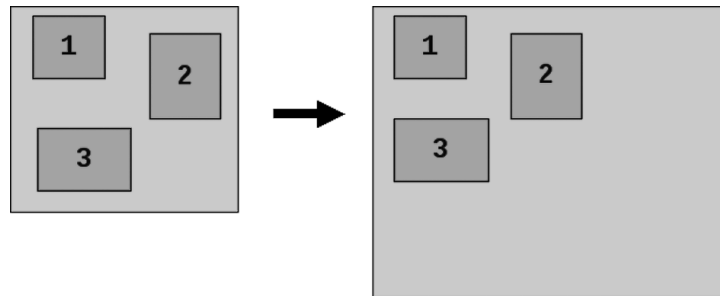
A l'hora de generar interfaces amb composicions complexes, en què cal ubicar els components de manera molt especial, hi ha una alternativa molt útil: els layouts de posicionament absolut (o,



simplement, layouts absoluts). Algunes biblioteques externes a les estàndard del Java proporcionen un tipus especial de layout que permet el posicionament absolut dels components. D'aquesta manera, es pot indicar la ubicació i mides exactes de cada component, que mai varia al llarg de l'execució de l'aplicació independentment que es redimensioni el contenidor.

El mètode **setResizable** serveix per habilitar/deshabilitar la capacitat de redimensionar una finestra. Evitant que es pugui redimensionar la finestra podem eludir el problema del redimensionat del layout absolut.

Si bé aquest tipus de layout pot estalviar molta feina, cal ser molt conscient de les implicacions del seu ús. Per una banda, al redimensionar el contenidor d'alt nivell, tot segueix igual, i per tant, tot el nou espai extra queda buit, tal com mostra la figura:



D'altra banda, mai no s'ha d'oblidar que els layouts absoluts són aliens a les biblioteques estàndard del Java. Cal incloure'ls com a classes addicionals dins de les aplicacions que els usen en desplegar-les, o aquestes no funcionaran.

**Exercici 11.6:** Crear dues finestres que ens permetin introduir les dades dels alumnes i professors de l'exercici 6.5. Més endavant afegirem la funcionalitat per crear els objectes d'aquestes classes.

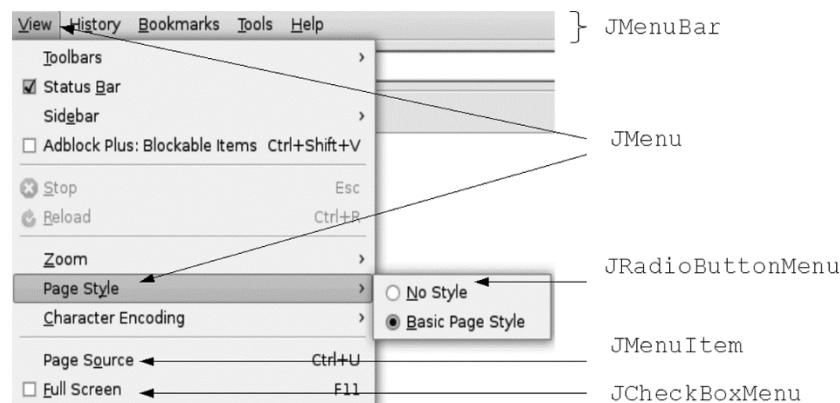
La imatge mostra una finestra de programari amb el títol "DADES DE L'ALUMNE". A la part superior hi ha els botons estàndard de Windows (minimitzar, maximitzar i tancar). El contingut de la finestra inclou: tres camps d'entrada de text etiquetats "DNI", "NOM" i "EDAT"; una secció titulada "NIVELL" amb quatre botons de selecció de raïo corresponents a "ESO", "Batxillerat", "Grau mitjà" i "Grau Superior"; i un botó "ACCEPTAR" a la cantonada inferior dreta.

## 11.5 La barra de menús

Un tret característic força típic de les aplicacions d'escriptori és la utilització de menús en la franja superior de la finestra principal, amb l'objectiu de donar accés a moltes opcions sense atapeir la pantalla. Els components principals vinculats als menús dins la biblioteca Swing són els següents:

- **JMenuBar**: Contenedor que representa la barra de menús. Només n'hi pot haver un per finestra (JFrame).
- **JMenu**: Contenedor que representa un menú individual d'entre els diferents que es poden incloure dins la barra de menús. L'usuari visualitza el seu nom i es desplega en pitjar amb el ratolí. Es poden incloure menús dins d'altres menús, que es despleguen consecutivament.
- **JMenuItem**: Control associat a una opció individual de menú, que l'usuari selecciona.
- **JCheckBoxMenuItem**: Control que combina el JMenuItem i el JCheckBox.
- **JRadioButtonMenuItem**: Control que combina el JMenuItem i el JRadioButtonMenu.

La figura mostra un exemple amb tots els components possibles d'una barra de menús. No es pot afegir cap altre tipus de component o contenidor dins un menú.



Els menús també usen el mètode add per afegir components als contenidors. Es visualitzaran dins el menú exactament en el mateix ordre en què s'han afegit. L'única excepció sobre el mecanisme general és assignar l'única barra de menús a la finestra principal, que es fa cridant el mètode següent:

```
public void setJMenuBar(JMenuBar menubar)
```

A continuació es presenta un fragment del codi que generaria una barra de menús com la que s'ha mostrat en la figura.3, centrant-se en el menú View. Analitzeu detingudament com s'afegeixen els elements del menú mitjançant crides successives del mètode add.

- El mètode addSeparator permet afegir línies de separació entre els elements d'un menú.
- El mètode setSelected serveix per marcar com a seleccionat o desseleccionat un control tipus o .
- El mètode setMnemonic serveix per establir una drecera de teclat per a un control donat. La classe defineix constants per a totes les tecles.
- El mètode setEnabled habilita o deshabilita un control.

```
JFrame frame = new JFrame("Example menu");

//Create the menu bar.
menuBar = new JMenuBar();

//Build the first menu.
```

```

menu = new JMenu("A Menu");
menu.getAccessibleContext().setAccessibleDescription(
    "The only menu in this program that has menu
items");
menuBar.add(menu);

//a group of JMenuItem
menuItem = new JMenuItem("A text-only menu item");
menuItem.getAccessibleContext().setAccessibleDescription(
    "This doesn't really do anything");
menu.add(menuItem);
menuItem = new JMenuItem("Another text-only menu item");
menu.add(menuItem);

//a group of radio button menu items
menu.addSeparator();
ButtonGroup group = new ButtonGroup();
rbMenuItem = new JRadioButtonMenuItem("A radio button menu item");
rbMenuItem.setSelected(true);
group.add(rbMenuItem);
menu.add(rbMenuItem);
rbMenuItem = new JRadioButtonMenuItem("Another one");
group.add(rbMenuItem);
menu.add(rbMenuItem);
//a group of check box menu items
menu.addSeparator();
cbMenuItem = new JCheckBoxMenuItem("A check box menu item");
menu.add(cbMenuItem);
cbMenuItem = new JCheckBoxMenuItem("Another one");
menu.add(cbMenuItem);

//a submenu
menu.addSeparator();
submenu = new JMenu("A submenu");
menuItem = new JMenuItem("An item in the submenu");
submenu.add(menuItem);
menuItem = new JMenuItem("Another item");
submenu.add(menuItem);
menu.add(submenu);

//Build second menu in the menu bar.
menu = new JMenu("Another Menu");
menu.setMnemonic(KeyEvent.VK_N);
menu.getAccessibleContext().setAccessibleDescription(
    "This menu does nothing");
menuBar.add(menu);

frame.setJMenuBar(menuBar);

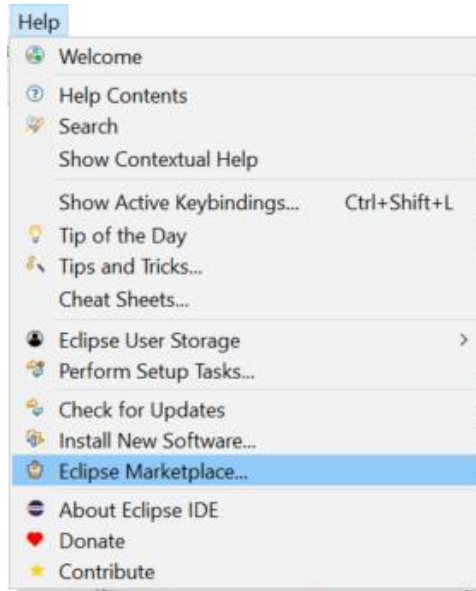
```

**Exercici 11.7:** Implementar i complementar l'exemple anterior per introduir un menú a qualsevol de les finestres que teniu implementades d'altres exercicis.

## 11.6 Eclipse WindowBuilder

L'entorn de desenvolupament Eclipse ens ofereix una eina per a construir les GUI d'una manera més senzilla aprofitant les eines modernes de desenvolupament.

Per instal·lar el complement WindowBuilder podeu anar a Help/Eclipse Marketplace...



A la casella del cercador podeu posar WindowBuilder i instal·lar la darrera versió.

### WindowBuilder 1.9.8



WindowBuilder is composed of SWT Designer and Swing Designer and makes it very easy to create Java GUI applications without spending a lot of time writing code.... [more info](#)

by [Eclipse Foundation](#), EPL

[SWT](#) [swing](#) [wysiwyg](#) [graphical](#) [WindowBuilder](#)

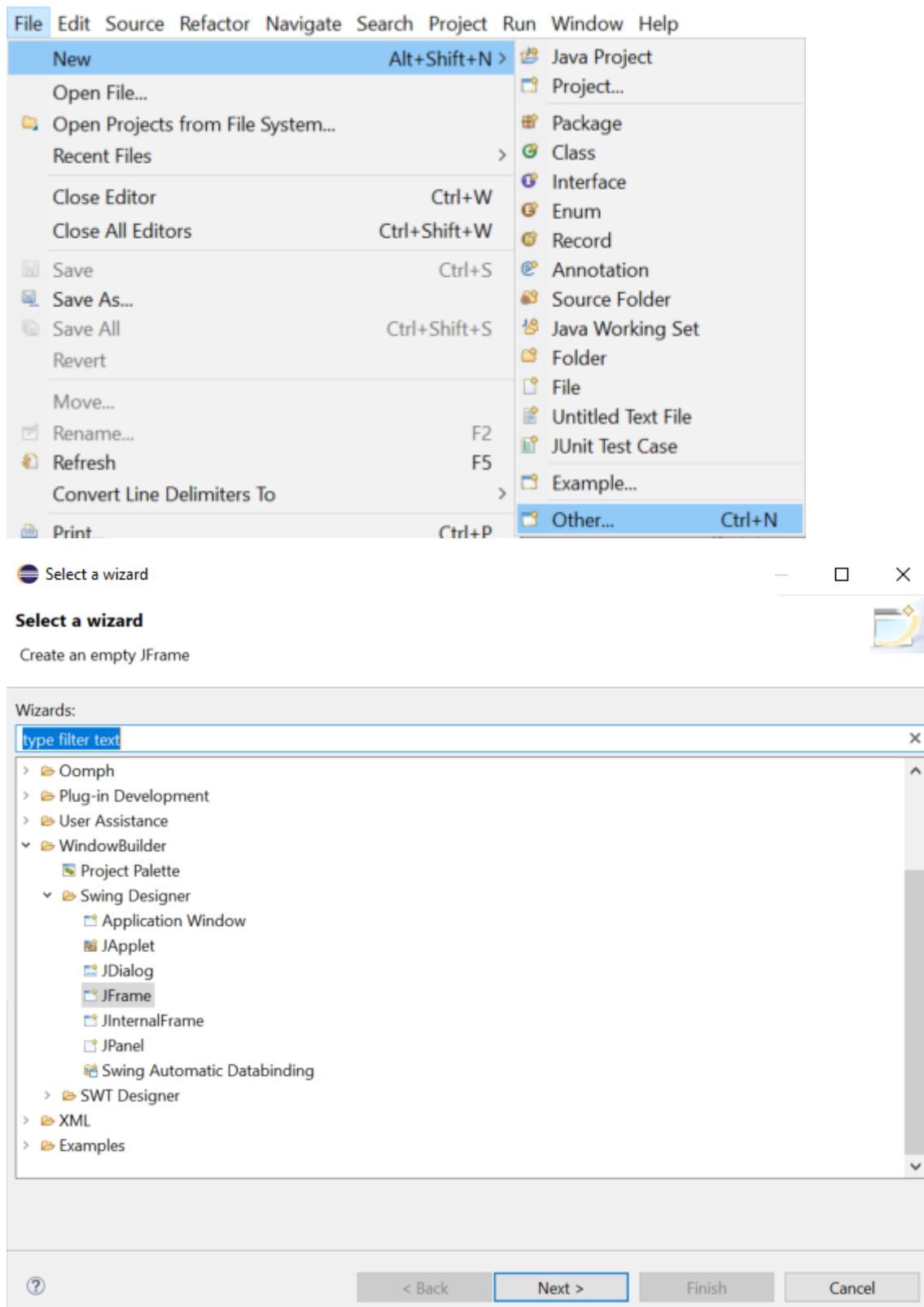
★ 815



Installs: **758K** (20.916 last month)

Change ▼

Un cop instal·lat, podreu crear noves finestres seleccionant File/New/Other... i us apareixerà l'opció de crear un nou JFrame automàticament


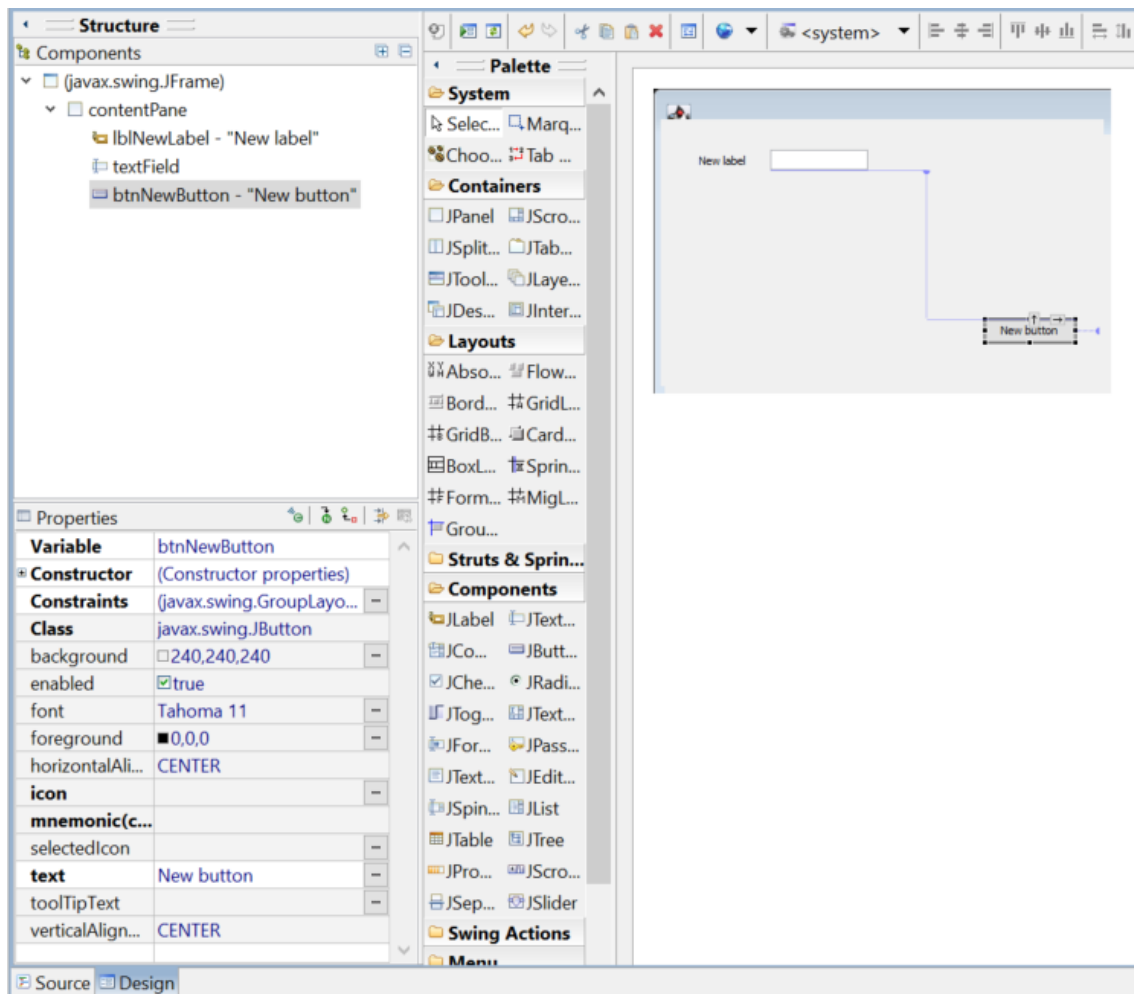


Els JFrames creats d'aquesta manera apareixeran amb dues pestanyes a la part de baix que us permetrà dissenyar la finestra de manera gràfica.

```

45         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
46         setBounds(100, 100, 450, 300);
47         contentPane = new JPanel();
48         contentPane.setBorder(new EmptyBorder(5, 5, 5,
49         setContentPane(contentPane);
50

```

Quan esteu en mode disseny haureu d'escollir primer igual com fèiem abans un layout pel panell del JFrame principal i després ja podreu anar afegint els components a la ubicació que vulgueu.

## 11.6 Esdeveniments. Concepte i controladors

Mitjançant la combinació d'objectes de les classes definides en la biblioteca Java Swing és possible generar finestres amb tots els components correctament ubicats i visualitzats, però qualsevol interacció per part de l'usuari no fa res. Per **interconnectar la interface gràfica generada amb la lògica interna del programa**, quan l'usuari dona una ordre, aquesta es tradueix en una interacció directa amb els objectes que componen la lògica interna del programa, i en canvia l'estat.

Aquesta tasca d'interconnexió no és trivial si és vol fer la feina ben feta, ja que hi ha problemes que, si no es preveuen, tenen un impacte greu en l'escalabilitat de l'aplicació, i incrementen la possibilitat que qualsevol lleugera modificació impliqui canvis en moltes altres classes. El més important de tot és no respectar el principi d'encapsulació, barrejant el codi vinculat

exclusivament a la gestió de la interface gràfica amb el de la lògica del programa. Si això succeeix, el disseny generat quedarà lligat per sempre a aquesta interface, i adaptar-lo a una altra implicarà modificacions. Les classes deixen de ser directament reusables. Per tant, **l'objectiu principal és separar les classes vinculades a la interface amb les vinculades a la lògica interna**, o estat, de l'aplicació.

El **cas ideal d'interconnexió** és aquell en què les classes del model estàtic UML, resultat del procés de disseny, es poden integrar dins de qualsevol interface, sigui quina en sigui l'aparença, sense haver de fer absolutament cap canvi sobre aquestes.

#### 11.6.1 El patró Model-Vista-Controlador

Un **patró de disseny** és una estratègia a seguir per resoldre un problema determinat dins el procés de disseny del programari, de manera es pugui emprar en un ampli ventall de situacions. De totes maneres, sempre cal adaptar aquesta estratègia als detalls de cada cas concret.

Per a interconnectar la lògica interna de l'aplicació, generada en el procés de disseny en forma de diagrama UML, amb la interface d'usuari, una immensa majoria de llenguatges, incloent-hi el Java, es decanta pel patró de disseny anomenat **Model-Vista-Controlador (MVC)**.

El patró MVC divideix les diferents classes de l'aplicació en **tres conjunts diferenciats**, segons el rol. Aquesta diferenciació és exclusivament conceptual i no s'ha de traduir en algun tipus de relació entre classes a nivell de diagrama UML (associació, herència, etc.).

Les classes del **Model** representen la **lògica interna** del programa i contenen l'estat de l'aplicació, i proporcionen totes les funcionalitats exclusives a l'aplicació, independents de la interface. Aquest grup està compost principalment per les classes que el dissenyador ha reflectit en el model estàtic UML.

Les classes de la **Vista** representen l'aspecte purament vinculat a la **interface d'usuari**, gràfica o no. Aquestes s'encarreguen tant de capturar les interaccions de l'usuari com d'accedir a les dades emmagatzemades en el model, de manera que l'usuari les pugui visualitzar i manipular correctament. Per tant, una de les seves responsabilitats principals és mantenir la consistència entre les dades internes i el que visualitza l'usuari. Qualsevol classe usada per gestionar un element on visualitzar la informació o donar ordres a l'aplicació forma part d'aquest grup. Per exemple, la classe que gestiona una impressora o un panell de LED també es consideraria part de la Vista. Tots els components gràfics Swing de l'aplicació pertanyen exclusivament a aquest grup.

Les classes del **Controlador** representen la capa intermèdia entre dades i interface que s'encarrega de traduir cada interacció de l'usuari, capturada per la Vista, en **crides a mètodes** definits en el Model, de manera que s'executi la lògica interna adequada a l'ordre donada per l'usuari. En aquest grup hi haurà **una classe per a cada funcionalitat** que es vulgui incorporar a la interface.

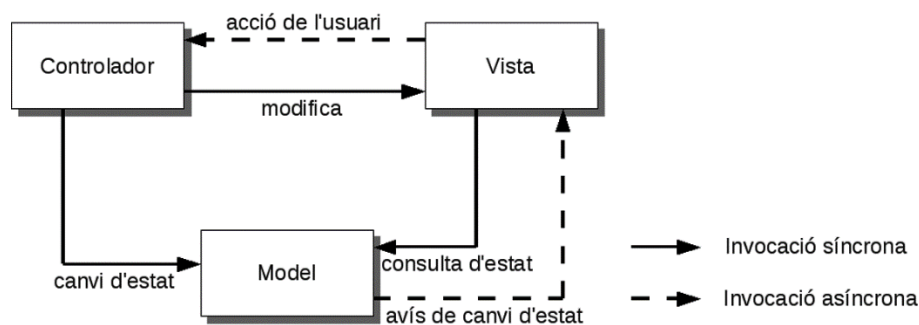
Mitjançant les interaccions dels objectes de classes dels diferents conjunts es genera una **aplicació que respon a les ordres de l'usuari**. Atès que aquestes interaccions, en darrera instància, sempre es tradueixen en crides a mètodes, aquest patró de disseny estableix els mecanismes necessaris perquè un usuari pugui cridar mètodes sobre les instàncies de les classes del diagrama UML resultat del disseny de l'aplicació.

L'esquema d'actuació dels elements del patró MVC és el següent:

1. L'usuari actua sobre una instància d'una classe de la Vista, per exemple, un botó.
2. L'objecte rep l'acció i la passa a una instància d'una classe del Controlador. En aquest procés es transmet tota la informació addicional necessària per al tractament correcte de l'acció. Per exemple, si s'ha fet amb el botó dret o esquerre del ratolí, o si s'ha fet doble clic.

3. L'objecte Controlador crida els mètodes que pertorqui del Model per aconseguir el resultat associat a l'acció de l'usuari.
4. L'estat dels objectes del Model canvia.
5. El Model avisa la Vista que hi ha hagut canvis.
6. Els objectes de la Vista que mostren la informació associada a l'estat del Model criden els mètodes de consulta necessaris per mostrar correctament el nou estat.

La figura mostra un resum de les interaccions entre els objectes dels tres conjunts. Les diferents interaccions normalment es divideixen entre les que es tradueixen en crides a mètodes en moments clarament identificables durant l'execució de l'aplicació, les crides síncrones, i les que no ho són, ja que poden sorgir en qualsevol moment, les crides asíncrones.



### 11.6.2 Control d'esdeveniments

Una de les particularitats de les aplicacions interactives és que, un cop es posen en marxa, aquestes es queden parcialment o totalment inactives a l'espera que l'usuari faci alguna acció. Fins que això no succeeix, no s'executa cap codi associat a les funcionalitats de l'aplicació. Aquesta circumstància es plasma dins el patró MVC amb les crides asíncrones, també anomenades esdeveniments, entre els objectes de la Vista i el Controlador. És per això que la biblioteca Java Swing implementa aquest patró mitjançant el mecanisme anomenat **control d'esdeveniments**. Els esdeveniments són generats pel motor gràfic del Java en resposta a accions de l'usuari. El programador no s'ha de preocupar de com es generen realment.

Les accions de l'usuari sobre components Swing generen **esdeveniments**. Aquests esdeveniments són associats a fragments de codi, que s'executen cada cop que tenen lloc.

Reflexionant una mica, el concepte d'esdeveniment asíncron amb un codi associat no és un aspecte totalment nou, ja que conceptualment no és gens diferent de l'usat en el Java per al control d'errors mitjançant excepcions.

Swing defineix un conjunt de classes que representen cada tipus d'interacció, genèricament, que l'usuari pot realitzar sobre un component. Com en el cas de les excepcions, **els esdeveniments són objectes**, resultants de la instanciació d'alguna d'aquestes classes, que el programador pot manipular per obtenir informació més detallada respecte a les particularitats de l'acció que l'han generat. Totes elles pertanyen al paquet **java.awt.event**.

Alguns dels tipus d'esdeveniments més típics són els següents:

- **ActionEvent**: Es genera en realitzar l'acció més típica, o estàndard, sobre un control. Cada control estableix quina considera que és la seva acció estàndard, que pot ser diferent per a cada cas. Per exemple, en el cas d'un botó, es genera en pitjar-lo.
- **MouseEvent**: Generat davant qualsevol acció vinculada exclusivament al ratolí. Per exemple, en pitjar qualsevol botó del ratolí, moure l'apuntador dins una àrea concreta, etc.

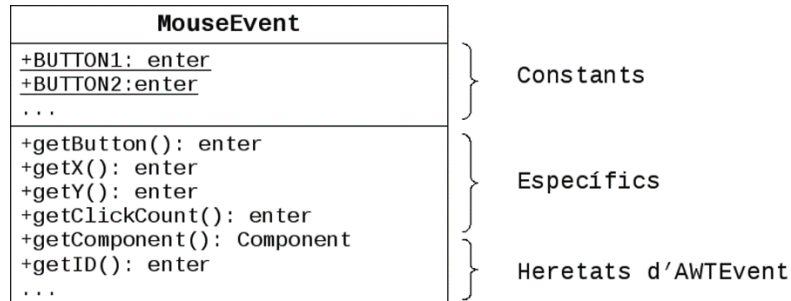


- **KeyEvent:** Associat a accions exclusivament relatives al teclat. Per exemple, pitjar una tecla o deixar-la anar.
- **WindowEvent:** Qualsevol esdeveniment relatiu a l'estat d'una finestra. Per exemple, minimitzar-la, maximitzar-la, redimensionar-la o tancar-la.
- **FocusEvent:** Aquest esdeveniment ve donat per accions vinculades al focus de controls. Amb focus es refereix al fet que un control queda remarcat dins la interface, de manera que s'hi pot interactuar directament mitjançant el teclat. Exemples d'aquest tipus d'esdeveniment són guanyar o perdre el focus.
- **TextEvent:** Generat en realitzar accions relatives a camps de text. Per exemple, modificar un camp de text.

No tots els components Swing poden generar absolutament tots els tipus d'esdeveniments, sinó que només generen els associats a interaccions que realment poden rebre. De fet, alguns estan molt vinculats a components molt específics: per exemple, en una aplicació d'escriptori, només els JFrame generen WindowEvent i només els JTextComponent generen TextEvent. Per veure amb detall quins esdeveniments llença cada component Swing davant cada tipus d'acció, és necessari mirar la documentació de Java.

La figura mostra un exemple d'esdeveniment: la classe MouseEvent. Com es pot veure, aquest disposa d'un conjunt de mètodes que permeten consultar-ne els detalls: quin botó s'ha pitjat, quantes pulsacions s'han fet, quines són les coordenades de la seva posició sobre la finestra principal, etc. Alguns d'aquest mètodes són específics d'un esdeveniment generat pel ratolí, i d'altres són aplicables a qualsevol, heretats de la superclasse AWTEvent.

El mètode getComponent retorna el component que ha generat l'esdeveniment.



### 11.6.3 Captura d'esdeveniments

La captura d'esdeveniments, de manera que es puguin tractar dins l'aplicació, es realitza mitjançant uns objectes especials anomenats **Listeners**. Aquests objectes conformen la part de Controlador del patró MVC plasmat en la biblioteca gràfica de Java.

#### Listeners

Hi ha un tipus de Listener diferent per cada tipus d'esdeveniment: objectes ActionListener, que capturen esdeveniments tipus ActionEvent, objectes MouseListener per capturar els MouseEvent, etc. Cada tipus de Listener només pot capturar el tipus d'esdeveniment al qual està associat, i absolutament cap altre.

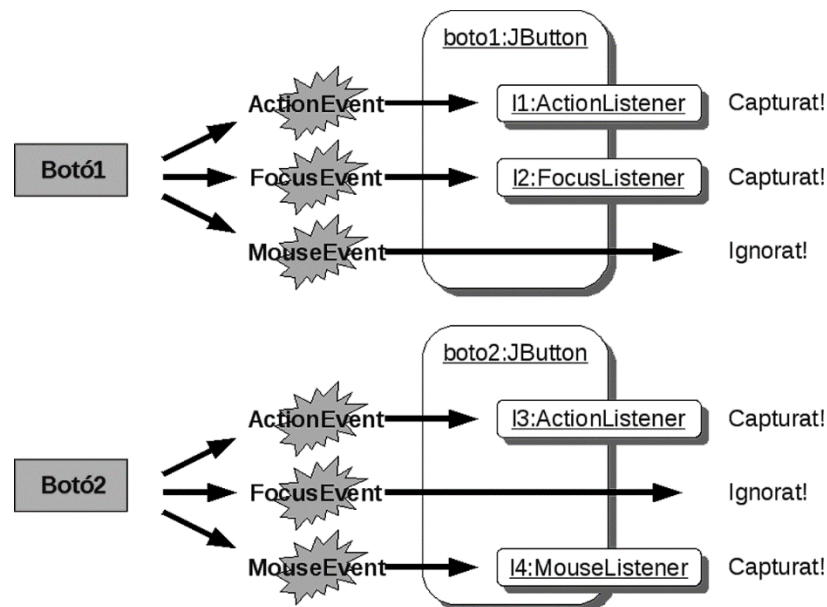
Donat un tipus d'esdeveniment "xxxEvent", normalment, el seu "Listener" associat té el nom "xxxListener". El mètode per assignar-lo a un component es diu "addxxxListener".

Un **Listener** captura els esdeveniments que genera **un únic component** dins la interface gràfica. Perquè pugui acomplir aquesta tasca cal **registrar-lo** en el component. Si per un tipus concret

d'esdeveniment el component no té cap Listener associat registrat, aquests s'ignoraran, independentment del fet que l'usuari pugui fer l'acció sobre el component. No passarà res a l'aplicació en fer-la. Per fer el registre, cada component disposa d'un mètode diferent per cada tipus d'esdeveniment que pot generar. Per exemple, els components que generen `ActionEvent` disposen d'un mètode `addActionListener` que permet registrar-hi un `ChangeListener`. En contraposició, els components que no poden generar aquest tipus d'esdeveniment no disposen d'aquest mètode i, per tant, no poden tenir registrat cap `ActionListener`.

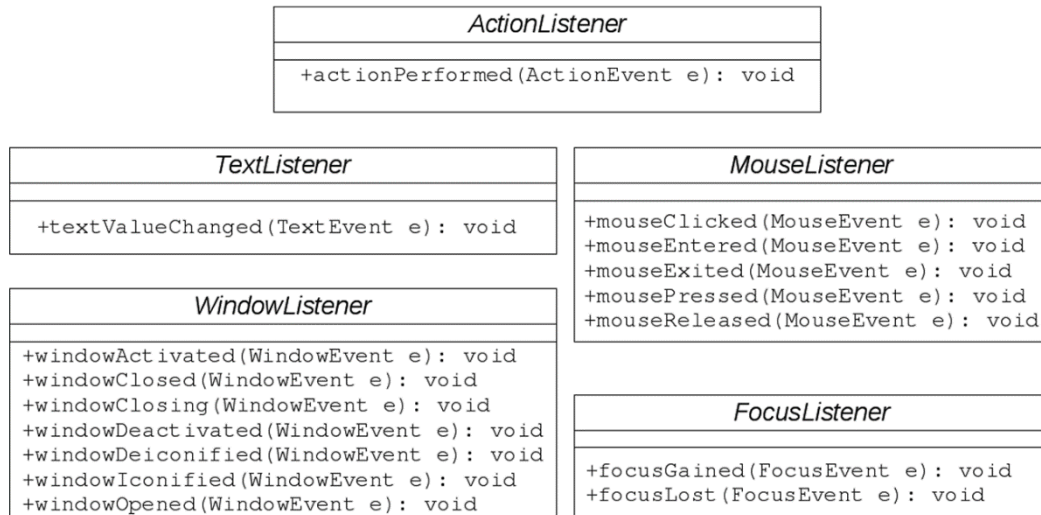
Els objectes `Listener` són els encarregats de **capturar els diferents tipus d'esdeveniment**, adoptant el rol de **Controlador** del patró MVC. Per poder fer-ho, cal que estiguin registrats en els components que generen els esdeveniments a capturar.

Cada component té la seva **llista individualitzada d'objectes Listener**, tants com tipus d'esdeveniment calgui capturar pel component. La figura mostra un esquema d'aquest mecanisme per dos botons diferents dins una interface gràfica. D'acord amb la seva llista de Listeners, cada botó respon de manera diferent davant els diferents esdeveniments que generen individualment. Val la pena remarcar que cada Listener que apareix en la figura és un objecte diferent.

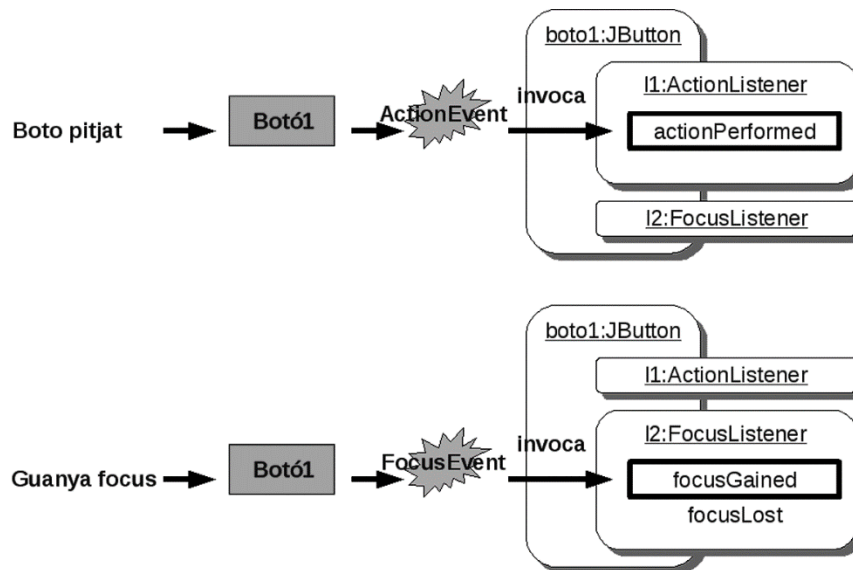


La biblioteca gràfica del Java defineix dins de cada tipus de Listener un **conjunt de mètodes**, cadascun dels quals correspon a una interacció més concreta del que marca l'esdeveniment. Per exemple, la generació d'un `MouseEvent` per si mateixa només indica que ha passat alguna cosa amb el ratolí, però res més. El nombre d'aquests mètodes varia segons el tipus d'interaccions més específiques que pot significar cada tipus d'esdeveniment.

La figura mostra els Listeners més usats d'entre els definits per Java, amb tots els seus mètodes.



Quan un Listener captura un esdeveniment, immediatament crida el mètode corresponent a l'acció més concreta que ha dut a terme la generació de l'esdeveniment. Per exemple, si es genera un MouseEvent per la pulsació d'un botó del ratolí, el MouseListener, en capturar-lo, executarà immediatament el mètode mouseClicked. D'aquesta manera, és possible fitar quina acció concreta ha dut a terme l'usuari sobre el component que ha generat l'esdeveniment. El mateix esdeveniment es passa com a paràmetre en aquesta crida, de manera que és possible consultar-ne els detalls (quin botó s'ha pitjat, la seva posició, etc.). Tot aquest comportament es produeix automàticament, gestionat pel motor gràfic de Java.



Un cop s'ha entès com es porta a terme la captura d'esdeveniments i què passa quan es porta a terme, és el moment de veure la darrera peça que falta: com s'executa un tros de codi concret quan un usuari realitza una acció sobre la interfície gràfica Swing.

Dins la biblioteca gràfica de Java, tots els diferents tipus de Listener es troben definits com a interfícies Java. Concretament, tots hereten de la interfície comú `EventListener`. Per tant, en realitat, no és possible instanciar directament un Listener a partir d'ells, ja que aquests només proporcionen la definició dels mètodes mostrats en la figura, però no contenen cap codi executable. Per poder instanciar un Listener d'un tipus concret, **el desenvolupador ha de crear una classe pròpia que implementi la interfície corresponent** i, per tant, codificar mitjançant sobreescritura tots i cadascun dels mètodes definits en la interfície. L'objecte Listener que

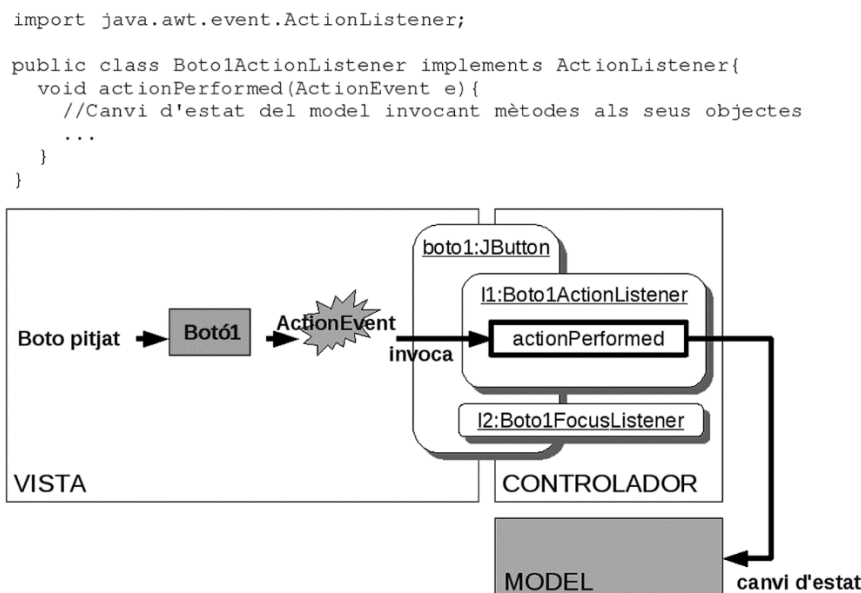
realment es registra en un component és una instància d'aquesta classe creada pel desenvolupador.

És justament en aquest punt, en la implementació dels mètodes definits en la interface, que s'assigna **el codi que es vol executar** davant l'acció d'un usuari. Pel mecanisme de polimorfisme, quan el Listener, prèviament registrat en un component, capturi un esdeveniment i cridi el mètode corresponent a l'acció de l'usuari, el codi que s'executarà serà l'assignat a aquest mètode en la classe creada pel desenvolupador. Així, doncs, es pot veure com gràcies a l'ús de mètodes polimorfes ha estat possible crear la biblioteca gràfica del Java de manera que pugui ser adaptada a qualsevol aplicació.

El desenvolupador ha de generar una classe per cada Listener que vulgui usar dins l'aplicació. Cadascuna d'aquestes classes implementa la interface Listener associada al tipus d'esdeveniment a controlar. Les seves instàncies són les que realment es registren en els components.

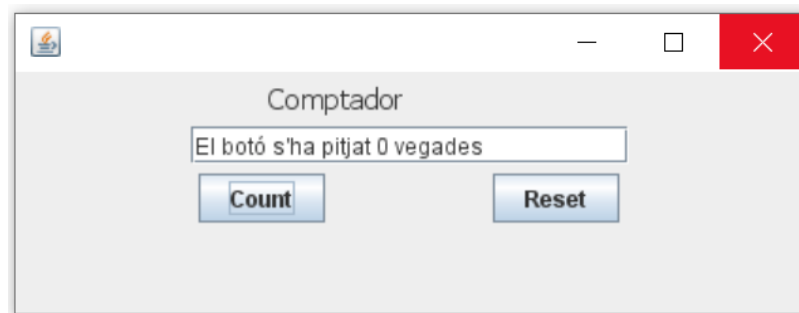
Donats els rols dels elements del patró MVC, és justament dins els mètodes sobreescrits a aquests Listeners personalitzats des d'on cal fer les crides cap a objectes del Model. La figura mostra una visió general de tot el sistema, associant ja cada part implicada al patró MVC. Recordem que, pel mecanisme d'herència, un objecte `Boto1ActionListener` també és un `ActionListener`.

Una de les conclusions d'aquest mecanisme és que, atès que el codi associat a una mateixa acció en diferents components normalment també serà diferent, això implica que el desenvolupador ha de crear **una classe pròpia Listener per cada tipus d'esdeveniment i cada component** dins la interface gràfica. Per exemple, si hi ha tres botons i es vol assignar codi diferent a l'acció de pitjar cada un, caldrà definir tres classes diferents que implementin la interface `ActionListener`. A cada una s'assignarà el codi que correspongui, diferent dels altres, als mètodes `actionPerformed` respectius. Un cop fet, ja només resta crear una instància de cadascuna d'aquestes tres classes i assignar una a cada botó, d'acord amb el codi que es vol executar en pitjar cada un. La figura esquematitza aquest comportament.



Tot i que normalment es registra un objecte Listener diferent en cada component per cada tipus d'esdeveniment, res no impedeix reusar el mateix objecte per a un mateix tipus d'esdeveniment en diferents components. La condició per fer-ho és que el codi que cal executar en capturar l'esdeveniment sigui exactament el mateix per a tots els casos.

Per veure un exemple feim una interface simplement amb un botó i amb un quadre de text que anirem modificant cada vegada que pitgem el botó.



Primer hem d'implementar un listener per a cada botó programant el que volem que facin. Per exemple volem que el **botó Count** incrementi una variable i modifiqui el text del JTextField. Per això podem definir una classe interna que implementi l'interfície ActionListener, el que ens obligarà a codificar el mètode actionPerformed, que s'executarà quan es produeixi el click sobre el botó Count

```
private class countActionListener implements ActionListener{
    public void actionPerformed(ActionEvent e) {
        times++;
        textField.setText("El botó s'ha pitjat " + times + "
vegades");
    }
}
```

Així però només tenim el listener programat, ens falta registrar-lo al botó corresponent i per això emprarem el mètode addActionListener.

```
btnCount.addActionListener( new countActionListener());
```

Ara ja sí, quan es produeixi un esdeveniment tipus ActionListener, que es produirà quan es pitgi el botó Count, s'executarà el codi definit dins actionPerformed.

Feim el mateix pel **botó Reset**. En aquest cas volem resetejar la variable comptador a zero i buidar el contingut del JTextField.

```
private class resetActionListener implements ActionListener{
    public void actionPerformed(ActionEvent e) {
        times=0;
        textField.setText("");
    }
}
```

I associam el listener al botó:

```
btnReset.addActionListener(new resetActionListener());
```

## Adaptadors

Atès que els Listeners són interfaces, el desenvolupador està obligat a sobreescrivre tots els seus mètodes sense cap excepció, o en cas contrari el compilador retornarà un error. Això vol dir que si per algun dels mètodes estesos definits no es vol realitzar realment cap acció, serà necessari deixar el mètode buit, sense codi. En els casos de Listeners amb diversos mètodes, per exemple, el WindowListener, això pot ser pesat i fer el codi més confús. Per aquest motiu, la biblioteca gràfica del Java defineix un conjunt de classes Listener no abstractes amb tots els mètodes ja sobreescrits, però de contingut buit: **els adaptadors (adapters)**. Per tant, el desenvolupador també pot generar diferents tipus de Listener heretant-ne d'ells i només sobreescrivint els mètodes que realment vol usar.

Tots els adaptadors també estan definits en el paquet java.awt.event. Hi ha un adaptador per cada tipus de Listener amb més d'un mètode: FocusAdapter, WindowAdapter, KeyAdapter, etc. No hi ha adaptadors per a Listeners amb un únic mètode, com l'ActionListener o el TextListener, ja que en aquest sentit no aporten res.

## Classes anònimes

En la immensa majoria de casos, els Listeners que codifica el desenvolupador són classes relativament senzilles i curtes, amb pocs mètodes, i de les quals només s'usa una instància dins de tota l'aplicació, ja que cada Listener se sol registrar a un únic component. Atès aquest fet, és útil conèixer un mecanisme que ofereix Java per definir classes auxiliars de manera encara més simple que les classes privades: les **classes anònimes**.

Una **classe anònima** no té nom. Es caracteritza perquè en lloc de definir-se com a entitat diferenciada amb una capçalera class, es defineix dins el codi just en el moment precís d'instanciar-la.

Les classes anònimes són un mecanisme del llenguatge Java que es pot usar per a qualsevol situació, però són especialment útils a l'hora de generar Listeners. Com les classes privades, les classes anònimes tenen accés directe a qualsevol atribut de la classe en què es defineixen.

Per usar-les, és prerequisite que la classe que es vol definir sigui subclasse d'una altra ja existent. La seva sintaxi és la següent:

```
new NomSuperclasse() {  
    //Definició normal de la classe (atributs + mètodes)  
}
```

Per exemple, els Listeners usats en exemples anteriors es poden definir de la manera següent, en lloc de mitjançant classes privades. Fixeu-vos com les classes són definides just en el moment d'instanciar-les (en fer un "new"), en lloc de fer-ho prèviament en un bloc a part.

```
btnCount.addActionListener( new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        times++;  
        textField.setText("El botó s'ha pitjat " + times + "  
vegades");  
    }  
});
```

```
btnReset.addActionListener(new ActionListener(){  
    @Override  
    public void actionPerformed(ActionEvent e) {
```

```
        times=0;
        textField.setText("");
    }
});
```

### Exemple d'aplicació separada en Model-Vista-Controlador

La **vista** conté l'entorn gràfic d'usuari. En aquest cas és una simple aplicació que incrementarà els anys o els decrementarà d'una persona:



La part que lliga la Vista amb el Controlador s'aconsegueix fent que el mètode que assigna els Listeners dels botons rebi com a paràmetre l'objecte Controlador on estan implementades les ordres que s'executaran al prémer els botons.

Els botons s'anomenen `btnMoreYears` i `btnLessYears`.

```
void setControlador(Controlador c) {
    btnMoreYears.addActionListener(c);
    btnLessYears.addActionListener(c);
}
```

El **Model** conté els objectes del programa i els mètodes per modificar-los

```

public class Model {
    String nombre = "";
    int edad = 0;

    void addYears(){
        edad++;
    }

    void subYears() {
        edad--;
    }
}

```

El **controlador** conté el codi que executaran els botons, donarà les ordres per a que els valors del model canviïn i s'actualitzin les vistes:

```

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Controlador implements ActionListener{

    Model model;
    Vista vista;

    public Controlador(Model model, Vista vista) {
        super();
        this.model = model;
        this.vista = vista;
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        model.edad = Integer.parseInt(vista.getTextFieldEdad().getText());
        if (e.getActionCommand().equals("ADDYEARS")){
            model.addYears();
            vista.getTextFieldEdad().setText(String.valueOf(model.edad));
            vista.getLblResult().setText(vista.getTextFieldNombre().getText() + " has " + String.valueOf(model.edad) + " years.");
        }
        if (e.getActionCommand().equals("SUBYEARS")){
            model.subYears();
            vista.getTextFieldEdad().setText(String.valueOf(model.edad));
            vista.getLblResult().setText(vista.getTextFieldNombre().getText() + " has " + String.valueOf(model.edad) + " years.");
        }
    }
}

```

Els botons han de tenir assignat un valor per **ActionCommand** (emprant `button.setActionCommand (String s)`) perquè és el que conté l'objecte `ActionEvent` i permet assignar les accions als components correctes.

El **programa** simplement crea els objectes i fa visible la vista



```

public class Programa {

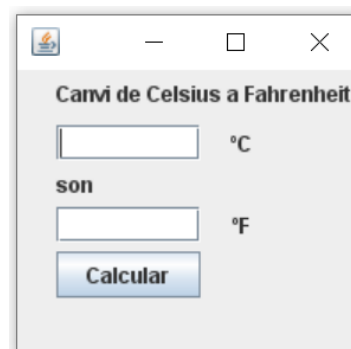
    public static void main(String[] args) {

        Model m = new Model();
        Vista v = new Vista();
        Controlador c = new Controlador(m,v);

        v.setControlador(c);
        v.setVisible(true);
    }
}

```

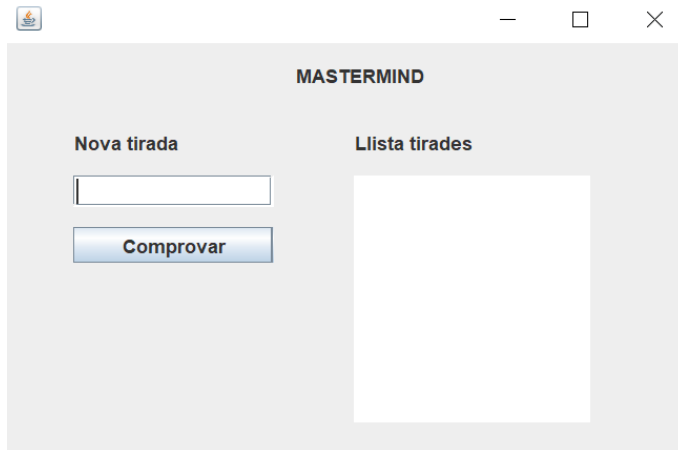
**Exercici 11.8:** Crear un entorn d'usuari gràfic per al problema de conversió de graus Celsius a graus Fahrenheit i construir tot el codi per a que funcioni correctament.



**Exercici 11.9:** Crear un entorn d'usuari per a l'exercici 6.5 on poguem introduir Alumnes nous, professors nous i els pugui emmagatzemar en un fitxer. La finestra principal ha de tenir un menú des d'on es puguin crear els diferents objectes nous i fer la lectura i escriptura dels objectes que hagueu fet en ocasions anteriors. Una estructura del menú podria ser:

- Alumne: Nou alumne
- Professor: Nou professor
- Fitxers:
  - o Llegir fitxer
  - o Escriure fitxer
- Tancar

**Exercici 11.10:** Crear entorn d'usuari per al mastermind on dins un JTextField el jugador pugui introduir les tirades i en un TextArea es puguin veure les respostes que dona l'ordinador.



El programa també haurà de tenir un menú on poder llegir les partides anteriors i emmagatzemar les noves que juguem. Una altra opció al menú ha de permetre llistar la classificació de les partides, ordenades de màxima a mínima puntuació, juntament amb el nom del jugador.