**INTERFACES GRÁFICAS DE USUARIO (GUI)**

La interfaz gráfica de usuario, conocida también como GUI (del inglés graphical user interface), es un programa informático que actúa de interfaz de usuario, utilizando un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz. Su principal uso consiste en proporcionar un entorno visual sencillo para permitir la comunicación con el sistema operativo de una máquina o computador.

[https://es.wikipedia.org/wiki/Interfaz\_gr%C3%A1fica\_de\_usuario].

En el contexto del proceso de interacción persona-computadora, la interfaz gráfica de usuario es el artefacto tecnológico de un sistema interactivo que posibilita, a través del uso y la representación del lenguaje visual, una interacción amigable con un sistema informático.

**INTERFACES EN JAVA.**

En el contexto de JAVA la palabra interface tiene varios significados:

a) Interface: parte visible y pública de una clase que describe qué hace y cómo usarla. La documentación de una clase en el API de Java vendría siendo su interface.

b) Interface: parte visible y pública de un método que describe qué hace y cómo usarlo (signatura del método + instrucciones de uso). La documentación de un método en el API de Java vendría siendo su interface.

c) Interfaz Gráfica de Usuario, interfaz de usuario o GUI (Graphical User Interface): es el entorno de objetos gráficos disponibles para un usuario en el marco de una aplicación o sistema operativo. El sistema operativo MS-Dos se basaba en intérpretes de comando (escritura de instrucciones por consola) pero Windows se basa en una interfaz gráfica de usuario (su entorno de escritorio), Linux en otra y Macintosh en otra.

d) Herramientas para crear Interfaces gráficas de usuario en Java. Hacemos referencia principalmente a los paquetes (packages) del API de Java swing y awt (Abstract Windowing Toolkit). Las clases de estos paquetes permiten crear interfaces gráficas de usuario basadas en ventanas estilo “Windows” para nuestras aplicaciones.

e) Interfaces de Java: son unas entidades abstractas conceptualmente por encima de las clases cuyo concepto vamos a introducir a continuación.

**AWT y Swing**

En el paquete estándar de Java, contamos con tres opciones para crear interfaces gráficas de usuario:

* AWT -Abstract Window Toolkit.
* Swing.
* JavaFX.

**AWT** es una biblioteca pesada -heavy weight-, mientras que **Swing** es una biblioteca ligera -light weight- de componentes. **JavaFX** permite crear intefaces gráficas de usuario tanto para aplicaciones de escritorio como para la web y dispositivos móviles.

La idea de pesada o ligera, en este caso, está relacionada con la dependencia de Java con el sistema operativo, para visualizar y gestionar los elementos de la interface gráfica de usuario.

En el caso de AWT, la creación, visualización y gestión de los elementos gráficos depende del SO. Es el propio SO quien dibuja y gestiona la interacción sobre los elementos.

En el caso de Swing, es Java quien visualiza y gestiona la interacción del usuario sobre los elementos de la interface gráfica.

Existen clases con el mismo cometido tanto en AWT como en Swing. Por ejemplo, en ambas bibliotecas tenemos una clase para crear ventana Frame en el caso de AWT y JFrame en el caso de Swing. Fíjate que si es una clase del paquete Swing su nombre empieza por J.

**SWING**

Swing forma parte del paquete estándar, no hace falta importar ningún fichero adicional en nuestros proyectos.

**JAVAFX**

JavaFX es la apuesta actual de Oracle la comunidad de código abierto en Java para construir interfaces gráficas de usuario multi-dispositivo.

La tendencia actual en la construcción de interfaces gráficas de usuario es mantener la misma calidad, y riqueza de componentes, tanto si el cliente es de escritorio, como un navegador, un dispositivo móvil, la televisión, etcétera.

**CONTENEDORES Y COMPONENTES**

Existen dos elementos básicos para la creación de interfaces gráficas de usuario usando Swing:

Contenedores: Elementos capaces de albergar otros elementos (tanto contenedores como componentes).

Componentes: Elementos que se añaden en los contenedores. Usualmente los componentes tienen aspecto gráfico, como un botón.

Swing proporciona tres tipos de contenedores de alto nivel. Esto significa que, cualquier otro contenedor que no sea de alto nivel, o componente, debe ir en su interior.

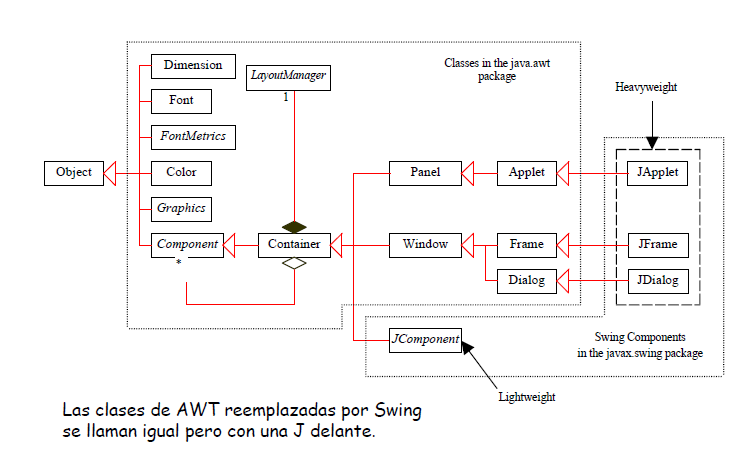
Estos tres contenedores de alto nivel son: **JFrame, JDialog y JApplet**

**JFrame:** Se visualiza como una ventana principal con marco y barra de título, suele utilizarse como la ventana principal de una aplicación.

**JDialog**: Se visualiza como una ventana independiente de la ventana principal para mostrar información, como por ejemplo el contenido de un directorio.

**JApplet:** Permite crear aplicaciones con interface gráfica que se ejecutan en el contexto de un navegador web.

**JERARQUÍA DE CLASES**



**Component**: Superclase de todas las clases de interfaz gráfica.

**Container:** Para agrupar componentes.

**JComponent:** Superclase de todos los componentes de Swing que se dibujan directamente en los lienzos (canvas). Sus subclases son los elementos básicos de la GUI.

**JFrame:** Ventana que no está contenida en otras ventanas.

**JDialog:** Cuadro de diálogo.

**JApplet:** Subclase de Applet para crear applets tipo Swing.

**JPanel:** Contenedor invisible que mantiene componentes de interfaz y que se puede anidar, colocándose en otros paneles o en ventanas. También sirve de lienzo.

**Graphics:** clase abstracta que proporciona contextos gráficos donde dibujar cadenas de texto, líneas y otras formas sencillas.

**Color:** color de los componentes gráficos.

**Font:** aspecto de los caracteres.

**FontMetrics:** clase abstracta para propiedades de las fuentes.

**Categorías de clases:**

Contenedores:

JFrame, JApplet, JWindow, JDialog

Componentes intermedios:

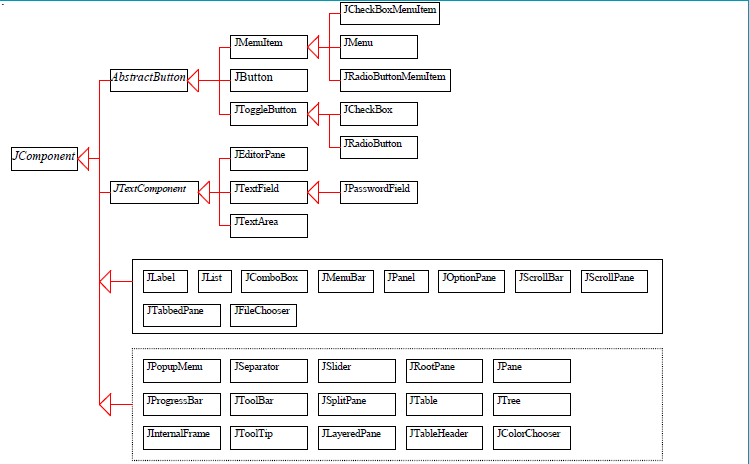
JPanel, JScrollPane

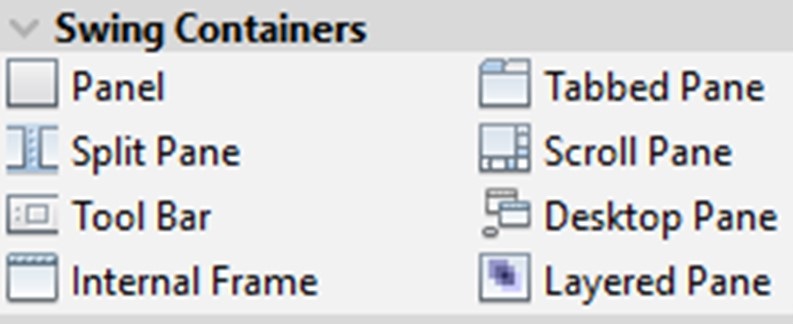
Componentes:

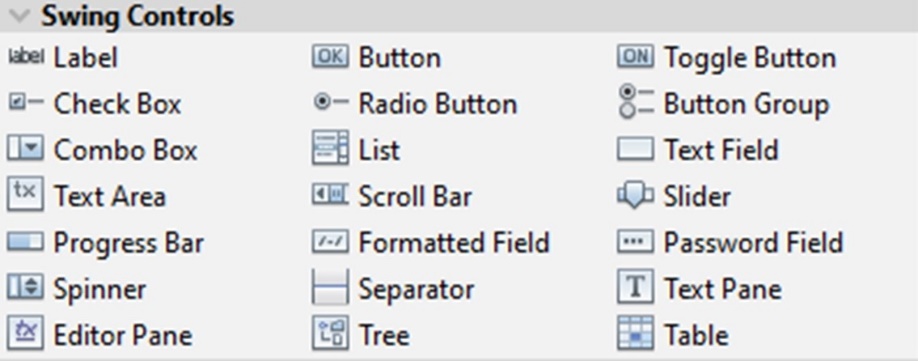
JLabel, JBbutton, JTextField, JTextArea, ...

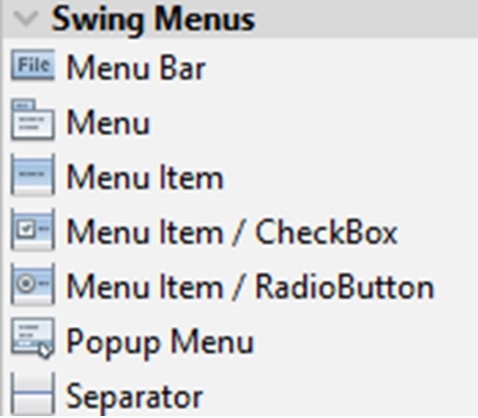
Clases de soporte:

Graphics, Color, Font, ...



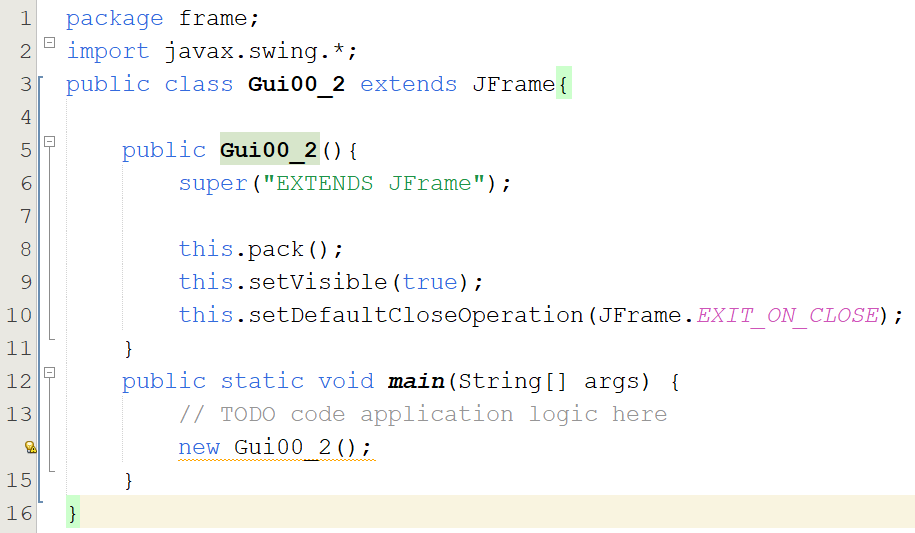




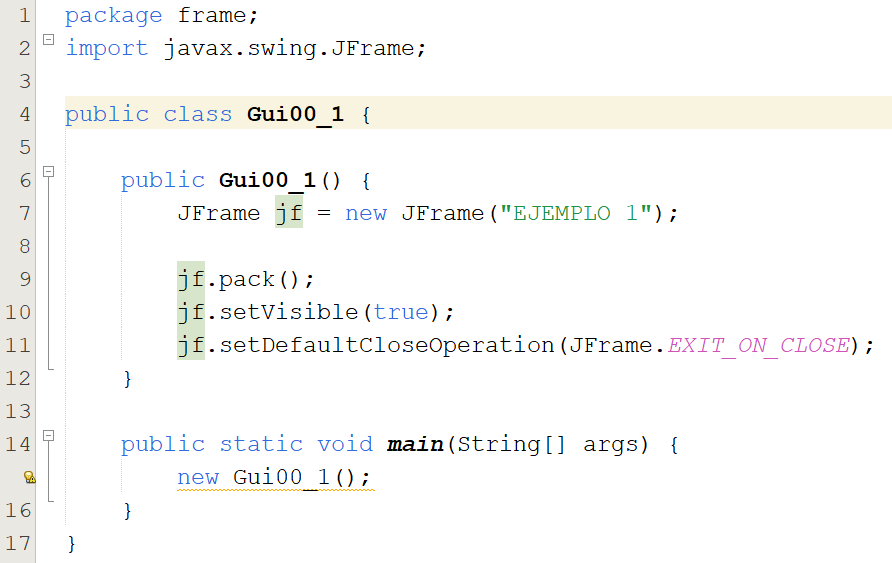


**Iniciando Código**

Crear ventana con Herencia de JFrame:



Crear ventana sin utilizar Herencia



**Agregar elementos**

Modo 1:

1. Obtenemos el panel de contenido del frame:

Container panel = this.getContentPane();

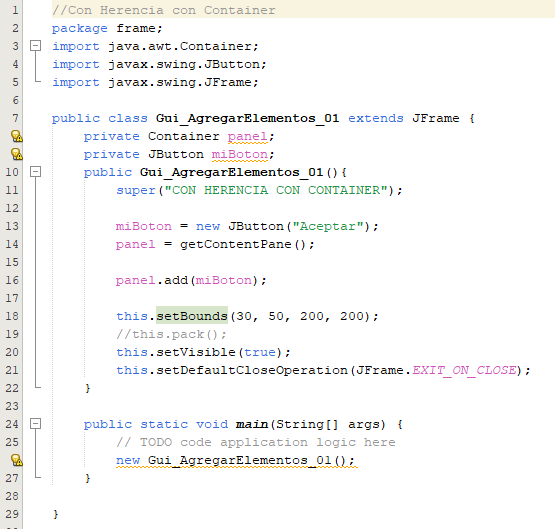
2. Añadimos componentes a dicho panel:

panel.add(unComponente);

Modo 2:

A partir de 1.5 también se puede hacer directamente sobre el JFrame add(unComponente);

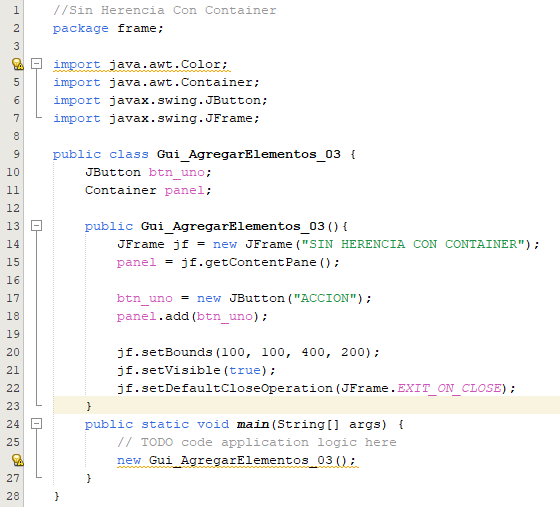
Con herencia de JFrame y con Container



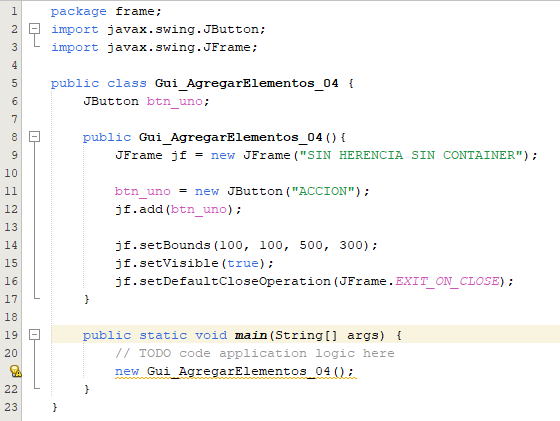
Con herencia de JFrame y sin Container



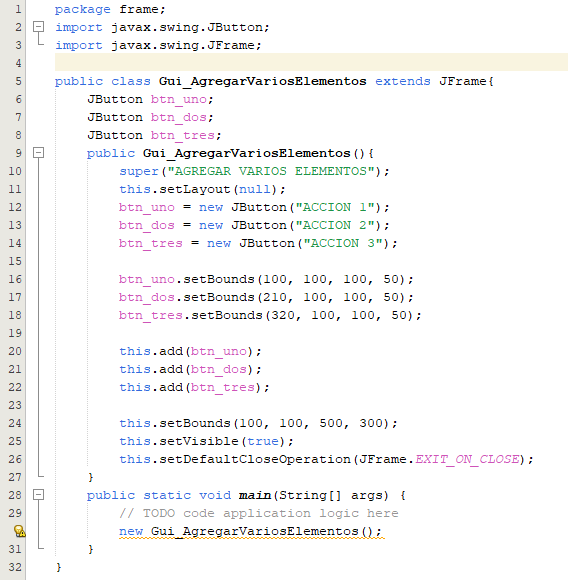
Sin herencia de JFrame y Con Container



Sin Herencia y sin Container



NOTA: Agregar varios componentes:



**EL MODELO DE EVENTOS DE JAVA**

En Java, cada evento está representado por un objeto que es una subclase de la clase EventObject en el paquete java.util. Cada subclase de EventObject representa un tipo de evento particular. Por ejemplo:

1. MouseEvent, para representar acciones del ratón: mover, arrastrar, hacer clic en un botón del ratón, etc.
2. KeyEvent, para representar acciones del teclado, esto es, pulsar teclas.
3. ActionEvent, para representar una acción del usuario en la interfaz, por ejemplo, pulsar un botón en la pantalla.

No obstante, en el modelo de eventos de Java, los EventObject no realizan acciones por sí mismos, sino que esos eventos son enviados a otro objeto encargado de responder a un tipo de evento en particular. Estos objetos son los que conocemos como listeners o "escuchadores", y existen diferentes listeners que "escuchan" a los diferentes eventos, tales como los mouse listeners, key listeners o action listeners.

Los listener no se implementan como clases en java, sino como interfaces. Una interface es una colección de métodos que definen un comportamiento en particular. De este modo, cualquier clase que suministre información para dichos métodos puede declarar que implementa a dicho interface.

La gran ventaja de implementar interfaces frente a heredar de otras clases es que una misma clase puede implementar varios interfaces simultáneamente pero solo puede heredar de una.

Los listeners que usaremos estarán en el paquete java.awt.event (que debemos importar antes de implementarlos) y nos podemos encontrar los siguientes **LISTENER**:

**Nombre**: ActionListener

**Descripción**: Se produce al hacer click en un componente, también si se pulsa Enter teniendo el foco en el componente.

**Método**: public void actionPerformed(ActionEvent e)

Eventos:

JButton: click o pulsar Enter con el foco activado en él.

JList: doble click en un elemento de la lista.

JMenuItem: selecciona una opción del menú.

JTextField: al pulsar Enter con el foco activado.



**Nombre**: KeyListener

**Descripción**: Se produce al pulsar una tecla. según el método cambiara la forma de pulsar la tecla.

**Métodos**:

public void keyTyped(KeyEvent e)

public void keyPressed(KeyEvent e)

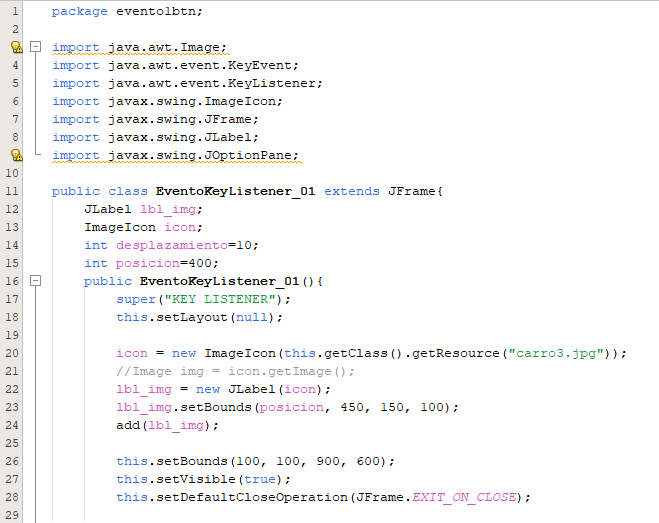
public void keyReleased(KeyEvent e)

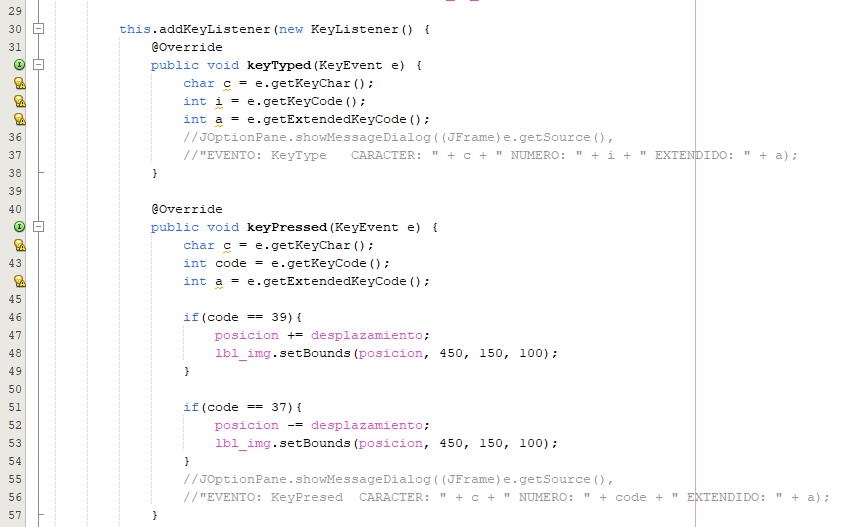
**Eventos**: Cuando pulsamos una tecla, según el listener:

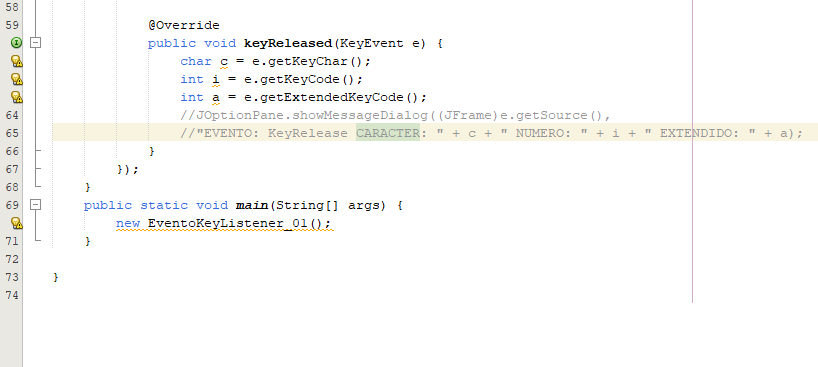
keyTyped: al pulsar y soltar la tecla.

keyPressed : al pulsar la tecla.

keyReleased : al soltar la tecla.







**Nombre**: FocusListener

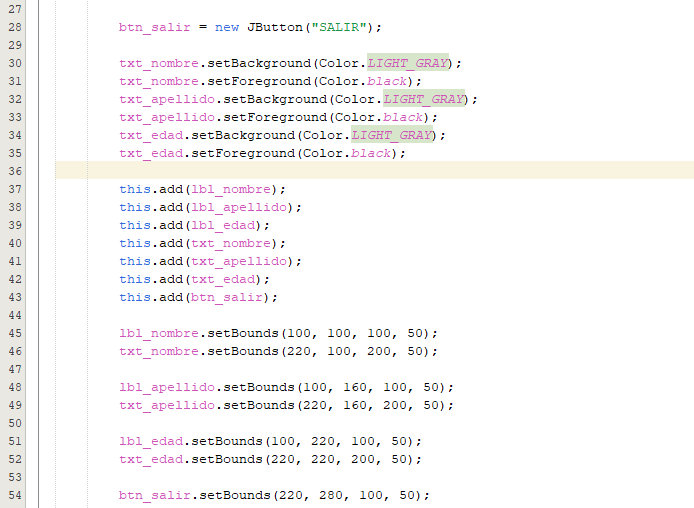
**Descripción**: Se produce cuando un componente gana o pierde el foco, es decir, que esta seleccionado.

**Métodos**: Recibir o perder el foco.

public void focusGained(FocusEvent e)

public void focusLost(FocusEvent e)









**Nombre**: MouseListener

**Descripción**: Se produce cuando realizamos una acción con el ratón.

**Métodos**:

public void mouseClicked(MouseEvent e)

public void mouseEntered(MouseEvent e)

public void mouseExited(MouseEvent e)

public void mousePressed(MouseEvent e)

public void mouseReleased(MouseEvent e)

**Eventos**: Según el listener:

mouseClicked: presionar y soltar.

mouseEntered: entrar en un componente con el puntero.

mouseExited: salir de un componente con el puntero

mousePressed: presionar el botón.

mouseReleased: soltar el botón.

**Nombre**: MouseMotionListener

**Descripción**: Se produce con el movimiento del mouse.

**Métodos**:

public void mouseDragged(MouseEvent e)

public void mouseMoved(MouseEvent e)

**Eventos**: Según el listener:

mouseDragged: click y arrastrar un componente.

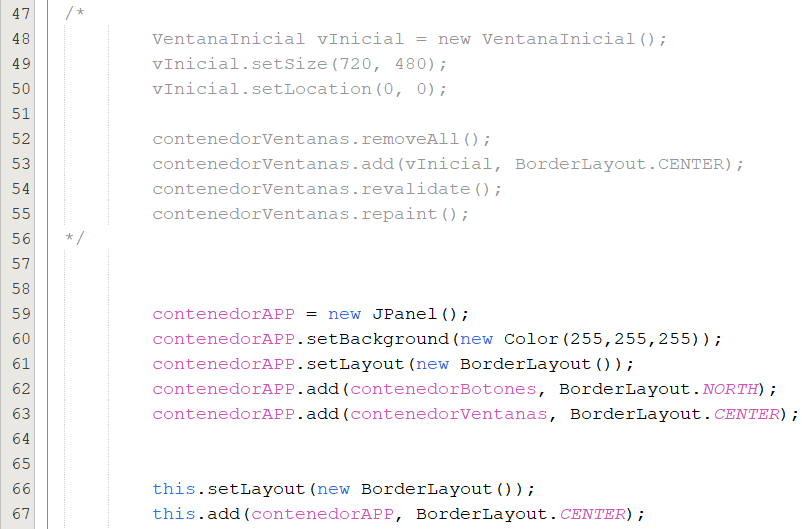
mouseMoved: al mover el puntero sobre un elemento.

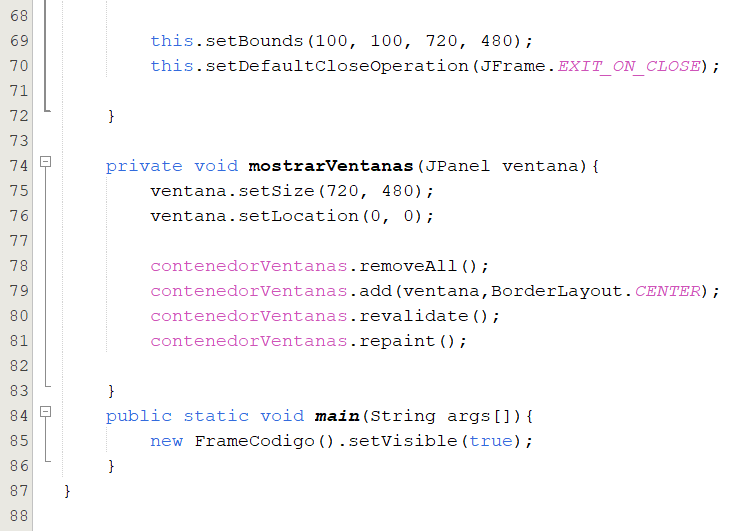
Practica “Crear un JFrame que contenga varias ventanas”

VENTANA PRINCIPAL (JFrame)

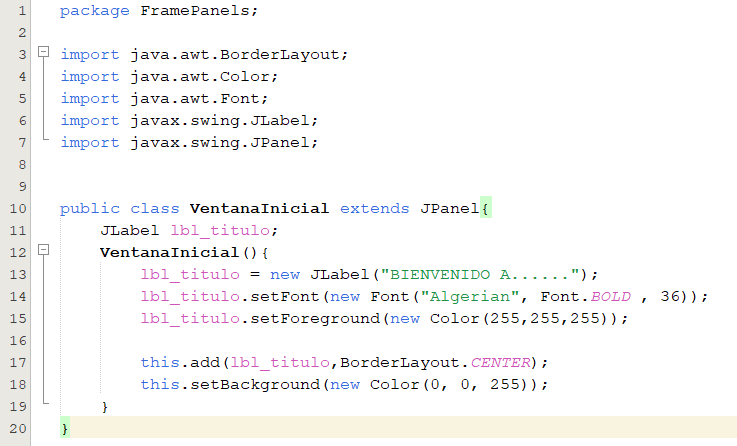


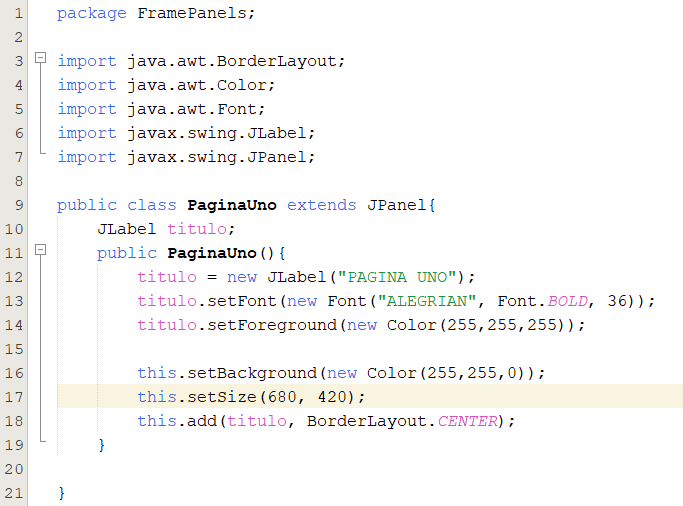






VENTANAS SECUNDARIAS (JPanel)







**LAYOUT**

En java, cuando hacemos ventanas, la clase que decide cómo se reparten los componentes dentro de la ventana se llama **Layout**. Esta clase es la que decide en qué posición van los botones y demás componentes, si van alineados, en forma de matriz, cuáles se hacen grandes al agrandar la ventana, etc. Otra cosa importante que decide el **Layout** es qué tamaño es el ideal para la ventana en función de los componentes que contiene.

Con un layout adecuado, el método pack() de la ventana hará que esta coja el tamaño necesario para que se vea todo lo que tiene dentro.

ventana.pack();

Las ventanas vienen con un Layout por defecto. En java hay varios layouts disponbiles y podemos cambiar el de defecto por el que queramos.

Puede existir infinidad de Layout incluso usted mismo puede diseñar su propio Layou, lo cual no es recomendado ya que existen los suficientes diseños para que usted resuelva cualquier necesidad de disposición de componentes dentro d una ventana.

A continuación, listo algunos de los Layout más importantes que existen.

**LAYOUT NULL**

Es uno de los Layouts más utilizados por la gente que empieza, por ser el más sencillo, consiste en NO usar layout. Somos nosotros desde el código los que decimos a cada componente en qué posición va y qué tamaño ocupa.

ventana.setLayout(null); // Eliminamos el layout

ventana.add (boton); // Añadimos el componente previamente creado

boton.setBounds (10,10,40,20); // Establecemos la posición del componenete 10,10 con ancho 40 pixels y un alto 20 de pixeles.

Esto, aunque sencillo, no es recomendable. Si estiramos la ventana los componentes seguirán en su sitio, no se estirarán con la ventana. Si cambiamos de sistema operativo, la resolución de pantalla o la fuente de letra, tenemos casi asegurado que no se vean bien las cosas: etiquetas cortadas, letras que no caben, etc.

Además, al no haber layout, la ventana no tiene tamaño adecuado. Deberemos dárselo nosotros con un ventana.setSize(...). Y si hacemos que sea un JPanel el que no tiene layout, para que este tenga un tamaño puede que incluso haga falta llamar a panel.setPreferredSize(...) o incluso en algunos casos, sobreescrbiendo el método panel.getPreferredSize().

El tiempo que ahorramos no aprendiendo cómo funcionan los Layouts, lo perderemos echando cuentas con los pixels, para conseguir las cosas donde queremos, sólo para un tipo de letra y un tamaño fijo.

**FlowLayout**

Es el más simple y el que se utiliza por defecto en los Paneles si no se fuerza el uso de alguno de los otros. Los Componentes añadidos a un Panel con **FlowLayout** se encadenan en forma de lista. La cadena es horizontal, de izquierda a derecha, y se puede seleccionar el espaciado entre cada Componente, también se puede configurar para que los componentes queden justificados a la izquierda, a la derecha o centrados.

Por ejemplo, podemos poner un grupo de botones con la composición por defecto que proporciona FlowLayout:

**BorderLayout**

**GridBagLayout**

El GridBagLayout es uno de los Layout de java más potentes, pero también es el que más cuesta entender.

Cualquier contenedor java tiene una clase layout por defecto, pero la podemos cambiar con el método setLayout().

Para ventanas complejas Java nos proporciona un layout bastante potente, aunque bastante complejo de usar cuando no se comprende bien: el GridBagLayout. Este layout será suficiente para hacer casi cualquier distribución de componentes en una ventana, por compleja que parezca, pero debemos entender bien su funcionamiento para que el resultado final se parezca a lo que queremos.

El primer paso es dibujar la ventana que queremos. Es importante dibujarla más o menos estirada, para tener claro si los componentes deben hacerse grandes o no.



Por ejemplo, en la figura anterior, los botones deben mantener su tamaño, aunque estiremos la ventana, sin embargo, el área de texto debe crecer con la ventana y el campo de texto sólo debe crecer horizontalmente.

Luego, sobre el dibujo, hay que trazar unas líneas horizontales y verticales, para tratar de delimitar una rejilla en la que irán colocados los botones.

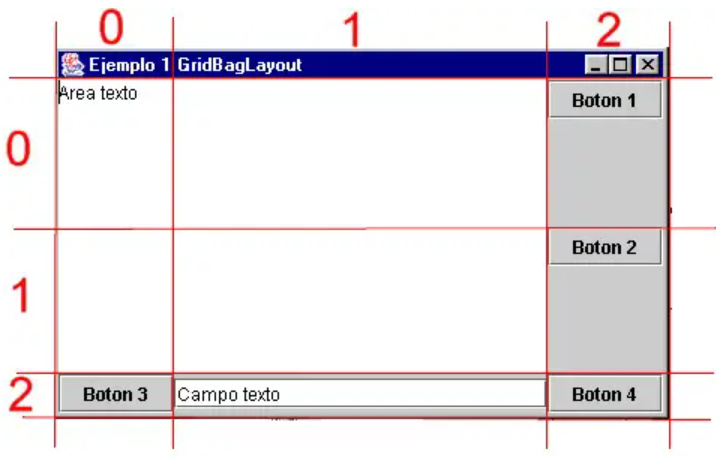
Para trazar esta rejilla y que luego el código no nos salga muy complicado conviene tener en cuenta lo siguiente:

* Debemos tratar de meter los componentes en las celdas.
* Cada componente debe ocupar una o más celdas.
* Dos componentes no pueden ocupar la misma celda.
* No es necesario que las celdas de la rejilla sean todas del mismo tamaño ni es necesario que un componente ocupe una celda completa.

Para que no se lie el código, conviene:

* Que el componente ocupe toda la celda,
* O bien que esté centrado en la misma,
* O bien pegado al centro de uno de los bordes de la celda.
* O bien a una esquina de la misma.
* Vaya, que no pongamos una celda enorme con un componente dentro a tres cuartos de la parte derecha y medio séptimo de la parte de arriba.

Un ejemplo de rejilla para nuestra ventana puede ser la de la siguiente figura:



Vamos con las líneas verticales:

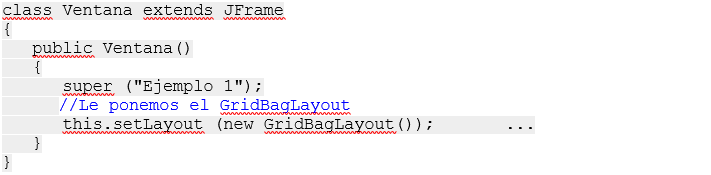
* La primera en el borde izquierdo de la ventana.
* La segunda entre el boton 3 y el campo de texto.
* La tercera entre el campo de texto y el boton 4.
* La última en el borde derecho de la ventana.

Estas líneas son casi obligadas, puesto que el boton 3, el campo de texto y el boton 4 nos las limitan. No importa que una de ellas pase por en medio del área de texto, simplemente haremos que esta ocupe dos columnas. Estas líneas nos limitan tres columnas que hemos numerado como 0, 1 y 2.

Vamos ahora con las líneas horizontales.

* La primera por la parte de arriba de la ventana
* La segunda puede pasar por cualquier sitio entre el boton 1 y el boton 2.
* La tercera pasa justo por debajo del área de texto y por encima de boton 3, campo de texto y boton 4
* La última por el borde inferior de la ventana.

Resumiendo, nuestra área de texto ocupa cuatro celdas, los botones una cada uno y el campo de texto otra. Boton 1 y 2 no ocupan toda la celda, pero están en la parte superior de la misma.



Ahora debemos empezar a añadir componentes. Tenemos por un lado el método add()con un sólo parámetro.

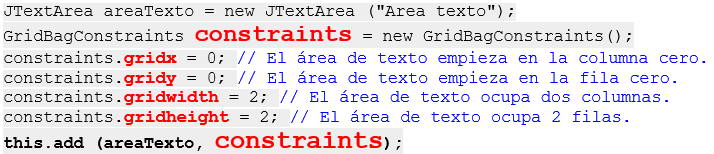


Esta llamada añade el área de texto, dejando al GridBagLayout total libertad para decidir dónde y cómo ponerlo. No es lo que queremos. Necesitamos el método add() que admite un segundo parámetro. Este segundo parámetro le indica al layout dónde y cómo colocar el componente. El segundo parámetro es un Object, lo que quiere decir que podríamos meter cualquier cosa. Evidentemente esto no es así, debemos meter algo que el layout entienda. Cada layout entiende su propio segundo parámetro. El GridBagLayout admite como segundo parámetro un **GridBagConstraints.**

**GridBagConstraints** en una clase en cuyos atributos se guarda información de cómo y dónde añadir el componente:

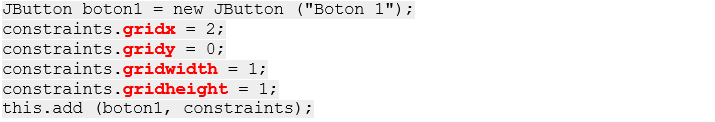
* **GridBagConstraints.gridx** nos dice la posición x del componente, es decir, el número de columna en la que está el componente, siendo la columna 0 la primera columna de la izquierda. Si el componente ocupa varias columnas (como nuestra área de texto), debemos indicar la columna en la que está la esquina superior izquierda del componente.
* **GridBagConstraints.gridy** nos dice la posición y del componente, es decir, el número de fila en la que está el componente, siendo la fila 0 la primera fila de la parte de arriba. Si el componente ocupa varias filas (como nuestra área de texto), debemos indicar la fila en la que está la esquina superior izquierda del componente.
* **GridBagConstraints.gridwidth** nos dice cuántas celdas en horizontal debe ocupar el componente. El ancho del componente.
* **GridBagConstraints.gridheight** nos dice cuántas celdas en vertical debe ocupar el componente. El alto del componente.

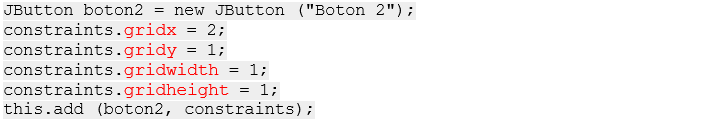
Con esto podemos añadir nuestra área de texto

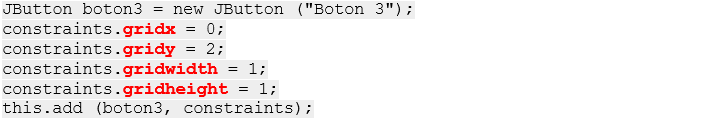


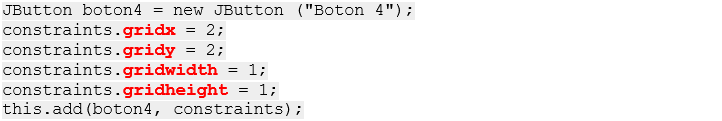
Hay un detalle importante a tener en cuenta. Sólo hemos hecho un new de GridBagConstraints, por lo que todos ellos comparten la misma instancia de ese objeto. Hay que tener en cuenta que, si cambiamos uno de los atributos para un componente, debemos restaurarlo luego para el siguiente. Por ejemplo, para el área de texto pusimos gridwidth a 2. Para el siguiente componente (el botón1), debemos volver a poner este valor a 1.

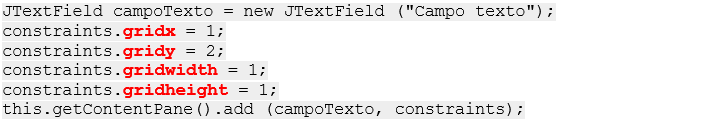
Por ello es aconsejable poner siempre todos los atributos. Otra opción es hacer un new GridBagConstraints para cada componente (claramente menos eficiente) o finalmente la que he seguido en el código de ejemplo, ser muy cuidadoso e ir restaurando valores según se van cambiando.











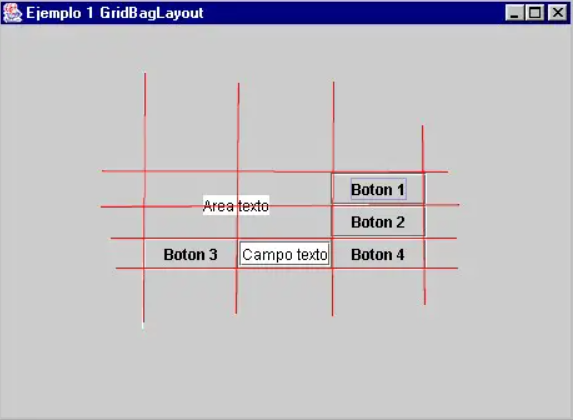
\*NOTA LA ULTIMA LINEA DEBE SER: this.add(campoTexto, constraints)

Si ejecutamos lo que tenemos hasta ahora y estiramos un poco la ventana resultante nos sale lo de la siguiente figura:



**Estirar las filas y las columnas**

Lo que ha pasado es que sólo le hemos dicho al GridBagLayout dónde colocar los componentes y eso lo ha hecho bien. Cada componente está donde debe. Lo que pasa es que no le hemos dicho nada de cómo estirar las filas y columnas, así que ha hecho lo que hace por defecto: Cada fila y columna es del tamaño mínimo necesario para albergar sus componentes y están centradas dentro de la ventana. En la siguiente figura vemos como están nuestras filas y columnas:



Las filas son de la altura justa para contener a los botones. Las columnas también. El área de texto que ocupa cuatro celdas se ha situado en el medio de ellas y le sobra un montón de espacio.

El siguiente paso a dar consiste en decir cómo se deben estirar las filas y las columnas. Si comparamos con la figura en las que marcamos inicialmente las filas y columnas, vemos que las dos primeras filas deben ser más anchas, la tercera fila ya está bien de ancho. En cuanto a las columnas, la segunda es la que debe estirarse, la primera y tercera están bien.

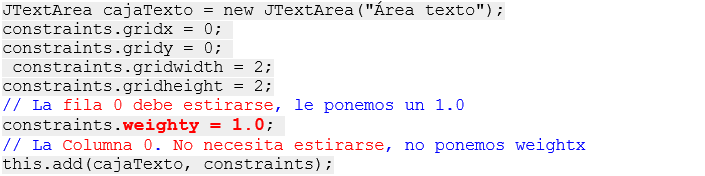
Para estirar filas y columnas, dentro del GridBagConstraints tenemos los atributos **weigthx y weigthy**. El primero indica cómo estirar las columnas. El segundo las filas.

* A estos atributos debemos darles el valor 0.0 (que es el valor por defecto) si no queremos que la fila o columna se estire. Este es el caso para la primera y tercera columna, así como para la tercera fila.
* Debemos dar el valor 1.0 a las filas o columnas que queremos que se estiren hasta completar toda la ventana. Es el caso de las dos primeras filas y de la columna del medio.

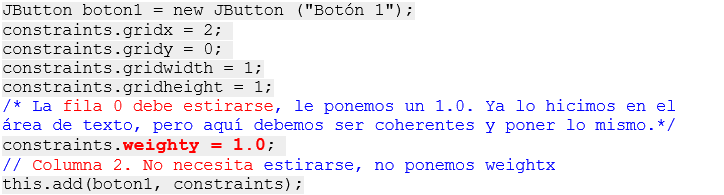
**Hay dos detalles con estos campos:**

* El primero es que widthy afecta a una fila completa, no solo a un componente. Por ello, cada vez que añadamos un componente a esa fila, debemos dar a widthy el mismo valor (o 0.0 o 1.0). Lo mismo pasa con widthx y las columnas.
* El segundo detalle es el que comentamos antes, sólo estamos usando una instancia de GridBagConstraints, así que después de poner uno de estos campos a 1.0 y añadir un componente, debemos volver a ponerlo a 0.0 para el siguiente (suponiendo que sea eso lo que necesita).

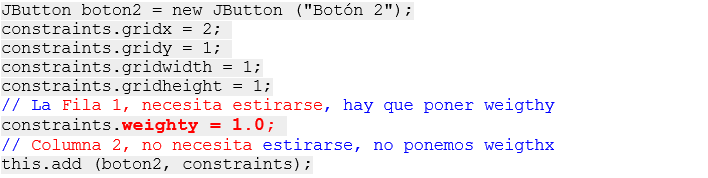
El código ahora, añadiendo estos dos campos, quedaría así:



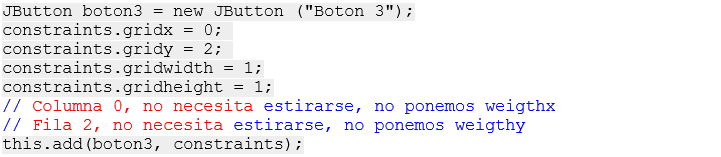


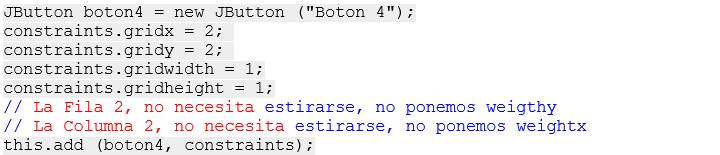


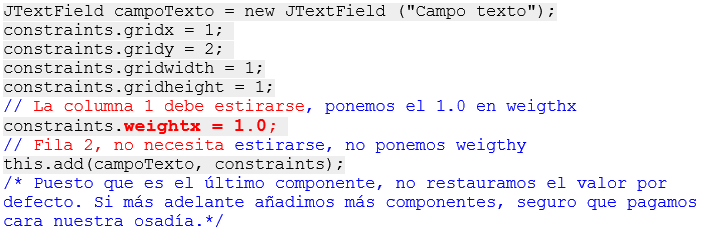














¿Qué es lo que ha pasado ahora? Como dice un viejo dicho de informática, la computadora se empeña en hacer lo que le decimos y no lo que queremos. **Le hemos dicho que estire las filas y columnas** **y es lo que ha hecho**, **pero como no le hemos dicho nada sobre los componentes**, **no los ha estirado en absoluto**.

**Estirar los componentes**

Si pintamos sobre la ventana las rayas que delimitan filas y columnas, vemos que efectivamente se han ensanchado las que hemos dicho (filas y columnas), pero los componentes siguen con su tamaño original y en el centro de las celdas que tienen asignadas. El área de texto sigue igual, en el centro de sus cuatro celdas, los botones 1 y 2 se han centrado en sus celdas y lo mismo le pasa al campo de texto.



Lo siguiente que tenemos que hacer es decir **qué componentes deben estirarse** (el área de texto y el campo de texto para nuestro ejemplo). Para aquellos componentes que no deben estirarse, podemos **indicar en qué parte de la celda queremos que se sitúen** (caso del boton 1 y boton 2).

Para hacer que un componente se estire, tenemos el atributo **fill** del GridBagConstraints. Este puede tomar los siguientes valores:

GridBagConstraints.**NONE** para que no se estire en ningún sentido, es la opción por defecto.

GridBagConstraints.**VERTICAL** para que se estire sólo en vertical

GridBagConstraints.**HORIZONTAL** para que se estire sólo en horizontal.

GridBagConstraints.**BOTH** para que se estire en ambas dimensiones

Si el componente no se estira en alguna dirección, podemos decirle por medio de **anchor** en GridBagConstraints qué posición queremos que ocupe.

Las posibilidades son:

GridBagConstraints.**CENTER** para que el componente ocupe el centro de la celda. Es la opción por defecto.

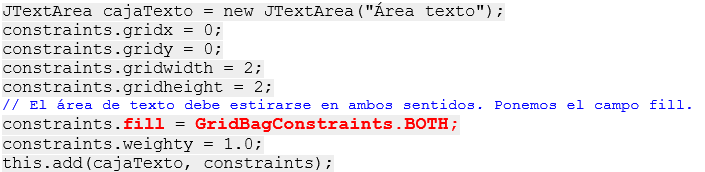
GridBagConstraints.**NORTH** para que se pegue a la parte superior, centrado en la misma.

GridBagConstraints.**NORTHEAST** para que se pegue a la esquina superior derecha.

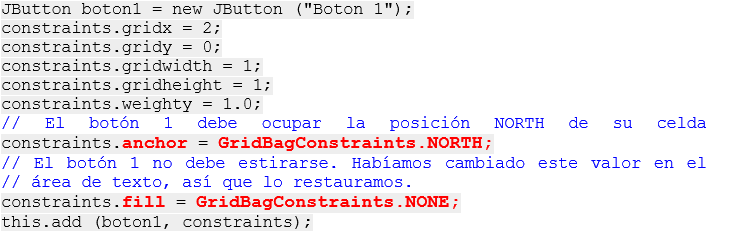
... (qué pereza me da escribir todos los puntos cardinales ...)

GridBagConstraints.**WEST** para que se pegue al lado izquierdo, centrado en ese lado.

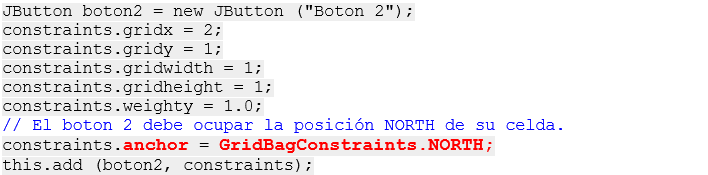
GridBagConstraints.**NORTHWEST** para que se pegue a la esquina superior izquierda.



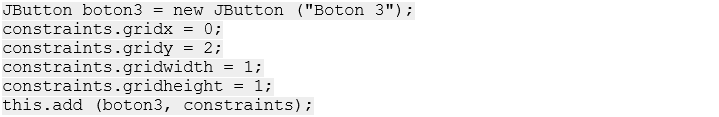


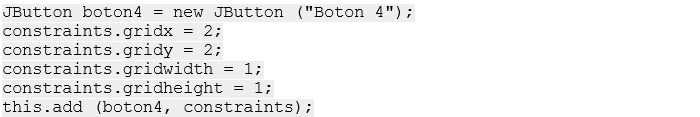


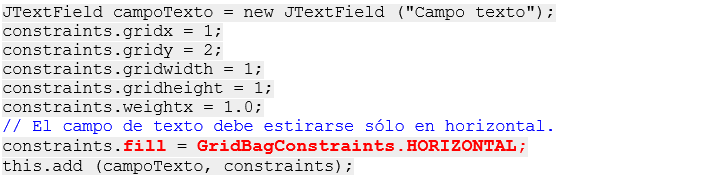












Algunos comentarios extras

Con esto ya deberíamos ser capaces de situar los componentes en nuestra ventana en la mayoría de las ocasiones. El GridBagConstraints nos permite más cosas, de las que señalo aquí un par de ellas, por si alguien quiere indagar un poco más.

Los atributos weightx y weighty, como hemos visto, hace que se estiren las filas y columnas. Hemos puesto 0.0 para las que no se estiran y 1.0 para las que sí. En realidad cada fila o columna se estira en proporción al valor de este campo respecto a los demás. Por ejemplo, si a la columna 1 le ponemos weightx=1.0 y a la 2 le ponemos weightx=0.5, ambas se estirarían , pero la primera el doble que la segunda.

Hay campos insets, ipadx e ipady que permiten fijar márgenes entre los componentes, de forma que no queden pegados entre sí.

**RESUMEN:**

El GridBagLayout es uno de los Layout de java más potentes, pero también es el que más cuesta entender.

Se debe dividir la ventana en una cuadricula (filas y columnas)

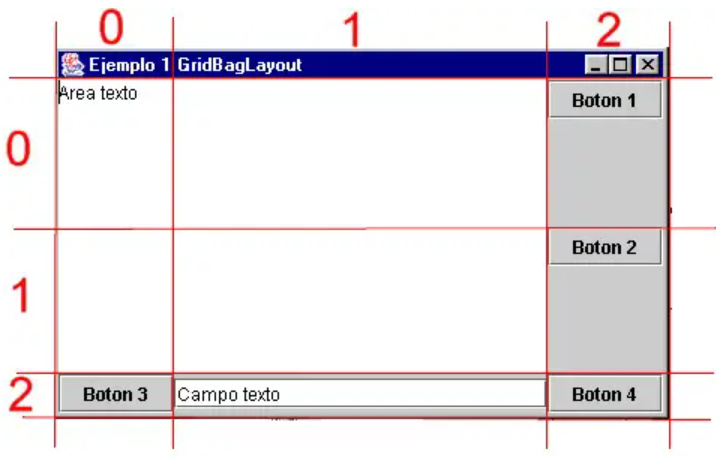
Para trazar esta rejilla y que luego el código no nos salga muy complicado conviene tener en cuenta lo siguiente:

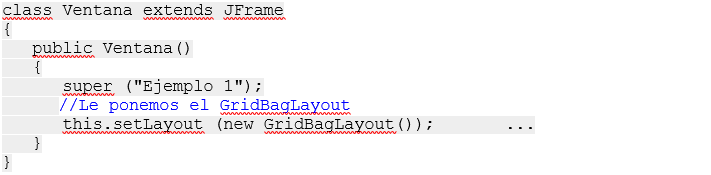
* Debemos tratar de meter los componentes en las celdas.
* Cada componente debe ocupar una o más celdas.
* Dos componentes no pueden ocupar la misma celda.
* No es necesario que las celdas de la rejilla sean todas del mismo tamaño ni es necesario que un componente ocupe una celda completa.

Para que no se lie el código, conviene:

* Que el componente ocupe toda la celda,
* O bien que esté centrado en la misma,
* O bien pegado al centro de uno de los bordes de la celda.
* O bien a una esquina de la misma.
* Vaya, que no pongamos una celda enorme con un componente dentro a tres cuartos de la parte derecha y medio séptimo de la parte de arriba.

Un ejemplo de rejilla para nuestra ventana puede ser la de la siguiente figura:





Ahora debemos empezar a añadir componentes. Tenemos por un lado el método add()con un sólo parámetro.

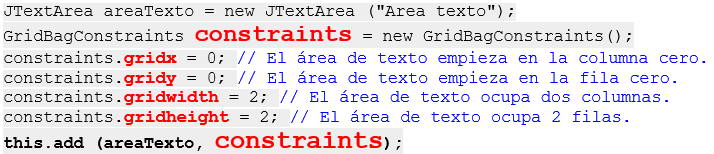


El GridBagLayout admite como segundo parámetro un **GridBagConstraints.**

**GridBagConstraints** en una clase en cuyos atributos se guarda información de cómo y dónde añadir el componente:

* **GridBagConstraints.gridx** nos dice la posición x del componente, es decir, el número de columna en la que está el componente
* **GridBagConstraints.gridy** nos dice la posición y del componente, es decir, el número de fila en la que está el componente
* **GridBagConstraints.gridwidth** nos dice cuántas celdas en horizontal debe ocupar el componente.
* **GridBagConstraints.gridheight** nos dice cuántas celdas en vertical debe ocupar el componente.

**Ejemplo**

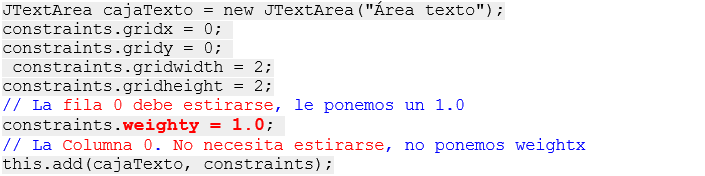


Las filas son de la altura justa para contener a los botones. Las columnas también.

El siguiente paso a dar consiste en decir cómo se deben estirar las filas y las columnas.

Para estirar filas y columnas, dentro del GridBagConstraints tenemos los atributos **weigthx y weigthy**. El primero indica cómo estirar las columnas. El segundo las filas.

* A estos atributos debemos darles el valor 0.0 (que es el valor por defecto) si no queremos que la fila o columna se estire.
* Debemos dar el valor 1.0 a las filas o columnas que queremos que se estiren hasta completar toda la ventana.



¿Qué es lo que ha pasado ahora? Como dice un viejo dicho de informática, la computadora se empeña en hacer lo que le decimos y no lo que queremos. **Le hemos dicho que estire las filas y columnas** **y es lo que ha hecho**, **pero como no le hemos dicho nada sobre los componentes**, **no los ha estirado en absoluto**.

Lo siguiente que tenemos que hacer es decir **qué componentes deben estirarse** (el área de texto y el campo de texto para nuestro ejemplo).

Para aquellos componentes que no deben estirarse, podemos **indicar en qué parte de la celda queremos que se sitúen** (caso del boton 1 y boton 2).

Para hacer que un componente se estire, tenemos el atributo **fill** del GridBagConstraints. Este puede tomar los siguientes valores:

GridBagConstraints.**NONE** para que no se estire en ningún sentido, es la opción por defecto.

GridBagConstraints.**VERTICAL** para que se estire sólo en vertical

GridBagConstraints.**HORIZONTAL** para que se estire sólo en horizontal.

GridBagConstraints.**BOTH** para que se estire en ambas dimensiones

Si el componente no se estira en alguna dirección, podemos decirle por medio de **anchor** en GridBagConstraints qué posición queremos que ocupe.

Las posibilidades son:

GridBagConstraints.**CENTER** para que el componente ocupe el centro de la celda. Es la opción por defecto.

GridBagConstraints.**NORTH** para que se pegue a la parte superior, centrado en la misma.

GridBagConstraints.**EAST** para que se pegue al lado derecho, centrado en la misma.

GridBagConstraints.**WEST** para que se pegue al lado izquierdo, centrado en ese lado.

GridBagConstraints.**SOUTH** para que se pegue a la parte inferior, centrado en ese lado.

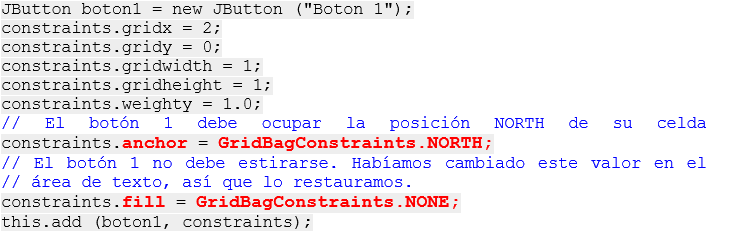
GridBagConstraints.**NORTHEAST** para que se pegue a la esquina superior derecha.

GridBagConstraints.**NORTHWEST** para que se pegue a la esquina superior izquierda.

GridBagConstraints.**SOUTHEAST** para que se pegue a la esquina inferior derecha.

GridBagConstraints.**SOUTHWEST** para que se pegue a la esquina inferior izquierda.

Ejemplo:



**Algunos comentarios extras**

Con esto ya deberíamos ser capaces de situar los componentes en nuestra ventana en la mayoría de las ocasiones. El GridBagConstraints nos permite más cosas, de las que señalo aquí un par de ellas, por si alguien quiere indagar un poco más.

Los atributos weightx y weighty, como hemos visto, hace que se estiren las filas y columnas. Hemos puesto 0.0 para las que no se estiran y 1.0 para las que sí. En realidad cada fila o columna se estira en proporción al valor de este campo respecto a los demás. Por ejemplo, si a la columna 1 le ponemos weightx=1.0 y a la 2 le ponemos weightx=0.5, ambas se estirarían , pero la primera el doble que la segunda.

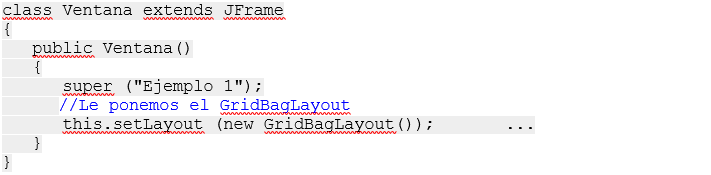
Hay atributos insets, ipadx e ipady que permiten fijar márgenes entre los componentes, de forma que no queden pegados entre sí.

**Resumen del Resumen**

Establecer Layout:



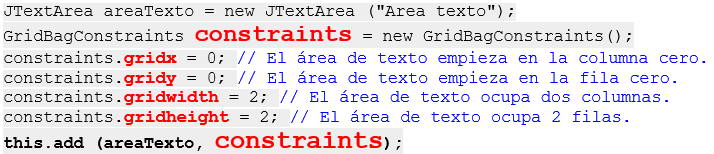
O, bien:



Crear Objeto GridBagConstarint:



Configurar el objeto para cada uno de los componentes que vamos a agregar a la interfaz gráfica.



**Atributos principales para configurar:**

|  |  |
| --- | --- |
| constraints.gridx = 2; | //Posicionar el componente: Indicar columna |
| constraints.gridy = 0; | //Posicionar el componente: Indicar fila |
| constraints.gridwidth = 1; | //Indicar cuantas celdas horizontales va a ocupar. |
| constraints.gridheight = 1; | //Indicar cuantas celdas hverticales va a ocupar. |
| constraints.weighty = 1.0; | //Indicar si la celda se va a hacer más ancha |
| constraints.weightx = 1.0; | //Indicar si la celda se va a hacer más alta |
| constraints.fill = GridBagConstraints.NONE; | //Indicar si el componente se va a estirar. |
| constraints.anchor = GridBagConstraints.NORTH; | //Indicar la posición del componente dentro de la celda. |



Los valores de los atributos que se queden en el objeto constraint se aplicaran a todos los componentes que se quieran agregar.

**COMPONENTE JTABLE**

**JTABLE**

Existen 4 formas para elaborar una JTable en tu GUI.

En realidad, tienes cuatro posibilidades:

* Trabajar directamente con JTable
* Implementando la Interfaz TableModel
* Heredar de la clase AbstractTableModel
* Heredar de la DefaultTableModel.

La primera está bien si la tabla es fija y no vas a requerir cambio en los datos. Fijate que JTable apenas tiene métodos para borrar filas, añadir columnas, etc, etc.

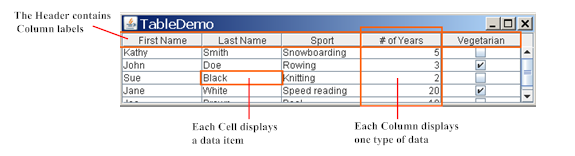
La de TableModel es la más costosa de programar, pero la que da más versatilidad. TableModel es una interface y tienes que escribir tú todos los métodos para añadir, borrar y modificar datos, además de implementar un mecanismo de suscripción a cambios en los datos http://www.chuidiang.com/java/tablas/tablamodelo/tablamodelo.html. Puedes hacerlo como quieras y de la forma que mejor te venga, pero tienes que hacerlo tú todo.

La de DefaultTableModel es la más sencilla. No tienes que escribir nada de código. DefaultTableModel tiene todos los métodos necesarios para añadir, modificar y borrar celdas. http://www.chuidiang.com/chuwiki/index.php?title=JTable

AbstractTableModel es una intermedia entre las dos anteriores. Tiene implementado y por tanto no tienes que hacer tú todos los mecanismos de "suscripción" a cambios de datos. Únicamente tienes que codificar: añadir, borrar y modificar datos.

**How to Use Tables**

With the JTable class you can display tables of data, optionally allowing(permitiendo) the user to edit the data. JTable does not contain or cache data; it is simply a view of your data. Here is a picture of a typical table displayed within a scroll pane:



**MODELO PARA IMPLEMENTAR UNA JTable utilizando AbstractTableModel**

AbstractTableModel

ModeloPersonas

JTablePersonas

Persona

- List<Tipo> listaDatos;

- String[] nombresColumnas

+ Datos()

+ obtenerDatos()

+ getPersonas():List<Persona>

+ getColumnNames(): String[]

DatosPersonas

- List<Tipo> listaDatos;

- String[] nombresColumnas

+ Datos()

+ obtenerDatos()

+ getPersonas():List<Persona>

+ getColumnNames(): String[]

**JTable (Componente Visual) JTablePersonas.java**





Línea 13. Estamos creando una referencia a un componente Visual JTable

Línea 14. Estamos creando una referencia a un JScrollPane

Línea 23. Estamos creando el objeto JTable

Línea 25. Estamos asignando el modelo a nuestro objeto (Este modelo se explica líneas mas adelante)

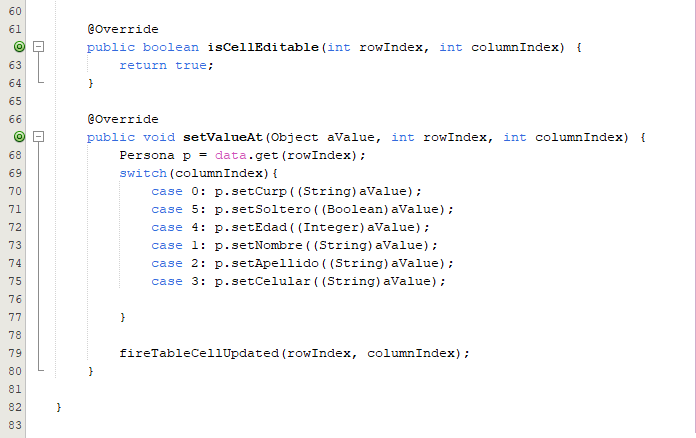
Línea 28. Estamos creando el objeto JScrollPane y le agregamos el objeto JTable

Líneas 43 – 48 No se necesitan para la JTable

**Nuestro Modelo ModeloPersonas.java**







Línea 7. Clase de nuestro modelo, está heredando a la clase: AbstractTableModel, la clase AbstarctTableModel implementa a la interfaz TableModel, de los métodos definidos(declarados) en la interfaz, la clase AbstractTableModel no implementa todos los métodos de esta interfaz concretamente los métodos:

* public int getRowCount();
* public int getColumnCount();
* public Object getValueAt(int row, int column);

por lo tanto, para que se pueda instanciar su modelo deberá implementar estos 3 métodos

Líneas 8. En esta línea estamos declarado una referencia para un List de nombre “data” https://www.arquitecturajava.com/list-vs-arraylist-que-es-mejor/

**DatosPersonas.java**



**Persona.java**



