Tabla de contenidos

Introducción de interactividad. Componentes Swing básicos

1. Gestores de distribución (Layout)

Gestores de distribución (Layout)

El desarrollo de una aplicación, como habrás podido observar a lo largo de este curso, es un proceso complejo que obliga al programador/a (o desarrollador/a) a dominar multitud de técnicas y particularidades del lenguaje en cuestión, o incluso del entorno de desarrollo empleado. Y todo este trabajo lo que pretende es conseguir una aplicación útil para el cliente, que lleve a cabo tareas complejas o laboriosas de forma rápida y con el mínimo esfuerzo para el usuario.
*En **SI Andalucía** concretaron en su momento que una aplicación útil es aquella que además de realizar todas las funciones para las que ha sido diseñada, resulta fácil e intuitiva para el usuario que la va a usar en última instancia. Porque, generalmente, una aplicación sencilla facilita el trabajo, la comprensión y la obtención de resultados útiles.A **Víctor** le cuesta asimilar esta filosofía y sostiene que si la aplicación funciona correctamente no merece la pena perder el tiempo en "maquillarla". **José** discrepa y le explica que una persona que trabaja diariamente con aplicaciones informáticas, tarde o temprano llegará a dominarlas (aunque sólo sea en aquellas funciones que utiliza habitualmente), pero si a eso le unimos un diseño agradable y sencillo, el usuario empleará menos tiempo en aprender su funcionamiento y lo hará con mejor disposición, lo que sin duda mejorará su trabajo final. Lo mejor para una empresa es que sus empleados trabajen a gusto y con agrado.*
***Carmen**, que ha estado presente durante toda la conversación, le explica a **Víctor** que en Java no es muy costoso mejorar el aspecto de las aplicaciones, aunque es necesario planificarlo de algún modo durante la fase de diseño, antes de comenzar con la programación, que no debemos olvidar que será multiplataforma.*

En la unidad anterior pretendíamos mostrar algunos de los conceptos básicos de la programación de interfaces gráficas de usuario. Aprendimos:

- a crear ventanas,
- cuadros de diálogo,
- a crear en ellas paneles de contenido en los que colgar los distintos componentes que queremos que se muestren en nuestra ventana para interactuar con el usuario,
- y le dimos un repaso rápido al aspecto de cada uno de esos componentes que podemos usar.
- También se vio cómo cerrar las ventanas y la aplicación,
- así como la manera de darle distintos aspectos y comportamientos a una aplicación, según la plataforma en la que la quisiéramos ejecutar.

También se ofrecía un ejemplo en el que poder ver en ejecución algunos de esos elementos, con numerosas demostraciones de cómo podemos construirlo con la ayuda del entorno integrado de desarrollo.

1.1. Distribuir los componentes en nuestra aplicación

Distribuir los componentes en nuestra aplicación

Pero para que este ejemplo fuese completo y funcional, necesariamente nos veíamos obligados a usar algunos elementos y conceptos que todavía no se habían analizado en la unidad, y avisábamos que se incluirían en próximas unidades. Uno de esos elementos era el uso de un **gestor de distribución** adecuado. En nuestro ejemplo usábamos `null` como gestor de distribución, que es lo mismo que no usar ninguno. Eso tiene como efecto que puedes situar los componentes en el panel sin más que pinchar y arrastrar con el ratón, dándole el tamaño exacto que deseas, y colocándolos también en la posición que deseas. A primera vista parece lo más deseable, pero no tenemos que olvidar que Java es un lenguaje multiplataforma, muy usado en el desarrollo de aplicaciones web. Resulta difícil, por no decir imposible, saber qué resolución va a tener el ordenador en el que se ejecute nuestra aplicación, ni qué tamaño de pantalla, ni si la aplicación se va a mostrar con la ventana maximizada o no.

Todos estos detalles influirán bastante en la presentación de nuestra aplicación, de forma que si los componentes se distribuyen en la ventana de forma fija, seguramente el aspecto de la ventana será magnífico en el ordenador en que se ha desarrollado, o con el tamaño que le hemos dado a la aplicación, pero será bien diferente en otros ordenadores, con pantallas de otro tamaño, con otras resoluciones, y con la ventana abierta a otro tamaño.

Piensa en la aplicación del ejemplo de la unidad anterior. Tiene un aspecto bastante bueno, al ejecutarla, con todos los elementos distribuidos de forma más o menos simétrica y ocupando todo el tamaño disponible en la ventana.

Pero, sin embargo, si **maximizamos** esa ventana, el efecto sería que nos quedarían todos los componentes arrinconados en la esquina superior izquierda, mientras que prácticamente las otras tres cuartas partes de la ventana, o incluso más, permanecen vacías. Algo como lo que te ponemos a continuación.

Esta apariencia no es tan agradable ni tan amigable. Y eso influye enormemente en el grado de **satisfacción** de los usuarios con la aplicación.

Incluso si para el mismo ordenador y con la ventana también maximizada, cambiamos la resolución, de 1280x1024 a 800x600, tanto la distribución relativa de los componentes, como en consecuencia el aspecto final de la ventana de la aplicación pueden ser bastante distintos. A continuación tienes el aspecto con que quedaría en nuestro caso:

¿No sería deseable poder decirle a la ventana que distribuya sus componentes teniendo en cuenta todas esas características del ordenador en que se va a ejecutar la aplicación?

La idea es que sean cuales sean esas características, el aspecto y la distribución de los componentes en nuestra ventana se adapten a las características de la ventana en el ordenador en que se ejecuta, de acuerdo a las reglas que hayamos establecido, para que la apariencia de nuestra aplicación siga siendo más o menos la deseada, o al menos, la mejor posible en cada circunstancia.

¿Es posible hacer eso en Java?

Es posible, y relativamente **fácil**. Para conseguirlo, **debemos hacer un buen diseño de la interfaz gráfica**, en el que tengamos presentes las posibilidades a considerar, y **debemos elegir el mejor gestor de distribución (Layout) para cada uno de los contenedores o paneles de nuestra ventana**.

Esto se puede hacer mediante el método `setLayout()`, al que se le pasa como argumento un objeto del tipo de Layout que se quiere establecer.

Veamos ahora, a lo largo de los siguientes apartados, cuales son esos gestores de distribución disponibles en Java, y sus características más destacadas.

1.2. FlowLayout

FlowLayout

Una de las posibilidades a la hora de distribuir los componentes dentro de un panel es **situarlos uno detrás de otro, en hilera. Esto se consigue mediante el gestor de distribución** `FlowLayout`. El método `setLayout()` es el encargado de establecer el tipo de gestor de distribución (Layout) que se va a usar para un contenedor (un panel, por ejemplo).

Si nos detenemos en la aplicación que hemos usado en este apartado, el código que genera el diseñador de forma automática para establecer el gestor de distribución `FlowLayout` para el panel de contenido por defecto (`contentPane`) es el siguiente:

```
getContentPane().setLayout(new java.awt.FlowLayout(java.awt.FlowLayout.LEFT, 25, 25));
```

Con esa sentencia establecemos que ese `FlowLayout`, además, va a disponer los componentes alineados a la izquierda, y con un espaciado de 25 píxeles, tanto vertical

como horizontal entre los componentes que se añadan, y entre estos y el borde del panel contenedor.

> **Un `FlowLayout` organiza los componentes en una hilera direccional, como las líneas de texto en un párrafo.**

La dirección de la hilera queda determinada por la propiedad `componentOrientation` del contenedor, y puede tomar uno de los dos valores siguientes:

1. `ComponentOrientation.LEFT_TO_RIGHT`
   Los componentes se van añadiendo empezando por la izquierda, es decir, por el principio de la línea, y completando la línea hacia la derecha, en el orden normal que se escribiría el texto en una línea. Ésta es, por tanto, la orientación habitual que tendremos establecida, y que en raras ocasiones modificaremos.
2. `ComponentOrientation.RIGHT_TO_LEFT`
   Los componentes se van añadiendo empezando por la derecha, es decir, por el final de la línea, y van completando la línea hacia la izquierda.

`FlowLayout` es utilizado típicamente para organizar botones en un panel. **Distribuye los botones horizontalmente hasta que no caben más botones en la misma línea, en cuyo caso se comienzan a añadir a otra nueva línea**. La alineación de la línea queda determinada por la propiedad **align**. Los valores posibles son los representados por las constantes públicas de clase de tipo entero (`public static final int`) que indicamos a continuación, y cuyos auténticos valores enteros son los que se indican entre paréntesis detrás de su descripción, al final de la línea:

`LEFT`	**Cada línea de componentes debe ser alineada a la Izquierda**	(Valor: 0)
`CENTER`	**Cada línea de componentes debe ser centrada.**	(Valor: 1)
`RIGHT`	**Cada línea de componentes debe ser alineada a la Derecha**	(Valor: 2)
`LEADING`	**Cada fila de componentes debe ser alineada al borde principal según el sentido de la orientación del contenedor. Por ejemplo, a la izquierda con orientación de izquierda a derecha (LEFT_TO_RIGHT)**	(Valor: 3)
`TRAILING`	**Cada fila de componentes debe ser alineada al borde posterior según el sentido de la orientación del contenedor. Por ejemplo, a la derecha con orientación de izquierda a derecha (LEFT_TO_RIGHT)**	(Valor: 4)

Pero en condiciones normales, las propiedades que nos permite manipular el entorno NetBeans para el gestor de distribución `FlowLayout`, son las siguientes:

- **Alignment (Left, Center o Right).** Al modificar esta propiedad, realmente estamos invocando al método `setAlignment (int align)` de la clase `FlowLayout`, donde `align` **es un valor de 0 a 4, o sus equivalentes constantes de clase** (`LEFT, CENTER, RIGHT, LEADING, TRAILING`). Estas dos últimas, no son muy usadas, y NetBeans no las incluye en el diseñador.
- **Horizontal Gap**. Establece un valor entero que fija el espacio horizontal de separación entre los distintos componentes, y entre estos y los bordes del contenedor. Realmente invoca al método `setHgap (int hgap)`
- **Vertical Gap.** Establece un valor entero que fija el espacio vertical de separación entre los distintos componentes, y entre estos y los bordes del contenedor. Realmente invoca al método `setVgap (int vgap)`

Por ejemplo, en la siguiente demostración se muestra el uso de `FlowLayout` para posicionar varios botones, pulsa sobre cada opción para ver una imagen de su comportamiento con las propiedades anteriores:

Visualiza la representación de las ventanas con gestor FlowLayout

A continuación puedes ver una simulación de su uso en un proyecto bajo NetBeans:

DEMO: Visualiza cómo se comportan una serie de botones al utilizar FlowLayout

1.3. BorderLayout

BorderLayout

Aunque `FlowLayout` se usa mucho para distribuir grupos de botones (y otros tipos de componentes), existen muchas más posibilidades a la hora de seleccionar un gestor de distribución.

En concreto, ¿te parecería interesante distribuir elementos de forma que se indique la parte del contenedor en el que deben situarse? Eso es lo que hace `BorderLayout`.

`BorderLayout` **divide el contenedor** (normalmente este contenedor será un panel) **en 5 zonas, de forma que al añadir un componente al contenedor con el método `add()`, como segundo argumento de este método podamos indicarle en qué zona del contenedor queremos situar ese componente.**

Esas 5 zonas posibles son:

- **CENTER**
- **NORTH**
- **SOUTH**
- **EAST**
- **WEST**

Como puedes imaginar, se trata de constantes de la clase `BorderLayout`, (definidas como `public static final String`) que podemos usar en lugar de sus valores concretos, a modo de mnemotécnicos. Puedes consultar sus valores en la documentación de la API de Java, que podemos suponer que ya sabes consultar con soltura.

En cualquier caso, representan exactamente la zona que cabe esperar de sus nombres.

En la siguiente imagen, aparece la ventana de una aplicación, que tiene un único panel con `BorderLayout` como gestor de distribución, al que hemos añadido 5 botones, cada uno de ellos situado en una zona diferente de las 5 que establece `BorderLayout`.

> **Si cambiamos el tamaño de la ventana, cambiará el tamaño de cada componente, pero no su posición relativa dentro de la ventana.**

La siguiente imagen muestra el resultado al aumentar el tamaño de la ventana, de forma que el botón Centro amplía su tamaño tanto en anchura como en altura para ocupar toda la zona centro, los botones Sur y Norte se hacen más anchos para ocupar toda su zona respectiva, y los botones Este y Oeste se hacen más altos también para ocupar toda su zona.

El código que genera automáticamente el diseñador de NetBeans es el siguiente (los comentarios en azul no los añade el diseñador, los hemos añadido nosotros a modo de explicación):

```
        /* Se crean los 5 botones*/
           jButton1 = new javax.swing.JButton();
           jButton2 = new javax.swing.JButton();
           jButton3 = new javax.swing.JButton();
           jButton4 = new javax.swing.JButton();
           jButton5 = new javax.swing.JButton();
    /* Se establece la operación por defecto a realizar cuando se cierre la ventana*/
```

```
        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
/*Se fija el texto que va a mostrar cada botón, y se añade al panel de contenido de la ventana
* ese botón, pasándoselo como primer parámetro al método add(), cuyo segundo parámetro es la
* constante de la clase BorderLayout que le indica la zona en la que lo debe situar. En este
* caso, cada botón está en una zona diferente.*/
        jButton1.setText("Centro (CENTER)");
         getContentPane().add(jButton1, java.awt.BorderLayout.CENTER);

        jButton2.setText("Sur (SOUTH)");
         getContentPane().add(jButton2, java.awt.BorderLayout.SOUTH);

        jButton3.setText("Este (EAST)");
         getContentPane().add(jButton3, java.awt.BorderLayout.EAST);

        jButton4.setText("Oeste (WEST)");
         getContentPane().add(jButton4, java.awt.BorderLayout.WEST);

        jButton5.setText("Norte (NORTH)");
         getContentPane().add(jButton5, java.awt.BorderLayout.NORTH);
```

En el ejemplo anterior, los 5 botones están tan juntos al ser insertados y dimensionados automáticamente por **`BorderLayout`** que casi no se distingue que son botones.

1.3.1. Espacios de separación

Espacios de separación

¿Podríamos establecer unos espacios de separación tanto verticales como horizontales para los componentes del contenedor?

Podemos, y se hace mediante las propiedades **`Horizontal Gap`** y **`Vertical Gap`**, tal y como se vio anteriormente en **`FlowLayout`**.

La imagen siguiente muestra el aspecto de la aplicación al establecer una separación de **15**, tanto horizontal como vertical, entre componentes del contenedor. En este caso, la separación no se aplica entre los componentes y los bordes del contenedor.

El código que genera automáticamente el diseñador de NetBeans al modificar esas dos propiedades del gestor de distribución **`BorderLayout`** es la siguiente línea de código:

```
getContentPane().setLayout(new java.awt.BorderLayout(15, 15));
```

- **El método `getContentPane()` devuelve el panel de contenido por defecto de la ventana (`contentPane`).**
- Para ese panel **se usa el método `setLayout()` para establecer el gestor de distribución que se va a usar con ese panel.**
- Como argumento **al método `setLayout()` se le pasa un nuevo `BorderLayout`, que se crea en línea invocando al constructor con el operador `new`.**
- **Al constructor de `BorderLayout` se le pasan como argumentos dos números enteros, que representan el espaciado horizontal y vertical, respectivamente, (`hgap` y `vgap`) que debe haber entre los distintos componentes que se añadan al panel.**
- Esos dos valores **`hgap`** y **`vgap`**, también pueden ser fijados para el gestor de distribución mediante los métodos **`setHgap()`** y **`setVgap()`** de la clase **`BorderLayout`**, pero en este caso, se han fijado directamente mediante el constructor.

Puede que hayas pensado qué ocurre si queremos insertar varios componentes en una misma zona del contenedor. ¿Es posible?

Es posible, pero no es recomendable hacerlo directamente, ya que:

- **Si se añade un segundo botón (un segundo componente) a una zona, sólo uno será visible.**
- **El botón que queda visible está superpuesto a todos los demás, que no serán visibles ni utilizables, por tanto carece de sentido hacerlo así.**
- **Una posibilidad**, que veremos con más detalle en el apartado 1.8, será **situar un panel en esa zona, y sobre ese panel situar todos los componentes que deseemos, usando el gestor de distribución que estimemos oportuno**, como por ejemplo **`FlowLayout`**.

En la siguiente imagen se aprecia el resultado de sustituir el botón de la zona Norte (**`NORTH`**) por un panel que contiene dos botones y dos casillas de verificación, con distribución **`FlowLayout`**, y al que se le ha puesto un borde titulado.

El código que se genera al hacer esos cambios es el siguiente, al que hemos añadido algunos comentarios en azul, para resaltar las diferencias:

```
jCheckBox2 = new javax.swing.JCheckBox();

        jButton1 = new javax.swing.JButton();
        jButton2 = new javax.swing.JButton();
        jButton3 = new javax.swing.JButton();
        jButton4 = new javax.swing.JButton();

        getContentPane().setLayout(new java.awt.BorderLayout(15, 15));

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

/* Se establece  FlowLayout  como gestor de distribución del panel  jPanel1 , y se le fija un
 * borde titulado, para identificarlo claramente como el elemento que está situado en la zona
 * Norte del  BorderLayout  de la ventana.
 */

         jPanel1.setLayout(new java.awt.FlowLayout(java.awt.FlowLayout.CENTER, 15, 15));
         jPanel1.setBorder(new javax.swing.border.TitledBorder(new
                 javax.swing.border.TitledBorder("Panel en posici\u00f3n Norte (NORTH)")));

/* Añadimos los dos botones y las dos casillas de verificación al panel  jPanel1 , fijando
 * previamente el texto que mostrará cada componente.
 */

         jButton5.setText("jButton5");
         jPanel1.add(jButton5);

         jButton6.setText("jButton6");
         jPanel1.add(jButton6);

         jCheckBox1.setText("jCheckBox1");
         jPanel1.add(jCheckBox1);

         jCheckBox2.setText("jCheckBox2");
         jPanel1.add(jCheckBox2);

/* Se añade al panel de contenido de la ventana que nos devuelve el método  getContentPane() ,
 * un nuevo componente, mediante el método  add() , al que le indicamos como primer parámetro
 * el componente a añadir ( jPanel1 ), y como segundo parámetro la zona en la que lo queremos
 * situar según  BorderLayout  (NORTH)
 */
```

```
        getContentPane().add(jPanel1, java.awt.BorderLayout.NORTH);

/* El resto de componentes siguen ocupando las posiciones que previamente se habían
 * establecido según  BorderLayout .
 */
        jButton1.setText("Centro (CENTER)");
        getContentPane().add(jButton1, java.awt.BorderLayout.CENTER);

        jButton2.setText("Este (EAST)");
        getContentPane().add(jButton2, java.awt.BorderLayout.EAST);

        jButton3.setText("Oeste (WEST)");
        getContentPane().add(jButton3, java.awt.BorderLayout.WEST);

        jButton4.setText("Sur (SOUTH)");
        getContentPane().add(jButton4, java.awt.BorderLayout.SOUTH);
/* Se crea el panel  jPanel1 , y los componentes que queremos que contenga, que son dos botones
 * y dos casillas de verificación.
 */

        jPanel1 = new javax.swing.JPanel();
        jButton5 = new javax.swing.JButton();
        jButton6 = new javax.swing.JButton();
        jCheckBox1 = new javax.swing.JCheckBox();
```

1.4. GridLayout

GridLayout

Con **`BorderLayout`** podíamos decir la zona en la que queríamos situar un componente para que se mantenga en esa posición aunque cambie el tamaño de la ventana, pero las zonas disponibles están limitadas a 5, las indicadas en el apartado anterior.

Seguramente habrás pensado en la posibilidad de establecer más zonas dentro del contenedor. ¿Sería útil poder dividirlo en una especie de cuadrícula, de forma que se definan tantas zonas como se deseen? Ése es más o menos el planteamiento de **`GridLayout`**.

`GridLayout` distribuye los componentes de un contenedor en una rejilla o cuadrícula rectangular. El contenedor es dividido en rectángulos de igual tamaño, y cada componente se sitúa dentro de uno de esos rectángulos.

El aspecto de una ventana en la que hemos establecido como gestor de distribución **`GridLayout`**, con 3 filas y 2 columnas, y con el espaciado horizontal y vertical (**`hgap`** y **`vgap`**) a 10, es el siguiente:

No obstante, **cuando el número de filas y columnas han sido fijados ambos a valores distintos de cero, ya sea por el constructor de `GridLayout` o bien por haber usado los métodos `setRows()` y `setColumns()`, el número de columnas se ignora**.

> **De hecho el número de columnas se calcula a partir del número de filas y del número de componentes que se han añadido realmente al contenedor gestionado por `GridLayout`.**

Por ejemplo, si se han especificado 3 filas y 2 columnas, y se añaden 9 componentes al contenedor, se mostrarán en 3 filas y 3 columnas, ignorando el valor especificado para columnas.

> **Sólo tiene sentido especificar el número de columnas cuando el número de filas es cero.**

En ese caso, **se distribuirán los componentes en el número de columnas especificado, usando tantas filas como sean necesarias para mostrar todos los componentes añadidos.**

Puedes ver algunos ejemplos en la siguiente de demostración:

Visualiza los ejemplos

A continuación puedes ver una simulación de su comportamiento como parte de un proyecto bajo NetBeans:

DEMO: Mira cómo se comportan los componentes si utilizamos GridLayout

Al igual que en los casos anteriores, **cada uno de esos elementos, en vez de ser botones, podrían ser paneles que a su vez tuvieran otro gestor de distribución** para los componentes que contuvieran.

De entre los métodos disponibles en la clase `GridLayout`, además de los constructores, destacamos los siguientes:

- **`setHgap()`** Establece el espacio horizontal entre componentes.
- **`setVgap()`** Establece el espacio vertical entre componentes.
- **`setRows()`** Establece el número de filas para el **`GridLayout`**.
- **`setColumns()`** Establece el número de columnas para el **`GridLayout`**.
- **`getRows()`** Devuelve el número de filas que tiene el **`GridLayout`**.
- **`getColumns()`** Devuelve el número de columnas que tiene el **`GridLayout`**.

Para una completa referencia a los métodos disponibles en la clase **`GridLayout`**, como siempre, debes consultar la documentación de la API de Java.

1.5. GridBagLayout

GridBagLayout

Con **`GridLayout`** tenemos bastantes más posibilidades a la hora de gestionar la distribución de los componentes de un contenedor, pero aún tenemos una restricción en cierto modo incómoda. ¿Sabes cuál es?

> **Con `GridLayout` todas las celdas de la cuadrícula tienen el mismo tamaño, y cada componente está en una sola celda, por lo que todos los componentes tienen el mismo tamaño, lo cual es una limitación.**

Esto se soluciona mediante el uso de **`GridBagLayout`**, que nos aporta mucha más flexibilidad. A cambio, su uso adecuado implica una mayor complejidad.

- **`GridBagLayout` es un gestor de distribución flexible, que alinea componentes vertical y horizontalmente, sin requerir que sean del mismo tamaño.**

Cada objeto `GridBagLayout` mantiene una cuadrícula dinámica y rectangular de celdas, y cada componente ocupa una o más celdas, que conforman su área de visualización (display area).

- **Cada componente manejado por `GridBagLayout` se asocia con una instancia de `GridBagConstraints`**. El objeto `constraints` especifica **dónde debe colocarse sobre la cuadrícula el área de visualización de un componente, y cómo el componente debe ser posicionado dentro de su área de visualización.**
- Además de las restricciones del objeto `constraints`, `GridBagLayout` también considera el tamaño mínimo (`minimum size`) y el tamaño preferido (`preferred size`) del componente para determinar el tamaño del mismo.
- **Para la orientación habitual del contenedor, que suponemos será left-to-right, la coordenada (0,0) representa la esquina superior izquierda del contenedor, con x incrementándose hacia la derecha, e y incrementándose hacia abajo.**

1.5.1. Uso

Uso

Para usar un `GridBagLayout` con efectividad, debes personalizar uno o más de los objetos `GridBagConstraints` que están asociados con sus componentes, fijando una o más de las variables de instancia del `GridBagConstrains`, que te mostramos a continuación:

GridBagConstraints.gridx, GridBagConstraints.gridy

Especifica la celda que contiene la primera esquina del área de visualización del componente, donde la primera celda de la cuadrícula tiene como dirección `gridx=0, gridy=0`.

Se usa `GridBagConstraints.RELATIVE` (el valor por defecto) para especificar que el componente será colocado inmediatamente a continuación (a lo largo del eje x para `gridx` o del eje y para `gridy`) del componente que fue añadido al contenedor justo antes que el componente en cuestión.

GridBagConstraints.gridwidth, GridBagConstraints.gridheight

Especifica el número de celdas en una fila (para la variable `>gridwidth`) o en una columna (para la variable `gridheight`) que ocupa el área de visualización de un componente. El valor por defecto es 1.

- Se usa `GridBagConstraints.REMAINDER` para especificar que el área de visualización del componente ocupará desde la columna de comienzo del componente (`gridx`) hasta la última celda en la fila (para la variable `gridwidth`) o desde la fila de comienzo del componente (`gridy`) hasta la última celda en la columna (para la variable `gridheight`).
- Se usa `GridBagConstraints.RELATIVE` para especificar que el área de visualización del componente ocupará desde la columna de comienzo del componente (`gridx`) hasta la primera celda ocupada por el siguiente componente en su fila (para la variable `gridwidth`) o desde la fila de comienzo del componente (`gridy`) hasta la primera celda ocupada por el siguiente componente en su columna (para la variable `gridheight`)

GridBagConstraints.fill

Se usa cuando el área de visualización del componente es mayor que el tamaño requerido para determinar si (y como) redimensionar el componente. Valores posibles son:

- `GridBagConstraints.NONE` (por defecto)
- `GridBagConstraints.HORIZONTAL` (hace el componente suficientemente ancho para ajustar su área de visualización horizontalmente, pero no cambia su altura)
- `GridBagConstraints.VERTICAL` (hace el componente suficientemente alto para ajustar su área de visualización verticalmente, pero no cambia su anchura)
- `GridBagConstraints.BOTH` (hace que el componente se ajuste para ocupar su área de visualización completa.

GridBagConstraints.ipadx, GridBagConstraints.ipady

Especifica el relleno interno del componente dentro de la distribución, cuánto añadir al tamaño mínimo del componente. El ancho del componente será al menos su ancho mínimo más `ipadx` píxeles. Similarmente, la altura del componente será al menos su mínima altura más `ipady`.

GridBagConstraints.insets

Especifica el relleno externo del componente, el espacio mínimo entre el componente y los bordes de su área de visualización.

GridBagConstraints.anchor

Se usa cuando el componente es más pequeño que su área de visualización, para determinar dónde (dentro del área de visualización) se debe colocar el componente. Hay dos tipos de valores posibles: relativos y absolutos.

Los **valores relativos** son interpretados como relativos a la propiedad `ComponentOrientation` del contenedor, mientras que los **valores absolutos** no. Son valores válidos:

Valores Absolutos	Valores Relativos
`GridBagConstraints.NORTH`	`GridBagConstraints.PAGE_START`
`GridBagConstraints.SOUTH`	`GridBagConstraints.PAGE_END`
`GridBagConstraints.WEST`	`GridBagConstraints.LINE_START`
`GridBagConstraints.EAST`	`GridBagConstraints.LINE_END`
`GridBagConstraints.NORTHWEST`	`GridBagConstraints.FIRST_LINE_START`
`GridBagConstraints.NORTHEAST`	`GridBagConstraints.FIRST_LINE_END`
`GridBagConstraints.SOUTHWEST`	`GridBagConstraints.LAST_LINE_START`
`GridBagConstraints.SOUTHEAST`	`GridBagConstraints.LAST_LINE_END`
`GridBagConstraints.CENTER (por defecto)`	

GridBagConstraints.weightx, GridBagConstraints.weighty

Se usa para determinar cómo distribuir espacio, lo que es importante para especificar el comportamiento al redimensionar. A menos que se especifique un ancho para al menos un componente en una fila (`weightx`) y columna (`weighty`), todos los componentes se agrupan juntos en el centro del contenedor. Esto es porque cuando el ancho es cero (por defecto), el objeto `GridBagLayout` pone algún espacio extra entre su cuadrícula de celdas y los bordes del contenedor.

La imagen siguiente muestra diez componentes (botones) manejados por un `GridBagLayout`.

Cada uno de los diez componentes tiene el campo **`fill`** de su objeto asociado **`GridBagConstraints`** establecido al valor **`GridBagConstraints.BOTH`**.

Además, los componentes tienen las siguientes restricciones (**`GridBagConstraints`**):

Propiedades constraints (entre paréntesis el nombre en NetBeans)							
Componentes	**gridx (Grid X)**	**gridy (Grid Y)**	**gridwidth (Grid Width)**	**gridheight (Grid Height)**	**fill (Fill)**	**weightx (Weight X)**	**weight (Weight Y)**
JButton1	**0**	**0**	**1**	**1**	**BOTH**	**1**	**1**
JButton2	1	0	1	1	BOTH	1	1
JButton3	**2**	**0**	**1**	**1**	**BOTH**	**1**	**1**
JButton4	3	0	**REMAINDER**	1	BOTH	1	1
JButton5	**0**	**1**	**REMAINDER**	**1**	**BOTH**	**0**	**1**
JButton6	0	2	RELATIVE	1	BOTH	0	1
JButton7	**3**	**2**	**REMAINDER**	**1**	**BOTH**	**0**	**0**
JButton8	0	3	1	2	BOTH	0	1
JButton9	**1**	**3**	**REMAINDER**	**1**	**BOTH**	**0**	**1**
JButton10	1	4	**REMAINDER**	1	BOTH	0	1

A continuación tienes el código asociado a este ejemplo con **`GridBagLayout`** .

Descarga el proyecto EjemploGridBagLayout

Veamos también una simulación de su uso con NetBeans:

DEMO: Mira el comportamiento de los componentes si utilizamos GridBagLayout

Autoevaluación

1.6. CardLayout

CardLayout

Seguramente te habrás quedado con la impresión de que **`GridBagLayout`** ofrece toda la flexibilidad que podemos desear a la hora de colocar y distribuir componentes en un contenedor, pero ahí no terminan las posibilidades. Aún existen más gestores de distribución, y uno de ellos es **`CardLayout`**.

> **`CardLayout`** **es un gestor de distribución que trata cada componente en el contenedor como una carta. Sólo una carta es visible en cada momento, y el contenedor actúa como una "baraja de cartas". El primer componente añadido al** **`CardLayout`** **es el componente visible cuando el contenedor se muestra por primera vez.**

El orden de las "cartas" (componentes) viene determinado por el orden interno de los componentes del contenedor. **`CardLayout`** define un conjunto de método que permiten a la aplicación ir mostrando esas cartas secuencialmente, o mostrar una carta especificada. El método **`addLayoutComponent (java.awt.Component, java.lang.Object)`** puede usarse para asociar un identificador de tipo **`String`** con una carta dada, para acceder directamente a visualizar esa carta.

Si pensamos que cada una de esas "**cartas**" es un componente Swing, incluso un panel que contenga otros componentes adicionales, con su propio gestor de distribución, podemos ver que esto lo que nos permite es ir cambiando de forma dinámica los componentes y la funcionalidad asociada, de una ventana.

A continuación tienes un ejemplo que usa **`CardLayout`** como gestor de distribución. En él aparece una ventana **`Jframe`**, que tiene **`CardLayout`** como gestor de distribución para su **`contentPane`**, al que se añaden 3 paneles distintos, cada uno de los cuales muestra la foto de una mascota, su nombre, y un botón para pasar a la siguiente mascota. El código asociado a la pulsación de cada botón es una invocación al método **`show()`** de la clase **`CardLayout`**, indicando cual es el siguiente componente a mostrar. En este caso, el siguiente componente, es otro panel, con la foto y el nombre de otra mascota, y con el botón "**Siguiente mascota** >>"

Observa el código que permite usar **`CardLayout`** :

```
        //...
/*Se establece que el panel de contenido va a usar como gestor de distribución CardLayout,
* con separación horizontal y vertical entre componentes de 20*/
     getContentPane().setLayout(new java.awt.CardLayout(20, 20));

//...
/*Se añaden al panel de contenido (el contenedor) los tres paneles, asignándole a cada uno
* un nombre a modo de mnemotécnico que nos permita referirnos a él en el resto del programa. */

     getContentPane().add(jPanel1, "card2");
//...
     getContentPane().add(jPanel3, "card1");
//...
     getContentPane().add(jPanel2, "card3");
//...
/*A continuación están los tres métodos asociados a los tres botones, uno de cada panel,
* que lo que hacen es indicar cual va a ser el siguiente componente o "carta"
* (en este caso panel) que se va a visualizar para el CardLayout establecido.*/

    private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {

       /* Capturamos el Layout que tiene el panel de contenido con el método getLayout(),y le
        aplicamos un casting explícito, para asegurarnos que sea un objeto  Layout de tipo
        CardLayout, y se lo asignamos a la referencia cL */

       CardLayout cL =(CardLayout)(getContentPane().getLayout());

      /* Invocamos al método show() para cL (CardLayout)que se encarga de mostrar el
       componente que le indiquemos. Alternativamente, podríamos haber usado los métodos
       next(),previous(), first() o last() para movernos a la "carta" siguiente, anterior,
       primera o última, respectivamente. */

       cL.show(getContentPane(), "card1");
    }
/* Los otros dos métodos son similares*/
    private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
       CardLayout cL =(CardLayout)(getContentPane().getLayout());
       cL.show(getContentPane(), "card2");
    }

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
```

```
            CardLayout cL =(CardLayout)(getContentPane().getLayout());
            cL.show(getContentPane(), "card3");
        }
    //..
```

Descarga el proyecto EjemploCardLayout

Aunque en este ejemplo el aspecto de los tres paneles es bastante parecido, cambiando sólo la foto y el nombre, debes observar que se trata de tres paneles totalmente distintos, con componentes distintos. Nada me impide poner paneles que tengan distinto número de componentes, de distinto tipo, y con una funcionalidad totalmente distinta asociada, con lo que el aspecto de una ventana puede cambiar totalmente, con relativa facilidad.

1.7. BoxLayout

BoxLayout

Todavía existen más formas de distribuir los componentes en un panel, de forma que se tengan en cuenta las características del entorno en el que se va a ejecutar la aplicación.

Uno de ellos es **`BoxLayout`**, **que permite distribuir sobre él múltiples componentes, ya sea horizontal o verticalmente. Los componentes, una vez distribuidos sobre el gestor, quedarán fijados en su sitio y no se moverán de ahí, de tal manera que, por ejemplo, si redimensionamos el marco y lo hacemos más pequeño, es posible que algún componente no se vea completo, pues estos no van a ser bajados a otra línea automáticamente por el simple hecho de disminuir el tamaño de la ventana.**

Se puede considerar que `BoxLayout` es una versión más amplia de `FlowLayout`

Anidando múltiples paneles con diferentes combinaciones horizontales y verticales se consigue un efecto similar a **`GridBagLayout`**, pero sin tanta complejidad. Un ejemplo:

El diagrama muestra dos paneles (P1 y P2) ordenados horizontalmente, cada uno conteniendo 3 componentes ordenados verticalmente.

El gestor de distribución **`BoxLayout`** está construido con un parámetro **`axis`** (eje) que especifica el tipo de distribución que se hará. Hay cuatro posibilidades:

1. **`X_AXIS`** - **Los componentes son distribuidos horizontalmente de izquierda a derecha**.
2. **`Y_AXIS`** - **Los componentes son distribuidos verticalmente de arriba a abajo.**
3. **`LINE_AXIS`** - **Los componentes son distribuidos de la misma forma que las palabras se colocan en una línea**.
   - **Si la propiedad `ComponentOrientation` del contenedor es horizontal, los componentes se distribuyen horizontalmente**, y en caso contrario se distribuyen verticalmente.
   - **Para orientaciones horizontales, si la propiedad `ComponentOrientation` del contenedor es de izquierda a derecha (left to right), los componentes se distribuyen de izquierda a derecha**, y en caso contrario se distribuyen de derecha a izquierda.
   - **Para orientaciones verticales, los componentes siempre se distribuyen de arriba abajo.**
4. **`PAGE_AXIS`**- **Los componentes se distribuyen como las líneas en una página**.
   - **Si la propiedad `ComponentOrientation` del contenedor es horizontal, los componentes se distribuyen verticalmente**, y en caso contrario se distribuyen horizontalmente.
   - **Para orientaciones horizontales, si la propiedad `ComponentOrientation` del contenedor es de izquierda a derecha (left to right), los componentes se distribuyen de izquierda a derecha**, y en caso contrario se distribuyen de derecha a izquierda.
   - **Para orientaciones verticales, los componentes siempre se distribuyen de arriba abajo**.

   Para cualquiera de las cuatro posibilidades, los componentes se ordenan en el mismo orden en que fueron añadidos al contenedor.

`BoxLayout` intenta ordenar los componentes respetando su ancho preferido (**`preferred width`**), para distribución horizontal, o su altura preferida (**`preferred height`**) para distribución vertical.

- Para distribución horizontal, si no todos los componentes en la fila tienen la misma altura, **`BoxLayout`** intenta hacerlos a todos tan altos como el más alto de los componentes.
- Si esto no es posible para un componente particular, entonces **`BoxLayout`** alinea este componente verticalmente, de acuerdo con la propiedad **`Yalignment`** del componente.
- Por defecto, un componente tiene la propiedad **`Yalignment`** al valor de 0.5, lo que significa que el centro vertical del componente tendría la misma coordenada Y que los centros verticales de otros componentes con la propiedad **`Yalignment`** al valor 0.5
- Similarmente, para una distribución vertical, **`BoxLayout`** intenta hacer todos los componentes en la columna tan anchos como el más ancho de los componentes.
- Si esto falla, los alinea horizontalmente de acuerdo a la propiedad **`Xalignment`**.
- Para la distribución **`PAGE_AXIS`** , la alineación horizontal se hace en base al borde principal del componente. En otras palabras, un valor de **`Xalignment`** de 0.0 representa el borde izquierdo de un componente si el contenedor tiene la propiedad **`ComponentOrientation`** fijada como de izquierda a derecha (**left to right**) y representa el borde derecho del componente en caso contrario.
- Lo habitual en nuestro caso, será que la propiedad **`ComponentOrientation`** sea efectivamente **left to right**.

En vez de usar `BoxLayout` directamente, muchos programas usan la clase `Box`, que es un contenedor de peso ligero que usa `BoxLayout` como gestor de distribución y que además proporciona métodos que ayudan al programador a usar y manejar `BoxLayout` con más facilidad. Añadir componentes a múltiples objetos **`Box`** anidados es una poderosa forma de conseguir el ordenamiento que se desea.

> ***PARA SABER MÁS:***
> ***Puedes encontrar más información y ejemplos sobre el uso de `BoxLayout` viendo la sección "Cómo Usar `BoxLayout`" (How to Use `BoxLayout`) del Tutorial de Java. Es especialmente interesante que te descargues los ejemplos que te proporciona el tutorial, y que además de ejecutarlos, mires el código, para ver cómo se usa `BoxLayout`.***
> ***El Tutorial está en inglés. Lamentablemente, el dominio de los países anglosajones en todo lo referente a informática en general, y a lenguajes de programación en particular, hace imprescindible que te vayas familiarizando con la consulta de documentación técnica en Inglés.***
> ***Sección "How to Use BoxLayout" (cómo usar BoxLayout) del Tutorial de Java*** [Versión en caché]

Autoevaluación

1.8. AbsoluteLayout y Null Layout

AbsoluteLayout y Null Layout

A estas alturas ya eres casi una persona experta a la hora de gestionar la distribución de los componentes dentro de la ventana de una aplicación, pero aún nos quedan por comentar dos posibilidades más a tener en cuenta como gestores de distribución disponibles en el entorno de desarrollo.

¿Recuerdas que en los ejemplos de la unidad 17 se usaba como gestor de distribución **`null`**? Justamente eso es lo que representa **`Null Layout`**: La ausencia de un gestor de distribución. La otra posibilidad, muy similar, es **`AbsoluteLayout`**, pero en este caso es un gestor de distribución propio del entorno NetBeans, que no es por

tanto compatible con el estándar de Java, y que si lo usamos, no funcionará correctamente si no disponemos de la clase que lo implementa, y que nos la proporciona NetBeans. O sea, que no funcionará correctamente en otros entornos si no nos preocupamos de distribuir junto a nuestra aplicación las clases **`AbsoluteLayout`** necesarias.

AbsoluteLayout

1. **Permite situar los componentes exactamente donde se desee dentro de la ventana**.
2. **Permite moverlos arrastrándolos con el ratón** en el diseñador del entorno
3. **Permite redimensionarlos arrastrando sus bordes con el ratón**.
4. **Es particularmente útil para hacer prototipos rápidos,** en los que no hay limitaciones formales ya que no tienes que preocuparte por tener en cuenta ninguna configuración concreta.
5. Sin embargo **no es recomendable para producir aplicaciones "finales",** dado que las localizaciones y dimensiones fijas de los componentes no cambiarían con el entorno, pudiendo afectar negativamente al aspecto que presenta la aplicación.
6. Además, para distribuir una aplicación con el gestor de distribución **`AbsoluteLayout`** , propio del IDE, debes incluir las clases **`AbsoluteLayout`** , ya que no son parte del estándar de Java. Estas clases están listas para ser usadas con este fin en la ruta:

   ```
   <dir-instal>/ide5/modules/ext/AbsoluteLayout.jar
   ```

donde **`<dir-instal>`** representa el directorio de instalación de NetBeans, que en principio puede ser distinto para cada usuario.

Null Layout

1. **Se usa para diseñar ventanas sin ningún gestor de distribución.**
2. Como en el caso de **`AbsoluteLayout`**, **es útil para hacer prototipos rápidos**
3. Pero **no se recomienda para producir aplicaciones finales**, ya que las localizaciones y dimensiones fijas de los componentes no cambiarían cuando el entorno cambiara, lo que pude afectar a la correcta visualización de la ventana.

A pesar de que como dijimos en la unidad anterior, **cuidar el aspecto** de la aplicación es algo muy importante, también dijimos que en este módulo no nos debíamos obsesionar demasiado con las cuestiones de presentación, ya que eso se aborda con más detalle en el módulo de Diseño y Realización de Servicios de Presentación en Entornos Gráficos. Por ahora lo fundamental será el correcto funcionamiento. Como además el tiempo en el módulo de Programación en Lenguajes Estructurados está bastante ajustado, y como usar **`Null Layout`** simplifica sensiblemente el tiempo que debe dedicarse al diseño de la distribución, consiguiendo acortar los tiempos de producción, será bastante habitual, por sencillez y rapidez, que usemos directamente **`Null Layout`** como gestor de distribución.

1.9. Uso de Paneles combinados con "Layouts"

Uso de Paneles combinados con "Layouts"

Hasta ahora **hemos visto muchas posibilidades** a la hora de distribuir los componentes en un contenedor, **pero todas, salvo `Null Layout` tienen en mayor o menor medida ciertas limitaciones** a la hora de escoger las zonas en las que queremos situar cada componente, estableciendo que los componentes deben ir bien en una fila, o en una columna, o en celdas de una cuadrícula más o menos flexible, pero evidentemente, con ciertas limitaciones.

¿De qué elemento podemos servirnos para romper totalmente esas limitaciones?

Seguramente lo imaginas, porque ya lo hemos usado en alguno de los ejemplos anteriores:

> **El elemento que nos permite salvar la mayoría de limitaciones que nos imponen los Layout son los Paneles (`Jpanel`).**

- Si consideramos un panel como un componente que queremos añadir a un contenedor que tiene un gestor de distribución determinado,
- Si pensamos que ese Panel, además de un componente es a su vez un contenedor que tiene su propio gestor de distribución,
- Si pensamos que ese gestor de distribución del panel puede ser distinto al del contenedor principal,
- Si consideramos que cada panel a su vez puede contener como componentes a otros paneles con sus propios gestores de distribución,
- Nos damos cuenta de que las posibilidades son casi infinitas, y las limitaciones se difuminan.

Por tanto, **la estrategia adecuada a seguir para hacer un buen diseño de la interfaz gráfica y una buena gestión de la distribución de los componentes es** la siguiente:

El uso combinado de Paneles y Layouts.

A continuación hemos rediseñado el ejemplo de la ecuación de segundo grado, que en la unidad anterior usaba **`Null Layout`**, de forma que se combine el uso de paneles y layout para que el aspecto de la aplicación sea el mejor posible aunque cambien las características del entorno en el que se ejecuta.

Descarga el proyecto EcuacionSegundoGradoConLayout

Un último comentario sobre los Layouts. A pesar de que son un elemento importante para conseguir mejorar la apariencia de las aplicaciones, para mejorar su usabilidad, y para hacerlas más agradables y amigables para el usuario, y a pesar de la extensión que en esta unidad hemos dedicado a su estudio, debes tener claro que no es ni mucho menos el aspecto más importante de la misma, y que el grueso de lo que es fundamental en esta unidad está contenido en los apartados siguientes.

2. Cuadros de texto

Cuadros de texto

> *__Carmen__ dice que uno de los aspectos principales de una aplicación informática ha sido desde sus comienzos el diálogo entre el usuario y la aplicación. Este diálogo se sucede en ambas direcciones pero lo que más nos interesa es el modo en que el usuario proporciona los datos que nuestra aplicación debe procesar para proporcionar los resultados deseados, de forma rápida y precisa. __Víctor__ dice que eso es fácil, con los cuadros de texto que ya se han utilizado en los ejemplos anteriores, el usuario ya sabe que se trata de una zona de la ventana destinada a la entrada de datos (aunque a veces también para mostrar los resultados).*
> *__Carmen__ le felicita por su agudeza visual, pero le advierte que las posibilidades que puede presentar un cuadro de texto, son muchísimas. Dependiendo del uso que se hace de sus propiedades su comportamiento puede cambiar considerablemente e incluso, si no se utiliza adecuadamente puede llegar a confundir al usuario. Finalmente y después de varios ejemplos que __Carmen__ le explica detenidamente, __Víctor__ se percata de las múltiples utilizaciones de este componente y se convence de que también necesita estudiar a fondo este componente.*

En la unidad anterior ya hicimos una introducción a la mayoría de los componentes Swing, y entre ellos mencionamos los **cuadros de texto, que para Swing vienen implementados en Java por la clase `JtextField`**.

Incluso los hemos utilizado ya en alguna de las aplicaciones de ejemplo. Es natural, ya que **los cuadros de texto son una de las formas más simples de conseguir interactividad con el usuario**, y seguro que tú lo has visto y utilizado infinidad de veces, al utilizar casi cualquier aplicación.

Son numerosas las ocasiones en que queremos que el usuario de nuestra aplicación introduzca algún texto o un dato. Y frecuentemente la aplicación recogerá ese dato en

un cuadro de texto.

Pero, ¿cuáles son los métodos que debemos tener en cuenta para manejar los campos de texto y cuales son las características fundamentales que debemos considerar?

Contestar a esa pregunta es uno de los empeños de este apartado y sus subapartados.

En primer lugar, debemos saber cómo insertar un cuadro de texto en una ventana, y cuáles son las propiedades fundamentales que normalmente modificaremos desde el diseñador de NetBeans.

Para insertar un campo de texto, el procedimiento es sencillo. **Basta seleccionar el botón correspondiente a `JtextField` en la paleta de componentes, en el diseñador, y pinchar sobre el área de diseño encima del panel en el que queremos situar ese campo de texto**. El tamaño y el lugar en el que se sitúe, como hemos visto antes, dependerá del Layout elegido para ese panel. En la demo que hay a continuación, lo verás con más detalle.

2.1. Propiedades de un cuadro de texto

Propiedades de un cuadro de texto

¿Podemos usar los campos de texto si más, una vez insertados, o será necesario establecer algunas propiedades para dotarlos de **interactividad** y manejar su aspecto?

Al insertar el cuadro de texto en la ventana, ya podemos escribir texto en él. Pero una cosa distinta es que ese texto tenga el **formato** deseado, que el cuadro mismo tenga el aspecto deseado, y que podamos "recoger" el texto introducido para que la aplicación haga algo con él. Para conseguir esto, debemos establecer valores adecuados en algunas de las propiedades que define la clase `JtextField`.

La **lista de propiedades** es larga. Si quieres ver las principales propiedades asociadas a `JtextField`, y los métodos que permiten modificarlas, puedes utilizar la aplicación siguiente, recuerda que tienes un enlace a un archivo PDF por si prefieres consultar todas las propiedades juntas o imprimirlas. Algunos de esos métodos están definidos en la propia clase, y otros son heredados de otras clases. Por ejemplo, el método `setBackground()` es heredado de la clase `Jcomponent`. Para consultar la lista completa de métodos, tanto propios como heredados, de la clase `JtextField`, debes consultar la documentación de la API de Java.

Visualiza las propiedades de los cuadros de texto

Por ejemplo, el aspecto del cuadro de texto, con la propiedad `opaque = false`, `enable = true` y `editable = true`, y situado encima de un botón, sería algo así como lo que muestra la imagen siguiente:

Relativas al Layout, aparecerán distintas propiedades dependiendo del Layout que se haya seleccionado, que permitirán establecer la posición en la que estará el `JtextField` para ese Layout. Para el caso concreto de `Null Layout` serán:

Propiedad que `NullLayout` asocia al componente `JTextField`	Utilidad de la propiedad	Métodos asociados a la propiedad

`X`	Establece la posición del componente, concretamente el valor de la coordenada x de la esquina superior izquierda del cuadro de texto. (la esquina superior izquierda del panel contenedor, tiene coordenada (0,0), con x aumentando hacia la derecha, e y aumentando hacia abajo. En el ejemplo se ha fijado su valor a 150	En este caso hay un único método para fijar las cuatro propiedades simultáneamente:
		`setBounds()`
		Ejemplo de uso
		`jTextField1.setBounds(150, 30, 200, 25);`

`Y`	Establece la posición del componente, concretamente el valor de la coordenada y de la esquina superior izquierda del cuadro de texto. En el ejemplo se ha fijado su valor a 30	
`Width`	Establece el ancho del cuadro de texto. En el ejemplo se ha fijado su valor a 200.	
`Height`	Establece el alto del cuadro de texto En el ejemplo se ha fijado su valor a 25.	

En la imagen siguiente se resalta el significado de cada una de las propiedades que establece `Null Layout` para el cuadro de texto `jTextField1`.

Para el caso de `BorderLayout`, sin embargo será la propiedad `Direction`, que indicará si el campo de texto va a estar al norte, al sur, al este, al oeste o en el centro.

DEMO: Mira cómo se modifican las propiedades de texto

Ten en cuenta que para todos los componentes Swing, **cualquier propiedad de ese componente suele tener un método set y un método get asociado, que permiten respectivamente establecer un valor para una propiedad o consultar el valor que tiene asignado**. Por ejemplo, para la propiedad text de `JtextField` están los métodos `setText()` y `getText()` que hemos indicado en las tablas anteriores. También debes tener en cuenta que la tabla anterior no es ni mucho menos exhaustiva, y que la clase `JtextField` tiene muchos más métodos y propiedades asociadas. Pero para eso está la documentación de la API de Java. Allí podrás consultar con detalle la lista de métodos y propiedades de `JtextField` y de cualquier otro componente Swing.

Autoevaluación

2.2. Escritura y borrado en cuadros de texto

Escritura y borrado en cuadros de texto

Como ya hemos indicado, los cuadros de texto suelen usarse para que el usuario introduzca algún dato de entrada a la aplicación, o para mostrar algún tipo de información que la aplicación ofrece al usuario.

¿Pero cómo podemos hacer que nuestra aplicación escriba el texto deseado en el cuadro de texto?

En realidad, ya hemos contestado a esa pregunta en el apartado anterior: **Modificando el valor de su propiedad texto mediante el método `setText()`, al que se le pasa como argumento el `String` que queremos escribir en el campo de texto.**

¿Y cómo podemos saber lo que el usuario ha escrito en el campo de texto?

Recogiendo el valor de su propiedad texto mediante el método `getText()`, que devuelve un `String` , correspondiente al texto que en ese momento contenga el campo de texto.

Otro aspecto importante a destacar es la forma de **identificar cada cuadro de texto** en la ventana, de forma que el usuario sepa fácilmente el tipo de información que se va a introducir o que se está mostrando en cada momento en ese cuadro de texto. También es algo conocido por haberlo usado anteriormente:

Cada cuadro de texto suele ir acompañado de una etiqueta (`Jlabel`) que se pone delante del mismo para identificar su utilidad, y para que el usuario pueda interpretar adecuadamente el significado de los datos que aparecen en el cuadro de texto.

Por ejemplo, en la aplicación de la ecuación de segundo grado, tenemos tres campos de texto para recibir el valor de los coeficientes de la ecuación, cada uno de ellos con una etiqueta delante que indica de qué coeficiente se trata (a, b o c). Hay otros dos campos de texto para las soluciones, cada uno de ellos con otra etiqueta que indica la solución de la que se trata (x_1 o x_2)

Esta asociación entre etiqueta y campo de texto es tan importante, que incluso **existe un método `setLabelFor()`, para `Jlabel`, cuya finalidad es establecer un vínculo entre una etiqueta y un cuadro de texto**, de forma que se pueda usar para que se active el campo de texto a través del mnemónico de la etiqueta.

(Si por ejemplo, la etiqueta tiene asociado en la propiedad **`displayedMnemonic`** como mnemónico la letra b, que aparece subrayada en la etiqueta, para usar el campo de texto asociado a esa etiqueta podremos pulsar **Alt+b**, en vez de usar el ratón para acceder al campo de texto.)

El método `setLabelFor()` es usado para mejorar la accesibilidad de la aplicación.

2.3. Formatear el cuadro de texto con NumberFormat

Formatear el cuadro de texto con NumberFormat

¿Qué pasa si en un cuadro de texto queremos mostrar el resultado de un cálculo numérico, y ese resultado tiene más cifras de las que caben en el espacio que hemos establecido como tamaño del cuadro de texto?

En esa situación es necesario darle un determinado formato al contenido de un campo de texto, y usar la clase `NumberFormat` para ello es bastante útil.

Por ejemplo, en el programa para calcular las soluciones de segundo grado, hay ocasiones en las que la solución tiene un número bastante grande de cifras decimales.

1. Como el tamaño o longitud que destinamos al cuadro de texto es limitado, el efecto es que no se ven todas las cifras de la solución, y probablemente sólo veremos las últimas cifras de la solución, y no las primeras, que son las más significativas.
2. Para ver esas primeras cifras tendríamos que situarnos en ese cuadro de texto y con las flechas del cursor "retroceder" hasta el comienzo del número.
3. De esta forma, el cuadro de texto sería como una ventana a través de la cual sólo vemos un trozo del valor que almacena, y a lo más que podemos llegar es a desplazar ese contenido por el cuadro de texto para ver la parte que más nos interesa.
4. Esto no suele ser práctico. Si hay muchas cifras decimales en el resultado, por ejemplo, lo normal es que nos interesen las dos o tres primeras, y las demás son despreciables. Con una precisión de 3 cifras decimales (milésimas) puede ser suficiente, y cualquier resultado que tenga más cifras se mostrará sólo con las 3 necesarias.

En realidad le damos el formato al texto que ponemos como contenido, no al propio cuadro de texto. Lo que ocurre es que esto cobra especial sentido cuando el cuadro de texto nos limita el tamaño de texto que podemos mostrar.

En el ejemplo de la ecuación de segundo grado, el código necesario para asignar el formato adecuadamente, lo vemos en el siguiente cuadro.

```
        /* Debemos importar la clase NumberFormat, para poder usarla en nuestro programa*/
 import java.text.NumberFormat;
//...

private void jB_CalcularActionPerformed(java.awt.event.ActionEvent evt) {
//...

/* En el método asociado al botón calcular, que también se encarga de escribir los resultados
* del cálculo en los campos de texto de las soluciones, nos encargamos de que previamente
* les haya asignado a los valores a mostrar el formato deseado.
*/
jTF_SolX1.setText(""+ formateadorNumerico.format(x1) );
jTF_SolX2.setText(""+ formateadorNumerico.format(x2) );
//...
public static void main(String args[]) {
 //...

 /* Dentro del método main(), debemos hacer lo siguiente:
  * Inicializamos el formateador numérico y establecemos que el formato que define consiste en
  * tener un mínimo de una cifra decimal y un máximo de 4, redondeándose el resultado si es
  * preciso para aquellos números a los que se les aplique este formato. En nuestro ejemplo,
  * serán las soluciones.
  */
                formateadorNumerico=NumberFormat.getInstance();
                formateadorNumerico.setMaximumFractionDigits(4);
                formateadorNumerico.setMinimumFractionDigits(1);
      //...
    }
//...
```

Para formatear un número, al crear una instancia del formateador numérico puede especificarse como parámetro del método **`getInstance()`** un conjunto de características "**`Locale`** ".

`Locale` es una clase que establece características locales para un país o idioma, tales como el carácter a usar para coma decimal o para separación de miles, etc.

Inexplicablemente, la clase **`Locale`** tiene establecidas características específicas para multitud de países e idiomas, entre los que figuran Canadá, Canadá francófona, China, Francia, Reino Unido, Alemania, Italia, Japón, Corea, Taiwán o Estados Unidos, pero no existen características particulares establecidas para España como país, ni para el español como idioma. Afortunadamente para nosotros, en Francia tienen básicamente las mismas convenciones que nosotros, por lo que podemos usar algo como lo que sigue:

```
NumberFormat formateadorNumerico = NumberFormat.getInstance(Locale.FRENCH);
```

Esto establece el punto como separador de miles, y la coma como separador decimal, entre otras características.

Una vez establecido el formateador numérico, hay una serie de métodos que se pueden usar sobre él. Ya hemos visto que uno de ellos es el método **`format()`**, que da el formato establecido para el formateador numérico al número pasado como argumento del método **`format()`**.

Otro método que merece la pena ser destacado es el método **`setParseIntegerOnly (boolean value)`**.

Si se le pasa el argumento **`true`**, establece que este formateador sólo va a formatear la parte entera del número. Cualquier cifra decimal, será ignorada al dar el formato. Así, 2343.54 se formateará como 2343 (suponemos que en el **`Locale`** establecido la coma decimal se representa con punto)

3. Etiquetas

Etiquetas

*Parece sencillo construir una ventana para entrada y salida de datos con componentes **`JTextField`**. Pero **Carmen** insiste en que es imprescindible el uso de componentes JLabel, con el fin de aclarar determinados conceptos al usuario.*
*Esto ocurre normalmente porque no todos los usuarios tienen el mismo nivel de conocimiento ni las mismas destrezas en lo que se refiere a aplicaciones informáticas. Le hace ver a **Víctor** que el uso de las etiquetas es básico para conseguir un aspecto agradable de la ventana, combinando colores, estilos y tamaños de letra. Pero además las etiquetas pueden mejorar la funcionalidad de la aplicación con textos de ayuda y guía para aclarar al usuario los resultados que muestra la aplicación o aportar información sobre los datos de entrada.*

A estas alturas, poco queda por decir de las etiquetas. Como recordarás, las hemos usado frecuentemente en los ejemplos de la unidad 17 y en los de esta misma unidad. Sólo recordaremos que las etiquetas, **para Swing vienen implementadas en Java por la clase `Jlabel`**.

Las principales propiedades que se pueden asociar a una etiqueta te las resumimos a continuación. La mayoría de ellas son similares a las vistas para el campo de texto, y no vamos a entrar en más detalles. Te destacamos junto al nombre de la propiedad las que realmente son novedosas, por no estar disponibles en **`JtextField`**.

Visualiza las propiedades de las etiquetas con NetBeans

Autoevaluación

4. Casillas de verificación, botones de radio y grupos de botones

Casillas de verificación, botones de radio y grupos de botones

***José** le cuenta a **Víctor** que en una ocasión fue el propio cliente quien le sugirió el diseño de las ventanas. Se trataba de una aplicación con la que se pretendía simular el funcionamiento de una máquina con interruptores y varios conjuntos de sensores que accionaba el operario. El cliente decidió hacerle un dibujo de los controles y rápidamente **José** lo asoció a controles **`JCheckBox`** y **`JRadioButton`**. En aquella ocasión el resultado fue muy satisfactorio para el cliente, especialmente por su parecido con el panel de control original y por la facilidad de uso.*

¿Recuerdas de la unidad anterior qué tipo de componente son las casillas de verificación y los botones de radio?

Una imagen vale más que mil palabras:

Las casillas de verificación en Swing están implementadas para Java por la clase **`JCheckBox`**, y los botones de radio o de opción por la clase **`JRadioButton`**. Los grupos de botones, por la clase **`JButtonGroup`**.

La funcionalidad de estos dos tipos de componentes es en realidad la misma.

- **Ambos tienen dos "estados**": Seleccionados o no seleccionados. (marcados o no marcados)
- **Ambos se marcan o desmarcan usando el método `setSelected(boolean estado)`**, que establece el valor para su propiedad **`selected`**. (el estado toma el valor **`true`** para seleccionado y **`false`** para no seleccionado)
- **A ambos le podemos preguntar si están seleccionados o no mediante el método `isSelected()`**, que devuelve **`true`** si el componente está seleccionado y **`false`** si no lo está.
- **Para ambos podemos asociar un icono distinto para el estado de seleccionado y el de no seleccionado**.

¿Y para qué queremos entonces tener dos componentes distintos?

La razón es **que por convenio, se les asigna un significado distinto a cada tipo de componente**.

- **Cuando en un contenedor aparezcan agrupadas varias casillas de verificación, entenderemos** que cada una de ellas se podrá marcar o desmarcar con independencia de las demás. Así podrán estar todas seleccionadas, o sólo algunas de ellas, o ninguna. **Son por tanto, opciones compatibles entre sí**.
  Por ejemplo, si miramos la imagen anterior, no hay ningún impedimento para que un trabajador no pueda ser fijo en la empresa y al mismo tiempo que tenga familiares en la empresa, o que resida en el municipio (o ninguna de ellas, o todas ellas).

- **Cuando en un contenedor aparezcan agrupadas varios botones de radio (o de opción), entenderemos** que no son opciones independientes, sino que sólo uno de ellos podrá estar activo en cada momento, y necesariamente uno debe estar activo. **Son por tanto opciones excluyentes entre sí.**
  Por ejemplo, si miramos la imagen anterior, si un trabajador está casado, su estado civil no es divorciado, ni soltero ni con pareja de hecho. Una y sólo una de estas opciones puede cumplirse a un tiempo.

Sin embargo, **seguir ese convenio, que es altamente recomendable, por estar ampliamente aceptado y difundido, no es algo que se haga automáticamente, sino que es el programador el que debe preocuparse de que se cumpla**.

Nada impide por ejemplo, adoptar justamente el criterio contrario para asignarles ese significado, o incluir casillas de verificación y botones de radio que tengan exactamente el mismo significado, si ese fuera nuestro deseo. Pero no debemos hacerlo. Hay que seguir el criterio de sorprender al usuario lo menos posible. Si todas las aplicaciones siguen un convenio al respecto, cuando el usuario vea nuestra aplicación lo normal es que piense que también lo sigue y no hacerlo seguramente le va a crear problemas y disgustos.

4.1. Cómo hacer que se cumpla el convenio

Cómo hacer que se cumpla el convenio

¿Cómo hacer entonces que se cumpla ese convenio?

Mediante el uso de grupos de botones, implementados para Swing en Java por la clase `JButtonGroup`

Un objeto de tipo **`JButtonGroup`** no es un objeto con una representación visible. Al añadirlo al diseño del interfaz, de hecho se guarda dentro del grupo de "otros componentes" en el diagrama de estructura del IDE (pestaña **Inspector** de la parte inferior izquierda de la ventana de NetBeans, en la que se ve la representación visual de pertenencia de los componentes a sus respectivos contenedores.)

- **Cuando hemos añadido un nuevo grupo de botones (objeto `JButtonGroup`), podemos seleccionarlo dentro de la propiedad `buttonGroup` del componente en cuestión, que normalmente será un `JRadioButton`**.
- **Si varios componentes (botones de opción `JRadioButton`) pertenecen al mismo grupo de botones, sólo uno de ellos podrá estar activo al mismo tiempo**.
- **Si seleccionamos un componente distinto del grupo de botones, el anterior se desmarcará, quedando marcado sólo el último seleccionado**.
- Aunque nada impide que varios componentes pertenecientes a distintos paneles o incluso a distintas ventanas puedan pertenecer al mismo grupo de botones, **lo**

usual, y lo deseable, es que todas aquellas opciones que sean dependientes entre sí se encuentren situadas visualmente cercanas y agrupadas, para que se aprecie mejor su mutua dependencia, y el hecho de que una y sólo una de ellas puede estar activa en cada momento.

- **Aunque nada impide meter varias casillas de verificación en un mismo grupo de botones (`buttonGroup`) y utilizarlas como si fueran botones de Opción, se desaconseja totalmente hacerlo.**

Entre la lista de propiedades de `JRadioButton` y `JCheckBox`, muchas son ya conocidas, al tener el mismo nombre y significado que para los cuadros de texto o las etiquetas. Por eso no las incluimos en la siguiente tabla, en la que nos limitamos a poner la lista de las propiedades más destacadas como novedosas en ambas clases:

Lista de propiedades de las casillas de verificación y los botones en NetBeans

Autoevaluación

5. Botones de acción y botones On/Off

Botones de acción y botones On/Off

*Frecuentemente tras una jornada de trabajo, los empleados de SI Andalucía, suelen quedar en la cafetería de la esquina. El tema de conversación se centra, en esta ocasión, en los componentes de las ventanas para programación visual con Java. Ante las quejas de Víctor del excesivo número de controles y las numerosas propiedades de cada uno, **José** dice que todos los controles tienen su utilidad cuando se usan de forma adecuada.*
***Carmen** opina que raramente se usan todos y que sus preferidos son los botones. Dice que desde su punto de vista son los más intuitivos y fáciles de usar. A cualquier usuario le basta con hacer un clic de ratón para utilizarlos correctamente, además son fácilmente identificables en una ventana y no hay que explicar su uso, prácticamente a nadie. **José** argumenta que, efectivamente tiene razón, pero que cada control es eficaz y útil en el momento adecuado.*
*Finalmente **Víctor**, apoyando a **Carmen**, dice que salvo quizás las etiquetas y los campos de texto, el resto de los controles pueden ser simulados con botones y no al contrario, aunque también reconoce que las ventanas son más atractivas y agradables cuando se emplean los componentes adecuados.*

Habrás notado que es frecuente que las aplicaciones incluyan botones que al ser pulsados realicen alguna tarea, tal como abrir una nueva ventana, hacer un cálculo con los datos que aparecen en una ventana, dar de alta un cliente, borrar los datos de la ventana, aceptar o cancelar las modificaciones que se han hecho en los datos, buscar una persona dentro de una lista, etc.

Esos botones reciben el nombre de botones de Acción, justamente porque al pulsarlos realizan una acción, y para Swing, la clase que los implementa en Java es `JButton`.

La lista de propiedades para `JButton` es muy similar a la de `JLabel` expuesta en apartados anteriores, coincidiendo la mayoría de las propiedades en nombre y en significado. Por eso no merece la pena reproducirla aquí, ya que sería bastante repetitivo, y te remitimos a consultar la documentación de la API de Java para más detalles.

Pero hay un tipo especial de botones, que funcionan como **interruptores de dos posiciones** o estados (pulsados-on, no pulsados-off). Esos botones especiales reciben el nombre de botones on/off o más frecuentemente `JToggleButton`.

En realidad, tanto `JCheckBox` como `JRadioButton` son subclases de `JToggleButton`.

Como cabe esperar después de lo comentado en el párrafo anterior, para manejar un `JToggleButton`, dispondremos de los métodos `isSelected()` y `setSelected()`.

6. Listas y Listas desplegables

Listas y Listas desplegables

*Cuando llega **María** y se interesa por el tema de conversación, opina que sus preferidas son las listas desplegables. Primero porque facilitan la selección de opciones, de forma muy rápida y también porque permiten incluir un gran número de datos en el mínimo espacio posible de una ventana. Dice que hay lenguajes en los que las listas pueden visualizar de uno a varios de sus elementos (incluso todos), lo que le aporta un mayor número de posibilidades. Además presentan una gran variedad de métodos a la hora de programar acciones.*
*De inmediato **Víctor** se da cuenta de que aún no ha visto este tipo de control que ha despertado su interés con las palabras de **María**, pero sin duda será su principal objetivo el próximo día.*

Frecuentemente te encontrarás con aplicaciones en las que se pide que el usuario introduzca un dato, pero ese dato no puede tomar el valor que se quiera, sino que se sabe que debe ser introducido uno de entre una lista de valores válidos.

Podríamos introducir el valor mediante un cuadro de texto y posteriormente comprobar que el valor introducido por el usuario efectivamente es uno de los disponibles en la lista, de forma que si no lo es se rechace y se vuelva a introducir otro valor.

¿No sería mucho más fácil, directo y seguro que el usuario no pudiera meter más que un valor correcto? Es más, ¿no sería más cómodo que pudiera ver todos los valores posibles y que sólo tuviera que escoger con el ratón el que desea, sin tener que teclearlo siquiera?

Evidentemente sí, y el componente que nos permite hacerlo es una lista o una **lista desplegable**.

Las listas vienen implementadas para Swing en Java por la clase `JList` y las listas desplegables por la clase `JComboBox`.

6.1. Listas (JList)

Listas (JList)

El aspecto de una lista `JList` es el siguiente:

Para las listas `JList`, un modelo separado (`ListModel`) representa los contenidos de la lista. Esto quiere decir que los datos de la lista se guardan en una estructura de datos independiente, que llamamos modelo de la lista. Es fácil mostrar en una lista los elementos de un array o vector de objetos, usando un constructor de `JList` que cree una instancia de `ListModel` automáticamente a partir de ese array.

En la siguiente tabla aparecen trozos de código que ejemplifican las posibilidades de crear un `Jlist` a partir de un array de datos. Ese array puede ser de cualquier tipo de objetos, incluso de la clase genérica `Object`.

```
        /* Ejemplo1: Como crear una lista JList que muestra los String contenidos en el array
        * datos[]
        */
    String[] datos = {"Uno", "Dos", "Tres", "Cuatro"};
```

```
 JList listaDatos = new JList(datos);

/* El valor de la propiedad  model  de  JList  es un objeto que proporciona una visión
 * sólo de lectura del array datos[], y se construye automáticamente. El método getModel()
 * permite recoger ese modelo en forma de Vector de objetos, y usar con él métodos de la
 * clase Vector, como getElementAt(i), que proporciona el elemento de la posición i del
 * Vector.
 */
for(int i = 0; i < listaDatos. getModel() . getSize() ; i++) {
    System.out.println(listaDatos. getModel().getElementAt(i) );
}

 /* Ejemplo 2: Cómo crear una lista JList que muestre las superclases de la
  * clase Jlist, y almacenar esa lista de nombres de clase en un objeto de la clase
  * java.util.Vector. La clase Vector permite crear arrays genéricos de cualquier tipo de
  * objeto, incluso Object.
  * Puedes ver más información sobre la clase Vector y sus métodos asociados en la
  * documentación de la API de Java.
  */
Vector superClases = new Vector();
/* Obtenemos el objeto Class correspondiente a JList */
Class claseRaiz = javax.swing.JList.class;

/* Mientras que tengamos una superclase distinta de null, añadimos la superclase al Vector
 * usando el método addElement() de la clase Vector, y apuntamos la referencia clase a al
 * superclase, devuelta por el método getSuperclass(), para la siguiente iteración
 */
for(Class clase = claseRaiz; clase != null; clase = clase.getSuperclass()) {
    superClases.addElement(clase);
}
/* Hacemos una lista de tipo JList pasándole al constructor el objeto Vector llamado
 * superClases que acabamos de crear, para que la lista tenga como elementos los de ese
 * Vector.
 */
 JList listaClases = new Jlist(superClases);
```

`JList` no soporta desplazamiento vertical (scrolling) con barra de desplazamiento por sí mismo. Para disponer de esta útil posibilidad, debemos incluir el componente `JList` dentro de un `JScrollPane`, que sí facilita esta característica.

Un ejemplo, suponiendo que **`listaDatos`** es, como en el código anterior, un **`JList`** formado a partir de un array de String llamado datos:

```
JScrollPane panelDesplazamiento= new JscrollPane(listaDatos);
```

Dos sentencias equivalentes a la anterior son:

```
JScrollPane panelDesplazamiento = new JScrollPane();
 panelDesplazamiento.getViewport().setView(listaDatos);
```

Pero las listas **`JList`** no se usan sólo para mostrar la lista de datos o elementos, si no para pedirle al usuario que seleccione alguno de esos datos.

- **En un `JList`, el usuario puede seleccionar un solo elemento, o varios elementos simultáneamente, que a su vez pueden formar un bloque de elementos contiguos o varios bloques no contiguos.**
- Cada elemento de la lista también suele llamarse ítem. Las posibilidades de hacerlo de una forma o de otra vienen establecidas por la propiedad **`selectionMode`**
- Los valores posibles para la propiedad selectionMode para cada una de esas opciones son las siguientes constantes de clase del interface **`ListSelectionModel`**:
  - **`MULTIPLE_INTERVAL_SELECTION`**: Es el valor por defecto**. Permite seleccionar múltiples intervalos**, manteniendo pulsada la tecla CTRL mientras se seleccionan con el ratón uno a uno, o la tecla de mayúsculas, mientras se pulsa el primer elemento y el último de un intervalo.
  - **`SINGLE_INTERVAL_SELECTION`**: **Permite seleccionar un único intervalo**, manteniendo pulsada la tecla mayúsculas mientras se selecciona el primer y último elemento del intervalo.
  - **`SINGLE_SELECTION`**: Permite **seleccionar cada vez un único elemento** de la lista.

En el ejemplo del final del apartado, hemos usado **`SINGLE_SELECTION`**, pero tú puedes ejecutar el programa cambiando el valor de la propiedad para ver el comportamiento en cada caso.

`JList` proporciona varias propiedades para manejar la selección, que tienen asociados sus correspondientes métodos:

- El método **`setSelectedIndex`** **permite establecer el valor de la propiedad `selectedIndex`, es decir, cual es el índice del elemento seleccionado. (Selecciona el elemento del índice que se le pasa como argumento)** Debemos recordar que los datos se almacenan en un modelo que no es otra cosa que un array, por lo que tiene sentido hablar de índice seleccionado.
- Análogamente, disponemos de un método **`getSelectedIndex()`** que **nos permite saber cual es el índice del elemento seleccionado.**

Pero además de conocer el índice, lo más seguro es que me interese conocer el mismo elemento seleccionado o ítem seleccionado. Para ello:

- El método **`getSelectecValue()`** **devuelve el objeto seleccionado, de tipo `Object`, sobre el que tendremos que aplicar un casting explícito para obtener el elemento que realmente contiene la lista (por ejemplo un `String`).** Fíjate que la potencia de usar como modelo un array de **`Object`**, es que en el **`JList`** podemos mostrar realmente cualquier cosa, como por ejemplo una imagen.
- El método **`setSelectedValue()`** que nos **permite establecer cual es el elemento que va a estar seleccionado**.

Para el caso de que se permitan selecciones múltiples, contamos con los métodos análogos a los anteriores:

- **`setSelectedIndices()`, al que se le pasa como argumento un array de enteros que representa los índices a seleccionar.**
- **`getSelectedIndices()`, que devuelve un array de enteros que representa los índices de los elementos o ítems que en ese momento están seleccionados en el `JList`.**
- **`getSelectedValues()`, que devuelve un array de `Object` con los elementos seleccionados en ese momento en el `JList`.**

Un ejemplo de cómo se pueden usar estos métodos:

```
String[] datos = {"Uno", "Dos", "Tres", "Cuatro"};

JList listaDatos = new JList(datos);

listaDatos. setSelectedIndex(1);         // selecciona "Dos"

listadatos.getSelectedValue();         // devuelve "Dos"
```

El contenido de un `JList` puede ser dinámico, es decir, los elementos de la lista pueden cambiar de valor y el número de elementos en la lista puede cambiar también, después de que el `JList` ha sido creado. Los cambios en el modelo del **`JList`** (recuerda que el modelo no es más que el array que contiene efectivamente los datos) son observados por un escuchador de eventos o listener de tipo **`swing.event.ListDataListener`**, que tendremos que implementar, y que observa eventos de tipo **`swing.event.ListDataEvent`**, que identifica el rango de índices que han sido modificados, añadidos o borrados. Veremos lo relacionado con escuchadores de eventos o listener en el apartado 7.1. de esta misma unidad.

También es posible usar el método **`setPrototypeCellValue()`** para establecer un tamaño fijo para todas las celdas, de forma que no se tenga que calcular el tamaño de cada celda.

```
        /* Este modelo de lista tiene sobre 2 elevado a 16 elementos.  Para que te diviertas
     * desplazando la lista. En este caso, el array lo definimos como de tipo ListModel, que
     * realmente implementa un objeto de la clase Vector, que proporciona el método getElementAt()
     */
     ListModel arrayDatosEnorme = new AbstractListModel() {
         public int getSize() { return Short.MAX_VALUE; }
         public Object getElementAt(int index) { return "índice " + index; }
     };
     JList listaDatosEnorme = new JList(arrayDatosEnorme);
    /*No queremos que la implementación de JList se tenga que poner a calcular la anchura
     * y altura de todas las celdas de la lista, y por eso le damos un String que es tan
     * grande como necesitemos para la más grande de las celdas, y que se usa para
     * calcular los valores para las propiedades fixedCellWidth y fixedCellHeight, que
     * delimitan el tamaño fijo de todas las celdas de la lista (cada celda contiene un
     * elemento)
     */
      listaDatosEnorme. setPrototypeCellValue("Index 1234567890");

    /* El valor concreto pasado como parámetro es lo de menos. En este caso se ha usado sólo como
     * modelo del tamaño que se necesita para el mayor ítem posible en la lista.
     */
```

DEMO: Mira cómo crear una lista

Autoevaluación

6.2. Listas Desplegables (JComboBox)

Listas Desplegables (JComboBox)

¿Hay grandes diferencias entre una lista de tipo **`JList`** y una lista desplegable de tipo **`JComboBox`**?

Realmente la lista desplegable es una mezcla entre un campo de texto editable y una lista. Si la propiedad **`editable`** de la lista desplegable la fijamos a **`true`** (verdadero), el usuario, además de poder seleccionar uno de los valores de la lista que se despliega al pulsar el botón de la flecha hacia abajo, dispondrá de la posibilidad de teclear directamente un valor en el campo de texto.

La propiedad **`editable`** del **`JComboBox`** se establece mediante el método **`setEditable()`** y se comprueba con el método **`isEditable()`**. Además de los ya mencionados, la clase **`JComboBox`** nos ofrece otra serie de métodos, que tienen nombres y funcionalidades similares a los de la clase **`JList`**.

También hay algunas diferencias visuales entre un **`JComboBox`** y un **`JList`**, es decir, las diferencias afectan más a la parte de la vista del componente. (Recuerda que en la unidad anterior hablamos de que cada componente tiene un modelo, que es el que contiene los datos, una vista, que establece la forma de dibujar ese componente en el contenedor, y un controlador, que establece lo que hay que hacer cuando se interactúa con ese componente)

En el caso de **`JComboBox`**, el modelo es bastante similar al de **`JList`**, un array de objetos (cualquier tipo de objetos, incluyendo objetos genéricos de la clase **`Object`**). En la mayoría de los casos esos objetos serán **`String`**, que a fin de cuentas sabemos que también son un subtipo de **`Object`**

La vista de una lista desplegable es similar a un cuadro de texto, con un botón con una flecha hacia abajo al final, que cuando se pulsa modifica el aspecto del componente, mostrando un cuadro similar a **`JList`** donde se ven los elementos de la lista.

La imagen siguiente nos muestra el aspecto de una lista desplegable cerrada y desplegada.

La tabla siguiente muestra de forma resumida el nombre y la utilidad de algunos de los métodos más usados. La mayoría están relacionados con una propiedad de las que se pueden seleccionar en NetBeans desde el diseñador gráfico (Form Editor), pero otros no directamente. Para consultar la lista de parámetros que admiten y los valores que devuelven estos métodos, deberás consultar la documentación de la API de Java.

Visualiza las propiedades y métodos de las listas desplegables

A continuación puedes ver el código de un ejemplo en el que aparece una lista desplegable con los nombres de los trabajadores de una empresa, que se pueden seleccionar de una lista **`JList`**, y asignarles categorías profesionales seleccionándolas de una lista desplegable (**`JComboBox`**). En el ejemplo los valores de ambas listas son estáticos, y se han establecido al hacer el programa. En la unidad 20, cuando se vea el acceso a bases de datos, tendremos ocasión de comprobar que ambas listas pueden construirse dinámicamente, tomando sus elementos de una consulta a una tabla de una base de datos, por ejemplo.

Descarga el proyecto EjemploJListYJComboBox

DEMO: Mira cómo crear una lista desplegable

DEMO: Y aquí, mira el código y sus comentarios paso a paso

Visualiza el código de VentanaPrincipal.java

Autoevaluación

7. Programación guiada por eventos

Programación guiada por eventos

Entender cómo funciona la programación por eventos es algo relativamente fácil, el problema está en utilizar los eventos más adecuados en cada momento y en utilizarlos bien.
***Víctor** tiene muy claro que el evento que se asocia a un botón debe ser cuando es pulsado, pero **José** le pone en duda cuando le dice que el evento se produce cuando el botón es soltado, además le explica que otro evento sobre el botón se produce al ser enfocado con el ratón, o también al accionar una combinación de teclas asociadas. **Carmen** añade que un buen programador debe conocer todos sus componentes y los eventos asociados a cada uno de ellos, para utilizar el más adecuado a cada situación. Dice que realmente no es tan complicado, porque se repiten muchos eventos y si nos paramos un momento a pensarlo, todos ellos son predecibles y bastante lógicos.*

Casi me atrevería a decir que a estas alturas ya sabes de lo que te vamos a hablar en este apartado, ya que venimos usando los eventos desde los ejemplos de la unidad anterior. Suele ocurrir en programación, que para explicar un concepto nuevo, debes recurrir o usar conceptos que se relacionan con él, pero que aún no se han explicado. Es lo que nos ha ocurrido con los eventos y la programación guiada por eventos, que aún no se han explicado correctamente. Pero ahora es el momento de tratar este tema como debe ser. Para ello, la primera pregunta que tenemos que hacernos es:

¿Qué es un evento?

Cualquier hecho que ocurre mientras se ejecuta la aplicación. Normalmente consideramos como eventos cualquier interacción que realiza el usuario con la aplicación, como puede ser pulsar un botón con el ratón, hacer doble clic, pulsar y arrastrar, pulsar una combinación de teclas en el teclado, pasar el ratón por encima de un componente, salir el puntero de ratón de un componente, o abrir una ventana, etc.

¿Y qué es la programación guiada por eventos?

Imagina la ventana de cualquier aplicación, por ejemplo la del entorno de programación NetBeans, o la del propio navegador Web en el que estás siguiendo el curso. En esa ventana aparecen multitud de **elementos gráficos interactivos**, de forma que no hay manera de que el programador haya previsto todas las posibles entradas que se pueden producir por parte del usuario en cada momento. Con el control de flujo de programa de la programación imperativa, al que estamos acostumbrados, el programador tendría que comprobar para cada entrada o interacción producida por el usuario, de cual se trata de entre todas las posibles, usando estructuras de flujo condicional (if-then-else, switch) para ejecutar el código conveniente en cada caso. Si piensas que para cada opción del menú, para cada botón o etiqueta, para cada lista desplegable, y en suma para cada componente de la ventana, incluyendo la propia ventana, habría que comprobar todos y cada uno de los eventos posibles, nos damos cuenta de que las posibilidades son casi infinitas, y desde luego impredecibles. El problema se volvería inmanejable.

Para abordar el problema de tratar correctamente las interacciones del usuario con el interfaz gráfico de la aplicación hay que cambiar de estrategia, y la programación guiada por eventos viene en nuestra ayuda como una buena solución.

¿Cómo?

- **Para cualquier componente gráfico en la ventana, habrá un conjunto limitado de eventos que nos interese controlar**. Por ejemplo, para un botón de acción `JButton`, lo más normal es que sólo nos interese comprobar un evento, para saber si se ha pulsado con el ratón sobre él o no, o si se ha elegido mediante un atajo de teclado. En definitiva, si se ha producido el evento de "realizar la acción" asociada a ese botón. Bueno, pues en vez de preocuparnos de hacer esa comprobación desde nuestra aplicación, se actúa de otra forma:
  - **Asociamos un escuchador de eventos (Listener) apropiado para el tipo de evento que queremos comprobar**. En el ejemplo del botón de acción, le asociaríamos un `ActionListener`.
  - **El listener no es más que un programa que permanece activo, en ejecución en segundo plano, de forma paralela a nuestra aplicación, y que se encarga exclusivamente de comprobar si se ha producido sobre el componente al que se ha añadido algún evento del tipo que él sabe escuchar**. En el caso del botón de acción, el `ActionListener` se encarga de comprobar si se ha producido algún evento del tipo `ActionEvent` (evento de acción, como pulsar con el ratón o seleccionar con el atajo de teclado, por ejemplo) sobre el botón.
  - **En el momento que el listener "escucha" o intercepta un evento de ese tipo, lo captura, y pasa el control del flujo de nuestra aplicación al Controlador del componente al que está asociado** (Recuerda: todos los componentes Swing se basan en el esquema Modelo - Vista - Controlador)
  - **El controlador no es más que el código que el programador ha escrito para que se ejecute cuando se produce exactamente ese evento sobre ese componente. Este código recibe del listener además del control, un parámetro que es un objeto Evento del tipo del evento escuchado por el listener**. En el caso del botón de acción, se transfiere el control al método `actionPerformed()` definido por el programador, pasándole como parámetro el objeto evento de tipo `ActionEvent` que se ha producido. El método `actionPerformed()` contiene todas las sentencias que deben ejecutarse.
- **Cada elemento interactivo de la ventana será un componente que tendrá asociados sus propios listener adecuados al tipo de eventos que puede recibir ese elemento.**
- **Para cada tipo de evento existe un interface que especifica los métodos que hay que implementar para tratar ese evento. Así, por ejemplo, para los eventos de acción (`ActionEvent`) existe un interface (`ActionListener`) que especifica que para tratar los eventos de acción hay que implementar el método `actionPerformed()`**
- **El listener que se asocia al componente es un objeto de una clase que implemente el interface correspondiente al evento en cuestión.** Así por ejemplo, para añadir un evento de acción a un botón tendremos que hacerlo mediante la invocación para el botón del método `addActionListener()`, al que le pasaremos como argumento el listener a asociar al botón, que no será otra cosa que un objeto de una clase que implemente `ActionListener`, es decir, una clase que tenga un método `actionPerformed()` que indique las tareas a realizar cuando sobre el botón se realice una acción.
- Esto significa que todos esos listener son programas que se están ejecutando concurrentemente, de forma que todos permanecen a la espera de las acciones del usuario. **En el momento que el usuario realiza una acción, se genera el correspondiente evento, que es capturado por el correspondiente listener, que pasa el control al correspondiente Controlador, que realiza las tareas indicadas por el programador.**

En la programación visual o guiada por eventos el programador no guía el flujo mediante sentencias de control de flujo, si no que se limita a ir "sembrando" escuchadores de eventos por todos los componentes de su aplicación, o al menos en los componentes que los necesitan**, y son los eventos que va produciendo el usuario, en un orden totalmente impredecible para el programador, los que van guiando el flujo de ejecución. De ahí el nombre de programación guiada por eventos.**

Este enfoque facilita enormemente la tarea de programación, y simplifica el código, haciendo que la tarea de programar interfaces gráficas sea sumamente sencilla, en vez de resultar algo inabordable.

En el siguiente apartado se va a ver cómo añadir listener a los componentes, y como gestionar los eventos que se capturen mediante los controladores oportunos.

7.1. Asociar listener a un componente con el diseñador

Asociar listener a un componente con el diseñador

¿Resulta complicado añadir un listener a un componente, y especificar el código que debe ejecutarse dentro del controlador de ese componente?

Mediante el uso del diseñador, es sumamente sencillo añadir un listener a un componente.

- Basta con seleccionar el componente en cuestión **en el área de diseño del diseñador**, o bien **en el diagrama de estructura de la pestaña Inspector**,
- para que en la **paleta Properties** del componente, seleccionando **el panel Events**, aparezca la lista de métodos a implementar para todos los eventos posibles que podemos capturar para el componente en cuestión,
- y podamos añadir un listener que los capture, añadiendo el código necesario para el controlador.
- Lo único que debemos hacer es darle nombre al método que contiene el código que queremos que se invoque y se ejecute.

Se sigue un convenio bastante recomendable para nombrar a ese método:

El método controlador empieza con el nombre del componente al que se ha asociado el escuchador, seguido del nombre del método del interfaz que hay que implementar para ese tipo de evento.

Así, por ejemplo, en el proyecto `EjemploListener` que tienes un poco más adelante en este mismo apartado, el método que contiene el código que nosotros escribimos para el evento de pulsar el botón `jB_BotonAccion` con el ratón se llama `jB_BotonAccionActionPerformed()`, ya que el interfaz `ActionListener` nos obliga a implementar el método `actionPerformed()` para tratar los eventos de acción (`ActionEvent`). Por tanto, si hay varios botones de acción, cada uno tiene su propio método `actionPerformed()` asociado, lo cual puede inducir a error.

Para minimizar ese problema, lo único que habrá dentro de cada uno de esos métodos `actionPerformed()` es una llamada invocando al método nombrado según el convenio, que claramente identifica el componente al que está asociado y el tipo de evento que lo desencadena.

Así, en el mismo ejemplo, hemos llamado a otro método `jB_BotonAccionMouseEntered()`, ya que es el método que se ejecuta cuando sobre el componente `jB_BotonAccion` se produce un evento de tipo `mouseEntered`, es decir, que el puntero del ratón entra dentro de la zona de pantalla donde está el botón, sobrevolándolo.

Por último, también hemos llamado a otro método **`jB_BotonAccionMouseExited()`**. Seguro que te imaginas a qué se debe ese nombre.

Incluso, si el evento que queremos capturar es un **`ActionEvent`**, que implica implementar el método **`actionPerformed()`** (el primero de la lista del **panel Events**), ni siquiera es necesario recurrir a esta ventana. Basta con hacer doble clic sobre el componente o su nombre en el diagrama de estructura, para que directamente se abra el editor del código en el lugar justo para que nosotros sólo tengamos que añadir el código necesario a ejecutar por el controlador.

- Automáticamente se ha creado una clase interna anónima que implementa el interface asociado al listener para el tipo de evento,
- se ha añadido un objeto listener de ese tipo al componente,
- y se ha implementado el código de la cabecera del método controlador, para que sólo tengamos que preocuparnos de escribir las sentencias necesarias dentro del cuerpo de ese método.

El siguiente ejemplo muestra un botón al que se le han añadido dos listener, uno de acción (**`ActionListener`**) y otro de ratón (**`MouseListener`**) para tres eventos distintos (un **`ActionEvent`** y dos **`MouseEvent`** distintos). Es importante que mires el código del ejemplo, fijándote en los comentarios que se han añadido, que explican lo que hace el código.

Descarga el proyecto EjemploListener

DEMO: Mira qué ocurre al asociar varios listeners al mismo botón

7.2. Clases internas anónimas

Clases internas anónimas

Imagina que para cada componente al que le quisiéramos añadir un listener tuviésemos que crear una clase sólo para implementar el interface correspondiente, y poder crear un objeto de esa clase que implementa el listener, para poder añadírselo al componente.

¿Cuántos componentes podemos tener en una sola ventana de una aplicación que necesiten un listener?

Yo he contado más de 100 en la pantalla de mi procesador de textos, sin contar las opciones del menú, y sin contar las decenas de nuevas ventanas y cuadros de diálogo que contiene. ¿Es sensato crear varios cientos de clases adicionales en nuestra aplicación, que ya de por sí puede tener un número considerable, sólo para implementar los listener?

La respuesta es que no sólo no es sensato, es que es un disparate. El número de clases a manejar durante el desarrollo sería tan alto, que resultaría intratable para el programador.

Por eso **Java permite una sintaxis especial, que nos permite hacer una declaración de las clases necesarias incrustadas en el código de la clase en la que se quiere insertar el listener, (por eso se llaman clases internas) y sin necesidad siquiera de darle nombre a esa clase por parte del usuario (por eso se llaman clases anónimas), aunque sí será nombrada por el compilador de forma automática, y generará el correspondiente archivo .class.**

En el proyecto, veremos que aparecen en la carpeta donde se guardan las clases compiladas, también las clases anónimas nombradas con el nombre de la clase que las contiene, y el símbolo $ seguido de un número, distinto para cada clase interna que se genere. Pero en realidad no nos hace falta para nada saber cómo se llaman. Las crea y las gestiona automáticamente el compilador. Ni sabemos cómo se llama esa clase anónima, ni falta que nos hace.

¿Y qué aspecto tiene el código generado por el diseñador para implementar la captura y tratamiento de eventos?

Empezaremos por el método que añade un listener a un componente. Usaremos el código generado para el botón "Asignar Categoría" (**`jB_AsignarCategoria`**) del Proyecto **`EjemploJListYJComboBox`** visto en el apartado 6.2.

```
        /* Para el botón jB_AsignarCategoria invocamos al método addActionListener(), al que se le
     * debe pasar como parámetro un objeto de una clase que implemente el interface
     * ActionListener. Pero para no aumentar excesivamente el número de clases, usamos la
     * sintaxis especial que nos permite Java para pasarle ese objeto directamente invocando al
     * constructor de esa clase, que además va a ser declarada de forma interna y anónima. Dentro
     * de los paréntesis de la llamada del método addActionListener(), metemos toda la
     * declaración de esa clase, a la que no le damos nombre, pero que sí decimos implícitamente
     * que implementa ActionListener. ¿Como? Escribimos new ActionListener(), y abrimos llaves y
     * cerramos llaves para indicar lo que contendría la definición de esa clase: la
     * implementación del método actionPerformed() para así implementar el interface
     * ActionListener.
     */
    jB_AsignarCategoria.addActionListener(new java.awt.event.ActionListener() {

          /* A su vez, el método actionPerformed recibe como parámetro el evento que se ha
           * capturado por el listener, y en su interior debe llevar definido el código que
           * queremos que se ejecute para tratar el evento, como controlador del componente.
           * Pero el diseñador, lo que hace es invocar desde su interior a un método que se
           * llama igual que el componente, en este caso el botón, seguido de ActionPerformed.
           * Este método, que en realidad es un paso innecesario, ya que podríamos escribir
           * directamente las acciones a realizar, tiene una misión de "autodocumentación" de la
           * aplicación, ya que  facilita la localización del controlador asociado a un
           *  componente, al comenzar con el nombre de ese componente . Al añadirle
           * ActionPerformed, se deja claro que ese es el método que realmente estamos
           * implementando.
           */
                public void actionPerformed(java.awt.event.ActionEvent evt) {
                    jB_AsignarCategoriaActionPerformed(evt);

                }       //Cierre del método actionPerformed()
          });       // Llave de cierre de la clase interna que se está definiendo, y paréntesis de
            // cierre del método addActionListner(), junto con el punto y coma de finalización
            // de la sentencia.

    /*... Aquí, en lugar de este comentario, en el ejemplo aparece el código relacionado
     * con otros componentes.
     */

    /*  Más adelante nos encontramos con  la definición del método que finalmente se va a ejecutar
     *  al pulsar el botón .
     * La cabecera del método así como las llaves de apertura y cierre del método son generados
     * automáticamente por el diseñador. Nosotros sólo tenemos que escribir las líneas del
     * interior para especificar las acciones a realizar cuando se capture el evento.
     */
      private void jB_AsignarCategoriaActionPerformed(java.awt.event.ActionEvent evt) {

         /* Aquí empieza el código que nosotros añadimos, que recoge el nombre de empleado
          * seleccionado en el JList, recoge la categoría seleccionada en el JComboBox, los
          * convierte a ambos en un String mediante un casting explícito, y modifica el texto de
          * la etiqueta para que muestre el empleado y la categoría seleccionada para él .
          */
            String nombreEmpleado= (String) jL_Empleados.getSelectedValue();
            String categoria= (String) jCB_Categorias.getSelectedItem();
            jL_Notificaciones.setText("Empleado: "+nombreEmpleado+". Categoría Profesional: "+
```

```
                                    categoria);
        /*Aquí termina el código que nosotros añadimos.*/
    }
```

7.3. DocumentListener para cuadros de texto.

DocumentListener para cuadros de texto

Hemos visto que se puede añadir fácilmente un listener con el diseñador, que crea una clase anónima, pero habrá veces que nos interese implementar el interface correspondiente al evento en una clase independiente en vez de implementar una clase anónima, y crear un objeto de esa clase independiente (un objeto listener del tipo adecuado), para añadírselo al correspondiente componente mediante un método **`add...()`**. Ese método **`add...()`** debe ser adecuado para añadir justamente ese tipo de listener. O sencillamente, que lo hagamos así, por conveniencia.

> **Debes tener claro que con independencia de la sintaxis que usemos (clase interna anónima o clase independiente) lo que queremos y debemos hacer es dejarle claro al compilador dónde está el método que contiene el código que hay que ejecutar cuando se presente un evento determinado sobre el componente.**

Por ejemplo, en el caso de un botón de acción:

- Lo más frecuente es añadirle un **`ActionListener`** mediante el método **`addActionListener(...)`**.
- Da lo mismo que el objeto que indiquemos dentro sea una instancia de una clase independiente o de una clase interna anónima.
- Lo importante es que ese objeto debe ser una instancia de una clase que implemente **`ActionListener`**, y que por tanto tenga un método **`actionPerformed()`**, que contiene el código que debemos ejecutar.

En el ejemplo que vas a encontrar a continuación, queremos ejemplificar dos cosas:

- **La posibilidad de gestionar eventos de documento (`DocumentEvent`) para que las modificaciones sobre un texto tengan una respuesta inmediata y automática, sin necesidad de usar ningún botón de acción. Esto se consigue con `DocumentListener`.**
- **Podemos crear una clase que implemente el interface adecuado para el tipo de evento que queremos capturar para un componente, crear un objeto de esa clase, y añadírselo como listener al componente en lugar de hacerlo mediante una clase interna anónima.**

En el ejemplo hay una clase llamada **`ImplementacionDocumentListener`** que implementa los métodos del interface **`DocumentListener`**, de la que creamos un objeto **`miPropioDocumentListener`**, que se añade al Documento del cuadro de texto **`jTF_Numero`**. Fíjate bien, es un listener añadido al documento que contiene el **`JtextField`**, y no al propio **`JtextField`**. En el proyecto, va el código comentado de las clases. Debes leer con atención los comentarios, porque explican lo que el código hace.

Descarga el proyecto DobleNumero

A continuación te ofrecemos el mismo programa que en el ejemplo anterior, pero hemos implementado el **`DocumentListener`** como una clase interna anónima. Te aconsejamos que busques dentro de las carpetas del proyecto la carpeta que guarda los ficheros ya compilados .class. Verás que además de la clase que nosotros hemos escrito en un fichero **`DobleDelNumeroBis.java`** y que hemos compilado para generar **`DobleDelNumeroBis.class`**, aparece otra clase llamada **`DobleDelNumeroBis$1.class`**. Es la clase interna que implementa el interface **`DocumentListener`**. Aunque al no darle nombre, trabajamos como si existiera sólo la clase que nosotros hemos compilado, el compilador sí nombra, gestiona y usa esta segunda clase.

Descarga el proyecto DobleNumeroConClaseInterna

Autoevaluación

8. Manejo básico de los eventos de ventana

Manejo básico de los eventos de ventana

> *Ahora es el momento de comenzar a programar como profesional y **Víctor** se siente seguro ante este reto, aunque no deja de pensar que más que por su capacidad, puede ser porque se siente arropado y protegido por sus compañeros, que han estado en todo momento observando su evolución y corrigiéndole continuamente. **José** le ha asignado una aplicación para programarla siguiendo una serie de pautas específicas de la empresa, sobre cómo usar el código y organizar el proyecto, básicamente para que pueda ser asignado a otro trabajador con el mínimo de inconvenientes en caso de que surja cualquier problema.*
> ***Víctor** tiene muy claro cuál va a ser su trabajo y cómo debe abordar el proyecto para programarlo en Java. Sabe que hacer esta aplicación por sí solo, le va a aportar más conocimientos en programación y la experiencia que necesita para dedicarse a lo que más le gusta.*

Además de poder asociar escuchadores de eventos o listener a todos los componentes de una ventana, ¿podemos añadir algún tipo de listener a la propia ventana? ¿Puede resultar eso útil?

Desde luego que puede hacerse y sí que tiene utilidad.

Por **ejemplo**, nos puede interesar tener un listener asociado a la ventana que cuente el número de ventanas del mismo tipo que hemos abierto, para evitar que podamos tener, por ejemplo, dos ventanas de altas de trabajadores abiertas simultáneamente. Es mejor dar de alta a los trabajadores de uno en uno, ya que al tener varias ventanas abiertas simultáneamente, correríamos el riesgo de equivocarnos y meter los datos en la ventana equivocada, por ejemplo.

Evidentemente, existe un tipo especial de eventos de ventana (**`WindowEvent`**) y de escuchadores apropiados para ellos (**`WindowListener`**)

El interface **`WindowListener`** obliga a implementar siete métodos, cuya utilidad se indica en la siguiente tabla:

Método del interface `WindowListener`	**Utilidad que tiene la implementación del método.**
`windowActivated(WindowEvent e)`	Este método debe implementarse con el código a ejecutar cuando la ventana pasa a ser la **ventana activa**.
`windowClosed(WindowEvent e)`	Este método debe implementarse con el código a ejecutar cuando la ventana ha sido **cerrada invocando al método `dispose()`**
`windowClosing(WindowEvent e)`	Este método debe implementarse con el código a ejecutar cuando el usuario **intenta cerrar la ventana desde el sistema de menús** de la misma
`windowDeactivated(WindowEvent e)`	Este método debe implementarse con el código a ejecutar cuando la ventana **deja de ser la ventana activa**.
`windowDeiconified(WindowEvent e)`	Este método debe implementarse con el código a ejecutar cuando la ventana **deja de estar minimizada**
`windowIconified(WindowEvent e)`	Este método debe implementarse con el código a ejecutar cuando la ventana **pasa a estar minimizada**

`windowOpened(WindowEvent e)`	Este método debe implementarse con el código a ejecutar cuando la ventana **se hace visible por primera vez**.

Hay muchos casos en los que no estaremos interesados en implementar todos los métodos. Por ejemplo, si lo único que nos interesa controlar, como hemos dicho antes, es que sólo pueda haber una ventana abierta en cada momento, basta con que en el método `windowOpened()` ponga a `true` una variable booleana llamada `ventanaAbierta` al abrir la ventana, y que el método `windowClosing()` la vuelva a poner a `false` cuando la ventana se cierra. Comprobando el valor de esa variable bandera, puedo evitar abrir una nueva ventana si ya tenía otra abierta, por ejemplo.

En este caso, realmente sólo necesito implementar 2 métodos de los 7 a los que me obliga el interface `WindowListener` luego los otros 5 tendré que implementarlos, pero vacíos sin código, es decir, abriendo llaves y cerrando llaves sin incluir ninguna sentencia en el cuerpo del método. Esto es algo bastante engorroso, tener que escribir 7 métodos cuando sólo se necesitan 2.

¿No hay alguna forma de evitarme ese esfuerzo inútil? Desde luego, la hay.

La forma de evitar implementar los métodos que no necesito del interface es usando las clases `Adapter`.

El uso de las clases `Adapter` se analizan en el siguiente apartado.

Autoevaluación

8.1. Clases Adapter

Clases Adapter

Algo mencionamos sobre las clases Adapter en el apartado 8.2 de la unidad 17, aunque sin entrar en muchos detalles.

¿Cómo nos ayudan las clases Adapter a no tener que escribir todos los métodos de un interface cuando queremos implementarlo?

- Sencillamente, **la clase Adapter nos proporciona ya una implementación para todos los métodos del interface que queremos implementar**.

Siguiendo con el ejemplo de las ventanas, para implementar los siete métodos del interface `WindowListener` disponemos de la clase `WindowAdapter`, que proporciona una implementación vacía de todos los métodos del interface, es decir, define e implementa los siete métodos abriendo llaves y cerrando llaves, sin incluir ninguna sentencia dentro.

¿Y qué funcionalidad tienen esos métodos implementados, si no incluyen ningún código?

Los métodos implementados por las clases Adapter no tienen ninguna funcionalidad, salvo la de cumplir con el requisito de implementar todos los métodos del interface, evitándonos la tediosa tarea de tener que escribir también los métodos que no necesitamos y que no nos interesan, lo cual no es poco.

¿Y qué pasa con los métodos que sí me interesan? ¿Me sirve de algo una implementación vacía, cuando yo quiero en realidad hacer algo?

- Una vez que me he ahorrado el tedioso trabajo de escribir todos los métodos que no necesito, **creo una subclase de la clase Adapter que redefina o sobrescriba los métodos que a mí me interesan con el código que sí me interesa**.
- **Al extender a la clase Adapter que implementa el interface, la nueva subclase también lo implementa.**
- **Los métodos que me interesan son redefinidos, y tienen la implementación que yo deseo,**
- **y los que no me interesan reciben la implementación vacía heredada de la superclase Adapter**.

Existen numerosas clases Adapter, y tienen más utilidad mientras mayor es el número de métodos que obliga a implementar un interface. Así, por ejemplo, **no existe una clase** `ActionAdapter` para implementar el interface `ActionListener`, puesto que tiene un solo método, que seguro que es el que queremos implementar con nuestro propio código, así que la clase **Adapter** carece de sentido.

Un ejemplo de uso de la clase `WindowAdapter`:

```
class VentanaPrincipal extends JFrame{
//...
VentanaPrincipal miVentana = new VentanaPrincipal("Ventana principal de la aplicación");
//...
 miVentana.addWindowListener( new WindowAdapter(){
      public void windowClosing(WindowEvent e)
        {
          System.exit(0);
        }
 });
//...
}
```

En el código anterior, vemos que el parecido de sintaxis en el uso de la clase Adapter con el uso de clases internas anónimas para implementar los interface Listener es bastante grande.

Ésta es la manera habitual de usar las clases Adapter, también como clases internas anónimas que extienden la clase Adapter que deseamos.

En el apartado siguiente analizamos esta analogía.

Autoevaluación

PARA SABER MÁS
En este enlace podrás profundizar un poco más sobre la clase Adapter de Java
Diseño de software con patrones

8.2. Clases Adapter como clases internas anónimas

Clases Adapter como clases internas anónimas

Vamos a comentar el código del ejemplo anterior, para que veas de qué parecidos estamos hablando:

```
class VentanaPrincipal extends JFrame{

//... código de la clase que no nos interesa para el ejemplo que nos ocupa

/* Creación de una ventana del tipo que define la clase VentanaPrincipal llamada miVentana.
```

```
 * Esta será la ventana a la que le añadiremos un escuchador de eventos de ventana
 * (WindowListener)
 */

VentanaPrincipal miVentana= new VentanaPrincipal("Ventana principal de la aplicación");

//...más código no relacionado con el ejemplo

/* Tras este comentario ejecutamos el método addWindowListener() para añadirle un escuchador
 * de eventos de ventana a la ventana miVentana.
 * Como argumento, a este método debemos darle un objeto de una clase que implemente el
 * interface.
 * Esa clase, podríamos haberla definido por separado, y haber creado un objeto de esa clase
 * para pasarlo como argumento, o podemos hacerlo en línea, al igual que vimos antes,
 * mediante una clase interna anónima.
  * Pero en este caso, mediante el operador new no estamos invocando a un interface, si no a
  * una auténtica clase, WindowAdapter. Pero al igual que ocurría con las clases internas
  * vistas en ejemplos anteriores, estamos usando una sintaxis especial que hay que
  * interpretar de la siguiente manera:
 *
 * Al método addWindowListener() le paso como argumento un objeto de una clase que  extiende  a
 * la clase WindowAdapter, que a su vez sabemos que implementa el interface WindowListener,
 * y al abrir llaves dentro de los paréntesis  de la llamada del método lo que estamos
 * haciendo es declarar aquí mismo, en línea, esa clase interna anónima que  extiende
 * WindowAdapter. Dentro de esas llaves de la clase interna, como definición, lo único que
 * tenemos que definir es aquellos métodos del interface que realmente estamos interesados
 * en implementar. Por tanto,  lo que estamos haciendo es sobrescribir o redefinir en la
  * clase interna el método para el que necesitamos un código de verdad, y los demás no
  * tenemos que ponerlos, ya que tienen una implementación heredada de la superclase .
 * En este caso el método a sobrescribir es windowClosing(), que indica lo que hay que hacer
 * cuando el usuario cierra la ventana.
 */


miVentana.addWindowListener( new WindowAdapter(){
     public void windowClosing(WindowEvent e)
       {
          /* En este caso, lo único que hay que hacer es cerrar la aplicación. */
          System.exit(0);
       }

});      // Llave de cierre de la clase interna anónima que extiende a WindowAdapter, paréntesis
   //de cierre del método addWindowListener(...), y punto y coma de finalización de la
   //sentencia de invocación al método.
//...

}       //Fin de la clase VentanaPrincipal
```

Puedes ver otro ejemplo de clase Adapter en el proyecto **`EjemploListener`** del apartado 7.1.

En este caso, queremos añadir a un botón dos **`listener`**, uno de acción para capturar eventos **`ActionEvent`** y otro de ratón, para capturar eventos **`MouseEvent`**.

En el caso de los eventos de ratón, el interface **`MouseListener`** obliga a implementar 5 métodos:

Método del interface MouseListener	Utilidad de implementar ese método
`mouseClicked(MouseEvent e)`	Se invoca cuando el botón del ratón ha sido cliqueado (pulsado y soltado) sobre un componente.
`mouseEntered(MouseEvent e)`	Se invoca cuando el puntero del ratón entra en la zona de ventana ocupada por un componente.
`mouseExited(MouseEvent e)`	Se invoca cuando el puntero del ratón sale de la zona de ventana ocupada por un componente.
`mousePressed(MouseEvent e)`	Se invoca cuando un botón del ratón se ha pulsado sobre un componente.
`mouseReleased(MouseEvent e)`	Se invoca cuando un botón del ratón se ha soltado sobre un componente.

- Pero en nuestro ejemplo, nosotros sólo necesitamos dos de esos métodos para indicar lo que hay que hacer al entrar y al salir con el puntero del ratón del componente en cuestión que en este caso es un botón.
- Por eso se recurre a la clase **`MouseAdapter`**, que proporciona implementación para todos los métodos de **`MouseListener`**,
- y usando una clase interna anónima que extiende **`MouseAdapter`**,
- sobrescribimos los métodos **`mouseEntered()`** y **`mouseExited()`**, que son los que verdaderamente nos interesan.

El código encargado de hacer esto, y que ha sido generado automáticamente por el diseñador, y al que después hemos añadido manualmente algunos comentarios explicativos, es el siguiente:

```
jB_BotonAccion.addMouseListener(new java.awt.event.MouseAdapter() {

    /* Primero de los métodos sobrescritos o redefinidos en la subclase interna anónima
     * que extiende a Mouse Adapter. Define lo que hay que hacer cuando el ratón entra en
     * el componente.
     */
     public void mouseEntered(java.awt.event.MouseEvent evt) {

         /* Lo único que hacemos es invocar al método cuyo nombre facilita la tarea de
        * identificación, al relacionarlo con el componente al que se ha añadido el
        * listener, y con el tipo de evento que se está capturando para ese componente.
        * La cabecera de la definición de ese método también es generada automáticamente
        * por el diseñador, y el cursor situado sobre el editor justamente debajo, para
        * que sólo tengamos que escribir el código que queremos que se ejecute cuando ese
        * evento se captura para ese componente.
        */

         jB_BotonAccionMouseEntered(evt);
     }
    /* Segundoo de los métodos sobrescritos o redefinidos en la subclase interna anónima
     * que extiende a Mouse Adapter. Define lo que hay que hacer cuando el ratón sale del
     * componente.
     */

     public void mouseExited(java.awt.event.MouseEvent evt) {

         /* Aquí de Nuevo lo único que se hace es invocar a un método
        * cuyo nombre facilita la tarea de identificación.
        */

         jB_BotonAccionMouseExited(evt);

     }
});
```

PARA SABER MÁS

En este enlace podrás encontrar un resumen con aplicaciones prácticas de lo visto en esta unidad.
Aplicaciones Gráficas en Java
En el siguiente enlace, y como aplicación a lo visto en esta unidad, encontrarás como manejar el control de Java denominado barra de progreso el cual es muy utilizado en las aplicaciones para indicar al usuario el estado de progreso de un proceso en concreto
Escalas y Barras de Progreso [Versión en caché]