Java Swing

AWT e Swing

No paquete estándar de Java, contamos con tres opcións para crear interfaces gráficas de usuario:

- **AWT** (*Abstract Windows Toolkit*): é unha biblioteca pesada, xa que os compoñentes inclúen código nativo do sistema operativo, polo que son dependentes da plataforma. Isto implica que os compoñentes son mostrados co estilo do sistema operativo.
- **Swing**: é unha biblioteca lixeira. Contén compoñentes lixeiros, independentes da plataforma e xestionados directamente por Java.
- **JavaFX**: permite crear interfaces gráficas de usuario tanto para aplicacións de escritorio como para a web. JavaFX é a aposta actual de Oracle para construír interfaces gráficas de usuario multi-dispositivo. A tendencia actual na construción de interfaces gráficas de usuario é manter a mesma calidade e riqueza de compoñentes, tanto se o cliente é de escritorio, como un navegador, un dispositivo móbil, unha televisión, etc.

Dise que unha biblioteca é pesada ou lixeira en función da dependencia do sistema operativo para visualizar e xestionar os elementos da interface gráfica. No caso de **AWT**, a creación, visualización e xestión dos elementos gráficos depende do sistema operativo, é dicir, é o propio sistema operativo quen debuxa e xestiona a interacción sobre os elementos. No caso de **Swing**, é Java quen visualiza e xestiona a interacción das persoas usuarias sobre os elementos da interface gráfica. Isto ofrece soporte para o denominado *"**pluggable look and feel**"*, é dicir, que a aparencia da plataforma é configurable.

O paquete **java.awt** proporciona clases para a API AWT como TextField, Label, TextArea, RadioButton, CheckBox, List, etc.

O paquete **javax.swing** proporciona clases para a API de Java Swing como JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser, etc. Fixarse que empezan pola letra "J".

Diferencias entre AWT e Java Swing:

Java AWT	Java Swing
Dependente da plataforma	Independente da plataforma
Compoñentes pesados	Compoñentes lixeiros
Non soporta personalización da aparencia.	Soporta personalización da aparencia.
Poucos compoñentes.	Numerosos e potentes compoñentes.
Non sigue MVC (*Model View Controller*)	Sigue MVC

Existen clases co mesmo cometido, tanto en AWT como en Swing. Por exemplo, nas dúas bibliotecas haberá unha clase para crear unha ventá: **Frame** no caso de AWT e **JFrame** no caso de Swing.

Existen outras clases do paquete AWT que tamén se usan en Swing como son os eventos e escoitadores. Dado que non teñen representación gráfica, en Swing úsanse os de AWT.

Que é Swing

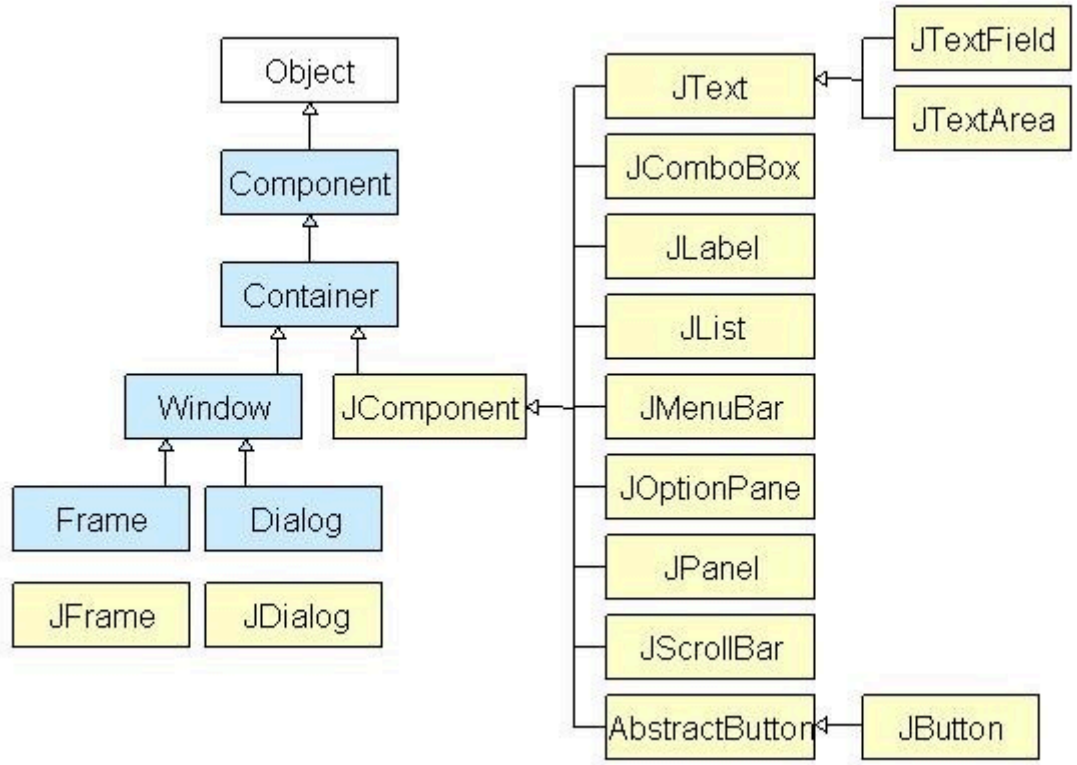
Swing é unha biblioteca de clases que permite crear interfaces gráficas de usuario en Java. O número de clases dentro da biblioteca Swing é enorme. Ten un amplo abano de compoñentes, dende botóns ata paneis e táboas.

Swing forma parte do paquete estándar, polo que non fai falta importar ningún ficheiro adicional ao proxecto.

Características de Swing:

- É **independente da plataforma.** Os compoñentes están escritos sen empregar código nativo, co que ofrecen máis versatilidade multiplataforma.
- Os compoñentes Swing son **lixeiros**, o que permite crear interfaces lixeiras.
- É **fácil de xestionar e configurar**. Permite incluso modificar a configuración en tempo de execución.
- Os seus controis poden ser **facilmente configurables**.
- Sigue o patrón **MVC** (Model - View - Controller). Os compoñentes foron deseñados seguindo o patrón MVC.
- É unha API **accesible**: permite ás tecnoloxías asistenciais, como lectores de pantalla, obter información da interface de usuario.
- **Java 2D API**: permite ás persoas desenvolvedoras incorporar facilmente gráficos 2D de alta calidade, texto e imaxes en aplicacións e applets.
- **Internacionalización**: permite crear aplicacións para interactuar con persoas en múltiples idiomas e inclúe tamén diferentes caracteres como xaponés, chino, etc.

A continuación móstrase unha imaxe da xerarquía de clases de Java Swing:



Contedores

Existen dous elementos básicos para a creación de interfaces gráficas usando Swing:

- **Contedores**: elementos capaces de albergar outros elementos.
- **Compoñentes**: elementos que se engaden aos contedores. Usualmente teñen aspecto gráfico, como por exemplo un botón.

Nunha interface gráfica (GUI) os compoñentes son contidos en contedores (*containers*). Un contedor é un obxecto que herda da clase *Container*, clase que á súa vez é subclase de *Component,* e ten a responsabilidade de conter compoñentes.

Para crear unha interface gráfica é necesario, polo menos, un obxecto contedor.

Swing proporciona tres tipos de contedores de alto nivel. Isto significa que calquera outro contedor que non sexa de alto nivel, ou compoñente, debe ir no seu interior. Estes tres compoñentes de alto nivel son: JFrame, JDialog e JApplet.

- **Marco (JFrame)**: visualízase como unha ventá principal con marco, botóns maximizar e minimizar e barra de título.
- **Diálogo (JDialog)**: pode considerarse como unha ventá emerxente que aparece cando se debe mostrar unha mensaxe. Non é unha ventá completamente funcional como o Marco. Úsase para mostrar un diálogo de confirmación, pedir un dato, mostrar unha mensaxe, …
- **Applet (JApplet)**: permite crear aplicacións con interface gráfica que se executan no contexto dun navegador web.

Existe outro contedor chamado **Panel (JPanel).** Este é un contedor puro e non é unha ventá en si mesma. O único propósito dun Panel é organizar os compoñentes nunha ventá.

Xeralmente unha GUI créase sobre un Frame. Este será o contedor principal que conterá os compoñentes da interface gráfica. Tamén poderá conter outros contedores.

Compoñentes

Os **compoñentes** son os elementos gráficos que se engaden aos contedores para formar aplicacións. Úsanse para mostrar información, como as etiquetas, aínda que tamén son usados para introducir datos como os cadros de texto, listas de selección, etc.

Un compoñente ten unha clase que define o seu aspecto e funcionalidade. Segundo a clase á que pertenza, terá unhas propiedades que poderán ser modificadas, como o texto a mostrar, a cor, tipo de letra, etc.

Os compoñentes, maioritariamente, son elementos con representación gráfica: botóns, caixas de texto, etiquetas,… Estes organízanse dentro dos contedores a través dos xestores de aspecto (***Layout Managers***). Aniñando uns contedores dentro doutros e escollendo os xestores de aspecto adecuados, pódense organizar os compoñentes como se queira.

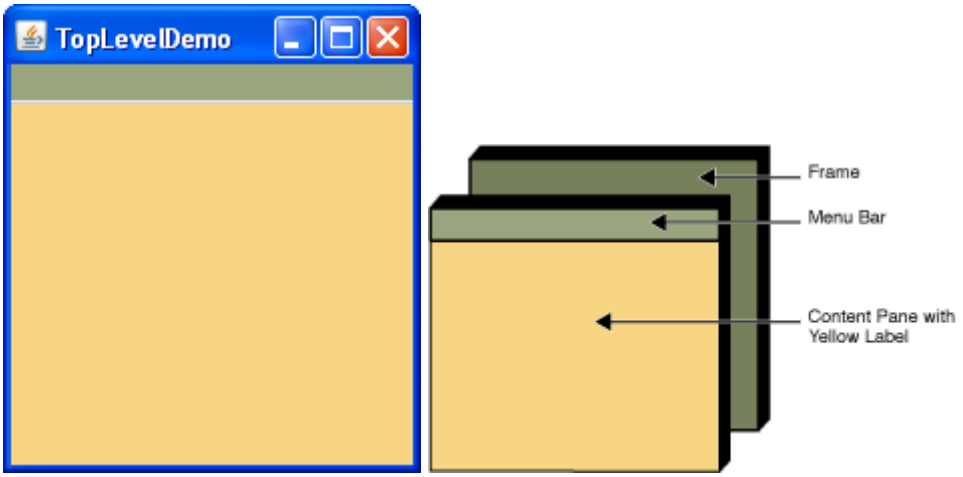
Un **control** é un tipo especial de compoñente que permiten a interacción entre as persoas usuarias e a aplicación para intercambiar información. Algúns exemplos de controis son: botóns, listas, cadros de edición, ...

Cando un compoñente serve para interactuar con el, debe poder xestionarse **mediante un manexador de eventos**, de tal forma que cando se produce o dito evento, debe executarse unha acción programada.

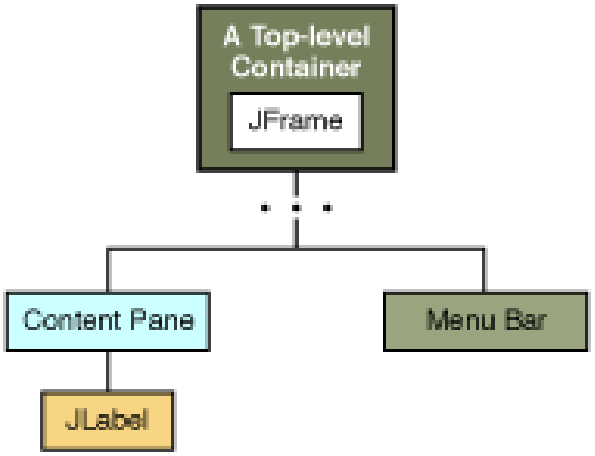
Características a ter en conta:

- Para aparecer en pantalla, cada compoñente gráfico debe ser parte dunha xerarquía de compoñentes que ten como raíz un contedor de alto nivel (JFrame, JDialog ou JApplet).
- Cada compoñente gráfico só pode ser contido unha vez. É dicir, o mesmo compoñente non pode estar en dous contedores diferentes.
- Cada contedor de alto nivel ten un panel de contido (**content pane**) que contén todos os compoñentes visibles.

- Pode engadirse unha barra de menú ao contedor de alto nivel. Esta barra de menú colócase no contedor de alto nivel, pero fóra do panel de contido.



Cando se engaden compoñentes, estes organízanse nunha xerarquía. Exemplo:



Convención nomes

A continuación aparece un recordatorio de regras de nomenclatura de paquetes, clases, interfaces, métodos, constantes, variables e parámetros.

Paquetes:

- Os nomes dos paquetes deben ser en minúsculas.
- Deben ser únicos.
- Soe usarse o nome do dominio ao revés: com.mycompany.
- O guión (-) é un carácter inválido. Hai que substituílo por _
- Non poden empezar por número (pode engadirse _ ao principio).

Clases (similar ás interfaces):

- Usan a nomenclatura **UpperCamelCase**. Cada palabra debe empezar por maiúscula.
- Deben ser nomes (representan cousas), non verbos.
- Empeza por letra maiúscula.

Métodos:

- Normalmente son verbos (en infinitivo) que reflicten a función realizada ou o resultado devolto: getName, setName, isGameOver, …

Constantes:

- Deben escribirse todas as letras en maiúsculas.
- Separar palabras por _. Exemplo: (static final int NUM_GEARS = 6).
- Son declaradas coa palabra chave **final**.

Variables ou propiedades:

- Son sensibles a maiúsculas e minúsculas.
- Usar notación **lowerCamelCase**: se a variable consiste nunha única palabra, debe escribirse en minúsculas. Se consiste en máis dunha palabra, debe ir en maiúsculas a primeira letra da segunda palabra e sucesivas (gearRatio, currentGear, …).
- Deben ter un nome significativo. Usar palabras completas en lugar de abreviaturas crípticas.
- Evitar os caracteres “$” e “_”.
- Deben comezar por unha letra, por convención.
- Non usar palabras reservadas.
- Cando se usan compoñentes de Swing, recoméndase que os nomes das súas variables inclúan información sobre o seu tipo. Para facer isto hai dúas alternativas:
    - que o nome da variable comece por tres letras que indiquen o tipo de compoñente: btn, chk, cmb, lbl, txt,… Abreviaturas do resto de compoñentes.
    - incluír o tipo de compoñente no nome da variable: nameTextField, nameLabel, cancelButton, …

Parámetros tipo:

- Unha única letra en maiúsculas. Ver tipos xenéricos.

  Os nomes dos tipos de parámetros máis usados son:
    - E: elementos (usado en coleccións - Collections).
    - K: chave (key).
    - N: número
    - T: tipo
    - V: valor
    - S, U, V, etc: 2º, 3º, 4º tipos

Primeiro Frame

Un frame é un contedor de alto nivel que se implementa como unha ventá con título e borde. Inclúe os botóns típicos do sistema operativo para minimizar, maximizar e cerrar a ventá.

O tamaño do frame inclúe a área designada para o borde.

Dende NetBeans, crea un novo proxecto da categoría **Java with Maven** -> **Java Application.** Asigna como nome do proxecto **OlaMundo.**

Modifica a clase principal creada por defecto no proxecto co seguinte código:

```
public class App {

    public static void main(String[] args) {
        // Schedule a job for the event-dispatch thread:
        // creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                // Crea un novo frame co título especificado
                JFrame frame = new JFrame("Ola Mundo!");

                // Establece o tamaño
                frame.setSize(600, 500);

                // Establece a funcionalidade ao cerrar o frame (pulsar X)
                // saír do programa
                frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

                // Fai o frame visible (por defecto é invisible)
                frame.setVisible(true);
            }
        });
    }
}
```

A interface gráfica dunha aplicación Swing execútase nun thread especial chamado EDT (Event Dispatch Thread - fío de despacho de eventos) para poder atender os eventos que se produzan na interface gráfica. Esta funcionalidade impleméntase executando o código da aplicación dentro de **SwingUtilities.invokeLater**(...).

Cando as persoas usuarias pechan unha ventá, o comportamento por defecto implementado é ocultala. Este comportamento pode cambiarse usando o método setDefaultCloseOperation(int), como se fai no exemplo anterior.

Exercicio:

1. Modifica o código anterior e proba os posibles valores definidos na documentación como argumentos de **setDefaultCloseOperation**.

Frame personalizado

Normalmente, cando se programa unha aplicación Swing é habitual crear unha clase personalizada que estende de JFrame. No construtor da clase personalizada colócase o código necesario para configurar a ventá.

Dende NetBeans, crea un novo proxecto da categoría **Java with Maven** -> **Java Application.** Asigna como nome do proxecto **FramePersonalizado.**

Engade unha nova clase ao paquete creado por defecto no proxecto co seguinte código e compárao co código mostrado en apartados anteriores:

```
public class MainFrame extends JFrame {
    public MainFrame() {
        super("Ola Mundo!");

        setSize(600, 500);
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        setVisible(true);
    }
}
```

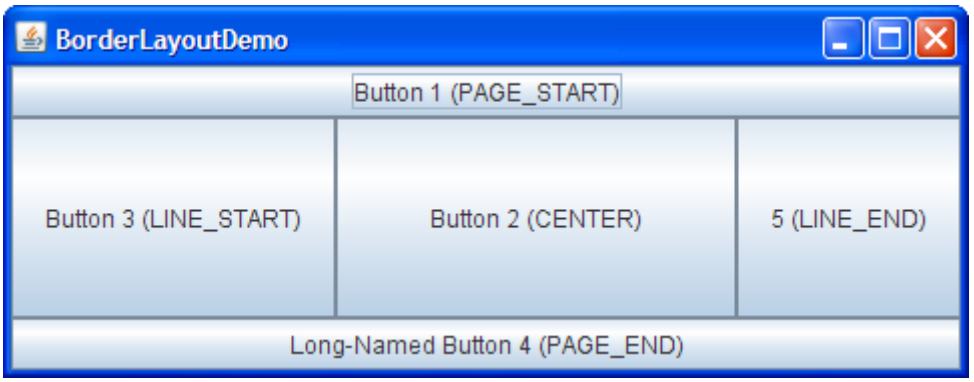
O proxecto terá igualmente unha clase principal que o único que fará é chamar á clase MainFrame personalizada:

```
public class Main {
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new MainFrame();
            }
        });
    }
}
```

O exemplo anterior é equivalente ao mostrado no apartado Primeiro Frame. Probar a executalo.

Antes de continuar engadindo compoñentes á ventá é necesario establecer o **Layout**. O layout é o encargado de decidir onde colocar os compoñentes na ventá e como redimensionalos cando a ventá cambia de tamaño. Máis información sobre Layout Managers.

No seguinte exemplo vaise usar un BorderLayout, que coloca os compoñentes en 5 áreas: arriba, abaixo, esquerda, dereita e centro. Estas áreas son especificadas polas constantes de BorderLayout: PAGE_START, PAGE_END, LINE_START, LINE_END e CENTER.



Normalmente, os **compoñentes decláranse como variables privadas de instancia**, para que se poida acceder a eles dende diferentes métodos.

Modifica a clase anterior como se indica a continuación:

```
public class MainFrame extends JFrame {
    // Declarar os compoñentes como variables privadas de instancia
    private JTextArea textArea;
    private JButton aceptarButton;

    public MainFrame() {
        super("Ola Mundo!");

        // Establecer o Layout
        setLayout(new BorderLayout());

        // Inicializar os compoñentes
        textArea = new JTextArea();
        aceptarButton = new JButton("Aceptar");

        // engadir os compoñentes
        add(textArea, BorderLayout.CENTER);
        add(aceptarButton, BorderLayout.PAGE_END);

        setSize(600, 500);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
}
```

Por comodidade, os métodos **add()**, **remove()** e **setLayout()** do obxecto JFrame foron reescritos, de tal forma que delegan a súa funcionalidade do correspondente método do ContentPane. Por exemplo:

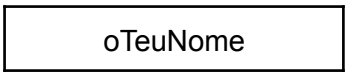
frame.add(...) -> sería equivalente a **frame.getContentPane().add(...);**

NOTA: fixarse que só estes tres métodos delegan a funcionalidade no panel de contido. Isto significa que getLayout() non devolverá o layout establecido con setLayout(). SetLayout establece o layout do panel de contido e getLayout devolve o layout do frame. Para acceder ao layout do panel de contido haberá que executar **getContentPane().getLayout().**

Probar a executar a aplicación anterior, comprobar que se mostra o frame aínda que non ten funcionalidade. Aínda que se pulse o botón, non ten asociada ningunha acción.

Exercicios:

1. Modifica o código anterior para que o frame apareza centrado na ventá.
2. Modifica o código anterior para que a ventá non poida cambiar de tamaño.
3. Engade na parte de arriba do BorderLayout (PAGE_START) unha etiqueta coas seguintes características: que teña como texto **oTeuNome,** tipo de letra Arial, cursiva, tamaño 24 e que o seu texto estea centrado na etiqueta. Para comprobar que o texto está centrado, pode ser necesario establecer un tamaño da etiqueta máis grande do texto que contén.



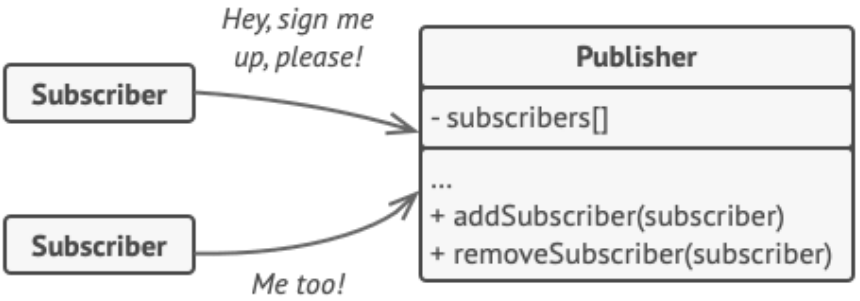
Programación dirixida por eventos

Imaxina que queres coñecer as ofertas de traballo publicadas na páxina do IES San Clemente. Para iso poderías conectarte frecuentemente á páxina web e comprobar se hai algunha nova entrada ou, poderías subscribirte ao boletín e recibir un correo cada vez que haxa unha nova publicación.

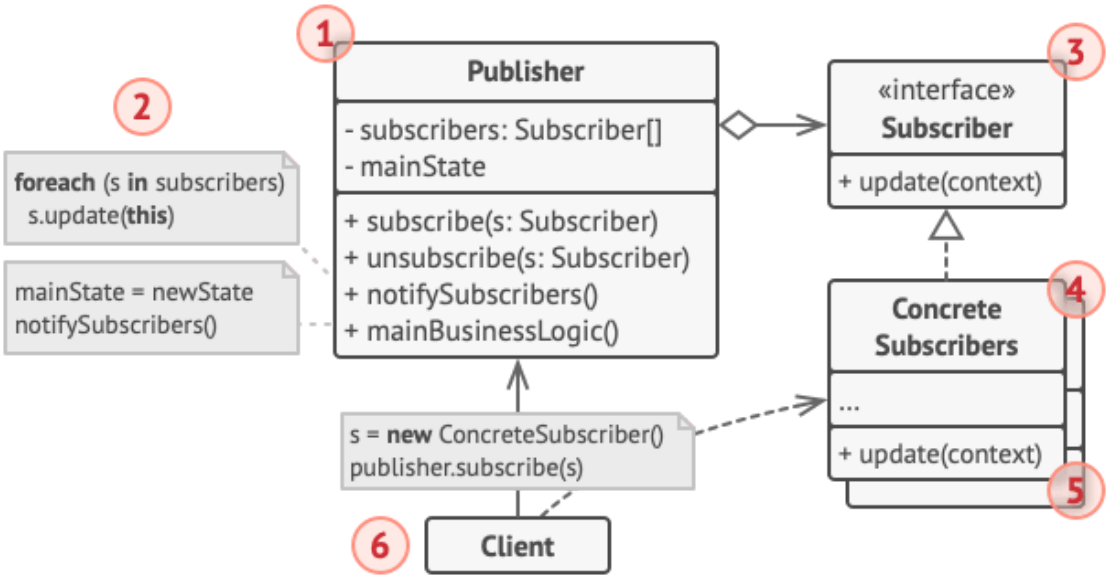
Nas interfaces gráficas, as persoas usuarias interaccionan cos distintos elementos da interface (botóns, caixas de texto, ventás, …) e a aplicación responde en consecuencia. De forma similar ao funcionamento do boletín de novas do parágrafo anterior, podería programarse unha rutina para comprobar frecuentemente se houbo algunha interacción. Sen embargo, sería mellor que os obxectos interesados recibisen unha mensaxe cada vez que se producise unha interacción. Este funcionamento é o que implementa Swing e denomínase programación dirixida por eventos.

O modelo de programación dirixida por eventos de Swing sigue o patrón de deseño **Observador/Observable**. Os compoñentes Swing son **Observables**. Os **Observadores** son obxectos interesados nos cambios producidos nos compoñentes **Observables.** Cando a persoa usuaria interacciona con un compoñente (observable), todos os **Observadores** son informados.

Funciona de forma similar a un sistema de subscrición a unha publicación: a publicación (*observable*) ten un sistema para subscribirse ou desuscribirse e as persoas subscritoras (*observador*) recibirán unha notificación cada vez que haxa unha nova publicación.



Explicación do funcionamento do patrón: a publicación (*observable*) ten métodos para subscribirse ou desuscribirse (1). Cando ocorre un novo evento, a publicación notifica a todos os subscritores (*observadores*) (2). Pode enviarse información contextual na notificación aos subscritores (5 - context).



En Swing, por exemplo, a clase **JButton** (*Publisher - observable*) ten un método equivalente a *subscribe* que é addActionListener(ActionListener l), que sería o equivalente a facer a subscrición. Este método recibe como parámetro un obxecto **ActionListener** (*Subscriber - observador*), tamén chamado manexador de evento. O obxecto ActionListener (*Subscriber*) ten que implementar a interface ActionListener, é dicir, implementar o método **actionPerformed,** que se executará cada vez que se produza o evento (sería o equivalente a *update* do diagrama anterior).

```
public interface ActionListener {
    void actionPerformed(ActionEvent e)
}
```

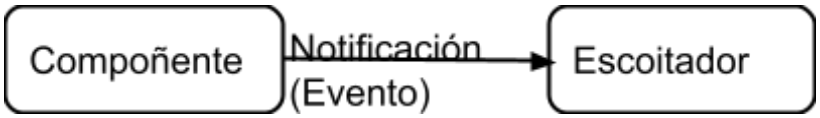
Este mecanismo permite que, cando unha persoa pulse o botón, se notifique a todos os manexadores de eventos rexistrados (listeners ou subscritores) executándose todos os seus métodos **actionPerformed**.

Pode enviarse información contextual (*context* no diagrama anterior). No caso de Swing envíase o **evento** xerado a todas as clases rexistradas como escoitadoras. O **evento** contén información sobre a acción que sucedeu.

O evento en Swing está encapsulado nun obxecto de tipo **Event**, que dependerá do tipo de acción realizada: ActionEvent, FocusEvent, KeyEvent, MouseEvent, MouseWheelEvent, etc.

Cando se executa unha aplicación Swing, cada interacción das persoas usuarias coa interface gráfica xera un ou máis eventos. As persoas programadoras escriben o código de resposta a estes eventos. O proceso que se leva a cabo é o seguinte:

- As persoas interaccionan sobre un compoñente.
- Créase un evento que describe o que ocorreu (información contextual).
- Notifícase ás clases escoitadoras rexistradas enviando a información do sucedido no obxecto Evento.
- Cada clase escoitadora responde ao evento, executando o método correspondente.



Poñendo un exemplo concreto dun evento que se produce nun botón:

- Compoñente: botón (JButton).
- Evento: ActionEvent
- Rexistro do escoitador (subscribirse): addActionListener(Escoitador)
- Obxecto escoitador: … implements ActionListener

Fixarse que se o nome da interface é xxxListener, o evento que escoita é xxxEvent e o método de rexistro é addxxxListener(xxxListener). Exemplo: ActionListener -> ActionEvent -> addActionListener(ActionListener).

Os principais Listeners en Swing, que serían equivalentse a posibles subscritores, son:

- **ActionListener**: detecta a acción típica sobre o compoñente. Por exemplo, para un botón será pulsalo, para un JTextField será pulsar INTRO, para un JComboBox será seleccionar unha opción, etc.
- **KeyListener**: detecta cando se pulsa unha tecla sobre o compoñente.
- **FocusListener**: detecta cando o compoñente recibe o foco.
- **MouseListener**: detecta diferentes eventos do rato sobre un compoñente: premer, soltar, clic, entrar ou saír co rato sobre o compoñente.
- **MouseMotionListener**: detecta o movemento do rato sobre un compoñente.
- **ItemListener**: detecta o cambio de estado dun compoñente, como un checkbox, combo box, toggle buttons, …

Para cada un dos Listeners anteriores é interesante observar cales son os métodos que proporciona a súa interface.

Máis información sobre os listeners que soportan os compoñentes Swing. Ao final da ligazón aparece unha táboa coa lista de compoñentes de Swing e os seus listeners asociados.

Rexistro de Listeners

En cada compoñente para o que se queiran escoitar eventos deberase rexistrar a clase de Listener que o procesará. Para isto será necesario executar o método **.add<TipoDe>Listener,** e como parámetro debe pasarse unha clase que implemente o tipo de Listener concreto.

Hai varias formas de facer isto por exemplo, usando unha clase anónima:

```
aceptarButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        textArea.append("Ola\n");
    }
});
```

Para asignar o mesmo Listener a varios compoñentes, é mellor crear unha nova clase usando un código similar ao seguinte:

```
ActionListener al = new ActionListener() {
   public void actionPerformed (ActionEvent e){
      // accións
    }
};

aceptarButton.addActionListener(al);
```

Outra forma de rexistrar un Listener é facer que a clase principal implemente ActionListener.

```
public class MyWindow extends JFrame implements ActionListener {

  public MyWindow() {
     …
     aceptarButton.addActionListener (this);
     …
  }

  public void actionPerformed (ActionEvent e) {
       // accións a executar cando se pulse o botón
  }
}
```

Máis información de como implementar un ActionListener.

Cando se utiliza o mesmo listener para varios compoñentes, por exemplo para varios botóns, é necesario un mecanismo para saber en que botón se produciu o evento. Para iso utilizarase o parámetro recibido polo método (**actionPerformed** no exemplo anterior)

implementado no Listener que é o parámetro evento (**ActionEvent** no exemplo). A partir deste evento pode obterse o obxecto no que se realizou a acción de diferentes formas:

- mediante o método **getSource()** do evento obtense o compoñente sobre o que se realizou a acción:

```
public void actionPerformed (ActionEvent e) {

    // Se sabemos que é un botón, pódese facer o cast
    JButton fonte = (JButton) e.getSource();

}
```

- outra forma de saber o compoñente no que se produciu o evento é indicar, para cada compoñente cal é o seu **comando**. Exemplo:

```
// Borrar é un texto calquera que non é visible na interface gráfica
boton.setActionCommand("Borrar");
…
public void actionPerformed (ActionEvent e) {
    String comando = e.getActionCommand();
    if (comando.equals("Borrar"))
        ….
}
```

  Por defecto, o “actionCommand” dun botón é igual á súa etiqueta, aínda que pode cambiarse con setActionCommand(...). É posible asignar o mesmo comando a diferentes compoñentes cando se quere que realicen a mesma acción.

Os Listeners están definidos como unha interface. Unha clase que implemente a interface ten que proporcionar unha implementación para **todos** os métodos da interface. Haberá veces que non é necesario implementar todos os métodos. Por exemplo, a interface **MouseListener** ten 5 métodos (mouseClicked, mouseEntered, mouseExited, mousePressed e mouseReleased). Se só queremos implementar un método no listener, pódense usar as clases **adaptadoras.**

As clases adaptadoras (**Adapters**) son clases abstractas que proporcionan unha implementación baleira de todos os métodos. Poden crearse unha clase que estenda de **Adapter** e sobreescribir só o método que interese.

```
MouseAdapter ma = new MouseAdapter() {
        @Override
        public void mouseClicked(MouseEvent e) {
            System.out.println("Botón pulsado");
        }
};
```

Exercicios:

1. Crea un novo frame que conteña un botón, de tal forma que cando se pulse se saia da aplicación.
2. Crea un novo frame con FlowLayout que teña tres botóns. Ao pulsar cada botón debe cambiarse o título do frame informando do botón pulsado. **Debes utilizar un único Listener**.
3. Modifica o frame anterior para que cando o rato pase por riba dun botón este cambie a cor de fondo. E cando o rato deixa de estar sobre o botón, este volve a ter a cor de fondo por defecto.
4. Crea un novo frame con FlowLayout que teña dous botóns e unha etiqueta. Engade o código necesario para que cando se pulse un botón se informe sobre cal foi o botón pulsado no texto da etiqueta.
5. Engade ao exercicio anterior un evento de tal forma que ao pulsar sobre o panel de contido se cambie a cor de fondo. É dicir, a cor de fondo debe cambiar cada vez que se fai clic sobre o panel, cambiando entre a cor orixinal e outra que escollas.

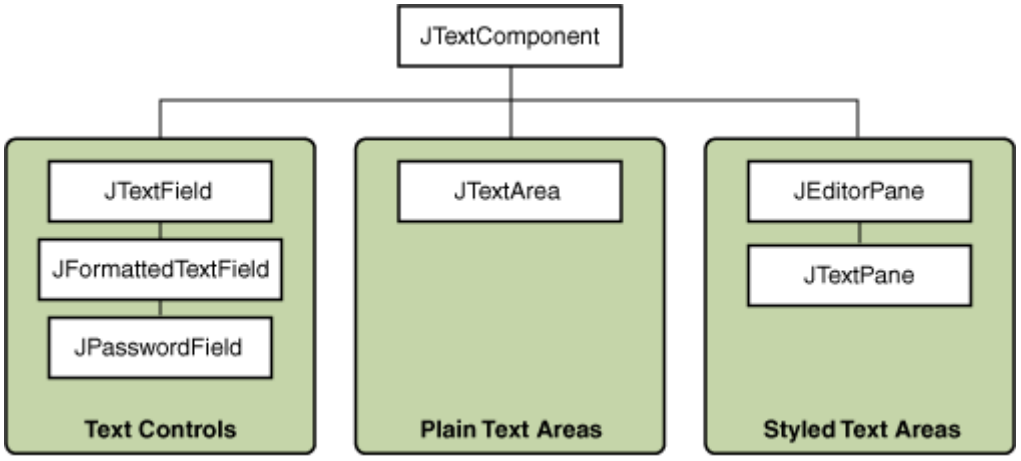
NOTA: cando unha clase anónima interna se define dentro do corpo dun método, todas as variables declaradas como **final** (ou efectivamente finais, é dicir, que o seu valor non se cambia) no ámbito do método son accesibles dende a clase interna. Recordar que unha variable **final** non se pode cambiar de valor. Se se trata dun obxecto, si que será posible cambiar as súas propiedades.

NOTA: unha clase anónima pode acceder ás variables de instancia da clase que a contén. Sen embargo non pode acceder a variables locais que non estean declaradas como final ou sexan realmente finais. Máis información.

```
public MainFrame() {
   final int variable = 5;
   …
   aceptarButton.addActionListener(new ActionListener() {
     @Override
     public void actionPerformed(ActionEvent e) {
       // O seguinte código funciona se a variable é final ou “effectively final”.
       System.out.println(variable);
     }
   });

   …
 }
```

Compoñentes de texto

Os compoñentes de texto permiten mostrar texto e, opcionalmente, permiten editalo. Algúns deles traballan con texto sen aplicar estilos (JTextField, JFormattedTextField, JPasswordField, JTextArea), mentres que outros permiten aplicar estilo ao texto (JEditorPane e JTextPane).



Unha caixa de texto (JTextField) é un compoñente de texto básico que permite ás persoas usuarias escribir un pequeno texto. Se o que se necesita é un compoñente con varias liñas de texto, mellor usar un **JTextArea**.

Para establecer o texto do compoñente e lelo, utilízanse os métodos **getText** e **setText** respectivamente.

Os eventos máis usados con un JTextField son:

- **ActionEvent**: sucede cando se pulsa a tecla ENTER sobre o JTextField.
- **FocusEvent**: sucede cando o JTextField gaña ou perde o foco.

```
textField.addFocusListener(new FocusListener() {
   public void focusLost(FocusEvent e) {
      …
   }

   public void focusGained(FocusEvent e) {
      ...
   }
});
```

Outros compoñentes de texto:

- **JFormattedTextField** permite especificar un conxunto de caracteres permitidos como entrada. Concretamente, o JFormattedTextField engade un **formateador** e un obxecto **valor** ás características herdadas de JTextField. O formateador traduce o valor ao texto a mostrar en pantalla e viceversa.

  Usando os formateadores que proporciona Swing, poden especificarse caixas de texto para datas ou números que usen formato local. Outros formateadores permiten definir unha máscara para especificar o conxunto de caracteres válidos. Por exemplo, pode especificarse un formato para escribir números de teléfono como (XX)XXX-XXX-XXX. Máis información de como usar JFormattedTextField.

  Un JFormattedTextField terá as propiedades text e value. Mentres a persoa usuaria teclea, a propiedade text cambia. A propiedade value non cambiará ata que se confirmen os cambios.

  Para obter o valor actual usarase o método **getValue**.

```
private JFormattedTextField amountField;
private double amount;

amountFormat = NumberFormat.getNumberInstance();
amountField = new JFormattedTextField(amountFormat);
amountField.setValue(0.5);

…
// obter o valor do campo de texto
amount = ((Number) amountField.getValue()).doubleValue();
```

- **JPasswordField** proporciona unha caixa de texto para escribir o contrasinal, de tal forma que non mostra os caracteres que se pulsan.

Exercicios:

1. Crea un novo frame que conteña dúas caixas de texto, un botón co texto “Sumar” e unha etiqueta. Programa a funcionalidade da aplicación para que cando se pulse o botón “Sumar” a etiqueta mostre o resultado de sumar os dous números existentes nas caixas de texto.

2. Crea unha aplicación que conteña unha caixa de texto non editable e un botón. A caixa de texto mostrará inicialmente o valor 0. Cada vez que se pulse o botón, fai que o valor da caixa de texto se incremente en 1 unidade, de tal forma que mostre información do número de veces que se pulsou o botón. Ademais, engade outro botón á aplicación para reinicializar o contador da caixa de texto a 0.

3. Crea unha aplicación que teña unha etiqueta, un campo de contrasinal e un botón. Debe comprobarse se o contrasinal escrito coincide con un que ti escollas. Esta comprobación farase ao pulsar o botón e tamén ao pulsar a tecla ENTER cando o campo contrasinal ten o foco. Se os contrasinais coinciden o fondo do panel de

contido cambiará a cor verde e se non coinciden porase de cor vermella. Debe usarse un único Listener.

4. Crea unha aplicación que conteña un área de texto con 5 filas e 20 columnas. Executa a aplicación e comproba que pasa cando escribes máis liñas das que se ven en pantalla.

   Modifica a aplicación para que a área de texto estea dentro dun scrollPane, de forma que aparezan as barras de scroll cando o contido do texto supere a área visible.

   Comproba tamén que pasa cando a liña que escribes é máis grande que as columnas visibles en pantalla.

   Busca como facer para que cando o texto dunha liña sexa máis grande que o número de columnas, o texto pase automaticamente á liña de abaixo. Configúraa para que as palabras non se rompan.

5. Crea unha aplicación que conteña un campo de texto e unha área de texto **non editable**. Fai que o texto escrito na caixa de texto se concatene ao final da área de texto cando se pulsa INTRO (tendo a caixa de texto o foco). Ademais, todo o texto da caixa de texto quedará seleccionado despois de concatenalo na área de texto.

6. Crea unha aplicación que conteña dous JFormattedTextField coas súas correspondentes etiquetas e un botón. Configura unha das caixas de texto para que teña números e a outra para que almacene unha data. Inicializa ambas caixas de texto a un valor apropiado. A aplicación permitirá modificar os valores das caixas de texto e cando se pulse o botón, imprimiranse por consola os seus valores.

7. Crea unha aplicación con unha etiqueta e unha caixa de texto para escribir un código postal. Utiliza un JFormattedTextField configurado para poder introducir só un número de 5 cifras.

Layout - deseño

Un **Layout** pode definirse como a forma na que se colocarán as etiquetas, botóns, cadros de texto e demais compoñentes nun contedor.

Para facer deseños de forma máis cómoda, Java proporciona unha serie de obxectos denominados **Layouts**, que determinan a distribución que terán os elementos ao situarse nun contedor (Frame, Dialog, Panel, …).

O Layout tamén é o encargado de axustar o tamaño dos compoñentes adecuadamente cando cambia o tamaño da ventá. O tamaño real dun compoñente é calculado en función das súas propiedades **minimunSize**, **maximumSize** e **preferredSize**, que son as propiedades que hai que modificar para configurar o tamaño desexado do compoñente. Aínda que existe o método setSize() non se recomenda usalo, pois é o layout o que determina o tamaño final dos compoñentes. É máis recomendable indicar o tamaño desexado para un compoñente usando o método setPreferredSize (tamaño preferido).

Tamén é posible establecer o **aliñamento** entre compoñentes, por exemplo, pode ser necesario que dous compoñentes estean aliñados polo borde superior. Este aliñamento pode establecerse invocando os métodos setAlignmentX e setAlignmentY do compoñente. Ainda que a maioría dos layouts ignoran o aliñamento, esta é unha propiedade importante no BoxLayout.

A dirección na que se colocan os compoñentes vén determinada pola propiedade **componentOrientation** do contedor, que pode ter un dos seguintes valores:

- ComponentOrientation.LEFT_TO_RIGHT
- ComponentOrientation.RIGHT_TO_LEFT

O habitual será establecer o layout dun **JPanel** ou do **panel de contido (content pane)** do JFrame.

A clase **JPanel** é un contedor xenérico para compoñentes. Ao mesmo tempo, un JPanel tamén é un compoñente, polo que se pode engadir a outro contedor. Poderán crearse deseños complexos usando varios paneis con diferentes configuracións de layout.

Por defecto o panel de contido (Content pane) dun JFrame ten asignado un **BorderLayout** e un JPanel ten asignado un **FlowLayout**.

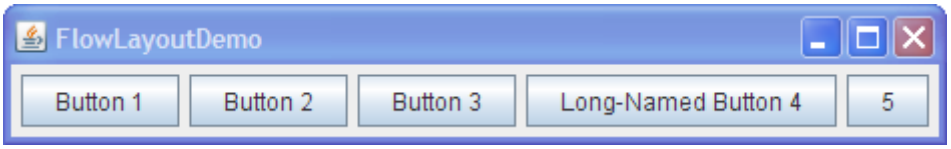
Tamén é posible deseñar interfaces gráficas sen layout, estratexia denominada posicionamento absoluto onde se debe especificar o tamaño e posición de cada compoñente no contedor. A desventaxa desta estratexia é que non se axusta adecuadamente cando o contedor se redimensiona e tampouco funciona ben en diferentes sistemas con tamaño de fonte diferentes.

A continuación detállanse algúns dos layouts máis utilizados. Pode encontrarse información de como utilizar layouts na documentación de Java.

NOTA: Nesta documentación non se van tratar o GroupLayout nin o SpringLayout por ser layouts deseñados para usar con ferramentas gráficas de elaboración de interfaces.

FlowLayout

FlowLayout é o layout por defecto de JPanel. Simplemente coloca os compoñentes nunha única fila, empezando unha nova fila se o contedor non ten ancho suficiente. É dicir, os compoñentes son colocados de maneira similar á colocación das palabras nun parágrafo dun procesador de textos.



O aliñamento da fila vén determinado pola propiedade align de FlowLayout que pode ter os valores: LEFT, RIGHT, CENTER, LEADING ou TRAILING.

Se o contedor cambia de tamaño en tempo de execución, as posicións dos compoñentes axústanse automaticamente, para colocar o máximo número posible de compoñentes na primeira liña.

Construtor:

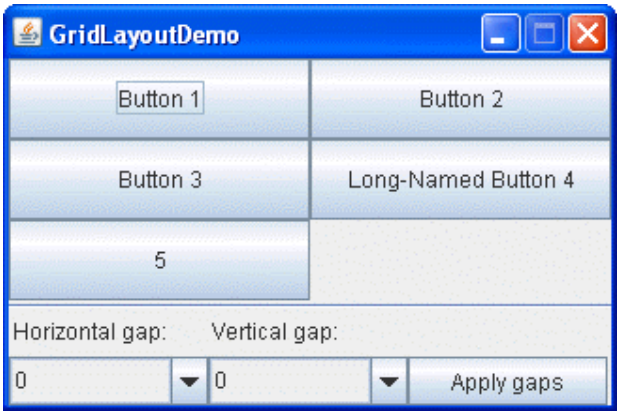
- FlowLayout(): constrúe un FlowLayout con aliñamento centrado e espazo horizontal e vertical de 5 unidades.

Métodos:

- setAlignment(int align)
- setHgap(int hgap)
- setVgap(int vgap)

GridLayout

GridLayout permite colocar os compoñentes nunha matriz de celas. Cada compoñente utiliza todo o espazo dispoñible na súa cela e todas as celas son do mesmo tamaño.



A orde na que se engaden os compoñentes determina a súa posición na matriz.

Cando a ventá se redimensiona, o tamaño da cela tamén se adapta automaticamente.

Construtor:

- GridLayout(int rows, int cols): crea un GridLayout co número especificado de filas e columnas. Se o número de filas é 0, iranse engadindo filas a medida que se necesiten para colocar os compoñentes. O mesmo pasa se o número de columnas é 0, pois iranse engadindo a medida que se necesiten.

Métodos:

- setHgap(int hgap)
- setVgap(int vgap)

BorderLayout

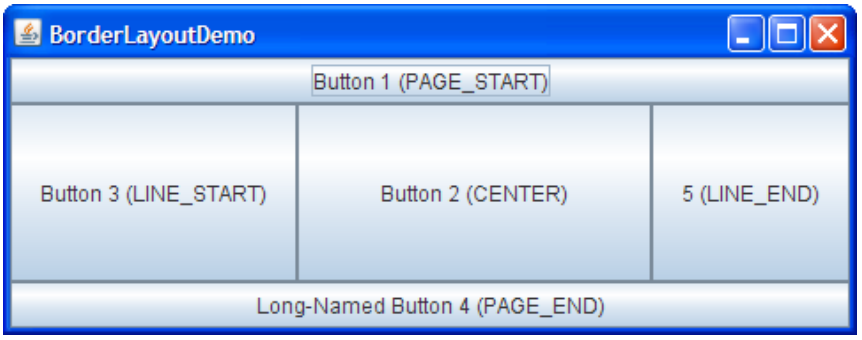
Por defecto, este é o Layout que utilizan os paneis de contido (content pane: panel principal de frames, applets e diálogos).

O BorderLayout coloca os compoñentes en ata 5 áreas: norte, sur, este, oeste e centro:

- A zona norte e sur estíranse de esquerda a dereita ata ocupar toda a ventá. En vertical ocupa o tamaño necesario para albergar os seus compoñentes.
- As zonas este e oeste estíranse de arriba abaixo ata ocupar todo o espazo dispoñible. O ancho destas zonas é o necesario para albergar o seu contido.
- A zona centro estírase en ambas direccións ata tocar as outras zonas.

A partir de JDK 1.4 prefírese usar como nomes das áreas PAGE_START, PAGE_END, LINE_START, LINE_END e CENTER. Estas constantes son preferidas porque permiten aos programas axustarse a idiomas que utilizan diferentes orientacións.

Cada unha destas áreas só pode ter un compoñente. Un compoñente pode ser un contedor, o que permitirá conter varios compoñentes ao mesmo tempo.



Construtor:

- BorderLayout(): constrúe un BorderLayout sen espazo entre compoñentes.

Cando se engade un compoñente hai que indicar a área onde engadilo, por exemplo:

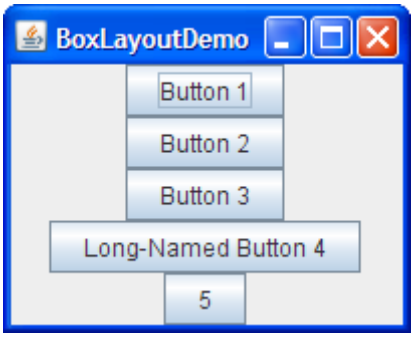
- panel.add(component, BorderLayout.CENTER);

BoxLayout

BoxLayout coloca os compoñentes nunha única fila ou columna, dependendo do valor usado no parámetro “axis” do construtor.

Á hora de colocar os compoñentes, tense en conta o seu aliñamento e o habitual é que todos os compoñentes nun BoxLayout teñan o mesmo aliñamento.

Ao diferencia do GridLayout, aquí os compoñentes poden ter diferente tamaño.



Construtor:

- BoxLayout(target, axis): crea un BoxLayout que coloca os compoñentes no eixo especificado como parámetro.
    - target: contedor ao que aplicar o layout
    - axis: eixo ao longo do cal se dispoñen os compoñentes:
        - BoxLayout.X_AXIS: horizontalmente de esquerda a dereita
        - BoxLayout.Y_AXIS: verticalmente de arriba abaixo
        - BoxLayout.LINE_AXIS: na mesma dirección que as palabras nunha liña, baseado na propiedade ComponentOrientation do contedor.
        - BoxLayout.PAGE_AXIS: na mesma dirección que as liñas nunha páxina, baseado na propiedade ComponentOrientation do contedor.

  LINE_AXIS e PAGE_AXIS teñen en conta linguaxes que se escriben de dereita a esquerda ou de arriba abaixo.

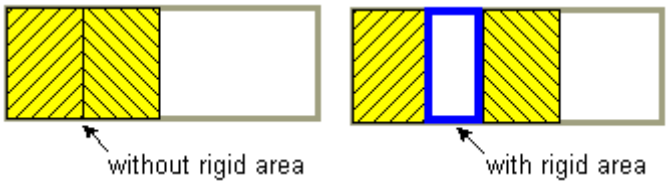
Exemplo:

buttonPane.setLayout(new BoxLayout(buttonPane, BoxLayout.LINE_AXIS));

Os compoñentes engadidos a un BoxLayout están pegados entre si por defecto. Pode engadirse espazo entre componentes engadindo un borde ou insertando un compoñente invisible:

- Engadir un área ríxida de tamaño fixo entre dous compoñentes

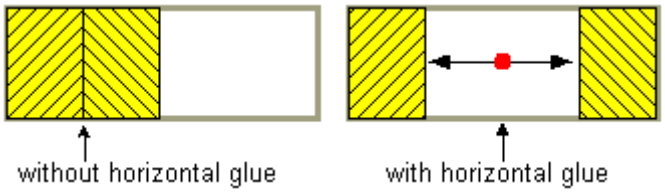
```
container.add(firstComponent);
container.add(Box.createRigidArea(new Dimension(5,0)));
container.add(secondComponent);
```



- Engadir o espazo sobrante entre compoñentes, podendo non ocupar espazo ou espandirse:

```
container.add(firstComponent);
```

```
container.add(Box.createHorizontalGlue());
container.add(secondComponent);
```



CardLayout

CardLayout permite implementar unha área que contén diferentes compoñentes en diferentes instantes de tempo. Normalmente un CardLayout é controlado por un combo box, que determina o panel que se mostra.

Conceptualmente, cada compoñente (ou contedor) dun CardLayout é como unha carta nunha pila, onde só a carta da cima é visible. Pode cambiarse a carta visible indicando que se mostre outra: a primeira, a última, a seguinte, a anterior ou especificando unha carta polo seu nome. Na API existen métodos para todas estas funcións.

Construtor:

- CardLayout(): crea un CardLayout sen ocos.

O seguinte trozo de código crea un obxecto CardLayout e os compoñentes que o xestionan:

```
//Where instance variables are declared:
JPanel cards;
final static String BUTTONPANEL = "Card with JButtons";
final static String TEXTPANEL = "Card with JTextField";

//Where the components controlled by the CardLayout are initialized:
//Create the "cards".
JPanel card1 = new JPanel();
...
JPanel card2 = new JPanel();
...

//Create the panel that contains the "cards".
cards = new JPanel(new CardLayout());
cards.add(card1, BUTTONPANEL);
cards.add(card2, TEXTPANEL);
```

Para escoller o compoñente a mostrar, usarase o seguinte código:

```
//Put the JComboBox in a JPanel to get a nicer look.
JPanel comboBoxPane = new JPanel(); //use FlowLayout
String comboBoxItems[] = { BUTTONPANEL, TEXTPANEL };
JComboBox cb = new JComboBox(comboBoxItems);
cb.setEditable(false);
cb.addItemListener(this);
comboBoxPane.add(cb);
...
pane.add(comboBoxPane, BorderLayout.PAGE_START);
pane.add(cards, BorderLayout.CENTER);
...

//Method came from the ItemListener class implementation,
//contains functionality to process the combo box item selecting
public void itemStateChanged(ItemEvent evt) {
    CardLayout cl = (CardLayout)(cards.getLayout());
    cl.show(cards, (String)evt.getItem());
}
```

Exemplo completo de utilización de CardLayout.

GridBagLayout

GridBagLayout é un layout sofisticado e flexible. Coloca os compoñentes nunha matriz de celas permitindo que un compoñente se espanda por máis dunha fila ou columna.

O GridBagLayout utiliza o tamaño preferido dos compoñentes (preferred size) para determinar o tamaño de cada cela.

Non todas as filas teñen que ter a mesma altura, nin todas as columnas teñen que ter o mesmo ancho. Unha fila ou columna pode estenderse para acomodarse ás dimensións do compoñente máis grande que conteña. Tamén é posible que os compoñentes se expandan por varias filas ou columnas. Se un compoñente é máis pequeno que unha cela pode configurarse a súa posición dentro da cela.

O seguinte exemplo mostra un GridBagLayout de 3 filas e 3 columnas. O botón da segunda fila espándese por todas as columnas e o botón da terceira fila espándese por 2 columnas. Observar tamén que cada compoñente ocupa todo o espazo horizontal, pero non o vertical (ver botón 5):

Cando se redimensiona a ventá, o novo tamaño é distribuído entre as filas e columnas en base a un sistema de pesos.

A posición dos compoñentes defínese mediante un par de coordenadas (x, y). Nun deseño con orientación horizontal de esquerda a dereita, a coordenada (0,0) estará na esquina superior esquerda. A coordenada x corresponderase co eixo horizontal e a coordenada y co eixo vertical.

A forma de especificar o tamaño e posición dun compoñente é asociarlle un obxecto **constraints**. É dicir, cada compoñente do GridBagLayout debe estar asociado con unha instancia da clase GridBagConstraints. A configuración das propiedades deste obxecto permite configurar a área onde se colocará o compoñente e a súa disposición.

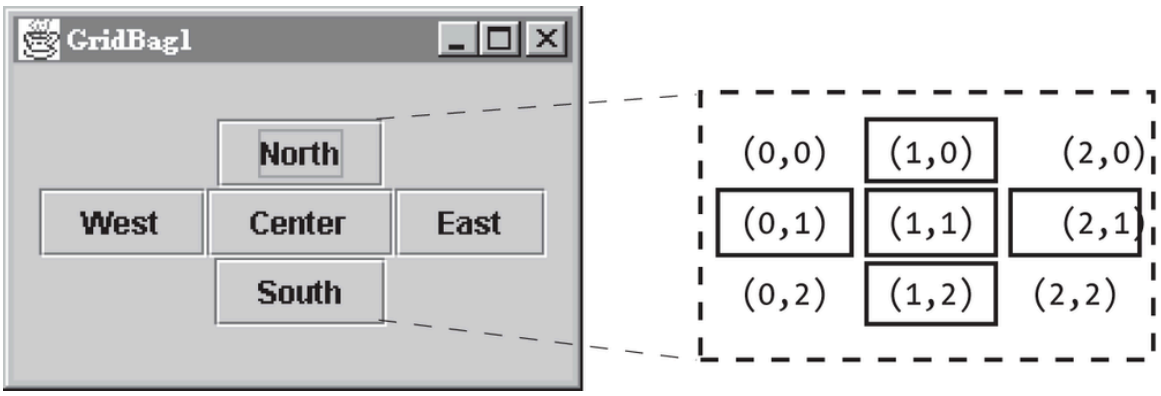
A forma máis fácil de asociar o GridBagConstraints con un compoñente é usar o método **add(component, constraints)**. Exemplo:

```
JButton button = new JButton("Aceptar");

GridBagConstraints gbc = new GridBagConstraints(...);
gbc.gridx = 0;
gbc.gridy = 0;
// Configurar o resto de atributos do GridBagConstraints

getContentPane().add(button, gbc);
```

A personalización dos valores das variables do obxecto GridBagConstraints permite establecer o deseño do GridBagLayout. É recomendable configuralas antes de engadir os compoñentes ao deseño. Algunhas destas variables están descritas a continuación:

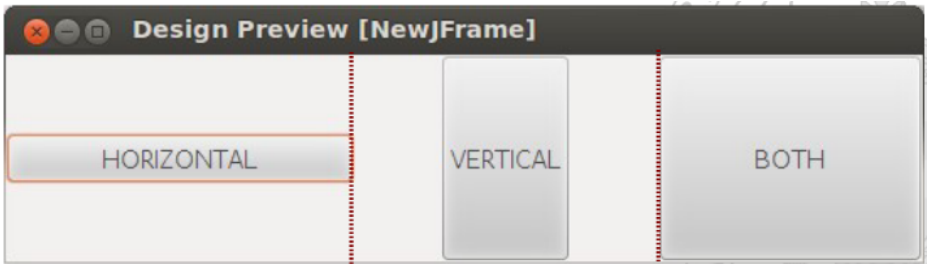
- **int gridBagConstraints.gridx, gridBagConstraints.gridy**: especifica a fila e columna onde se colocará o compoñente.

  Non hai forma de establecer o tamaño da matriz. A matriz terá as celas necesarias para albergar todos os compoñentes. Por exemplo, na imaxe seguinte o layout ten 3 filas e 3 columnas, porque así o determinaron os compoñentes. Ver exemplo.




```
GridBagConstraints constraints = new GridBagConstraints();

constraints.gridx = 0;
constraints.gridy = 1;
add(component, constraints);
```

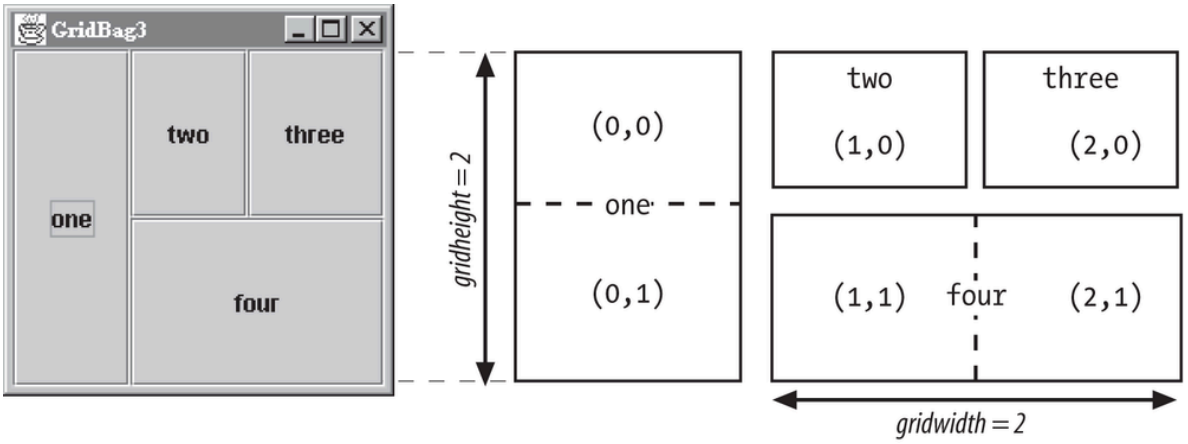
- **int gridBagConstraints.fill**: úsase para determinar como redimensionar o compoñente cando a cela é máis grande. Valores válidos:
    - GridBagConstraints.NONE - non se redimensiona (valor por defecto).
    - GridBagConstraints.HORIZONTAL - o compoñente ocupa todo o ancho.
    - GridBagConstraints.VERTICAL - o compoñente ocupa todo o alto.
    - GridBagConstraints.BOTH - ocupa todo o alto e ancho



NOTA: para que funcione é necesario establecer a un valor diferente de 0 as variables gridBagConstraints.**weightx** e gridBagConstraints.**weighty**. Ver exemplo.

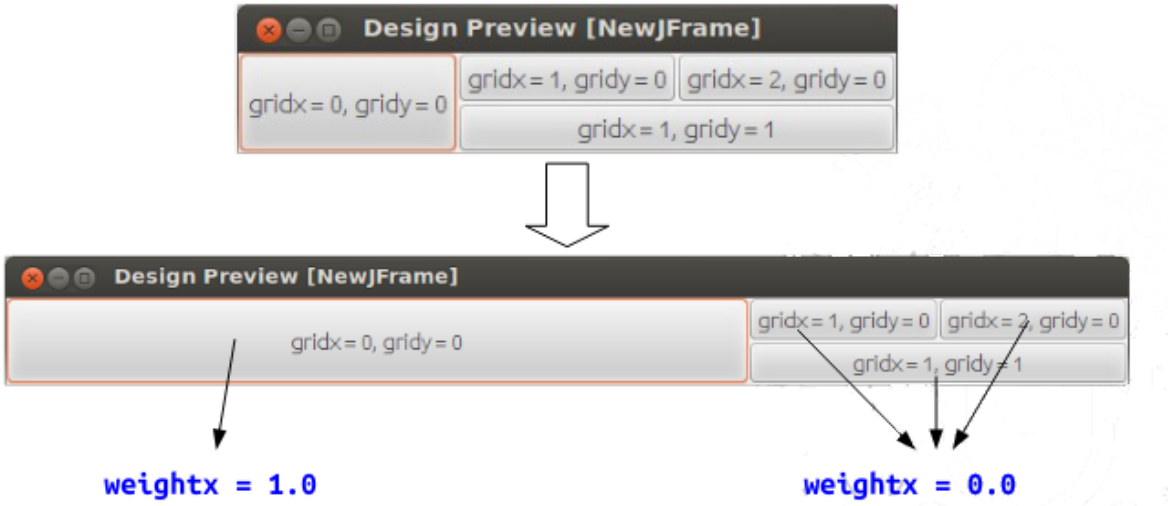
```
GridBagConstraints constraints = new GridBagConstraints();
constraints.weightx = 1.0;
constraints.weighty = 1.0;
constraints.fill = GridBagConstraints.BOTH;
constraints.gridx = 1;
constraints.gridy = 0;
add(component, constraints);
```

- **int gridBagConstraints.gridwidth, gridBagConstraints.gridheight**: especifica o número de columnas ou filas que ocupa o compoñente. O valor por defecto é 1. Exemplo.




```
GridBagConstraints constraints = new GridBagConstraints();
constraints.weightx = 1.0;
constraints.weighty = 1.0;
constraints.fill = GridBagConstraints.BOTH;
constraints.gridheight = 2;
constraints.gridx = 0;
constraints.gridy = 0;
add(component, constraints);
```

- **double weightx, weighty**. Úsase para determinar como distribuír o espacio sobrante entre as columnas e filas; isto é importante para determinar o comportamento dos compoñentes cando se redimensiona a ventá.



Os valores posibles son números positivos aínda que soen usarse valores entre 0.0 e 1.0. O valor en realidade non importa, o que importa é a súa relación co resto de valores das outras celas: valores máis altos indican que a fila/columna ocupará máis espacio.

Unha vez determinado o tamaño preferido dos compoñentes, incluíndo o padding e insets, o espazo sobrante repártese en función do peso dos compoñentes. Unha columna ou fila con valor 0 de peso, non recibirá espazo extra. Exemplo.

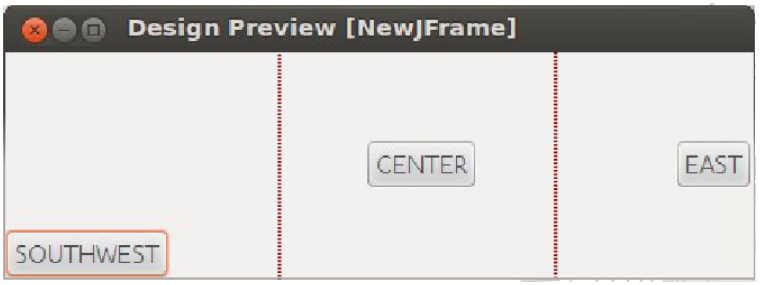
```
GridBagConstraints constraints = new GridBagConstraints();
constraints.weightx = 0.1;
constraints.weighty = 1.0;
constraints.fill = GridBagConstraints.BOTH;
constraints.gridheight = 2;
constraints.gridx = 0;
constraints.gridy = 0;
add(component, constraints);
```

NOTA: se non se especifica weightx ou weighty para ningún compoñente, xuntaranse todos no centro da ventá.

- **int gridBagConstraints.anchor**: úsase cando o compoñente é máis pequeno que a cela. Por defecto colócase centrado, aínda que poden establecerse outros posicionamentos: PAGE_START, PAGE_END, LINE_START, LINE_END, FIRST_LINE_START, FIRST_LINE_END, LAST_LINE_END e LAST_LINE_START.

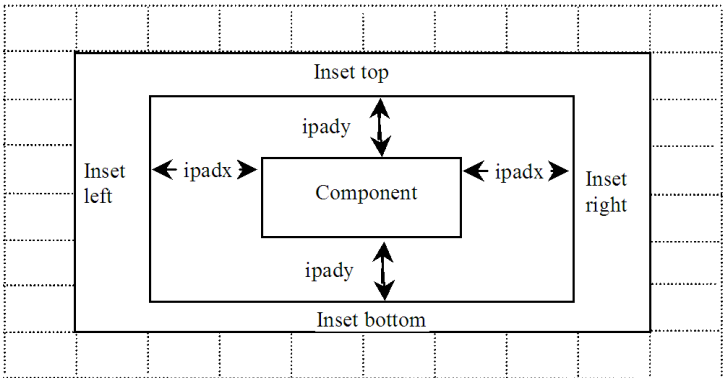
A seguinte imaxe mostra a interpretación destes valores nun contedor con orientación de esquerda a dereita:

FIRST_LINE_START	PAGE_START	FIRST_LINE_END
LINE_START	CENTER	LINE_END
LAST_LINE_START	PAGE_END	LAST_LINE_END



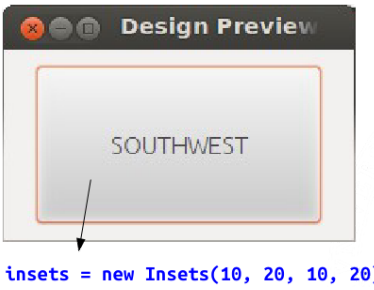

```
GridBagConstraints constraints = new GridBagConstraints();
constraints.anchor = GridBagConstraints.SOUTH;
add(component, constraints);
```

- **int gridBagConstraints.ipadx, gridBagConstraints.ipady**: especifica o padding interno dos compoñentes, que fará incrementar o tamaño do compoñente. O valor por defecto é 0. Exemplo.

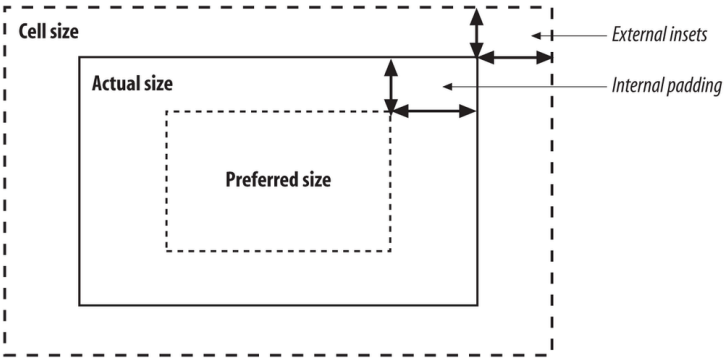



```
GridBagConstraints constraints = new GridBagConstraints();
constraints.ipadx = 25;  // add padding
constraints.ipady = 25;
add(component, constraints);
```

- **Insets gridBagConstraints.insets**: especifica o espazo externo, é dicir, espazo entre o compoñente e o borde da cela.



A relación entre o inset e o padding pode ser confusa. Tal e como se mostra na seguinte imaxe, o padding engádese ao compoñente, resultando nun incremento do seu tamaño. O inset é externo ao compoñente e representa a marxe entre o compoñente e a súa cela.




```
GridBagConstraints constraints = new GridBagConstraints();
constraints.insets = new Insets(10,0,0,0);  //top padding
add(component, constraints);
```

NOTA: Recoméndase configurar as propiedades do obxecto GridBagConstraints antes de engadir un compoñente.

O seguinte código mostra un exemplo completo.

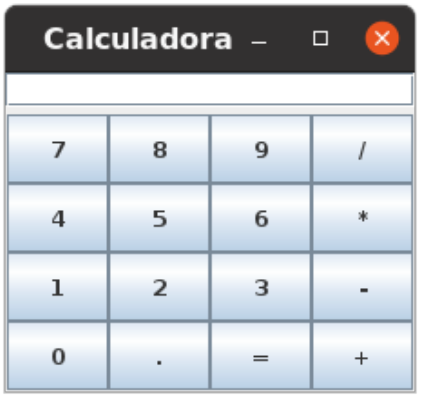
```
public class GridBagLayoutExample extends JFrame {
    public GridBagLayoutExample() {
        GridBagConstraints gbc = new GridBagConstraints();
        GridBagLayout layout = new GridBagLayout();
        this.setLayout(layout);

        // Primeira fila
        gbc.gridx = 0;
        gbc.gridy = 0;
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.ipady = 20;
        gbc.weightx = 0.5;
        this.add(new Button("Button One"), gbc);
```

```
        gbc.gridx = 1;
        gbc.gridy = 0;
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.ipady = 0;
        gbc.weightx = 0.5;
        this.add(new Button("Button Two"), gbc);

        // Segunda fila
        gbc.gridx = 0;
        gbc.gridy = 1;
        gbc.fill = GridBagConstraints.BOTH;
        gbc.gridwidth = 2;
        gbc.weightx = 1;
        gbc.weighty = 1;
        this.add(new Button("Button Three"), gbc);

        pack();
        setVisible(true);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

  public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new GridBagLayoutExample();
      }
    });
  }
}
```

Exercicios:

1. Crea unha interface utilizando o BorderLayout que teña 5 botóns na zona esquerda e outros 5 na zona dereita. Os botóns deben estar colocados en vertical e ter todos o mesmo ancho, independentemente do texto que conteñan.
   Engade á interface un título co contido que queiras na zona de arriba (norte), un pé abaixo centrado (sur) que inclúa o teu nome e un jTextArea no espacio central (centro).
2. Como ampliación e modificación do exercicio anterior, substitúe a área de texto da parte central por un panel con un CarLayout de 3 cartas (3 paneis de diferentes cores, as que ti queiras). Engade aos botóns laterais a funcionalidade que permita moverse polas diferentes cartas. Así, os botóns deben permitir moverse á carta **anterior, primeira, segunda, terceira** e **seguinte.** Adecúa o texto do botón á súa funcionalidade.

3. Deseña unha interface coma a da seguinte imaxe (non ten que ter funcionalidade). Configúraa para que ao redimensionar a ventá, manteña as proporcións da imaxe.



4. Usando o GridBagLayout, deseña un formulario coma o das seguintes imaxes. A imaxe da dereita mostra unha ventá máis grande e obsérvase que os campos do formulario permanecen na parte superior:

Fíxate que:

- as etiquetas están aliñadas á dereita.
- as etiquetas teñen un padding externo á dereita de 5px.
- ao facer a ventá máis grande, os campos permanecen na parte superior da ventá.

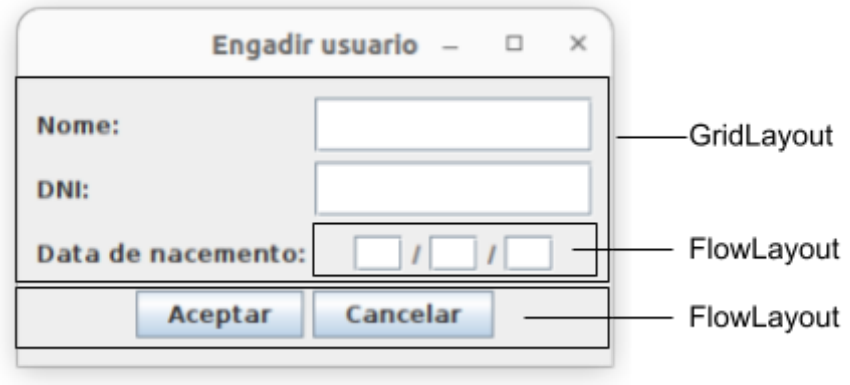
NOTA: o formulario anterior ten un borde. Como crear bordes.

```
JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createTitledBorder("title"));
```

Como o borde queda moi pegado ao extremo da ventá, pode engadirse un borde baleiro pola parte externa para separalo:

```
Border innerBorder = BorderFactory.createTitledBorder("Add Person");
Border outerBorder = BorderFactory.createEmptyBorder(5, 5, 5, 5);

pane.setBorder(BorderFactory.createCompoundBorder(outerBorder, innerBorder));
```

5. Deseña unha aplicación coma a da seguinte imaxe. Ademais, o panel de contido do JFrame ten un deseño BorderLayout, de tal forma que os botóns “Aceptar” e “Cancelar” están na zona sur do panel de contido:



Organizando a aplicación

Cando as aplicacións empezan a medrar e teñen grande cantidade de compoñentes é boa idea telas ben organizadas en compoñentes independentes, usando paneis personalizados para crear layouts complexos.

Imos partir da aplicación creada no apartado “Frame personalizado” que contiña un JFrame (MainFrame) con deseño BorderLayout cun área de texto na parte central e un botón na parte de abaixo.

A partir de agora configurarase o MainFrame para que estea formado de varios compoñentes implementados por separado. Esta forma de programar facilita a creación e mantemento de aplicacións grandes e complexas ao permitir separar os compoñentes para que se manteñan o máis independentes posible.

Swing proporciona a clase **JPanel** que é un contedor xenérico para compoñentes. Ao mesmo tempo, un JPanel tamén é un compoñente, polo que se pode engadir a outro contedor. Imos modificar a aplicación de exemplos anteriores para crear un panel personalizado (clase que estende de JPanel) e que conterá a área de texto.

Crearemos unha nova clase chamada TextPanel que estende de JPanel:

```
public class TextPanel extends JPanel {
    private JTextArea textArea;

    public TextPanel() {
        textArea = new JTextArea();

        // Por defecto, un panel ten FlowLayout
        setLayout(new BorderLayout());

        // JScrollPane engadirá barras de scroll en caso necesario
        // Ao non haber máis compoñentes no panel, a área de texto ocupará todo o espazo
        add(new JScrollPane(textArea), BorderLayout.CENTER);
    }
```

```
    public void appendText(String text) {
        textArea.append(text);
    }
}
```

Para usar o TextPanel anterior, temos que modificar o antigo MainFrama.java:

```
public class MainFrame extends JFrame {
    private TextPanel textPanel;
    private JButton aceptarButton;

    public MainFrame() {
        super("Ola Mundo!");

        // Establecer o Layout
        setLayout(new BorderLayout());

        // Inicializar os compoñentes
        textPanel = new TextPanel();
        aceptarButton = new JButton("Aceptar");

        aceptarButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                textPanel.appendText("Hello\n");
            }
        });

        // engadir os compoñentes
        add(textPanel, BorderLayout.CENTER);
        add(aceptarButton, BorderLayout.PAGE_END);

        setSize(600, 500);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }
}
```

Exercicios:

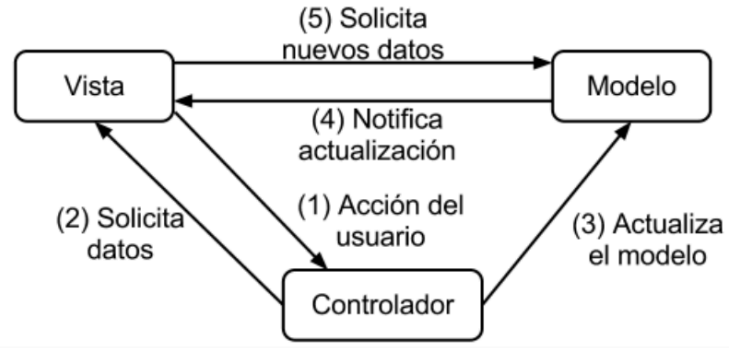
1. Modifica a aplicación do exemplo para que teña un panel personalizado na área norte do MainFrame. Este panel personalizado vai facer a función de barra de ferramentas (Toolbar). Engade ao toolbar dous botóns. O panel personalizado debe ter un deseño FlowLayout. A aplicación debe quedar parecida á da seguinte imaxe:

Patrón Modelo - Vista - Controlador

O patrón MVC (Modelo - Vista - Controlador) axuda a desacoplar as clases cando se programan interfaces gráficas.

A idea é agrupar as clases segundo a súa funcionalidade:

- **Modelo**: clases responsables de manter e xestionar os datos.
- **Vista**: clases responsables de visualizar datos.
- **Controlador**: clase que conecta a Vista co Modelo. Tamén implementa a funcionalidade da aplicación. É o responsable de facer os cambios no modelo cando as persoas interaccionan coa vista.



A vantaxe do MVC é o desacoplamento dos datos da súa visualización. É dicir, o aspecto da Vista pode cambiar sen necesidade de modificar o modelo e viceversa, ou pode cambiar o almacenamento dos datos de ficheiro a base de datos, sen afectar á Vista.

No MVC o Modelo só coñece a Vista. A Vista coñece o Controlador e o Modelo. O Controlador coñece a Vista e o Modelo.

Para implementar este patrón hai que facer que as clases teñan referencias a interfaces, en lugar de a clases concretas. Deste modo desacóplase a referencia da implementación concreta.

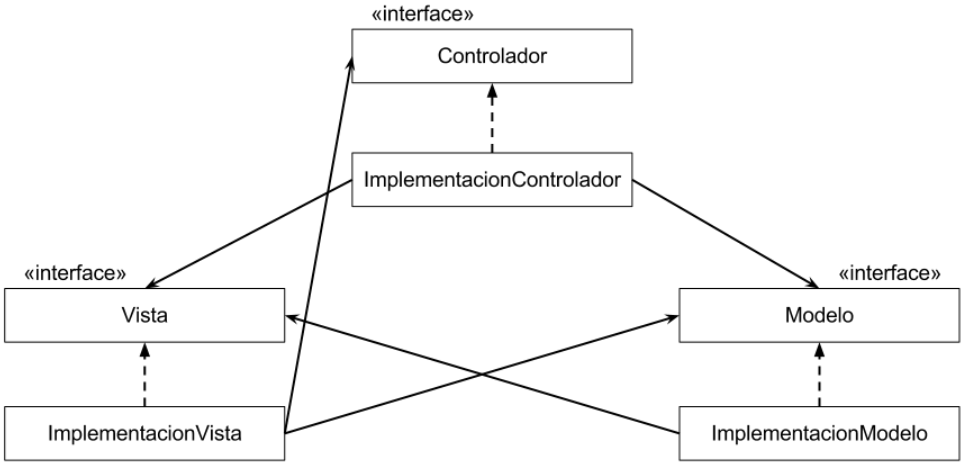
```
public class ImplementacionModelo implements Modelo {
    private Vista vista; //Vista é unha interface
    ...
}
```

```
public class ImplementacionVista implements Vista {
    private Controlador controlador; //Controlador é unha interface
    private Modelo modelo; //Modelo é unha interface
    ...
}
```

```
public class ImplementacionControlador implements Controlador {
    private Modelo modelo; //Modelo é unha interface
    private Vista vista; //Vista é unha interface
    ...
}
```

Os atributos poden establecerse no construtor ou con métodos set.

Diagrama UML onde cada clase concreta mantén referencias de tipo interface:



Máis información do MVC con un exemplo de implementación.

Separación entre compoñentes

A aplicación usada en exemplos anteriores contén un panel personalizado para a área de texto e outro panel para a barra de ferramentas. Se queremos que ao pulsar o botón “Hello”, da barra de ferramentas se escriba algo na área de texto, unha forma de implementalo é pasar á barra de ferramentas unha referencia á área de texto para que teña acceso directo

a ela e poida acceder aos seus métodos. Así, cando sexa necesario, poderase executar o método **textPanel.appendText(...)** dende a barra de ferramentas:

```
public class Toolbar extends JPanel {
    private TextPanel textPanel;
    private JButton helloButton;
    private JButton helloButton;

    public Toolbar() {
        helloButton = new JButton("Hello");
        goodbyeButton = new JButton("Goodbye");

        ActionListener al = new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JButton clickedButton = (JButton) e.getSource();

                if (clickedButton == helloButton) {
                  textPanel.appendText("Hello\n");
                } else if (clickedButton == goodbyeButton) {
                  textPanel.appendText("Goodbye\n");
                }
              }
        };

        helloButton.addActionListener(al);
        goodbyeButton.addActionListener(al);

        setLayout(new FlowLayout(FlowLayout.LEFT));

        add(helloButton);
        add(goodbyeButton);
    }

    public void setTextPanel(TextPanel textPanel) {
        this.textPanel = textPanel;
    }
}
```

NOTA: recordar establecer o textPanel con **setTextPanel(...)** no MainFrame.

A solución anterior fai que os compoñentes estean fortemente enlazados e complica o mantemento de aplicacións cando estas empezan a medrar.

O ideal é crear aplicacións con compoñentes simples separados e que teñan a mínima interacción entre eles. A comunicación debe facerse a través da clase controladora do patrón MVC - Model View Controller, (no noso exemplo a clase controladora é o MainFrame).

Para desacoplar o TextPanel da barra de ferramentas farase que a clase **Toolbar** teña asociado un Listener personalizado coa función a executar cando se pulse un dos botóns. Para crear o Listener personalizado crearemos unha interface chamada StringListener que

estenda da clase EventListener, que é a clase da que estenden todos os listeners no modelo de Swing.

```
public interface StringListener extends EventListener {
    public void textEmitted(String text);
}
```

O método **textEmitted** será executado cada vez que se pulse o botón da barra de ferramentas.

```
public class Toolbar extends JPanel {
    ….
    private StringListener stringListener;
    ….
    public Toolbar() {
        ….
        ActionListener al = new ActionListener() {
          @Override
          public void actionPerformed(ActionEvent e) {
            JButton clickedButton = (JButton) e.getSource();
            if (clickedButton == helloButton) {
              if (stringListener != null) {
                stringListener.textEmitted("Hello\n");
              }
            } else if (clickedButton == goodbyeButton) {
              if (stringListener != null) {
                stringListener.textEmitted("Goodbye\n");
              }
            }
          }
        };
    }

    public void setStringListener(StringListener listener) {
        this.stringListener = listener;
    }
}
```

NOTA: recordar engadir o listener aos botóns.

Tan só falta proporcionar a implementación do StringListener, que se fará na clase MainFrame (clase controladora):

```
public class MainFrame extends JFrame {
    ….
    private Toolbar toolbar;
    private TextPanel textPanel;

    public MainFrame() {
        ….
        toolbar = new Toolbar();
```

```
        toolbar.setStringListener(new StringListener() {
            @Override
            public void textEmitted(String text) {
                textPanel.appendText(text);
            }
        });
    }
}
```

Exercicios:

1. Actualiza a aplicación usada en exemplos anteriores para que a barra de ferramentas se comunique co panel de texto a través do MainFrame, como se explica neste apartado.
2. Engade á aplicación un formulario lateral coma o da imaxe e deséñao usando un GridBagLayout. Faino nunha clase separada (FormPanel). Neste caso só hai que facer o deseño.

   Utilizando o método setPreferredSize, establece o ancho do formulario a 250.

   Comprobarás tamén que ao facer a ventá máis pequena (alto e ancho), poden producirse problemas co tamaño dos compoñentes. Para evitalo, establece un tamaño mínimo (setMinimunSize) ao MainFrame.

Eventos personalizados

O seguinte exemplo mostra o código clásico de manexadores de eventos. Observar que o método **actionPerformed** ten un parámetro de tipo **ActionEvent**. Este obxecto (ActionEvent) é o mecanismo que Swing utiliza para enviar información desde o elemento que lanza o evento (botón no exemplo) á clase manexadora do evento (ActionListener).

```
aceptarButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        textArea.append("Ola\n");
    }
});
```

Vaise modificar a interface StringListener creada no apartado anterior, para que o método **textEmitted** reciba un evento personalizado, que almacenará a información que se queira enviar dende a fonte do evento ao manexador de dito evento. Neste caso vaise enviar unha cadea de texto.

Crearase unha nova clase **StringEvent** que estende de EventObject, que é a clase da que estenden todos os Eventos no modelo de Swing. Ademais, o construtor de EventObject recibe como parámetro o obxecto fonte do evento.

```
public class StringEvent extends EventObject {
    private String text;

    public StringEvent(Object source) {
        super(source);
    }

    public StringEvent(Object source, String text) {
        super(source);

        this.text = text;
    }

     // Engadir getters e setters
}
```

Tamén hai que modificar a interface StringListener para que o método textEmitted reciba o evento personalizado como parámetro:

```
public interface StringListener extends EventListener {
    public void textEmitted(StringEvent e);
}
```

A continuación hai que modificar a clase MainFrame para actualizar a implementación do StringListener. Agora será necesario utilizar o obxecto evento (StringEvent) para acceder o texto do evento.

```
public class MainFrame extends JFrame {
    ….
    public MainFrame() {
        ….
        toolbar.setStringListener(new StringListener() {
            @Override
            public void textEmitted(StringEvent e) {
                textPanel.appendText(e.getText());
            }
        });
        …
    }
}
```

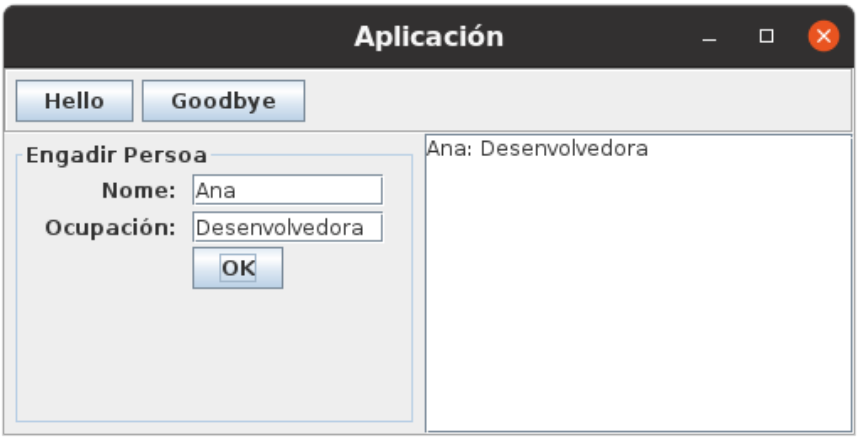
Por último, cando haxa que xerar o evento, haberá que crear un obxecto co evento personalizado (StringEvent) e envialo ao manexador de eventos (stringListener):

```
public class Toolbar extends JPanel {
    private StringListener stringListener;
    ….

    public Toolbar() {
        ….
        ActionListener al = new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                JButton clickedButton = (JButton) e.getSource();

                if (clickedButton == helloButton) {
                    if (stringListener != null) {
                        StringEvent se = new StringEvent(this, "...");
                        stringListener.textEmitted(se);
                    }
                } ….
            }
        };
        helloButton.addActionListener(al);
        …..
    }

    public void setStringListener(StringListener listener) {
        this.stringListener = listener;
    }
}
```

Con esta solución faise que a clase MainFrame funcione como controladora e toda a interacción entre o resto de clases pasa a través dela, facendo que o resto das clases permanezan o máis independentes posible.

Exercicio:

1. Modifica a aplicación de exemplos anteriores configurando o botón OK para que ao pulsalo se escriba na área de texto o nome e ocupación dos campos de texto do formulario. Debes crear un FormListener e un FormEvent personalizado para o formulario lateral.



JList

Os enxeñeiros de Swing crearon os compoñentes usando unha versión modificada do patrón **MVC**. En realidade os compoñentes están implementados de tal forma que a vista e o controlador son indivisibles e están implementados nun único obxecto. Ademais, cada compoñente ten un modelo para os datos, o que implica que para manipular os datos do compoñente é necesario acceder a unha instancia do modelo.

A vantaxe de usar un modelo de datos é que estes ofrecen flexibilidade á hora de determinar como se almacenan e recuperan os datos. Ademais, a implementación dos compoñentes garante que a modificación dos datos do modelo se vexa reflectida na vista automaticamente. É dicir, cando o modelo cambia, lánzanse os eventos necesarios para que a vista se actualice adecuadamente.

Un **JList** é un compoñente que permite presentar unha lista de elementos nunha ou máis columnas, e permite facer a selección dun ou de varios deses elementos. Dado que as listas teñen varios elementos, normalmente colócanse nun **scroll pane**.

O JList ten un **modelo de datos** onde se almacena a información sobre os elementos da lista. Hai tres formas de crear o modelo da lista:

- DefaultListModel: é o máis fácil, pois proporciona implementación para toda a funcionalidade da lista.
- AbstractListModel: hai que facer a implementación dunha subclase de AbstractListModel e implementar os métodos getSize e getElementAt.
- ListModel: hai que implementar todo a man.

Formas de agregar elementos a un JList:

- Usando un array:

```
JList listaNomes;
String nomes[] = { "Ana", "Susana", "Erea"};
listaNomes = new JList( nomes );
```

  Cando non se explicita o modelo, JList crea un automaticamente que non se pode modificar. É dicir, non se poden engadir nin eliminar elementos.

- Usando un modelo:

```
DefaultListModel modelo = new DefaultListModel();
modelo.addElement("Elemento1");
modelo.addElement("Elemento2");
modelo.addElement("Elemento3");

JList listaNomes = new JList();
listaNomes.setModel(modelo);
```

Toda aplicación debe programarse de forma que sexa robusta e flexible. No caso dos compoñentes JList, normalmente non conteñen texto estático, senón que mostran información recollida dunha base de datos. Para implementar esta funcionalidade sería mellor que os elementos do modelo do JList fosen obxectos que se correspondesen cos do modelo de datos. Exemplo:

```
class Dato {
  private int id;
  private String text;

  public Dato(int id, String text){
    this.id = id;
    this.text = text;
  }

  public int getId() {
    return id;
  }

  // Impleméntase para que no JList se mostre o texto
  public String toString() {
    return text;
  }
}
```

Unha vez definido o obxecto do modelo de datos, por exemplo, a clase Dato anterior, habería que modificar o JList:

```
JList listaNomes = new JList();
DefaultListModel modelo = new DefaultListModel();
modelo.addElement(new Dato(0,"Elemento1"));
modelo.addElement(new Dato(1,"Elemento2"));
modelo.addElement(new Dato(2,"Elemento3"));
listaNomes.setModel(modelo);
```

O JList ten métodos que permiten xestionar a lista como:

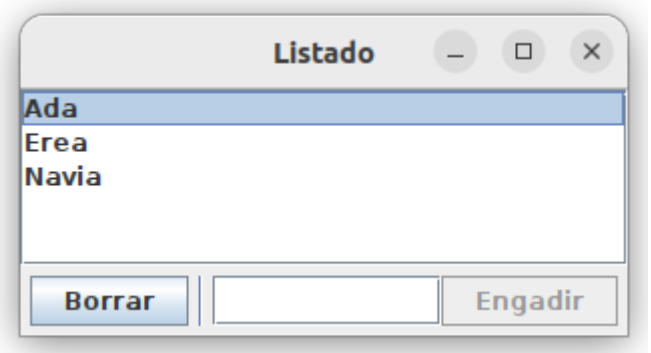
- setSelectionMode(int selectionMode): permite configurar cantos elementos se poden seleccionar na lista e se deben ou non estar contiguos.
- setLayoutOrientation(int layoutOrientation): permite configurar como debe ser a disposición dos elementos.
- setVisibleRowCount(int visibleRowCount): define cantas filas mostrar da lista.
- getSelectedIndex() e setSelectedIndex(int index): obter/establecer o índice da cela seleccionada.
- getSelectedValue(): devolve o valor da cela seleccionada

Cada vez que se cambia a selección de elementos da lista, esta lanza un evento que pode ser capturado con un ListSelectionListener, do que hai que implementar o método **valueChanged**:

```
list.addListSelectionListener(new ListSelectionListener() {
  @Override
  public void valueChanged(ListSelectionEvent lse) {
    ...
  }
});
```

Exercicios:

1. Crea unha aplicación que mostre un listado de cadeas utilizando un JList e permita engadir e eliminar elementos con unha interface similar á da seguinte imaxe:



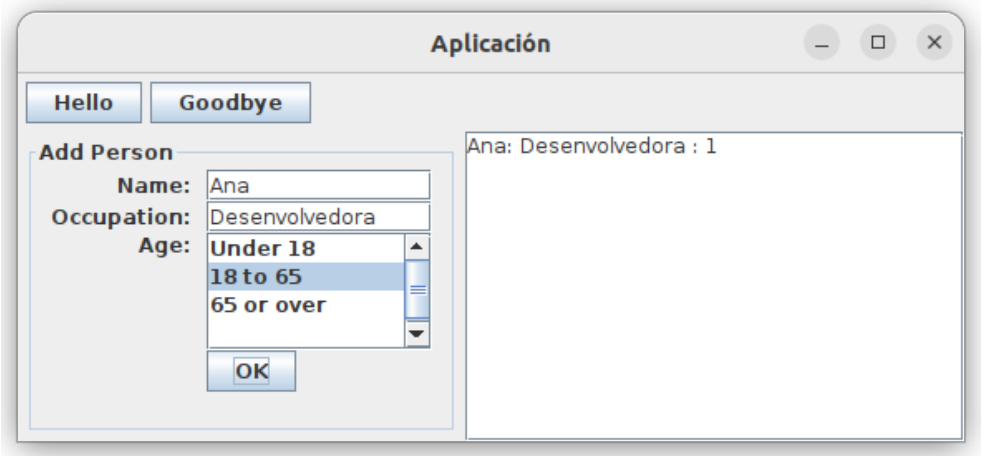
   a. Só se poderá seleccionar un elemento da lista ao mesmo tempo.
   b. Inicialmente a aplicación terá 3 cadeas e aparecerá seleccionada a primeira.
   c. O botón borrar permitirá eliminar o elemento seleccionado.

d. Unha vez eliminado un elemento, o novo elemento seleccionado será o que ocupaba a seguinte posición na lista. Se se elimina o último elemento, seleccionarase o que pase a ocupar a última posición.
e. Cando se elimine o último elemento, o botón Borrar deshabilitarase.
f. As caixas de texto tamén teñen un modelo de datos, denominado *document*, que permite xestionar o seu contido. Engade á caixa de texto un DocumentListener que permita activar o botón "Engadir" cando a caixa de texto contén información e desactivala en caso contrario.

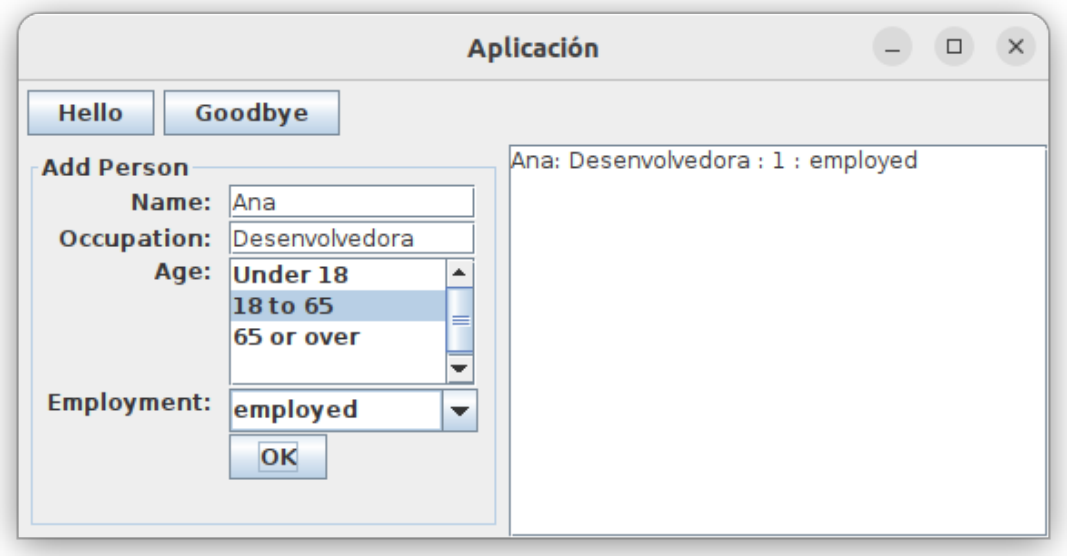
```
textField.getDocument().addDocumentListener(new DocumentListener…);
```

g. O botón engadir permitirá engadir un elemento á lista co texto da caixa de texto. O novo elemento engadido colocarase despois do elemento seleccionado da lista. Implementa tamén esta funcionalidade cando se pulse ENTER na caixa de texto

2. Modifica a aplicación creada en exercicios anteriores engadindo un JList coma o da imaxe. Ademais:
   a. Crea un obxecto AgeCategory para almacenar os elementos do modelo do JList. Este obxecto debe ter dúas propiedades: **id** de tipo int e **text** de tipo String.
   NOTA: crea a clase AgeCategory dentro do ficheiro FormPanel.java.
   b. Engade á aplicación unha lista coma a da imaxe e fai que por defecto estea seleccionada a categoría "18 to 65".
   c. Configura a lista para que só se poida seleccionar un elemento.
   d. Establece o tamaño do JList para que sexa similar ao da imaxe. Utiliza o método setPreferredSize().
   e. Engade a etiqueta "Age" e colócaa coma na imaxe.
   f. Ao pulsar o botón "OK", na área de texto debe mostrarse tamén información da categoría de idade seleccionada.



3. O compoñente JComboBox funciona de forma moi similar ao JList. Engade á aplicación anterior un ComboBox coma o da imaxe. Neste caso non é necesario crear un obxecto para os datos do modelo, poden usarse directamente Strings (employed, self-employed e unemployed). Configura ademais o JComboBox para que sexa **editable** e fai que estea seleccionado por defecto o texto "**employed**".

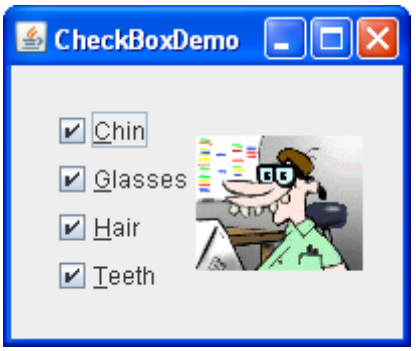


JCheckBox, JRadioButton

Tanto o JCheckBox como JRadioButton son subclases de AbstractButton, polo que se comportan como botóns e teñen as mesmas características.

Un **JCheckBox** permite implementar un cadro de selección (basicamente un botón con dous estados).

Os JCheckBox permite seleccionar varios elementos nun grupo, a diferencia dos JRadioButton, onde só un pode estar seleccionado do grupo.



Un JCheckBox xera un **ItemEvent** e un **ActionEvent** por clic do rato. Normalmente utilízase o ItemEvent (controlado mediante un ItemListener) porque permite saber se o checkbox está ou non seleccionado.

```
checkBox.addItemListener(new ItemListener() {
    @Override
    public void itemStateChanged(ItemEvent e) {
        if (e.getStateChange() == ItemEvent.SELECTED) {
            System.out.println("CheckBox seleccionado.");

        } else if (e.getStateChange() == ItemEvent.DESELECTED) {
            System.out.println("CheckBox deseleccionado.");
        }
    }
});
```

Os **botóns de radio** son grupos de botóns onde, por convenio, só un botón do grupo pode ser seleccionado ao mesmo tempo. Para implementar os botóns de radio usaranse as clases JRadioButton e ButtonGroup.

Para que actúen en conxunto, os botóns de radio deben agruparse, é dicir, haberá que incluír todos o JRadioButton no mesmo ButtonGroup. O grupo xa se encarga de deseleccionar o botón anterior cando se selecciona un novo. Normalmente créase un grupo de botóns e selecciónase un inicial de forma manual. A API non forza a que isto se cumpra, é dicir, inicialmente pode non haber ningún botón seleccionado. No momento que se selecciona un, xa sempre haberá un botón seleccionado.

Cada vez que se pulsa un botón de radio (aínda que xa estivese seleccionado) lánzase un **ActionEvent**. Ao mesmo tempo, tamén se lanzan **ItemEvents**, un para seleccionar o novo elemento e outro para deseleccionar o anterior. Normalmente para xestionar os botóns de radio utilízase un **ActionListener**.

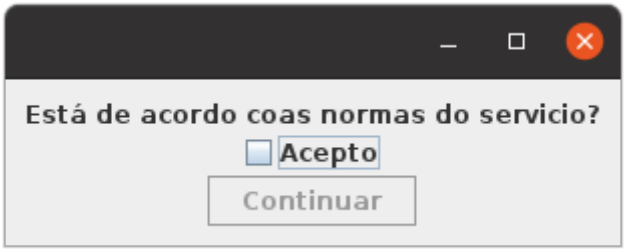
```
JRadioButton birdButton = new JRadioButton(birdString);
birdButton.setActionCommand("Bird");

JRadioButton catButton = new JRadioButton(catString);
catButton.setActionCommand("Cat");

//Group the radio buttons.
ButtonGroup group = new ButtonGroup();
group.add(birdButton);
group.add(catButton);

//Register a listener for the radio buttons.
birdButton.addActionListener(...);
catButton.addActionListener(...);
```

```
public void actionPerformed(ActionEvent e) {
    String command = e.getActionCommand();
    …
}
```

Exercicios:

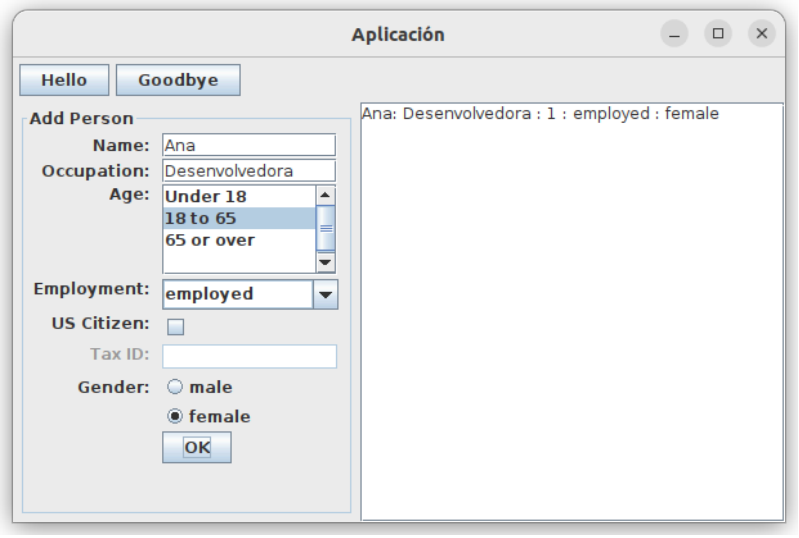
1. Crea unha aplicación que conteña 3 checkboxes polo menos (escolle ti o texto de cada un). Engade unha etiqueta para informar dos elementos seleccionados. É dicir, a etiqueta debe mostrar o texto “Elementos seleccionados: xxx”, onde xxx sexa a lista dos checkboxes seleccionados.

2. Crea unha aplicación similar á da imaxe. Cando o CheckBox se active, o botón debe habilitarse. Fíxate que os compoñentes están centrados na ventá.



3. Minicalculadora. Crea unha aplicación que teña dúas caixas de texto para os números, 4 botóns de radio para simular as operacións de suma, resta, multiplicación e división, e unha caixa de texto para mostrar o resultado. Cada vez que se pulse un botón de radio, realizarase a operación seleccionada cos números escritos na caixa de texto.

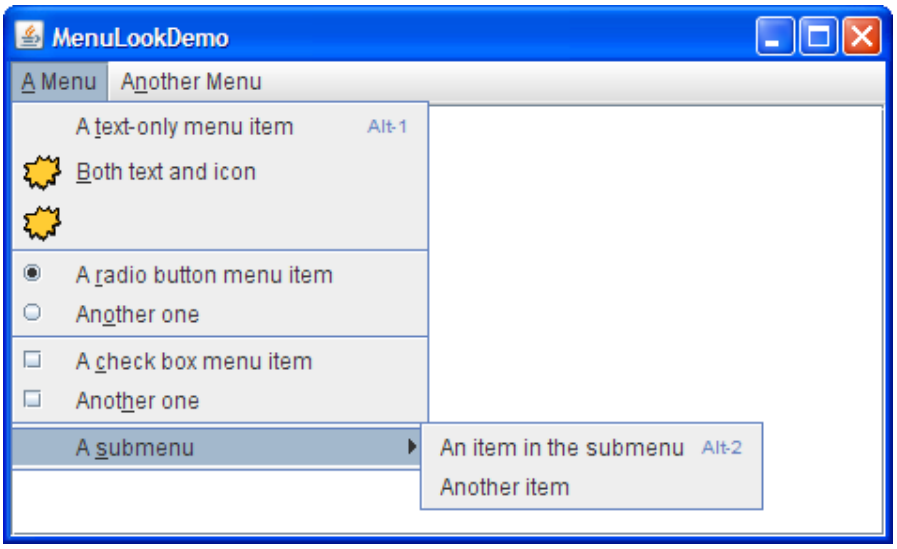
4. Modifica a aplicación creada en exercicios anteriores engadindo un JCheckBox, as etiquetas necesarias, unha caixa de texto e os RadioButtons, como se mostra na imaxe. Ademais:
    a. Cando o checkbox non está seleccionado, a etiqueta “Tax ID” e a caixa de texto deben estar deshabilitadas e viceversa.
    b. Ao pulsar o botón “OK”, na área de texto debe mostrarse tamén información do xénero seleccionado.



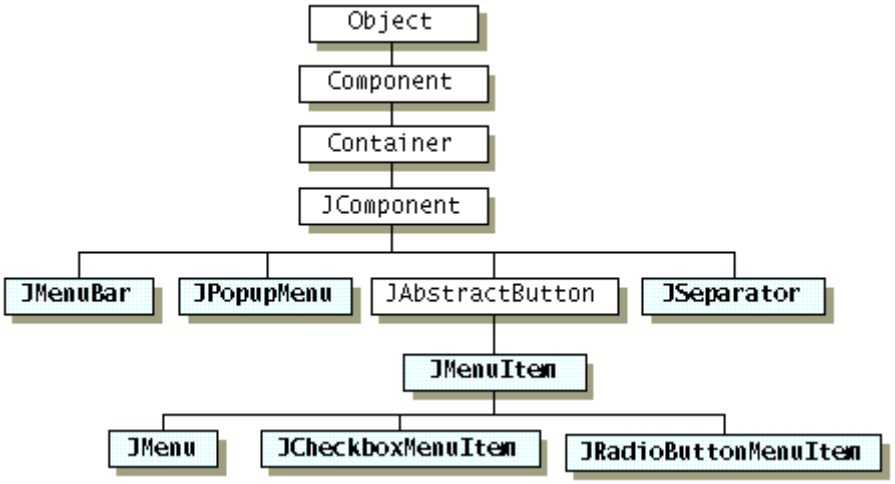
Menús

Os **menús** serven para que as persoas usuarias poidan escoller unha acción a realizar de entre varias. Non se colocan co resto dos compoñentes na interface, senón que aparecen nunha barra de menús ou como un menú emerxente.

A seguinte imaxe amosa os diferentes compoñentes que poden formar parte dun menú: a barra de menús, os items do menú, botóns de radio, checkBox e separadores. Ademais, un elemento do menú pode ter texto, unha imaxe, ou ambos. How to Use Menus.



A continuación móstrase a xerarquía de clases relacionadas co menú. Como pode observarse, tanto JMenu como JMenuItem, JCheckboxMenuItem e JRadioButtonMenuItem son simples botóns. Dado que os elementos do menú son botóns, poden asociárselles os mesmos listeners que aos botóns.



¿Como fai un menú para mostrar os seus submenús?. A resposta é que cando un menú se activa, mostra automaticamente unha ventá cos seus elementos.

Para incluír un menú nunha aplicación Java Swing, hai que engadir un elemento **JMenuBar** utilizando o método **setJMenuBar**. O JMenuBar é como un contedor que conterá os JMenu e os JMenuItem, que serán as accións que pulsen as persoas usuarias.

O seguinte código é un extracto de MenuLookDemo.java. Ligazón ao resto de ficheiros necesarios para o programa.

```
public class MainFrame extends JFrame {

  public MainFrame() {
    ….
    // engadir o menú ao frame
    setJMenuBar(createMenuBar());
    ….
  }

  private JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();
    JMenu menu = new JMenu("A Menu");
    JMenuItem menuItem = new JMenuItem(“A text-only menu item");

    menu.add(menuItem); // engade un JMenuItem a un JMenu
    menuBar.add(menu);  // engade o JMenu ao JMenuBar
    …
    menu.addSeparator();
    …
    return menuBar;
  }
}
```

NOTA: observar que para engadir un menú ao JFrame se usou o método setJMenuBar de JFrame.

Tanto nas barras de ferramentas como nos menús poden engadirse **separadores** - **menu.addSeparator() -** que permiten mellorar a visualización e presentación do menú dando significado aos diferentes grupos de elementos e mellorando o aspecto gráfico da aplicación.

Para detectar cando se selecciona un **JMenuItem**, poden escoitarse ActionEvents (igual que se faría para un botón):

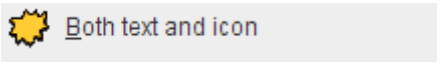
```
menuItem.addActionListener(...);
```

Para detectar cando se selecciona un **JCheckBoxMenuItem** normalmente escóitanse ItemEvents, igual que se facía cos JCheckBoxes. Para detectar cando se selecciona un **JRadioButtonMenuItem**, poden escoitarse ActionEvents ou ItemEvents, igual que se facía cos JRadioButtons.

Operar dende teclado

Os menús soportan dúas formas alternativas de interaccionar con eles a través do teclado: mnemonics e accelerators.

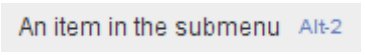
- **Mnemonics**: ofrecen unha forma de usar o teclado para navegar pola xerarquía do menú, incrementando a accesibilidade dos programas. Un mnemonic é unha **tecla** que combinada co modificador do look and feel (habitualmente Alt) activará un menú **visible**. Normalmente o mnemonic amósase subliñando o **carácter** no texto do menú.



  Para establecer o mnemonic hai que especificar a constante de **KeyEvent** correspondente á tecla que a persoa usuaria debe pulsar xunto con Alt. Exemplo:

```
jMenuItem1.setMnemonic(KeyEvent.VK_M);
```

- **Accelerators**: ofrecen atallos de teclado para activar de forma rápida a unha acción do menú, aínda que non estea visible.



  NOTA: só os menús folla poden ter accelerators.

  Para especificar o *accelerator* hai que usar un obxecto **KeyStroke** que combina unha tecla (especificada pola constante **KeyEvent)** e unha máscara modificadora (especificada por unha constante **ActionEvent**). Exemplo:

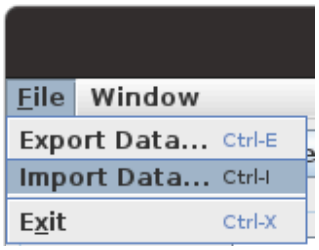
```
jMenuItem1.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_M,
ActionEvent.CTRL_MASK));
```

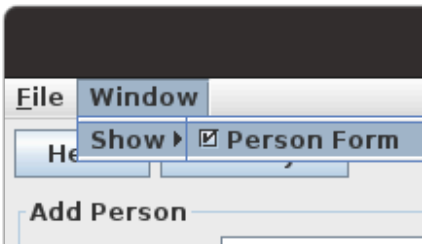
  O código anterior establece o accelerator coa combinación de teclas CRTL + M.

NOTA: *Mnemonics are for all users; accelerators are for power users.*

Exercicios:

1. Modifica a aplicación creada en exercicios anteriores engadindo un menú como se mostra nas imaxe. Ademais:
    a. Configura os mnemonics e accelerators como se observan nas imaxes.
    b. Configura as accións:
        - O menú “Exit” debe facer que a aplicación se peche.
        - O checkBox “Person Form” debe mostrar ou ocultar o formulario lateral “Add Person”.





Icona menús

Un proxecto Maven debe ter unha estrutura de carpetas e arquivos como establece o estándar.

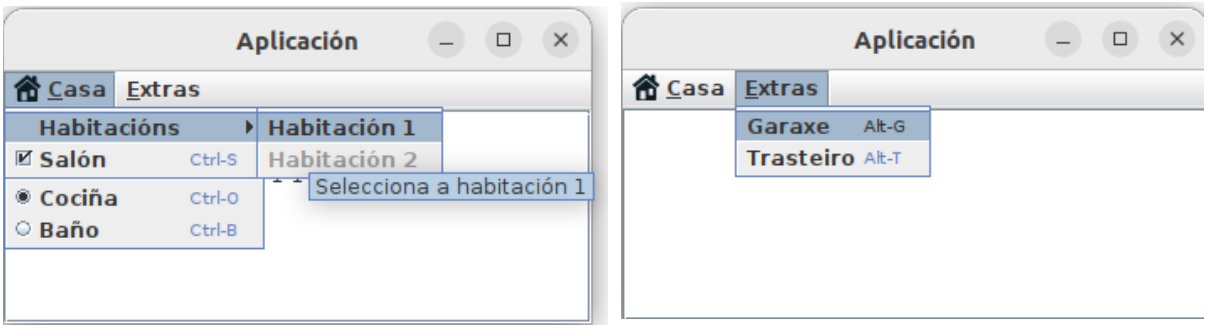
Cando se traballa con iconas do menú, o habitual é que se coloquen na carpeta **src/main/resources**. Todos os ficheiros e carpetas colocados en **resources** son copiados automaticamente á carpeta de compilación e permiten ser localizados mediante código a través do directorio raíz de compilación. Así por exemplo, cando se compile o proxecto, o ficheiro situado en **src/main/resources/icon.png** será copiado automaticamente ao directorio **target/classes/icon.png**.

Unha vez colocadas as imaxes na carpeta resources, para incluílas como unha icona nun menú, simplemente hai que indicalo co seguinte código:

```
public MainFrame() {
  ….
  jMenuCasa.setIcon(createIcon("/house.png"));
  …
}

private ImageIcon createIcon(String path) {
  URL url = getClass().getResource(path);

  if (url == null) {
    System.err.println("Unable to load image: " + path);
  }

  ImageIcon icon = new ImageIcon(url);

  return icon;
}
```

Exercicio:

Crea unha aplicación Java Swing que conteña o menú coma os das seguintes imaxes e programar os mnemonics e aceleradors das opcións do menú:

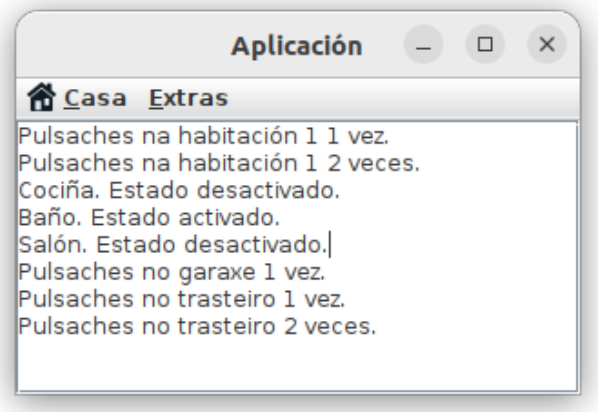


Descrición dos menús:

- Casa terá unha icona dunha casa.
  - Habitacións (JMenu)
    - Habitación 1 (JMenuItem). Engadir TooltipText.
    - Habitación 2 (JMenuItem): desactivado
  - Salón (JCheckBoxMenuItem): seleccionado. Ctrl+S.
  - Separador (JSeparator)
  - Cociña (JRadioButtonMenuItem): seleccionado. Ctrl+O
  - Baño (JRadioButtonMenuItem): Ctrl+B
- Extras:
  - Garaxe (JMenuItem): Alt+G
  - Trasteiro (JMenuItem): Alt+T

Ademais, o JFrame inclúe un **área de texto** na que se incluirán, a modo de log ou rexistro, as opcións seleccionadas pola persoa usuaria escribindo unha mensaxe do tipo:

"Pulsaches na habitación 1 3 veces." ou “Cociña. Estado desactivado”. O estado fai referencia a se o RadioButton ou Checkbox queda seleccionado ou non. O número de veces é un contador que se vai incrementando a medida que se pulsan as opcións, como se aprecia na seguinte imaxe:



Popup menu

Un popup é un menú que aparece de forma dinámica nunha posición específica dun compoñente.

Para abrir un menú popup hai que rexistrar un Listener en cada compoñente ao que se queira asociar o popup. O listener debe detectar as peticións que requiran mostrar o popup, que dependerán da contorna (non en todas as contornas se utiliza a mesma acción para mostrar un menú de popup).

Exemplo de utilización dun Popup menu:

```
JPopupMenu popup = new JPopupMenu();
…
componente.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent e) {
        maybeShowPopup(e);
    }

    public void mouseReleased(MouseEvent e) {
        maybeShowPopup(e);
    }

    private void maybeShowPopup(MouseEvent e) {
        if (e.isPopupTrigger()) {
            popup.show(e.getComponent(), e.getX(), e.getY());
        }
    }
});
```

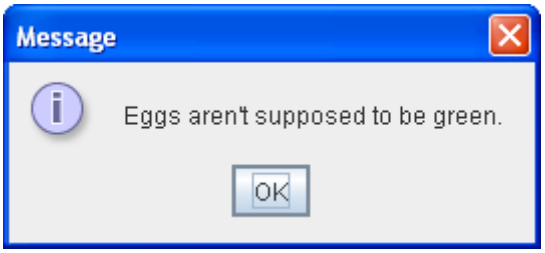
Diálogos

Un diálogo é unha ventá independente para mostrar información de forma temporal e separada da ventá principal da aplicación. Normalmente utilízanse para mostrar unha mensaxe de erro ou aviso, aínda que poden crearse diálogos complexos.

Nunha aplicación Java Swing é bastante habitual ter que pedir confirmación dunha acción, un dato sinxelo ou mostrar un aviso. Para realizar estas accións comúns existe a clase **JOptionPane** que mostra unha ventá sinxela sen ter que programar case nada.

A continuación móstrase o código para crear un diálogo con información e unha imaxe do mesmo:

```
JOptionPane.showMessageDialog(frame, "Eggs are not supposed to be green.");
```



Os diálogos son dependentes dun compoñente Frame. Cando o Frame se destrúe, tamén o fan os diálogos dependentes. Cando o frame se maximiza/minimiza, tamén o fan os diálogos que del dependen.

Un diálogo pode configurarse para ser modal ou non. Un diálogo modal bloquea toda interacción con outras ventás da aplicación.

Un JOptionPane permite crear de forma fácil unha subventá independente **modal** separada da ventá principal da aplicación. Soe usarse para mostrar mensaxes de erro, avisos ou pedir ás persoas usuarias introducir datos. Se non é suficiente usar un JOptionPane, terase que programar un JDialog personalizado.

Con JOptionPane poden crearse diferentes diálogos e personalizalos especificando o título, texto da ventá e o texto do botón. Tamén é configurable a icona que mostrará o diálogo: estándar (pregunta, información, warning, erro), personalizada ou ningunha. Ver exemplos.

Icons used by JOptionPane

Icon description	Java look and feel	Windows look and feel
question		
information		
warning		
error		

Aínda que pode parecer complexa de usar, case todos os usos desta clase consisten nunha chamada a un dos métodos estáticos **showXXXDialog:**

- **showMessageDialog**: mostra un diálogo simple con un botón.
- **showOptionDialog**: mostra un diálogo personalizable, con botóns e texto personalizados.
- **showConfirmDialog**: pide confirmación para realizar algunha acción. Ten os botóns estándar Si/Non.
- **showInputDialog**: mostra un campo de texto, combo box ou lista para pedir información á persoa usuaria.

Algúns dos parámetros destes métodos son:

- **parentComponent**: fai referencia ao frame ancestro.

  NOTA: cando se usan clases aniñadas pode ser necesario facer referencia á clase pai coa nomenclatura NomeClase.this. Consultar as seguintes referencias en Internet:

  - What is Class.this in Java?
  - What is the difference between Class.this and this in Java - Stack Overflow

- **message**: especifica unha mensaxe descritiva que se colocará na ventá.
- **title**: título da ventá de diálogo.
- **optionType**: define o conxunto de botóns de opción a mostrar no fondo do diálogo:
  - JOptionPane.DEFAULT_OPTION
  - JOptionPane.YES_NO_OPTION
  - JOptionPane.YES_NO_CANCEL_OPTION
  - JOptionPane.OK_CANCEL_OPTION
- **messageType**: tipo de mensaxe a mostrar, que determinará a icona do diálogo. Os posibles valores son:
  - JOptionPane.ERROR_MESSAGE
  - JOptionPane.INFORMATION_MESSAGE
  - JOptionPane.WARNING_MESSAGE
  - JOptionPane.QUESTION_MESSAGE
  - JOptionPane.PLAIN_MESSAGE.
- **icon**: icona para colocar no diálogo. Se é null, sairá a icona por defecto para o diálogo.
- **Object[] options**: para especificar a cadea a mostrar en cada botón.
- **initialValue**: valor seleccionado por defecto. Debe ser un dos obxectos pasado no parámetro **options**. Pode ser null.

Os métodos showMessageDialog, showConfirmDialog e showOptionDialog **devolven** un enteiro indicando a opción escollida pola persoa usuaria. Os posibles valores son YES_OPTION, NO_OPTION, CANCEL_OPTION, OK_OPTION, and CLOSED_OPTION. Excepto CLOSED_OPTION, cada opción corresponde ao botón pulsado pola persoa usuaria. Cando se devolve CLOSED_OPTION, significa que se cerrou o diálogo explicitamente en lugar de escoller un botón.

Exemplo:

```
int confirmado = JOptionPane.showConfirmDialog(componhentePai, "¿Queres saír?");

if (JOptionPane.OK_OPTION == confirmado)
   System.out.println("confirmado");
else
   System.out.println("non confirmado");
```

Máis información.

Exercicios:

1. Crea unha aplicación con 4 botóns. Cada botón debe usar un dos métodos de **JOptionPane:**
   - **showMessageDialog:** con unha mensaxe e que teña un título personalizado. Proba a utilizar as diferentes iconas: información, aviso, erro, ningunha e icona personalizada.
   - **showConfirmDialog**: que pregunte á persoa usuaria confirmación para realizar unha acción. As posibles respostas que debe mostrar o diálogo son: si, non e cancelar. Imprime por pantalla información do botón pulsado. ¿Que pasará se cerra o diálogo?

- **showInputDialog**: que pida datos á persoa usuaria usando unha lista despregable ou unha caixa de texto. Imprime por pantalla o valor seleccionado. ¿Que pasará se cerra o diálogo?
- **showOptionDialog**: con un diálogo personalizado onde a mensaxe e os botóns teñan texto personalizado. Configura tamén o diálogo para que o botón seleccionado por defecto sexa o segundo. Imprime por pantalla o valor da opción seleccionada.

2. Modifica a aplicación realizada en exercicios anteriores para que pida confirmación antes de saír da aplicación dende o menú “Saír”.

JFileChooser

JFileChooser proporciona un mecanismo simple para navegar polo sistema de ficheiros e permite seleccionar un ficheiro ou directorio de forma gráfica. É dicir, permite, de forma fácil, mostrar os diálogos para abrir e gardar ficheiros. Pode consultarse a documentación de como usar o compoñente JFileChooser.

Para mostrar un JFileChooser estándar só son necesarias as seguintes liñas:

```
JFileChooser fc = new JFileChooser();

// Abrir diálogo para abrir ficheiro
int returnVal = fc.showOpenDialog(parent);
if (returnVal == JFileChooser.APPROVE_OPTION) {
    System.out.println(fc.getSelectedFile());
} else {
    System.out.println("Open command cancelled by user.");
}

// Abrir diálogo para gardar ficheiro
int returnVal = fc.showSaveDialog(parent);
if (returnVal == JFileChooser.APPROVE_OPTION) {
    System.out.println(fc.getSelectedFile());
} else {
    System.out.println("Open command cancelled by user.");
}
```

O parámetro pasado ao método *showOpenDialog* especifica o compoñente pai do diálogo. Este compoñente pai afecta á posición que o diálogo ocupará e tamén especifica o frame ancestro do que depende.

Un JFileChooser que non foi usado previamente **mostrará por defecto a lista de ficheiros do directorio HOME**. Ademais, o JFileChooser recordará o directorio actual para a próxima vez que se volva a abrir.

O método showXxxDialog **devolve un enteiro** que indica se o usuario seleccionou ou non un ficheiro. Normalmente é suficiente comprobar se o valor devolto é JFileChooser.APPROVE_OPTION. Para obter o ficheiro seleccionado usarase o método **getSelectedFile()**.

Por defecto o fileChooser mostra todos os ficheiros e directorios, excepto os ocultos. É posible crear un filtro personalizado para que só mostre algúns ficheiros. O fileChooser invoca o método ***accept*** do filtro para cada ficheiro para determinar se debe ser mostrado ou non.

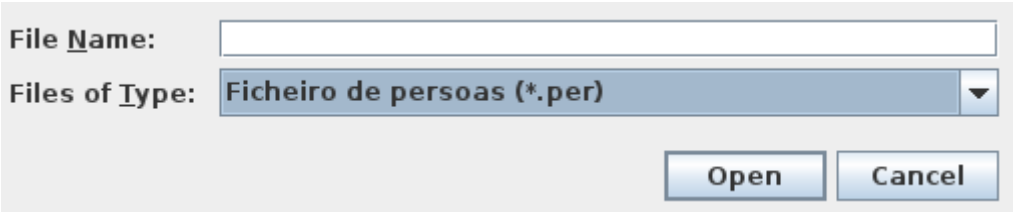
O seguinte exemplo mostra como crear un **filtro personalizado** para permitir só a selección de ficheiros con extensión *.per. Cando a persoa usuaria selecciona este filtro no JFileChooser, só se mostrarán os ficheiros con extensión *.per.

```
public class PersonFileFilter extends FileFilter {

  @Override
  public boolean accept(File file) {
    // Aceptar os directorios para poder navegar pola árbore de directorios.
    if (file.isDirectory()) {
      return true;
    }
    // getExtension da clase Utils.java.
    String extension = Utils.getExtension(file);
    if (extension == null) {
      return false;
    }
    if (extension.equals("per")) {
      return true;
    }
    return false;
  }

  @Override
  public String getDescription() {
    // Descrición que aparecerá na lista de filtros
    return "Ficheiro de persoas (*.per)";
  }
}
```

O JFileChooser proporciona unha lista de filtros para que as persoas usuarias seleccionen cal usar. Para asignar o filtro personalizado ao JFfileChooser debe utilizarse a seguinte instrución:

```
fileChooser.addChoosableFileFilter(new PersonFileFilter());
```

Fixarse que agora o fileChooser permite seleccionar o filtro personalizado para que se mostren só os ficheiros con extensión *.per.



Exercicios:

1. Engade á aplicación realizada en exercicios anteriores un JFileChooser como o explicado neste apartado. Configura as opcións de menú “Export Data…” e “Import Data…” para que se abra un diálogo para gardar ou abrir ficheiros con extensión *.per.

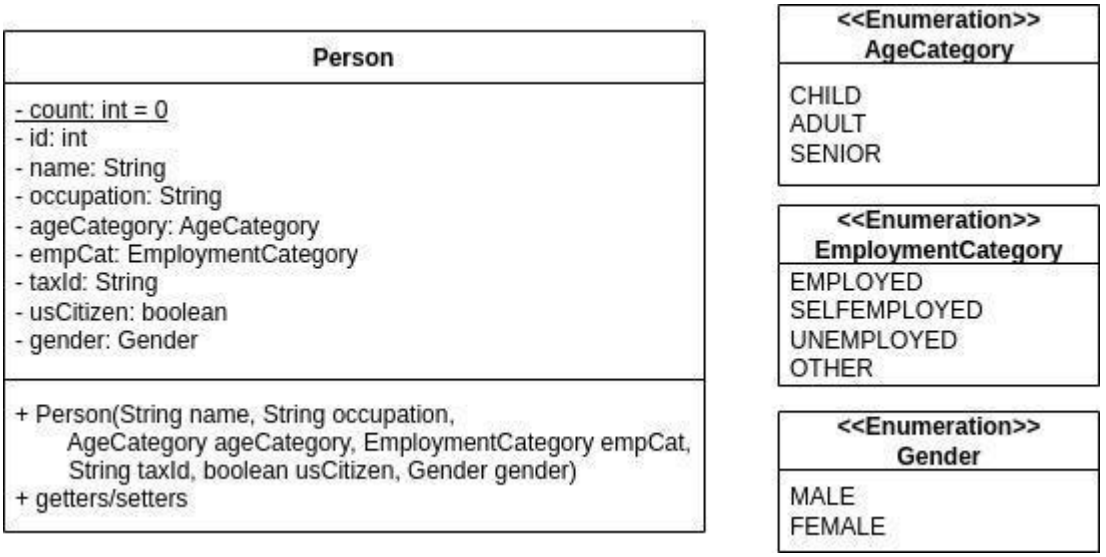
MVC - Modelo de datos e controlador

A aplicación realizada en exemplos anteriores só conta con parte gráfica. Agora vaise crear un modelo de datos sinxelo para almacenar a información introducida na interface relativa ás persoas.

Primeiramente vanse **crear** os paquetes que aparecen na seguinte imaxe e **mover** todas as clases existentes ata agora ao paquete **gui**:

Esta aplicación non afonda na implementación do patrón MVC. Neste caso vaise facer unha implementación tal que **a interface gráfica só se vai comunicar co modelo de datos a través da clase controladora**. Así, cando o MainFrame queira almacenar datos en ficheiro ou nunha base de datos, debe invocar algún método na clase controladora.

O paquete **model** vai conter todas as clases que teñan que ver coa base de datos: Person, AgeCategory, EmploymentCategory e Gender.



NOTA: un campo subliñado nun diagrama UML significa que é estático.

O campo **count** de Person serve para asignar identificadores aos obxectos Person creados, por iso é un campo estático. Cada vez que se cree un novo obxecto Person (é dicir, no

construtor), asignaráselle como id o valor almacenado en count e incrementarase o valor de count, para garantir que o seguinte obxecto creado teña un identificador diferente.

AgeCategory, EmploymentCategory e Gender están definidos como tipos de datos enum. Enum é un tipo especial de datos que permite que unha variable teña un conxunto de valores predefinidos. A variable debe ser igual a un dos valores predefinidos para ela. Exemplo:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

Explicación interesante de Enum en Java.

A base de datos vaise simular creando unha clase **Database** no paquete **model**. A base de datos almacenará unha lista de persoas e proporcionará métodos para engadir unha persoa á lista e tamén para recuperar a lista de persoas.

```
public class Database {

  private List<Person> people;

  public Database() {
    people = new LinkedList<Person>();
  }

  public void addPerson(Person person) {
    people.add(person);
  }

  public List<Person> getPeople() {
    // Para evitar que se modifique a lista devolta
    return Collections.unmodifiableList(people);
  }
}
```

NOTA: úsase unha LinkedList porque proporciona unha implementación eficiente para eliminar elementos do medio da lista.

A lóxica da aplicación será implementada con unha clase **Controller**, dentro do paquete controller. Esta clase actuará de intermediaria entre a vista e o modelo de datos.

```
public class Controller {
  // Garda unha referencia á base de datos
  private Database db = new Database();

  public List<Person> getPeople() {
    // Falta implementar este método
  }

  public void addPerson(FormEvent ev) {
    // Recuperar toda a información do obxecto ev
    String name = ev.getName();
    …
    int ageCatId = ev.getAgeCategory();
    String empCat = ev.getEmploymentCategory();

    AgeCategory ageCategory = null;
    switch (ageCatId) {
      case 0:
        ageCategory = AgeCategory.CHILD;
        break;
      …
    }

    EmploymentCategory empCategory;
    if (empCat.equals("employed")) {
      empCategory = EmploymentCategory.EMPLOYED;
    } else if … {
      ….
    }

    Person person = new Person(name, occupation, ageCategory, empCategory,
          taxId, isUs, genderCat);

    db.addPerson(person);
  }
}
```

Tamén hai que modificar a clase MainFrame para ter unha referencia ao **Controller**:

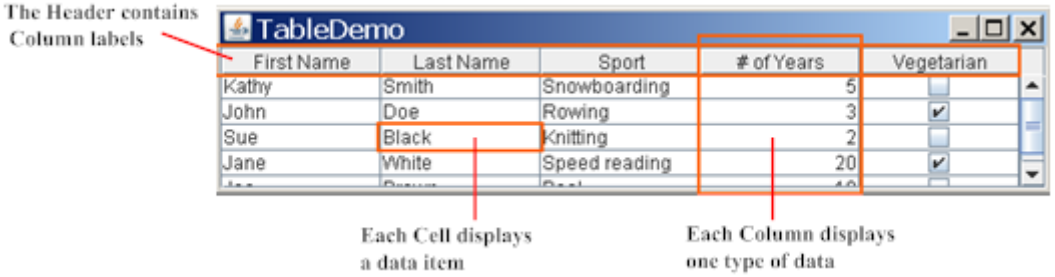
```
private Controller controller;

public MainFrame() {
  …
  controller = new Controller();
  …
  formPanel.setFormListener(new FormListener() {
      public void formEventOccurred(FormEvent ev) {
        controller.addPerson(ev);
      }
  });
  …
}
```

Exercicios:

1. Modifica a aplicación realizada en exercicios anteriores como se indica neste apartado, organizando as clases da aplicación nos paquetes controller, gui e model.

JTable

Un JTable permite mostrar datos en forma de táboa, permitindo opcionalmente editar a información mostrada.



O exemplo SimpleTableDemo.java demostra como crear unha táboa simple:

```
String[] columnNames = {"First Name", "Last Name", "Sport", "# of Years",
  "Vegetarian"};

Object[][] data = {
  {"Kathy", "Smith", "Snowboarding", new Integer(5), new Boolean(false)},
  {"John", "Doe", "Rowing", new Integer(3), new Boolean(true)},
  {"Sue", "Black", "Knitting", new Integer(2), new Boolean(false)},
  {"Jane", "White", "Speed reading", new Integer(20), new Boolean(true)},
  {"Joe", "Brown", "Pool", new Integer(10), new Boolean(false)}
};

final JTable table = new JTable(data, columnNames);
```

Hai dous construtores de JTable que aceptan datos directamente:

- JTable(Object[][] rowData, Object[] columnNames)
- JTable(Vector rowData, Vector columnNames)

A vantaxe dos **construtores** anteriores é que son fáciles de usar, sen embargo teñen desvantaxes:

- Fan que as celas sexa editables.
- Tratan todos os tipos de datos das celas de igual forma (como Strings). Por exemplo, se unha columna tivese tipo de dato booleano, gustaríanos que se mostrase como un checkbox. Sen embargo, ao usar os construtores anteriores, mostrarase como un String.
- Requiren que os datos estean en un array ou un vector, o que pode non ser apropiado para algún tipo de datos. Por exemplo, cando se mostran datos de obxectos, sería interesante utilizar os seus getters directamente.

Normalmente as táboas colócanse nun **JScrollPane**:

```
JScrollPane scrollPane = new JScrollPane(table);
table.setFillsViewportHeight(true);  // a táboa usa o alto do contedor.
```

O JScrollPane coloca automaticamente a cabeceira da táboa na cima do viewport e os nomes das columnas permanecen sempre visibles na cima.

Por defecto, todas as **columnas** da táboa teñen o mesmo tamaño e ocupan todo o ancho da táboa. Cando a táboa se redimensiona, o ancho das columnas adáptase. Pode personalizarse o ancho inicial dunha columna utilizando o método column.setPreferredWidth. Exemplo:

```
TableColumn column = table.getColumnModel().getColumn(2);
column.setPreferredWidth(100);
```

A configuración por defecto dunha táboa permite **seleccionar múltiples liñas**. Este comportamento pode modificarse usando o método JTable.setSelectionMode(int).

Para obter a fila seleccionada pode usarse o método JTable.getSelectedRow.

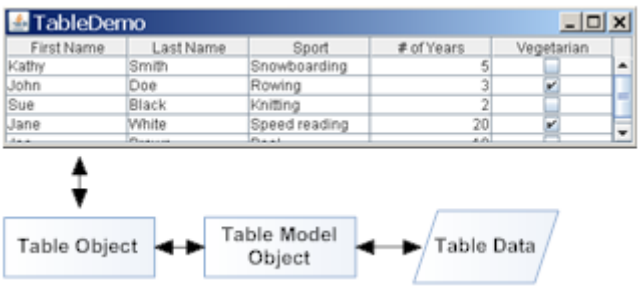
NOTA: as filas seleccionadas describen as filas seleccionadas na "vista" en lugar das filas seleccionadas no modelo. Se os datos visibles da táboa son o resultado de facer unha ordenación ou dun filtrado, haberá que convertir as coordenadas da vista ás súas correspondentes do modelo.

Creación dun modelo para unha táboa

Neste apartado continuarase ampliando a aplicación creada en apartados anteriores e substituirase a área de texto por un **JTable** para visualizar a información das persoas.

En Swing, os compoñentes que levan listados de datos (ComboBox, JList, JTable, …) teñen un **modelo de datos** asociado para xestionar os datos que se visualizan no compoñente.

As táboas (JTable) usan un modelo de datos para xestionar e manexar os datos. Se non se proporciona explicitamente un modelo, Java crea un de forma automática da clase **DefaultTableModel**.

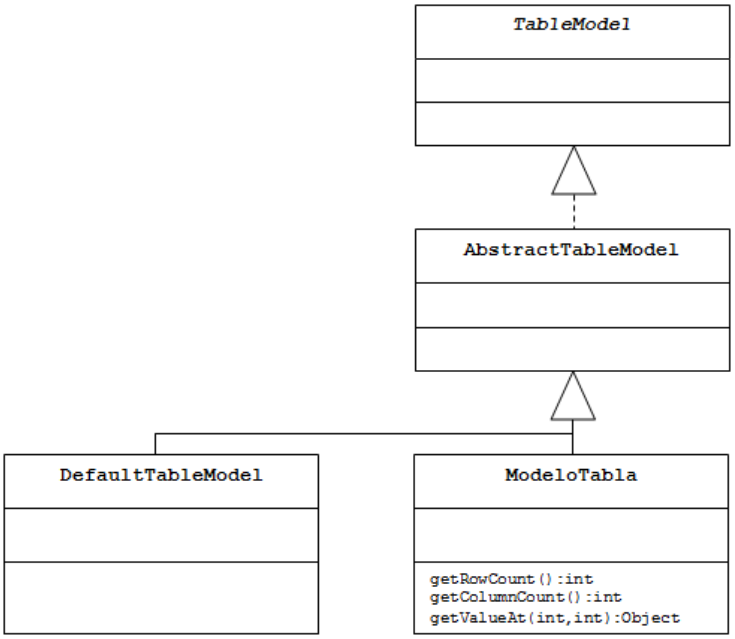


Swing ofrece unha implementación por defecto da interface **TableModel** que é a clase **DefaultTableModel.** Este modelo traballa internamente con un **array bidimensional** ou

matriz, onde cada elemento é o valor dunha das celas da táboa. Ademais, proporciona unha API propia para traballar con esta estrutura de datos. O modelo de datos por defecto trata todas as celas por igual, como se fosen Strings.

A aplicación coa que se está a traballar visualiza na táboa os datos dun obxecto **Persoa**, polo que cada cela da táboa corresponderase cunha propiedade do obxecto (nome, occupation, age Category, …). En casos coma este resulta interesante crear un modelo de datos personalizado para a táboa para poder personalizar e formatear a visualización dos datos.

A interface **TableModel** declara os métodos que se deben ter un modelo de táboa personalizado. A clase **AbstractTableModel** proporciona implementacións por defecto para a todos os métodos da interface **TableModel, excepto tres,** e encárgase do manexo de eventos e listeners.



Neste caso vaise crear un modelo personalizado para a táboa como subclase de **AbstractTableModel**, polo que só será necesario implementar os seguintes métodos:

- **getRowCount()**: devolve o número de filas da táboa.
- **getColumnCount()**: devolve o número de columnas da táboa.
- **getValueAt(int fila, int columna)**: devolve o valor que se mostra na cela da táboa indicada polos parámetros fila e columna.

Ademais dos métodos anteriores, tamén se se vai sobreescribir o método **getColumnName(int column)** que devolve o nome dunha columna. Se non se implementa este método, non se mostrarán as cabeceiras das columnas.

Neste apartado vaise mostrar como crear un modelo de táboa personalizado para o obxecto **Person** de exercicios anteriores. Este modelo permitirá traballar con unha **lista de persoas**, onde os atributos do obxecto Person se corresponderán coas columnas da táboa.

Para iso debe crearse unha nova clase chamada **PersonTableModel** dentro do paquete **.gui** que será a implementación personalizada do modelo de táboa**.** Esta clase estende de **AbstractTableModel,** que é unha clase abstracta, polo que haberá que implementar os seus métodos abstractos: **getRowCount, getColumnCount** e **getValueAt.** Neste caso tamén se vai implementar **getColumnName**.

```
public class PersonTableModel extends AbstractTableModel {
  private List<Person> personList;
  private String[] columnNames = {"ID", "Name", "Occupation", "Age Category",
        "Employment Category", "US Citizen", "Tax ID", "Gender"};

  public PersonTableModel() {  }

  public void setData(List<Person> personList) {
    this.personList = personList;
  }

  @Override
  public int getRowCount() {
    // o número de filas da táboa será o número de persoas na lista
  }

  @Override
  public int getColumnCount() {
    // devolver o número de columnas
  }

  @Override
  public Object getValueAt(int row, int column) {
    // obter o obxecto persoa na posición indicada por row
    // Se a columna é a 0, devolver o identificador da persoa (getId),
    // se a columna é a 1, devolver o nome da persoa, etc
    // se a columna está fóra do rango válido, devolver null
  }

 @Override
  public String getColumnName(int column) {
    if (column < 0 || column >= getColumnCount()) {
      throw new ArrayIndexOutOfBoundsException(column);
    } else {
      return columnNames[column];
    }
  }
}
```

Na aplicación vaise substituír a área de texto por un panel con unha táboa. Para iso crearase unha nova clase, **TablePanel** con BorderLayout, que estenda de JPanel e que conteña un compoñente **JTable** dentro dun **JScrollPane** na área central**.** O JScrollPane coloca automaticamente a cabeceira da táboa na parte superior do viewport e os nomes das columnas permanecen visibles na cima ao facer scroll.

```
public class TablePanel extends JPanel {

  private JTable table;
  private PersonTableModel personTableModel;

  public TablePanel() {
    personTableModel = new PersonTableModel();
    table = new JTable(personTableModel);

    setLayout(...);
    add(new JScrollPane(table), …);
  }

  public void setData(List<Person> personList) {
    personTableModel.setData(personList);
  }

  // para notificar á vista de que o modelo de datos cambiou
  public void refresh() {
    personTableModel.fireTableDataChanged();
  }
}
```

Cada vez que os datos do modelo son modificados por unha fonte externa, hai que chamar a un dos seguintes métodos (implementados en AbstractTableModel) para que a táboa se refresque e actualice os novos datos na vista: **fireTableCellUpdated, fireTableRowsUpdated, fireTableDataChanged, fireTableRowsInserted, fireTableRowsDeleted e fireTableStructureChanged .**

Tamén hai que modificar a clase MainFrame para substituír o textPanel polo TablePanel.

```
public class MainFrame extends JFrame {
  …
  private TablePanel tablePanel;

  public MainFrame() {
    …
    tablePanel = new TablePanel();
    tablePanel.setData(...);
    …
    formPanel.setFormListener(new FormListener() {
      public void formEventOccurred(FormEvent ev) {
        controller.addPerson(...);
        tablePanel.refresh();
      }
    });
    ….
  }
}
```

Exercicios:

1. Modifica a aplicación realizada en exercicios anteriores como se indica neste apartado, engadindo un TablePanel que conteña un JPanel con un modelo de datos personalizado (PersonTableModel).

   Cada vez que se pulse o botón OK do formulario lateral e se engada unha persoa á táboa, borra os datos do formulario.

Serialización

Neste apartado vaise engadir á aplicación a funcionalidade de exportar e importar os datos a un ficheiro.

A serialización de obxectos permite converter calquera obxecto nunha secuencia de bytes, así como reconstruír o obxecto a partir deses bytes no futuro. Este proceso permitirá almacenar obxectos en disco ou envialos pola rede.

Para que un obxecto poida serializarse, a súa clase debe implementar a interface **Serializable**. Esta interface non ten ningún método, só ten significado semántico. Isto significa que non hai que implementar ningún método a maiores no obxecto a serializar.

Ademais de implementar a interface Serializable, unha clase que se queira serializar debe ter unha variable privada e estática chamada **serialVersionUID.**

```
private static final long serialVersionUID = 1234L;
```

A API de Java utiliza esta variable para determinar se o obxecto a deserializar foi serializado coma mesma versión da clase do obxecto na que está intentando deserializalo. Imaxinar que un obxecto “persoa” foi serializado a disco. Máis tarde cámbiase a clase “Persoa” e inténtase deserializar o obxecto en disco na nova clase Persoa, que non son compatibles. Para detectar estes problemas utilízase a variable **serialVersionUID.** Cando se fai un cambio no código da clase do obxecto a serializar recoméndase cambiar tamén a variable **serialVersionUID.**

```
public class Person implements Serializable {
    private static final long serialVersionUID = 1234L;
}
```

NOTA: para que unha clase se poida serializar, todos os seus atributos deben ser tipos primitivos ou implementar a clase Serializable.

As clases ObjectInputStream e ObjectOutputStream están especializadas na lectura e escritura de obxectos.

O exemplo co que se está traballando ten unha clase chamada Database á que se lle van engadir métodos para almacenar e recuperar os datos dun ficheiro.

```
public class Database {
   private List<Person> people;
…
   public void saveToFile(File file) throws IOException {
      // crear un fluxo de saída a disco pasándolle un obxecto File
      FileOutputStream fos = new FileOutputStream(file);

      // Vincular o ObjectOuputStream co fluxo de saída
      ObjectOutputStream oos = new ObjectOutputStream(fos);

      // writeObject permite escribir a ficheiro un arrayList porque é un obxecto
      oos.writeObject(people);

      //cerrar os fluxos de saída
      oos.close();
      fos.close();
   }

   public void loadFromFile(File file) throws IOException {
      // crear un fluxo de entrada de disco pasándolle un obxecto File
      FileInputStream fis = new FileInputStream(file);
      // Vincular o ObjectInputStream co fluxo de entrada
      ObjectInputStream ois = new ObjectInputStream(fis);

      try {
         // para evitar problemas, eliminar elementos da lista e cargalos de ficheiro
         people.clear();
         people.addAll((LinkedList<Person>) ois.readObject());
      } catch (ClassNotFoundException e) {
         e.printStackTrace();
      }

      // cerrar os fluxos de entrada
      ois.close();
      fis.close();
   }
}
```

Dado que se quere que toda a comunicación se realice a través da clase controladora, engadiranse dous métodos nela para ler e gardar os obxectos en ficheiro:

```
public class Controller {
   private Database db = new Database();
….
   public void saveToFile(File file) throws IOException {
      db.saveToFile(file);
   }

   public void loadFromFile(File file) throws IOException {
      db.loadFromFile(file);
   }
}
```

Falta configurar as accións do menú no MainFrame para que invoquen os métodos anteriores. Así, a acción a executar para cargar datos de ficheiro será:

```
if (fileChooser.showOpenDialog(MainFrame.this) == JFileChooser.APPROVE_OPTION) {
    try {
        controller.loadFromFile(fileChooser.getSelectedFile());
        // refrescar os datos da táboa coa información lida de ficheiro
        tablePanel.refresh();
    } catch (IOException e) {
        // Mostrar mensaxe de erro
    }
}
```

E a acción a executar para gardar os datos no ficheiro será:

```
if (fileChooser.showSaveDialog(MainFrame.this) == JFileChooser.APPROVE_OPTION) {
    try {
        controller.saveToFile(fileChooser.getSelectedFile());
    } catch (IOException e) {
        // Mostrar mensaxe de erro
    }
}
```

Exercicios:

1. Modifica a aplicación realizada en exercicios anteriores como se indica neste apartado, engadindo a funcionalidade para permitir almacenar e recuperar datos dun ficheiro.

   Se se produce un erro ao ler ou escribir de ficheiro debe mostrarse unha mensaxe de erro nunha ventá independente modal.

2. Engade á aplicación un **menú popup** sobre a táboa de forma que apareza unha opción “Borrar fila”, que permita eliminar a fila sobre a que se pulsou co rato.

   Ademais, cando se mostre o menú popup, debe engadirse o código necesario para que a fila sobre a que se pulsa co rato apareza seleccionada.

   Recordar que a interface gráfica só se vai comunicar co modelo de datos a través da clase controladora, polo que para manter a mesma solución aplicada a exercicios anteriores, engadirase un listener ao TablePanel que terá un método que se execute ao eliminar unha fila (cando se seleccione a opción do menú de popup de “borrar fila da táboa”). A implementación do listener realizarase, como sempre, no MainFrame.

Utilizaríase un código coma o seguinte:

```
public TablePanel() {
  …
  removeItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
      int row = … // engadir o código para obter a fila seleccionada

      if (personTableListener != null) {
        personTableListener.rowDeleted(row);
        // para que visualmente se actualicen os datos
        personTableModel.fireTableRowsDeleted(row, row);
      }
    }
  });
}
```

```
public MainFrame() {
…
   tablePanel.setPersonTableListener(new PersonTableListener() {
     public void rowDeleted(int row) {
       controller.removePerson(row);
     }
   });
}
```

```
public class Controller {
  public void removePerson(int index) {
    …
  }
}
```

Barras de ferramentas

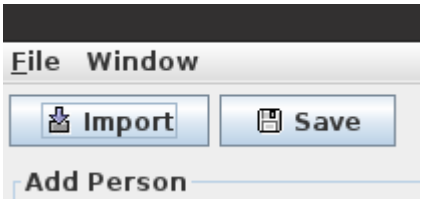
Unha barra de ferramentas (JToolBar) é un contedor que agrupa varios compoñentes, normalmente botóns con iconas, nunha fila ou columna. Serven para ofrecer acceso fácil a certas funcións da aplicación. How to Use Tool Bars.

```
// Creación dunha barra de ferramentas con título
JToolBar toolBar = new JToolBar("Still draggable");

…
// Engadir botóns á barra de ferramentas
JButton btnBoton = new JButton(...);
toolBar.add(btnBoton);

// engadir separador
toolBar.addSeparator();

// Engadir a barra de ferramentas ao panel
panel.add(tolBar);
```

NOTA: Por defecto, as barras de ferramentas poden arrastrarse a unha nova ventá.

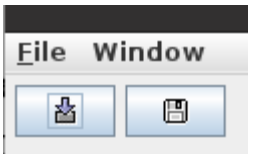
Exercicio:

1. Neste exercicio vanse realizar os pasos para transformar a barra de ferramentas (Toolbar.java) creada en exercicios anteriores nun JToolbar.

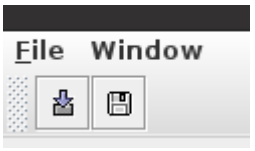
   

   Primeiro modifica a aplicación de tal forma que os botóns de Toolbar.java permitan abrir o diálogo de ler ou gardar arquivo (igual que as opcións de menú correspondentes). Recorda que a clase Toolbar tiña asociado un Listener, polo que terás que facer modificacións necesarias para incluír estas novas funcionalidades. É aconsellable cambiar os nomes de variables, clases, interfaces, etc. implicados para adaptar o código.

   Engade tamén aos botóns unha icona significativa. Para as iconas podes usar o Java Look and Feel Graphics Repository. Descárgao e coloca as imaxes no directorio **resources** do proxecto.

   Agora elimina o texto dos botóns, polo que a barra de ferramentas ten un aspecto máis parecido a un JToolBar. Neste caso sería boa idea engadir un **Tooltip** para que informe da funcionalidade dos botóns.

   Modifica o código da clase Toolbar para que en lugar de estender de JPanel estenda de JToolbar. Ademais, agora xa non será necesario establecer o layout, polo que debes eliminar a liña de código onde se establece. Esta simple modificación permite ter unha JToolBar funcional na aplicación.

   Proba a executar a aplicación e arrastra a barra de ferramentas fóra do sitio. ¿Pódese? Se é posible, ¿pódese colocar en calquera outra zona da ventá? ¿Como se faría para desactivar esta funcionalidade?.

JSplitPane

Un JSplitPane permite mostrar dous compoñentes, un ao lado do outro ou un arriba e outro abaixo. Tamén é posible aniñar varios JSplitPanes dividindo o espacio entre tres ou máis compoñentes.

O JSplitPane ten unha barra divisora entre os seus compoñentes que permite á persoa usuaria movela e redimensionar interactivamente o espacio asignado a cada compoñente.

Na cima da barra divisoria aparece dúas pequenas frechas que permiten ocultar un dos compoñentes con un simple clic. Este comportamento pode configurarse co método JSplitPane.setOneTouchExpandable(boolean newValue).

Comportamento por defecto dun JSplitPane:

- Por defecto, a barra divisoria entre compoñentes colócase de tal forma que o compoñente da esquerda teña o tamaño preferido e o resto do espacio sexa para o compoñente da dereita.
- Cando a ventá se fai máis grande, a barra divisoria permanece no mesmo sitio e o espacio extra vai para o compoñente da dereita.
- Cando a ventá se fai máis pequena, a barra divisoria permanece no mesmo sitio e o compoñente da dereita é o que cambia de tamaño.
- A ventá non pode facerse máis pequena que o tamaño mínimo dos compoñentes.

Este comportamento pode modificarse invocando os métodos apropiados do JSplitPane.

A barra divisoria pode colocarse nunha posición manualmente usando un dos métodos seguintes:

- setDividerLocation(double proportionalLocation)
- setDividerLocation(int location)

Tamén é posible resetear o JSplitPane co método resetToPreferredSizes().

Máis información de como usar JSplitPanes.

Exercicio:

1. Modifica a aplicación de exercicios anteriores de forma que agora teña na área central un JSplitPane do formulario na parte esquerda e a táboa na parte dereita.

   Despois de comprobar que o JSplitPane funciona, proba a ocultar o formulario lateral coa opción de menú Window -> Show -> Person Form. Se volves a mostrar o formulario lateral comprobarás que non se ve. Fai as modificacións necesarias para que se vexa o formulario ao activar a opción do menú “Person Form”.

Táboas editables

Actualmente a aplicación non permite editar directamente os datos da táboa porque se está usando un modelo de datos personalizado. Se se usase o DefaultTableModel, a táboa sería editable por defecto.

Para facer que o modelo de datos personalizado da táboa (**PersonTableModel**) sexa editable, hai que sobreescribir o método boolean isCellEditable(int row, int column). Por exemplo, para facer que a columna 1 sexa editable, o método anterior debe devolver true cando column é igual a 1.

Sen embargo isto non é suficiente, pois se se proba a aplicación compróbase que a cela é editable mais o dato modificado pérdese tan pronto como se pulse Enter. Para almacenar no modelo de datos o novo valor da cela, hai que sobreescribir o método setValueAt(Object aValue, int row, int column) en **PersonTableModel**. Este método debe actualizar o modelo de datos, é dicir, actualizar o dato da columna indicada no parámetro “column” para a persoa indicada no parámetro “row” co valor **aValue**.

Exercicio:

1. Modifica a aplicación creada en exercicios anteriores para que a columna “Name” sexa editable e non se perdan os datos modificados. Só debe ser editable a columna “Name”.

Cell Renderer

Pode pensarse que as táboas utilizan un compoñente para cada cela. Sen embargo, por razóns de rendemento, as táboas utilizan un “cell renderer” para todas as celas que conteñen o mesmo tipo de datos.

Un “***cell renderer***” é como un selo configurable para estampar o valor da cela no formato apropiado.

Por outra parte, cando unha persoa comeza a editar unha cela, é un “**cell editor**” o que toma o control do comportamento da cela.

Así, por exemplo, o cell renderer por defecto para unha columna que conteña números (Integer) utilizará unha instancia de JLabel aliñada á dereita para mostrar os números. Cando a cela comece a editarse, o “cell editor” por defecto será un JTextField aliñado á dereita.

O modelo de datos da táboa usado ata agora na aplicación considera que o tipo de datos de todas as celas é unha cadea de caracteres (String), polo que todas as celas usan o *“cell renderer”* de String.

Cando Swing ten que escoller o *renderer* que vai usar para visualizar o contido da cela, comproba se a columna ten un renderer personalizado. Se non ten un renderer personalizado, invócase o método ***getColumnClass*** do modelo da táboa e búscase un renderer rexistrado para ese tipo de datos. Por defecto os renders que se utilizan son:

- Boolean: check box.
- Number: etiqueta aliñada á dereita.
- Double, Float: igual que Number, utilizando unha instancia de NumberFormat para convertir a texto. Por defecto utiliza o formateador do Locale.
- Date: etiqueta que usa unha instancia de DateFormat.
- ImageIcon, Icon: etiqueta centrada.
- Object: etiqueta que mostra o valor do obxecto como String.

Cando se utiliza un modelo de datos personalizado para unha táboa, por defecto toda columna é de tipo **Object**. Se se quere precisar o tipo de dato da columna hai que sobreescribir o método public Class<?> getColumnClass(int columnIndex). Este método devolve a **clase** do tipo de dato en función do número de columna, por exemplo, para unha columna de tipo String debe devolver **String.class**.

Exercicio:

1. Modifica a aplicación creada en exercicios anteriores para que a columna “US Citizen” mostre un cell renderer de tipo checkbox. O resto das columnas deben mostrar un cell renderer para tipos de dato String.

   Fai que a columna “US Citizen” sexa editable e se actualice a súa modificación nos datos do modelo da táboa.

Renderer personalizado

Swing permite crear un renderer personalizado e rexistralo para todas as columnas da táboa que teñan un determinado tipo de dato utilizando o método setDefaultRenderer().

É fácil personalizar o texto ou imaxe renderizada por defecto por DefaultTableCellRenderer. Simplemente hai que crear unha subclase e implementar o método **setValue** para que invoque **setText** ou **setIcon** co texto ou imaxe apropiada. Exemplo:

```
static class DateRenderer extends DefaultTableCellRenderer {
    DateFormat formatter;
    public DateRenderer() { super(); }

    public void setValue(Object value) {
        if (formatter==null) {
            formatter = DateFormat.getDateInstance();
        }
        setText((value == null) ? "" : formatter.format(value));
    }
}
```

Se estender DefaultTableCellRenderer non é suficiente, pode construirse un renderer de forma fácil creando unha subclase do compoñente Swing (que se usará para mostrar o valor da cela) e facendo que implemente a interface TableCellRenderer. Simplemente haberá que implementar o método **getTableCellRendererComponent** onde se configurará o compoñente para que mostre o valor pasado como parámetro e devolva o propio compoñente. Exemplo de renderer para a columna EmploymentCategory:

```
public class EmploymentCategoryRenderer extends JComboBox implements TableCellRenderer {

  public EmploymentCategoryRenderer() {
    // inicializar o JComboBox cos valores da clase EmploymentCategory
    super(...);
  }

  @Override
  public Component getTableCellRendererComponent(JTable table, Object value,
          boolean isSelected, boolean hasFocus, int row, int col) {
    // value é o valor actual da cela a ser renderizada
    this.setSelectedItem(value);
    return this;
  }
}
```

Despois de creado o renderer personalizado hai que rexistralo para as columnas con tipo de dato EmploymentCategory.class. Todas as columnas da táboa da clase EmploymentCategory terían este *renderer* personalizado:

```
public TablePanel() {
  …
     table.setDefaultRenderer(EmploymentCategory.class,
        new EmploymentCategoryRenderer());
}
```

De momento a cela non é editable, pois só se implementou o renderer.

NOTA: Non se mostra explicitamente o código de PersonTableModel, aínda que habería que adaptar o método **getColumnClass** adecuadamente.

Exercicio:

1. Modifica a aplicación creada en exercicios anteriores para que a columna "Employment Category" teña un renderer personalizado.

   Comprobarás que visualmente as filas da táboa quedan pequenas para o novo renderer, polo que terás que buscar unha solución para mellorar o aspecto da aplicación.

   Investiga como modificar a clase **EmploymentCategory** para que teña un método **toString** que mostre un texto máis apropiado que o nome das constantes. Por exemplo, debe mostrar "self employed" en lugar de "SELFEMPLOYED" (non serve cambiar o nome das constantes).

Editor de cela personalizado

Swing tamén permite personalizar o editor de cela usado nas táboas. Así, por exemplo, para establecer un combo box como un editor simple pode utilizarse o seguinte código:

```
// crear o comboBox e engadir os seus elementos
JComboBox comboBox = new JComboBox();
comboBox.addItem(...);

TableColumn employmentCategoryColumn = …
employmentCategoryColumn.setCellEditor(new DefaultCellEditor(comboBox));
```

Se non serven como editores un TextField, un check box ou un comboBox, é posible crear un editor personalizado. Para isto haberá que crear unha clase que implemente TableCellEditor.

Partirase da clase **AbstractCellEditor**, que proporciona implementacións dos métodos da interface CellEditor, superinterface de **TableCellEditor,** evitando ter que implementar o código para lanzar os eventos necesarios nun editor de celas. Polo tanto, só haberá que implementar os seguintes métodos no editor personalizado:

- getCellEditorValue(): devolve o valor actual da cela.
- getTableCellEditorComponent(...): devolve o compoñente a usar como editor.

O seguinte exemplo mostra unha implementación dun editor personalizado que utiliza un ComboBox como compoñente personalizado para mostrar na cela da táboa. Só se mostra a modo de exemplo. Tal e como se viu anteriormente, non sería necesario un editor personalizado para un ComboBox.

```
public class EmploymentCategoryEditor extends AbstractCellEditor implements
      TableCellEditor {

  private JComboBox combo;

  public EmploymentCategoryEditor() {
    // inicializar o JComboBox cos valores da clase EmploymentCategory
    combo = new JComboBox(...);

    combo.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent ae) {
        // para parar de usar o editor e facer que se mostre o renderer
        // e actualizar o valor no TableModel
        fireEditingStopped();
      }
    });
  }

  @Override
  public Object getCellEditorValue() {
    return combo.getSelectedItem();
  }

  @Override
  public Component getTableCellEditorComponent(JTable table, Object value,
        boolean isSelected, int row, int column) {
    // value é o valor actual da cela a ser renderizada
    combo.setSelectedItem(value);

    return combo;
  }
}
```

NOTA: Non se mostra explicitamente o código de PersonTableModel, aínda que habería que adaptar os métodos **getColumnClass**, **isCellEditable** e **setValueAt** adecuadamente.

Falta asignar á táboa o editor personalizado ás columnas da clase EmploymentCategory. Todas as columnas da táboa da clase EmploymentCategory terían este editor personalizado.

```
public TablePanel() {
  …
  table.setDefaultEditor(EmploymentCategory.class,
        new EmploymentCategoryEditor());
}
```

Tamén é posible crear editores personalizados para outro tipo de datos. En internet poden encontrarse múltiples exemplos, entre eles, como crear un renderer personalizado para celas de tipo Date.

Exercicio:

1. Modifica a aplicación de exercicios anteriores e engade un editor de cela personalizado como se indica neste apartado.
2. Imaxina que tes unha lista de cores. Crea unha aplicación que mostre as cores nunha táboa utilizando un renderer personalizado que teña unha etiqueta con cor de fondo o valor de cada unha das cores de cada fila da táboa.

   Engade á táboa un editor personalizado para as cores que sexa un botón que ao pulsarse mostre un diálogo JColorChooser e permita cambiar a cor seleccionada.

Ordenación dun JTable

Neste apartado vaise explicar como ordenar un JTable.

En exercicios anteriores os datos para a táboa estaban almacenados nun obxecto Database. Neste caso non se vai tocar este obxecto, nin ordenar os elementos que almacena, a ordenación farase simplemente na vista, no JTable.

Para que se poida ordenar a táboa basta con pasarlle á táboa unha instancia de TableRowSorter, que é unha implementación de RowSorter e que permite ordenar e filtrar os datos dun TableModel. Isto permitirá que ao pulsar sobre a cabeceira dunha columna se ordenen as filas da táboa visualmente:

```
TableRowSorter sorter = new TableRowSorter(personTableModel);
table.setRowSorter(sorter);
```

Cando se usa un sorter, a información que se mostra na vista pode estar en orde diferente á información almacenada do modelo. É dicir, todos os métodos da táboa baseados en filas, como por exemplo **getSelectedRow**, fan referencia á vista e non ao modelo subxacente. Polo tanto, é necesario facer unha conversión entre os dous. Esta conversión será necesario facela sempre que se faga unha ordenación da táboa.

```
int filaVista = table.getSelectedRow();
int filaModelo = table.convertRowIndexToModel(filaVista);
```

Tamén pode definirse a orde por defecto para unha columna. Para iso utilízase a clase SortKey que describe o método de ordenación para unha determinada columna. Por exemplo, o seguinte código define que a columna 0 terá orde ascendente.

SortKey sortKey = new RowSorter.SortKey(0, SortOrder.ASCENDING);

Por exemplo, para establecer a ordenación por defecto da columna 0 en orde ascendente haberá que facer o seguinte:

```
List<RowSorter.SortKey> sortKeys = new ArrayList<RowSorter.SortKey>();
sortKeys.add(new RowSorter.SortKey(0, SortOrder.ASCENDING));
sorter.setSortKeys(sortKeys);
```

Filtrar datos dun JTable

Cando temos unha táboa con moitos datos pode resultar interesante establecer un filtro para mostrar só as filas que conteñan unha cadea específica.

O seguinte exemplo de código permite mostrar só as filas da columna 1 que conteñan o texto almacenado na variable “resposta”:

```
RowFilter<PersonTableModel, Object> rf = null;
//If current expression doesn't parse, don't update.
try {
  rf = RowFilter.regexFilter(resposta.toString(), 1);
} catch (java.util.regex.PatternSyntaxException e) {
  return;
}
sorter.setRowFilter(rf);
```

Exercicio:

1. Engade unha opción de menú á nosa aplicación para filtrar por nome. Programa este menú para que mostre un JOptionPane solicitando o texto polo que filtrar. Despois, a táboa deberá mostrar só as filas que coincidan co texto escrito.

Diálogos personalizados

Un JFrame é o contedor de alto nivel que se utiliza como pantalla de inicio de todas as aplicacións de Swing. Este compoñente encapsula unha ventá clásica dun sistema operativo con contorna gráfica.

Nunha aplicación Java con ventás debería haber un único JFrame, correspondente á ventá principal da aplicación. O resto de ventás secundarias deberían ser JDialog, aínda que non é a única posibilidade, pois tamén poderían crearse ventás con JOptionPane.

Un JDialog é unha ventá secundaria que só ten o botón "X" (Cerrar) e non ten os botóns para redimensionar a ventá.

Un dos construtores de JDialog é:

public JDialog(Frame owner, String title, boolean modal)

- **owner** fai referencia ao Frame ancestro
- **title** fai referencia ao título do diálogo
- **modal** especifica se o diálogo bloquea a interacción da persoa usuaria con outras ventás ancestras da aplicación. É dicir, un diálogo modal (modal = true) deshabilita todas as demais ventás da aplicación ata que a persoa usuaria o peche.

JDialog admite como ascendentes un JFrame ou outro JDialog. Isto é importante porque unha ventá filla sempre quedará por riba da súa ventá ancestra. Se ao crear un JDialog se pasa como parámetro o JFrame ancestro, o diálogo sempre estará visible por riba do JFrame.

Tamén pode utilizarse o frame ancestro para centrar o diálogo con respecto a el utilizando o método:

dialogo.setLocationRelativeTo(parent)

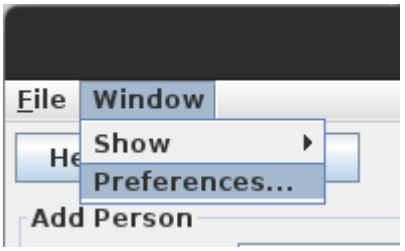
Un diálogo, ademais, é dependente do JFrame da aplicación. Cando o JFrame é destruído, tamén o é o diálogo. O mesmo pasa cando o JFrame se minimiza, o diálogo tamén o fai.

NOTA: para que non haxa problemas, cando hai varias ventás modais simultaneamente en pantalla, cada unha que se mostre debe ser descendente da anterior. É dicir, debe haber unha xerarquía entre elas. Ter dúas ventás modais no mesmo nivel da xerarquía (irmáns, primas, etc), dará problemas.

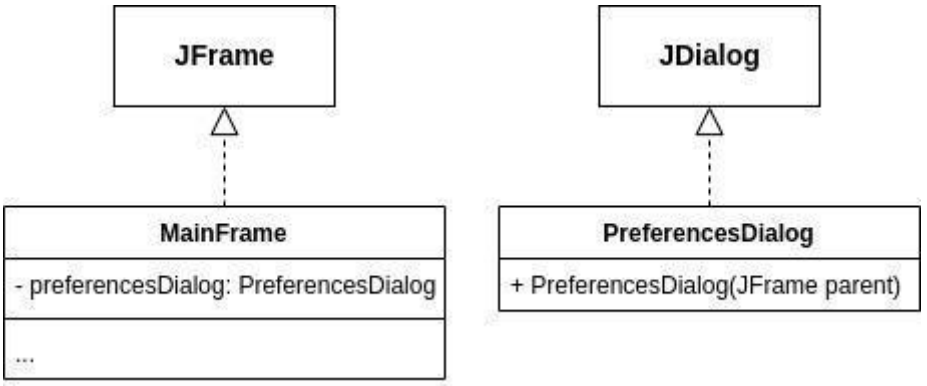
Exercicio:

1. Engade ao menú da aplicación creada en exercicios anteriores unha opción "Preferences…" que abra un novo diálogo. De momento abrirase unha ventá sen compoñentes, pois incluiranse máis adiante.

   Para iso, crea a clase **PreferencesDialog** que estenda de **JDialog** e fai que o seu construtor cree



un diálogo con **título** “Preferences” que non sexa modal.



Fai que cando o diálogo se mostre apareza sempre centrado con respecto ao JFrame.

NOTA: Por convenio, cando un menú abre unha nova ventá, engádense os puntos suspensivos (...).

JSpinner

Un JSpinner é un compoñente que permite seleccionar un número ou un obxecto dunha secuencia ordenada. Normalmente proporciona un par de botóns con frechas para moverse entre os elementos da secuencia. Ainda que o JComboBox tamén proporciona unha funcionalidade similar, os JSpinners son preferidos algunhas veces porque non requiren despregar unha lista.

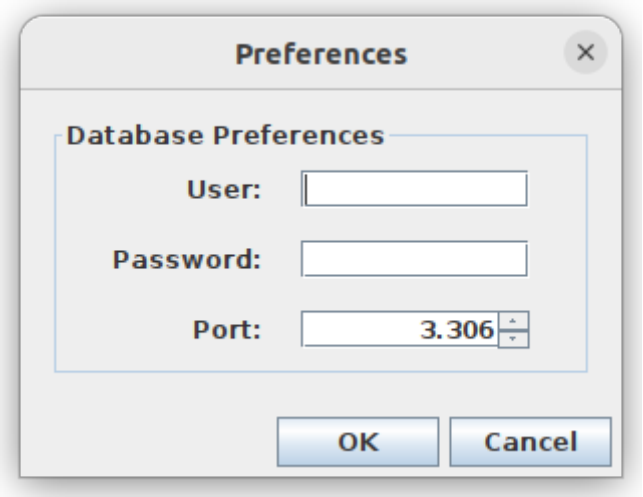
A secuencia de valores do JSpinner está definida no seu modelo (SpinnerModel). Java Swing ten clases de modelos predefinidas para os tipos comúns: SpinnerListModel, SpinnerNumberModel e SpinnerDateModel.

Exemplo de creación dun JSpinner:

```
SpinnerNumberModel spinnerModel = new SpinnerNumberModel(value, min, max, step);
JSpinner portSpinner = new JSpinner(spinnerModel);
```

Exercicio:

1. Na aplicación creada en exercicios anteriores, configura o diálogo personalizado cos seguintes compoñentes, de tal forma que teña un aspecto coma o da seguinte imaxe.



Notas:

- O diálogo de preferencias está formado por dous paneis: un para o formulario e outro para os botóns.
- O diálogo ten un BorderLayout que ten no centro o panel co formulario e na parte de abaixo o panel cos botóns.
- O panel co formulario ten un GridBagLayout.
- O jspinner permite escoller un número entre 0 e 9999, sendo o 3306 o valor por defecto e 1 o incremento.
- Fai que o caracter que se mostre por defecto no campo de contrasinal sexa "*".
- Fai que os dous botóns (OK e Cancel sexan do mesmo tamaño).
- Engade un *acelerator* ao menú do frame principal para abrir o diálogo utilizando a combinación Ctrl + P.
- Ao pulsar o botón OK, debes mostrar con un System.out.println o valor de todos os campos do formulario e tamén cerrar o diálogo.
- Ao pulsar o botón Cancel, debe cerrarse o diálogo.

2. Seguindo a técnica explicada no apartado Separación de compoñentes, para ter acceso dende o MainFrame aos datos introducidos no diálogo de preferencias e manter os compoñentes o máis independentes posibles utilizarase un Listener. A implementación deste Listener faise na clase MainFrame, polo que é nesta clase onde se define a súa funcionalidade.

   Crea unha interface **PreferencesListener** con un método chamado **setPreferences** que se executará cada vez que se pulse o botón OK do diálogo.

   Asigna ao diálogo unha implementación da interface PreferencesListener na "MainFrame". De momento, dende MainFrame, imprime por pantalla os datos de todos os campos do formulario.

Java Preferences

A API Java Preferences proporciona unha forma de almacenar as preferencias dun programa. É dicir, permite almacenar información como, por exemplo, o último valor introducido nun campo dun formulario.

As preferencias son pares chave/valor, onde a chave é un nome arbitrario para a preferencia. O valor pode ser un booleano, cadea, enteiro ou outro tipo primitivo.

As preferencias pódense ler e almacenar utilizando métodos **get** e **put.** O método get tamén permite proporcionar un valor por defecto no caso de que a preferencia non estea establecida.

Esta API proporciona unha funcionalidade útil eliminando a sobrecarga de traballo de implementar o almacenamento das preferencias por parte das persoas desenvolvedoras.

Actualmente, a forma de almacenar os datos é dependente da plataforma: pode ser no rexistro do sistema operativo ou nun arquivo. Mais isto faise de forma transparente, polo que non hai que preocuparse á hora de programar.

As Preferences son fáciles de usar. Primeiramente hai que definir un nodo onde almacenar os datos:

```
Preferences prefs = Preferences.userRoot().node(this.getClass().getName());
```

Para establecer o valor das preferencias utilízase o método put:

```
prefs.putBoolean(ID1, false);
prefs.put(ID2, "Hello Europa");
prefs.putInt(ID3, 45);
```

Para acceder aos valores, utilízase o método get. Se o valor non está definido, devolve o valor por defecto que se pasa como segundo parámetro:

```
prefs.getBoolean(ID1, true);
prefs.get(ID2, "Hello World");
prefs.getInt(ID3, 50);
```

Máis información: Java Preferences API - Tutorial

Exercicio:

1. Modifica a aplicación creada en exercicios anteriores para almacenar os valores do diálogo de preferencias usando a API Java Preferences.

   Cada vez que se pulse o botón “Ok” do formulario, deben actualizarse os valores almacenados en *Preferences*.

   Cada vez que se lance o diálogo de preferencias, debe aparece cuberto o formulario cos valores dos campos almacenados en *Preferences.*

Lanza a aplicación varias veces e comproba tamén que se conservan os valores no formulario entre diferentes execucións.

2. Un JTabbedPane é un compoñente que permite que a persoa usuaria cambie entre un grupo de compoñentes, pulsando nunha pestana con título.

   Modifica a aplicación de exercicios anteriores para que o compoñente da dereita do JSplitPane sexa un JTabbedPane que inclúa nunha pestana a táboa (TablePane) e noutra o TextPanel de exercicios anteriores.

   Información de como usar un JTabbedPane.

Creación dun modelo de táboa xenérico

En apartados anteriores dos apuntes viuse como crear un modelo personalizado para unha táboa.

Pode implementarse un modelo de táboa máis completo e personalizado baseándose na codificación deste enderezo web onde se dá unha implementación xenérica para calquera obxecto utilizando Java Generics.

Na mesma web, nestra outra páxina explica como crear un modelo de táboa concreto, para un tipo de obxecto determinado, estendendo o modelo xenérico. É dicir, créase un modelo para unha táboa xenérico que pode utilizarse para calquera táboa.

Pasos para usar o modelo de datos xenérico de táboa:

- Descargar a clase GenericDomainTableModel.

```
public abstract class GenericDomainTableModel<T> implements TableModel {}
```

- Crear unha implementación concreta reutilizando o modelo xenérico anterior. Para iso hai que crear unha nova clase que estenda a xenérica e implementar tres métodos:

```
public class PersonTableModelGeneric extends GenericDomainTableModel<Person> {
...}
```

- Na clase **PersonTableModelGeneric** implementar os métodos:
  - Class<?> getColumnClass(int columnIndex)
  - Object getValueAt(int rowIndex, int columnIndex)
  - void setValueAt(Object value, int rowIndex, int columnIndex)

  Podes basearte no exemplo que se proporciona ao inicio desta páxina para facelo.

- Ao usar o modelo de táboa xenérico, cando se modifica unha persoa do modelo, pode modificarse o obxecto directamente e notificar ao modelo usando o método **notifyDomainObjectUpdated.**

Primeira GUI con NetBeans

Nos exemplos anteriores creouse unha aplicación Swing programando a man dende cero. Neste apartado imos ver un exemplo de creación dunha interface gráfica de usuario en NetBeans.

Pasos a realizar:

- Crear un novo proxecto co nome **SumaNumeros** (**Ficheiro -> Novo proxecto -> Java with Maven -> Java Application**).
- Eliminar a clase creada por defecto (App.java).

Para construír a interface gráfica necesitamos un contedor Java no que se situarán o resto dos compoñentes da interface:

- No paquete por defecto creado no proxecto, crear un novo JFrame chamado **PantallaPrincipal**. Actuará como pantalla principal da aplicación.

Unha verz creado o JFrame, veremos a interface de deseño do IDE preparada para arrastrar compoñentes directamente ao frame.

Observa os diferentes compoñentes do IDE:

- **Paleta**: contén todos os compoñentes ofrecidos pola API de Swing.
- **Área de deseño**: zona onde se construirá a interface gráfica. Ten dúas vista: fonte e deseño, podendo intercambiar entre elas para traballar de forma diferente.
  Observar, na vista fonte, o código xerado automaticamente que inclúe a acción a realizar por defecto cando se pecha a aplicación e un layout por defecto.
- **Editor de propiedades**: permite editar as propiedades de calquera compoñente.
- **Navegador:** contén os elementos gráficos do formulario e permite que sexan facilmente localizados.

Para deseñar un JFrame simplemente hai que ir arrastrando compoñentes á área de deseño e o IDE xera automaticamente o código apropiado.

Poden modificarse as propiedades de calquera compoñentes dende o editor de propiedades.

Principais propiedades dun JFrame:

- **title**: título da ventá.
- **preferredSize**: establece o tamaño preferido do Frame. É un tamaño que proporcionará unha visión adecuada dos compoñentes.
  NOTA: Non todos os layout repetan esta variable.
- **location**: indica a posición (X,Y) na que aparecerá o Frame ao inicio.
- **bounds**: permite, nunha soa propiedade, especificar as coordenadas da posición inicial do JFrame e o ancho e alto inicial.
- **minimumSize** e **maximumSize**: permiten especificar o tamaño máximo e mínimo da ventá.
- **resizable**: permite configurar se a ventá é redimensionable ou non.

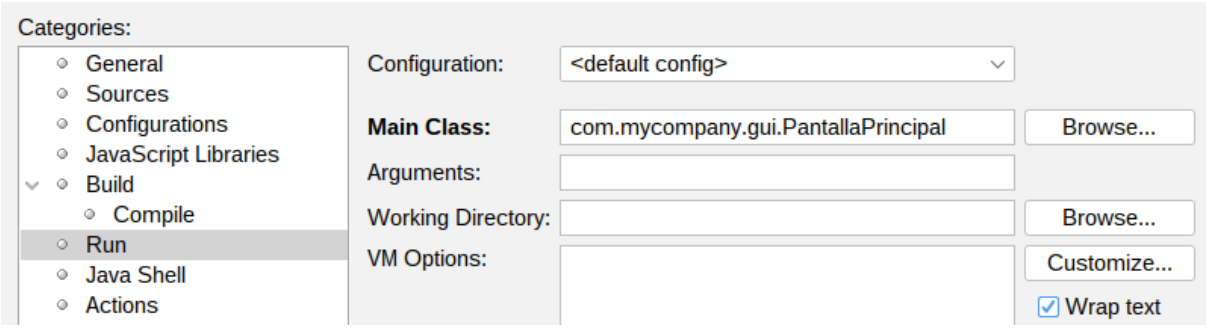
Todas estas propiedades poden establecerse de forma gráfica utilizando o IDE. Unha vez establecidas recoméndase revisar o código que xera NetBeans.

Se se quere que a ventá apareza centrada na pantalla haberá que realizar o seguinte: botón dereito sobre o JFrame -> **Properties** -> pestana "**Code**" -> Activar "**Generate Center**".

Observar que o código que crea NetBeans para centrar a ventá na pantalla é:

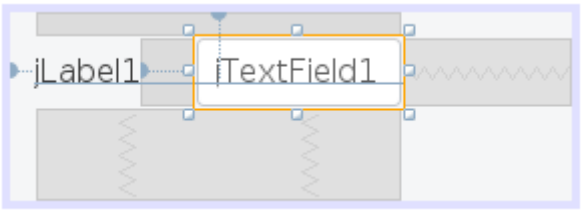
```
setLocationRelativeTo(null);
```

Pode probarse a executar a aplicación. Para iso, comprobar nas propiedades do proxecto que a clase a executar sexa PantallaPrincipal.



O Layout por defecto dun JFrame é **Free Design,** que significa que se poden colocar compoñentes en calquera posición definida pola persoa deseñadora. O IDE xera o código automaticamente para lograr o deseño indicado.

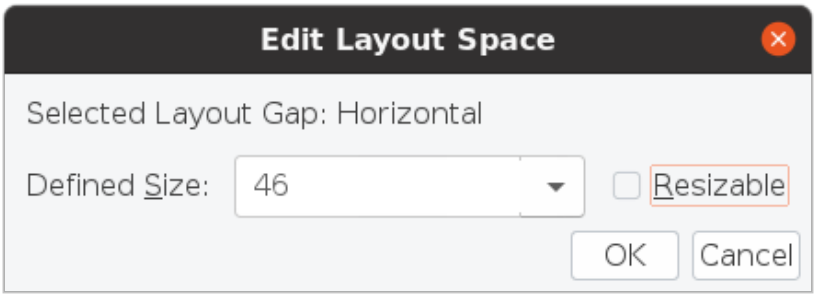
Ao ir engadindo compoñentes, o IDE proporciona información visual que axuda no deseño á hora de que os compoñentes se coloquen aliñados. Esta información especifica a ancoraxe dos compoñentes e as relacións entre eles mediante unhas liñas e ocos. Máis información sobre lenda e comentarios visuais de NetBeans.



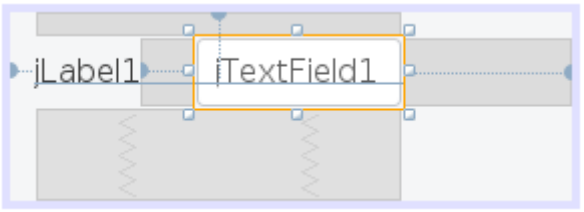
Os ocos (***gaps***) que aparecen coas liñas en zigzag, significa que son redimensionables. É dicir, se a ventá se redimensiona, estes bloques redimensiónanse para axustarse ao novo tamaño.

Os ocos que teñen unha liña recta son de tamaño fixo, polo que aínda que aumente o tamaño da ventá, o tamaño destes bloques non cambia.

Podemos cambiar un oco redimensionable para que sexa de tamaño fixo pulsando co botón dereito sobre o oco -> **Edit Layout space** e desmarcando a opción **Resizable**.



E o deseño quedará así:



Pero se os ocos non son redimensionables, teremos que permitir que a caixa de texto poida medrar. Para iso pulsamos co botón dereito sobre ela **-> Auto resizing -> Horizontal.**

É interesante comprobar tamén a ancoraxe dos compoñentes (pulsar co botón dereito sobre o compoñente -> **Anchor**) e tamén comprobar o seu comportamento cando se redimensiona a ventá (botón dereito sobre o compoñente -> **Auto resizing**).

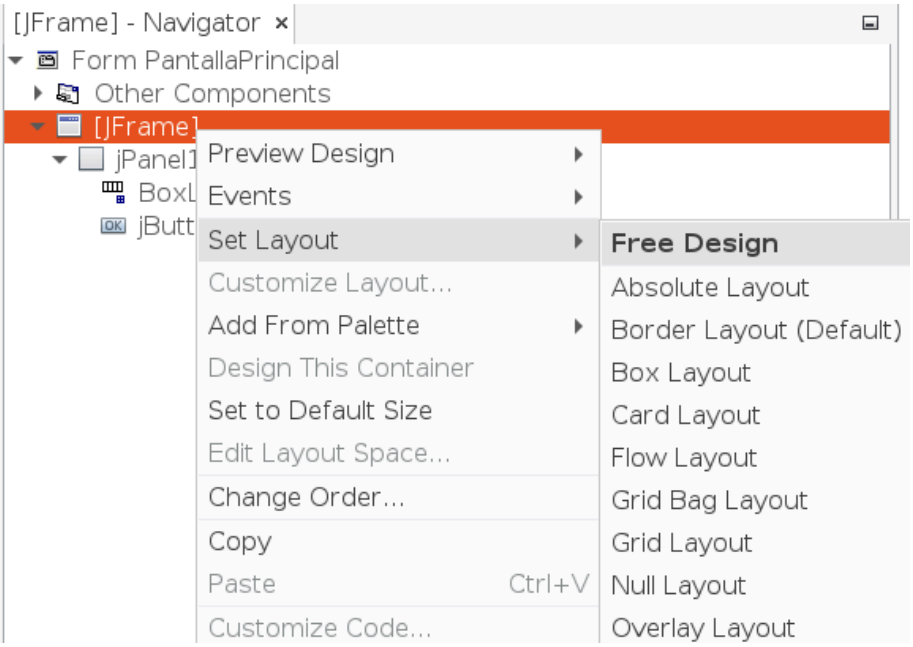
Este comportamento é o que aplica NetBeans por defecto e que se chama Deseño Libre (**Free Design**).

Este tipo de Layout diferencia as distancias horizontais e verticais, polo que haberá que configuralas por separado.

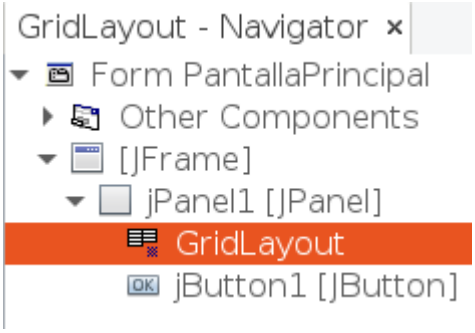
O Deseño Libre permite que os elementos manteñan unha distribución relativa independentemente do tamaño da ventá. É dicir, os elementos redistribúense ao cambiar o tamaño da ventá.

Con este deseño, ás veces pérdese o control sobre os elementos e non se colocan na posición desexada debido a que teñen que manter unhas distancias relativas con outros.

O Layout pode cambiarse pulsando co botón dereito sobre o Frame ou contedor:



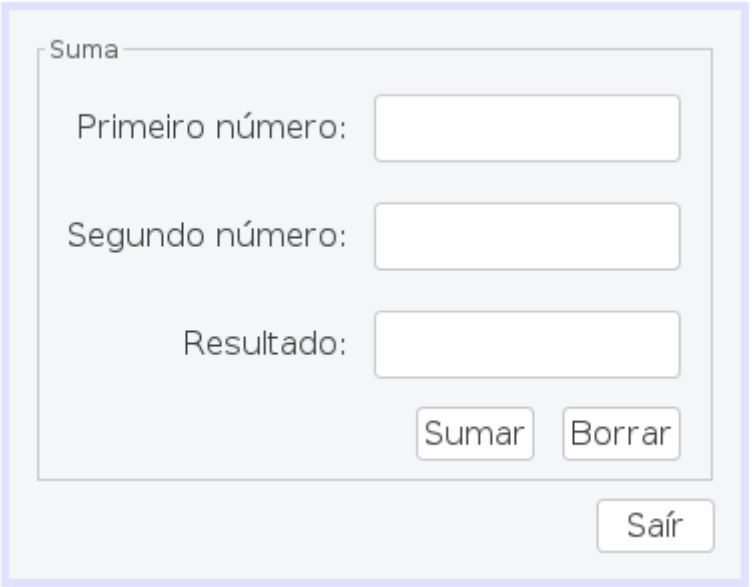
Unha vez se selecciona un Layout, aparecerá a propiedade asignada ao contedor na ventá "Navigator". Observar tamén que en NetBeans dende a ventá de propiedades poderanse configurar as diferentes opcións para este Layout.



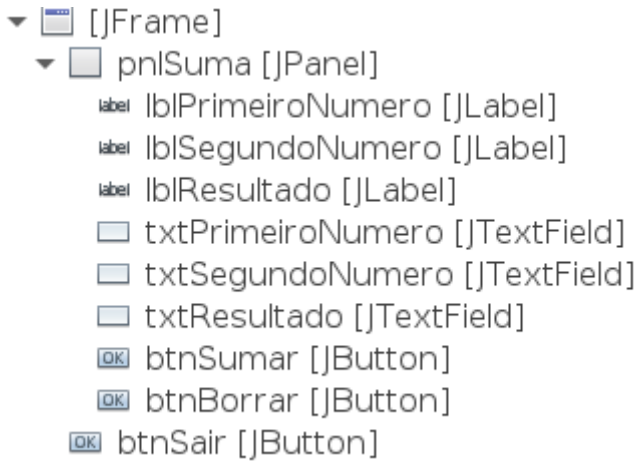
Unha aplicación pode ter un Frame con un Layout definido e engadírselle un Panel ao que se estableza un Layout diferente. Cada contedor pode ter un Layout específico.

Crear unha aplicación cos seguintes compoñentes:

- Engadir ao JFrame un JPanel, 3 etiquetas (JLabel), 3 campos de texto (JTextFields) e 3 botóns.
- O título do panel establécese na propiedade **border,** seleccionando o valor **TitledBorder** e establecendo o valor no **Title**.
- Configura o texto e a disposición dos elementos anteriores para que teñan o seguinte aspecto:



- Configura as seguintes restricións:
    - As etiquetas deben estar aliñadas á dereita.
    - As etiquetas e as caixas de texto están aliñadas pola liña base (*baseline*).
    - As caixas de texto son do mesmo tamaño e redimensionables en horizontal.
    - O texto que se escriba nas caixas de texto debe estar aliñado á dereita.
    - O panel tamén é redimensionable horizontalmente.
    - Fai que o borde dereito do botón **Borrar** coincida á mesma altura que o borde dereito das caixas de texto.
- Modifica o nome das variables como se indica na seguinte imaxe:



A continuación hai que engadir funcionalidade aos botóns. Isto faise engadindo un manexador de eventos a cada un, que responderá ao evento correspondente. Neste caso queremos que respondan ao evento de pulsar o botón, ben sexa por un clic do rato ou mediante o teclado.

- Facer dobre clic no botón “Saír”.

  Ao facer isto, NetBeans automaticamente engade un ActionListener ao botón “Saír” e crea o método a executar cando se detecta o evento, neste caso o ***actionPerformed***.

```
btnSair.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent evt) {
            btnSairActionPerformed(evt);
        }
});
```

  Tan só haberá que proporcionar a implementación do método btnSairActionPerformed. Dado que cando se pulse o botón, queremos que a aplicación remate, teremos que escribir a seguinte instrución:

```
private void btnSairActionPerformed(java.awt.event.ActionEvent evt) {
    System.exit(0);
}
```

- Engadir funcionalidade ao botón “Borrar”. Facer dobre clic no botón e completar o código:

```
private void btnBorrarActionPerformed(java.awt.event.ActionEvent evt) {
    txtPrimeiroNumero.setText("");
    txtSegundoNumero.setText("");
    txtResultado.setText("");
}
```

- Executar a suma. Facer dobre clic no botón “Sumar” e completar o seguinte código:

```
private void btnSumarActionPerformed(java.awt.event.ActionEvent evt) {
    float numero1, numero2, resultado;

    numero1 = Float.parseFloat(txtPrimeiroNumero.getText());
    numero2 = Float.parseFloat(txtSegundoNumero.getText());
    resultado = numero1 + numero2;

    txtResultado.setText(String.valueOf(resultado));
}
```

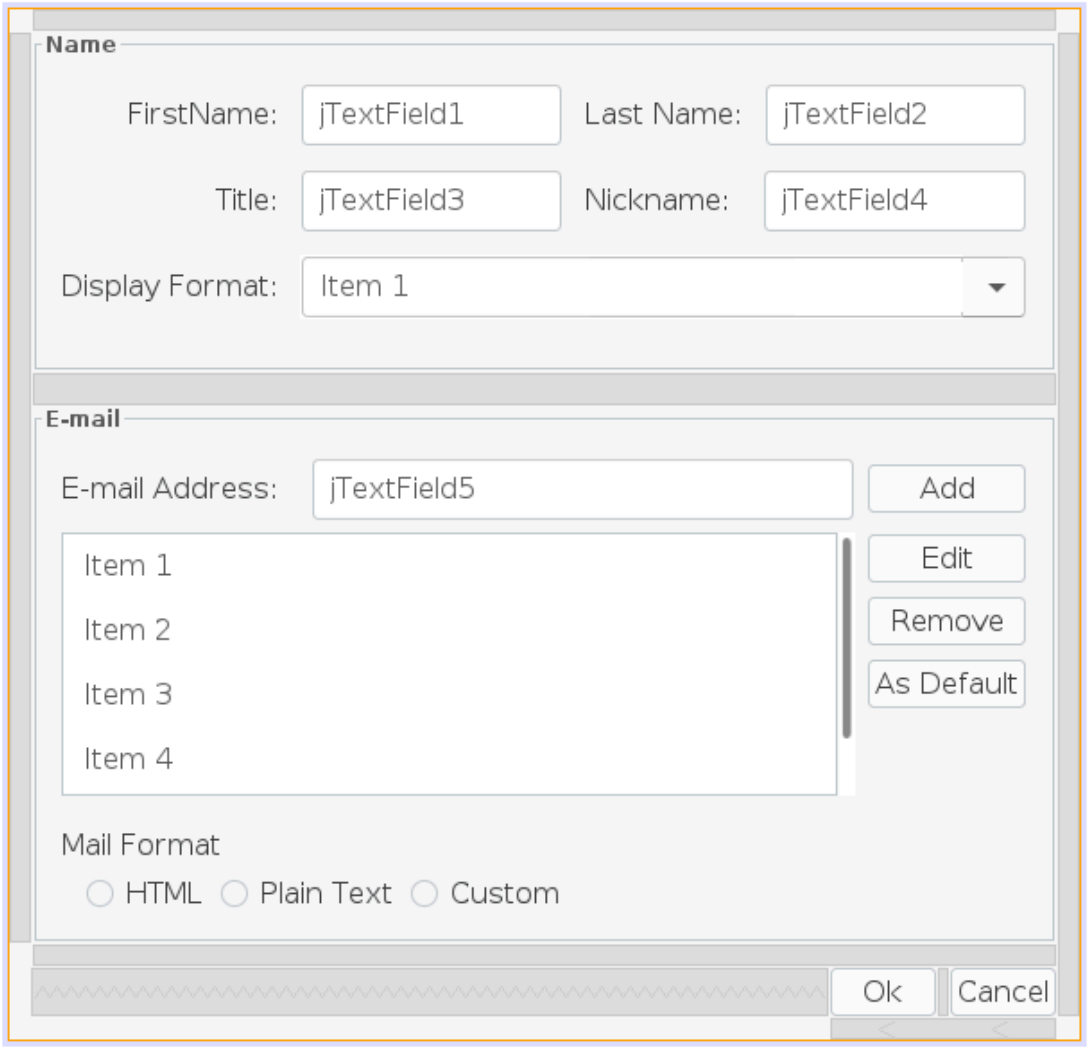
Tan só queda executar o programa. Para iso pódese pulsar o menú **Executar -> Executar Proxecto (F6).**

Máis información:

- Learning Swing with the NetBeans IDE
- Designing a Swing GUI in NetBeans IDE

Exercicios:

1. Crea un novo frame que conteña unha caixa de texto, un botón e un comboBox. Cada vez que se pulse o botón engadirase o texto da caixa de texto como unha nova entrada no comboBox e borrarase o texto da caixa de texto. Fai o deseño utilizando o IDE de NetBeans. Decide ti a disposición dos elementos. Asegúrate de que cando a ventá se redimensiona, os elementos tamén.
2. Usando NetBeans deseña unha aplicación similar á da seguinte imaxe:



3. Dos exercicios realizados no apartado de Layout, escolle dous e impleméntaos usando a axuda da ferramenta gráfica de NetBeans.

JFrame e JDialog en NetBeans

Exemplo de creación dunha aplicación con un JFrame que contén un botón que, ao pulsarse, abre un diálogo e este, á súa vez, ten un botón que ao ser pulsado volve ao frame principal.

Pasos a realizar:

- Crear unha aplicación e engadirlle un JFrame.
- Engadir un botón ao JFrame, axustar o seu tamaño e cambiar o texto que mostra.
- Cambiar o nome da variable botón por **btnDialogo**.

A continuación vaise engadir un formulario JDialog á aplicación:

- Pulsar co botón dereito sobre o nome da aplicación **-> Novo -> JDialog**. Se non aparece, pulsar Novo -> Outro -> Formulario de interface gráfica -> JDialog.
- Asignar o nome “PantallaSecundaria”.

Visualmente, en NetBeans, un JDialog é igual a un JFrame, sen embargo, un JDialog está destinado a pantalla secundaria.

Unha aplicación terá un JFrame como pantalla principal e o resto de ventás que haxa que abrir, serán, por exemplo, JDialog.

Un JFrame non admite ningunha ventá como pai. JDialog si que admite como ascendentes un JFrame ou un JDialog. Unha ventá filla sempre quedará por riba da súa ventá pai.

A continuación:

- Engadir un botón ao JDialog, axustar o seu tamaño e cambiar por “Volver”.
- Cambiar o nome da variable botón por “btnVolver”.

Agora imos configurar a funcionalidade da aplicación. Queremos que cando a persoa usuaria pulse o botón “Abrir diálogo” se abra a pantalla secundaria.

- Facer dobre clic no botón **btnDialogo**. Isto abre a vista de código. Escribir o código a executar cando unha persoa pulse sobre o botón:

```
private void btnDialogoActionPerformed(java.awt.event.ActionEvent evt) {
        PantallaSecundaria pantallaSecundaria = new PantallaSecundaria(this, true);
        pantallaSecundaria.setVisible(true);
}
```

  Cando se constrúe un diálogo hai que pasar dous parámetros: frame pai e un booleano que indica se o diálogo é modal ou non.

  Neste caso o frame pai vai ser a pantalla principal (this), que é o frame que está creando o diálogo.

  Se un diálogo é modal (true) significa que non se poderá acceder a outras ventás da aplicación ata que esta se cerre.

Na pantalla secundaria queremos que ao pulsar o botón “Volver” se peche o diálogo. Para iso:

- Facer dobre clic no botón “Volver”, co que se abrirá a vista de código.
- O código necesario para ocultar o diálogo é:

```
private void btnVolverActionPerformed(java.awt.event.ActionEvent evt) {
        setVisible(false);
}
```

Para executar a aplicación pulsamos co botón dereito sobre o JFrame -> Executar arquivo. Unha vez compilada a aplicación mostrarase o JFrame e ao pulsar o botón mostrarse o diálogo.

Probar que o diálogo é modal, é dicir, non se pode pulsar noutra zona da aplicación mentres o diálogo se mostra.

Exercicios:

1. Crea unha nova aplicación que ao pulsar un botón abra un diálogo non modal. Comproba cantos diálogos modais podes abrir.

Eventos en NetBeans

Cando en NetBeans facemos dobre clic sobre un botón, créase un novo método que se executa cando unha persoa pulsa un botón.

O seguinte método imprime unha mensaxe cada vez que unha persoa pulsa o botón.

```
private void jToggleButtonProbaActionPerformed(java.awt.event.ActionEvent evt) {
      System.out.println("O botón foi pulsado");
}
```

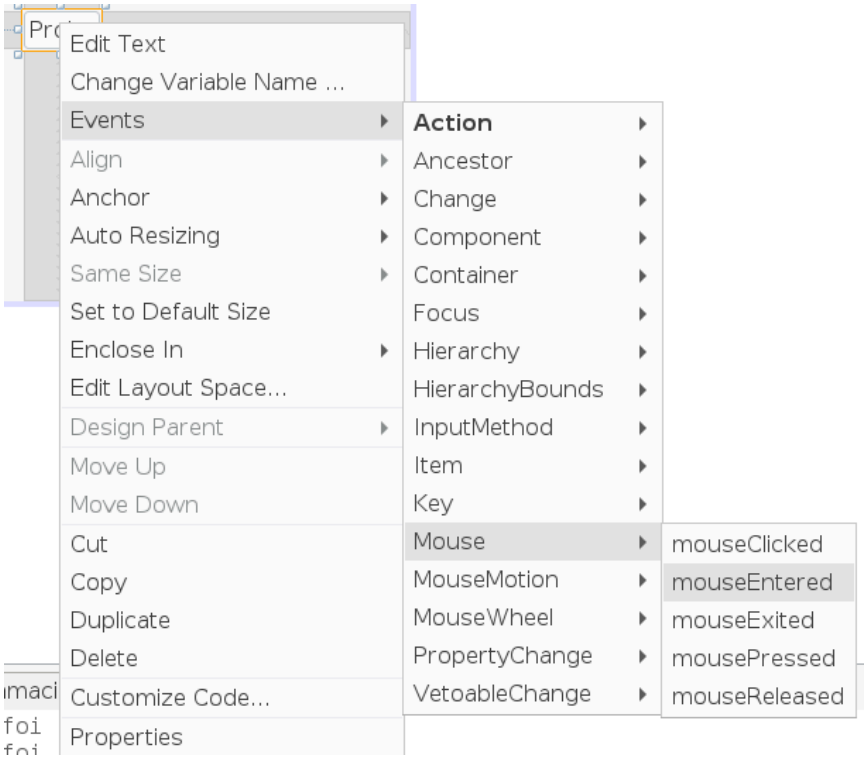
Pero, ¿quen chama ao método anterior?. Para entendelo, hai que despregar o código que xera automaticamente NetBeans no método initComponents()

```
private void initComponents() {

      jToggleButtonProba = new javax.swing.JToggleButton();

      jToggleButtonProba.addActionListener(new java.awt.event.ActionListener() {
         public void actionPerformed(java.awt.event.ActionEvent evt) {
            jToggleButtonProbaActionPerformed(evt);
         }
      });
     ....
}
```

O método **addActionListener** recibe como parámetro unha implementación dunha interface, neste caso **ActionListener**. Esta interface ten un único método que hai que implementar que é **actionPerformed()**. Este último método é o que se executa cando unha persoa pulsa o botón. NetBeans enlaza este último método con **jToggleButtonProbaActionPerformed** que implementamos co código que queiramos.

Ata agora só se fixeron exemplos co clic do botón, pero existen moitos outros eventos que NetBeans xestiona automaticamente.

Por exemplo, para executar certo código cando o rato entre dentro do botón, se queremos facelo automaticamente pulsamos co botón dereito do rato sobre o botón:



Isto crea un método en NetBeans **jToggleButtonProbaMouseEntered** que debemos implementar co código que queiramos executar cando se produza o evento.

Fixarse que NetBeans engade automaticamente o código para rexistrar un escoitador (Listener) no botón.

```
jToggleButtonProba.addMouseListener(new java.awt.event.MouseAdapter() {
        public void mouseEntered(java.awt.event.MouseEvent evt) {
           jToggleButtonProbaMouseEntered(evt);
        }
    });
```

Exercicios:

1. Crea unha aplicación que converta millas a quilómetros. Escolle ti os compoñentes necesarios e a disposición dos mesmos. Se queres ampliar a aplicación, podes tamén engadir a unidade iardas para converter millas e iardas a quilómetros.

2. Crea unha aplicación con dous botóns. Un debe estar deshabilitado inicialmente. Engade o código necesario para que ao pulsar o primeiro botón, este se deshabilite e se habilite o segundo e viceversa.

   Engade un **tooltip** aos botóns con información do seu funcionamento.

3. Crear unha aplicación que teña tres botóns coas etiquetas “Botón 1”, “Botón 2” e “Botón 3” e unha caixa de texto non editable na que se vaian mostrando información das teclas pulsadas. Cando se pulse calquera dos botóns debe **engadirse** á caixa de texto o número do botón pulsado. Ademais, ao pasar co rato por riba do botón, debes cambiar algunha propiedade, por exemplo o tamaño da letra do botón. Cando o rato saia, a propiedade cambiada volverá aos valores por defecto.

   Engade á aplicación un novo botón “Limpar” que baleire o texto da caixa de texto.

Formatear datas

A continuación móstrase un código que exemplifica como se pasa un obxecto Date a String especificando o formato e viceversa.

```
public static void main(String[] args) {
	// recibe como parámetros o formato da data
	SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");

	// crea unha data co día e hora actual
	Date data = new Date();

	// Imprime por pantalla a data co formato dd-MM-yyyy
	// M fai referencia ao mes
	// m fai referencia aos minutos
	System.out.println(sdf.format(data));

	// Pasar un String a Date
	String cadea = "20-10-2020";

	// Pode lanzarse unha excepción se o formato non coincide co de sdf
	try {
	   data = sdf.parse(cadea);
	} catch (ParseException e) {
	   System.out.println("Erro coa data");
	}
	//Imprime día, mes e ano coa hora
	System.out.println(data.toString());

	// Imprime co formato de sdf
	System.out.println(sdf.format(data));
}
```

Referencias

Para a elaboración deste material utilizáronse, entre outros, os recursos que se enumeran a continuación:

- Java® Platform, Standard Edition & Java Development Kit Version 17 API Specification
- The Java™ Tutorials
- Trail: Creating a GUI With Swing (The Java™ Tutorials)
- Lesson: Laying Out Components Within a Container (The Java™ Tutorials > Creating a GUI With Swing)
- http://www3.uji.es/~belfern/Docencia/Presentaciones/ProgramacionAvanzada/
- Java Swing Tutorial - javatpoint
- JTable with Custom Table Model
- Creando un TableModel reutilizable en Swing (parte I) | Java deep Café
- El EDT (Event Dispatch Thread) de Java - ChuWiki
- Java Swing Layout | A Concise Guide to Swing Layout in Java
- The Definitive Guide to Java Swing
- Interfaces de usuario con Java Swing | Jairo García Rincón
- swing => Lo esencial
- The Catalog of Design Patterns