

Introducción



***SI Ex** quiere ser una empresa de peso en lo que a programación de aplicaciones se refiere, especialmente en relación a la programación para la Web. Han llegado con ganas al mercado de servicios informáticos y eso se nota. Constantemente crece el número de clientes y de momento no se pueden quejar, tienen mucho trabajo y necesitan trabajadores especializados.*



***Carmen** intenta preparar a **Víctor** para que pueda hacerse cargo de aplicaciones él solo y de ese modo aumentar las posibilidades de servicio de la empresa. La verdad es que **Víctor** ha dado un estupendo resultado, y si al principio demostraba una actitud algo rebelde y poco flexible, ha conseguido un buen nivel de programación en tiempo record y desde luego una magnífica disposición para aprender y ser útil.*

*Sin ninguna duda el éxito de esta tarea hay que atribuírselo a **Carmen**, que ha sabido "manejarlo" con destreza y enseñarle todo cuanto ella conoce sobre Java. Pero a **Víctor** le falta algo. Él dice que sus aplicaciones no se parecen a las que hacen sus compañeros, ni tampoco a las que se utilizan en las empresas. **Carmen** le responde que le falta hacer atractivas sus aplicaciones, con un aspecto corporativo que identifique los diferentes procesos.*

En las dos últimas unidades hemos venido empapándonos de la "metodología a seguir" para hacer una aplicación visual, añadiendo a nuestras aplicaciones algunos de los principales componentes Swing para dotar a nuestra aplicación de un interfaz gráfico adecuado, con una funcionalidad apropiada, y con un aspecto que podríamos calificar de "casi profesional".

Algunos de esos componentes, como los menús o los cuadros de diálogo, los hemos tratado en los ejemplos de esas unidades, pero sin explicarlos demasiado, a modo de introducción, ya que es en esta unidad en la que se verán con más detalle. Otros son nuevos, y vienen a completar nuestro "repertorio de herramientas" a la hora de construir un interfaz gráfico potente, funcional y visualmente agradable.



Creación de un menú



Para empezar **Carmen** le sugiere que intente concentrar todas las operaciones posibles de la aplicación en un menú principal, de modo que el usuario siempre pueda recurrir a esta herramienta cuando necesite localizar una operación. Insiste en que al menos debe contener todas las acciones que el usuario pueda realizar y lo ideal es hacerlo clasificándolas por categorías en submenús.

Víctor sabe que Java tiene grandes posibilidades para elaborar menús de opciones con una gran variedad de ítems, como casillas de verificación o botones de radio.



En la unidad anterior hemos visto ejemplos de aplicaciones que incorporaban menús, pero no nos hemos detenido a comprobar las posibilidades que existen, ni todos los elementos que intervienen en un menú, ni a ver el código que se genera al ir construyendo un menú de forma "gráfica" desde el diseñador de **NetBeans**. De eso nos vamos a ocupar justamente en este apartado.

Distintos componentes de un menú

Vamos a analizar los distintos componentes que pueden intervenir en la formación de un menú, viendo las clases Java que nos proporcionan cada uno de esos elementos, así como sus propiedades, y los métodos que podemos usar para incluirlos y usarlos dentro de un menú.

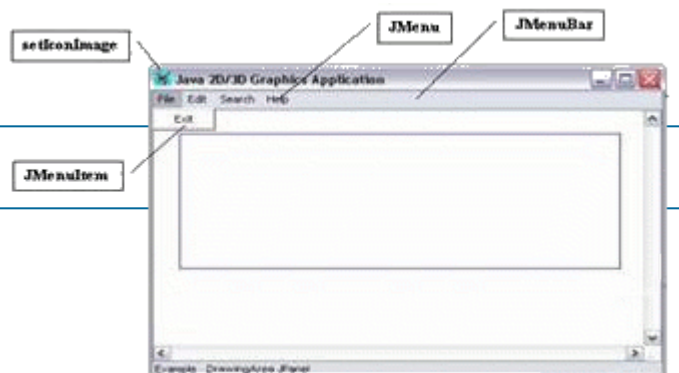
El primero de ellos, será la propia **"barra de menú"** de la ventana, que contendrá distintas palabras, cada una de ellas correspondiente a un **"menú"** distinto que se desplegará al seleccionar esa palabra con el ratón o con el teclado. A su vez cada uno de estos menús puede tener elementos de distintos tipos, cada uno de los cuales es un **"ítem de menú"** de un tipo determinado. Vamos a pasar a verlos con más detalle.



JMenuBar, JMenu, JMenuItem

¿Dónde tienen habitualmente los menús las aplicaciones que sueles usar, como por ejemplo la ventana del propio navegador que estás usando para seguir este curso?

Aparece como una barra de menú, justo debajo de la barra de título de la ventana.



La barra de menú no es más que el contenedor donde colocar todos los menús que queremos que incluya nuestra aplicación.

Por **ejemplo**: es casi un convenio que el primero de esos menús sea el menú Archivo, que tiene opciones relacionadas con las operaciones sobre ficheros que puede realizar la aplicación: (Abrir, Cerrar, Guardar, Imprimir, Archivos recientes usados, Salir, opciones de configuración, etc.), aunque nada nos obliga a disponer de este menú, o a que esté justamente en la primera posición de la barra de menú. No obstante, nunca olvides el principio de **"No sorprender al usuario"**. Si todas las aplicaciones tienen ese menú en esa posición, y con ese tipo de opciones, no hacerlo así en nuestra aplicación puede generar confusión y problemas a los usuarios de la misma. Mejor seguir los convenios, que si son convenios es por algo.

Archivo Editar Ver Ir Marcadores Herramientas Ayuda

¿Cómo podemos indicarle a nuestra aplicación que disponga de una barra de menú?

Se hace añadiendo esa barra de menú como un objeto de la clase **JMenuBar**. Puede hacerse mediante el propio diseñador del IDE, o mediante el código, generando un método **setJMenuBar()**, que añade una barra de menú ya creada a la ventana **JFrame**.

```
jMenuBar1 = new javax.swing.JMenuBar();      /* Se crea la barra de menú*/
...
setJMenuBar(jMenuBar1);                     /* Una vez creada la barra de menú, y después de haberle añadido
                                             todos los menús que se deseen, se establece como barra de menú
                                             de nuestra ventana JFrame. */
```

A esa barra de menú, una vez creada, se irán añadiendo los distintos menús, que serán objetos de la clase **JMenu** mediante el método **add()**

```
jMenu1 = new javax.swing.JMenu();           /* Se crea un nuevo menú, que se verá como una nueva entrada en la barra de menú. */
jMenu1.setText("Menu");                     /* Asociamos un texto con ese menú, que será el que se muestre en
                                             la barra de menú de la ventana. */
...
jMenuBar1.add(jMenu1);                      /* Una vez que se hayan añadido todos los elementos o ítems del
                                             menú, este se añade a la barra de menú. */
```

A cada uno de los menús, se irán añadiendo ítems o elementos de menú, que pueden ser de distintas clases, correspondiendo con los distintos tipos de elementos. El más básico sería un ítem estándar, de la clase **JMenuItem**. Sean del tipo que sean, todos los ítems se añaden al menú con el método **add()**. Incluso uno de esos ítems puede ser a su vez un nuevo menú que se despliegue como submenú al seleccionar ese ítem del menú.

```
jMenuItem1 = new javax.swing.JMenuItem()
/* Creamos el nuevo ítem de menú, como un objeto de la clase JMenuItem*/
```

```

jMenuItem1.setIcon(new javax.swing.ImageIcon(getClass().getResource("/ix.png")))
/* Le asociamos un icono para que se muestre con una imagen, además de con un texto
* en la barra de menú*/
jMenuItem1.setText("Item")
/* Le asociamos un texto, que será el que se mostrará para este ítem al desplegar el menú. */
jMenu1.add(jMenuItem1)
/* Añadimos el ítem al menú correspondiente*/

```

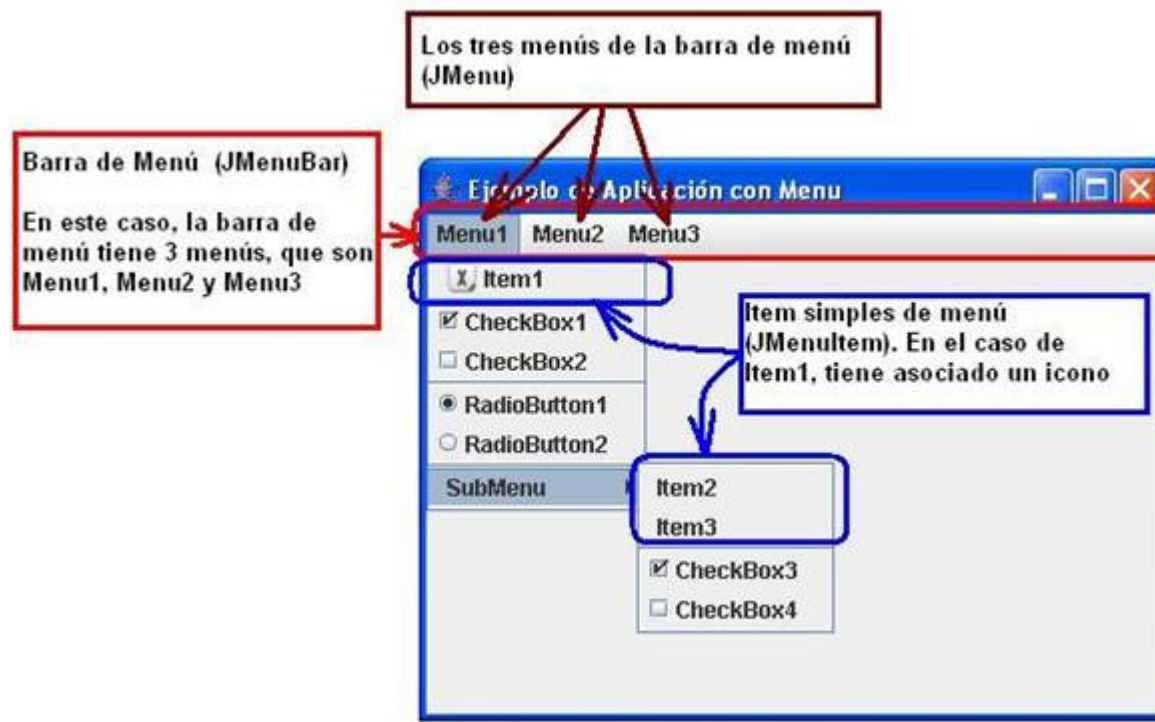
Todo el código, en el orden que se generaría, tal y como quedaría escrito, sería el siguiente:

```

jMenuBar1 = new javax.swing.JMenuBar()
jMenu1 = new javax.swing.JMenu()
jMenuItem1 = new javax.swing.JMenuItem()
jMenu1.setText("Menu")
jMenuItem1.setIcon(new javax.swing.ImageIcon(getClass().getResource("/ix.png")))
jMenuItem1.setText("Item")
jMenu1.add(jMenuItem1)
jMenuBar1.add(jMenu1)
setJMenuBar(jMenuBar1)

```

La siguiente imagen muestra cada uno de los elementos que hemos estudiado en este apartado.



DEMO: Mira cómo insertar una barra de menús en un JFrame

Autoevaluación

Para añadir una barra de menú a nuestra aplicación debemos añadir esa barra de menú como un objeto de la clase.

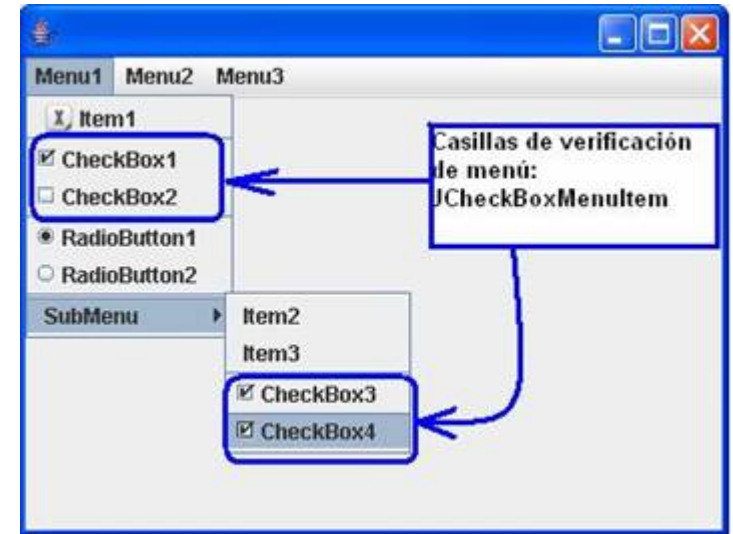
- ☐ a) **JMenuBar.**
- ☐ b) **setJMenuBar.**
- ☐ c) **Jframe.**
- ☐ d) **JBarTool.**

Comprobar

Casillas de verificación (JCheckBoxMenuItem)

En la imagen del apartado anterior ya hemos visto cómo se pueden insertar distintos tipos de componentes, y el aspecto de cada cual. Entre las opciones del menú, vemos que hay algunas que pueden marcarse, de manera similar a como se marcan las casillas de verificación. Son los **JCheckBoxMenuItem**, es decir, **casillas de verificación para un menú**.

El aspecto es ligeramente distinto a las otras casillas de verificación que no son para menús, de tipo **JCheckBox**, pero muy similar. El comportamiento sí es exactamente el mismo. Se trata de **opciones que pueden marcarse o no, de forma independiente unas de otras**. En la imagen, CheckBox1 y CheckBox2 pueden marcarse de forma independiente: Podrían estar marcadas ambas, o una de las dos, o ninguna.



Botones de opción dentro de un menú (JRadioButtonMenuItem)

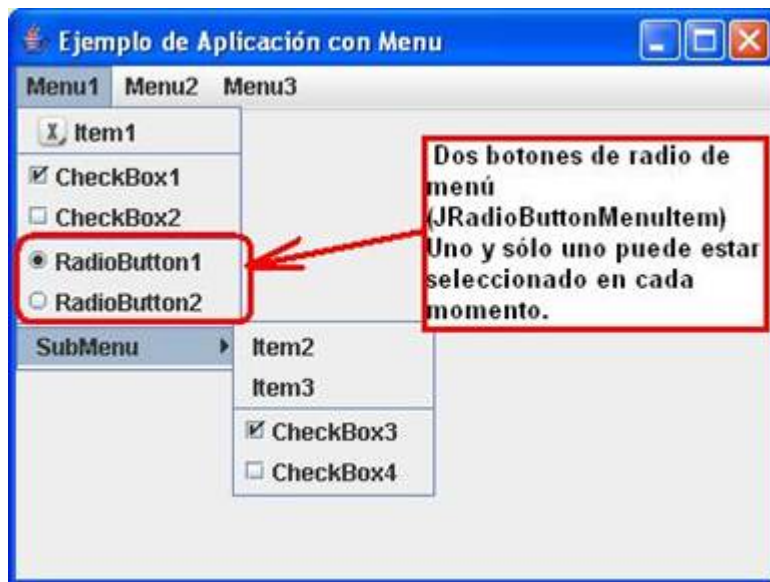
Los **botones de opción de menú** también son bastante similares a los botones de opción corrientes, sólo que los de menú son objetos de la clase **JRadioButtonMenuItem**.

Al igual que los botones de opción de aplicación, de tipo **JRadioButton**, los **JRadioButtonMenuItem** suelen usarse para seleccionar opciones que son excluyentes entre sí, de forma que **sólo una de ellas puede estar activa en cada momento, y necesariamente una de ellas debe estar activa**.

En la unidad anterior, en el ejemplo de la ecuación de segundo grado, teníamos un menú **Look&Feel** que nos permitía seleccionar un aspecto de entre dos posibles, pero necesariamente la aplicación debería tener uno de ellos activo.

No obstante, para conseguir este comportamiento, al igual que con los **JRadioButton**, tenemos que hacer que los distintos **JRadioButtonMenuItem** pertenezcan al mismo **JButtonGroup**, es decir, **debemos incluirlos a todos en el mismo grupo de botones**, seleccionándolo por medio de la propiedad **buttonGroup** en el diseñador.

En la imagen, puedes ver el aspecto de estos nuevos componentes que pueden incluirse en un menú.



Separadores (JSeparator)

En un menú pueden aparecer, como venimos viendo en los apartados anteriores, distintos componentes.

Entre ellos en algunas ocasiones nos puede interesar destacar un determinado grupo de elementos del menú como relacionados entre sí por referirse a un mismo tema, o por estar relacionados de alguna manera, o sencillamente para separarlos de otros que no tienen ninguna relación con ellos.



Un caso claro son los botones de opción o botones de radio de menú (**JRadioButtonMenuItem**). Ya que pertenecen a un mismo grupo de botones, nos puede interesar separarlos de alguna manera para que de un simple vistazo pueda verse claramente que de todos los que se ven agrupados y separados de los demás, sólo uno podrá estar activo al mismo tiempo. De esta forma, si otros botones de radio de menú pertenecieran a otro grupo de botones distinto, haremos que visualmente se distingan y se separen ambos grupos.

Esto se consigue insertando un **elemento separador**, de tipo **JSeparator**, que nos dibuja una línea horizontal en el menú, que separa visualmente en dos partes a los componentes de ese menú.

En la imagen siguiente te destacamos los objetos **JSeparator** que hemos incluido en el menú de la aplicación de ejemplo.




DEMO: Visualiza paso a paso la colocación de separadores en el menú

Submenús (JMenu como ítem dentro de un JMenu)

Frecuentemente dispondremos de alguna opción dentro de un menú que nos da paso a un conjunto más amplio de opciones posibles cuando la elegimos.

Por ejemplo, es habitual en los procesadores de textos disponer de un menú "Ver" en el que una de las opciones sea "Barras de Herramientas". Esta opción servirá para decidir qué barras de herramientas de las disponibles en la aplicación queremos que se muestren en cada momento (Barra estándar, barra de Dibujo, barra de Imagen, barra de Formato, barra de Tablas, barra de Revisión, etc).

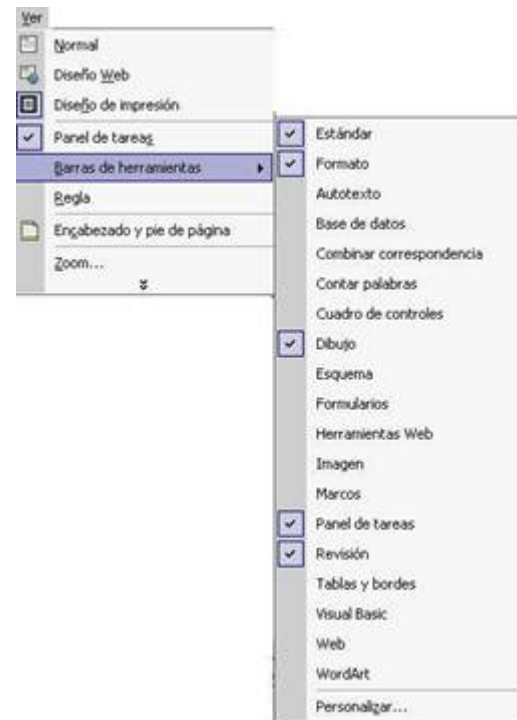
En lugar de meter todas esas opciones directamente en el menú "Ver", que haría que éste quedara un poco recargado, podemos incluir una única entrada "Barras de Herramientas", que al ser seleccionada nos abrirá un submenú (un menú dependiente del anterior) que nos ofrecerá todas las posibles barras de herramientas entre las que podemos elegir (así se hace, de hecho).

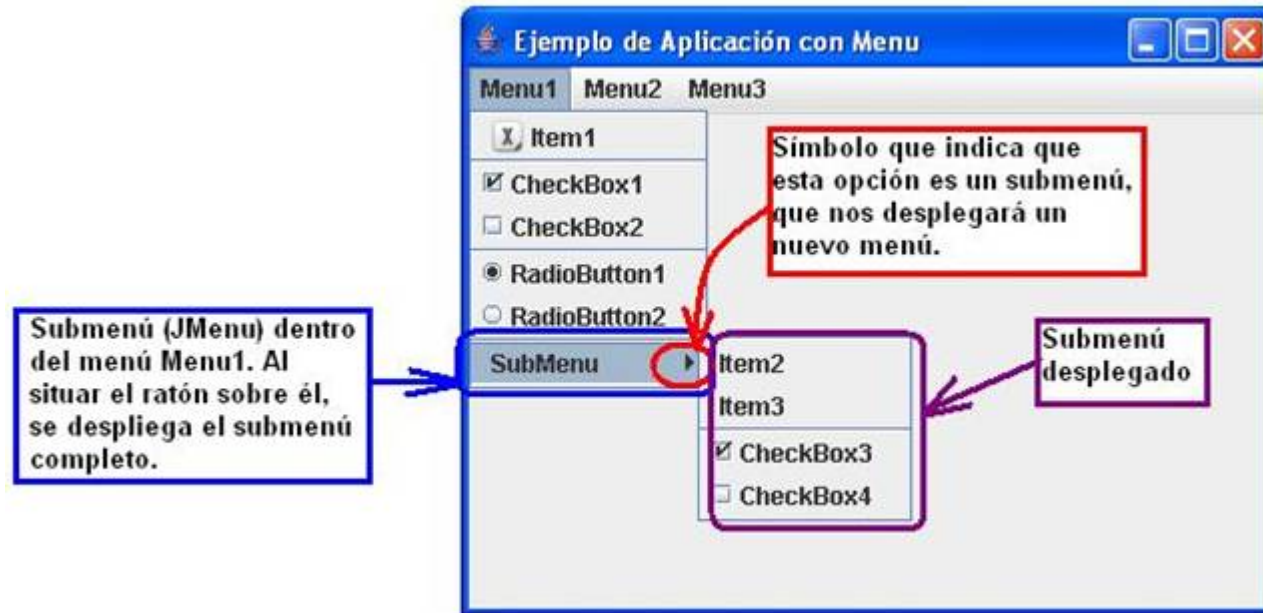
Cuando en un menú un ítem del mismo es a su vez un menú, se indica con el símbolo  al final de esa opción, de forma que se sepa que al seleccionarla nos abrirá un nuevo menú.

¿Cómo incluir un menú como submenú de otro?

Basta con incluir como ítem del menú, a un objeto que también sea un menú, es decir incluir dentro del **JMenu** un ítem de tipo **JMenu**.

Puedes ver el aspecto de un submenú en la imagen siguiente correspondiente a nuestra aplicación de ejemplo.





ERGONOMIA

Aceleradores de teclado y mnemónicos.

¿Alguna vez te has visto en la necesidad de usar una aplicación sin poder usar el ratón, porque se te haya roto, o porque sencillamente no lo tenías a mano?



Te aseguro que tales situaciones no son infrecuentes, es más, existe por parte de algunas compañías la afirmación de que el uso del ratón no sólo no facilita el trabajo con el ordenador, sino que lo hace más lento que si sólo se usa el teclado, y además su uso prolongado puede producir lesiones y problemas médicos, como el [síndrome del ratón o síndrome del túnel carpiano](#).

No es por tanto una cuestión sin importancia, y cuando diseñamos una aplicación debemos preocuparnos de las características de [Accesibilidad](#).

Incluso algunas empresas de programación obligan a todos sus empleados a trabajar sin ratón al menos un día al año, para obligarles a tomar conciencia de la importancia de que todas sus aplicaciones deben ser usables sin disponer de este dispositivo.

En este sentido, **es importante que cualquier componente interactivo de la ventana (cuadro de texto, botón de acción, lista desplegable, etc) y en particular cualquier opción del menú pueda seleccionarse sin el ratón, haciendo uso exclusivamente del teclado.**

¿Cómo podemos hacer que nuestros menús sean accesibles mediante el teclado?

La solución es la inclusión en todos los elementos del menú de [aceleradores de teclado o atajos de teclado](#) y de mnemónicos.



Un acelerador o atajo de teclado no es más que una combinación de teclas que se asocia a una opción del menú, de forma que pulsándola se consigue el mismo efecto que abriendo el menú y seleccionando esa opción.

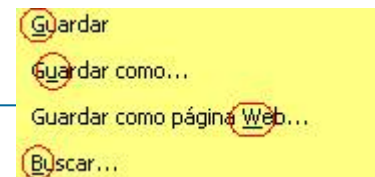
- Esa combinación de teclas aparecerá escrita a la derecha de esta opción del menú, de forma que el usuario pueda tener conocimiento fácilmente de su existencia.
- Añadir un atajo de teclado se consigue mediante la **propiedad accelerator** en el diseñador, que genera un método **setAccelerator()** como el que sigue:

```
jMenuItem2.setAccelerator(javax.swing.KeyStroke.getKeyStroke(
java.awt.event.KeyEvent.VK_M, java.awt.event.InputEvent.CTRL_MASK));
//literal
```

- Los atajos de teclado, como se puede ver en la imagen incluida en este apartado, no suelen asignarse a todas las opciones del menú, sino sólo a las que son frecuentemente usadas, y que merece la pena tener disponibles de la manera más rápida posible.



Un Mnemónico consiste en resaltar una letra dentro de esa opción del menú, mediante un subrayado, de forma que pulsando Alt+ se abra el menú correspondiente, y volviendo a pulsar la letra correspondiente, se seleccione la acción correspondiente a esa opción del menú.

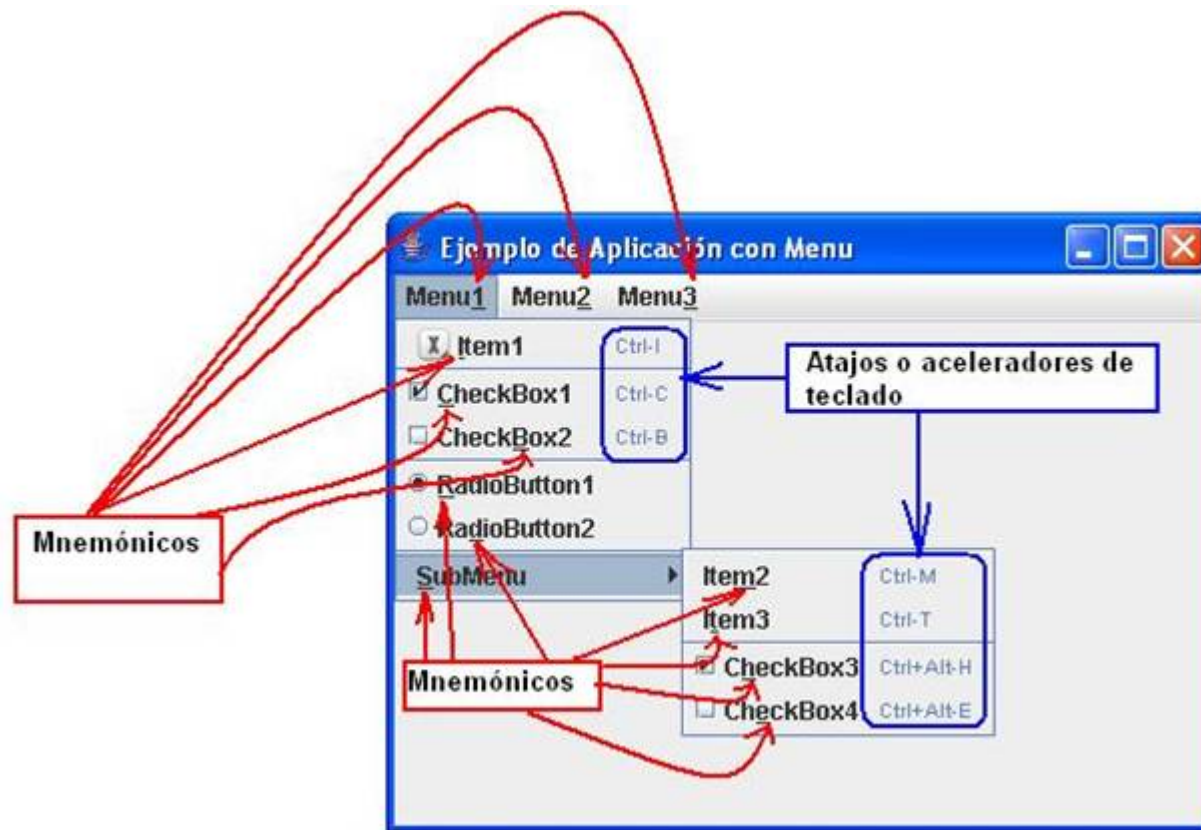


- Añadir un mnemónico a una opción del menú, o al propio menú, se hace igual que casi para cualquier otro tipo de componente, mediante la propiedad **mnemonic**, que genera un método **setMnemonic()** como el que sigue:

```
jMenuItem2.setMnemonic('m');
```

- Los Mnemónicos sí que deben ser incluidos para todas las opciones del menú, de forma que todas puedan ser elegidas haciendo uso sólo del teclado, mejorando así la accesibilidad de nuestra aplicación.

En la imagen siguiente puedes ver los atajos de teclado y los mnemónicos asociados a cada una de las opciones del menú.



En esa imagen vemos, por ejemplo, que el mnemónico para **Menu1** es **1**, ya que es el carácter que aparece subrayado, y para **RadioButton1** es **R**, que vuelve a ser el carácter subrayado, y para **Item2** es **m**, que es de nuevo el carácter subrayado.

Además, para algunos componentes, como para Item2, además de mnemónico hemos puesto un atajo de teclado. En este caso, para **Item2** hemos establecido como acelerador de teclado la combinación de teclas **CTRL+M**

De esta forma, tenemos **3 formas de seleccionar la opción Item2** del menú:

- **Con el ratón**, sin más que desplegar, haciendo clic con el ratón, el menú Menu1, seleccionar con el ratón SubMenú, y en el menú que se despliega volver a seleccionar la opción Item2.
- **Con el teclado, usando los mnemónicos**, pulsando Alt+1+S+M. De esta forma, Alt+1 abre el menú Menu1, la siguiente pulsación de S abre el submenú Submenu y la siguiente pulsación de M selecciona la opción Item2.
- **Con el teclado, usando el atajo de teclado**, pulsando la combinación de teclas CTRL+M. En este caso se accede a esa opción de forma directa, sin tener que abrir previamente ningún otro menú.



DEMO: Visualiza cómo insertar los mnemónicos y los aceleradores de teclado



PARA SABER MÁS:

En el siguiente enlace podrás encontrar información sobre [ergonomía](#) frente al ordenador, y en especial, información relativa al uso del ratón, vídeos y demostraciones interactivas incluidas.

Síndrome del ratón [\[Versión en caché\]](#)

Puedes encontrar información más detallada sobre pautas de accesibilidad que deben tenerse en consideración al diseñar una aplicación o un sitio Web en el enlace siguiente, que te proporcionamos en la versión original en inglés, así como una traducción en español:

Pautas de Accesibilidad del Contenido en la Web 1.0. Versión, original en inglés [\[Versión en caché\]](#)

[Pautas de Accesibilidad del Contenido en la Web 1.0. Versión traducida al español](#) [\[Versión en caché\]](#)

Autoevaluación



¿Cuál es la utilidad de los separadores de Java (**JSeparator**)?

- ☐ a) Sirven para separar los distintos objetos de un menú gráfico.
- ☐ b) Su utilidad es la de separar los métodos asociados a los controles de un menú para que no se solapen.
- ☐ c) Dibujan una línea que divide en dos un menú.
- ☐ d) Su utilidad es dibujar una línea horizontal en el menú que separa visualmente en dos partes a los componentes de ese menú.

Comprobar



Supongamos que hemos realizado una aplicación con un menú llamado **Menú**, que a su vez incluye un submenú llamado **Pregunta**, y ese submenú contiene las opciones: **Pregunta1** y **Pregunta2**. Suponemos también que hemos definido un mnemónico para **Pregunta2** que es **Ctrl+Alt+F1** y un atajo de teclado que es **F7**. De las siguientes maneras de acceder por parte del usuario a la opción **Pregunta2**, señala la correcta:

- ☐ a) Con el teclado, usando los mnemónicos, pulsando **F7**.
- ☐ b) Con el teclado, usando el atajo de teclado, pulsando la combinación de teclas **Ctrl+Alt+F1**.
- ☐ c) Con el ratón, sin más que desplegar, haciendo clic con el ratón, el menú **Menú**, seleccionar con el ratón **Pregunta**, y en el menú que se despliega volver a seleccionar la opción **Pregunta2** o con el teclado, bien usando el atajo de teclado pulsando **F7** o usando los mnemónicos, pulsando la combinación de teclas **Ctrl+Alt+F1**.
- ☐ d) Todas las anteriores son correctas.

Comprobar

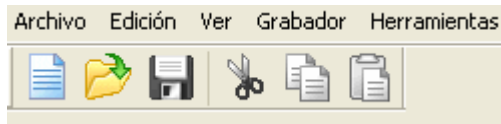
Barras de herramientas: JToolBar



Una vez que **Víctor** utiliza perfectamente los menús de opciones, **José** le explica que habitualmente introducen en todas las aplicaciones Barras de Botones con imágenes de las operaciones principales (o más habituales) que realiza dicha aplicación. Especialmente para abrir y cerrar ficheros, guardar cambios, o las típicas cortar, copiar y pegar, y sobre todo la que nunca puede faltar: **Deshacer**. Todo ello se incluye como parte de la aplicación con la posibilidad de ocultarla o visualizarla según las preferencias del usuario. Víctor sabe de qué está hablando y le apetece mucho utilizar este componente.



Siguiendo con el tema de la **accesibilidad**, ¿crees que será útil disponer de la misma funcionalidad repetida en varias partes de la aplicación?



Por **ejemplo**, si ya tenemos la opción Salir en un menú para terminar la ejecución de la aplicación, ¿resultará útil disponer de esa funcionalidad en algún otro sitio, como por ejemplo un botón?

Desde luego, resulta tan útil, que es lo que suelen hacer la mayoría de las aplicaciones, repetir la funcionalidad de sus menús en botones.

Normalmente será rentable repetir funcionalidades para aquellos aspectos de nuestra aplicación que se ejecuten con mucha frecuencia, garantizando un acceso más rápido que por medio de los menús. En tales casos, a estas opciones se les suele asociar un icono, de forma que el botón que duplica la funcionalidad lo único que contiene es ese icono, sin texto alguno. A veces ese icono se pone también en la opción del menú, a la izquierda, pero no siempre.



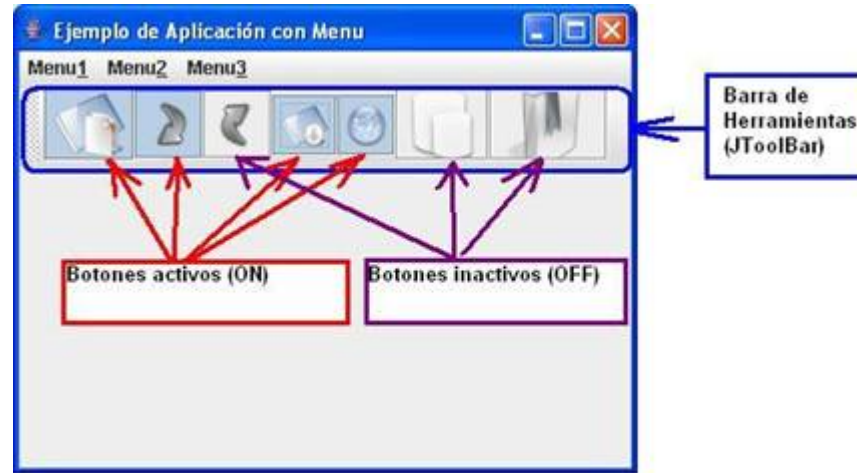
Por tanto es el icono común el que identifica que se trata de una función común.

Pero imagina que tenemos muchas funciones en nuestra aplicación que por su importancia queremos tener repetidas en botones. Podría resultar incómodo ir añadiendo montones de botones sueltos por toda la ventana de nuestra aplicación. Por tanto, resulta mucho más cómodo agruparlos todos en una **barra de herramientas**. Seguramente te resulta familiar el uso de estas barras de herramientas, ya que la mayoría de las aplicaciones disponen de alguna de ellas, pero ¿cómo se añade una barra de herramientas a nuestra aplicación Java?

- En Java las barras de herramientas se añaden como objetos de la clase **JToolBar**, y haciendo uso del diseñador.

- Basta con añadir un objeto **JToolBar** con el ratón al área de diseño, y luego añadir los botones a esa barra de herramientas, en vez de colocarlos en un panel, por ejemplo.
- Si queremos que los botones sólo contengan una imagen, basta con eliminarles el texto en la **propiedad text** y asociarles un icono con la **propiedad icon** desde el diseñador.

Hemos añadido una barra de herramientas a nuestra aplicación de ejemplo:



Las principales propiedades de una barra de herramientas **JToolBar** son:

- **floatable**: Indica que va a ser (o no) una barra flotante, que va a poder desplazarse en forma de ventana flotante al lugar de la ventana de la aplicación al que la desplazemos arrastrándola con el ratón, o estar fija en una posición.
- **orientation**: Si toma el valor 0, la barra estará orientada en horizontal. Si toma el valor 1, estará orientada en vertical.
- **rollover**: Indica si los botones de la barra van a tener (o no) bordes alrededor.

Por último añadir que es conveniente que todos los botones de una barra de herramientas tengan las mismas dimensiones, para que el aspecto sea visualmente más agradable (requisito que no cumplen los de la imagen de ejemplo).

Paneles múltiples con pestañas: JTabbedPane



Víctor está entusiasmado con lo que está aprendiendo. Esto más que programación parece diseño de pantallas y la verdad es que resulta hasta divertido. **Carmen** continúa mostrándole las posibilidades de Java en este terreno y él aprende muy deprisa. Simplemente parece que esto es lo suyo, con gran facilidad está asimilando el uso de estos componentes rápidamente e intenta descubrir nuevas posibilidades por sí mismo. Eso le ha ocurrido con los paneles de varias pestañas, que ha descubierto que es posible que un panel contenga nuevos paneles e incluso los puede ubicar en vertical.



A veces nuestra aplicación tiene tantas funcionalidades, y necesitamos incluir tantos elementos en nuestra ventana, que no resulta posible incluirlos todos para que se puedan ver cómodamente en una pantalla.

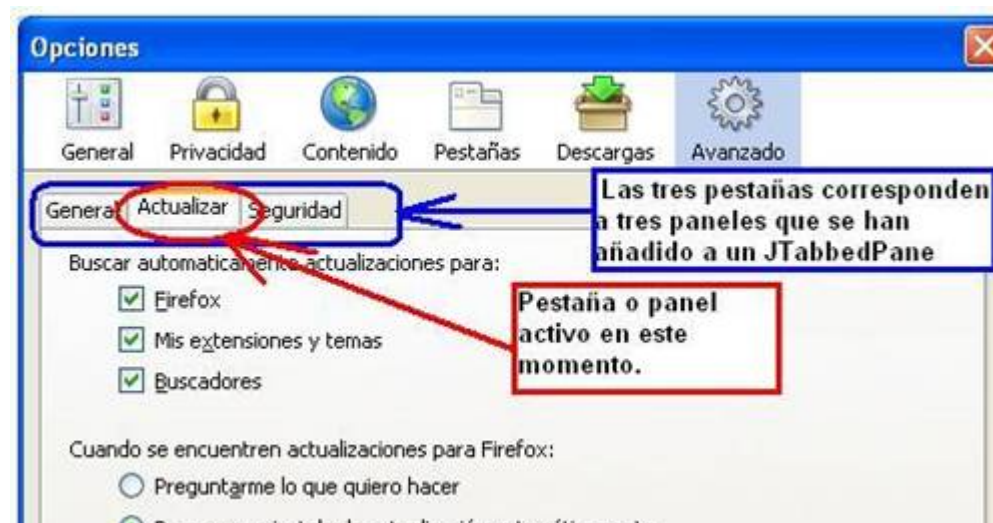
¿Cómo lo podemos solucionar?

Sería estupendo disponer de varios paneles, cada uno de los cuales pudiera contener una parte de los componentes que necesitamos, y poder elegir de forma sencilla qué panel es el que queremos ver en cada momento, de forma que ocupe el espacio disponible entero para él, y que los demás paneles momentáneamente no se vean.

¿Es eso posible en Java?

Lo es. Ya vimos una posible solución en la unidad anterior, usando **CardLayout**, pero existe otra posibilidad todavía más simple: **El uso de pestañas o paneles de tipo JTabbedPane**.

Como ejemplo te vamos a mostrar el uso que de este tipo de componentes gráficos se hace en el navegador Mozilla Firefox, eligiendo la opción Herramientas - Opciones - Avanzado. Podemos ver que aparecen tres pestañas, cada una de las cuales es en realidad un panel que se muestra delante de los demás cuando se pulsa sobre su pestaña.



Para conseguir este efecto, no tenemos más que:

- Añadir como panel de fondo de nuestra ventana un **JTabbedPane**, es decir, un panel con pestañas.
- A ese panel añadiremos nuevos paneles de tipo **JPanel**.
- Por cada panel que añadimos se crea una nueva pestaña en el **JTabbedPane**.
- Esos nuevos paneles **JPanel**, son contenedores a los que podemos añadir cualquier otro componente.

Las principales propiedades de **JTabbedPane** son:

1. **selectedIndex**: Nos permite especificar el índice de la pestaña que queremos que se muestre en primer plano, como activa por defecto, al mostrarse el **JTabbedPane**.


```
jTabbedPane1.setSelectedIndex(0);      /*hace que se muestre en primer plano la primera pestaña*/
```
2. **tabPlacement**: Nos permite escoger el lugar donde se mostrarán las pestañas para seleccionar los distintos paneles. Por defecto toma el valor TOP (arriba), tal y como se muestra en la imagen que hay a continuación, pero puede tomar también los valores BOTTOM (abajo), LEFT (izquierda) o RIGHT (derecha).
3. **Para cada uno de los paneles** que se incluyen en el **JTabbedPane**:
 - **Tab Title**: Permite indicar el texto que queremos que aparezca escrito en la pestaña de ese panel.
 - **Tab Icon**: Permite incluir un icono en la pestaña de ese panel.
 - **Tab Tooltip**: Permite indicar el texto de ayuda que aparecerá al parar durante unos segundos el puntero del ratón sobre la pestaña de ese panel.

En la siguiente imagen podemos ver el efecto sobre la aplicación de ejemplo que venimos construyendo. Hemos puesto un panel de pestañas (**JTabbedPane**), que contiene 3 paneles accesibles mediante sus respectivas pestañas. La segunda de esas pestañas es un panel que a su vez contiene otro **JTabbedPane**, al que le hemos modificado la propiedad **tabPlacement** dándole el valor **BOTTOM**, con lo que las dos pestañas de los dos paneles que contiene se muestran en la parte baja del panel.

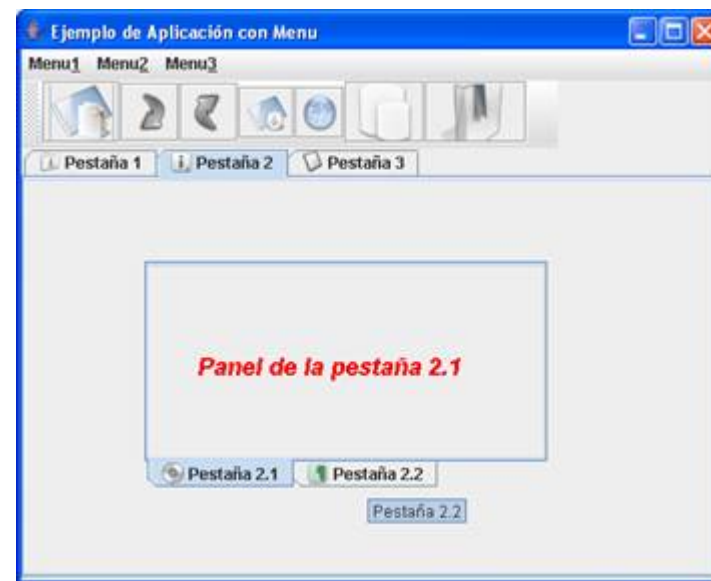
A todos los paneles (o pestañas) les hemos modificado la propiedad **Tab Title**, la propiedad **Tab Icon** y la propiedad **Tab Tooltip**.

Para que puedas probar esta aplicación y ver el código que el diseñador genera para menús, barras de herramientas, y para paneles con pestañas (**JTabbedPane**), te ofrecemos a continuación el enlace al proyecto NetBeans completo:

[!\[\]\(3211b5d1d968fc1665909b34f9f16010_img.jpg\) Descarga el proyecto JavaApplication7](#)



DEMO: Mira cómo se establece el Frame como BorderLayout



Inclusión de elementos gráficos decorativos en el diseño.



*En una de las reuniones de empresa, **María** explica que para **SI Ex** es importante desarrollar productos de buena calidad, elaborados cuidando los detalles y con una presentación atractiva. **Carmen** le recuerda ese comentario a **Víctor** y le explica las posibilidades que Java ofrece a la hora de "maquillar" las aplicaciones. La apariencia de una aplicación la determinan los atributos del texto utilizado; las imágenes o iconos empleados, las combinaciones de colores y la utilización de los componentes adecuados. Si todo eso se utiliza con buen gusto el resultado es perfecto, aunque eso no siempre ocurre.*



En el ejemplo anterior, y en algunos más de los vistos hasta ahora en el curso, puede que te hayas dado cuenta de que hemos incluido algunos iconos o imágenes para decorar y hacer más atractiva nuestra aplicación.

¿Es esa la única función de los iconos o imágenes que ponemos en nuestras aplicaciones?

No es la única, pero sí es quizás la más importante. El aspecto de una aplicación ayuda a que resulte agradable a los usuarios, y a que mejore enormemente la percepción de calidad de la aplicación que se les presenta. Por tanto, como las aplicaciones van destinadas a facilitarles la vida a los usuarios, no debemos menospreciar en absoluto este aspecto.

En algunas ocasiones, como hemos visto en las barras de herramientas, la misión de un icono es asociar una acción determinada de nuestra aplicación a un dibujo o imagen, de forma que en poco espacio, podemos incluir botones pequeños que son muy descriptivos de la tarea que realizan, sin necesidad de contener texto. Esto hace bueno también para la programación de aplicaciones el dicho "Una imagen vale más que mil palabras".

Incluso, **si una determinada funcionalidad aparece repetida en varios lugares de nuestra aplicación** (en un menú, en una barra de herramientas, en un botón de acción) **podemos asociarle en todos los casos el mismo icono, de forma que será más inmediato identificar la función de ese componente, y asociar que esa función es la misma que la de los otros componentes con los que comparte icono.**

En los apartados siguientes veremos cómo se añaden por medio del diseñador imágenes a nuestra aplicación.



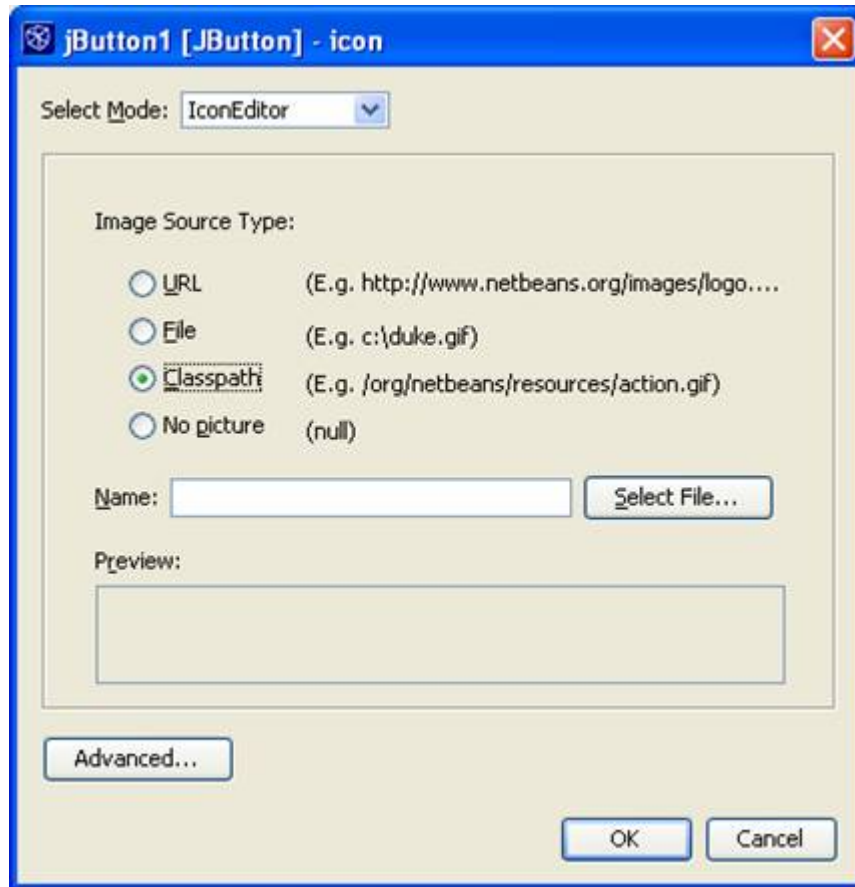
Iconos en botones y etiquetas

Uno de los primeros elementos que hemos aprendido a introducir en nuestras aplicaciones son los **botones de acción y las etiquetas**. También ahora son los primeros elementos en los que vamos a aprender a meter imágenes como iconos.

La forma más directa de hacerlo es por medio del diseñador, y el proceso es exactamente el mismo para las etiquetas y botones. Los **pasos a seguir** son los siguientes:

1. Nos aseguramos de incluir en la carpeta de fuentes del proyecto (carpeta src del proyecto) los ficheros de imagen que contienen la imagen a usar como icono (.gif, .jpeg o .png). Al compilar el proyecto se va a hacer una copia de esos ficheros en la carpeta build\classes que se encuentra en la propia carpeta del proyecto.
2. Seleccionamos el botón o la etiqueta a la que le queremos asociar un icono.
3. Seleccionamos la propiedad **icon** en la ventana de propiedades del diseñador. Esto nos muestra una ventana como la siguiente, en la que podemos seleccionar el lugar desde el que obtener el fichero para construir el icono. Puede ser una [URL](#) completa para descargarlo desde un lugar en Internet, o bien indicar la ruta completa a un archivo del sistema de archivos en el que se está ejecutando la aplicación, o bien una ruta relativa, que comienza en el paquete de nuestro proyecto. Esto último es lo más recomendable, ya que al incluir esos ficheros en el proyecto, cuando queramos ejecutarlo en otro ordenador distinto, no va a tener ningún problema con las rutas para encontrar esos ficheros.





4. Pinchamos sobre el botón **"Select File"**, que nos permite seleccionar el fichero concreto que vamos a usar para crear el icono. Nos abre una ventana **"Open Image File"**, que si hemos elegido la opción **Classpath** tendrá el siguiente aspecto:



5. Seleccionamos el fichero que deseemos (en la imagen, el fichero guardar.png) y pulsamos el botón **"OK"**.

6. El cuadro **"Preview"** nos muestra el aspecto de la imagen de ese fichero.



7. Pulsando de nuevo **"OK"** obtenemos el botón con el icono deseado.



PARA SABER MÁS:

En este enlace encontrarás un muy buen artículo sobre el uso de iconos en el desarrollo de interfaces gráficos y además incluye buenos enlaces para profundizar en los mismos

[Usando iconos en el desarrollo de interfaces](#) [\[Versión en caché\]](#)

En este enlace hay un manual de diseño gráfico con el cual te puedes introducir en estas técnicas y conocer las herramientas usadas para el mismo.

También hay una sección dedicada a iconos y logotipos

[Manual de diseño gráfico](#) [\[Versión en caché\]](#)

En esta página encontrarás tanto enlaces a iconos para descargar gratis como herramientas de edición y diseño de los mismos.

[Programas Gratis de Iconos y botones](#)

Imágenes como fondo en paneles

¿Y podríamos poner **imágenes de fondo** de un panel?

Seguramente habrás pensado que estaría bien poder colocar alguna **imagen** sugerente, o relacionada con el motivo de la aplicación como fondo de las ventanas, ocupando todo el panel de contenidos de la ventana.

Por ejemplo, estaría bien que de fondo de la aplicación estuviera el logotipo o un anagrama de la empresa para la que estamos diseñando nuestra aplicación, algo que le diera "imagen corporativa". Efectivamente, no es mala idea.

Y es posible que hayas pensado que la forma de hacerlo sea idéntica a la que hemos seguido para añadir un icono a un botón o a una etiqueta. Pero ahí te equivocas.



Los paneles no tienen propiedad `icon`, así que no es posible asociarles una imagen de fondo mediante esta propiedad.

Lo que sí podemos hacer es:

1. **Poner una etiqueta que ocupe todo el panel.**
2. **Asignarle a esa etiqueta un icono o imagen.**
3. **Añadir más elementos a ese panel, de forma que queden situados por encima de la etiqueta**, que de esta forma seguiría actuando de fondo.

Algo como lo que puedes ver en la imagen, en la que aparece una imagen como fondo de la ventana, con un botón, un cuadro de texto (en el que puede verse el cursor de edición entre las palabras Ángel y Azul) y una casilla de verificación que se muestran sobre la imagen de la etiqueta que contiene como icono la imagen de un ángel. Es sólo un ejemplo en el que se aprecia que el fondo está ocupado por la imagen, y que sobre ese fondo se pueden colocar los componentes que se deseen.

¿Pero si situamos los componentes y la etiqueta en un mismo lugar, no es posible que la etiqueta oculte a los otros componentes?

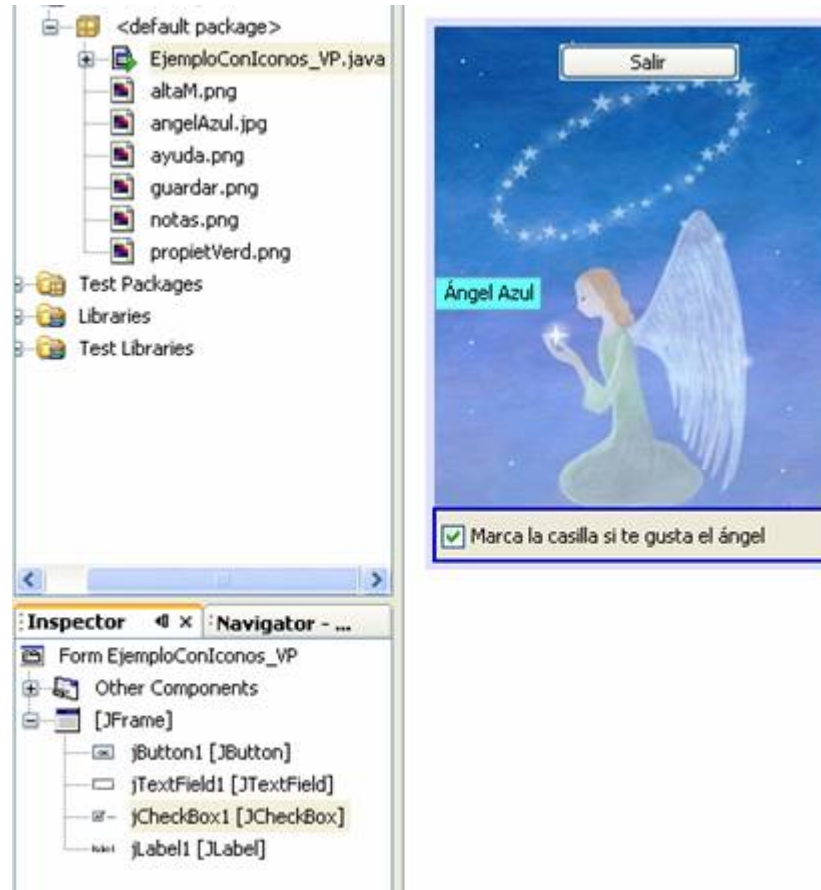
Es posible, y de hecho es lo que ocurre si insertas la etiqueta, le pones la imagen, y después empiezas a añadir los otros componentes. Para evitarlo, lo que debes hacer es lo siguiente:

- En la ventana "**Inspector**" de NetBeans, donde se muestra la jerarquía de componentes de nuestra aplicación, arrastra el componente que quieres que se muestre para que esté encima de la etiqueta que tiene la imagen del fondo (encima, pero en el mismo nivel de la jerarquía).
- Es decir, **para que la etiqueta se muestre realmente como fondo, tiene que estar al final de la lista de componentes incluidos en el panel.**

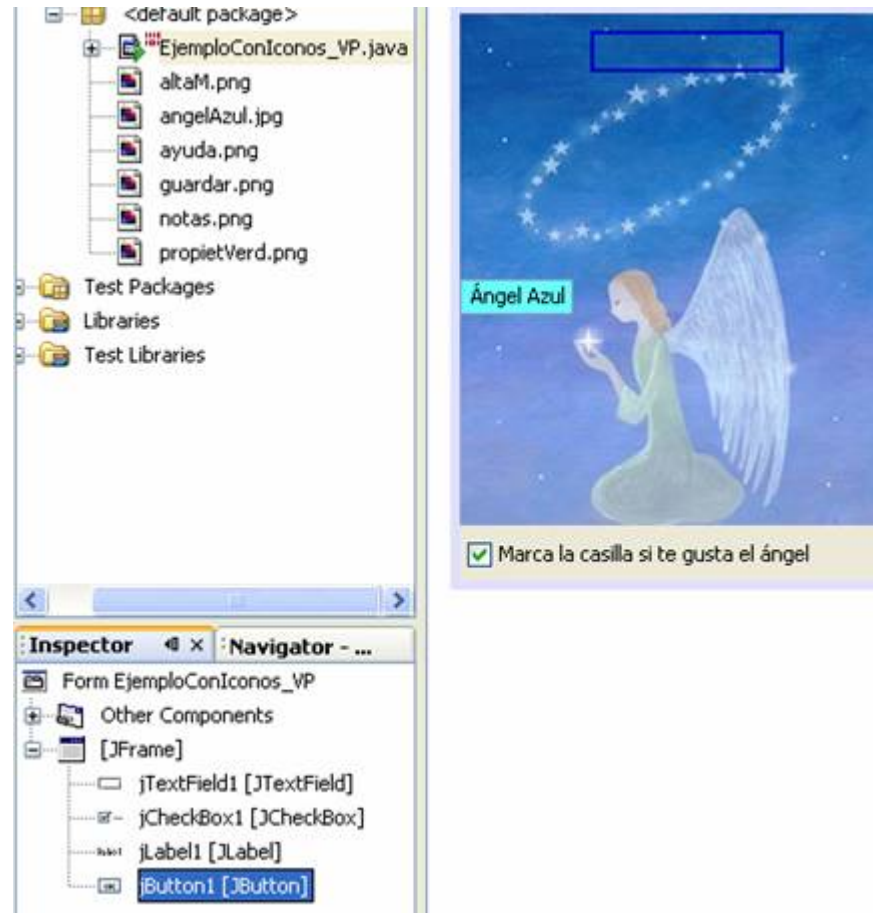
Por ejemplo:

En la imagen siguiente se observa tanto el diagrama de estructura de la ventana **Inspector** de NetBeans, como el resultado de esa configuración, que es justamente el efecto visto en la imagen anterior, con todos los componentes visibles sobre la imagen de fondo.





Pero si cambiamos de lugar el botón **jButton1** en el diagrama de estructura, pinchando y arrastrándolo debajo de la etiqueta **jLabel1** que tiene la imagen, **el botón seguirá situado en el mismo lugar del panel, pero no se verá porque la etiqueta lo tapará**. Observa que en el diseñador, se aprecia un borde azul que indica dónde está el botón, puesto que lo tenemos seleccionado en la ventana Inspector, y es el mismo lugar en el que estaba antes, pero ahora no se ve.



Una cosa más debes tener en cuenta.

- Si quieres mover algún componente de lugar en el panel, no puedes pincharlo y arrastrarlo directamente, ya que lo que estarías seleccionando y moviendo sería la etiqueta del fondo.
- Tendrás que seleccionarlo en la ventana **Inspector**, y llevarlo debajo de la etiqueta, tal y como hemos visto con el botón en las imágenes anteriores.
- El botón deja de verse, pero sí se ve su borde.
- Ahora sí puedes pinchar directamente sobre ese borde y arrastrarlo hasta otra posición.
- Para que se vea en esa nueva posición, volverás a seleccionarlo y arrastrarlo en la ventana **Inspector** para que esté encima de la etiqueta con la imagen de fondo.
- Una vez que lo sueltes en esa posición, volverá a estar visible.

Para cambiarle el tamaño a uno de los componentes, sí se puede hacer directamente, sin tener que cambiar nada en la ventana **Inspector**.



DEMO: Mira cómo insertar una etiqueta JLabel

Autoevaluación



Si quisiéramos poner una imagen de fondo para un panel de nuestra aplicación, de las siguientes afirmaciones señala la que consideres correcta.

- ☐ a) No es posible de ninguna manera ya que los paneles no tienen la propiedad **icon**.
- ☐ b) Debemos usar pestañas o paneles de tipo **JTabbedPane**.
- ☐ c) Podemos insertar una etiqueta de fondo y a esa etiqueta asignarle una imagen.
- ☐ d) Para que la etiqueta se muestre realmente como fondo tiene que estar al principio de la lista de componentes incluidos en el panel.

[Comprobar](#)

Inclusión de gif animados

¿Y podríamos incluir una imagen con animación para darle todavía más vistosidad a nuestra aplicación?

Naturalmente, podemos incluir un gif animado como icono. La forma de hacerlo no varía en lo esencial, salvo que en vez de seleccionar cualquier otro fichero, deberemos seleccionar el fichero **.gif** que se corresponda con el gif animado.



Hay muchas páginas en Internet donde puedes encontrar gif animados, pero siempre tendrás la opción de diseñarlos tú, si eres una persona creativa y se te da bien el diseño.

En el ejemplo del apartado 4 aparece un botón "Guardar" al que se le ha asociado un gif animado, en el panel de la pestaña 1.

Se trata de un disquete que camina. Puedes verlo si ejecutas el código del ejemplo.



PARA SABER MÁS:

En este enlace encontrarás un curso para construir tus propios gifs. Es necesario que tengas instalado Un editor gráfico: Photoshop, Autocad, Paint Shop Pro, Fireworks...etc y un animador (éste lo puedes descargar de la página)

[Construye tu propio Gif](#)

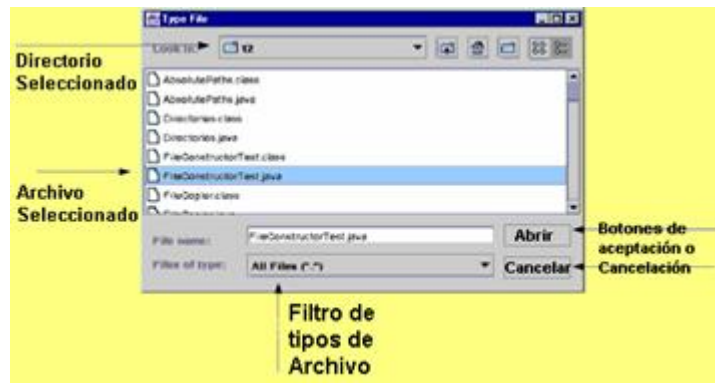
En esta página encontrarás un programa gratuito (pixia) que puede servir para la construcción de gifs y una ayuda para aprender a manejarlo

[El sitio oficial de Pixia en español](#)

Selección de ficheros para entrada/salida mediante FileChooser



La mayoría de las aplicaciones informáticas requieren la utilización de archivos de disco y en ocasiones tienen que ser localizados en la estructura de directorios del equipo. **Carmen** dice que lo mejor es utilizar una ventana específica para este fin, entre otras cosas porque es lo habitual y lo que todo usuario espera ante esa situación. **Víctor** sabe de qué está hablando ya que ha utilizado ventanas de este tipo frecuentemente y desde luego quiere aprender a programarlas, lo que no sabía es que ya vienen casi construidas.



En algunas de las aplicaciones anteriores hemos tenido necesidad de salvar algunos datos, para poder recuperarlos posteriormente en otras ejecuciones de nuestra aplicación. Sería impensable que cada vez que cerramos la aplicación se perdieran todos los datos que hemos introducido y elaborado trabajosamente.

Si bien es cierto que en las aplicaciones de gestión cada vez se usan más las **bases de datos**, y cada vez se usan menos los ficheros individuales para almacenar toda la información de "negocio" de la empresa, es bastante frecuente que una aplicación tenga una necesidad puntual de almacenar alguna información que no está recogida en la Base de Datos (por ejemplo, anotar el texto de una reclamación de un cliente, ya que como no es algo habitual, ni que forme parte de la aplicación de ventas-facturación-gestión de personal, no queremos que se almacene en la Base de Datos de nuestra empresa).

Para permitirlo, debemos usar **ficheros**.



En las unidades anteriores hemos visto numerosos ejemplos que usaban ficheros para almacenar información, así que no vamos a entrar de nuevo en los detalles sobre el funcionamiento de los ficheros, pero sí en la forma de indicarle a la aplicación cómo seleccionar el fichero desde el que hay que leer o en el que hay que guardar la información.

¿Recuerdas cómo lo hacíamos antes, en las aplicaciones que no tenían interface gráfica?

Sencillamente indicábamos en un **String** la ruta hasta ese fichero, y a partir de ese **String** se creaba un objeto **File**, que era la ruta sobre la que montábamos el flujo, etc.

Pero en las aplicaciones que tienen interfaz gráfica, **lo usual es usar una ventana que nos permita navegar por la estructura de discos, carpetas y ficheros de nuestro sistema**, hasta seleccionar el que deseamos, y en ese momento pulsar el botón Abrir, o Guardar, según lo que estemos haciendo, para obtener la ruta de ese fichero (el objeto **File**) sobre el que montar el flujo de salida.

Esta funcionalidad en Java nos la proporciona la clase **JFileChooser**.

Ejemplo con las posibilidades de JFileChooser

En la siguiente imagen puedes ver el aspecto que tiene un **JFileChooser**. Esa imagen está sacada de la ejecución de un ejemplo del Tutorial de Java, que nos ofrece éste y otros ejemplos del uso de **JFileChooser**, además de explicaciones sobre sus posibilidades y características.



PARA SABER MÁS:

Consulta el siguiente enlace para obtener más información sobre el uso de **JFileChooser** en el apartado "How to Use **JFileChooser**", del Tutorial de Java.

[Sección How To Use JFileChooser, del Tutorial de Java.](#) [Versión en caché]

Un **JFileChooser** no es más que un tipo especial de cuadro de diálogo que ya nos proporciona Java, ahorrándonos el trabajo de construirlo, y que se puede usar con bastante facilidad.

El siguiente código de ejemplo, sacado de la documentación de la API de Java para la clase **JFileChooser**, (**FileChooserDemo**) nos muestra prácticamente todo lo que tenemos que saber para usar correctamente esta clase, de forma que nos permita seleccionar ficheros para lectura o escritura, con bastante facilidad. Las únicas modificaciones que se han realizado sobre el código consisten en comentar aquellas sentencias sobre las que tienes que prestar especial atención, además de cambiar los mensajes para que se muestren en español.

Debes leer con atención los comentarios, ya que son una parte importante de lo que debes saber de esta clase, y de los contenidos de la unidad, sólo que resulta más práctico explicarlo sobre el código del ejemplo que aquí.

Hemos mantenido también los comentarios originales en inglés, por respeto a los autores.

Al haber construido el proyecto a partir de ficheros fuente existentes, en la carpeta del proyecto en lugar de la habitual carpeta **src**, encontrarás el código de las clases en la carpeta **JFileChooser1**. Por lo demás, para ejecutarlo, puedes hacerlo como de costumbre, indicando la carpeta del proyecto que puedes descargar del enlace siguiente:

 [Descarga el proyecto EjemploJFileChooserUno](#)

También te incluimos el código del otro ejemplo que aparece en la sección How To Use **JFileChooser** (**FileChooserDemo2**), del que ya hemos incluido una imagen al principio de este apartado, para ejemplificar todas las potencialidades de los cuadros de diálogo **JFileChooser**.

En este caso, no se va a comentar el código, ya que este ejemplo se proporciona más con el propósito de que lo ejecutes y experimentes con él. Fíjate en la forma de establecer un filtro para que el **JFileChooser** sólo muestre los archivos y directorios que cumplan las características que nosotros definamos para ese filtro. Para ello **lee atentamente los comentarios de la clase FileChooserDemo2 y de la clase ImageFilter, fundamentalmente.**

 [Descarga el proyecto EjemploJFileChooserDos](#)

Ventanas internas con JInternalFrame



Jesús se interesa por los progresos de Víctor con la programación en Java. Le comenta que lo más usual en la empresa son las aplicaciones MDI, en las que cada una de las operaciones se realiza en una ventana dentro de la ventana principal de la aplicación. Víctor no sabe a qué se refiere y Jesús le comenta que se trata de una forma de presentar las aplicaciones. Del mismo modo que lo hacen normalmente los procesadores de textos que tratan cada documento en una ventana diferente dentro de la ventana principal del procesador.



Seguramente habrás visto en alguna ocasión que una aplicación abre varias ventanas, para llevar a cabo algunas de sus funcionalidades. Por ejemplo, podemos tener una ventana en la que incluir toda la funcionalidad relacionada con las altas de clientes.

Esta **ventana** no es del todo independiente de la de nuestra aplicación. Por ejemplo, no tiene sentido mantener la ventana de altas de clientes abierta después de haber cerrado la aplicación. Incluso es posible que nos interese que **esa ventana se muestre siempre dentro del marco de nuestra ventana principal de la aplicación**, con el objetivo de que si movemos la ventana principal, la "subventana" de altas de clientes se mueva con ella, en vez de quedarse en el lugar que estuviera, como ocurriría con una ventana independiente.

Justamente **ésa es la funcionalidad que nos ofrecen las ventanas internas, de la clase `JInternalFrame`**, que vamos a estudiar en este apartado.

En la unidad 17, cuando hablábamos de Java WebStart, mencionamos una aplicación de ejemplo (SwingSet Demo) que mostraba las posibilidades de Swing, y entre ellas, el uso de ventanas internas.

Para ver ahora esa funcionalidad, usaremos el ejemplo de la sección "How to Use Internal Frames", del Tutorial de Java, que puedes encontrar en el siguiente enlace. Observa que las ventanas que se abren no pueden arrastrarse fuera del marco o ventana principal.



 [Descarga el proyecto EjemploJInternalFrame](#)



PARA SABER MÁS:

Aquí podrás encontrar más información sobre el uso de ventanas internas, de tipo `JInternalFrame` Sección How to Use Internal Frames (cómo usar Ventanas Internas) del Tutorial de Java. [Versión en caché]

Creación de JFrame

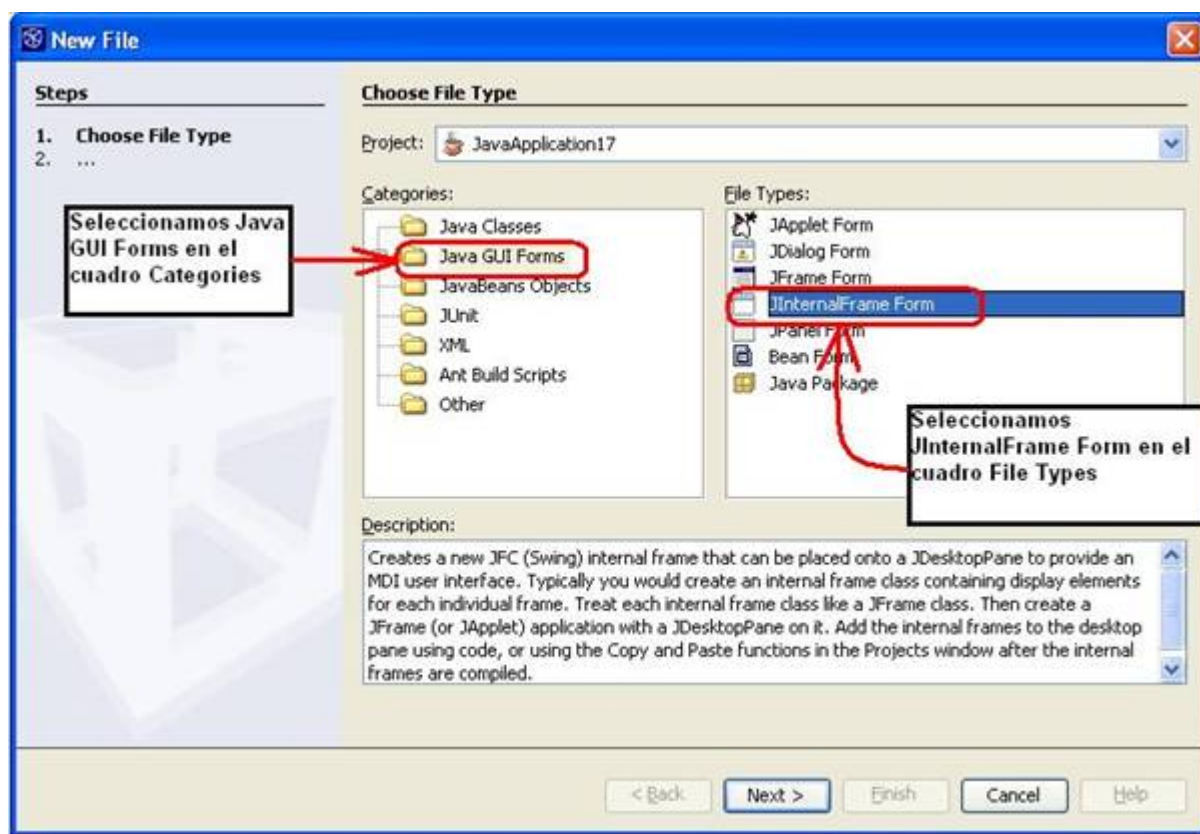
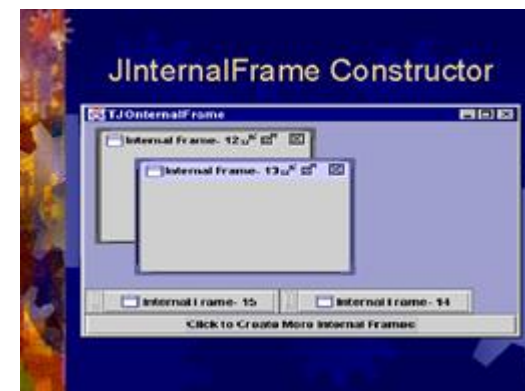
En el ejemplo del Tutorial de Java, se muestra una clase definida por el programador, **MyInternalFrame**, que extiende a **JInternalFrame**, pero no se le añaden componentes a esta ventana.

¿Cómo podemos diseñar una ventana de tipo **JInternalFrame**?

Afortunadamente, de la misma forma que diseñábamos las ventanas de tipo **JFrame**, sólo que al añadir a nuestro proyecto la clase de tipo **JInternalFrame**, debemos indicarlo al diseñador, de la misma manera que lo hacíamos con **JFrame**, pero seleccionando ahora **JInternalFrame**.

En la ventana "New File", en el cuadro "Categories" seleccionamos "Java GUI Forms", y en el cuadro "File Types" seleccionamos "JInternalFrame Form".

De nuevo, una imagen vale más que mil palabras:



A partir de ese momento, la ventana se diseña exactamente igual que las ventanas de tipo **JFrame**, y podemos añadirle todo tipo de componentes, incluso su propia barra de menú, usando el diseñador.

Listener de eventos para ventanas internas.

Piensa de nuevo en el ejemplo de una aplicación en la que al seleccionar la opción "Alta de clientes" en un menú, se abra una ventana interna para dar de alta nuevos clientes.

¿Consideras razonable permitir que el usuario empiece a abrir ventanas de alta de clientes, de forma que pueda tener varias ventanas de alta abiertas al mismo tiempo?

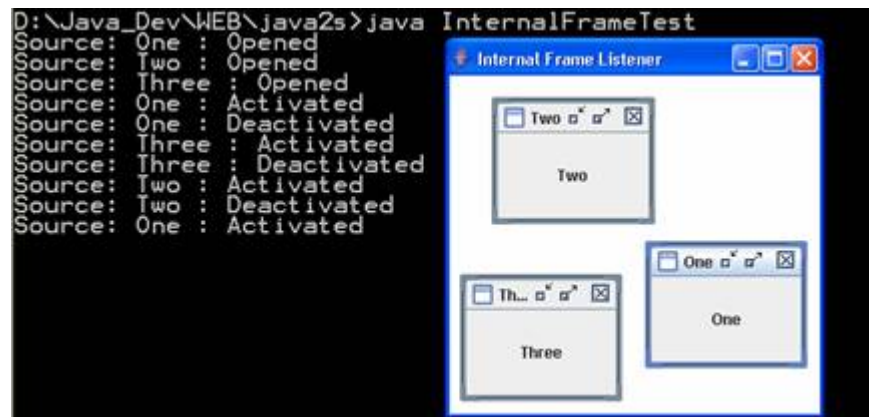
No parece útil, y sí puede generar problemas en un momento dado, al confundir al propio usuario, que puede acabar sin saber en qué ventana ha puesto qué datos ni de qué cliente. O en el mejor de los casos, acabaría usando una de esas ventanas, y todas las demás estarían abiertas ocupando recursos del sistema innecesariamente.

¿Cómo podemos evitarlo?



Capturando y manejando eventos de ventana, concretamente añadiendo a la ventana un `InternalFrameListener`.

En el ejemplo anterior, la clase `MyInternalFrame` define una variable estática `contadorDeVentanasAbiertas`, que usamos para numerar en la barra de título las ventanas, y para calcular la posición en la que se mostrará cada ventana creada.



Podríamos usar esa variable dentro de un manejador de eventos de ventana, de forma que cuando se abre una nueva ventana se le suma uno, y cuando se cierra la ventana se le resta uno. Además, lo primero que haríamos antes de abrir la ventana es comprobar si ya hay alguna ventana interna abierta, en cuyo caso, no se permite la apertura de una nueva ventana.

Si sólo queremos que se pueda abrir una ventana simultáneamente, podría ser una variable boolean `ventanaAbierta`, de forma que cuando abrimos una ventana la ponemos a `true`, y cuando la cerramos, la ponemos a `false`. **Sólo se permitirá abrir una nueva ventana cuando la variable tenga el valor `false`.**

Otra posibilidad para evitar que se cree más de una ventana es desactivar la opción correspondiente del menú al abrir la ventana, y activarla al cerrar la ventana.

Vamos a modificar el ejemplo anterior para que sólo pueda abrirse una ventana simultáneamente. Para ello vamos a seguir la estrategia de usar un variable lógica **ventanaAbierta**, y comprobaremos el valor de esa variable cada vez que queramos crear una nueva ventana.

- La clase **InternalFrameDemo**, que define la ventana principal, también va a implementar **InternalFrameListener**, lo que **nos obliga a implementar sus 7 métodos, aunque sólo necesitamos dos de ellos**, para indicarle que actualice el valor de la variable **ventanaAbierta** al cerrar y al abrir una ventana interna:

```
public void internalFrameClosed(InternalFrameEvent e) {
    ventanaAbierta=false;
}    /*Al cerrar la ventana, ponemos el valor de la variable a false*/

public void internalFrameOpened(InternalFrameEvent e) {
    ventanaAbierta=true;
}    /*Al abrir la ventana, ponemos el valor de la variable a true*/
```



- Los demás métodos del interface no tendrán ninguna sentencia en el cuerpo del método (abrir y cerrar llaves, sin nada en el interior, compruébalo en el código del ejemplo, así como la lista completa de métodos implementados).
- La variable **ventanaAbierta** de la clase **InternalFrameDemo** se ha definido como **static**, para que ambos métodos puedan usarla.
- A cada nueva ventana de tipo **MyInternalFrame** que se crea en el método **createFrame()** de la clase **InternalFrameDemo**, le añadimos un **InternalFrameListener**, que será justamente la ventana de la aplicación, referenciada por **this**, ya que es un objeto de la clase **InternalFrameDemo**, que implementa el interface.

```
MyInternalFrame frame = new MyInternalFrame();
frame.addInternalFrameListener( this );
```

- Antes de crear la nueva ventana, comprobamos el valor de **ventanaAbierta**, y sólo si es **false** permite que creemos una nueva ventana.

```
if(ventanaAbierta ==false){

    //código de creación de la ventana
}
```

- También hemos tenido que importar **InternalFrameListener** e **InternalFrameEvent**, y por supuesto hacer que la clase **InternalFrameDemo** implemente el interface **InternalFrameListener**.

En el siguiente enlace podrás encontrar el código completo del ejemplo modificado.

[⬇️ Descarga el proyecto EjemploJInternalFrameModificado](#)

Autoevaluación



¿Cuál es la utilidad de controlar capturando y manejando eventos de ventana, el que el usuario pueda o no abrir varias subventanas simultáneamente?

- ☐ a) El evitar que se dé la situación en que el usuario pueda a llegar a poder confundirse o sean para él poco entendibles las funciones de las subventanas asociadas a una ventana debido a que ha abierto subventanas que pueden llegar a dar esta situación(p.e. que el usuario no llegue a saber en que subventana ha introducido unos datos).

- ☐ b) El evitar que se dé un error de ejecución al intentar el usuario seleccionar un fichero de entrada/salida mediante **JFileChooser** desde una subventana.
- ☐ c) El evitar que se dé un error de ejecución al intentar el usuario navegar por la estructura de discos, carpetas y ficheros de nuestro sistema usando un **JFileChooser** desde una subventana.
- ☐ d) Todas las anteriores son correctas.

[Comprobar](#)

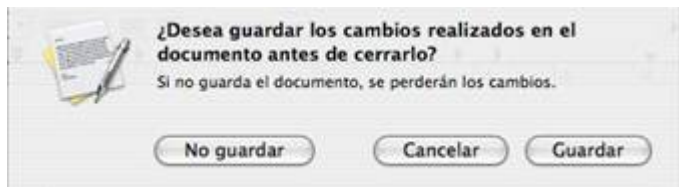
Cuadros de diálogo



*Sin duda los componentes preferidos de **Víctor** son los cuadros de diálogo mediante los que la aplicación establece la comunicación con el usuario. Se emplean cuadros de diálogo para que la aplicación avise de situaciones o para solicitar una elección ante varias posibilidades. También se utilizan para que el usuario proporcione datos a la aplicación. **Carmen** dice que tienen grandes posibilidades, pero que deben ser utilizados con mucho cuidado, porque su abuso puede aburrir al usuario.*



¿Qué entiendes por **cuadro de diálogo**?



Seguramente me dirías **que es cualquier ventana que se abre desde una aplicación para interactuar con el usuario, como indica su propio nombre, para dialogar con él, ofreciéndole información, o solicitándola.**

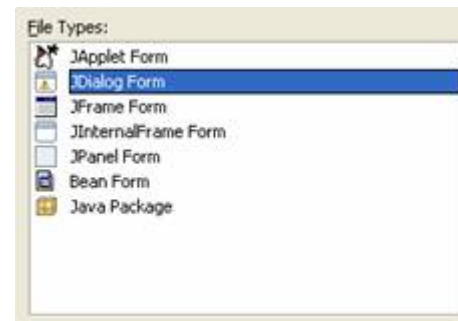
Esto nos da una idea de que se trata realmente de componentes **muy versátiles, y por lo tanto muy usados.**

Se diferencian de las ventanas de tipo **JFrame** en que, al igual que las ventanas internas, no son contenedores de alto nivel, es decir, no son independientes, sino que tienen que estar asociadas a un componente padre, normalmente a la ventana **JFrame** de la aplicación. Además son en cierta medida un tipo de ventana con posibilidades más limitadas.

¿Cómo diseñar un cuadro de Diálogo?

Diseñando una ventana de tipo **JDialog**, que es la clase a la que pertenecen este tipo de ventanas. Al igual que en el caso del diseño de **JInternalFrame**, basta con que al añadir un nuevo fichero, en la ventana "New File", en el cuadro "Categories" seleccionemos "Java GUI Forms", y en el cuadro "File Types" seleccionemos "JDialog Form". A partir de ahí el proceso es idéntico al del diseño de cualquier otra ventana.

En la unidad anterior, en el ejemplo de la ecuación de segundo grado, diseñamos un **JDialog** que servía de cuadro de Ayuda de la aplicación, mostrando información sobre las opciones disponibles, y algunos ejemplos de ejecución.



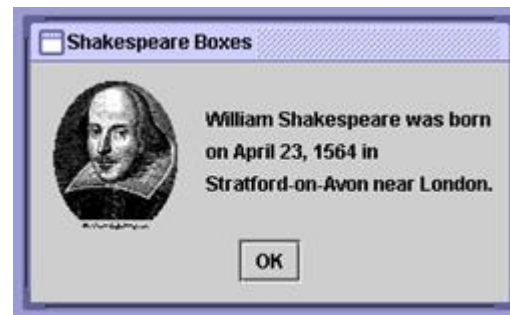
JDialog

La clase **JDialog** es la clase base para construir cuadros de diálogo.

Nos ofrece dos posibilidades:

- Crear cuadros de diálogo personalizados, que consiste en diseñar una ventana de tipo **JDialog**, tal y como hemos indicado en el apartado anterior.
- Crear cuadros de diálogo prefabricados, con la clase **JOptionPane**. En cierta medida esto es lo que nos permitía hacer **JFileChooser**, pero existe una gama amplia de posibilidades, que nos son ofrecidas por la clase **JOptionPane**, que no es más que una subclase de la clase **JDialog** que ya ha diseñado un repertorio más o menos amplio de cuadros de diálogo que necesitamos incluir frecuentemente en nuestras aplicaciones, evitándonos el engorroso trabajo de tener que diseñarlos siempre desde cero, cuando en realidad son siempre parecidos.

Piensa por ejemplo en el típico cuadro de diálogo que nos muestra una información, avisando de que una operación ha fallado, sólo para que leamos el mensaje, y una vez informados, lo cerremos, o los típicos cuadros de diálogo en los que se nos pregunta si deseamos continuar, y sólo tenemos las opciones de Aceptar o Cancelar.



- Éste es el tipo de cuadros de diálogo que podemos abrir de forma muy fácil con **JOptionPane**.



PARA SABER MÁS:

*En el Tutorial de Java, en la sección "How to Make Dialogs" encontrarás una información detallada sobre todo lo relativo a la creación de cuadros de diálogo. En especial resultaría interesante que ejecutaras con WebStart el código del ejemplo **DialogDemo**, para ver las posibilidades que nos brinda la clase **JOptionPane** para construir este tipo de ventanas, personalizándolas con poco esfuerzo. También es interesante que te descargues el código del ejemplo, compuesto por los ficheros **DialogDemo.java**, **CustomDialog.java** junto a un fichero usado para crear un icono, **middle.gif***

[Sección "How to Make Dialogs" del Tutorial de Java](#) [\[Versión en caché\]](#)

También te proporcionamos un enlace para que te descargues esos tres ficheros desde la plataforma del curso. Con ellos podrás crear un proyecto "con fuentes existentes", indicándole a NetBeans la carpeta en la que encontrará los ficheros fuentes, y ejecutarlo:

 [Descarga la carpeta JDialogDemo](#)

JOptionPane

JOptionPane parece una clase complicada, por la gran cantidad de métodos disponibles, pero en realidad permite mostrar cuadros de diálogo más o menos estándar con relativa facilidad, ya que su uso se limita a hacer una llamada en una sentencia de una línea a un método `showXxxDialog()`, donde **Xxx** representa el tipo de cuadro de diálogo que queremos mostrar.



Las posibilidades son:

Nombre del Método	Descripción
<code>showConfirmDialog()</code>	Plantea una pregunta de confirmación, con opciones como Si, No, Cancelar.
<code>showInputDialog()</code>	Pide al usuario la introducción de algún valor.
<code>showMessageDialog()</code>	Informa al usuario mediante un mensaje de algo que ha ocurrido.
<code>showOptionDialog()</code>	Es un tipo que unifica a los tres anteriores.

Te remitimos a la documentación de la API de Java para ver la definición y la lista de parámetros de todos esos métodos.

Los cuadros de diálogo generados por la clase **JOptionPane** tienen la propiedad **modal** a **true**, lo que significa que **mientras que el cuadro de diálogo esté abierto, no es posible interactuar con otras ventanas de la aplicación, incluida la principal.**

El aspecto general que tienen todos estos cuadros de diálogo, se corresponde con el del siguiente esquema.

ícono	mensaje
	Introducción de valores
Botones de opción	

Parámetros de los métodos ShowXxxDialog() de JOptionPane

Naturalmente esos métodos tienen una serie de parámetros que permiten personalizar el cuadro de diálogo, adaptándolo a las necesidades concretas de nuestra aplicación.

Los parámetros posibles (hay distintas versiones de los métodos y no todas necesitan todos los parámetros) son:

- **parentComponent**

Define el componente que va a ser el padre de este cuadro de diálogo, normalmente será la propia ventana **JFrame** desde la que se abre el cuadro de diálogo. Puede tomar el valor **null**, en cuyo caso se usa un Frame por defecto, y el cuadro de diálogo se sitúa en el centro de la pantalla.

- **message**

Un mensaje descriptivo que se escribirá en el cuadro de diálogo, para preguntar o informar de algo. El uso más común es que se trate de un **String**, pero el tipo de este parámetro es **Object**, por lo que podría ser cualquier otra cosa. Para más detalles, consulta la API de Java.

- **messageType**

Define el estilo del mensaje. El gestor del Look And Feel puede modificar el aspecto del cuadro de diálogo dependiendo del valor que se asigne, y **proporcionar un icono por defecto**. Los valores posibles, proporcionados como constantes de la clase **JOptionPane**, son:

- **ERROR_MESSAGE**
- **INFORMATION_MESSAGE**
- **WARNING_MESSAGE**
- **QUESTION_MESSAGE**
- **PLAIN_MESSAGE**

- **optionType**

Define el conjunto de botones que aparecerá en el fondo del cuadro de diálogo. Los valores posibles, definidos como constantes de la clase **JOptionPane**, son:

- **DEFAULT_OPTION**
- **YES_NO_OPTION**
- **YES_NO_CANCEL_OPTION**
- **OK_CANCEL_OPTION**

Pero no tenemos por qué limitarnos a esos valores.

De hecho, el siguiente parámetro nos permite definir un conjunto cualquiera de botones.

- **options**

Proporciona una descripción detallada del conjunto de botones que aparecerán en el fondo del cuadro de diálogo. El valor usual de este parámetro es un **array de String** (aunque realmente el tipo del parámetro sea un **array de Object**), de forma que se crea un botón por cada elemento del array al que se le pone dentro el texto del **String** para identificarlo.

- **icon**

Proporciona un icono decorativo para colocar en el cuadro de diálogo. No obstante, si no se indica lo contrario, se usará el icono por defecto establecido en el parámetro **messageType**.

- **title**

Establece el título para la barra de título del cuadro de diálogo. Es un **String**.



- **initialValue**

Establece el botón que aparecerá seleccionado por defecto.

Algunos de los métodos **ShowXxxDialog ()** devuelven un entero al ser cerrado el cuadro de diálogo que habían abierto (de la misma manera que ocurría con **JFileChooser**, si recuerdas). **Ese número entero devuelto indica el botón que se ha seleccionado para cerrar el cuadro de diálogo.**

Los valores posibles son:

- **YES_OPTION** Indica que se ha salido pulsando el botón Sí (Yes)
- **NO_OPTION** Indica que se ha salido pulsando el botón No
- **CANCEL_OPTION** Indica que se ha salido pulsando el botón Cancelar (Cancel)
- **OK_OPTION** Indica que se ha salido pulsando el botón Aceptar (Accept)
- **CLOSED_OPTION** Indica que se ha salido pulsando el botón Cerrar (Close)

Si hemos definido un conjunto distinto de botones con el parámetro **options**, se devolverá el índice correspondiente al elemento del array **options** cuyo botón se ha seleccionado.

Ejemplos de Cuadros de diálogo generados con JOptionPane.

Para finalizar vamos a incluir algunas de las sentencias que serían necesarias para mostrar determinados cuadros de diálogo, junto con el aspecto que tendría ese cuadro de diálogo:



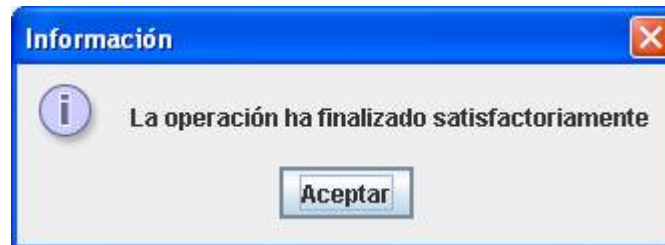
- Mostrar un cuadro de diálogo de error que muestre el mensaje "¡Alerta!, se produjo un error":

```
JOptionPane.showMessageDialog(null,
    "¡Alerta!, se produjo un error.", "¡Alerta!", JOptionPane.ERROR_MESSAGE);
```



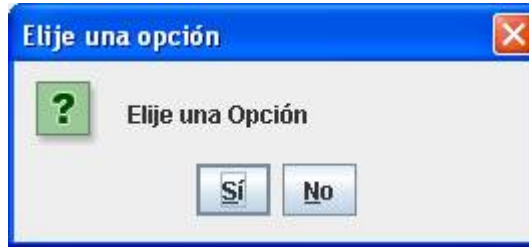
- Mostrar un cuadro de diálogo mostrando un mensaje con información sobre la aplicación, como por ejemplo "La operación ha finalizado satisfactoriamente" :

```
JOptionPane.showMessageDialog(null, "La operación ha finalizado satisfactoriamente ",
    "Información", JOptionPane.INFORMATION_MESSAGE);
```



- Mostrar un cuadro de diálogo de información con las opciones Si/No y el mensaje "Elige una opción":

```
JOptionPane.showConfirmDialog(null,
    "Elige una Opción", "Elige una opción", JOptionPane.YES_NO_OPTION);
```

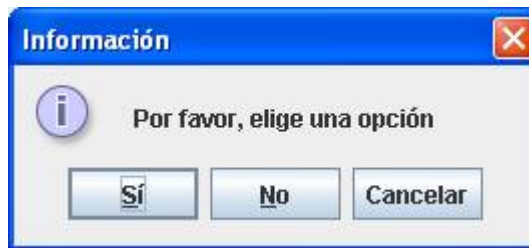


- Mostrar un diálogo de información con las opciones Si/No/Cancelar y el mensaje "Por favor, elige una opción", y con el título "Información" en la barra de título:

```

JOptionPane.showConfirmDialog(null,
    " Por favor, elige una opción ", "Información",
    JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.INFORMATION_MESSAGE);

```

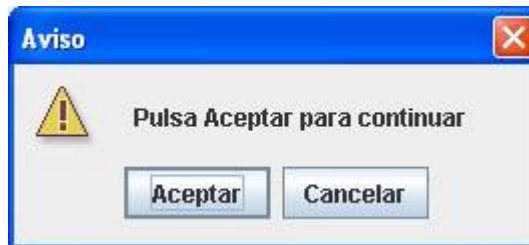


- Mostrar un cuadro de diálogo de aviso con las opciones Aceptar /Cancelar y con título "Aviso" para la ventana, y el mensaje "Pulsa Aceptar para continuar":

```

Object[] opciones = { "Aceptar", "Cancelar" };
JOptionPane.showOptionDialog(null, " Pulsa Aceptar para continuar ", "Aviso",
    JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE,
    null, opciones, opciones[0]);

```



- Mostrar un cuadro de diálogo solicitando al usuario introducir algún dato:

```

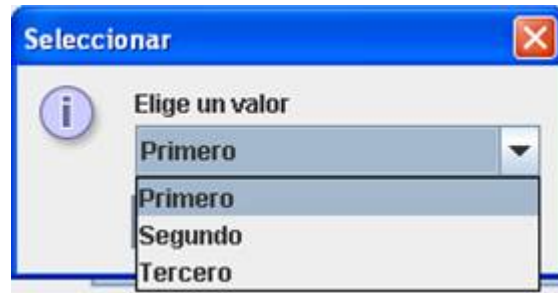
String ValorIntroducido = JOptionPane.showInputDialog("Por favor, introduce un valor");

```



- Mostrar un cuadro de diálogo solicitando al usuario que seleccione un String:

```
Object[] valoresPosibles = { "Primero", "Segundo", "Tercero" };
Object valorSeleccionado = JOptionPane.showInputDialog(null,
    "Elige un valor", "Seleccionar",
    JOptionPane.INFORMATION_MESSAGE, null,
    valoresPosibles, valoresPosibles[0]);
```



Autoevaluación



De las siguientes afirmaciones referidas a los cuadros de diálogo, señala la correcta.

- ☐ a) La clase **JDialog** es la clase base para construir cuadros de texto.
- ☐ b) La clase **JOptionPane** permite mostrar cuadros de diálogo más o menos estándar con relativa facilidad, ya que su uso se limita a hacer una llamada en una sentencia de una línea a un método **showXXXDialog()**.
- ☐ c) La clase **JOptionPane** permite mostrar cuadros de diálogo más o menos estándar con relativa facilidad, ya que su uso se limita a hacer una llamada en una sentencia de una línea a un método **Jdialog**.
- ☐ d) La clase **JOptionPane** permite mostrar cuadros de diálogo más o menos estándar con relativa facilidad, ya que su uso se limita a hacer una llamada en una sentencia de una línea a un método **showTextDialog()**

Comprobar

Tablas. JTable



Una de las mejores formas de mostrar datos o imágenes en una ventana es mediante el uso de Tablas. **Carmen** dice que es la representación preferida por todo tipo de clientes y que realmente es la forma más clara de representar grandes volúmenes de datos por ser intuitivas y editables. **Víctor** opina que son muy sencillas y no necesita que le explique nada, pero **Carmen** insiste y le explica lo importante que es conocer el código generado por NetBeans durante la fase de diseño y cómo adaptarlo a las necesidades del programador.



Imagina que tenemos, por ejemplo, una lista bastante grande de datos sobre una lista de personas, y que queremos presentarlos adecuadamente en una ventana de nuestra aplicación.

Un elemento muy útil para presentar listas de datos compuestos, son las tablas, en las que cada fila se correspondería con un registro, y cada columna con un dato de ese registro.

¿Puedo presentar los datos en tablas cómodamente en una aplicación Java?

Desde luego, es posible, y la clase **JTable** es la que nos ofrece esta funcionalidad.



Una vez más, hay que tener en cuenta lo que ya conocemos sobre la arquitectura modelo-vista-controlador. En este caso, **la tabla es la vista**, pero **el modelo es la estructura de datos que queremos representar**.

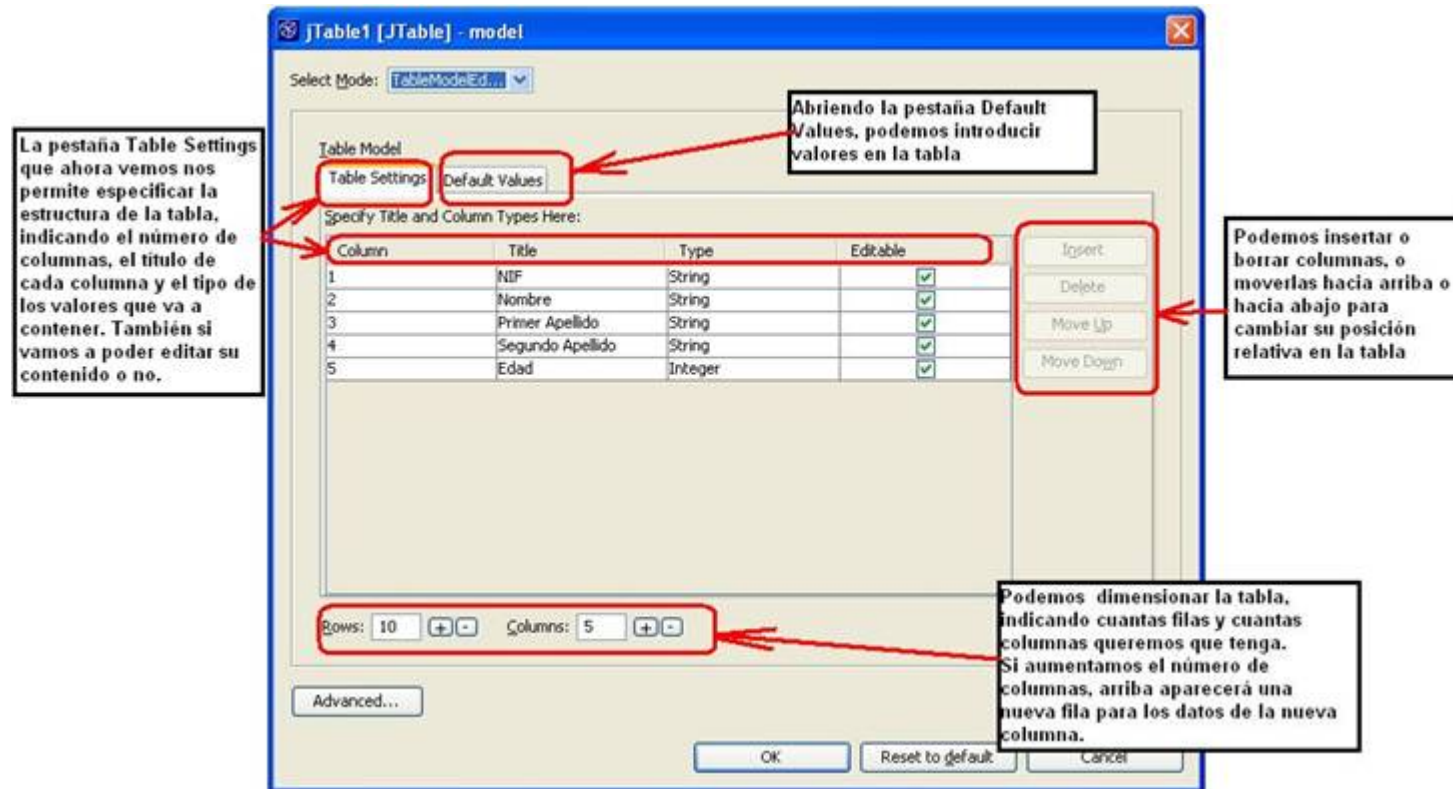
Por eso **lo realmente complicado es gestionar el modelo, es decir, gestionar esa estructura de datos**. Lo demás es sólo asociar la vista (tabla) con el modelo (estructura de datos) y el compilador se encarga de colocar los datos dentro de la tabla.

¿Qué modelo o estructura de datos crees que sería adecuado usar para una tabla?

Parece evidente que un **array bidimensional**, sería una estructura muy adecuada.

Incluir un **JTable** con la ayuda del diseñador es muy sencillo. Para ello debes:

- Arrastrar el icono de **JTable** desde la paleta de componentes Swing hasta el área de diseño.
- Es una buena idea colocarla sobre un **JScrollPane**, de forma que si la tabla crece y tiene muchas filas, podamos desplazarnos con la barra de desplazamiento para ver los datos de todas las filas.
- Una vez insertada la tabla en la ventana, podremos cambiarle las propiedades.
- La principal propiedad que debemos modificar es **model**.
- La propiedad **model** de **JTable**, nos permite decirle al compilador de dónde debe sacar los datos para llenar la tabla. El diseñador nos muestra una ventana donde poder definir el modelo de forma cómoda. La imagen siguiente muestra el aspecto de esa ventana:



Te mostramos también el aspecto de la ventana, seleccionando la pestaña "Default Values", que es la que nos permite introducir datos en la tabla.

jTable1 [JTable] - model

Select Mode: TableModelEd...

Table Model

Table Settings Default Values

Default Table Values:

NIF	Nombre	Primer Ap...	Segundo ...	Edad
74747474	Ana	López	Toro	24
12345678	Juan	Cruz	Ojeda	34
87654321	Antonio	Luna	Rodriguez	32
01234567	José	Sánchez	Mancha	12
76543210	Pablo	Martínez	Reverte	21
09876543	María	Andújar	Torres	54
34567890	Marisol	Piedra	Pica	

Columns:

Insert

Delete

Move Left

Move Right

Rows:

Insert

Delete

Move Up

Move Down

Rows: 10 + - Columns: 5 + -

Advanced...

OK Reset to default Cancel

Código generado y Principales propiedades para JTable

Las principales propiedades que podemos modificar para **JTable** son las siguientes:

- **model**: Ya la hemos visto ampliamente en el apartado anterior.
- **autoCreateColumnsFromModel**: Permite actualizar automáticamente la tabla cuando se producen cambios en el modelo.
- **autoResizeMode**: Permite determinar el modo en el que las columnas de la tabla cambiarán su tamaño cuando sea necesario, por aumentar el número de columnas, o por disminuir el tamaño del contenedor de la tabla.
- **cellSelectionEnabled**: Indica si se pueden o no seleccionar celdas individuales.
- **columnModel**: Establece el objeto que gobernará la forma en que las columnas se muestran en la vista.
- **columnSelectionAllowed**: Indica si se puede o no seleccionar por columnas, es decir, varias celdas de una misma columna.
- **intercellSpacing**: Establece el espacio vertical y horizontal de separación entre las celdas de la tabla.
- **rowSelectionAllowed**: Indica si se pueden hacer selecciones por filas, es decir, seleccionar varias celdas de la misma fila.
- **columnCount**: Indica el número de columnas de la tabla.
- **rowCount**: Indica el número de filas de la tabla.
- La lista incluye un largo etc, pero para conocer otras propiedades, te remitimos a la documentación de la API de Java.



En cuanto al código generado, es el siguiente:

```

jTable1 = new javax.swing.JTable();           /*Se crea el objeto JTable*/

//... Código de inicialización de otros elementos de la ventana

/* Indicamos cual va a ser el modelo de datos, definiéndolo "en línea". Sólo es un array
 * bidimensional de Object, definido explícitamente, de 10 filas y 5 columnas, con los
 * valores que se muestran. Como primer parámetro del método DefaultTableModel() se
 * indica ese array de Object, que es el modelo de datos, propiamente dicho, y como segundo
 * parámetro se indica un array de String que es el modelo para la cabecera de la tabla, es
 * decir, un array que contiene los títulos de todas las columnas de la tabla.
 */
jTable1.setModel(new javax.swing.table.DefaultTableModel(

    new Object [][] {

        {"74747474", "Ana", "López", "Toro", new Integer(24)},
        {"12345678", "Juan", "Cruz", "Ojeda", new Integer(34)},
        {"87654321", "Antonio", "luna", "Rodríguez", new Integer(32)},
        {"01234567", "José", "Sánchez", "Mancha", new Integer(12)},
        {"76543210", "Pablo", "Martínez", "Reverte", new Integer(21)},
        {"09876543", "María", "Andújar", "Torres", new Integer(54)},
        {"34567890", "Marisol", "Piedra", "Pica", null},
        {null, null, null, null, null}, //Estas últimas filas de la tabla están vacías.
        {null, null, null, null, null},
        {null, null, null, null, null}

    },

    /*Ahora, como segundo parámetro, el array de String con las cabeceras de las
     * columnas.
     */
    new String [] {
        "NIF", "Nombre", "Primer Apellido", "Segundo Apellido", "Edad"
    }
)

```

```

)      /* Fin del método DefaultTableModel(), que es el constructor de la clase
 * anónima que implementa el interface DefaultTableModel, y comienza la definición
 * de la clase anónima.
 */
{
    /*Aquí creamos un array de objetos Class que realmente nos guarda los tipos de los
    * datos de cada una de las columnas de la tabla
    */
    Class[] types = new Class [] {
        java.lang.String.class, java.lang.String.class, java.lang.String.class,
        java.lang.String.class, java.lang.Integer.class
    };
    /* Es obligatorio implementar el método getColumnClass del interface. */
    public Class getColumnClass(int columnIndex) {
        return types [columnIndex];
    }
};      /*Fin de la implementación de la clase anónima*/

/* Establecemos el valor para algunas propiedades de la tabla. */
jTable1.setAutoResizeMode(javax.swing.JTable.AUTO_RESIZE_ALL_COLUMNS);
jTable1.setCellSelectionEnabled(true);
jTable1.setInterCellSpacing(new java.awt.Dimension(5, 5));

/* Indicamos que la tabla se va a mostrar en el JScrollPane jScrollPane que hemos
 * creado.
 */
jScrollPane.setViewportView(jTable1);

```



DEMO: Mira cómo se inserta una tabla



PARA SABER MÁS:

Puedes encontrar información adicional sobre el uso de Tablas en Java consultando la sección "How to Use Tables" del Tutorial de Java, disponible en la documentación que se descarga con el JDK y también disponible para consultar en línea en el siguiente enlace. En él encontrarás importantes conceptos necesarios para un uso eficiente de las tablas, como por ejemplo, detectar las selecciones del usuario, detectar cambios en los datos, o crear un modelo de tabla (Table Model), entre otros. También encontrarás ejemplos que podrás descargar y ejecutar, así como analizar su código.

[Sección "How to Use Tables" del Tutorial de Java](#) [\[Versión en caché\]](#)

Extras



Carmen da por concluido el aprendizaje de **Víctor**. Ahora lo único que necesita para programar aplicaciones agradables y de calidad, es tener buen gusto y practicar mucho con los componentes para conocer al máximo las posibilidades que aportan a la programación. Finalmente le muestra un par de ejemplos que ha programado ella misma en los que utiliza, para la confección de informes, ficheros PDF. Esto le permite además de imprimir en prácticamente cualquier equipo, la posibilidad de transmitirlos fácilmente por la Internet y que sean visualizados en todos los sistemas operativos.

En uno de estos ejemplos, ejecutados bajo el sistema operativo Windows, **Víctor** observa que se pueden abrir aplicaciones como la calculadora de Windows o la información del sistema. Esto le parece importante y se interesa por la posibilidad de incluir en una aplicación, llamadas a otras ya existentes. **Carmen** se lo enseña, pero le avisa que eso limita las posibilidades multiplataforma de la aplicación.



Llevamos ya tres unidades viendo aspectos relacionados con el diseño de aplicaciones Java más o menos profesionales, ¿y todavía no hemos terminado?

Pues no. Sólo es un módulo pensado para el primer curso del módulo profesional, y por tanto ha debido centrarse más en los **aspectos generales de programación** que en el lenguaje concreto, que ha sido más una herramienta que un fin en sí mismo.

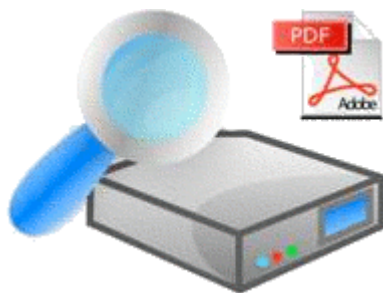
Eso quiere decir que necesariamente quedarán en el tintero muchos aspectos que son interesantes, y hasta necesarios para construir aplicaciones comerciales. No debes preocuparte. El camino recorrido hasta ahora es enorme, y además es la parte más difícil. En **otros módulos** de este mismo Ciclo Formativo profundizarás en otros aspectos relacionados con programación, que ampliarán lo que ahora sabes. Pero puedes estar seguro de que con lo que sabes, te puedes defender bastante bien en Java. Ten en cuenta que parte del trabajo de cualquier programador consiste en buscar documentación, estudiarla y ponerla en práctica. Lo más probable es que cuando termines este ciclo y emprendas la búsqueda de trabajo, ninguna empresa busque exactamente el perfil profesional que te han dado estos estudios, o conocimientos exactamente del mismo lenguaje que has aprendido. Pero sí que apreciarán las **capacidades** desarrolladas, sea en Java o en cualquier otro lenguaje, para entender los conceptos de programación, y para aprender de forma autónoma para adaptarte a nuevas situaciones. Por eso, como parte de la formación que pensamos debes adquirir, está la investigación y el aprendizaje de nuevas técnicas.



En esta unidad, si lo observas, hay frecuentes referencias a completar la información en la documentación de la API o consultando el Tutorial de Java. No es casual. Forma parte del aprendizaje necesario para un programador.

Pero ahora vamos un poco más allá, y te proponemos, que investigues y amplíes tus conocimientos profundizando sobre algunos temas en Java que no hemos visto hasta ahora, y que sin duda te resultarán de utilidad. Es lo que pretendemos con los siguientes apartados, proporcionarte ejemplos que son sólo un punto de arranque para que tú profundices e investigues.

Impresión y Generación de ficheros PDF en Java



¿Necesitan las empresas que sus aplicaciones impriman? La respuesta es evidente, y más si te hacemos la pregunta de otra forma. ¿Necesitas imprimir habitualmente con las aplicaciones que usas?

No todas las aplicaciones necesitan imprimir. Piensa en un juego, por ejemplo. Pero lo habitual es que las aplicaciones de gestión necesiten imprimir al menos de vez en cuando informes, listados, documentos, ...

Java por supuesto proporciona una serie de librerías útiles para facilitar la tarea de impresión, y te proponemos que investigues un poco este tema.

Aparte de la impresión, resulta útil generar informes en formato PDF (Portable Document Format, o Formato de Documento Portable) que tiene la ventaja de que una vez generado, todo el mundo lo va a ver igual, tal y como se generó, sin que dependa de las configuraciones locales de cada usuario. Es evidente que esta característica es muy deseable, y por ello la distribución de

documentos en formato PDF es un estándar que seguro no te resulta desconocido. De hecho, es una posibilidad que incluyen algunos procesadores de textos modernos, como el procesador de la [suite](#) OpenOffice.

Tampoco se te escapará la utilidad de que tus programas pudieran generar informes en este formato. Aparte de su portabilidad para distribuir el documento en formato electrónico, podemos considerarlo como un mecanismo que permite facilitar la impresión de nuestros documentos. Algunas aplicaciones así lo hacen. Por ejemplo, la aplicación web Séneca, de gestión académica de centros públicos de la Consejería de educación (programada en JSP, por cierto, que en cierta manera es Java) usa esta técnica. Cuando un profesor o en la Secretaría tiene necesidad de imprimir un documento, como por ejemplo el boletín de un alumno, hace una petición a la aplicación para que genere ese documento. Tras consultar los datos del alumno en la base de datos, se genera ese documento en formato pdf, y se le permite al usuario (profesor o secretaria) descargarse ese documento en su ordenador e imprimirlo en su impresora.



Pues bien, Java también proporciona librerías útiles para facilitar la generación de documentos PDF.

Para ayudarte con un punto de partida, te proporcionamos el código de un ejemplo en el que se hace tanto una impresión directa como la generación de un fichero pdf. El ejemplo consta de dos ficheros de clase (**GeneraPDF.java** e **Impresion.java**) y de dos librerías externas que se han tenido que añadir (**itext-1.3.jar** y **activation.jar**). Ambas librerías se han colocado en una carpeta llamada **lib** en el directorio del proyecto.

Las dos clases tienen su propio método **main()** por lo que tendrás que elegir en las propiedades del proyecto que ejecute como "Main class" primero una, y luego otra.

El efecto de ejecutar la clase **Impresion.java** es inmediato, siempre y cuando tengas una impresora conectada: Imprimirá una hoja con un texto de prueba.

El efecto de ejecutar la clase **GeneraPDF.java** es la generación de un fichero pdf en el directorio del proyecto, con el nombre indicado en el programa.

 [Descarga el proyecto PDFImpresiónJava](#)

Ejecución de aplicaciones externas

¿Alguna vez has visto una aplicación de un comercio?

Frecuentemente además de gestionar las ventas, el control de stock en almacén, la contabilidad etc. propias de la gestión, permite abrir y controlar desde el mismo menú de la aplicación de gestión del comercio, aplicaciones externas, como la de reproducción de música, o el bloc de notas, o la calculadora del sistema operativo, como si se tratara de ventanas de la propia aplicación.

Conseguir integrar en nuestra aplicación la llamada a otras aplicaciones que pueden estar programadas en cualquier otro lenguaje, de forma que el usuario perciba su ejecución como parte de una única aplicación, es una herramienta poderosa, que mejora la percepción que ese usuario tendrá de nuestra aplicación.

A continuación te proporcionamos un ejemplo en el que se hace uso de esta técnica. El menú "**Aplicaciones**" permite abrir el **bloc de notas**, la **calculadora**, el programa **Paint** y el reproductor de música **Winamp**.

En este último caso, se ha habilitado una opción del menú para "Abrir Winamp" desde la aplicación, y otra opción para cerrarlo, de forma que al abrir, esa opción del menú se deshabilite hasta que se cierre de nuevo. Así se evita que puedan abrirse más de una ejecución de la misma aplicación.

De igual manera, la opción de cerrar está inicialmente inhabilitada, y se habilita cada vez que se abre una nueva ejecución, para volver a inhabilitarse cada vez que se cierra, para no cerrar algo que no está abierto.

 [Descarga el proyecto EjecutarAplicacionExterna](#)



