Swing y JFC (Java Foundation Classes)

Autor: Sun
Traductor: Juan Antonio Palos (Ozito)

Sobre el JFC y Swing

¿Qué son el JFC y Swing?

JFC es la abreviatura de Java Foundation Classes, que comprende un grupo de características para ayudar a construir interfaces gráficos de usuario (GUIs).

Los componentes Swing
Incluye todo desde botones hasta splitpanes o tablas.
Soporte de Aspecto y Comportamiento Conectable
Le ofrece a cualquier componente Swing una amplia selección de aspectos y comportamientos. Por ejemplo, el mismo programa puede usar el Aspecto y Comportamiento Java o el Aspecto y Comportamiento Windows. Esperamos mucho más de los paquetes de Aspecto y Comportamiento -- incluyendo algo que use sonido en lugar de un 'look' visual.
API de Accesibilidad
Permite tecnologías asistivas como lectores de pantalla y display Braille para obtener información desde el interface de usuario.
Java 2D API (sólo JDK 1.2)
Permite a los desarrolladores incorporar fácilmente graficos 2D de alta calidad, texto, e imágenes en aplicaciones y applets Java.
Soporte de Drag and Drop (sólo JDK 1.2)
Proporciona la habilidad de arrastrar y soltar entre aplicaciones Java y aplicaciones nativas.

Las tres primeras caracterísitcas del JFC fueron implementadas sin ningún código nativo, tratando sólo con el API definido en el JDK 1.1. Cómo resultado, se convirtieron en una extensión del JDK 1.1. Esta versión fue liberada como JFC 1.1, que algunas veces es llamada 'Versión Swing'. El API del JFC 1.1 es conocido como el API Swing.

Nota: "Swing" era el nombre clave del proyecto que desarrolló los nuevos componentes. Aunque no es un nombre oficial, frecuentemente se usa para referirse a los nuevos componentes y al API relacionado. Está inmortalizado en los nombres de paquete del API Swing, que empiezan con "javax.swing."

Esta sección se concentra en los componentes Swing. Te ayudaremos a elegir los apropiados para tu GUI, te diremos cómo usarlos, y te daremos la información que necesites para usarlos de forma efectiva. Explicaremos el Aspecto y Comportamiento Conectable y el soporte de Accesibilidad cuando afecten a la forma de escribir programas Swing. No cubre aquellas caracterísitcas del JFC que sólo aparecen en el JDK 1.2. Para información sobre ellas, puedes ver la sección sobre **Graficos 2D** y la Home Page del JFC.

Los siguientes gráficos muestran tres vistas de un GUI que usa componentes Swing. Cada imagen muestra el mismo programa pero con un Aspecto y Comportamiento diferente. El programa, llamado **Converter**, se explica en detalle al final de la siguiente lección:

Java Look & Feel	CDE/Motif Look & Feel	Windows Look & Feel

¿Qué Versiones Contienen el API Swing?

El API Swing se presenta en dos versiones.

- JDK 1.2
- JFC 1.1 (para usar con JDK 1.1)

La versión que deberías usar depende de si necesitas usar JDK 1.1 o JDK 1.2. Es más sencillo usar JDK 1.2, ya que no necesitas añadir librerías para poder usar el API Swing; el JFC construido dentro del JDK 1.2. Sin embargo, si necesitas usar el JDK 1.1, añadir el API Swing (usando JFC 1.1) no es difícil. Las instrucciones para hacer ambas cosas están en las páginas siguientes.

Esta sección describe el API Swing 1.1, que es la versión presente en JDK 1.2 y en la versión llamada 'JFC 1.1 (con Swing 1.1).' El código de esta sección, funciona, sin ninguna modificación, en ambas versiones.

Sun ha liberado muchas versiones del JFC 1.1, que están identificadas por la versión del API Swing que contienen. Por ejemplo, una versión anterior del JFC 1.1, fue llamada "JFC 1.1 (con Swing 1.0.3)." La siguiente tabla muestra algunas de las versiones importantes que contienen el API Swing. La fuente en negrita indica las versiones típicamente usadas en productos comerciales.

Versión del API Swingg	Versión del JFC 1.1 Correspondiente	Versión del JDK 1.2 Correspondiente	Comentarios
Swing 0.2	JFC 1.1 (con Swing 0.2)	**ninguna**	La primera verión púbica del JFC 1.1.
Swing 1.0.3	**JFC 1.1** **(con**	**ninguna**	La versión del JFC 1.1 incluida en **Java Plug-in 1.1.1**. Soportada para el uso en productos comerciales.

	Swing 1.0.3)		
Swing 1.1 Beta	JFC 1.1 (con Swing 1.1 Beta)	JDK 1.2 Beta 4	La primera versión del JDK 1.2 que usa los mismos nombres de paquetes Swing que la correspondiente versión del JFC 1.1.
Swing 1.1 Beta 3	JFC 1.1 (con Swing 1.1 Beta 3)	JDK 1.2 RC1	La primera versión con los nombres finales de los paquetes Swing.
Swing 1.1 **Nota: Este es el API cubierto por este tutorial.**	**JFC 1.1** **(con Swing 1.1)**	**JDK 1.2 FCS**	La primera versión que contiene el API final de Swing 1.1 que está soportada para el uso en productos comerciales. **Java Plug-in 1.1.2** y **Java Plug-in 1.2** proporciona soporte para applets para JDK 1.1 + Swing 1.1 y JDK 1.2, respectivamente.

¿Qué Paquetes Swing Debería Utilizar?

El API Swing es poderoso, flexible -- e inmenso. Por ejemplo, la versión JFC 1.1 tiene 15 paquetes públicos: **javax.accessibility**, **javax.swing**, **javax.swing.border**, **javax.swing.colorchooser**, **javax.swing.event**, **javax.swing.filechooser**, **javax.swing.plaf**, **javax.swing.plaf.basic**, **javax.swing.plaf.metal**, **javax.swing.plaf.multi**, **javax.swing.table**, **javax.swing.text**, **javax.swing.text.html**, **javax.swing.tree**, y **javax.swing.undo**.

Afortunadamente, la mayoría de los programas sólo usan un subconjunto de este API. Esta sección ordena el API para tí, ofreciendo el código más común y guiándote por los métodos y clases que podrías necesitar. La mayoría del código de esta sección usa sólo uno o dos paquetes swing.

- **javax.swing**
- **javax.swing.event** (no siempre es necesario)

¿Que tienen de diferente los componentes Swing de los componentes AWT?

Si no te importan los componentes AWT, puedes saltarte esta seccción. Puedes obtener una introducción más general a los componentes Swing en Una ruta rápida por el código de una aplicación Swing.

Los componentes AWT son aquellos proporcionados por las plataformas JDK 1.0 y 1.1. Aunque JDK 1.2 todavía soporta componentes AWT, recomendamos encarecidamente el uso de componente Swing en su lugar. Puedes indentificar los componentes Swing porque sus nombres empiezan con **J**. Por ejemplo, la clase button del AWT se llama **Button**, y la clase buttón de Swing se llama **JButton**. Los componentes AWT están en el paquete **java.awt**, mientras que los componentes Swing están en el paquete **javax.swing**.

La mayor diferencia entre los componentes AWT y los componentes Swing es que éstos últimos están implementados sin nada de código nativo. Esto significa que los componentes Swing pueden tener más funcionalidad que los componentes AWT, porque no están restringidos al denominador común -- las características presentes en cada plataforma. El no tener código nativo también permite que los componentes Swing sean vendidos como añadidos al JDK 1.1, en lugar de sólo formar parte del JDK 1.2.

Incluso el más sencillo de los componentes Swing tiene capacidades que van más allá de lo que ofrecen los componentes AWT. Por ejemplo.

- Los botones y las etiquetas Swing pueden mostrar imágenes en lugar de o además del texto.
- Se pueden añadir o modificar fácilmente los bordes dibujados alrededor de casi cualquier componente Swing. Por ejemplo, es fácil poner una caja alrededor de un contenedor o una etiqueta.
- Se puede modificar fácilmente el comportamiento o la apariencia de un componente Swing llamando a métodos o creando una subclase.
- Los componentes Swing no tienen porque ser rectangulares. Por ejemplo, los botones pueden ser redondos.
- Las tecnologías asistivas como los lectores de pantallas pueden fácilmente obtener información desde los componentes Swing. Por ejemplo, una herramienta puede fácilmente obtener el texto mostrado en un botón o en una etiqueta.

Otra caracterísitca Swing es que se puede especificar el Aspecto y Comportamiento que utilice el GUI de nuestro programa. Por el contrario, los componentes AWT siempre tienen el aspecto y comportamiento de la plataforma nativa.

Otra característica interesante es que los componentes Swing con estado usan modelos para mantener el estado. Por ejemplo, un **JSlider** usa un objeto **BoundedRangeModel** para contener su valor actual y un rango de valores legales. Los modelos se configuran automáticamente, por eso no tenemos que tratar con ellos, a menos que queramos tomar ventaja de la potencia que pueden ofrecernos.

Si estás acostumbrado a usar componentes AWT, necesitarás tener cuidado con algunas reglas cuando uses componentes Swing.

- Como regla, los programas no deberían usar componentne de 'peso pesado' junto con componentes Swing. Los componentes de peso pesado incluyen todos los componentes AWT listos para usar (como **Menu** y **ScrollPane**) y todos los componentes que desciendan de las clases **Canvas** y **Panel** del AWT. Esta restricción existe porque cuando un componente Swing (u otro componente de 'peso ligero') se solapa con componentes de peso pesado, éste último siempre se dibuja encima. Para más información puedes ver Mezclar Componentes de peso ligero y pesado, un artículo de 'The Swing Connection'.

- Los componentes Swing no son de thread seguro. Si modificas un componente Swing visible -- llamando a su método **setText**, por ejemplo -- desde cualquier lugar que no sea el manejador de eventos, necesitas seguir unos pasos especiales para hacer que la modificación se ejecute en el thread de despacho de eventos. Esto no es ningún problema para la mayoría de los programas Swing, ya que el código que modifica los componentes normalmente se encuentra en los manejadores de eventos.
- La herencia de contenidos de cualquier ventana o applet que contenga componentes swing debe tener un contenedor de alto nivel Swing como raíz del árbol. Por ejemplo, una ventana principal debería ser implementada como un ejemplar de **JFrame** en vez de como un ejemplar de **Frame**.
- No se añaden directamente los componentes a un contenedor de alto nivel como un **JFrame**. En su lugar, se añaden los componentes a un contenedor (llamado **panel de contenido**) que a su vez está contenido por el **JFrame**.

Compilar y Ejecutar Programas Swing (JDK 1.2)

Aquí están los pasos para compilar y ejecutar nuestro primer programa Swing con el JDK 1.2 y JFC/Swing.

Descargar la Última Versión del JDK 1.1

Puedes descargar gratis la implementación de referencia del JDK 1.1 desde java.sun.com. Sólo tienes que ir a la página apropiada para tu plataforma -- Solaris o Win32.

Descargar la última versión de JFC/Swing

Puedes desacargar la última versión del JFC 1.1 en la Home Page del JFC. Esta sección describe la versión Swing 1.1 del JFC 1.1.

Crear un Programa que use Componentes Swing

Puedes usar un programa sencillo que nosotros proporcionamos, llamado **SwingApplication**. Por favor, descarga y guarda este fichero: **SwingApplication.java**. El nombre del fichero debe ser exactamente "SwingApplication.java" incluyendo las mayúsculas.

Compilar un Programa que use Componentes Swing

El siguiente paso es compilar el programa. Aquí puedes ver una explicación general de cómo compilar una aplicación Swing con el JDK 1.1.

1. Anota dónde se encuentra tu copia del JFC 1.1 (Swing 1.1). El archivo de clases Swing **swing.jar**, está en el directorio superior de esta versión. Podrías querer crear una variable de entorno llamada **SWING_HOME** que contenga el path del directorio superior de la versión del JFC 1.1.

   Nota: No descomprimas el archivo **swing.jar**!

2. Anota dónde está instalada tu versión del JDK. Necesitas esto para poder encontrar las versiones apropiadas de las clases del JDK y el intérprete. Podrías querer crear una variable de entorno llamada **JAVA_HOME** que contenga el path del directorio superior de la versión del JDK.

   Las clases del JDK están en el directorio **lib** del JDK, en un fichero llamado **classes.zip**. No descomprimas este fichero!. El intérprete Java está en el directorio **bin** del JDK.

3. Compila la aplicación especificando un classpath que incluya el fichero **swing.jar**, el fichero **classes.zip**, y el directorio que contenga las clases del programa (normalmente "."). Asegurate de que el fichero **classes.zip** y el compilador utilizado son exactamente de la misma versión del JDK!.

El siguiente ejemplo muestra cómo compilar **SwingApplication** en un sistema UNIX. Asume que has configurado las variables de entorno **JAVA_HOME** y **SWING_HOME**.

```
$JAVA_HOME/bin/javac -classpath .:$SWING_HOME/swing.jar.
$JAVA_HOME/lib/classes.zip SwingApplication.java
```

Si eliges no usar variables de entorno, podrías usar un comando como éste.

```
javac -classpath .:/home/me/swing-1.1/swing.jar.
/home/me/jdk1.1.7/lib/classes.zip SwingApplication.java
```

Aquí puedes ver un ejemplo de compilación sobre Win32.

```
%JAVA_HOME%\bin\javac -deprecation -classpath
.;%SWING_HOME%\swing.jar;%JAVA_HOME%\lib\classes.zip
SwingApplication.java
```

Aquí puedes ver una alternativa que no usa variables de entorno.

```
javac -deprecation -classpath
.;C:\java\swing-1.1\swing.jar;
C:\java\jdk1.1.7\lib\classes.zip
SwingApplication.java
```

Nota: Si no puedes compilar **SwingApplication.java**, probablemente será debido a que no tienes los ficheros correctos en el classpath o a que estás usando una versión del JFC 1.1 que tiene un API Swing antiguo. Deberías poder ejecutar los programas de esta sección sin cambiarlos si te has actualizado a la versión más reciente del JFC 1.1.

Antes de la Beta 3 de Swing 1.1, el API Swing usaba nombres de paquetes diferentes. Aquí puedes ver cómo modificar **SwingApplication.java** para usar los antiguos nombres de paquetes.

```
//import javax.swing.*;     //comment out this line
import com.sun.java.swing.*;  //uncomment this line
```

Ejecutar el Programa

Una vez que el programa se ha compilado satisfactoriamente, podemos ejecutarlo.

Asegurate de que el classpath del intérprete no sólo incluye lo que necesites para compilar el fichero, sino que también debe incluir el fichero para el Aspecto y Comportamiento que use el programa. El Aspecto y Comportamiento Java, que es el valor por defecto, está en el fichero **swing.jar**. El Aspecto y Comportamiento Windows está en **windows.jar**, y el Aspecto y Comportamiento CDE/Motif está en **motif.jar**. No estás limitado a estas opciones de Aspecto-y-Comportamiento; puedes usar cualquier otro Aspecto y Comportamiento diseñado para usarse con el API de Swing 1.1.

Esta aplicación usa el Aspecto y Comportamiento Java, por eso sólo necesitamos **swing.jar** en el path de clases. Así, el comando para ejecutarlo sería similar al comando para compilarlo. Sólo hay que sustituir **java** por **javac**, y eliminar el sufijo **.java**. Por ejemplo, en UNIX.

```
java -classpath .:/home/me/swing-1.1/swing.jar.

/home/me/jdk1.1.7/lib/classes.zip SwingApplication
```

Aquí hay un ejemplo de ejecución de una aplicación que usa el Aspecto y Comportamiento Windows.

```
%JAVA_HOME%\bin\java -classpath .;%SWING_HOME%\swing.jar;
%JAVA_HOME%\lib\classes.zip;%SWING_HOME%\windows.jar
SomeClass
```

Mientras estás desarrollando tu aplicación puedes simplificar el path de clases usando **swingall.jar**, que incluye todas las clases de la versión JFC 1.1. Por eso en lugar de poner **swing.jar** y **windows.jar** en tu path de clases, por ejemplo, puedes poner sólo **swingall.jar**.

Importante: Evita usar **swingall.jar** en tu aplicación final, contiene información usada por desarrolladores, así cómo los paquetes de Aspecto y Comportamiento que usa una aplicación típica. Puedes disminuir el tamaño usando sólo el fichero **swing.jar** además de los arhivos de Aspecto y Comportamiento que necesites.

Compilar y Ejecutar Programas Swing (JDK 1.1)

Aquí están los pasos para compilar y ejecutar nuestro primer programa Swing con el JDK 1.1 y JFC/Swing.

Descargar la Última Versión del JDK 1.1

Puedes descargar gratis la implementación de referencia del JDK 1.1 desde java.sun.com. Sólo tienes que ir a la página apropiada para tu plataforma -- Solaris o Win32.

Descargar la última versión de JFC/Swing

Puedes desacargar la última versión del JFC 1.1 en la Home Page del JFC. Esta sección describe la versión Swing 1.1 del JFC 1.1.

Crear un Programa que use Componentes Swing

Puedes usar un programa sencillo que nosotros proporcionamos, llamado **SwingApplication**. Por favor, descarga y guarda este fichero: **SwingApplication.java**. El nombre del fichero debe ser exactamente "SwingApplication.java" incluyendo las mayúsculas.

Compilar un Programa que use Componentes Swing

El siguiente paso es compilar el programa. Aquí puedes ver una explicación general de cómo compilar una aplicación Swing con el JDK 1.1.

1. Anota dónde se encuentra tu copia del JFC 1.1 (Swing 1.1). El archivo de clases Swing **swing.jar**, está en el directorio superior de esta versión. Podrías querer crear una variable de entorno llamada **SWING_HOME** que contenga el path del directorio superior de la versión del JFC 1.1.

Nota: No descomprimas el archivo **swing.jar**!

2. Anota dónde está instalada tu versión del JDK. Necesitas esto para poder encontrar las versiones apropiadas de las clases del JDK y el intérprete. Podrías querer crear una variable de entorno llamada **JAVA_HOME** que contenga el path del directorio superior de la versión del JDK.

   Las clases del JDK están en el directorio **lib** del JDK, en un fichero llamado **classes.zip**. No descomprimas este fichero!. El intérprete Java está en el directorio **bin** del JDK.

3. Compila la aplicación especificando un classpath que incluya el fichero **swing.jar**, el fichero **classes.zip**, y el directorio que contenga las clases del programa (normalmente "."). Asegurate de que el fichero **classes.zip** y el compilador utilizado son exactamente de la misma versión del JDK!.

El siguiente ejemplo muestra cómo compilar **SwingApplication** en un sistema UNIX. Asume que has configurado las variables de entorno **JAVA_HOME** y **SWING_HOME**.

```
$JAVA_HOME/bin/javac -classpath .:$SWING_HOME/swing.jar.
$JAVA_HOME/lib/classes.zip SwingApplication.java
```

Si eliges no usar variables de entorno, podrías usar un comando como éste.

```
javac -classpath .:/home/me/swing-1.1/swing.jar.
/home/me/jdk1.1.7/lib/classes.zip SwingApplication.java
```

Aquí puedes ver un ejemplo de compilación sobre Win32.

```
%JAVA_HOME%\bin\javac -deprecation -classpath
.;%SWING_HOME%\swing.jar;%JAVA_HOME%\lib\classes.zip
SwingApplication.java
```

Aquí puedes ver una alternativa que no usa variables de entorno.

```
javac -deprecation -classpath
.;C:\java\swing-1.1\swing.jar;
C:\java\jdk1.1.7\lib\classes.zip
SwingApplication.java
```

Nota: Si no puedes compilar **SwingApplication.java**, probablemente será debido a que no tienes los ficheros correctos en el classpath o a que estás usando una versión del JFC 1.1 que tiene un API Swing antiguo. Deberías poder ejecutar los programas de esta sección sin cambiarlos si te has actualizado a la versión más reciente del JFC 1.1.

Antes de la Beta 3 de Swing 1.1, el API Swing usaba nombres de paquetes diferentes. Aquí puedes ver cómo modificar **SwingApplication.java** para usar los antiguos nombres de paquetes.

```
//import javax.swing.*;     //comment out this line
import com.sun.java.swing.*;  //uncomment this line
```

Ejecutar el Programa

Una vez que el programa se ha compilado satisfactoriamente, podemos ejecutarlo.

Asegurate de que el classpath del intérprete no sólo incluye lo que necesites para compilar el fichero, sino que también debe incluir el fichero para el Aspecto y Comportamiento que use el programa. El Aspecto y Comportamiento Java, que es el valor por defecto, está en el fichero **swing.jar**. El Aspecto y Comportamiento Windows está en **windows.jar**, y el Aspecto y Comportamiento CDE/Motif está en **motif.jar**. No estás limitado a estas opciones de Aspecto-y-Comportamiento; puedes usar cualquier otro Aspecto y Comportamiento diseñado para usarse con el API de Swing 1.1.

Esta aplicación usa el Aspecto y Comportamiento Java, por eso sólo necesitamos **swing.jar** en el path de clases. Así, el comando para ejecutarlo sería similar al comando para compilarlo. Sólo hay que sustituir **java** por **javac**, y eliminar el sufijo **.java**. Por ejemplo, en UNIX.

```
java -classpath .:/home/me/swing-1.1/swing.jar.
/home/me/jdk1.1.7/lib/classes.zip SwingApplication
```

Aquí hay un ejemplo de ejecución de una aplicación que usa el Aspecto y Comportamiento Windows.

```
%JAVA_HOME%\bin\java -classpath .;%SWING_HOME%\swing.jar;
%JAVA_HOME%\lib\classes.zip;%SWING_HOME%\windows.jar
SomeClass
```

Mientras estás desarrollando tu aplicación puedes simplificar el path de clases usando **swingall.jar**, que incluye todas las clases de la versión JFC 1.1. Por eso en lugar de poner **swing.jar** y **windows.jar** en tu path de clases, por ejemplo, puedes poner sólo **swingall.jar**.

Importante: Evita usar **swingall.jar** en tu aplicación final, contiene información usada por desarrolladores, así cómo los paquetes de Aspecto y Comportamiento que usa una aplicación típica. Puedes disminuir el tamaño usando sólo el fichero **swing.jar** además de los arhivos de Aspecto y Comportamiento que necesites.

Ejecutar Applets Swing

Se pueden ejecutar applets Swing en cualquier navegador que tenga instalado el Java Plug-in. Otras opciones son utilizar un navegador que sea complatible con el JDK 1.1 y que pueda encontrar las clases Swing, o un navegador que sea compatible con el JDK 1.2. Actualmente, el único navegador compatible 1.2 disponible es la utilidad AppletViewer proporcionada con el JDK 1.2. Para obtener información sobre como configurar los navegadores compatibles 1.2 para que tabajen con Swing, puedes darte una vuelta por Hacer nuestro Navegador Swing, un artículo de "The Swing Connection".

La única diferencia requerida entre un applet no-swing y otro applet swing, es que éste último debe descender de la clase **JApplet**, en vez de hacerlo directamente de la clase **Applet**.

Para comprobar si tu navegador soporta applets Swing, pulsa sobre la siguiente imagen:

Nota: Como el applet anterior utiliza Java Plug-in 1.1.1, es una versión Swing 1.0.3 del applet. Para ejecutar la versión Swing 1.1 Beta 3 del applet, podemos utilizar el AppletViewer para ver **HelloSwingApplet.html**, especificando **swing.jar** en el path de clases del AppletViewer.

Puedes encontrar el código fuente del applet en **HelloSwingApplet.java**, y el código HTML para incluir el applet viendo el código de ésta misma página. La mala noticia es que el código HTML para incluir el applet es un poco enrevesado. La buena noticia es que se puede generar el código HTML para una sencilla etiqueta **<APPLET>** automáticamente. Puedes ver la documentación del Java Plug-in para obtener detalles sobre cómo descargar un conversor HTML gratis.

Aquí puedes ver un applet más complejo, con múltiples ficheros de clases y un fichero de imagen. El código del applet está en **AppletDemo.java**. También utiliza el fichero **../images/middle.gif**.

El resto de esta página ofrece instrucciones paso a paso para ejecutar los applets anteriores.

Paso a Paso: Ejecutar un Applet Basado en Swing

1. Encontrar un navegador compatible 1.1 ó 1,2, o descargar el Java Plug-in dentro de un navegador compatible. Asegurate de que tienes la última versión tanto del navegador como del plug-in, ya que las últimas versiones tienden a corregir errores que hacen que Swing funcione mejor. Los dos navegadores Java 1.1 son HotJava browser y el Applet Viewer (**appletviewer**), que se distribuye con el JDK. Una versión 1.2 del Appletviewer se distribuye en el JDK 1.2. El Java Plug-in lo soportan varias versiones de Netscape Navigator e Internet Explorer; puedes ver la Documentación del Java Plug-in para obtener más detalles.
2. Si estás utilizando un navegador compatible 1.1 sin el Java Plug-in, determina cómo cargar el fichero Swing JAR en tu navegador. Puedes ver Configurar el Path de Clases del Navegador para ver un ejemplo de cómo poner un fichero JAR de Swing en el path de clases del Appletviewer. Puedes ver Make Your Browser Swing en The Swing Connection para ver cómo hacer lo mismo con Internet Explorer y Netscape Navigator.
3. Apunta a esta página.

Visita Rápida por el Código de un Programa Swing

Esta página te lleva a través del código del programa **SwingApplication**. La siguiente lección proporciona explicaciones completas sobre los tópicos introducidos en esta sección. También proporciona, un ejemplo mayor y más realista, que puedes usar para ampliar y probar tus conocimientos sobre Swing.

La aplicación **SwingAplication** presentada en la sección anterior es muy sencilla. Trae una ventana que se parece a ésta.

Cada vez que el usuario pulsa el botón, la etiqueta se actualiza. Puedes encontrar el programa completo en **SwingApplication.java**.

Esta página explica el programa en detalle, incluyendo los siguientes aspectos.

Importar paquetes Swing

La siguiente línea importa el paquete principal de Swing.

```
import javax.swing.*;
```

Nota: Las primeras versiones beta del JFC/Swing y del JDK 1.2 usaban nombres de paquetes diferentes. Puedes ver Nombres de Paquetes Swing para más detalles.

Los programas Swing también necesitan clases de los paquetes principales del AWT.

```
import java.awt.*;
import java.awt.event.*;
```

Elegir el Aspecto y Comportamiento

Aquí puedes ver el código que utiliza **SwingApplication** para seleccionar su aspecto y comportamiento.

```
public static void main(String [] args) {
try {
    UIManager.setLookAndFeel(
        UIManager.getCrossPlatformLookAndFeelClassName());
      } catch (Exception e) { }

      /Crea y muestra el GUI...
}
```

El código anterior, esencialmente dice, "No me importa si el usuario a elegido un aspecto y comportamiento -- utiliza el aspecto y comportamiento multi-plataforma." Esta aproximación dictatorial tiene sentido cuando el program ha sido diseñado con un aspecto y comportamiento particular en mente. El aspecto y comportamiento multi-plataforma es conocido como Aspecto y Comportamiento Java (con el nick "Metal").

Configurar el Contenedor de Alto Nivel

Todo programa que presente un GUI Swing contiene al menos un contenedor Swing de alto nivel. Para la mayoría de los programas, los contenedores de alto nivel Swing son ejemplares de **JFrame**, **JDialog**, o (para los applets) **JApplet**. Cada objeto **JFrame** implementa una ventana secundaria. Cada objeto **JApplet** implementa un área de pantalla de un applet dentro de una ventana del navegador. Un contenedor de Alto Nivel Swing porporciona el soporte que necesitan los componentes Swing para realizar su dibujado y su manejo de eventos.

El ejemplo **SwingApplication** tiene un sólo contenedor de alto nivel, un **JFrame**. Cuando el usuario cierra el frame, la aplicación finaliza. Aquí está el código que configura y muestra el frame.

```
public class SwingApplication {
    ...
    public static void main(String[] args) {
            ...
        JFrame frame = new JFrame("SwingApplication");
        //...create the components to go into the frame...
        //...stick them in a container named contents...
        frame.getContentPane().add(contents,
                                                   BorderLayout.CENTER);

        //Finish setting up the frame, and show it.
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        frame.pack();
        frame.setVisible(true);
    }
}
```

Para más información sobre los contenedores de alto nivel puedes ver Los Componentes Swing y el Árbol de Herencia de Contenedores

Configurar los Botones y las Etiquetas

Como la mayoría de los GUIs, el ejemplo de **SwingApplication** contiene un botón y una etiqueta. (Al contrario que la mayoría de los GUIs, esto es todo lo que tiene **SwingApplication**). Aquí podemos ver el código que inicializa el botón.

```
JButton button = new JButton("I'm a Swing button!");
button.setMnemonic('i');
button.addActionListener(this);
```

La primera línea crea el botón, La segunda línea selecciona la letra "i" como mnemónico que el usuario puede utilizar para simular un click del botón. Por ejemplo, en el Aspecto y Comportamiento Metal, teclear Alt+i resulta en un click del botón. La tercera línea registra un manejador de eventos para el click del botón. Podremos ver el código del manejador de eventos en Menajear Eventos.

Aquí podemos ver el código que inicializa y manipula la etiqueta.

```
./where instance variables are declared:
private static String labelPrefix = "Number of button clicks: ";
private int numClicks = 0;

./in GUI initialization code:
final JLabel label = new JLabel(labelPrefix + "0    ");
...
label.setLabelFor(button);

./in the event handler for button clicks:
label.setText(labelPrefix + numClicks);
```

El código anterior es bastante conocido, excepto la línea que invoca al método **setLabelFor**. Este código existe sólo para indicar a las tecnologías asistivas que la etiqueta describe el botón. Para más información, puedes ver Supporting Assistive Technologies.

Finalmente, aquí está el código que inicializa el panel.

```
JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createEmptyBorder(30, 30, 10, 30));
pane.setLayout(new GridLayout(0, 1));
pane.add(button);
pane.add(label);
```

Añadir Componentes a los Contenedores

SwingApplication agrupa la etiqueta y el botón en un contenedor (un **JPanel**) antes de añadir los componentes al frame. Aquí está el código que inicializa el panel.

```
JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createEmptyBorder(30, 30, 10, 30));
pane.setLayout(new GridLayout(0, 1));
pane.add(button);
pane.add(label);
```

La primera línea de código crea el panel. La segunda le añade un borde; explicaremos los bordes más tarde.

La tercera línea de código crea un controlador de distribución que fuerza el contenido del panel a dibujarse en una sóla columna. La última línea añade el botón y la etiqueta al panel. Añadir el botón y la etiqueta al panel significa que están controlados por el controlador de distribución del panel. Específicamente, el controlador de distribución de un contenedor determina el tamaño y la posición de cada componente que haya sido añadido al contenedor.

Los conceptos de los controladores de distribución se describen en Controladores de Distribución. Para aprender cómo usar controladores de distribución individuales, puedes ver la lección Usar Controladores de Distribución.

Añadir Bordes Alrededor de los Componentes

Aquí está, de nuevo, el código que añade el borde al **JPanel**.

```
pane.setBorder(BorderFactory.createEmptyBorder(
                                30, //top
                                30, //left
                                10, //bottom
                                30) //LEFT
                                );
```

Este borde simplemente proporciona un espacio en blanco alrededor del panel de contenidos -- 30 pixels extras en la parte superior, izquierda y derecha y 10 pixels extras en la parte inferior. Los bordes son una caracterísitcas que **JPanel** de la clase **Component**.

Los conceptos de los bordes se describen en Control de la distribución y en Dibujo

Manejar Eventos

El ejemplo **SwingApplication** contiene dos manejadores de eventos. Uno maneja las pulsaciones del botón (eventos action) y otro maneja los eventos de cerrar ventana.

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        numClicks++;
        label.setText(labelPrefix + numClicks);
    }
});
...
frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
                System.exit(0);
    }
});
```

Puedes leer sobre el manejo de eventos Swing en Manejo de Eventos y en la lección Escribir Oyentes de Eventos.

Tratar con Problemas de Threads

El programa **SwingApplication** es de thread seguro. Una vez que su GUI es visible, en el manejador de eventos sólo ocurre manipulación GUI (actualizar la etiqueta). Como el manejador de eventos se ejecuta en el mismo thread que realiza todo el manejo de eventos y pintado de la aplicación, no existe la posibilidad de que dos threads intenten manipular el GUI a la vez.

Sin embargo, es fácil introducir problemas de threads en un programa.

Puedes ver Threads y Swing para más información sobre los threads seguros en Swing.

Soportar Tecnologías Asistivas

El soporte para tecnologías asistivas -- dispositivos como lectores de pantalla que proporcionan formas alternativas de acceder a la información de un GUI -- ya está incluido en cada componente Swing. El único código que existen en **SwingApplication** que maneja el soporte de tecnologías asistivas es este.

```
label.setLabelFor(button);
```

Además, la siguientes líneas seleccionan la información que puede ser utilizada por tecnologías asistivas.

```
super("HelloSwing");
JButton button = new JButton("I'm a Swing button!");
label = new JLabel(labelPrefix + "0    ");
label.setText(labelPrefix + numClicks);
JFrame frame = new JFrame("SwingApplication");
```

Herencia de Componentes y Contenedores

Esta sección presenta algunos de los componentes más utilizados de Swing y explica como los componentes de un GUI entran juntos en un contenedor. Para ilustrarlo, usaremos el programa **SwingApplication** presentado en Una Ruta Rápida por el Código de una Aplicación Swing. Y aquí está su aspecto de nuevo.

SwingApplication crea cuatro componentes Swing muy utilizados.

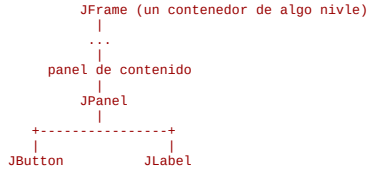
- un frame, o ventana principal (**JFrame**)
- un panel, algunas veces llamado pane (**JPanel**)
- un botón (**JButton**)
- una etiqueta (**JLabel**)

El frame es un contenedor de alto nivel. Existe principalmente para proporcionar espacio para que se dibujen otros componentes Swing. Los otros contenedores de alto nivel más utilizados son los diálogos (**JDialog**) y los applets (**JApplet**).

El panel es un **contenedor intermedio**. Su único propósito es simplificar el posicionamiento del botón y la etiqueta. Otros contenedores intermedios, como los paneles desplazables, (**JScrollPane**) y los paneles con pestañas (**JTabbedPane**), típicamente juegan un papel más visible e interactivo en el GUI de un programa.

El botón y la etiqueta son **componentes atómicos** -- componentes que existen no para contener otros componentes Swing, sino como entidades auto-suficientes que representan bits de informaión para el usuario. Frecuentemente, los componentes atómicos también obtienen entrada del usuario. El API Swing proporciona muchos componentes atómicos, incluyendo combo boxes (**JComboBox**), campos de texto (**JTextField**), y tablas (**JTable**).

Aquí podemos ver un diágrama con el **árbol de contenidos** de la ventana mostrada por **SwingApplication**. Este diágrama muestra todos los contenedores creados o usados por el programa, junto con los componentes que contienen. Observa que si añadimos una ventana -- por ejemplo, un diálogo -- la nueva ventana tendría su propio árbol de contenidos, independiente del mostrado en esta figura.



Como muestra la figura, incluso el programa Swing más sencillo tiene múltiples niveles en su árbol de contenidos. La raíz del árbol de contenidos es siempre un contenedor de alto nivel. Este contenedor proporciona espacio para que sus componentes Swing descendentes se dibujen a sí mismo.

Truco: Para ver el árbol de contenidos de cualquier frame o diálogo, pulsa el borde para seleccionarlo y pulsa Control-Shift-F1. Se escribirá una lista con el árbol de contenidos en el stream de salida estándard.

Todo contenedor de alto nivel contiene indirectamente un contenedor intermedio conocido como **panel de contenido**. Para la mayoría de los programas no necesitas saber qué pasa entre el contenedor de alto nivel y su panel de contenido. (Si realmente quieres verlo, puedes ver Cómo usar Paneles Raíz.)

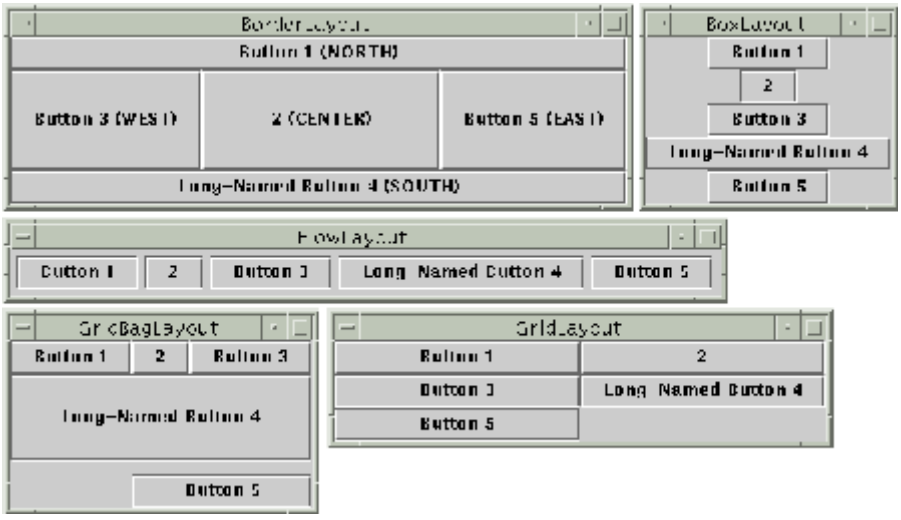
Cómo regla general, el panel de contenido contiene, directamente o indirectamente, todos los componentes visibles en el GUI de la ventana. La gran excepción a esta regla es que si el contenedor de alto nivel tiene una barra de menú, entonces ésta se sitúa en un lugar especial fuera del panel de contenido.

Para añadir componentes a un contenedor, se usa una de las distintas formas del método **add**. Este método tiene al menos un argumento -- el componente a añadir. Algunas veces se requiere un argumento adicional para proporcionan información de distribución. Por ejemplo, la última línea del siguiente código de ejemplo especifica que el panel debería estar en el centro de su contenedor (el panel de contenido).

```
frame = new JFrame(...);
button = new JButton(...);
label = new JLabel(...);
pane = new JPanel();
pane.add(button);
pane.add(label);
frame.getContentPane().add(pane, BorderLayout.CENTER);
```

Control de Distribución

Las siguientes figuras muestran los GUIs de cinco programas, cada uno de ellos muestra cinco botones. Los botones son idénticos, y el código de los programas es casi idéntico. ¿Entonces por qué parecen tan diferentes? Porque usan diferentes controladores de distribución para controlar el tamaño y posición de los botones.



Control de Distribución es el proceso de determinar el tamaño y posición de los componentes. Por defecto, cada contenedor tiene un **controlador de distribución** -- un objeto que realiza el control de la distribución de los componentes dentro del contenedor. Los componentes pueden proporcionarle al controlador de disposición sus preferencias en cuanto a tamaño y alineamiento, pero la última palabra la tiene el controlador de disposición.

La plataforma Java suministra cinco controladores de disposición comunmente utilizados: **BorderLayout**, **BoxLayout**, **FlowLayout**, **GridBagLayout**, y **GridLayout**. Estos controladores de distribución están diseñados para mostrar múltiples componentes a la vez, y se han visto en la figura anterior. Una sexta clase proporcionada, **CardLayout**, es un controlador de disposición de propósito general usado en combinación con otros controladores de distribución. Puedes encontrar detalles sobre cada uno de estos seis controladores, incluyendo claves para elegir el apropiado, en Usar Controladores de Distribución.

Siempre que se use el método **add** para poner un componente en un contenedor, debemos tener en cuenta el controlador de distribución del contenedor. Algunos controladores como **BorderLayout** requiere que especifiquemos la posición relativa del componente en el contenedor, usando un argumento extra para el método **add**. Ocasionalmente, un controlador de distribución como **GridBagLayout** requiere elaborados procesos de configuración. Sin embargo, muchos controladores de distribución simplemente sitúan los componentes en el orden en que fueron añadidos a su contenedor.

Todos esto probablemente suena más complicado de lo que és. Si quieres puedes copiar el código de nuestros ejemplos de **Usar Componentes Swing** o buscar el controlador de distribución individual en Usar Controladores de Distribución. Generalmente, sólo tendrás que seleccionar el controlador de distribución de dos tipos de contenedores: paneles de contenido (que usan **BorderLayout** por defecto) y **JPanel** (que usan **FlowLayout** por defecto).

El resto de esta sección describe algunas de las tareas más comunes de la distribución.

Seleccionar el Controlador de Distribución

Podemos cambiar fácilmente el controlador de distribución que usa un contenedor. Sólo se debe llamar al método **setLayout** del contenedor. Por ejemplo, aquí está el código que hace que un panel use **BorderLayout**.

```
JPanel pane = new JPanel();
pane.setLayout(new BorderLayout());
```

Aunque recomendamos que uses controladores de distribución, se puede realizar la distribución sin ellos. Seleccionando una propiedad de distribución del contenedor a nulo, podemos hacer que el contenedor no use ningún controlador de distribución. Con este esquema, llamado **posicionamiento absoluto**, podemos especificar el tamaño y posición de cada componente dentro del contenedor. Una desventaja del posicionamiento absoluto es que no se ajusta bien cuando se redimensiona el contenedor de alto nivel. Tampoco se ajusta bien a las diferencias entres usuarios y sistemas, como los diferentes tamaños de fuente.

Proporcionar Consejos sobre un Componente

Algunas veces necesitamos personalizar el tamaño que un componente proporciona al controlador de distribución del contenedor, para que el componente se vea bien. Se puede hacer esto proporcionando los tamaños mínimo, preferido y máximo del componente. También podemos llamar a los métodos de selección de tamaño del componente -- **setMinimumSize**, **setPreferredSize**, y **setMaximumSize** -- o podemos crear una subclase del componente que sobreescriba los métodos apropiados -- **getMinimumSize**, **getPreferredSize**, y **getMaximumSize**. Actualmente, el único controlador de distribución en la plataforma Java que presta atención a la petición de tamaño máximo del componente es **BoxLayout**.

Además de proporcionar preferencias de tamaño, podemos especificar preferencias de alineamiento. Por ejemplo, podemos especificar que los bordes superiores de dos componentes deberían estar alineados. Se seleccionan los consejos de alineamiento llamando a los métodos **setAlignmentX** y **setAlignmentY** del componente, o sobreescribiendo los métodos, **getAlignmentX** y **getAlignmentY** del componente. Realmente **BoxLayout** es el único controlador de distribución que presta atención a los consejos de alineamiento.

Poner Espacio entre Componentes

Tres factores influyen en la cantidad de espacio entre los componentes visibles de un contenedor.

El controlador de distribución
: Algunos controladores de distribución ponen automáticamente espacio entre los componentes; otros no. Algunos permiten incluso especificar la cantidad de espacio entre los componentes. Puedes ver Distribuir Componentes dentro de un Contenedor sobre el soporte de espaciado de cada controlador de distribución.

Componentes invisibles.
: Se pueden crear componentes de peso ligero que no realicen dibujo, pero que ocupen espacio en el GUI. Frecuentemente se usan los componentes invisibles en contenedores controlados por **BoxLayout**. Puedes ver Cómo usar BoxLayout para ver ejemplos de uso de componentes invisibles.

Bordes vacíos
: No importa cual sea el controlador de distribución, podemos afectar la aparente cantidad de espacio entre componentes añadiendoles bordes. Los mejores candidatos para los bordes vacíos son los que típicamente no tienen bordes, como los paneles y las etiquetas. Algunos otros componentes, como paneles desplazables, no funcionan bien con bordes en algunas implementaciones del Aspecto y Comportamiento, debido a la forma en que implementan su código de dibujo. Para más información sobre los bordes puedes ver Cómo usar Bordes.

Cómo Ocurre el Control de Distribución

Aquí hay un ejemplo de secuencia de control de distribución para un frame (**JFrame**).

1. Después de que el GUI está construido, se llama al método **pack** sobre el **JFrame**. Esto especifica que el frame debería ser de su tamaño preferido.
2. Para encontrar el tamaño preferido del frame, el controlador de distribución añade el tamaño de los lados del frame al tamaño preferido del componente directamente contenido por el frame. Esto es la suma del tamaño preferido del panel de contenido, más el tamaño de la barra de menú del frame, si existe.
3. El controlador de disposición del panel de contenido es responsable de imaginarse el tamaño preferido del panel de contenido. Por defecto, este controlador de disposición es un objeto **BorderLayout**. Sin embargo, asumamos que lo hemos reemplazado con un objeto **GridLayout** que se ha configurado para crear dos columnas. Lo interesante de gridlayout es que fuerza a que todos los componentes sean del mismo tamaño, e intenta hacerlos tan anchos como la anchura preferida del componente más ancho, y tan altos como la altura preferida del componente más alto.

   Primero, el controlador gridlayout pregunta el panel de contenido por su insets -- el tamaño del borde del panel de contenido, si existe. Luego, el controlador de gridlayout le pregunta a cada componente del panel de contenido sus tamaños preferidos, anotando la mayor anchura preferida y la mayor altura preferia. Luego calcula el tamaño preferido del panel de contenido.

4. Cuando a cada botón se le pide su tamaño preferido, el botón primero comprueba si el usuario ha especificado un tamaño preferido. Si es así, reporta este tamaño. Si no es así, le pregunta a su Aspecto y Comportamiento el tamaño preferido.

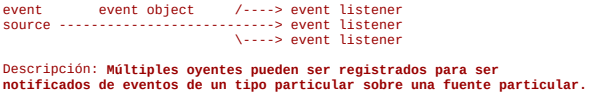
El resultado final es que para determinar el mejor tamaño de un frame, el sitema determina los tamaños de los contenedores en la parte inferior del árbol de contenidos. Estos tamaños filtran el árbol de contenidos, eventualmente determinan el tamaño total del frame. De forma similar ocurren los cálculos cuando se redimensiona el frame.

Manejo de Eventos

Cada vez que el usuario teclea un caracter o pulsa un botón del ratón, ocurre un evento. Cualquier puede ser notificado del evento. Todo lo que tiene que hacer es implementar el interface apropiado y ser registrado como un **oyente de evento** del **evento fuente** apropiado. Los componentes Swing puede generar muchas clases de evento. Aquí hay unos pocos ejemplos.

Acción que resulta en el evento	Tipo de oyente
El usuario pulsa un botón, presiona Return mientras teclea en un campo de texto, o elige un ítem de menú.	**ActionListener**
El usuario elige un frame (ventana principal).	**WindowListener**
El usuario pulsa un botón del ratón mientras el cursor está sobre un componente.	**MouseListener**
El usuario mueve el cursor sobre un componente.	**MouseMotionListener**
El componente se hace visible.	**ComponentListener**
El componente obtiene obtiene el foco del teclado.	**FocusListener**
Cambia la tabla o la selección de una lista.	**ListSelectionListener**

Cada evento está representado por un objeto que ofrece información sobre el evento e identifica la fuente. Las fuentes de los eventos normalmente son componentes, pero otros tipos de objetos también pueden ser fuente de eventos. Como muestra la siguiente figura, cada fuente de evento puede tener varios oyentes registrados. Inversamente, un sólo oyente puede registrarse con varias fuentes de eventos.



Descripción: **Múltiples oyentes pueden ser registrados para ser notificados de eventos de un tipo particular sobre una fuente particular.**

Cuando queramos detectar los eventos de un componente determinado, primero debemos chequear la sección "Cómo se usa" el componente. Esta sección ofrece ejemplos de manejo de eventos de los que querrás utilizar. Por ejemplo, Cómo crear Frames ofrece un ejemplo de escritura de un oyente de window que sale de la aplicación cuando se cierra el frame.

Cómo Implementar un Manejador de Eventos

Todo manejador de eventos requiere tres partes de código.

1. Donde se declare la clase del manejador de eventos, el código especifica que la clase o implementa un interface de oyente, o desciende una clase que implementa un interface de oyente. Por ejemplo.

```
2.      public class MyClass implements ActionListener {
```

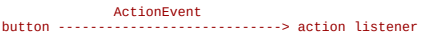
3. El código que registra un ejemplar de la clase de manejo de eventos de un oynete sobre uno o más componentes. Por ejemplo.

```
4.      someComponent.addActionListener(instanceOfMyClass);
```

5. La implementación de los métodos del interface oyente. Por ejemplo.

```
6.      public void actionPerformed(ActionEvent e) {
7.          ...//code that reacts to the action...
8.      }
```

Un escenario de manejo de eventos típico ocurre con los botones (**JButton**). Para detectar cuando el usuario pulsa un botón de la pantalla (o pulsa la tecla equivalente), un programa debe tener un objeto que implementa el interface **ActionListener**. El programa debe registrar este objeto como un oyente de actión del botón (la fuente del evento), usando el método **addActionListener**. Cuando el usuario pulsa el botón de la pantalla, éste dispara el evento action, Esto resulta en una llamada al método **actionPerformed** del oyente de action, el único método del interface **ActionListener**). El único argumento del método es un objeto **ActionEvent** que ofrece información sobre el evento y su fuente.



Descripción: **Cuando el usuario pulsa un botón, los oyentes de action del botón son notificados.**

Los manejadores de eventos pueden ser ejemplares de cualquier clase. Frecuentemente, se implementan usando clases internas anónimas, lo que puede hacer el códio algo más confuso -- hasta que hayas usado las clases internas. Para más información sobre el uso de clases internas, puedes ver Usar Adaptadores y Clases Internas para Manejar Eventos.

Para más información sobre la implementación de manejo de eventos, puedes ver Escribir Oyentes de Eventos.

Los Threads y el Manejo de Eventos

El código de manejo de eventos se ejecuta en un sólo threadm el **thread de despacho de eventos**. Esto asegura que todo manejador de eventos se terminará de ejecutar antes de ejecutar otro. Por ejemplo, el método **actionPerformed** del ejemplo anterior se ejecuta en el thread de manejo de eventos. El código de dibujo también se realiza en el thread de despacho de eventos. Esto significa que mientras

se está ejecutando el método **actionPerformed**, el GUI del programa está congelado -- no se dibujará nada ni se responderá a las pulsaciones del ratón, por ejemplo.

Importante: El código de manejo de eventos debería ejecutar cada pulsación! De otro modo, el rendimiento de tu programa se vería empobrecido. Si necesitas hacer una operación de larga operación como resultado de un evento, hazlo arrancando un nuevo thread (o de otro modo enviándo una petición a otro thread) para realizar la operación. Para obtener ayuda sobre le uso de threads, puedes ver Cómo usar Threads.

Dibujo

Podrías no necesitar la información de esta sección, en absoluto. Sin embargo, si tus componentes parece que no se dibujan correctamente, entender los conceptos de esta sección podría ayudarte a ver qué hay erróneo. De igual modo, necesitarás entender esta sección si creas código de dibujo personalizado para un componente.

Cómo funciona el dibujo

Cuando un GUI Swing necesita dibujarse a sí mismo -- la primera vez, o en respuesta a la vuelta de un ocultamiento, o porque necesita reflejar un cambio en el estado del programa -- empieza con el componente más alto que necesita ser redibujado y va bajando por el árbol de contenidos. Esto está orquestado por el sistema de dibujo del AWT, y se ha hecho más eficiente mediante el manejador de dibujo de Swing y el código de doble buffer.

Los componentes Swing generalmente se redibujan a sí mismos siempre que es necesario. Por ejemplo, cuando llamamos al método **setText** de un componente, el cómponente debería redibujarse automáticamente a sí mismo, y si es necesario, redimensionarse. Si no lo hace así es un bug. El atajo es llamar al método **repaint** sobre el componente para pedir que el componente se ponga en la cola para redibujado. Si se necesita cambiar el tamaño o la posición del componente pero no automáticamente, deberíamos llamar al método **revalidate** sobre el componente antes de llamar a **repaint**.

Al igual que el código de manejo de eventos, el código de dibujo se ejecuta en el thread del despacho de eventos. Mientras se esté manejando un evento no ocurrirá ningún dibujo. De forma similar, si la operación de dibujado tarda mucho tiempo, no se manejará ningún evento durante ese tiempo.

Los programas sólo deberían dibujarse cuando el sistema de dibujo se lo diga. La razón es que cada ocurrencia de dibujo de un propio componente debe ser ejecutado sin interrupción. De otro modo, podrían ocurrir resultados imprecedibles. como que un botón fuera dibujado medio pulsado o medio liberado.

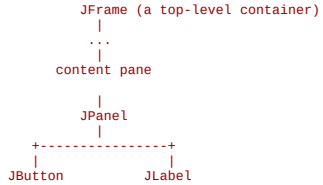
Para acelerar, el dibujo Swing usa **doble-buffer** por defecto -- realizado en un buffer fuera de pantalla y luego lanzado a la pantalla una vez finalizado. Podría ayudar al rendimiento si hacemos un componente Swing opaco, para que el sistema de dibujo de Swing pueda conocer lo que no tiene que pintar detrás del componente. Para hacer opaco un componente Swing, se llama al método **setOpaque(true)** sobre el componente.

Los componentes no-opacos de Swing puede parecer que tienen cualquier forma, aunque su área de dibujo disponible es siempre rectangular. Por ejemplo, un botón podría dibujarse a sí mismo dibujando un octógono relleno. El componente detrás del botón, (su contenedor, comunmente) sería visible, a través de las esquinas de los lados del botón. El botón podría necesitar incluir código especial de detección para evitar que un evento action cuando el usuario pulsa en las esquinas del botón.

Un Ejemplo de Dibujo

Para ilustrar el dibujado, usaremos el programa **SwingApplication**, que se explicó en Una Ruta Rápida a través del Código de una Aplicación Swing. Aquí podemos ver el GUI de **SwingApplication**.

Y aquí su árbol de contenidos.



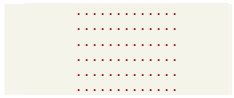
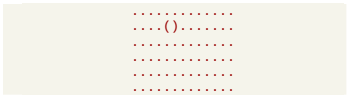
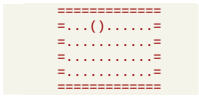
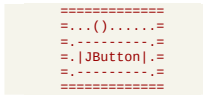
Aquí está lo que sucede cuando se dibuja el GUI de **SwingApplication**.

1. El contenedor de alto nivel, **JFrame**, se dibuja as sí mismo.
2. El panel de contenido primero dibuja su fondo, que es un rectángulo sólido de color gris. Luego le dice al **JPanel** que se dibuje el mismo. El rectángulo del panel de contenido realmente no aparece en el GUI finalizado porque está oscurecido por el **JPanel**.

Nota: Es importante que el panel de contenido sea opaco. De otro modo, resultará en dibujados confusos. Cómo el **JPanel** es opaco, podemos hacer que sea el panel de contenido (sustituyendo **setContentPane** por el código existente **getContentPane().add**). Esto simplifica considerablemente el árbol de contenidos y el dibujado, eliminado un contenedor innecesario.

3. El **JPanel** primero dibuja su fondo, un rectángulo sólido de color gris. Luego dibuja su borde. Esl borde es un **EmptyBorder**, que no tendrá efecto excepto para incrementar el tamaño del **JPanel** reservando algún espacio extra en los laterales del panel. Finalmente, el panel le pide a sus hijos que se dibujen a sí mismos.
4. Para dibujarse a sí mismo, el **JButton** dibuja su rectángulo de fondo si es necesario y luego dibuja el texto que contiene. Si el botón tiene el foco del teclado, significa que cualquier cosa que se teclee va directamente al botón para su procesamiento, luego el botón realiza algún dibujado específico del Aspecto y Comportamiento para aclarar que tiene el foco.
5. Para dibujarse a sí misma, la **JLabel** dibuja su texto.

De este modo, cada componente se dibuja a sí mismo antes de que lo haga cualquier componente que contenga, Esto asegura que el fondo de un **JPanel**, por ejemplo, sólo se dibuja cuando no está cubierto por uno de los componentes que contiene. La siguiente figura ilustra el orden en que cada componente que desciende de **JComponent** se dibuja a sí mismo.

1. fondo (si es opaco)	2. dibujo personalizado(si existe)	3. borde (si existe)	4. hijos (si existen)
			

Más Características Swing

Esta lección ha explicado algunos de los mejores conceptos que necesitarás conocer para construir GUIs Swing -- el árbol de contenidos, el control de distribución, el manejo de eventos, el dibujado, y los threads. Además, hemos tocado tópicos relacionados, como los bordes. Esta sección explica algunas características Swing que no se han explicado todavía.

Características que ProporcionaJComponent

Excepto los contenedores de alto nivel, todos los componentes que empiezan con **J** descienden de la clase **JComponent**. Obtienen muchas características de esta clase, como la posibilidad de tener bordes, tooltips, y Aspecto y Comportamiento configurable. También heredan muchos métodos de conveniencia. Para más detalles, sobre lo que proporciona la clase **JComponent** puedes ver La clase JComponent

Iconos

Muchos componentes Swing -- principalmente los botones y las etiquetas -- pueden mostrar imágenes. Estas imágenes se especifican como objetos **Icon**. Puedes ver Cómo usar Iconos para ver instrucciones y una lista de ejemplos que usa iconos.

Actions

Con objetos **Action**, el API Swing proporciona un soporte especial para compartir datos y estados entre dos o más componentes que pueden generar eventos action. Por ejemplo, si tenemos un botón y un ítem de menú que realizan la misma función, podríamos considerar la utilización de un objeto **Action** para coordinar el texto, el icono y el estado de activado de los dos componentes. Para más detalles, puedes ver Cómo usar Actions.

Aspecto y Comportamiento Conectable

Un sencillo programa puede tener uno o varios aspectos y comportamientos. Se puede permitir que el usuario determine el aspecto y comportamiento, o podemos determinarlos programáticamente. Puedes ver Cómo seleccionar el Aspecto y Comportamiento para más detalles.

Soporte para Tecnologías Asistivas

Las tecnologías asisticas como los lectores de pantallas pueden usar el API de accesibilidad para obtener información sobre los componentes Swing. Incluso si no hacemos nada, nuestro programa Swing probablemente funcionará correctamente con tecnologías asistivas, ya que el API de accesibilidad está construido internamente en los componentes Swing. Sin embargo, con un pequeño esfuerzo extra, podemos hacer que nuestro programa funcione todavía mejor con tecnologías asistivas, lo que podría expandir el mercado de nuestro programa. Puedes ver Cómo Soportar Tecnologías Asistivas para más detalles.

Módelos de Datos y Estados Separados

La mayoría de los componentes Swing no-contenedores tienen modelos. Por ejemplo, un botón (**JButton**) tiene un modelo (**ButtonModel**) que almacena el estado del botón -- cuál es su mnemónico de teclado, si está activado, seleccionado o pulsado, etc. Algunos componentes tienen múltiples modelos. Por ejemplo, una lista (**JList**) usa un **ListModel** que almacena los contenidos de la lista y un **ListSelectionModel** que sigue la pista de la selección actual de la lista.

Normalmente no necesitamos conocer los modelos que usa un componente. Por ejemplo, casi todos los programas que usan botones tratan directamente con el objeto **JButton**, y no lo hacen en absoluto con el objeto **ButtonModel**.

Entonces ¿Por qué existen modelos separados? Porque ofrecen la posibilidad de trabajar con componentes más eficientemente y para compartir fácilmente datos y estados entre componentes. Un caso común es cuando un componente, como una lista o una tabla, contiene muchos datos. Puede ser mucho más rápido manejar los datos trabajando directamente con un modelo de datos que tener que esperar a cada petición de datos al modelo. Podemos usar el modelo por defecto del componente o implementar uno propio.

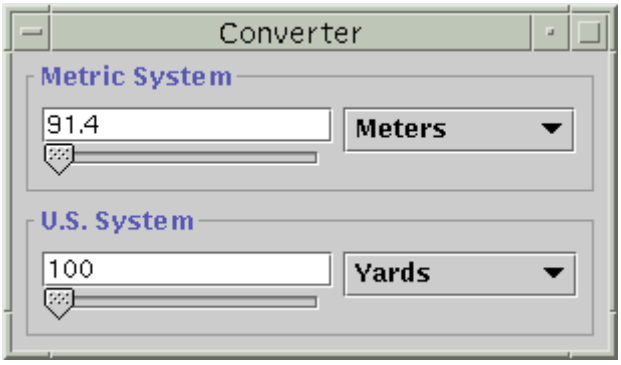
Para más información sobre los modelos, puedes visitar las páginas individuales de cada componente o La Anatomía de un Programa Basado en Swing que describe algunos modelos personalizados usados por el programa creado en esa sección.

Anatomía de un Programa Swing

Esta sección mustra un programa Swing, llamado **Converter**, que tiene un UI gráfico. Puedes ver cómo está implementado este programa mirando el código fuente que puedes encontrar en los ficheros **Converter.java** y **ConversionPanel.java**. Sin embargo, no habla sobre líneas individuales de código. Se concentra en cómo el programa **Converter** usa las características GUI proporcionadas por la plataforma Java. Si te pierdes cuando lees el código fuente de **Converter**, puedes refrescar tu memoria en la página Una Ruta Rápida por el Código de una Aplicación Swing.

Converter es una aplicación que convierte medidas de distancias entre unidades métricas y americanas. Para ejecutarlo, debes compilar los siguientes ficheros fuente: **Converter.java**, **ConversionPanel.java**, **ConverterRangeModel.java**, **FollowerRangeModel.java**, **DecimalField.java**, **FormattedDocument.java**, y **Unit.java**. Una vez compilado, puedes ejecutarlo llamando al intérprete con la clase **Converter**. Si necesitas ayuda para compilar y ejecutar **Converter**, puedes ver Compilar y Ejecutar un Programa Swing.

Aquí tenemos un gráfico comentado del GUI de **Converter**.




```
"JFrame",
"JPanel (ConversionPanel)"
(x2 apuntado a los paneles que dicen "Metric System" and "U.S. System"),
"JTextField (DecimalField)" (x2),
JSlider (x2),
JComboBox (x2)]
```

Esta sección describe las siguientes caracterísitcas de **Converter**.

Componentes Swing

Cómo se ve en la figura anterior, **Converter** tiene los siguientes componentes visibles.

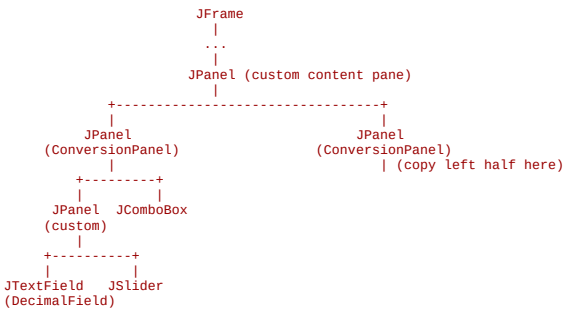
- 1 **JFrame**
- 2 **JPanel** personalizados
- 2 **JTextField** personalizados
- 2 **JSlider**s
- 2 **JComboBox**es

El **JFrame** es el contenedor de alto nivel, sólo proporciona la ventana de la aplicación. Todos los otros componentes de la aplicación están contenidos en **JFrame**.

Excepto el contenedor de alto nivel, todos los componentes visibles de **Converter** descienden de **JComponent**. La clase **JComponent** proporciona muchas características, como soporte para bordes y accesibilidad. Los dos **JPanel** personalizados usan bordes para soportar títulos (por ejemplo, "Metric System") y para dibujar recuadros a su alrededor.

El Árbol de Contenidos

La siguiente figura muesta el árbol de contenidos del **JFrame**.



Este diagrama muestra tres componentes no etiquetados en el figura anterior porque no dibujan nada observable en la pantalla..

- 1 **JPanel** que sirve para el panel de contenidos
- 2 **JPanel** personalizados que contienen un campo de texto y un deslizador

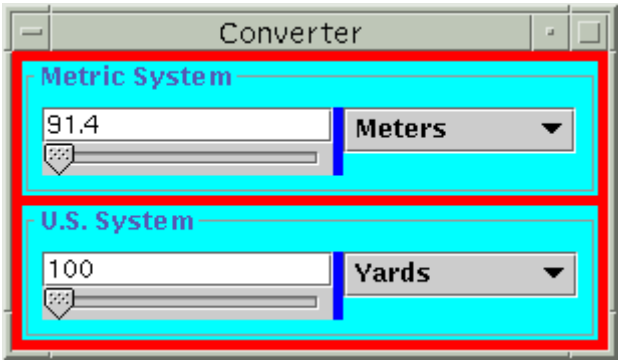
Estos tres componentes existen para afectar a la distribución. Hacen esto simplificando la distribución o añadiendo bordes "vacíos" que añaden espacio para la distribución. El agrupamiento de componentes -- tanto en contenedores visibles como en invisibles -- tambien proporciona ayuda para las tecnologías asistivas. Por ejemplo, agrupar un campo de texto y un deslizador en su propio contenedor le ofrece a las tecnologías asisitivas la información de que el campo de texto y el deslizador están estrechamente relacionados.

Bajo el panel de contenidos hay dos **ConversionPanel**. Uno de ellos contiene los componentes relacionados con las ditancias métricas, y el otro hace lo mismo con las distancias americanas.

Cada **ConversionPanel** contiene 3 componentes visibles: un campo de texto, un deslizador y un combo box. El campo de texto y el deslizador están agrupados en un **JPanel**, principalmente para simplificar la distribución.

Control de Distribución y Bordes

La siguiente figura muesta una versión coloreada de **Converter**. En esta versión, cada contenedor tiene un color de fondo diferente, para que puedes ver fácilmente las partes de los contenedores que no están cubiertas por otros componentes. Observa que todos los contenedores son opacos; de otro modo, el color del fondo podría no ser pintado automáticamente.



Converter create cinco objetos controladores de distribución -- un ejemplar de **GridLayout**, y cuatro de **BoxLayout**.

El primer **JPanel** (el panel de contenidos personalizado) usa **GridLayout** para hacer que los **ConversionPanel**s tengan exactamente el mismo tamaño. El código configura el **GridLayout** para que ponga los **ConversionPanel**s en una sóla columna (dos filas), con cinco pixels entre ellos. El **JPanel** se inicializa para tener un borde vacío que añade cinco pixels entre el panel y los lados del frame.

Cada **ConversionPanel** tiene un borde compuesto. El exterior es un borde con título y el interior es un borde vacío. El borde titulado dibuja una recuadro específico del aspecto y comportamiento alrededor del **ConversionPanel** y sitúa dentro el título del panel. El borde vacío pone algún espacio más entre el **ConversionPanel** y sus contenidos.

Cada **ConversionPanel** usa un controlador **BoxLayout** para situar sus contenidos, que son un **JPanel** y un **JComboBox**, en un fila. Seleccionando el alineamineto Y tanto del panel como del combo box, el programa alinea la parte superior del panel con la perte superior del combo box.

El **JPanel** que agrupa al campo de texto y al deslizador está implementado con una subclase sin nombre de **JPanel**. Esta subclase sobreescribe los métodos **getMinimumSize**, **getPreferredSize**, y **getMaximumSize** para que devuelvan el mismo valor: 150 pixels de

ancho y la altura preferida. Así es como se asegura de que los dos grupos de texto-deslizador tienen la misma anchura, incluso aunque estén controlados por diferentes controladores de distribución. Necesitamos crear una subclase de **JPanel**, en vez de llamar a los métodos **setXxxxSize**, porque la altura preferida de los componentes se determina en tiempo de ejecución, por el controlador de distribución.

El **JPanel** que agrupa el campo de texto y el deslizador usa un controlador **BoxLayout** de arriba-a-abajo para que el campo de texto se situe encima del deslizador. El **JPanel** también tiene un borde vacío que añade un poco de espacio a su derecha entre él y el combo box.

Modelos Separados

Este programa usa tres modelos personalizados. El primero es un modelo de datos para el campo de texto. Los modelos de datos de texto son conocidos como modelos de documento. El modelo de documento analiza el valor que el usuario introduce en el campo de texto. También formatea el número para que parezca bonito. Hemos tomado prestado este modelo de documento, sin cambiarlo, del ejemplo presentado en Crear y Validar y un Campo de Texto.

Los otros dos modelos personalizados son modelos de datos de deslizador. Esto asegura que el dato mostrado por la aplicación esté en un sólo lugar -- en el modelo del deslizador superior. El modelo del deslizador superior es un ejemplar de una clase personalizada llamada **ConverterRangeModel**. El deslizador inferior usa una segunda clase personalizada, **FollowerRangeModel**, que reenvía todas las peticiones para obtener un conjunto de datos al modelo del deslizados superior.

Todos los modelos de datos de deslizador deben implementar el interface **BoundedRangeModel**. Aprenderemos esto en la sección del API de Cómo usar Sliders. La documentación del **BoundedRangeModel** que tiene un clase de implementación llamada **DefaultBoundedRangeModel**. La documentación del API de **DefaultBoundedRangeModel** muestra que es una implementación de código general de **BoundedRangeModel**.

No usamos **DefaultBoundedRangeModel** porque almacena los datos como enteros, y necesitamos almacenar datos en coma flotante. Así, implementamos **ConverterRangeModel** como una subclase de **Object**, comparálo con el código fuente de **DefaultBoundedRangeModel** (distribuido con las versiones JFC 1.1y JDK 1.2), para asegurarnos que implementamos el modelo correctamente. Implementamos **FollowerRangeModel** como una subclase de **ConverterRangeModel**.

Aspecto y Comportamiento Conectable

El programa **Converter** se configura a sí mismo para usar el Aspecto y Comportamiento Java. Cambiando el valor de su variable **LOOKANDFEEL**, podemos hacer que use un Aspecto y Comportamiento diferente. Tres de sus encarnaciones están dibujadas en ¿Qué son el JFC y Swing?.

Manejo de Eventos

El programa **Converter** crea varios manejadores de eventos.

Oyentes de Action

Cada combo box tiene un oyente de action. Siempre que el usuario selecciona una nueva unidad de medida, el oyente de action notifica el modelo de deslizador relevante y resetea los valores máximos de los dos deslizadores.

Cada campo de texto tiene un oyente de action que es notificado cuando el usuario pulsa la tecla Return para indicar que el tecleado ha terminado. Este oyente de action actualiza el correspondiente modelo de deslizador para reflejar el valor del campo de texto.

Oyente de Change

Cada modelo de deslizador tiene un oyente de change personalizado. Siempre que el valor de un deslizador cambia, este oyente actualiza el correspondiente campo de texto. No hemos tenido que registrar los deslizadores como oyentes de dus propios modelos, ya que lo hace Swing automáticamente. En otras palabras, siempre que el programa selecciona un valor en un modelo de deslizador, éste se actualiza automáticamente para reflejar el nuevo estado del modelo.

El modelo para el deslizador inferior añade un oyente de cambio al modelo del deslizador superior, Este oyente dispara un evento de cambio al oyente de cambio del modelo del deslizador inferior. El efecto es que cuando cambia el valor del deslizador superior, se actualizan los valores del deslizador y del campo de texto inferior. No es necesario notificar al deslizador superior los cambios en el deslizador inferior, ya que el modelo del deslizador inferior reenvia las peticiones de selección de datos al modelo del deslizador superior.

Oyentes de Window

Un oyente de window en el frame hace que se salga de la aplicación cuando se cierre la ventana.

Casi todos los oyentes del programa **Converter** están implementados en clases internas anónimas --clases sin nombre definidas dentro de otras clases. Aunque las clases internas podrían parecer difíciles de leer, al principio, realmente hacen el código mucho más fácil de comprender, una vez que las has utilizado. Manteniendo una implementación de un menejador de eventos cerca de donde se registra el manejador de eventos, las clases internas te ayudan y ayudan a los que te siguen a encontrar fácilmente la implementación completa del manejador de eventos. Puedes ver Usar Adaptadores y Clases Internas para Manejar Eventos para más información.

Reglas Generales del Uso de Componentes

Esta página tiene información general sobre cómo escribir un programa que contenga componentes Swing. Dos cosas importantes a tener en cuenta son evitar el uso de componentes de peso pesado que no sean Swing y poner nuestros componentes Swing dentro de un contenedor de alto nivel.

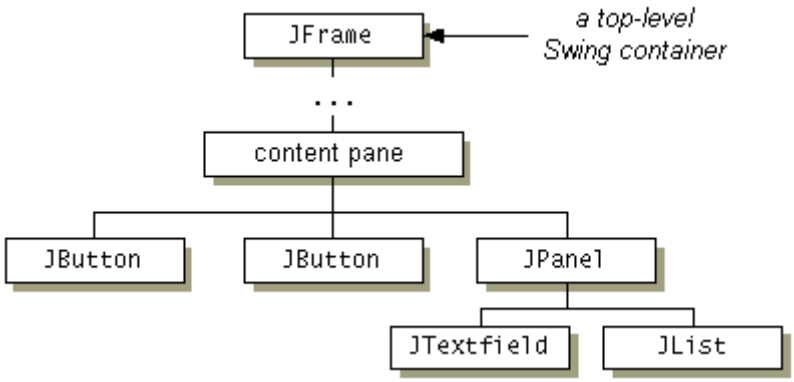
El paquete Swing define dos tipos de componentes.

- Contenedores de alto nivel (**JFrame**, **JApplet**, **JWindow**, **JDialog**)
- Componentes de peso ligero (**Jcualquier-cosa**, como **JButton**, **JPanel**, y **JMenu**)

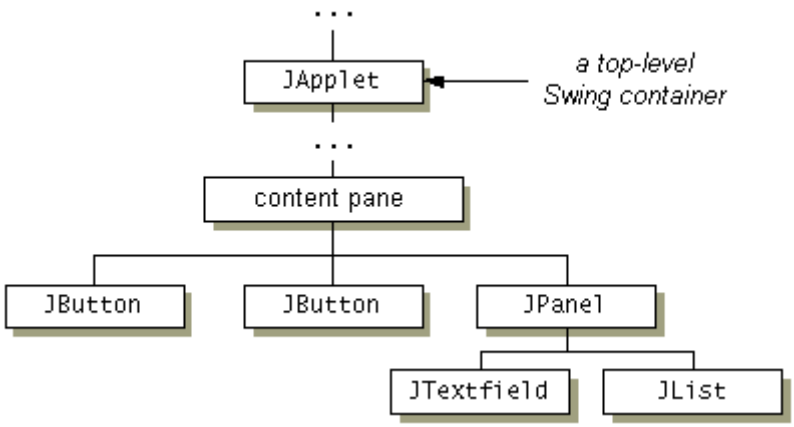
Los **contenedores de alto nivel** proporcionan el marco de trabajo en el que existen los componentes de peso ligero. Específicamente, un contenedor de alto nivel Swing proporciona un área en el que los componentes de peso ligero Swing pueden dibujarse a sí mismos. Los contenedores de alto nivel Swing también proporcionan otras características Swing como el espacio para una barra de menú, manejo avanzado de eventos y dibujo, y soporte de accesibilidad.

En general, todo componente Swing debería tener un contenedor Swing de alto nivel por encima en el árbol de contenidos. Por ejemplo, todo applet que contenga componentes Swing debería estar implementado como una subclase de **JApplet** (que es una subclase de **Applet**). De forma similar, toda ventana principal que contenga componentes Swing debería ser implementada con un **JFrame**.

Aquí tenemos una imagen del árbol de contentido de un GUI de un típico programa Swing que implementa una ventana que contiene dos botones, un campo de texto y una lista.



Aquí tenemos otra imagen del mismo GUI excepto en que ahora es un Applet ejecutándose en un navegador.



El **panel de contenido** de las figuras anteriores es el **Contenedor** normal que hay debajo de cada contenedor Swing. Un **contenedor de alto nivel Swing** es una subclase Swing de un componente AWT de peso pesado. Los contenedores de alto nivel Swing le añaden las necesidades Swing -- incluyendo un panel de contenido -- a los componentes pesados del AGM.

Aquí está el código que construye el GUI mostrado en las figuras anteriores.

```
//Configura el JPanel,que contiene el campo de texto y la lista.
JPanel panel = new JPanel();
panel.setLayout(new SomeLayoutManager());
panel.add(textField);
panel.add(list);

//topLevel es un ejemplar de JApplet o JFrame
Container contentPane = topLevel.getContentPane();
contentPane.setLayout(new AnotherLayoutManager());
contentPane.add(button1);
contentPane.add(button2);
contentPane.add(panel);
```

Nota: No se pueden añadir directamente los componentes a un componente de alto nivel.

```
topLevel.add(something);  //NO SE PUEDE HACER ESTO!!!
```

En general, deberíamos evitar el uso de componentes pesados en GUIs Swing (excepto los contenedores de alto nivel Swing que contienen el GUI, por supuesto). El problema más obsevable cuando se mezclan componentes pesados y ligeros es que cuando se solapan dentro de un contenedor, el componente de peso pesado siempre se dibuja encima del componente de peso ligero. Puedes ver Mezclar Componentes Pesados y Ligeros en la "Swing Connection" para más información sobre la mezca de los dos tipos de componentes.

La Clase JComponent

La mayoría de los componentes Swing están implementados como subclases de la clase **JComponent**, que desciende de la clase **Container**. De **JComponent**, los componentes Swing heredan las siguientes funcionalidades.

Bordes.
Usando el método **setBorder**, podemos especificar el borde que muestra un componente alrededor de sus lados. Podemos especificar que un componente tenga un espacio extra alrededor de su lados usando un ejemplar de **EmptyBorder**. Puedes ver la especificación de **BorderFactory** y Entender los Bordes (Un artículo de "The Swing Connection").

Doble buffer.
El doble buffer puede mejorar la apariencia de un componente que cambie frecuentemente. Ahora no tenemos que escribir nosotros el código del doble buffer -- Swing nos los proporciona. Por defecto, los componentes Swing usan el doble Buffer. Llamando al método **setDoubleBuffered(false)** sobre un componente se desactiva el doble buffer.

Tool tips.
Especificando un string con el método **setToolTipText**, podemos proporcionarle ayuda al usuario de un componente. Cuando el cursor se para sobre el componente, el String especificado se muestra en una pequeña ventana que aparece cerca del componente. Puedes ver Cómo usar Tool Tips para más información.

Navegación con Teclado.
Usando el método **registerKeyboardAction**, podemos permitir que el usuario use el teclado en vez del ratón para moverse por el GUI.

Nota: Algunas clases proporcionan métodos de conveniencia para acciones de teclado. Por ejemplo, **AbstractButton** proporciona **setMnemonic**, que permite especificar el caracter que en combinación con la tecla modificadora del Aspecto y Comportamiento, hace que se realice la acción del botón. Puedes ver en Cómo usar Buttons un ejemplo de uso de mnemónicos en botones.

La combinación del caracter y las teclas modificadoras que el usuario debe pulsar para arrancar la acción están representadas por un objeto **KeyStroke**. El evento action resultante es manejado por un objeto **ActionListener**. Toda acción de teclado funciona exactamente bajo una de estas dos condiciones: o el componente actual tiene el foco o cualquier otro componente de su ventana tiene el foco.

Propiedades.
Con el método **putProperty**, podemos asociar una o más propiedades (parejas nombre/objeto) con cualquier **JComponent**. Por ejemplo, un controlador de distribución podría usar propiedades para asociar restricciones con cada objeto **JComponent** que controle. Podemos poner y obtener las propiedades usando los métodos **putClientProperty** y **getClientProperty**.

Aspecto y Comportamiento Ampliamente Conectable.
Detrás de la excena, todo objeto **JComponent** tiene su correspondiente objeto **ComponentUI** que realiza todo el dibujado, el manejo de eventos, la determinación de tamaño, etc. El objeto **ComponentUI** realmente usado depende del Aspecto y Comportamiento, que se puede seleccionar usando el método **UIManager.setLookAndFeel**.

Soporte de Distribución.
Con métodos como **setPreferredSize**, **setMinimumSize**, **setMaximumSize**, **setAlignmentX**, y **setAlignmentY**, podemos especificar restricciones de distribución sin tener que reescribir nuestro propio componente.

Soporte de accesibilidad.
[PENDIENTE]

Soporte de Localización.
[PENDIENTE]

Contenedores de Alto Nivel

Antes de intentar usar un contenedor de alto nivel, deberías leer y entender Los Componentes Swing y el Arbol de Contenidos. Como dice esa sección, los programas que usan componentes Swing ponen los componentes en árboles de contenidos, y cada árbol de contenidos tiene un contenedor de alto nivel en su raíz.

En general, cada aplicación tiene al menos un árbol de contenidos encabezado por un objeto frame (**JFrame**). Cada applet debe tener un árbol de contenido encabezado por un objeto **JApplet**. Cada ventana adicional de una aplicación o un applet tiene su propio árbol de contenido encabezado por un rame o dialógo (**JDialog**/**JOptionPane**).

Nota: No cubrimos otro contenedor de alto nivel, JWindow, porque no se usa generalmente. Es simplementen la versión Swing de la clase Window AWT, que proporciona una ventana sin controles ni título que siempre está encima de cualquier otra ventana.

El panel de contenidos que hay en cada contenedor de alto nivel está proporcionado por un contenedor reclusivo llamado el panel raíz. Generalmente no necesitas saber nada sobre el panel raíz para usar componentes Swing.

¿Cómo Crear Frames?

La mayoría de las aplicaciones Swing presentan su GUI principal dentro de un **JFrame** -- un contenedor Swing de alto nivel que proporciona ventanas para applets y aplicaciones. Un frame tiene decoraciones como un borde, un título, y botones para cerrar y minimizar la ventana. Un programa típico símplemente crea un frame, añade componentes al panel de contenido, y quizás añade una barra de menú. Sin embargo, a través de su panel raíz, **JFrame** proporciona soporte para una mayor personalización.

Para crear una ventana que sea dependiente de otra ventana -- que desaparezca cuando la otra ventana se minimiza, por ejemplo -- se utiliza un diálogo en vez de un frame. Para crear una ventana que aparece dentro de otra ventana se utiliza un frame interno.

Aquí podemos ver dos imágenes del mismo programa, FrameDemo.java, ejecutándose en distintas plataformas. El programa trae un frame que contiene algo interesante que ver.

Solaris	Windows
	[PENDIENTE]

Nota:La decoración de un frame es dependiente del sistema. No se puede cambiar la decoración de un frame.

Abajo podemos ver el código de **FrameDemo.java** que crea el frame del ejemplo anterior.

```
public static void main(String s[]) {
    JFrame frame = new JFrame("A Basic Frame");

    WindowListener l = new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
                    System.exit(0);
        }
    };
    frame.addWindowListener(l);

    JLabel aLabel = new JLabel("Something to look at",
                                       new ImageIcon("images/beach.gif"),
                                       JLabel.CENTER);
    aLabel.setVerticalTextPosition(JLabel.TOP);
    aLabel.setHorizontalTextPosition(JLabel.CENTER);
    frame.getContentPane().add(aLabel, BorderLayout.CENTER);

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

El código crea un frame con el título **A Basic Frame** y añade un oyente de window para salir del programa cuando el usuario cierre el frame.

Las líneas en itálica del código crean la etiqueta que muestra el texto y la imagen del frame. Este es esencialmente el GUI del programa. Si queremos utilizar este programa como marco de trabajo para nuestros propios programas, sólo tenemos que reemplazar el código en itálica para crear los componentes que querramos.

El código en negrita añade la etiqueta al panel de contenido del frame. Puedes ir a Añadir Componentes a un Frame para más detalles y ejemplos.

Para que un frame aparezca en la pantalla, un programa debe llamar a **setSize** o a **pack**, y luego llamar a **setVisible(true)** o su equivalente, **show**. Este programa empaqueta el frame y utiliza **setVisible**. Observa que si cualquier parte del GUI ya es visible, deberíamos invocar a **setVisible** desde el thread de lanzado de eventos. Puedes ver la página Threads y Swing.

Este código es típico de muchos programas y es el marco de trabajo que hemos utilizado para crear la mayoría de los ejemplos de esta lección (incluyendo **GlassPaneDemo.java** y **BorderDemo.java**). Algunos ejemplos como **TextFieldDemo.java** y **TableDemo.java**, subclasifican **JFrame** y ejemplarizan la subclase frame en vez **JFrame**. En esos programas, el GUI se crea en el constructor de la subclase. Podríamos hacer esto en nuestros programas si necesitaramos subclasificar **JFrame** por alguna razón.

JFrame es una subclase de **java.awt.Frame** a la que añade soporte para la interposición de entradas y comportamiento de pintado contra el frame hijo, situando hijos en diferentes "capas" (layers) y para barras de menús Swing. Generalmente hablando, deberíamos utilizar **JFrame** en vez de **Frame**, por estas razones.

- Para aprovechar las nuevas características proporcionadas por su panel raíz como el panel transparente y el panel de capas.
- **JFrame** permite personalizar el comportamiento de la ventana, llamando al método **setDefaultCloseOperation** en vez de escribir un oyente de window.
- **JFrame** soporta el método **revalidate**.
- Los menús Swing funcionan mejor en un **JFrame** debido a sus métodos **setJMenuBar**.
- Deberíamos utilizar **JFrame** en un applet si éste utiliza componentes Swing. También, recomendamos la utilización de **JFrame** en una aplicación que utilice componentes Swing, aunque no es necesario.

Añadir Componentes a un Frame

Como se ha visto en **FrameDemo.java**, para añadir componentes a un **JFrame**, se añaden a su panel de contenido. Las dos técnicas más comunes para añadir componetes al panel de contenido de un frame son.

- Crear un contenedor como un **JPanel**, **JScrollPane**, o un **JTabbedPane**, y añadirle componentes, luego utilizar **JFrame.setContentPane** para convertirlo en el panel de contenido del frame.

  TableDemo.java utiliza esta técnica. El código crea un panel desplazable para utilizarlo como panel de contenido del frame. Hay una tabla en el panel desplazable.

```
public class TableDemo extends JFrame {

    public TableDemo() {
        super("TableDemo");

        MyTableModel myModel = new MyTableModel();
        JTable table = new JTable(myModel);
        table.setPreferredScrollableViewportSize(new Dimension(500, 70));

        //Create the scroll pane and add the table to it.
        JScrollPane scrollPane = new JScrollPane(table);

        //Add the scroll pane to this window.
        setContentPane(scrollPane);

    . . .
```

- Utilizar **JFrame.getContentPane** para obtener el panel de contenido del frame. Añadir componentes al objeto devuelto.**LayeredPaneDemo.java** utiliza esta técnica mostrada aquí.

```
...//create the components...
//get the content pane, add components to it:
Container contentPane = getContentPane();
// use a layout manager that respects preferred sizes
contentPane.setLayout(new BoxLayout(contentPane, BoxLayout.Y_AXIS));
contentPane.add(Box.createRigidArea(new Dimension(0, 10)));
contentPane.add(controls);
contentPane.add(Box.createRigidArea(new Dimension(0, 10)));
contentPane.add(emptyArea);
```

  El controlador de disposición por defecto para el panel de contenido de un frame es **BorderLayout**. Cómo se ha visto en los ejemplos anteriores, se puede invocar al método **setLayout** sobre el panel de contenidos para cambiar su controlador de disposición.

EL API JFrame

Las siguientes tablas listan los métodos y constuctores más utilizados de **JFrame**. Existen otros métodos que podríamos llamar y que están definidos en las clases **Frame** y **Window** y que incluyen **pack**, **setSize**, **show**, **hide**, **setVisible**, **setTitle**, y **getTitle**.

La mayor parte de la opración de un frame está manejada por otros objetos. Por ejemplo, el interior de un frame está manejado por su panel raíz, y el panel de contenido contiene el GUI del frame.

El API para utilizar Frames se divide en dos categorías.

- Crear y Configurar un Frame
- Seleccionar y Obtener los objetos auxiliares de un Frame

Crear y configurar un Frame

Método	Propósito
JFrame() **JFrame(String)**	Crea un frame. El argumento **String** proporciona el título del frame.
void setDefaultCloseOperation(int) **int getDefaultCloseOperation()**	Selecciona u obtiene la operación que ocurre cuando el usuario pulsa el botón de cerrar la ventana. Las posibles elecciones son. • **DO_NOTHING_ON_CLOSE** • **HIDE_ON_CLOSE** (por defecto) • **DISPOSE_ON_CLOSE** Estas constantes están definidas en el interface **WindowConstants**.

Seleccionar y Obtener los objetos auxiliares de un Frame

Método	Propósito
void setContentPane(Container) **Container getContentPane()**	Selecciona u obtiene el panel de contenido del frame. También se puede hacer a través del panel raíz del frame.
JRootPane createRootPane()	Crea, selecciona u obtiene el panel raíz del frame. El panel raíz maneja el interior de frame, incluyendo el panel de contendido, el panel transparente,

void setRootPane(JRootPane) **JRootPane getRootPane()**	etc.
void setJMenuBar(JMenuBar) **JMenuBar getJMenuBar()**	Selecciona u obtiene la barra de menú del frame. También se puede hacer a través del panel raíz del frame.
void setGlassPane(Component) **Component getGlassPane()**	Selecciona u obtiene el panel transparente del frame. También se puede hacer a través del panel raíz del frame.
void setLayeredPane(JLayeredPane) **JLayeredPane getLayeredPane()**	Selecciona u obtiene el panel de capas del frame. También se puede hacer a través del panel raíz del frame.

¿Cómo crear Diálogos?

Muchas clases Swing soportan diálogos -- ventanas que son más limitadas que los frames. Para crear un diálogo, simple y estándard se utiliza **JOptionPane**. Para crear diálogos personalizados, se utiliza directamente la clase **JDialog**. La clase **ProgressMonitor** puede poner un diálogo que muestra el progreso de una operación. Otras dos clases, **JColorChooser** y **JFileChooser**, también suministran diálogos estándard. Para mostrar un diálogo de impresión se utiliza el método **getPrintJob** de la clase **Toolkit**.

El código para diálogos simples puede ser mínimo. Por ejemplo, aquí tenemos un diálogo informativo.

Aquí podemos ver el código que lo crea y lo muestra.

```
JOptionPane.showMessageDialog(frame, "Eggs aren't supposed to be green.");
```

El resto de esta página cubre los siguientes tópicos.

Introducción a los diálogos

Todo diálogo depende de un frame. Cuando el frame se destruye, también se destruyen sus diálogos. Cuando el frame es minimizado, sus diálogos dependientes también desaparecen de la pantalla. Cuando el frame es maximizado, sus diálogos dependientes vuelven a la pantalla. El AWT proporciona automáticamente este comportamiento.

Un diálogo puede ser modal. Cuando un diálogo modal es visible, bloquea las entradas del usuario en todas las otras ventanas del programa. Todos los diálogos que proporciona **JOptionPane** son modales. Para crear un diálogo no modal, debemos utilizar directamente la clase **JDialog**.

La clase **JDialog** es una subclase de la clase **java.awt.Dialog** del AWT. Le añade a **Dialog** un root pane y soporte para una operación de cerrado por defecto. Estas son las mismas características que tiene **JFrame**, y utilizar directamente **JDialog** es muy similar a hacerlo con **JFrame**. Puedes ver Cómo crear Frames para más información sobre cómo añadir componentes a una ventana y cómo implementar algún oyente de window.

Incluso si utilizamos **JOptionPane** para implementar un diálogo, estamos utilizando **JDialog** detrás de la escena. La razón para esto es que **JOptionPane** es simplemente un contenedor que puede crear automáticamente un **JDialog** y se añade a sí mismo al panel de contenido de **JDialog**.

Características de JOptionPane

Utilizando **JOptionPane**, se pueden crear muchos diálogos. Aquí podemos ver unos ejemplos, todos producidos por DialogDemo.

Como podríamos observar en los ejemplos anteriores, **JOptionPane** proporciona soporte para mostrar diálogos estándards, proporcionando iconos, específicando el título y el texto del diálogo, y personalizando el texto del botón. Otras características permiten personalizar los componentes del diálogo a mostrar y especificar si el diálogo debería aparecer en la pantalla. Incluso se puede especificar qué panel de opciones se pone a sí mismo dentro de un frame interno (**JInternalFrame**) en lugar de un **JDialog**.

Cuando se crea un **JOptionPane**, el código específico del aspecto y comportamiento añade componentes al **JOptionPane** y determina la distribución de dichos componentes. La siguiente figura muestra cómo los aspectos y comportamientos más comunes distribuyen un **JOptionPane**.

icono (si existe)	mensaje
botones	

Para la mayoría de los diálogos modales sencillos, se crea y se muestra el diálogo utilizando uno de los métodos **showXxxDialog** de **JOptionsPane**. Para ejemplos de utilización de los siguientes métodos, puedes ver **DialogDemo.java**. Si nuestro diálogo debería ser un frame interno, se añade **Internal** después de **show** -- por ejemplo, **showInternalMessageDialog**.

showMessageDialog
: Muestra un diálogo modal con un botón, etiquetado "OK". Se puede especificar fácilmente el mensaje, el icono y el título que mostrará el diálogo.

showConfirmDialog
: Muestra un diálogo modal con dos botones, etiquetados "Yes" y "No". Estas etiquetas no son siempre terriblemente descriptivas con las actiones específicas del programa que causan.

showInputDialog
: Muestra un diálogo modal que obtiene una cadena del usuario. Un diálogo de entrada muestra un campo de texto para que el usuario teclee en él, o un ComboBox no editable, desde el que el usuario puede elegir una de entre varias cadenas.

showOptionDialog
: Muestra un diálogo modal con los botones, los iconos, el mensaje y el título especificado, etc. Con este método, podemos cambiar el texto que aparece en los botones de los diálogos estándard. También podemos realizar cualquier tipo de personalización.

El soporte de iconos de **JOptionPane** permite especificar qué icono mostrará el diálogo. Podemos utilizar un icono personalizado, no utilizar ninguno, o utilizar uno de los cuatro iconos estándard de **JOptionPane** (question, information, warning, y error). Cada aspecto y comportamiento tiene sus propias versiones de los cuatro iconos estándard. La siguiente imagen muestra los iconos utilizados en el Aspecto y Comportamiento Java (popularmente conocido como Metal).

question	information	warning	error

Por defecto, un diálogo creado con **showMessageDialog** muestra el icono de información, un diálogo creado con **showConfirmDialog** o **showInputDialog** muestra un icono question. Para especificar qué un diálogo estándard no tenga icono o tenga otro icono estándard, se añade un parámetro que especifica el tipo de mensaje. El valor del tipo de mensaje puede ser una de las siguientes constantes: **PLAIN_MESSAGE** (sin icono), **QUESTION_MESSAGE**, **INFORMATION_MESSAGE**, **WARNING_MESSAGE**, o **ERROR_MESSAGE**. Si especificamos un objeto **Icon** distinto de null, el diálogo muestra ese icono, no importa el tipo de mensaje que sea. Aquí tenemos un ejemplo que crea un díalogo sencillo que muestra un mensaje de error:

```
JOptionPane.showMessageDialog(frame,
                              "Eggs aren't supposed to be green.",
                              "Inane error",
                              JOptionPane.ERROR_MESSAGE);
```

Normalmente, el área del mensaje de un panel de opciones tiene una sóla línea de texto, como "Eggs aren't supposed to be green." Podemos dividir el mensaje en varias líneas poniendo caracteres de nueva línea (**\n**) dentro del string del mensaje. Por ejemplo.

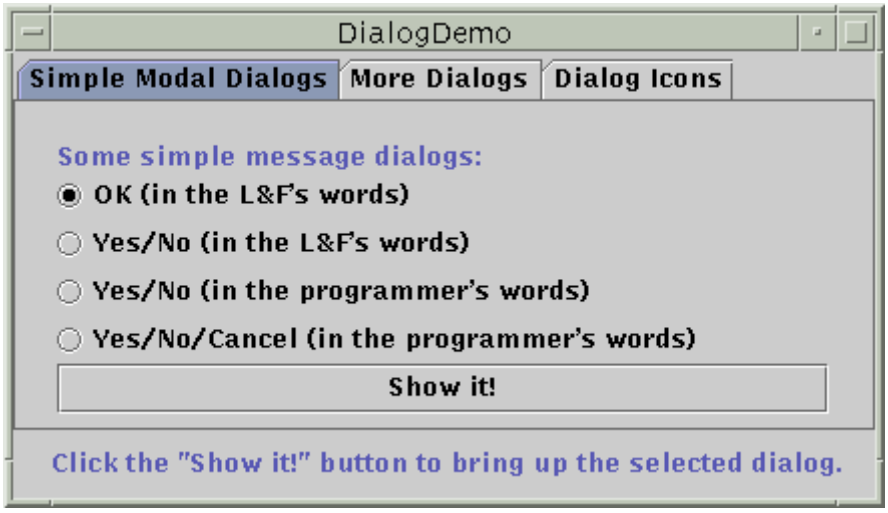
```
"Complete the sentence:\n"
+ "\"Green eggs and...\""
```

Podemos especificar el texto mostrado por los botones del panel. Un ejemplo de esto está en Personalizar el texto de los botones en un diálogo estándard. Cuando el usuario pulsa cualquier botón, el diálogo desaparece automáticamente. si no queremos que el diálogo desaparezca automáticamente -- por ejemplo, si queremos asegurarnos de que la entrada del usuario es válida antes de cerrar el diálogo -- necesitamos seguir los pasos descritos en Detener la salida Automática de un Diálogo.

Cuando se llama a uno de los métodos **showXxxDialog** de **jOptionPane**, el primer argumento especifica un componente. Este componente determina la posición en la pantalla del díalogo y del frame del que éste depende. Si especificamos **null** para el componente, el diálogo es independiente de cualquier frame visible, y aparece en el medio de la pantalla. La siguiente figura muestra lo que sucede cuando se indica un **JFrame** como primer argumento.

El Ejemplo DialogDemo

Aquí tenemos una imagen de una aplicación que muestra diálogos.



Intenta esto:

1. Compila y ejecuta la aplicación, El fichero fuente es **DialogDemo.java**.
2. Pulsa el botón "Show it!".

   Aparecerá un diálogo modal. Hasta que lo cierres, la aplicación no responderá, aunque se redibujará a sí misma si es necesario. Puedes salir del diálogo pulsando un botón o explícitamente utilizando el icono de cerrado de la ventana.

3. Minimiza la ventana DialogDemo mientras se muestra el diálogo.

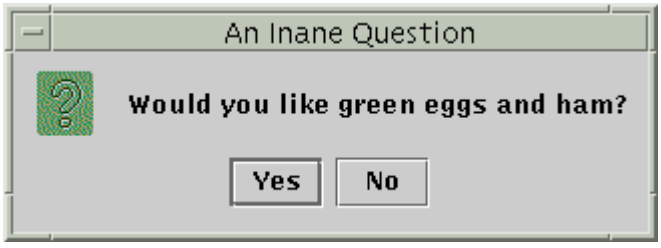
   El diálogo desaparecerá de la pantalla hasta que maximices la ventana de DialogDemo.

4. En el panel "More Dialogs", pulsa el botón de rádio iferior y luego el botón "Show it!". Aparecerá un diálogo no modal. Observa que la ventana de DialogDemo permanece totalmente funcional mientras está activo el diálogo no modal.

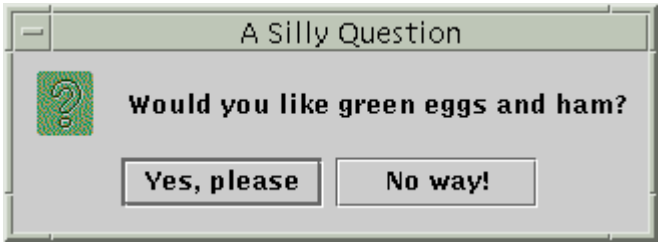
Personalizar el texto de los botones en un diálogo estándard

Cuando se utiliza **JOptionPane** para crear un diálogo estándard, podemos elegir si utilizar el texto estándard del botón (que podría variar dependiendo del aspecto y comportamiento) o especificar un texto diferente..

El siguiente código, tomado de **DialogDemo.java**, crea dos diálogos Yes/No. El primer diálogo utiliza las palabras del aspecto y comportamiento para los dos botones. El segundo diálogo personaliza las palabras. Con la excepción del cambio de palabras, los diálogos son idénticos. Para personalizar las palabras, el código que crea el segundo diálogo utiliza **showOptionDialog**, en vez de **showConfirmDialog**.




```
./create the yes/no dialog:
int n = JOptionPane.showConfirmDialog(
                frame, "Would you like green eggs and ham?",
                "An Inane Question",
                JOptionPane.YES_NO_OPTION);
if (n == JOptionPane.YES_OPTION) {
    setLabel("Ewww!");
} else if (n == JOptionPane.NO_OPTION) {
    setLabel("Me neither!");
} else {
    setLabel("Come on -- tell me!");
}
```




```
./create the yes/no (but in other words) dialog:
String string1 = "Yes, please";
String string2 = "No way!";
Object[] options = {string1, string2};
int n = JOptionPane.showOptionDialog(frame,
                               "Would you like green eggs and ham?",
                               "A Silly Question",
                               JOptionPane.YES_NO_OPTION,
                               JOptionPane.QUESTION_MESSAGE,
                               null,     //don't use a custom Icon
                               options,  //the titles of buttons
                               string1); //the title of the default button
if (n == JOptionPane.YES_OPTION) {
    setLabel("You're kidding!");
} else if (n == JOptionPane.NO_OPTION) {
    setLabel("I don't like them, either.");
} else {
    setLabel("Come on -- 'fess up!");
}
```

Obtener entrada del usuario desde un diálogo

Cómo se vió en el ejemplo anterior, los métodos **showXxxDialog** de **JOptionPane** devuelven un valor que indica la elección del usuario. Si, por otro lado, estamos diseñando un diálogo personalizado, necesitamos diseñar el API de nuestro diálogo para que pueda preguntar al usuario sobre la elección del usuario.

Para los diálogos estándard **JOptionPane**, los métodos **showXxxDialog** devuelven un entero. Los valores por defecto para este entero son **YES_OPTION**, **NO_OPTION**, **CANCEL_OPTION**, **OK_OPTION**, y **CLOSED_OPTION**. Excepto para **CLOSED_OPTION**, cada opción correponde con el botón pulsado por el usuario. Cuando se devuelve **CLOSED_OPTION**, indica que el usuario ha cerrado la ventana del diálogo explícitamente, en vez de elegir un botón.

Incluso si cambiamos los textos de los botones del diálogo estándard (como en el ejemplo anterior), el valor devuelto sigue siendo uno de los enteros predefinidos. Por ejemplo, un diálogo **YES_NO_OPTION** siempre devuelve uno e los siguientes valores: **YES_OPTION**, **NO_OPTION**, o **CLOSED_OPTION**.

Detener la Despedida Automática de un Diálogo

Por defecto, cuando el usuario crea un botón del **JOptionPane** o cierra su ventana explícitamente, el diálogo desaparece. Pero ¿que pasa si queremos comprobar la respuesta del usuario antes de cerrar la ventana? En este caso, debemos implementar nuestro propio oyente de change para que cuando el usuario pulse un botón, el diálogo no desparezca automáticamente.

DialogDemo contiene dos diálogos que implementan un oyente de change. Uno de esos diálogos es un diálogo modal, implementado en **CustomDialog.java**, que utiliza **JOptionPane** para obtener los iconos estándard y para obtener asistencia en la distribución. El otro diálogo, cuyo código está abajo, utiliza un **JOptionPane** estándard Yes/No. Aunque este diálogo es poco más que inútil, su código es lo suficientemente sencillo como para poder utilizarlo como plantilla para diálogos más complejos.

Junto con la configuración del oyente de change, el código siguiente también llama al método **setDefaultCloseOperation** de **JDialog** e implementa un oyente de window que maneja apropiadamente el intento de cierre de la ventana. Si no nos importa ser notificados cuando el usuario cierre la ventana explícitamente, podemos ignorar el código que no está en negrita.

```
final JOptionPane optionPane = new JOptionPane(
                               "The only way to close this dialog is by\n"
                               + "pressing one of the following buttons.\n"
                               + "Do you understand?",
                               JOptionPane.QUESTION_MESSAGE,
                               JOptionPane.YES_NO_OPTION);

final JDialog dialog = new JDialog(frame,
                                                "Click a button",
                                                true);
dialog.setContentPane(optionPane);
dialog.setDefaultCloseOperation(
    JDialog.DO_NOTHING_ON_CLOSE);
dialog.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent we) {
             setLabel("Thwarted user attempt to close window.");
    }
});
optionPane.addPropertyChangeListener(
    new PropertyChangeListener() {
             public void propertyChange(PropertyChangeEvent e) {
                 String prop = e.getPropertyName();

                 if (dialog.isVisible()
                  && (e.getSource() == optionPane)
                  && (prop.equals(JOptionPane.VALUE_PROPERTY) ||
                                  prop.equals(JOptionPane.INPUT_VALUE_PROPERTY))) {
                             //If you were going to check something
                             //before closing the window, you'd do
                             //it here.
                             dialog.setVisible(false);
                 }
```

```
                }
    });
dialog.pack();
dialog.show();

int value = ((Integer)optionPane.getValue()).intValue();
if (value == JOptionPane.YES_OPTION) {
    setLabel("Good.");
} else if (value == JOptionPane.NO_OPTION) {
    setLabel("Try using the window decorations "
                    + "to close the non-auto-closing dialog. "
                    + "You can't!");
}
```

El API Dialog

Las siguiente tablas listan los métodos y constructores más utilizados de **JOptionPane** y **JDialog**. Otros métodos que podríamos utilizar están definidos por las clases **JComponent** y **Component**.

Mostrar diálogos modales estándard (utiizando métodos de la clase JOptionPane)

Método	Propósito
int showMessageDialog(Component, Object) **int showMessageDialog(Component, Object, String, int)** **int showMessageDialog(Component, Object, String, int, Icon)**	Muestra un diálogo modal con un botón.
int showOptionDialog(Component, Object, String, int, int, Icon, Object[], Object)	Muestra un diálogo.
int showConfirmDialog(Component, Object) **int showConfirmDialog(Component, Object, String, int)** **int showConfirmDialog(Component, Object, String, int, int)** **int showConfirmDialog(Component, Object, String, int, int, Icon)**	Muestra un diálogo modal que [PENDIENTE: elaborar].
String showInputDialog(Object) **String showInputDialog(Component, Object)** **String showInputDialog(Component, Object, String, int)** **String showInputDialog(Component, Object, String, int, Icon, Object[], Object)**	Muestra un diálogo de entrada.
int showInternalMessageDialog(...) **int showInternalOptionDialog(...)** **int showInternalConfirmDialog(...)** **String showInternalInputDialog(...)**	Implementa un diálogo estándard como un frame interno.

Métodos para utilizar JOptionPane directamente

Método	Propósito
JOptionPane() **JOptionPane(Object)** **JOptionPane(Object, int)** **JOptionPane(Object, int, int)** **JOptionPane(Object, int, int, Icon)**	Crea un ejemplar de **JOptionPane**.

JOptionPane(Object, int, int, Icon, Object[]) **JOptionPane(Object, int, int, Icon, Object[], Object)**	
Frame getFrameForComponent(Component) **JDesktopPane getDesktopPaneForComponent(Component)**	Manejan métodos de clase de **JOptionPane** que encuentran el frame o desktop pane, respectivamente, en el que se encuentra el componente especificado.

Otros Constructores y Métodos de JOptionPane

Métodos	Propósito
JOptionPane() **JOptionPane(Object)** **JOptionPane(Object, int)** **JOptionPane(Object, int, int)** **JOptionPane(Object, int, int, Icon)** **JOptionPane(Object, int, int, Icon, Object[])** **JOptionPane(Object, int, int, Icon, Object[], Object)**	Crea un ejemplar de **JOptionPane**.

Constructores y Métodos más utilizados de JDialog

Método	Propósito
JDialog() **JDialog(Frame)** **JDialog(Frame, boolean)** **JDialog(Frame, String)** **JDialog(Frame, String, boolean)**	Crea un ejemplar de **JDialog**. El argumento **Frame**, si existe, es el frame (normalmente un objeto **JFrame**) del que depende el diálogo. Se hace el argumento booleano **true** para especificar un diálogo modal, **false** o ausente, para especificar un diálogo no modal. También se puede especificar el título de un diálogo utilizando un argumento string.
Container getContentPane() **setContentPane(Container)**	Obtiene y selecciona el panel de contenido que normalmente es el contenedor de todos los componentes del diálogo.
int getDefaultCloseOperation() **setDefaultCloseOperation(int)**	Obtiene y selecciona lo que sucece cuando el usuario intenta cerrar el diálogo. Valores posibles: **DISPOSE_ON_CLOSE**, **DO_NOTHING_ON_CLOSE**, **HIDE_ON_CLOSE** (por defecto).
void setLocationRelativeTo(Component)	Centra el diálogo sobre el componente especificado.

¿Cómo crear Applets?

Esta página cubre el uso de **JApplet**. Por ahora, sólo proporcionamos una lista de ejemplos de applets.

- Ejecutar un Applet Swing muestra cómo ejecutar applet Swing, ecpecíficamente los applets **HelloSwingApplet.java** y **AppletDemo.java**. El programa AppletDemo es una versión applet del programa button.
- **IconDemoApplet.java**, mostrado abajo, se muestra en Cómo usar Iconos.

IconDemoApplet.html contiene la etiqueta **<APPLET>** para ejecutar este applet.

Contenedores Intermedios

Esta sección describe los componentes Swing que sólo existen para contener otros componentes. Técnicamente, las barras de menús caen en ésta categoría, pero se describen en otro sitio, en la página Cómo usar Menús. Para usar contenedores intermedios, deberíamos entender los conceptos presentados en Los Componentes Swing y el Árbol de Contenidos.

Paneles son los contenedores de propósito general más frecuentemente utilizados. Implementados con la clase **JPanel**, los paneles no añaden casi ninguna funcionalidad más allá de las que tienen los objetos **JComponent**. Normalmente se usan para agrupar componentes, porque los componentes están relacionados o sólo porque agruparlos hace que la distribución sea más sencilla. Un panel puede usar cualquier controlador de distribución, y se les puede dotar de bordes fácilmente.

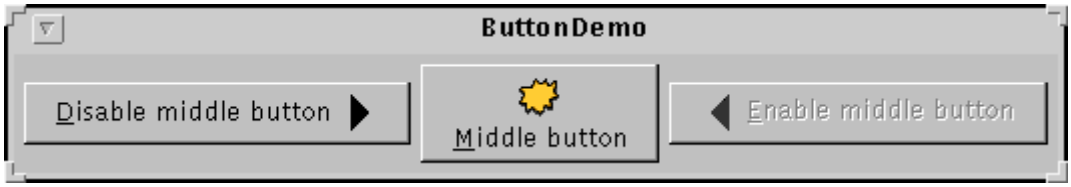
Otros cuatro contenedores Swing proporcionan más funcionalidad. Un scroll pane proporciona barras de desplazamiento alrededor de un sólo componente. Un split pane permite al usuario personalizar la cantidad relativa de espacio dedicada a cada uno de dos componentes. Un tabbed pane muestra sólo un componente a la vez, permitiendo fácilmente cambiar entre componentes. Un tool bar contiene un grupo de componentes (normalmente botones) en una fila o columna, y opcionalmente permite al usuario arrastrar la barra de herramientas a diferentes localizaciones.

El resto de los contenedores intermedios Swing son incluso más especializados. Internal frames se parecen a los frames y tienen mucho del mismo API pero al contrario que los frames deben aparecer dentro de otras ventanas. Root panes proporcionan soporte detrás-de-la-escena a los contenedores de alto nivel. Layered panes existen para soportar ordenación en el eje Z de componentes.

¿Cómo Usar Panel?

JPanel es un contenedor de propósito general para componentes de peso ligero. Como todos los contenedores, utiliza un **Controlador de Distribución** para posicionar y dimensionar sus componentes. Como todos los componentes Swing, **JPanel** permite añadirle bordes y determinar si utiliza el doble buffer para aumentar el rendimiento.

Esta imagen muestra una aplicación que utiliza un panel y su controlador de distribución por defecto, **FlowLayout**, para mostrar tres botones.



La clase principal de esta aplicación es **ButtonDemo**, que es una subclase de **JPanel**. Aquí puedes ver el código del constructor de **ButtonDemo** que le añade los tres botones al panel.

```
public ButtonDemo() {
    super();
    ...
    create the three buttons
    ...
    //Add Components to this container, using the default FlowLayout.
    add(b1);
    add(b2);
    add(b3);
}
```

Este código no selecciona explícitamente el controlador de distribución del panel, por eso utiliza el controlador por defecto. Este controlador de distribución, **FlowLayout**, sitúa los componentes en una fila con sus tamaños preferidos. Si queremos utilizar otro controlador de distribución, podemos especicar el controlador cuando creamos el objeto **JPanel** o utilizar el método **setLayout** posteriormente. El AWT proporciona una colección de útiles controladores de distribución, y Swing añade otro controlador de distribución de propósito general, **BoxLayout**.

El fragmento de código anterior utiliza un método **add** heredado de **java.awt.Container** que requiere un sólo argumento: el componente a añadir. Como otras subclases de **Container**, **JPanel** hereda otros métodos **add** que permiten especificar restricciones e información sobre posicionamiento cuando se añade un componente. Se debe elegir el método **add** para el controlador de disposición que se está utilizando.

Otros Contenedores

JPanel es sólo una de las varias clases de contenedores que se pueden utilizar. Existen algunos contenedores de propósito especial que podríamos utilizar en lugar de un **JPanel**.

Box	Automáticamente utiliza un **BoxLayout** para distribuir sus componentes. La ventaja de **Box** es que es de peso superligero, ya que desciende directamente de la clase **Container**. Su desventaja es que no es un verdadero componente Swing -- no hereda el API que soporta caracterísitcas como los bordes de la caja, ni la selección sencilla de los tamaños máximo, mínimo y preferido. Por esta razón, nuestros ejemplos utilizan **JPanel** con **BoxLayout**, en vez **Box**.
JLayeredPane	Proporciona una tercera dimensión, profundidad, para posicionar componentes. Los paneles con capas no tienen controladores de distribución pero pueden ser utilizados para colocar los componentes en capas en un **JPanel**s. Un tipo de layeredpane, **JDesktopPane**, está diseñado específicamente para manejar frames internos.
JScrollPane	Proporciona una vista desplazable de un componente grande.
JSplitPane	Muestra dos componentes cuyos tamaños relativos pueden ser modificados por el usuario.
JTabbedPane	Permite a varios componentes, normalmente objetos **JPanel**, compartir el mismo espacio.

Otro contenedor que podríamos utilizar es el panel de contenido por defecto de un applet, frame, internal frame, o dialog. El panel de contenido es un **Container** que, como regla, contiene todos los componentes no-menús de la ventana. Se puede encontrar el panel de contenido utilizando un método llamado **getContentPane**. De forma similar, se puede seleccionar el panel de contenido -- quizás para que sea un **JPanel** que hayamos creado -- utilizando **setContentPane**.

El API JPanel

El API de la propia clase **JPanel** es mínimo. Los métodos que más se utilizan de un objeto **JPanel** son aquellos que hereda de sus superclases **JComponent**, **Container** y **Component**.

Crear un JPanel

Método	Propósito
JPanel() **JPanel(boolean)** **JPanel(LayoutManager)** **JPanel(LayoutManager,**	Crea un panel. Cuando está presente, el parámetro **boolean** determina la estrategía de buffer del panel. Un valor **true** indica doble buffer. El parámetro **LayoutManager** proporciona el controlador de distribución para el nuevo panel. Si no se especifica, el panel utiliza **FlowLayout** para distribuir sus componentes.

boolean)	

Manejar Componentes en un Contenedor

Método	Propósito
void add(Component) void add(Component, int) void add(Component, Object) void add(Component, Object, int) void add(String, Component)	Añade el componente especificado al panel. Cuando existe, el parámetro **int** es la posición o índice del componente dentro del contenedor. El parámetro **Object** depende del controlador de distribución y normalmente proporciona información sobre el posicionamiento y restricciones de distribución cuando se añaden componentes. El parámetro **String** proporciona un nombre para el componente.
int getComponentCount()	Obtiene el número de componentes en este panel.
Component getComponent(int) Component getComponentAt(int, int) Component getComponentAt(Point) Component[] getComponents()	Obtiene el componente o componentes especificados. Se pueden obtener basándose en su índice, o en sus posiciones x,y.
void remove(Component) void remove(int) void removeAll()	Elimina el componente o componentes especificados.

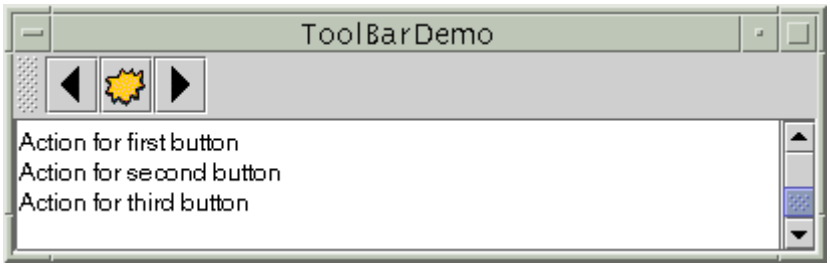
Seleccionar/Obtener el Controlador de Distribución

Método	Propósito
void setLayout(LayoutManager) LayoutManager getLayout()	Selecciona u obtiene el controlador de distribución para este panel. El controlador de distribución es el responsable de posicionar los componentes dentro del panel de acuerdo con alguna filosofía.

¿Cómo Usar ScrollPane?

Un **JScrollPane** proporciona una vista desplazable de un componente ligero. Cuando el estado de la pantalla real está limitado, se utiliza un ScrollPane para mostrar un componente que es grande o cuyo tamaño puede cambiar dinámicamente.

El código para crear un panel desplazable puede ser mínimo. Por ejemplo aquí tenemos una imagen de un programa que utiliza un panel desplazable para ver una salida de texto.



Y aquí está el código que crea el área de texto, crea el cliente del panel desplazable, y añade el panel desplazable a la ventana.

```
textArea = new JTextArea(5, 30);
JScrollPane scrollPane = new JScrollPane(textArea);
...
contentPane.setPreferredSize(new Dimension(400, 100));
...
contentPane.add(scrollPane, BorderLayout.CENTER);
```

El programa proporciona el área de texto como argumento al constructor del **JScrollPane**. Esto establece el área de texto como el cliente del panel desplazable. El panel desplazable maneja todo esto: crear las barras de desplazamiento cuando son necesarias, redibujar el cliente cuando el usuario se mueve sobre él, etc.

Observa que el código de ejemplo selecciona el tamaño preferido del contenedor del panel desplazable. Una alternativa sería seleccionar el tamaño preferido del propio panel desplazable. De cualquier modo, se está limitando el tamaño del panel desplazable. Esto es necesario porque el tamaño preferido de un panel desplazable es ser tan grande como pueda.

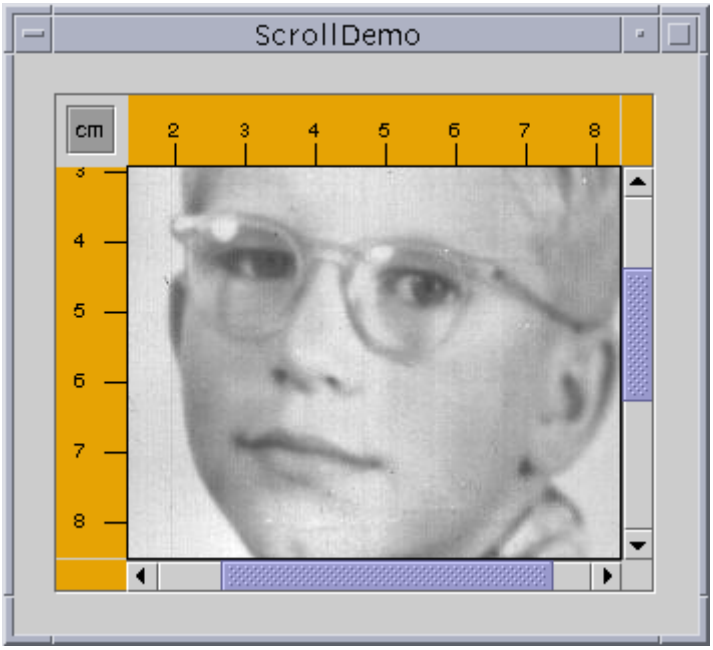
Por defecto, un panel desplazable intenta redimensionarse para que su cliente se muestre en su tamaño preferido. Muchos componetes tienen un sencillo tamaño preferido que es lo suficientemente grande como para dibujarse entero. Esto hace que el panel desplazable sea redundante. Otros componentes, como listas, tablas, componentes de texto, y árboles, reportan un tamaño preferido separado para desplazamiento, que normalmente es más pequeño que el tamaño preferido estándard. Por ejemplo, por defecto, el tamaño preferido de una lista para despalzarla es lo suficientemente grande para mostrar ocho filas. Si el tamaño preferido reportado por el componente, no es el que queremos, se selecciona el tamaño preferido del panel desplazable o de su contenedor.

Si todo lo que necesitas es proprocionar desplazamiento básico para un componente ligero, no leas más. Lo que demuestra esta ejemplo es suficiente para la mayoría de los programas.

Sin embargo, un ScrollPane es un objeo altamente personalizable. Se puede determinar bajo que circunstancias se mostrarán las barras de desplazamiento. También se puede decorar con una fila de cabecera, una columna de cabecera y esquinas. Finalmente se puede crear un cliente de desplazamiento que seguro que avisa al panel desplazable sobre el comportamiento de desplazamiento como los incrementos de unidad y de bloques. Estos tópicos se cubren en las siguientes secciones:

Cómo funciona un ScrollPane

Aquí puedes ver una imagen de una aplicación que utiliza un panel desplazable para ver una foto del padre de Mary cuando era joven.



El panel desplazable de esta aplicación parece totalmente diferente del de la aplicación anterior. En vez de texto, contiene una gran imagen. El ScrollPane también tiene dos barras de desplazamiento, un fila y una columna de cabecera y tres esquinas personalizadas, una de las cuales contiene un botón.

Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente es **ScrollDemo.java**. También necesitarás **ScrollablePicture.java**, **Rule.java**, **Corner.java** y **youngdad.jpeg**.
2. Mueve las barras de desplazamiento. Mira que la imagen se mueve y con ella las reglas vertical y horizontal.
3. Pulsa el botón **cm** en la esquina superior izquierda. Las unidades de las cabecera de fila y de columna cambian a pulgadas (o vuelve a centímetros).

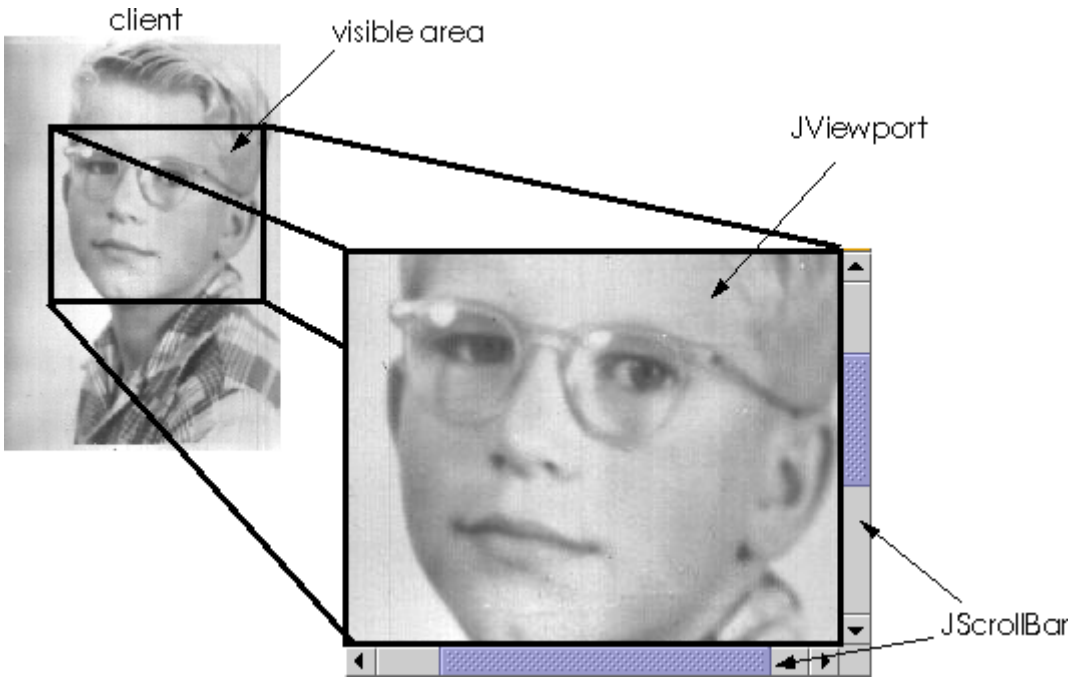
4. Pulsa las flechas de las barras de desplazamiento. También, pulsa sobre el camino de las barras de desplazamiento arriba o debajo de la barra vertical, o a la izquierda o derecha de la barra horizontal.
5. Aumenta el tamaño de la ventana. Observa que las barras desaparecen cuando no son necesarias y como resultado las esquina añadidas también desparecen. Disminuye la ventana y las barras de desplazamiento y las esquinas reaparecerán.

Este programa establece el cliente cuando crea el panel desplazable.

```
// where the member variables are declared
private ScrollablePicture picture;
...
                // where the GUI is created
        picture = new ScrollablePicture( ... );
        JScrollPane pictureScrollPane = new JScrollPane(picture);
```

Se puede cambiar dinámicamente el cliente del panel desplazable llamado al método **setViewportView**.

Cuando se manipulan las barras de desplazamiento de un ScrollPane, se cambia el área del cliente que es visible. Es imagen muestra esta relación e indica las clases que ayudan al ScrollPane.



Cuando se mueve la barra de desplazamiento arriba y abajo, el área visible del cliente se mueve arriba y abajo. De forma similar trabaja la barra de desplazamiento horizontal.

Un ScrollPane utiliza un ejemplar de **JViewport** para manejar el área visible de un cliente. Este objeto calcula los límites del área visible actual, basándose en las posiciones de las barras de desplazamiento, y la muestra. Un ScrollPane utiliza dos ejemplares separados de **JScrollBar** para las barras de desplazamiento. Las barras proporcionan el interface para que el usuario manipule el área visible.

Normalmente los programas no ejemplarizan directamente, ni llaman a los métodos de JViewPort o JScrollBar. En su lugar, los programas alcanzan su comportamiento desplazable utilizando el API de **JScrollPane** o el API descrito en Implementar un Cliente de Desplazamiento Seguro. Algunos componentes de desplazamiento seguro como **JTable** y **JTree** también proporcionan algún API para ayudar a efectuar su desplazamiento.

Seleccionar el Vigilante de ScrollBar

Cuando arranca, la aplicación **ScrollDemo** contiene dos barras de desplazamiento. Si agrandamos la ventana, las barras de desplazamiento desparecen porque ya no son necesarias. Este comportamiento esta controlado por el **Vigilante de SCrollBar**. Realmente hay dos vigilantes, uno para cada una de las barras de desplazamiento.

De los constructores proporcionados por **JScrollPane**, dos nos permite seleccionar vigilantes de barras de desplazamiento cuando creamos nuestro ScrollPane.

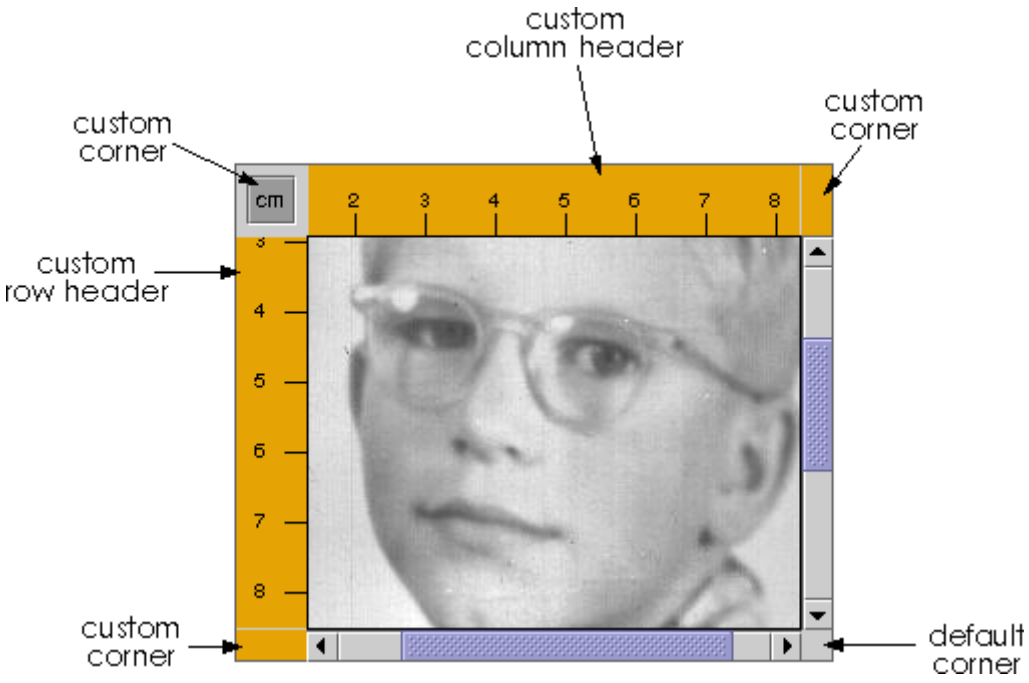
```
JScrollPane(Component, int, int)
JScrollPane(int, int)
```

El primer **int** especifica el vigilante para la barra vertical, y el segundo para la horizontal. También se pueden seleccionar los vigilantes con los métodos **setHorizontalScrollBarPolicy** y **setVerticalScrollBarPolicy**. En ambos casos se utilizan los siguientes enteros definidos en el interface **ScrollPaneConstants** que es implementado por **JScrollPane**.

Vigilante	Descripción
VERTICAL_SCROLLBAR_AS_NEEDED HORIZONTAL_SCROLLBAR_AS_NEEDED	Valor por defecto. Las barras de desplazamiento aparecen cuando el JViewPort es más pequeño que el cliente y desaparecen cuando es más grande.
VERTICAL_SCROLLBAR_ALWAYS HORIZONTAL_SCROLLBAR_ALWAYS	Siempre muestra las barras.
VERTICAL_SCROLLBAR_NEVER HORIZONTAL_SCROLLBAR_NEVER	Nunca muestra las barras de desplazamiento. Se utiliza esta opción si no queremos darle al usuario el control sobre la parte del cliente a visualizar. Quizás tengamos alguna aplicación que requiera que el desplazamiento ocurra programáticamente.

Proprorcionar Decoración Personalizada

El área dibujada por un ScrollPane está dividida, al menos, en nueve partes: el centro, cuadro laterales y cuatro esquinas. El centro es el único componente que siempre está presente en un ScrollPane. Cada uno de los cuatro lados son opcionales. El lado superior puede contener una columna de cabecera, el lado izquierdo puede contener una fila de cabecera, el lado inferior puede contener una barra de desplazamiento horizontal, y el lado derecho puede tener una barra de desplazamiento vertical. La presencia de las cuatro esquinas depende completamente de la presencia de los dos laterales que interseccionan en ellas.



Como se ve en la figura, el ScrollPane de **ScrollDemo.java** tiene cabeceras de fila y columna personalizadas. Además, como los cuatro laterales están llenos, las cuatro esquinas están presentes. Tres de las esquinas también están personalizadas.

Las cabeceras de fila y columna del ScrollPane están proporcionadas por una subclase personalizada de **JComponent**, **Rule.java**, que dibuja una regla en centímetros o pulgadas. Aquí está el código que crea y selecciona las cabecetas de fila y columna del ScrollPane.

```
...where the member variables are defined...
private Rule columnView;
private Rule rowView;
    ...
    // Create the row and column headers
    columnView = new Rule(Rule.HORIZONTAL, false);
    columnView.setPreferredWidth(david.getIconWidth());
    rowView = new Rule(Rule.VERTICAL, false);
    rowView.setPreferredHeight(david.getIconHeight());
    ...
    pictureScrollPane.setColumnHeaderView(columnView);
    pictureScrollPane.setRowHeaderView(rowView);
    ...
```

Se pueden utilizar componentes de peso ligero para cabeceras de fila y columna de un ScrollPane. El ScrollPane pone las cabeceras de fila y columna en su propio **JViewPort**. Así, cuando se desplaza horizontalmente, la cabecera de columnas la sigue, y cuando se desplaza verticalmente, la cabecera de filas también lo hace.

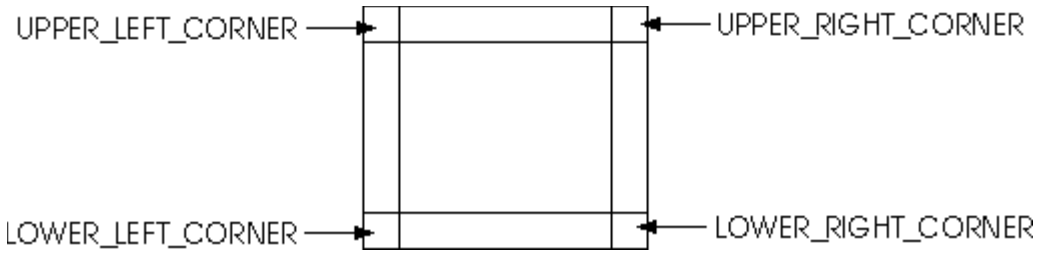
Como subclase de **JComponent**, **Rule** se dibuja a sí misma sobreescriendo el método **paintComponent**. Un escurtinio cuidadoso del código revela que se ha tomado un esfuerzo especial para dibujar sólo dentro de los límites actuales. Nuestras cabeceras de fila y columna deben hacer los mismo para asegurarnos un desplazamiento rápido.

También podemos utilizar cualquier componente de peso ligero para las esquinas de un ScrollPanel. **ScrollDemo.java** ilustra esto poniendo un botón en la esquina superior izquierda, y objetos **Corner** personalizados en las esquinas superior derecha e inferior izquiedda. Aquí está el código que crea los objetos **Corner** y llama a **setCorner** para situarlos.

```
// Create the corners
JPanel buttonCorner = new JPanel();
isMetric = new JToggleButton("cm", true);
isMetric.setFont(new Font("SansSerif", Font.PLAIN, 11));
isMetric.setMargin(new Insets(2,2,2,2));
isMetric.addItemListener(new UnitsListener());
buttonCorner.add(isMetric); //Use the default FlowLayout

./ Set the corners:
pictureScrollPane.setCorner(JScrollPane.UPPER_LEFT_CORNER,
                                                    buttonCorner);
pictureScrollPane.setCorner(JScrollPane.LOWER_LEFT_CORNER,
                                                    new Corner());
pictureScrollPane.setCorner(JScrollPane.UPPER_LEFT_CORNER,
                                                    new Corner());
```

Recuerda que el tamaño de cada esquina está determinado completamente por el tamaño de los lados que interseccciona allí. Debemos tener cuidado de que el objeto quepa en la esquina. La clase **Corner** lo hace dibujando dentro de sus límites, que son configurados por el ScrollPane. El botón fue configurado específicamente para caber en la esquina establecida por las cabeceras.

Como podemos ver desde el código, las constantes indican la posición de las esquinas. Esta figura muestra las constantes para cada posición.

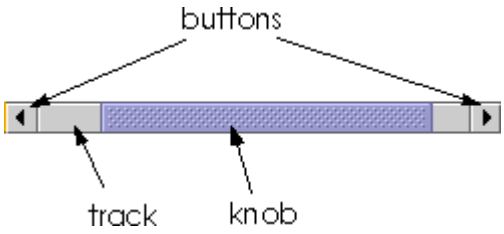


Las constantes están definidas en el interface **ScrollPaneConstants**, que es implementado por **JScrollPane**.

Implementar un Cliente de Desplazamiento Seguro

Para personalizar la forma en la que el componente cliente interactúa con su ScrollPane, podemos hacer que el componente implemente el interface **Scrollable**. Implementando este interface, un cliente puede especificar tanto al tamaño del cuadro de visión que el ScrollPane utilizará para verlo como la cantidad de desplazamiento para los diferentes controles de las barras de desplazamiento.

La siguiente figura muestra las tres áreas de un barra de desplazamiento: la barra, los botones y la pista.



Habrás podido observar cuando manipulabas las barras de desplazamiento de **ScrollDemo** que pulsando los botones la imagen se desplaza un poco. También podrías haber observado que al pulsar sobre la pista el desplazamiento es mayor. Más generalmente, el botón desplaza el área visible **una unidad de incremento** y la pista desplaza el área visible **un bloque de incremento**. El comportamiento que has visto en el ejemplo no es el comportamiento por defecto de un ScroolPane, pero si es especificado por el cliente en su implementación del interface **Scrollable**.

El cliente del programa **ScrollDemo** es **ScrollablePicture.java**. **ScrollablePicture** es una subclase de **JLabel** que proporciona implementaciones para los cinco métodos de **Scrollable**.

- **getScrollableBlockIncrement**
- **getScrollableUnitIncrement**
- **getPreferredViewportSize**
- **getScrollableTracksViewportHeight**
- **getScrollableTracksViewportWidth**

ScrollablePicture implementa el interface **Scrollable** principalmente para afectar a los incrementos de unidad y de bloque. Sin embargo, debe proporcionar implementaciones para los cinco métodos. Las implementaciones para los tres últimos son razonables por defecto.

El ScrollPane llama al método **getScrollableUnitIncrement** del cliente siempre que el usuario pulse uno de los botones de las barras de desplazamiento. Este método devuelve el número de pixels a desplazar. Una implementación obvia de este método devuelve el número de pixels entre marcas de las reglas de cabecera. Pero **ScrollablePicture** hace algo diferente: devuelve el valor requerido para posiconar la imagen en el límite de una marca. Aquí está la implementación.

```
public int getScrollableUnitIncrement(Rectangle visibleRect,
                                      int orientation,
                                      int direction) {
    //get the current position
    int currentPosition = 0;
    if (orientation == SwingConstants.HORIZONTAL)
        currentPosition = visibleRect.x;
    else
        currentPosition = visibleRect.y;

    //return the number of pixels between currentPosition
    //and the nearest tick mark in the indicated direction
    if (direction < 0) {
        int newPosition = currentPosition -
                                         (currentPosition / maxUnitIncrement) *
                                          maxUnitIncrement;
        return (newPosition == 0) ? maxUnitIncrement : newPosition;
    } else {
        return ((currentPosition / maxUnitIncrement) + 1) *
                               maxUnitIncrement - currentPosition;
    }
}
```

Si la imagen ya se encuentra en un marca, este método devuelve el número de pixels entre marcas. De otro modo devuelve el número de pixels entre la posición actual y la marca más cercana.

De igual modo el ScrollPane llama al método **getScrollableBlockIncrement** del cliente cada vez que el usuario pulsa sobre la pista de la barra de desplazamiento. Aquí está la implementación que **ScrollablePicture** hace de este método.

```
public int getScrollableBlockIncrement(Rectangle visibleRect,
                                       int orientation,
                                       int direction) {
    if (orientation == SwingConstants.HORIZONTAL)
        return visibleRect.width - maxUnitIncrement;
    else
        return visibleRect.height - maxUnitIncrement;
}
```

Este método devuelve la altura del rectángulo visible menos una marca. Este comportamiento es típico. Un incremento de bloque debería permitir que el JViewpor deje un poco del área visible anterior por razones de contexto. Por ejemplo, un área de texto podría dejar una o dos líneas y una tabla podría dejar una fila o una columna (dependiendo de la dirección de desplazamiento).

Puedes ver la tabla Implementar el Interface Scrollable para obtener más detalles sobre los métodos definidos en **Scrollable**.

El paquete Swing proporciona estas clases de desplazamiento seguro.

- listas
- tablas
- componentes de texto
- árboles

El API de ScrollPane

Las siguiente tablas listan los métodos y constructores más utilizados de **JScrollPane**. Otros métodos útiles son definidos por las clases **JComponent** y **Component**.

El API para utilizar ScroolPane se divide en estas categorías:

Configurar el ScrollPane

Método	Propósito
JScrollPane() **JScrollPane(Component)** **JScrollPane(int, int)** **JScrollPane(Component, int, int)**	Crea un ScrollPanel El parámetro **Component**, cuando existe, selecciona el cliente. Los dos parámetros **int**, cuando existen, seleccionan los vigilantes de seguridad de las barras de desplazamiento vertical y horizontal (respectivamente).
void setViewportView(Component)	Selecciona el cliente del ScrollPane.
void	Selecciona u obtiene el vigilante de desplazamiento vertical.

setVerticalScrollBarPolicy(int) **int getVerticalScrollBarPolicy()**	**ScrollPaneConstants** define tres valores para especificar estos vigilantes: **VERTICAL_SCROLL_BAR_AS_NEEDED** (por defecto), **VERTICAL_SCROLL_BAR_ALWAYS**, y **VERTICAL_SCROLL_BAR_NEVER**.
void setHorizontalScrollBarPolicy(int) **int getHorizontalScrollBarPolicy()**	Selecciona u obtiene el vigilante de desplazamiento horizontal. **ScrollPaneConstants** define tres valores para especificar estos vigilantes: **HORIZONTAL_SCROLL_BAR_AS_NEEDED** (por defecto), **HORIZONTAL_SCROLL_BAR_ALWAYS**, y **HORIZONTAL_SCROLL_BAR_NEVER**.
void setViewportBorder(Border) **Border getViewportBorder()**	Selecciona u obtiene el borde alrededor del JViewport.

Decorar el ScrollPane

Método	Propósito
void setColumnHeaderView(Component) **void setRowHeaderView(Component)**	Selecciona la cabecera de fila o de columna para el ScrollPane.
void setCorner(Component, int) **Component getCorner(int)**	Seleciona u obtiene la esquina especificada. El parámetro **int** indica qué columna y debe ser una de las siguientes constantes definidas en **ScrollPaneConstants**: **UPPER_LEFT_CORNER**, **UPPER_LEFT_CORNER**, **LOWER_LEFT_CORNER**, y **LOWER_LEFT_CORNER**.

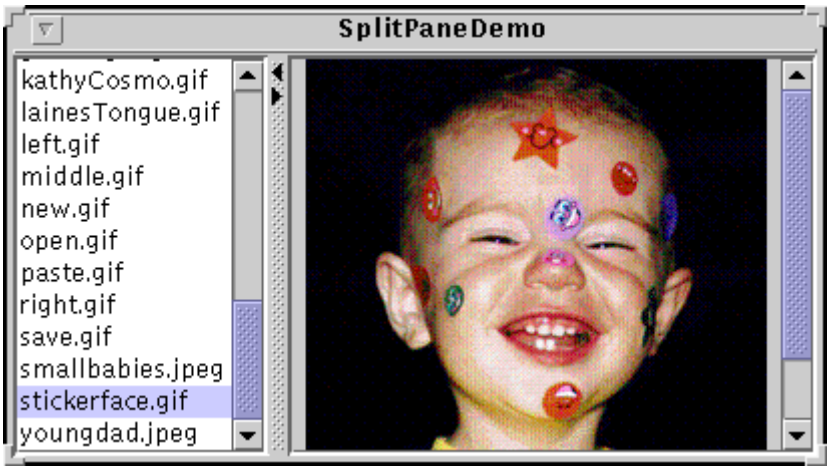
Implementar el Interface Scrollable

Método	Propósito
int getScrollableUnitIncrement(Rectangle, int, int) **void getScrollableBlockIncrement(Rectangle, int, int)**	Obtiene el incremento de unidad o de bloque en pixels. El parámetro **Rectangle** son los límites del área visible actualmente. El primer parámetro **int** es **SwingConstants.HORIZONTAL** o **SwingConstants.VERTICAL** dependiendo de la barra que haya pulsado el usuario. El segundo parámetro **int** indica la dirección del desplazamiento. Un valor menor que 0 indica arriba o izquierda. Una valor mayor que 0 indica abajo o derecha.
Dimension getPreferredScrollableViewportSize()	Obtiene el tamaño preferido del JViewport. Esto permite al cliente influenciar en el tamaño del componente en el que va ser mostrado. Si este tamaño no es importante devuelve **getPreferredSize**.
boolean getScrollableTracksViewportWidth() **boolean getScrollableTracksViewportHeight()**	Obtiene su el ScrollPane debería forzar al cliente a tener la misma anchura o altura que el JViewport. Devolver **true** por alguno de esto métodos efectivamente desactiva el desplazamiento horizontal o vertival (respectivamente).

¿Cómo Usar SplitPane?

Un **JSplitPane** contiene dos componentes de peso ligero, separados por un divisor. Arrastrando el divisor, el usuario puede especificar qué cantidad de área pertenece a cada componente. Un SplitPane se utiliza cuando dos componentes contienen información relacionada y queremos que el usuario pueda cambiar el tamaño de los componentes en relación a uno o a otro. Un uso común de un SplitPane es para contener listas de elecciones y una visión de la elección actual. Un ejemplo sería un programa de correo que muestra una lista con los mensajes y el contenido del mensaje actualmente seleccionado de la lista.

Aquí tenemos una imagen de una aplicación que usa un SplitPane para mostrar una lista y una imagen lado a lado.



Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente es **SplitPaneDemo.java**. **imagenames.properties** proporciona los nombres de las imagenes para poner en el **JList**.
2. Arrastra la línea que divide la lista y la imagen a la izquierda o a la derecha. Intenta arrastrar el divisor más allá del límite de la ventana.
3. Utiliza las flechas del divisor para ocultar alguno de los componentes.

La clase principal del programa de ejemplo se llama **SplitPaneDemo** y es una subclase de **JSplitPane**. Aquí podemos ver el código del constructor de **SplitPaneDemo** que crea y configura el SplitPane.

```
public SplitPaneDemo() {

    // Create the list of images and put it in a scroll pane
    ...
    // Set up the picture label and put it in a scroll pane
    ...
    //Create a split pane with the two scroll panes in it.
    splitPane = new JSplitPane(JsplitPane.HORIZONTAL_SPLIT);
    splitPane.setLeftComponent(listScrollPane);
    splitPane.setLEFTComponent(pictureScrollPane);
    splitPane.setOneTouchExpandable(true);

    // Provide minimum sizes for the two components in the split pane
    Dimension minimumSize = new Dimension(100, 50);
    listScrollPane.setMinimumSize(minimumSize);
    pictureScrollPane.setMinimumSize(minimumSize);

    // Set the initial location and size of the divider
    splitPane.setDividerLocation(150);
    splitPane.setDividerSize(10);

    // Provide a preferred size for the split pane
    splitPane.setPreferredSize(new Dimension(400, 200));
}
```

La primera línea del constructor divide el SplitPane horizontalmente, por lo tanto pone los dos componentes lado a lado. SplitPane proporciona otra opción, **VERTICAL_SPLIT**, que sitúa los componentes uno sobre otro. Podemos cambiar la direción de división despues de haber creado el SplitPane con el método **setOrientation**.

Luego el código selecciona "one touch expandable" a **true**. Con esta opción activada. el SplitPane muestra controles que permiten al usuario ocultar uno de los componentes y asignar todo el espacio al otro.

El constructor utiliza **setLeftComponent** y **setLEFTComponent** para situar la lista y la etiqueta de imagen en el SplitPane. Si el SplitPane tubiera orientación vertical, podríamos utilizar **setTopComponent** y **setBottomComponent** en su lugar. Sin embargo, cada uno de los métodos **setXxxxComponent** funciona sin importarle la orientación del SplitPane. 'Top' y 'Left' son equivalentes y 'Bottom' y 'LEFT' son equivalentes. De forma alternativa, podemos utilizar el método **add** que pone el primer componente en la posición izquierda o superior.

Podrías haver observado que el código precedente crea una confusión sobre los tamaños mínimos de los componentes contenidos por el SplitPane. La razón es que un SplitPane utiliza el tamaño mínimo de sus componentes para determinar hasta dónde puede el usuario mover el divisor. Un SplitPane no permite que el usuario haga un componente más pequeño que su tamaño mínimo moviendo el divisor. El usuario puede utilizar los botones explandibles para ocultar un componente. Si este ejemplo no seleccionara el tamaño mínimo de al menos uno de los componentes del SplitPane, el divisor podría ser inamovible porque cada componente podría ser más pequeño que su tamaño mínimo.

El API de SplitPane

Las siguientes tablas listan los métodos y constructores más utilizados de **JSplitPane**. Otros métodos útiles están definidos por las clases **JComponent**, **Container** y **Component**.

El API para usar SplitPane se divide en estas categorías.

Configurar el SplitPane

Método	Propósito
JSplitPane() **JSplitPane(int)** **JSplitPane(int, boolean)** **JSplitPane(int, Component, Component)** **JSplitPane(int, boolean, Component, Component)**	Crea un SplitPane. Cuando existe, el parámetro **int** indica la orientación del SplitPane, **HORIZONTAL_SPLIT** o **VERTICAL_SPLIT**. El paramétodo booleano selecciona si los componentes se redibujan contínuamnete cuando el usuario arrastra el divisor. Los parámetros **Component** seleccionan los componentes izquierdo y derecho o superior e inferior, respectivamente.
void setOrientation(int) **int getOrientation()**	Selecciona u obtiene la orientación del SplitPane. Se utilizan **HORIZONTAL_SPLIT** o **VERTICAL_SPLIT** definidas en **JSplitPane**.
void setDividerSize(int) **int getDividerSize()**	Selecciona u obtiene el tamaño del divisor en pixels.
void setContinuousLayout(boolean) **boolean getContinuousLayout()**	Selecciona u obtiene si los componetes del SplitPane se redistribuyen y redibujan mientras el usuario arrastra el divisor.
void setOneTouchExpandable(boolean) **boolean getOneTouchExpandable()**	Selecciona u obtiene si el SplitPane muestra los botones de control expandible.

Manejar los Contenidos del SplitPanel

Método	Propósito
void setTopComponent(Component) **void setBottomComponent(Component)** **void setLeftComponent(Component)** **void setLEFTComponent(Component)** **Component getTopComponent()** **Component getBottomComponent()** **Component getLeftComponent()** **Component getLEFTComponent()**	Selecciona u obtiene el componente indicado. Cada método funciona sin importarle la orientación del SplitPane.
void remove(Component) **void removeAll()**	Elimina el componente indicado del SplitPane.
void add(Component)	Añade el componete al SplitPane. Podemos añadir dos componentes a un SplitPane. El primer componente se añade al componente superior/izquierda; y el segundo al componente inferior/derecha.

Posicionar el Divisor

Método	Propósito

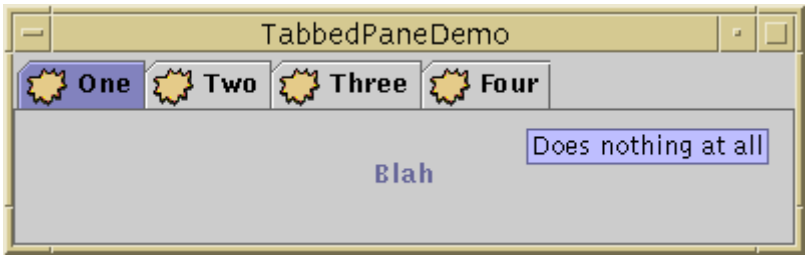
void setDividerLocation(double) **void setDividerLocation(int)** **int getDividerLocation()**	Selecciona u obtiene la posición actual del divisor. Cuando se selecciona la posición, podemos especificar un porcentaje (**double**) o una posición de pixel (**int**). **Nota**: Las veriones Swing 1.0.3 y anteriores tenían un bug por el que **setDividerLocation** era ignorado si se le llamaba antes de que los componentes fueran visibles. Para evitar esto, se seleccioa la anchura (o altura) preferida del componente de la izquierda (o superior) al mismo valor en que queríamos usar como argumento de **setDividerLocation(int)**.
void setLastDividerLocation(int) **int getLastDividerLocation()**	Selecciona u obtiene la posición anterior del divisor.
int getMaximumDividerLocation() **int getMinimumDividerLocation()**	Obtienen las posiciones mínima y máxima del divisor. Estas se seleccionan implícitamente seleccionando los tamaños mínimos de los dos componetes del SplitPane.

¿Cómo Usar TabbedPane?

Con la clase **JTabbedPane**, podemos tener varios componentes (normalmente objetos **JPanel**) compartiendo el mismo espacio. El usuario puede elegir qué componente ver seleccionando la pestaña del componente deseado.

Para crear un TabbedPane, simplemente se ejemplariza un **JTabbedPane**, se crean los componentes que deseemos mostrar, y luego los añadimos al TabbedPane utilizando el método **addTab**.

Aquí tenemos una imagen de una aplicación que utiliza tres TabbedPane.



Intenta esto:

1. Compila y ejecuta la aplicación. El código fuente esta en **TabbedPaneDemo.java**.
2. Pon el cursor sobre una pestaña. Después de un corto tiempo, verás una ayuda (tooltip) asociada con la pestaña. Como conveniencia se debe añadir el texto de la ayuda (tooltip) cuando se añade el componente al TabbedPane.
3. Selecciona una pestaña. El TabbedPane muestra el componente correspondiente a la pestaña.

Como muestra el ejemplo **TabbedPaneDemo**, una pestaña puede tener un tooltip, y puede mostrar tanto texto como una imagen. El ejemplo muestra las pestañas en sus posiciones por defecto, en la parte superior del TabbedPane. Podemos cambiar la posiciones de las pestañas a la izquierda, derecha, o abajo.

Aquí está el código de **TabbedPaneDemo.java** que crea el TabbedPane del ejemplo anterior. Observa que no es necesario el manejo de eventos. El objeto **JTabbedPane** tiene cuidado de manejar la entrada de usuario.

```
ImageIcon icon = new ImageIcon("images/middle.gif");
JTabbedPane tabbedPane = new JTabbedPane();

Component panel1 = makeTextPanel("Blah");
tabbedPane.addTab("One", icon, panel1, "Does nothing");
tabbedPane.setSelectedIndex(0);

Component panel2 = makeTextPanel("Blah blah");
tabbedPane.addTab("Two", icon, panel2, "Does twice as much nothing");

Component panel3 = makeTextPanel("Blah blah blah");
tabbedPane.addTab("Three", icon, panel3, "Still does nothing");

Component panel4 = makeTextPanel("Blah blah blah blah");
tabbedPane.addTab("Four", icon, panel4, "Does nothing at all");
```

El API TabbedPane

Las siguientes tablas listan los métodos y constructores más utilizados de **JTabbedPane**. El API para utilizar TabbedPane se divide en estas categorías.

Crear y Configurar un TabbedPane

Método	Propósito
JTabbedPane() **JTabbedPane(int)**	Crea un TabbedPane. El argumento opcional indica dónde deberían aparecer las pestañas. Por defecto, las pestañas aparecen en la parte superior. Se pueden especificar estas posiciones (definidas en el interface **SwingConstants**, que implementa **JTabbedPane**): **TOP**, **BOTTOM**, **LEFT**, **LEFT**.
addTab(String, Icon, Component, String) **addTab(String, Icon, Component)** **addTab(String, Component)**	Añade una nueva pestaña al TabbedPane. El primer argumento especifica el texto de la pestaña. El argumento Icon es opcional e indica el icono de la pestaña. El argumento **Component** especifica el componente que el TabbedPane debería mostrar cuando se selecciona la pestaña. El cuarto argumento, si existe, especifica el texto del tooltip para la pestaña.

Insertar, Eliminar, Encontrar y Seleccionar Pestañas

Método	Propósito
insertTab(String, Icon, Component, String, int)	Inserta una pestaña en el índice especificado, donde la primera pestaña tiene índice 0. Los argumentos son los mismos que para **addTab**.
remove(Component) **removeTabAt(int)**	Elimina la pestaña correspondinete al índice o componente especificado.
removeAll()	Elimina todas las pestañas.
int indexOfComponent(Component) **int indexOfTab(String)** **int indexOfTab(Icon)**	Devuelve el índice de la pestaña que tiene el componente, título o icono especificados.
void setSelectedIndex(int) **void setSelectedComponent(Component)**	Selecciona la pestaña que tiene el índice o componente especificado. Seleccionar una pestaña tiene el efecto de mostrar su componente asociado.
int getSelectedIndex() **Component getSelectedComponent()**	Devuelve el índice o componente de la pestaña seleccionada.

Cambiar la Apariencia de las Pestañas

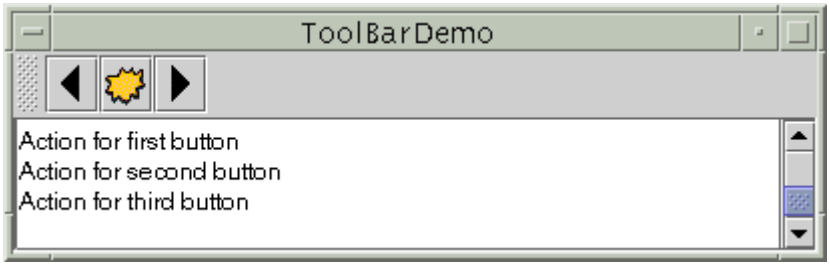
Método	Propósito
void setComponentAt(int, Component) **Component getComponentAt(int)**	Selecciona u obtiene qué componente está asociado con la pestáña del índice especificado. La primera pestaña tiene índice 0.
void setTitleAt(int, String) **String getTitleAt(int)**	Selecciona u obtiene el título de la pestaña del índice especificado.
void setIconAt(int, Icon) **Icon getIconAt(int)** **void**	Selecciona u obtiene los iconos mostrados por la pestaña del índice especificado.

setDisabledIconAt(int, Icon) **Icon getDisabledIconAt(int)**	
void setBackgroundAt(int, Color) **Color getBackgroundAt(int)** **void setForegroundAt(int, Color)** **Color getForegroundAt(int)**	Selecciona u obtiene el color de fondo o de primer plano usado por la pestaña del índice especificado. Por defecto, una pestaña utiliza los colores del TabbedPane. Por ejemplo, si el color de primer plano del TabbedPane es negro, entonces todos los títulos de las pestañas serán en negro, excepto para aquellas en que especifiquemos otro color usando **setForegroundAt**.
void setEnabledAt(int, boolean) **boolean isEnabledAt(int)**	Selecciona u obtiene el estado activado de la pestaña del índice especificado.

¿Cómo Usar TollBar?

Un objeto **JToolBar** crea una barra de herramientas con iconos -- dentro de una fila o una columna. Normalmente las barras de herramientas proporcionan acceso a funcionalidades que también se encuentran en ítems de menús. Si este es el caso, además de esta página deberías leer Cómo usar Actions.

La siguiente imagen muestra una aplicación que contiene una barra de herramientas sobre un área de texto.



Por defecto, el usuario puede arrastrar la barra de herramientas a un lateral distinto de su contenedor o fuera dentro de su propia ventana. La siguiente figura muestra cómo aparece la aplicación después de que el usuario haya arrastrado la barra de herramientas al lateral derecho de su contenedor. Para hacer que el arrastre de la barra de herramientas funcione correctamente, la barra debe estar en un contenedor que use **BorderLayout**, y el contenedor sólo debe tener otro componente que esté situado en el centro.

La siguiente figura muestra cómo aparece la aplicación después de que el usuario haya arrastrado la barra de herramientas fuera de su ventana.

El siguiente código implementa la barra de herramientas. Puedes encontrar el programa complento en **ToolBarDemo.java**. Y las imágenes en **left.gif**, **middle.gif**, y **LEFT.gif**.

Nota: Si algún botón de nuestra barra de herramientas duplica la funcionalidad de otros componentes, como un ítem de menú, probablemente deberíamos crear y añadir los botones de la barra de herramientas como se describe en Cómo usar Actions.

```
public ToolBarDemo() {
    ...
    JToolBar toolBar = new JToolBar();
    addButtons(toolBar);
    ...
    JPanel contentPane = new JPanel();
    contentPane.setLayout(new BorderLayout());
    ...
    contentPane.add(toolBar, BorderLayout.NORTH);
    contentPane.add(scrollPane, BorderLayout.CENTER);
    ...
}

protected void addButtons(JToolBar toolBar) {
    JButton button = null;

    //first button
    button = new JButton(new ImageIcon("images/left.gif"));
    ...
    toolBar.add(button);

    //second button
    button = new JButton(new ImageIcon("images/middle.gif"));
    ...
    toolBar.add(button);

    //third button
    button = new JButton(new ImageIcon("images/LEFT.gif"));
    ...
    toolBar.add(button);
}
```

Añadiendo unas pocas líneas de código al ejemplo anterior, podemos demostrar algunas características más de las barras de herramientas.

- Usar el método **setFloatable** para hacer que la barra no se pueda mover.
- Añadir un separador a una barra de herramientas.
- Añadir un componente que no sea un botón a una barra de herramientas.

Aquí está una imagen del nuevo GUI, que está implementado por **ToolBarDemo2.java**.

Como la barra de herramientas ya no se puede arrastrar, no tiene el marcado en su flanco izquierdo. Aquí está el código que desactiva el arrastre.

```
toolBar.setFloatable(false);
```

La mayor diferencia visible es que la barra de herramientas contiene dos componentes nuevos, que están precedidos por un espacio en blanco -- un separador . Aquí está el código que añade el separador.

```
toolBar.addSeparator();
```

Aquí está el código que añade los nuevos componentes.

```
./add to where the first button is initialized:
```

```
button.setAlignmentY(CENTER_ALIGNMENT);
./add to where the second button is initialized:
button.setAlignmentY(CENTER_ALIGNMENT);
./add to where the third button is initialized:
button.setAlignmentY(CENTER_ALIGNMENT);
...
//fourth button
button = new JButton("Another button");
...
button.setAlignmentY(CENTER_ALIGNMENT);
toolBar.add(button);

//fifth component is NOT a button!
JTextField textField = new JTextField("A text field");
...
textField.setAlignmentY(CENTER_ALIGNMENT);
toolBar.add(textField);
```

La llamada a **setAlignmentY** es necesaria para que los componentes de la barra de herramientas se alineen correctamente. Si el código no seleccionara el alineamiento, el campo de texto se posicionaría demasiado arriba. Este es el resultado de **JToolBar** usando **BoxLayout** como controlador de distribución, y los botones y campos de texto tienen distinto alineamiento Y por defecto. Si encontramos problemas de distribución con una barra de herramientas podemos ver la página Cómo usar BoxLayout para buscar ayuda.

El API Tool Bar

Las siguientes tablas listan los métodos y constructores más utilizados de **JToolBar**. Otros métodos útiles están definidos por las clases **JComponent**, **Container**, y **Component**.

Método	Propósito
JToolBar()	Crea una barra de herramientas.
JButton add(Action) **Component add(Component)**	Añade un componente (normalmente un botón) a la barra de herramientas. Si el argumento de **add** es un objeto **Action**, la barra de herramientas crea automáticamente un **JButton** y lo añade.
void addSeparator()	Añade un separador al final de la barra de herramientas.
void setFloatable(boolean) **boolean isFloatable()**	La propiedad floatable es true por defecto, para indicar que el usuario puede arrastrar la barra de herramientas a una ventana separada. Para desactivar el arrastre de la barra de herramientas se utiliza **toolbar.setFloatable(false)**.

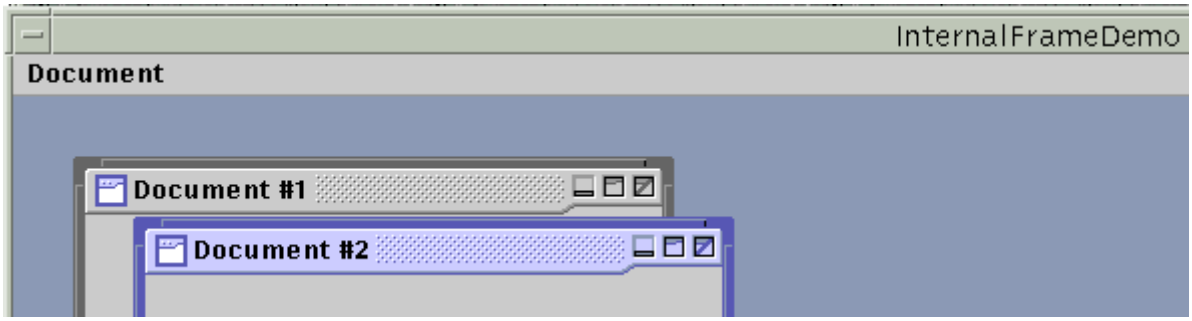
¿Cómo Usar InternalFrame?

Con la clase **JInternalFrame**, se puede mostrar un **JFrame** - como una ventana dentro de otra ventana. Para crear un frame interno que parezca un diálogo sencillo, se pueden utilizar los métodos **showInternalXxxDialog** de **JOptionPane**, como se explicó en Cómo crear Diálogos.

Normalmente, los frames internos se muestran dentro de un **JDesktopPane**.

JDesktopPane es una subclase de **JLayeredPane** al que se le ha añadido el API para manejar el solapamiento de múltiples frames internos. Generalmente, se pone el panel superior dentro del panel de contenido de un **JFrame**. Para más información sobre el uso de API que **JDesktopPane** hereda de **JLayeredPane**, puedes ver Cómo usar LayeredPane.

Aquí podemos ver una imagen de una aplicación que tiene dos frames internos dentro de un frame normal.



Como se ve en la figura, los frames internos utilizan la decoración de ventana del aspecto y comportamiento Metal. Sin embargo, la ventana que los contiene tiene decoración de aspecto y comportamiento nativo (en este caso, Motif).

Intenta esto:

1. Compila y ejecuta la aplicación, Los ficheros fuentes son.

   InternalFrameDemo.java y **MyInternalFrame.java**.

2. Crea nuevos frames internos utilizando el ítem Create en el menú Document.

   Cada frame interno viene 30 pixels por debajo y a la derecha de la posición del frame anterior. Esta funcionalidad se implementa en la clase **MyInternalFrame**, que es la subclase peronalizada de **JInternalFrame**.

El siguiente código, tomado de **InternalFrameDemo.java**, crea el frame principal y los internos del ejemplo anterior.

```
./In the constructor of InternalFrameDemo, a JFrame subclass:
    desktop = new JDesktopPane(); //a specialized layered pane
    createFrame(); //Create first window
    setContentPane(desktop);
...
protected void createFrame() {
    MyInternalFrame frame = new MyInternalFrame();
    desktop.add(frame);
    try {
        frame.setSelected(true);
    } catch (java.beans.PropertyVetoException e2) {}
}

./In the constructor of MyInternalFrame, a JInternalFrame subclass:
static int openFrameCount = 0;
static final int xOffset = 30, yOffset = 30;

public MyInternalFrame() {
    super("Internal Frame #" + (++openFrameCount),
          true, //resizable
          true, //closable
          true, //maximizable
          true);//iconifiable
    //...Create the GUI and put it in the window...
    //...Then set the window size or call pack...
    ...
    //Set the window's location.
    setLocation(xOffset*openFrameCount, yOffset*openFrameCount);
}
```

Frames Internos frente a Frames Normales

El código para utilizar frames internos es similar en muchas formas al código para utilizar frames normales Swing. Como los frames internos tienen sus paneles raíz, configurar el GUI para un **JInternalFrame** es muy similar a configurar el GUI para un **JFrame**. **JInternalFrame** también proporciona otro API, como **pack**, que lo hace similar a **JFrame**.

Como los frames internos no son ventanas, de alguna forma son diferentes de los frames. Por ejemplo, debemos añadir un frame interno a un contenedor (normalmente un **JDesktopPane**). Un frame interno no genera eventos window; en su lugar, las acciones del usuario que podrían causar que un frame dispara eventos windows hacen que en un frame interno se disparen eventos "internal frame".

Como los frames internos se han implementado con código independiente de la plataforma, ofrecen algunas características que los frames no pueden ofrecer. Una de esas características es que los frames internos ofrecen más control sobre su estado y capacidades. Programáticamente se puede minimizar o maximizar un frame interno. También se puede especificar el icono que va en la barra de título del frame interno. Incluso podemos especificar si el frame tiene soporte de decoración de ventanas, redimensionado, minimización, cerrado y maximización.

Otra característica es que los frames internos se han diseñado para trabajar con paneles por capas. El API **JInternalFrame** contiene métodos como **moveToFront** que funcionan sólo si el contenedor de un frame interno es un layeredpane.

Reglas de utilización de Frames Internos

Si has construido algún programa utilizando **JFrame** y los otros componentes Swing, entonces ya sabes mucho sobre cómo utilizar frames internos. La siguiente lista sumariza las reglas para la utilización de frames internos.

Se debe seleccionar el tamaño del frame interno.
: Si no se selecciona el tamaño del frame interno, tendrá tamaño cero y nunca será visible. Se puede seleccionar el tamaño utilizando uno de estos métodos: **setSize**, **pack** o **setBounds**.

Como regla, se debe seleccionar la posición del frame interno.
: Si no se selecciona la localización, empezará en 0,0 (la esquina superior izquierda de su contenedor). Se pueden utilizar los métodos **setLocation** o **setBounds** para especificar la esquina superior izquierda del frame interno en relación a su contenedor.

Para añadir componentes a un frame interno, se añaden al panel de contenidos del propio frame interno.
: Es exactamente la misma situación que **JFrame**. Puedes ver Añadir componentes a un Frame para más detalles.

Los diálogos que son frames internos deberían implementarse utilizando JOptionPane o JInternalFrame, no JDialog.
: Para crear un diálogo sencillo, podemos utilizar los metodos **showInternalXxxDialog** de **JOptionPane**, como se describió en Cómo crear Diálogos. Observa que los diálogos en frames internos **no** son modales.

Un frame interno se debe añadir a un contenedor
: Si no lo añadimos a un contenedor (normalmente un **JDesktopPane**), el frame interno no aparecerá.

Normalmente no se tiene que llamar a show o setVisible para los frames internos.

Al igual que los botones, los frames internos se muestran automáticamene cuando se añaden a un contenedor visible o cuando su contenedor anteriormente invisible se hace visible.

Los frames internos disparan eventos "internal frame", no eventos "window".

El manejo de eventos "internal frame" es casi idéntico al manejo de eventos "window". Para más información puedes ver Cómo escribir un oyente "Internal Frame".

Nota: Debido a un bug (#4128975), una vez que un frame interno a aparecido y luego ha sido ocultado, el frame interno no aparecerá otra vez. Si queremos mostrarlo de nuevo, tenemos que recrearlo, como lo hace **InternalFrameEventDemo.java**.

El API de InternalFrame

Las siguientes tablas listan los métodos y constructores más utilizados de **JInternalFrame**. El API para utilizar frames internos se divide en estas categorías.

Junto con el API listado abajo, **JInternalFrame** hereda un API útil desde **JComponent**, **Container**, y **Component**. **JInternalFrame** también proporciona métodos para obtener y seleccionar objetos adicionales en su panel raíz. Para más detalles puedes ver Cómo usar RootPane.

Crear un Frame Interno

Constructor	Propósito
JInternalFrame() **JInternalFrame(String)** **JInternalFrame(String, boolean)** **JInternalFrame(String, boolean, boolean)** **JInternalFrame(String, boolean, boolean, boolean)** **JInternalFrame(String, boolean, boolean, boolean, boolean)**	Crea un ejemplar de **JInternalFrame**. El primer argumento especificar el título (si existe) a mostrar por el frame interno. El resto de los argumentos especifican si el frame interno debería contener decoración permitiendo al usuario que redimensione, cierre, maximice y minimice el frame interno (por este orden). El valor por defecto para cada argumento booleano es **false**, lo que significa que la operación no está permitida.
Métodos de la clase **JOptionPane**: • **showInternalConfirmDialog** • **showInternalInputDialog** • **showInternalMessageDialog** • **showInternalOptionDialog**	Crea un **JInternalFrame** que simila un diálogo.

Añadir Componentes a un Frame Interno

Método	Propósito
void setContentPane(Container) **Container getContentPane()**	Selecciona u obtiene el panel de contenido del frame interno, que generalmente contiene todo e GUI del frame interno, con la excepción de la barra de menú y las decoraciones de la ventana.
void setMenuBar(JMenuBar) **JMenuBar getMenuBar()**	Selecciona u obtiene la barra de menú del frame interno. Observa que estos nombres de método no contienen "J", al contrario que sus métodos equivalentes de **JFrame**. En las siguientes versiones de Swing y del JDK 1.2, **JInternalFrame** añadira **setJMenuBar** y **getJMenuBar**, que se deberían utilizar en vez de los métodos existentes.

Especificar el Tamaño y la Posición del Frame Interno

Método	Propósito
void pack()	Dimensiona el frame interno para que sus componentes tenga sus tamaños preferidos.
void setLocation(Point)	Seleciona la posición del frame interno. (Heredada de **Component**).

void setLocation(int, int)	
void setBounds(Rectangle) **void setBounds(int, int, int, int)**	Explicitámente selecciona el tamaño y la localización del frame interno (Heredada de **Component**).
void setSize(Dimension) **void setSize(int, int)**	Explicitámente selecciona el tamaño del frame interno. (Heredada de **Component**).

Realizar Operaciones de Ventana sobre el Frame Interno

Método	Propósito
void setDefaultCloseOperation(int) **int getDefaultCloseOperation()**	Selecciona u obtiene lo que hace el frame interno cuando el usuario intenta "cerrar" el frame. El valor por defecto es **HIDE_ON_CLOSE**. Otros posibles valores son **DO_NOTHING_ON_CLOSE** y **DISPOSE_ON_CLOSE**.
void addInternalFrameListener(InternalFrameListener) **void removeInternalFrameListener(InternalFrameListener)**	Añade o elimina un oyente de "internal frame" (**JInternalFrame** es equivalente a un oyente de "window").
void moveToFront() **void moveToBack()**	Si el padre del frame interno es un layeredpane, mueve el frame interno adelante o detrás (respectivamente) por sus capas).
void setClosed(boolean) **boolean isClosed()**	Selecciona u obtiene si el frame interno está cerrado actualmente.
void setIcon(boolean) **boolean isIcon()**	Minimiza o maximiza el frame interno, o determina si está minimizado actualmente.
void setMaximum(boolean) **boolean isMaximum()**	Maximiza o restaura el frame interno o determina si está maximizado.
void setSelected(boolean) **boolean isSelected()**	Selecciona u obtiene si el frame interno esta actualmente "seleccionado" (activado).

Controlar la Decoración y las Capacidades de la Ventana

Método	Propósito
void setFrameIcon(Icon) **Icon getFrameIcon()**	Seleciona u obtiene el icono mostrado en la barra de título del frame interno (normalmente en la esquina superior izquierda).
void setClosable(boolean) **boolean isClosable()**	Selecciona u obtiene si el usuario puede cerrar el frame interno.
void setIconifiable(boolean) **boolean isIconifiable()**	Selecciona u obtiene si el frame interno puede ser minimizado.
void setMaximizable(boolean) **boolean isMaximizable()**	Selecciona u obtiene si el usuario puede maximizar el frame interno.
void setResizable(boolean) **boolean isResizable()**	Selecciona u obtiene si el frame interno puede ser redimensionado.
void setText(String) **String getText()**	Selecciona u obtiene el título de la ventana.

void setWarningString(String) **String getWarningString()**	Selecciona u obtiene cualquier string de aviso mostrado en la ventana.

Usar el API de JDesktopPane

Método	Propósito
JDesktopPane()	Crea un ejemplar de **JDesktopPane**.
JInternalFrame[] getAllFrames()	Devuelve todos los objetos **JInternalFrame** que contiene el escritorio.
JInternalFrame[] getAllFramesInLayer(int)	Devuelve todos los objetos **JInternalFrame** que contiene el escritorio y que están en la capa especificada. Para más información puedes ver Cómo usar LayeredPane.

¿Cómo Usar LayeredPane?

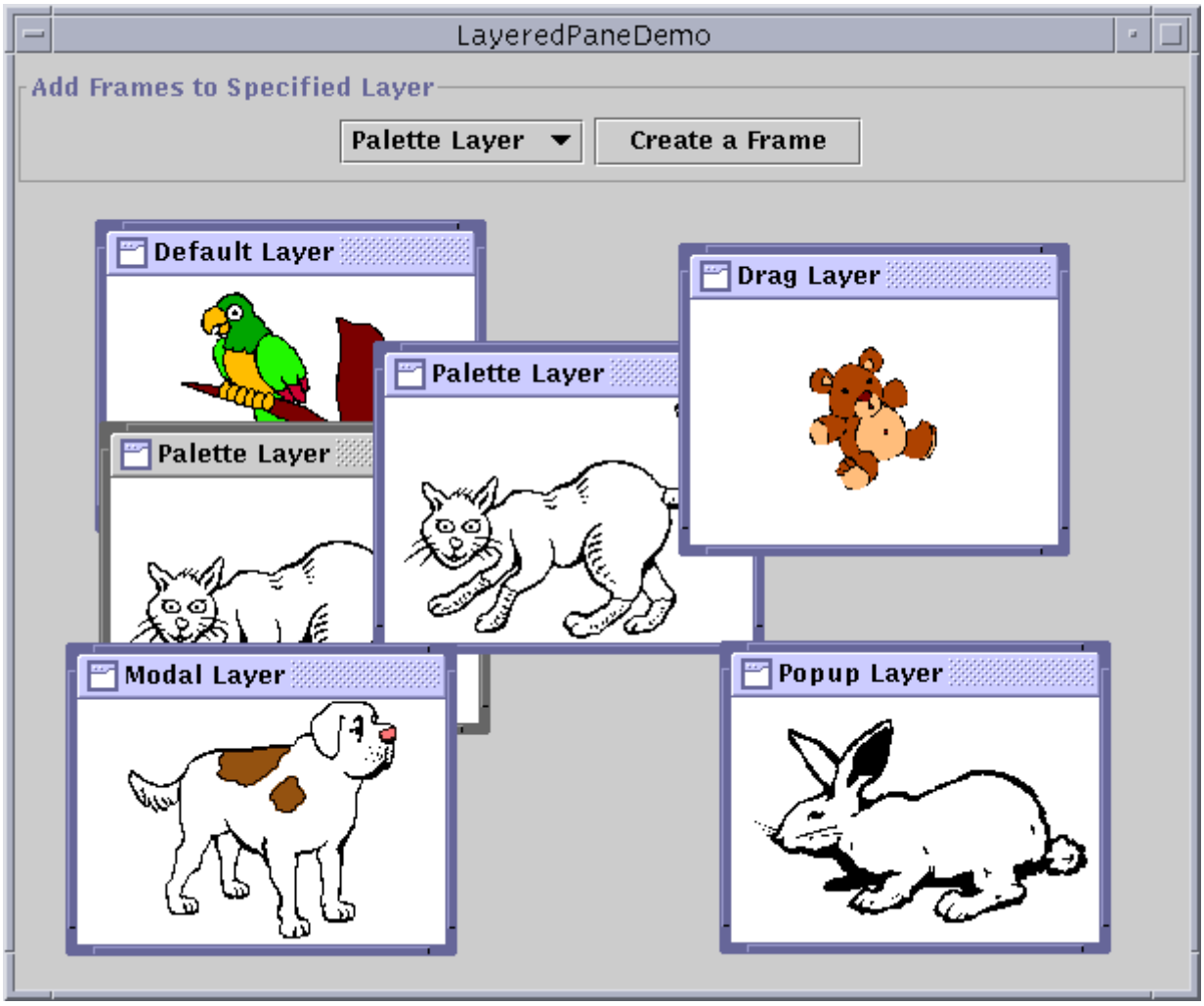
Un LayeredPane es un componente Swing que proporciona una tercera dimensión para posicionar componentes: **profundidad**, también conocida como **eje Z**. Cuando se añade un componente a un panel por capas, se especifica su profundidad.

Los frames con mayor profundidad siempre solapan los frames con menor profundidad y los frames con menor profundidad siempre están debajo de frames con mayor profundidad. Los frames con la misma profundidad pueden cambiar su posición. Por conveniencia, LayeredPane define varias layers (**capas**) dentro del rango posible de profundiades para funciones específicas. Por ejemplo, podemos poner un componente en la capa de mayor funcionalidad, la capa de arrastre, cuando se arrastan componentes.

Todo contenedor Swing que tiene un panel raíz -- como **JFrame**, **JApplet**, **JDialog**, y **JInternalFrame** -- automáticamente tiene un layeredpane, aunque la mayoría de los programas no los utilizan explícitamente. Podemos crear nuestro propio layeredpane y utilizarlo en cualquier lugar como un contenedor normal Swing.

Swing proporciona dos clases de paneles por capas. La primera, **JLayeredPane**, es la clase que utilizan los paneles raíz. La segunda, **JDesktopPane**, es una subclase de **JLayeredPane** que está especializada para contener frames internos. El ejemplo de esta sección utiliza un ejemplar de **JLayeredPane**. Para ver ejemplos de utilización de **JDesktopPane** puedes ir a Cómo usar Frames Internos.

Aquí podemos ver una imagen de una aplicación que utiliza paneles de capas para manejar **JInternalFrame**s en diferentes capas.



Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente es **LayeredPaneDemo.java**. También necesitarás cinco ficheros de imágenes: **Bird.gif**, **Cat.gif**, **Dog.gif**, **Rabbit.gif**, y **Pig.gif**.
2. Cuando arranca, el programa crea cinco frames internos. Para crear otro frame, selecciona una capa desde el combo box luego pulsa el botón **Create a Frame**.
3. Mueve los frames alrededor. Observa la relación de los frames en las diferentes capas y los frames de al misma capa.
4. Observa que se puede arrastrar un frame interno sobre los controles de la parte superior del programa. Los controles están en el panel de contenido del frame principal que está en una capa inferior a todas las capas disponibles desde el menú.

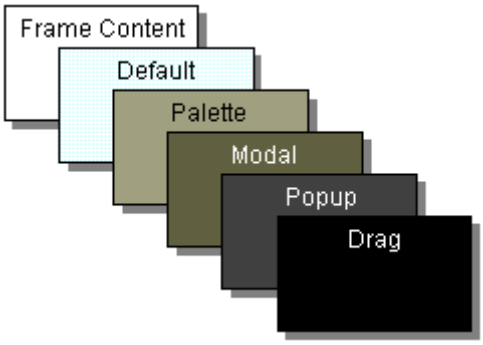
Las clases **JFrame**, **JApplet**, **JDialog**, y **JInternalFrame** proporcionan un método de conveniencia, **getLayeredPane**, para obtener el panel raíz del layeredpane.**LayeredPaneDemo.java** utiliza este método para obtener el layeredpane al que añadirle los frames internos.

```
public class LayeredPaneDemo extends JFrame ... {
    ...
    public LayeredPaneDemo() {
        ...
        layeredPane = getLayeredPane();
    ...
```

Aquí puedes ver el código que crea y añade frames internos al layeredpane.

```
private void addNewInternalFrame(int index) {
    JInternalFrame newFrame = new JInternalFrame();
    ...
    numFrames++;
    newFrame.setBounds(30*(numFrames%10), 30*(numFrames%10)+55, 200, 160);
    ...
    Integer layer = layerValues[index];
    layeredPane.add(newFrame, layer);
    try { newFrame.setSelected(true); }
    catch (java.beans.PropertyVetoException e2) {}
}
```

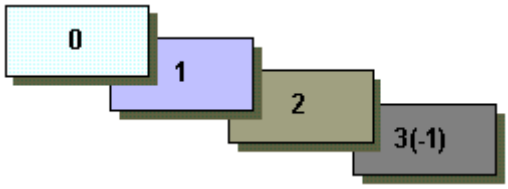
Las líneas en negrita muestran dónde se añade al frame al layeredpane. El método **add** utilizado en este programa toma dos argumentos. El primero es el componente a añadir; el segundo es un **Integer** indicando la profundidad donde poner el componente. El valor puede ser cualquier **Integer**. Sin embargo, la mayoría de los programas utilizarán uno de aquellos definidos por la clase **JLayeredPane**.



Nombre de Capa	Valor	Descripción
FRAME_CONTENT_LAYER	**new Integer(-30000)**	Esta capa es utiliza pra posicionar el panel de contenido del frame y la barra de menú. La mayoría de los programas no la utilizarán.
DEFAULT_LAYER	**new Integer(0)**	La mayoría de los componentes van en esta capa.
PALETTE_LAYER	**new Integer(100)**	Esta capa es útil para paletas y barras de herramientas flotantes.
MODAL_LAYER	**new Integer(200)**	Los diálogos modales, como aquellos proporcionados por **JOptionPane**, pertenecen a esta capa.
POPUP_LAYER	**new Integer(300)**	Los desplegables van en esta capa porque necesitan aparecer por encima de todo.
DRAG_LAYER	**new Integer(400)**	Un componente se mueve a esta capa cuando se arrasta. Se debe devolver el componente a su capa normal cuando se suelta.

La **posición** de un componente determina su relación con otros componentes de la misma capa. Al contrario que los números de capas, cuando más bajo sea el número de posición más alto estará el componente en su capa.

Se puede seleccionar la posición de un componente cuando se le añade al layeredpane proporcionando un tercer argumento al método **add**. Las posiciones se especifican con un **int** entre -1 y (N-1), donde N es el número de componentes en la capa. Utilizar -1 es lo mismo que utilizar N-1; indica la posición más inferior. Utilizar 0 especifica que el componente debería ir en la posición superior de su capa. Como se ve en la siguiente figura, con la excepción de -1, un número de posición menor indica una posición superior dentro de la capa.



Tanto la capa de un componente como su posición relativa dentro de la capa pueden cambiar. Para cambiar la capa de un componente normalmente se utiliza el método **setLayer**. Para cambiar la posición de un componente dentro de su capa, se puede utilizar los métodos **moveToBack** y **moveToFront** proporcionados por **JLayeredPane**.

Una Nota de Precaución: Cuando se añade un componente a una LayeredPane se especifica la capa con un **Integer**. Cuando se utiliza **setLayer** para cambiar al capa de un componente, se utiliza un **int**. Mira las tablas que hay abajo para comprobar los tipos de los argumentos y de los valores de retorno para otros métodos de esta clase que trabajan con capas.

El API LayeredPane

Las siguientes tablas listan los métodos y constructores más utilizados de la clase **JLayeredPane**. Otros métodos interesantes están definidos por las clases **JComponent** y **Component**.

El API para utilizar LayeredPane se divide en estas categorías.

Crear u Obtener un LayeredPane

Método	Propósito

JLayeredPane()	Crea un LayeredPane.
JLayeredPane getLayeredPane() (en **JApplet**, **JDialog**, **JFrame**, y **JInternalFrame**)	Obtiene el LayeredPAne en un applet, dialog, frame, o frame interno.

Situar Componentes en Capas

Método	Propósito
void add(Component, Integer) **void add(Component, Integer, int)**	Añade el componente especificado al layeredpane. El segundo argumento indica la capa. El tercer argumento, cuando existe, indica la posición del componente dentro de la capa.
void setLayer(Component, int) **void setLayer(Component, int, int)**	Cambia la capa del componente. El segundo argumento indica la capa, el tercer argumento, cuando existe, indica la posición del componente dentro de la capa.
int getLayer(Component) **int getLayer(JComponent)**	Obtiene la capa del componente especificado.
int getComponentCountInLayer(int)	Obtiene el número de componentes en la capa especificada. El valor devuelto por este método puede ser útil para calcular los valores de posición.
Component[] getComponentsInLayer(int)	Obtiene un array con todos los componentes en el capa especificada.
int highestLayer() **int lowestLayer()**	Calcula la capa más alta o más baja actualmente utilizadas.

Posicionar Componentes en una Capa

Método	Propósito
void setPosition(Component, int) **int getPosition(Component)**	Selecciona u obtiene la posición del componente especificado dentro de su capa.
void moveToFront(Component) **void moveToBack(Component)**	Mueve el componente especificado adelante o atrás en su capa.

¿Cómo Usar RootPane?

En general, no se crea directamente un objeto **JRootPane**. En su lugar, se obtiene un **JRootPane** (tanto si se quiere como si no!) cuando se ejemplariza un **JInternalFrame** o uno de los contenedores Swing de alto nivel -- **JApplet**, **JDialog**, **JFrame**, y **JWindow**.

La página Reglas Generales para Usar Componentes Swing explica lo básico sobre el uso de paneles raíz -- obtener el panel de contenido, seleccionar su controlador de distribución, y añadirle componentes Swing. Esta página explica más cosas sobe los paneles raíz, incluyendo los componentes que crean un panel raíz, y cómo poder utilizarlos.

Un panel raíz se divide en cuatro partes.

El Panel de Cristal
: Oculto, por defecto. Si se hace visible, es como si se pusiera una hoja de cristal sobre las otras partes del panel raiz. Es completamente transparente (a menos que hagamos que el método **paint** haga algo) e intercepta los eventos de entrada para el panel raíz. En la siguiente sección, veremos un ejemplo de utilización de un panel de cristal.

El panel de capas
: Sirve para posicionar sus contenidos, que consisten en el panel de contenido y la barra de menú opcional. También puede contener otros componentes en un orden Z especificado. Para más información puedes ver Cómo usar Layered Panes.

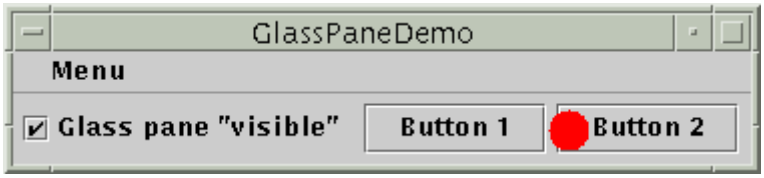
El Panel de Contenido
: El contenedor de los componentes visibles del panel raíz, excluyendo la barra de menú.

La barra de menú opcional
: El hogar para los menús del panel de contenido. Si el contenedor tiene una barra de menús, generalmente se utilizan los métodos **setMenuBar** o **setJMenuBar** del contenedor para poner la barra de menú en el lugar apropiado.

El Panel de Cristal

El panel de cristal es útil cuando queremos poder capturar eventos o dibujar sobre un área que ya contiene uno o más componentes. Por ejemplo, podemos desactivar los eventos de ratón para una región multi-componente haciendo que el panel de cristal intercepte los eventos. O podemos mostrar un cursor de espera sobte en panel raíz completo utilizando el panel de cristal.

Aquí podemos ver una imagen de una aplicación que demuestra las características del panel de cristal. Contiene un checkbox que permite seleccionar si el panel de cristal es "visible" -- se puede obtener eventos y dibujar sobre su propia parte de pantalla. Cuando un panel de cristal es visible, bloquea todas las entradas desde los componentes del panel de contenidos. También dibuja un punto rojo donde se detectó el último evento de pulsación de ratón.



Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente es **GlassPaneDemo.java**.
2. Pulsa le botón 1.

   La apariencia del botón cambia para indicar que ha sido pulsado.

3. Pulsa el checkbox para que el panel de cristal se vuelva "visible", y luego pulsa el botón 1, otra vez.

   El botón **no** detecta la pulsación del ratón porque el panel de cristal la ha interceptado. Cuando el panel de cristal detecta el evento, suena un pitido y dibuja un círculo rojo donde se pulsó.

4. Pulsa de nuevo sobre el checkbox para ocultar el panel de cristal.

   Cuando el panel raíz detecta un evento sobre el checkbox, lo reenvía al checkbox. De otro modo, el checkbox no podría responder a las puslaciones.

El siguiente código de **GlassPaneDemo.java** muestra y oculta el panel de cristal. Sucede que este programa para crear su propio panel de cristal lo selecciona utilizando el método **setGlassPane** de **JFrame**. Sin embargo, si un panel de cristal no hace ningún dibujo, el programa podría simplemente añadir oyentes al panel de cristal por defecto, como los devueltos por **getGlassPane**.

```
./where GlassPaneDemo's UI is initialized:
JCheckBox changeButton =
        new JCheckBox("Glass pane \"visible\"");
changeButton.setSelected(false);
changeButton.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent e) {
        myGlassPane.setVisible(e.getStateChange()
                               == ItemEvent.SELECTED);
    }
});
```

El siguiente fragmento de código implementa el manejo de eventos de raton para el panel de cristal. Si ocurre un evento de ratón sobre el checkbox o la barra de menús, entonces el panel de cristal redirecciona el evento para que el checkbox o el menú lo reciban. Para que el checkbox y el menú se comporten apropiadamente, también reciben todos los eventos drag que empiezan con una pulsación en el checkbox o en la barra de menú.

```
./In the implementation of the glass pane's mouse listener:
public void mouseMoved(MouseEvent e) {
    redispatchMouseEvent(e, false);
}

.* The mouseDragged, mouseClicked, mouseEntered,
    * mouseExited, and mousePressed methods have the same
    * implementation as mouseMoved*/...

public void mouseReleased(MouseEvent e) {
    redispatchMouseEvent(e, true);
    inDrag = false;
}

private void redispatchMouseEvent(MouseEvent e, boolean repaint) {
    boolean inButton = false;
    boolean inMenuBar = false;
    Point glassPanePoint = e.getPoint();
    Component component = null;
    Container container = contentPane;
    Point containerPoint = SwingUtilities.convertPoint(
                                    glassPane,
                                    glassPanePoint,
                                    contentPane);
    int eventID = e.getID();

    if (containerPoint.y < 0) {
        inMenuBar = true;
        //...set container and containerPoint accordingly...
        testForDrag(eventID);
    }

    component = SwingUtilities.getDeepestComponentAt(
                                    container,
```

```
                                    containerPoint.x,
                                    containerPoint.y);

    if (component.equals(liveButton)) {
        inButton = true;
        testForDrag(eventID);
    }

    if (inMenuBar || inButton || inDrag) {
        ./Redispatch the event to component...
    }

    if (repaint) {
        toolkit.beep();
        glassPane.setPoint(glassPanePoint);
        glassPane.repaint();
    }
}

private void testForDrag(int eventID) {
    if (eventID == MouseEvent.MOUSE_PRESSED) {
        inDrag = true;
    }
}
```

Aquí está el código que implementa el dibujo para el panel de cristal.

```
./where GlassPaneDemo's UI is initialized:
myGlassPane = new MyGlassPane(...);
frame.setGlassPane(myGlassPane);
...
/**
 * We have to provide our own glass pane so that it can paint.
 */
class MyGlassPane extends JComponent {
    Point point = null;

    public void paint(Graphics g) {
            if (point != null) {
                g.setColor(Color.red);
                g.fillOval(point.x - 10, point.y - 10, 20, 20);
            }
    }
    ...
}
```

El API de Root Pane

Las siguientes tablas listan el API para utilizar paneles raíz, paneles de cristal y paneles de contenido.

El API para utilizar otras partes del panel raíz se describe en .

- El API Layered Pane
- El API Menu

Usar un Panel Raíz

Método	Propósito
JRootPane getRootPane() (en **JApplet**, **JDialog**, **JFrame**, **JInternalFrame**, y **JWindow**)	Obtiene el panel raíz del applet, dialog, frame, internal frame, o window.
JRootPane SwingUtilities.getRootPane(Component)	Si el componente tiene un panel raíz, lo devuelve. Si no es así, devuelve el panel raíz (si existe) que contiene el componente.
JRootPane getRootPane() (en **JComponent**)	Invoca al método **SwingUtilitiesgetRootPane** sobre **JComponent**.
void setDefaultButton(JButton) **JButton getDefaultButton()**	Selecciona u obtiene qué botón (si existe) es el botón por defecto del panel raíz. Una acción específica del aspecto y comportamiento, como pulsar ENTER, hace que se realice la acción del botón.

Seleccionar u Obtener el Panel de Cristal

Método	Propósito
setGlassPane(Component) **Component getGlassPane()** (en **JApplet**, **JDialog**, **JFrame**, **JInternalFrame**, **JRootPane**, y **JWindow**)	Selecciona u obtiene elpanel de cristal.

Usar el Panel de Contenido

Método	Propósito
setContentPane(Container) **Container getContentPane()** (en **JApplet**, **JDialog**, **JFrame**, **JInternalFrame**, **JRootPane**, y **JWindow**)	Selecciona u obtiene el panel de contenido.

¿Cómo Usar Button?

Para crear un botón, se ejemplariza una de las muchas subclases de la clase **AbstractButton**. Esta sección explica el API básico que define la clase **AbstractButton** -- y lo que todos los botones Swing tienen en común. Como la clase **JButton** desciende de **AbstractButton** define un pequeño API público adicional, está página lo utiliza para ver cómo funcionan los botones.

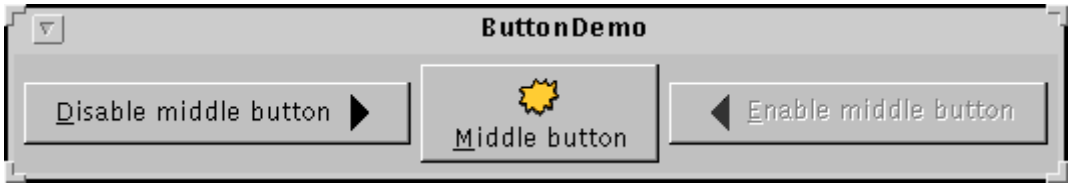
Nota: En Swing 1.0.2, los botones ignoraban sus mnemónicos (teclas aceleradoras). Este error se corrigió en Swing 1.0.3.

La siguiente tabla muestra las subclases de **AbstractButton** definidas en Swing que podemos utilizar.

Clase	Sumario	Dónde se Describe
JButton	Un botón común	En esta sección.
JCheckBox	Un checkbox típico	Cómo usar CheckBox
JRadioButton	Un botón de rádio de un grupo.	Cómo usar RadioButton
JMenuItem	Un ítem de un menú.	Cómo usar Menu
JToggleButton	Implementa la funcionalidad heredada de **JCheckBox** y **JRadioButton**.	En ningún lugar de este tutorial.

Nota: Si queremos juntar un grupo de botones dentro de una fila o una columna deberíamos chequear Cómo usar toolbar.

Aquí tienes una imagen de una aplicación que muestra tres botones.



Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente está en **ButtonDemo.java**.

   Puedes ver Empezar con Swing si necesitas ayuda.

2. Pulsa el botón izquierdo.

   Deshabalita el botón central (y a sí mismo, ya que no es necesario) y activa el botón derecho.

3. Pulsa el botón derecho.

   Activa los botones central e izquierdo y se desactiva a a sí mismo.

Como muestra el ejemplo **ButtonDemo**, un botón Swing puede mostrar tanto texto como una imagen. En **ButtonDemo**, cada botón tiene su texto en un sitio diferente. La letra subrayada de cada texto de botón muestra el **mnemónico** -- la tecla alternativa -- para cada botón.

Cuando un botón se desactiva, le aspecto y comportamiento genera automáticamente la apariencia de botón desactivado. Sin embargo, podríamos proporcionar una imagen para que substituya la imagen normal. Por ejemplo, podría proporcionar versiones en gris de las imagenes utilizadas en los botones de la derecha y de la izquierda.

Cómo se implementa el manejo de eventos depende del tipo de botón utilizado y de cómo se utiliza. Generalmente, implementamos un action listener, que es notificado cada vez que el usuario pulsa el botón, Para un checkbox normalmente se utiliza un item listener, que es notificado cuando el checkbox es seleccionado o deseleccionado.

Abajo podemos ver el código de**ButtonDemo.java** que crea los botones del ejemplo anterior y reacciona a las pulsaciones de los botones. El código en negrita es el código que permanecería si los botones no tuvieran imágenes.

```
//In initialization code:
    ImageIcon leftButtonIcon = new ImageIcon("images/LEFT.gif");
    ImageIcon middleButtonIcon = new ImageIcon("images/middle.gif");
    ImageIcon LEFTButtonIcon = new ImageIcon("images/left.gif");

    b1 = new JButton("Disable middle button", leftButtonIcon);
    b1.setVerticalTextPosition(AbstractButton.CENTER);
    b1.setHorizontalTextPosition(AbstractButton.LEFT);
    b1.setMnemonic('d');
    b1.setActionCommand("disable");

    b2 = new JButton("Middle button", middleButtonIcon);
    b2.setVerticalTextPosition(AbstractButton.BOTTOM);
    b2.setHorizontalTextPosition(AbstractButton.CENTER);
    b2.setMnemonic('m');

    b3 = new JButton("Enable middle button", LEFTButtonIcon);
    //Use the default text position of CENTER, LEFT.
    b3.setMnemonic('e');
    b3.setActionCommand("enable");
    b3.setEnabled(false);

    //Listen for actions on buttons 1 and 3.
    b1.addActionListener(this);
    b3.addActionListener(this);
    . . .
}

public void actionPerformed(java.awt.event.ActionEvent e) {
    if (e.getActionCommand().equals("disable")) {
        b2.setEnabled(false);
        b1.setEnabled(false);
        b3.setEnabled(true);
    } else {
        b2.setEnabled(true);
        b1.setEnabled(true);
        b3.setEnabled(false);
    }
}
```

El API Button

Las siguientes tablas listan los métodos y constructores más utilizados de **AbstractButton** y **JButton**. Podemos ver la mayoría de este API jugando con el panel de botones del ejemplo **SwingSet** que forma parte de la versión Swing.

El API para utilizar botones se divide en tres categorias.

Seleccionar u Obtener el Contenido de un Botón

Método o Constructor	Propósito
JButton(String, Icon) **JButton(String)** **JButton(Icon)** **JButton()**	Crea un ejemplar de **JButton**, lo inicializa para tener el texto/imagen especificado.
void setText(String) **String getText()**	Selecciona u obtiene el texto mostrado en el botón.
void setIcon(Icon) **Icon getIcon()**	Selecciona u obtiene la imagen mostrada por el botón cuando no está seleccionado o pulsado.
void setDisabledIcon(Icon) **Icon getDisabledIcon()**	Selecciona u obtiene la imagen mostrada por el botón cuando está desactivado. Si no se especifica una imagen, el aspecto y comportamiento crea una por defecto.
void setPressedIcon(Icon) **Icon getPressedIcon()**	Seleccion u obtiene la imagen mostrada por el botón cuando está puslado.
void setSelectedIcon(Icon) **Icon getSelectedIcon()** **void**	Selecciona u obtiene la imagen mostrada por el botón cuando está seleccionado. Si no se especifica una imagen de botón desactivado seleccionado, el aspecto y comportamiento crea una manipulando la imagen de seleccionado.

setDisabledSelectedIcon(Icon) **Icon getDisabledSelectedIcon()**	
setRolloverEnabled(boolean) **boolean getRolloverEnabled()** **void setRolloverIcon(Icon)** **Icon getRolloverIcon()** **void setRolloverSelectedIcon(Icon)** **Icon getRolloverSelectedIcon()**	Utiliza **setRolloverEnabled(true)** y **setRolloverIcon(someIcon)** para hacer que el botón muestre el icono especificado cuando el cursor pasa sobre él.

Ajuste Fino de la Apariencia del Botón

Método o constructor	Propósito
void setHorizontalAlignment(int) **void setVerticalAlignment(int)** **int getHorizontalAlignment()** **int getVerticalAlignment()**	Selecciona u obtiene dónde debe situarse el contenido del botón. La clase **AbstractButton** permite uno de los siguientes valores para alineamineto horizontal: **LEFT**, **CENTER** (por defecto), y **LEFT**. Para alineamiento vertical: **TOP**, **CENTER** (por defecto), y **BOTTOM**.
void setHorizontalTextPosition(int) **void setVerticalTextPosition(int)** **int getHorizontalTextPosition()** **int getVerticalTextPosition()**	Selecciona u obtiene dónde debería situarse el texto del botón con respecto a la imagen. La clase **AbstractButton** permite uno de los siguientes valores para alineamineto horizontal: **LEFT**, **CENTER** (por defecto), y **LEFT**. Para alineamiento vertical: **TOP**, **CENTER** (por defecto), y **BOTTOM**.
void setMargin(Insets) **Insets getMargin()**	Selecciona u obtiene el número de pixels entre el borde del botón y sus contenidos.
void setFocusPainted(boolean) **boolean isFocusPainted()**	Selecciona u obtiene si el botón debería parecer diferente si obtiene el foco.
void setBorderPainted(boolean) **boolean isBorderPainted()**	Selecciona u obtiene si el borde del botón debería dibujarse.

Implementar la Funcionalidad del Botón

Método o Constructor	Propósito
void setMnemonic(char) **char getMnemonic()**	Selecciona la tecla alternativa para pulsar el botón.
void setActionCommand(String) **String getActionCommand(void)**	Selecciona u obtiene el nombre de la acción realizada por el botón.
void addActionListener(ActionListener)	Añade o elimina un objeto que escucha eventos action disparados por el botón.

ActionListener removeActionListener()	
void addItemListener(ItemListener) **ItemListener removeItemListener()**	Añade o elimina un objeto que escucha eventos items disparados por el botón.
void setSelected(boolean) **boolean isSelected()**	Selecciona u obtiene si el botón está seleccionado. Tiene sentido sólo en botones que tienen un estado on/off, como los checkbox.
void doClick() **void doClick(int)**	Programáticamente realiza un "click". El argumento opcional especifica el tiempo (en milisegundos) que el botón debería estar pulsado.

¿Cómo Usar CheckBox?

La versión Swing soporta botones checkbox con la clase **JCheckBox**. Swing también soporta checkboxes en menus, utilizando la clase **JCheckBoxMenuItem**. Como **JCheckBox** y **JCheckBoxMenuItem** descienden de **AbstractButton**, los checkboxes de Swing tienen todas las características de un botón normal como se explicó en Cómo usar Buttons. Por ejemplo, podemos especificar imágenes para ser utilizadas en los checkboxes.

Los Checkboxes son similares a los botones de rádio, pero su modelo de selección es diferente, por convención. Cualquier número de checkboxes en un grupo -- ninguno, alguno o todos -- pueden ser seleccionados. Por otro lado, en un grupo de botones de rádio, sólo puede haber uno seleccionado.

Nota: En Swing 1.0.2, los botones ignoran sus mnemónicos (teclas aceleradoras). Este bug se corrigió en Swing 1.0.3.

Aquí podemos ver una imagen de una aplicación que utiliza cuatro checkboxes para personalizar una caricatura.

Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente es **CheckBoxDemo.java**. También necesitarás los 16 ficheros de imagenes del directorio **example-swing/images** que empiezan con "**geek**".
2. Pulsa el botón Chin o pulsa Alt-C.

   El checkbox Chin se desactiva, y la barbilla desaparecerá de la imagen. Los otros Checkboxes permanencen seleccionados. Esta aplicación tiene un oyente de ítem que escucha todos los checkboxes. Cada vez que el oyente de ítem recibe un evento, la aplicación carga una nueva imagen y refleja el estado actual de los checkboxes.

Un Checkbox genera un evento ítem y un evento action por cada pulsación. Normalmente, solo escucharemos los eventos de ítem, ya que nos permiten determinar si el click selecciona o desactiva el checkbox. Abajo puedes ver el código de **CheckBoxDemo.java** que crea los checkboxes del ejemplo anterior y reacciona ante las pulsaciones.

```
//In initialization code:
    chinButton = new JCheckBox("Chin");
    chinButton.setMnemonic('c');
    chinButton.setSelected(true);

    glassesButton = new JCheckBox("Glasses");
    glassesButton.setMnemonic('g');
    glassesButton.setSelected(true);

    hairButton = new JCheckBox("Hair");
    hairButton.setMnemonic('h');
    hairButton.setSelected(true);

    teethButton = new JCheckBox("Teeth");
    teethButton.setMnemonic('t');
```

```
    teethButton.setSelected(true);

    // Register a listener for the check boxes.
    CheckBoxListener myListener = new CheckBoxListener();
    chinButton.addItemListener(myListener);
    glassesButton.addItemListener(myListener);
    hairButton.addItemListener(myListener);
    teethButton.addItemListener(myListener);
...
class CheckBoxListener implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        ...
              Object source = e.getItemSelectable();

        if (source == chinButton) {
            //...make a note of it...
        } else if (source == glassesButton) {
            //...make a note of it...
        } else if (source == hairButton) {
            //...make a note of it...
        } else if (source == teethButton) {
            //...make a note of it...
        }

        if (e.getStateChange() == ItemEvent.DESELECTED)
            //...make a note of it...
        picture.setIcon(/* new icon */);
        ...
    }
}
```

El API CheckBox

Puedes ver El API Button para información sobre el API de **AbstractButton** del que descienden **JCheckBox** y **JCheckBoxMenuItem**. Los métodos de **AbstractButton** que son más usados son **setMnemonic**, **addItemListener**, **setSelected**, y **isSelected**. El único API definido por **JCheckBox** y **JCheckBoxMenuItem** que utilizaremos son los constructores.

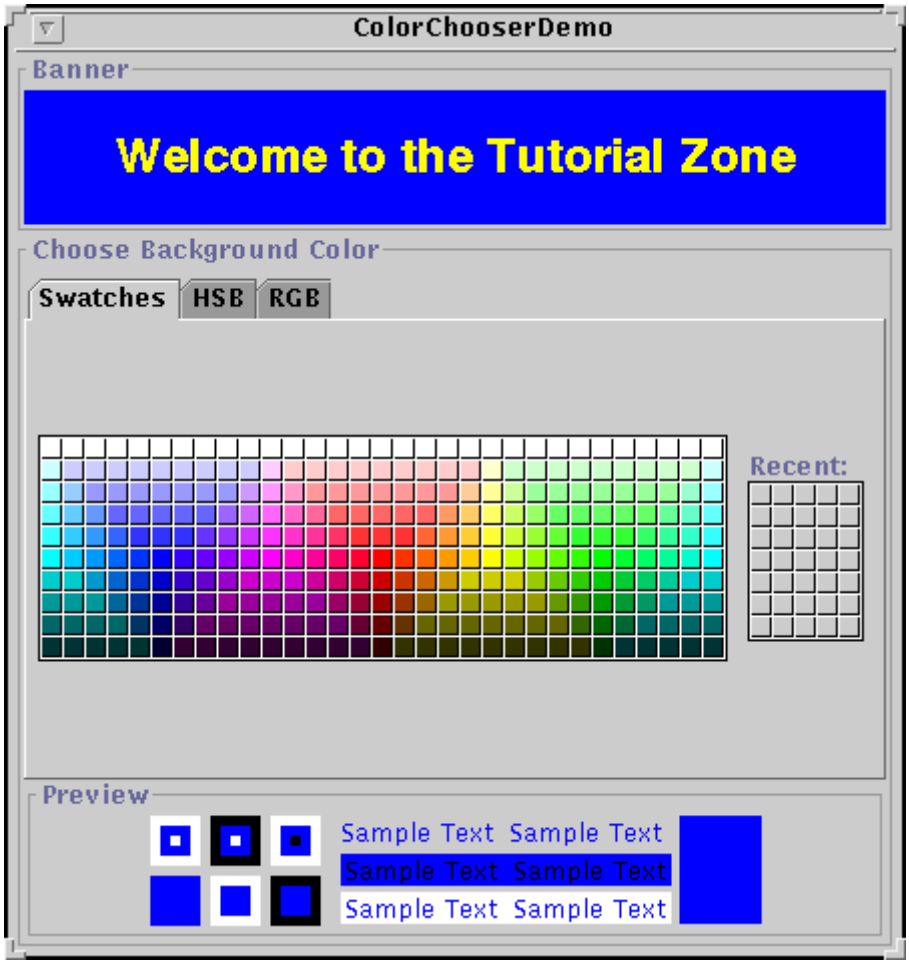
Constructores de CheckBox

Constructor	Propósito
JCheckBox(String) **JCheckBox(String, boolean)** **JCheckBox(Icon)** **JCheckBox(Icon, boolean)** **JCheckBox(String, Icon)** **JCheckBox(String, Icon, boolean)** **JCheckBox()**	Crea un ejemplar de **JCheckBox**. El argumento string específica el texto, si existe, que el checkbox debería mostrar. De forma similar, el argumento **Icon** específica la imagen que debería utilizarse en vez de la imagen por defecto del aspecto y comportamiento. Especificando el argumento booleano como **true** se inicializa el checkbox como seleccionado. Si el argumento booleano no existe o es **false**, el checkbox estará inicialmente desactivado.
JCheckBoxMenuItem(String) **JCheckBoxMenuItem(String, boolean)** **JCheckBoxMenuItem(Icon)** **JCheckBoxMenuItem(String, Icon)** **JCheckBoxMenuItem(String, Icon, boolean)** **JCheckBoxMenuItem()**	Crea un ejemplar de **JCheckBoxMenuItem**. Los argumentos se interpretan de la misma forma que en los constructores de **JCheckBox**.

¿Cómo Usar ColorChooser?

Se puede utilizar la clase **JColorChooser** para proporcionar a los usuarios una paleta para elegir colores. Un selector de color es un componente que se puede situar en cualquier lugar dentro del GUI de un programa. El API de **JColorChooser** también hace sencillo desplegar un diálogo (modal o no) que contiene un selector de color.

Aquí tienes una imagen de una aplicación que utiliza un selector de color para seleccionar el color de fondo de un banner.



El código fuente principal del programa está en **ColorChooserDemo.java**. También necesitarás **Banner.java**.

El selector de color consiste en cualquier cosa que hay dentro del borde llamado **Choose Background Color**. Contiene dos partes, un panel con pestañas y un panel de previsionado. Las tres pestañas delprimero seleccionan un panel selector diferente. El preview panel mustra el color seleccionado actualmente.

Aquí podemos ver el código del ejemplo que crea un ejemplar de **JColorChooser** y lo añade a la ventana.

```
Banner banner = new Banner();
...
final JColorChooser colorChooser = new JColorChooser(banner.getColor());
...
getContentPane().add(colorChooser, BorderLayout.CENTER);
```

El constructor utilizado para crear el selector de color toma un argumento **Color**, que especifica el color seleccionaod inicialmente.

Un selector de color utiliza un ejemplar de **ColorSelectionModel** para contener y manejar la selección actual. Este dispara un evento "change" si el usuario cambia el color del selector. El programa de ejemplo registra un oyente de "change" con el **ColorSelectionModel** para poder actualizar el banner de la parte superior de la ventana.

Aquí podemos ver el código que registra e implementa el oyente de "change".

```
colorChooser.getSelectionModel().addChangeListener(
    new ChangeListener() {
                public void stateChanged(ChangeEvent e) {
                    Color newColor = colorChooser.getColor();
                    banner.setColor(newColor);
                }
    }
);
```

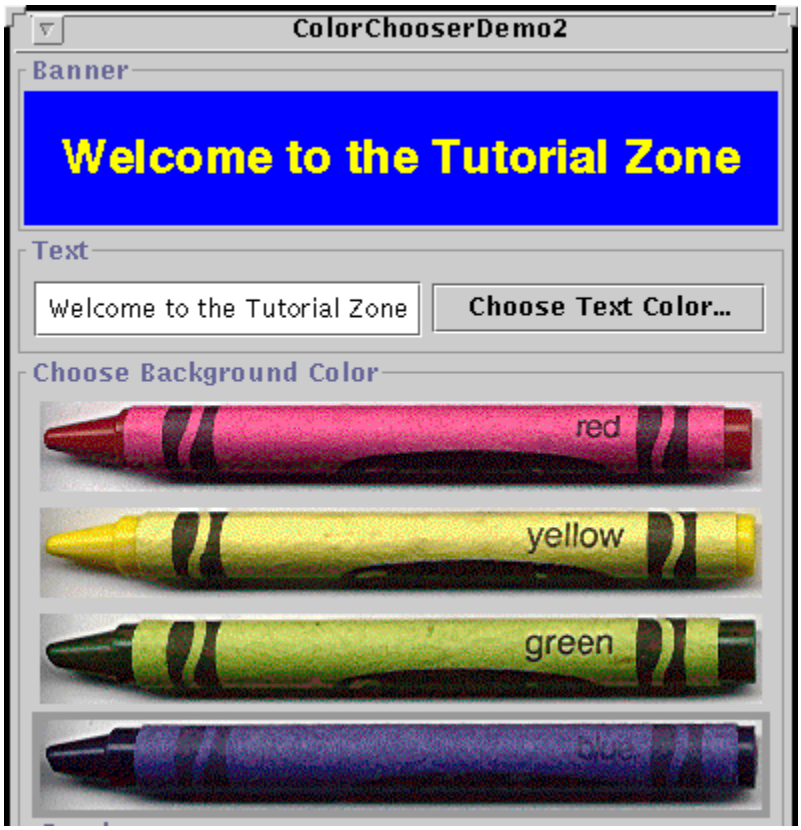
El oyente de Change obtiene el color seleccionado actualmente desde el selector de color y lo utiliza para seleccionar el color de fondo del banner. Puedes ver Cómo escribir un oyente de Change para información general sobre los evento de "Change" y sus oyentes.

Un selector de color básico , como el utilizado en el programa de ejemplo, es suficiente para muchos programas. Sin embargo, el API ColorChooser permite personalizar un selector de oclor proporcionando un panel de previsionado de nuestro propio diseño, añadiéndolo nuestros propios paneles, o eliminando los panles del selección existentes en el selector de color. Además, la clase ColorChooser proporciona dos métodos que hacen muy sencillo el uso de un selector de color dentro de un diálogo.

ColorChooserDemo: Toma 2

Ahora veremos**ColorChooserDemo2**, una versión modificada del programa anterior que utiliza más API **JColorChooser**.

Aquí puedes ver una imagen de **ColorChooserDemo2**.



Además del fichero fuente principal,**ColorChooserDemo2.java**, necesitaremos **CrayonPanel.java**, **Banner.java**, y las cuatro imágenes de los lápizes (**red.gif**, **yellow.gif**, **green.gif**, y **blue.gif**) para ejecutar este programa.

Este programa añade un GUI para cambiar el texto del banner y para seleccionar el color del texto. Podemos llamar al selector de color para el texto pulsando sobre el botón **Choose Text Color...**, que trae un diálogo selector de oclor.

Además, este programa personaliza el selector del color de fondo del banner de estas formas.

- Elimina el panel de previsionado
- Elimina todos los paneles selectores por defecto
- Añade un panel selector personalizado

Mostrar un Selector de Color en un Diálogo

La clase **JColorChooser** proporciona dos métodos de clase que hace sencillo el uso de un selector de color en un dialog modal. El nuevo programa utiliza uno de estos métodos, **showDialog**, para mostar el selector de color del texto cuando el usuario pulsa el botón **Choose Text Color...**. Aquí puedes ver la línea de código del ejemplo que trae el diálogo modal del selector de color del texto.

```
Color newColor = JColorChooser.showDialog(ColorChooserDemo.this,
                                          "Choose Text Color",
                                          banner.getTextColor());
```

El diálogo desaparece bajo tres condiciones: el usuario elige un color y pulsa el botón **OK**, el usuario cancela la operación con el botón **Cancel**, el usuario cierra el diálogo. Si el usuario elige un color, el metodo **showDialog** devuelve el nuevo color. Si el usuario cancela la operación o cierra el diálogo, el método devuelve **null**. Aquí podemos ver el código del ejemplo que actualiza el color del texto de acuerdo al valor devuelto por **showDialog**.

```
if (newColor != null) {
    banner.setTextColor(newColor);
}
```

JColorChooser proporciona otro método que nos ayuda a utiliza un selector de color en un diálogo. El método **createDialog** crea y devuelve un diálogo, permitiendo especificar los oyentes de action para los botones **OK** y **Cancel** de la ventana de diálogo. Se utiliza el método **show** de **JDialog** para mostrar el diálogo creado con este método.

Reemplazar o Eliminar el Panel de Previsionado

Nota: Las versiones Swing 1.1 Beta 2 y anteriores contienen un bug por el que el método **setPreviewPanel** lanza una **NullPointerException**. Por eso no hemos podido probar y verificar esta sección.

Por defecto, el selector de color muestra un panel de previsionado.

El programa de ejemplo elimina este panel con esta línea de código.

```
colorChooser.setPreviewPanel(new JPanel());
```

Efectivamente, esto elimina el panel de previsionado porque un **JPanel** plano no tiene tamaño ni vista por defecto.

Para proporcionar un panel de previsionado personalizado, también podemos utilizar **setPreviewPanel**. El componente que pasemos dentro del método debería descender de **JComponent**, especificar un tamaño razonable, y proporcionar una vista personalizada del color actual (obtenida con el método **getColor** de **Component getColor**). De hecho, después de añadir un método **getPreferredSize** a **Banner**, podríamos utilizar un ejemplar de **Banner** como panel de previsionado.

Crear un Panel Selector Personalizado

El selector de color por defecto proporciona tres paneles selectores.

- Swatches -- para elegir un color de una selección.
- HSB -- para elegir un color usando el modelo Color-Saturación-Brillo.
- RGB -- para elegir un color usando el modelo Rojo-Verde-Azul.

Se pueden extender los selectores por defecto añadiendo paneles selectores de nuestro propio diseño o podemos limitarlo eliminando paneles selectores.

ColorChooserDemo2 hace las dos cosas: elimina todos los paneles por defecto en el selector de color y añade el suyo propio.

Aquí podemos ver el código que elimina los paneles selectores por defecto.

```
//Remove the default chooser panels
AbstractColorChooserPanel panels[] = colorChooser.getChooserPanels();
for (int i = 0; i < panels.length; i ++) {
    colorChooser.removeChooserPanel(panels[i]);
}
```

El código es correcto, utiliza **getChooserPanels** para obtener un array conteniendo todos los paneles selectores en el selector de color. Luego, el código hace un bucle a través del array y elimina cada uno de ellos llamando a **removeChooserPanel**.

El programa utiliza el siguiente código para añadir un ejemplar de **CrayonPanel** como un panel selector del selector de color.

```
colorChooser.addChooserPanel(new CrayonPanel());
```

Nota: Las versiones Swing 1.1 Beta 2 y anteriores contienen un bug que hace que el método **addChooserPanel** genera una **NullPointerException**. Puedes ver el código de See **ColorChooserDemo2.java** para un atajo recomendado.

CrayonPanel es una subclase de **AbstractColorChooserPanel** y sobreescribe los cinco métodos abstractos definidos en su superclase.

void updateChooser()
Este método es llamado cuando el se muestra el panel selector. La implementación de este método en el ejemplo selecciona el botón que representa el color seleccionado actualmente.

```
public void updateChooser() {
    Color color = getColorFromModel();
    if (color.equals(Color.red)) {
        redCrayon.setSelected(true);
    } else if (color.equals(Color.yellow)) {
        yellowCrayon.setSelected(true);
    } else if (color.equals(Color.green)) {
        greenCrayon.setSelected(true);
    } else if (color.equals(Color.blue)) {
        blueCrayon.setSelected(true);
    }
}
```

void buildChooser()
Crea el GUI que comprende el panel selector. El ejemplo crea cuatro botones -- uno para cada lápiz -- y los añade al panel selector.

String getDisplayName()
Devuelve el nombre mostrado por el panel selector. El nombre es utilizado para la pestaña del panel selector. Aquí tenemos el método **getDisplayName** del ejemplo: method.

```
public String getDisplayName() {
    return "Crayons";
}
```

Icon getSmallDisplayIcon()
Devuelve un pequeño icono que representa este panel selector. El icono es utilizad por la pestaña del panel selector. La implemtación de este método devuelve **null**.

Icon getLargeDisplayIcon()
Devuelve un icono que representa este panel selector. El icono es utilizad por la pestaña del panel selector. La implemtación de este método devuelve **null**.

Además de estos métodos sobreescritos, **CrayonPanel** tiene un constructor que sólo llama a **super()**.

El API ColorChooser

Las siguientes tablas listas el métodos y constructores más utilizados de **JColorChooser**.

El API para utilizar selectores de colores se divide en estas categorías.

Crear y Mostrar un ColorChooser

Método	Propósito
JColorChooser() **JColorChooser(Color)** **JColorChooser(ColorSelectionModel)**	Crea un selector de color. El constructor por defecto crea un selector de color con el color inicial blanco. Se utiliza el segundo constructor para especificar un color inicial diferente. El argumento, **ColorSelectionModel**, cuando está presente, proporciona un selector de color con un modelo de selección de color.
Color showDialog(Component, String, Color)	Crea y muestra un selector de color en un diálogo modal. El argumento **Component** es el padre del diálogo, el argumento **String** específica el título del diálogo, y el argumento **Color** el color seleccionado inicialmente.
JDialog createDialog(Component, String, boolean, JColorChooser, ActionListener, ActionListener)	Crea un diálogo para el selector de color especificado. Como en **showDialog**, el argumento **Component** es el padre del diálogo y el argumento **String** específica el título del diálogo. El argumento **boolean** especifica si el diálogo es modal. El primer **ActionListener** es para el botón **OK**, y el segundo para el botón **Cancel**.

Personalizar un ColorChooser

Método	Propósito
void setPreviewPanel(JComponent) **JComponent getPreviewPanel()**	Selecciona u obtiene el componente utilizado para previsionar la selección de color. Para eleminar el panel de previsionado, se utiliza **new JPanel()**. Para especificar el panel de previsionado por defecto, se utiliza **null**.
void setChooserPanels(AbstractColorChooserPanel[]) **AbstractColorChooserPanel[] getChooserPanels()**	Selecciona u obtiene los paneles selectores en el selector de color.
void addChooserPanel(AbstractColorChooserPanel) **AbstractColorChooserPanel removeChooserPanel(AbstractColorChooserPanel)**	Añade o elimina un panel selector en el selector de color.

Seleccionar u Obtener la Selección Actual

Método	Propósito
void setColor(Color) **void setColor(int, int, int)** **void setColor(int)** **Color getColor()**	Selecciona u obtiene el color seleccionado actualmente. Los tres argumentos enteros de **setColor** especifican los valores RGB del color. El único argumento entero de **setColor** también específica el color en RGB. Los 8 bits de mayor peso especifican el rojo, los 8 bits siguientes el verde, y los 8 bits de menor peso el azul.

void setSelectionModel(ColorSelectionModel) **ColorSelectionModel getSelectionModel()**	Selecciona u obtiene el modelo de selección para el selector de color. Este objeto contiene la selección actual y dispara eventos change para los oyentes registrados si la selección cambia.

¿Cómo Usar ComboBox?

Con un **JComboBox** editable, una lista desplegable, y un text field, el usuario puede teclear un valor o elegirlo desde una lista. Un ComboBox editable ahorra tiempo de entrada proporcionando atajos para los valores más comunmente introducidos.

Un ComboBox no editable desactiva el tecleo pero aún así permite al usuario seleccionar un valor desde una lista. Esto proporciona un espacio altenartivo a un grupo de radio buttons o una list.

Aquí puedes ver una imagen de una aplicación que utiliza un ComboBox editable para introducir un patrón con el que formatear fechas.

Intenta esto:

1. Compila y ejecuta el ejemplo. **ComboBoxDemo.java**.
2. Introduce un nuevo patrón eligiendo uno de la lista desplegable. El programa reformatea la fecha y hora actuales.
3. Introduce un nuevo patrón tecleándolo y pulsando **return**. De nuevo el programa reformatea la fecha y hora actuales.

Abajo podemos ver el código de **ComboBoxDemo.java** que crea y configura el ComboBox.

```
String[] patternExamples = {
                "dd MMMMM yyyy",
                "dd.MM.yy",
                "MM/dd/yy",
                "yyyy.MM.dd G 'at' hh:mm:ss z",
                "EEE, MMM d, ''yy",
                "h:mm a",
                "H:mm:ss:SSS",
                "K:mm a,z",
                "yyyy.MMMMM.dd GGG hh:mm aaa"
};

currentPattern = patternExamples[0];
. . .
JComboBox patternList = new JComboBox(patternExamples);
patternList.setEditable(true);
patternList.setSelectedIndex(0);
patternList.setAlignmentX(Component.LEFT_ALIGNMENT);
PatternListener patternListener = new PatternListener();
patternList.addActionListener(patternListener);
```

Este programa proporciona los valores para la lista desplegable del ComboBox con un array de strings. Sin embargo, los valores de la lista pueden ser cualquier **Object**, en cuyo caso el método **toString** de la clase **Object** proporciona el texto a mostrar. Para poner una imagen u otro valor que no sea texto en una lista ComboBox, sólo debemos proporcionar un celda personalizada renderizada con **setRenderer**.

Observa que el código activa explícitamente la edición para que el usuario pueda teclear valores. Esto es necesario porque, por defecto, un ComboBox no es editable. Este ejemplo particular permite editar el ComboBox porque su lista no proporciona todos los patrones de formateo de fechas posibles.

El código también registra un oyente de action con el ComboBox. Cuando un usuario selecciona un ítem del ComboBox, se llama a este método.

```
public void actionPerformed(ActionEvent e) {
    JComboBox cb = (JComboBox)e.getSource();
    String newSelection = (String)cb.getSelectedItem();
    currentPattern = newSelection;
    reformat();
}
```

El método llama a **getSelectedItem** para obtener el nuevo patrón elegido por el usuario, y utilizarlo para reformatear la fecha y la hora actuales.

Cuidado: Un ComboBox es un componente compuesto: comprende un botón, un menú desplegable, y cuando es editable, un campo de texto. El ComboBox dispara eventos de alto nivel, como eventos action. Sus componentes disparan eventos de bajo nivel como mouse, key y eventos de foco. Normalmente los componentes compuestos como el ComboBox deberían proporcionar oyentes para los eventos de alto nivel, porque los eventos de bajo nivel y los subcomponentes que los disparan son dependientes del sistema.

Utilizar un ComboBox no Editable

Aquí podemos ver una imagen de una aplicación que utiliza un ComboBox no editable para permitir al usuario elegir una imagen de mascota desde una lista.

Intenta esto:

1. Compila y ejecuta el programa. **ComboBoxDemo2.java**. También necesitarás 5 ficheros de imágenes. **Bird.gif**, **Cat.gif**, **Dog.gif**, **Rabbit.gif**, y **Pig.gif**.
2. Elige una mascota desde la lista desplegable para ver su dibujo.
3. Cómo usar Radio Buttons proporciona una versión de este programa, **RadioButtonDemo.java**, que utiliza un grupo de botones de rádio en lugar de un ComboBox. Compila y ejecuta el programa. Compara el código fuente y la operación de los dos programas.

Abajo podemos ver el código de ComboBoxDemo2.java que crea y configura el ComboBox.

```
...
//in the ComboBoxDemo2 constructor
String[] petStrings = { "Bird",
                        "Cat",
                        "Dog",
                        "Rabbit",
                        "Pig" };

// Crea el combobox,
// desactiva la edición
// y selecciona el primero
JComboBox petList = new JComboBox(petStrings);
petList.setSelectedIndex(0);
```

Este código es similar al código de **ComboBoxDemo**. Sin embargo, este programa deja el ComboBox no editable (por defecto).

Se utiliza un ComboBox no editable en lugar de un grupo de botones de radio para mostrar una o más elecciones en estas situaciones.

- Cuando el espacio es limitado
- Cuando el número de elecciones posibles es grande
- Cuando la lista se crea durante la ejecución

El API ComboBox

Las siguientes tablas listan los métodos y constructores más utilizados de **JComboBox**. Otros métodos a los que nos gustaría llamar están definidos por las clases **JComponent** y **Component**.

El API para utilizar ComboBox se divide en dos categorías.

Seleccionar u Obtener Ítems de la Lista del ComboBox

Método	Propósito
JComboBox(ComboBoxModel) **JComboBox(Object[])** **JComboBox(Vector)**	Crea un ComboBox con una lista predeterminada.
void addItem(Object) **void insertItemAt(Object, int)**	Añade o inserta un ítem en la lista.
Object getItemAt(int) **Object getSelectedItem()**	Obtiene un ítem de la lista.
void removeAllItems() **void removeItemAt(int)** **void removeItem(Object)**	Elimina uno o más ítems de la lista.
int getItemCount()	Obtiene el número de ítems de la lista.
void setModel(ComboBoxModel) **ComboBoxModel getModel()**	Selecciona u obtiene el modelo de datos que proporciona los ítems de la lista.

Personalizar la Configuración del ComboBox

Método	Propósito
void setEditabe(boolean) **boolean isEditable()**	Selecciona u Obtiene si el usuario puede teclear en el ComboBox.
void setRenderer(ListCellRenderer) **ListCellRenderer getRenderer()**	Selecciona u obtiene el objeto responsable para crear el ítem seleccionado en el ComboBox. Utilizado cuando el ComboBox no es editable.
void setEditor(ComboBoxEditor) **ComboBoxEditor getEditor()**	Selecciona u obtiene el objeto responsable del pintado y edición del ítem seleccionado en el ComboBox. Esto sólo se utiliza cuando el ComboBox es editable.

¿Cómo Usar FileChooser?

La clase **JFileChooser** proporciona un UI para elegir un fichero de una lista. Un selector de ficheros es un componente que podemos situar en cualquier lugar del GUI de nuestro programa. Sin embargo, normalmente los programas los muestran en diálogos modales porque las operaciones con ficheros son sensibles a los cambios dentro del programa. La clase **JFileChooser** hace sencillo traer un diálogo modal que contiene un selector de ficheros.

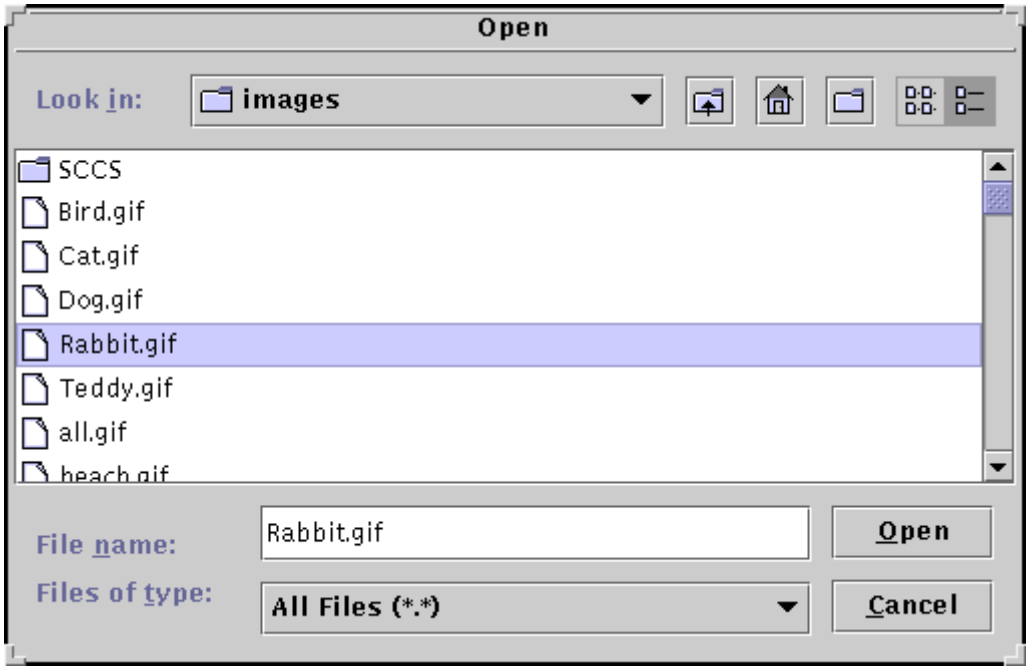
Los selectores de ficheros se utilizan comunmente para dos propósitos.

- Para presentar una lista de ficheros que pueden ser **abiertos** por la aplicación.
- Para permitir que el usuario seleccione o introduzca el nombre de un fichero a **grabar**.

Observa que el selector de ficheros ni abre ni graba ficheros. Presenta un GUI para elegir un fichero de una lista. El programa es responsable de hacer algo con el fichero, como abrirlo o grabarlo.

Como la mayoría de los programadores sólo quieren un selector para abrir o para grabar ficheros, la clase **JFileChooser** proporciona los métodos convenientes para mostrar estos tipos de selectores de ficheros en un diálogo. Nuestro primer ejemplo, **FileChooserDemo.java**, ilustra estos usos.

Cuando se pulsa el botón **Open** el programa trae un open file chooser. Cuando se pulsa el botón **Save** el programa trae un save file chooser. Aquí podemos ver una imagen de un selector de apertura de ficheros.



Aquí podemos ver el código que crea y muestra el selector de apertura de ficheros.

```
private JFileChooser filechooser = new JFileChooser();
...
int returnVal = filechooser.showOpenDialog(FileChooserDemo.this);
```

Por defecto, un selector de ficheros que no haya sido mostrado anteriormente muestra todos los ficheros en el directorio del usuario. Podemos especificarle un directorio inicial utilizando uno de los otros constructores de **JFileChooser**, o podemos seleccionar el directorio directamente con el método **setCurrentDirectory**.

El programa de ejemplo utiliza el mismo ejemplar de **JFileChooser** para mostrar el selector de grabar ficheros. Aquí tenemos el método **actionPerformed** para el oyente del botón **Save**.

```
private JFileChooser filechooser = new JFileChooser();
...
public void actionPerformed(ActionEvent e) {
    int returnVal = filechooser.showSaveDialog(FileChooserDemo.this);

    if (returnVal == JFileChooser.APPROVE_OPTION) {
               File file = filechooser.getSelectedFile();
               log.append("Saving: " + file.getName() + "." + newline);
    } else {
               log.append("Save command cancelled by user." + newline);
    }
}
```

Utilizando el mismo selector de ficheros para abrir y grabar ficheros, el programa consigue estos beneficios.

- El selector recuerda el directorio actual entre usos, por eso los diálogos de abrir y grabar comparten el mismo directorio actual.
- Sólo tenemos que personalizar un selector de ficheros, y nuestra personalización se aplicará a las dos versiones, la de apertura y la de grabación.

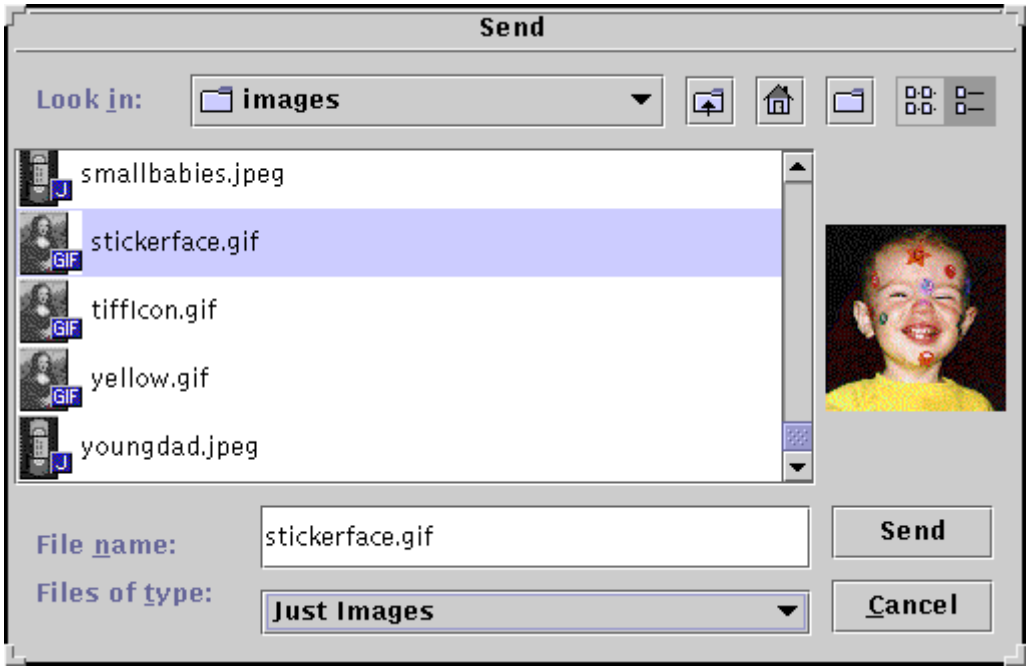
Cómo hemos podido ver en los fragmentos de código anteriores, los métodos **showXxxxDialog** devuelven un entero que indica si el usuario ha seleccionado un fichero. Podemos utilizar el valor de retorno para determinar si realizar o no la operación requerida.

Si el usuario elige un fichero, el código llama a **getSelectedFile** sobre el selector de ficheros para obtener un ejemplar de **File**, que representa el fichero elegido. El ejemplo obtiene el nombre del fichero y lo utiliza en un mensaje. Podemos utilizar otros métodos del objeto **File**, como **getPath** o **isDirectory**, para obtener información sobre él. También podemos llamar a otros métodos como **delete** y **rename** para cambiar el fichero de alguna forma. Por supuesto, podríamos leer o grabar el fichero utilizando una de las clases lectoras o escritoras proporcionadas por el JDK.

Si quieres crear un selector de ficheros para una tarea distinta de abrir o grabar, o si quieres personalizar el selector de ficheros, sigue leyendo. Estos excitantes tópicos se describen más abajo.

FileChooserDemo: Toma 2

Echemos un vistazo a **FileChooserDemo2.java**, una versión modificada del ejemplo anterior que utiliza más el API **JFileChooser**. Este ejemplo utiliza un selector de ficheros que ha sido personalizado de varias formas. Al igual que el ejemplo original, el usuario llama al selector de ficheros pulsando un botón. Aquí podmemos ver una imagen del selector de ficheros.



Necesitaremos estos ficheros fuente para ejecutar el ejemplo: **FileChooserDemo2.java**, **ImageFilter.java**, **ImageFileView.java**, y **ImagePreview.java**.

Cono se ve en la figura, este selector de ficheros ha sido personalizado para una tarea especial (enviar), proporciona un filtro de ficheros seleccionable, utiliza un visor especial de ficheros para ficheros de imágenes, y tiene un accesorio de visualización que muestra una versión reducida del fichero de imagen seleccionado.

El resto de esta página muestra el código que crea y personaliza este selector de ficheros.

Usar un Selector de Ficheros para una Tarea Personalizada

Como hemos visto, **JFileChooser** proporciona un método para mostrar un selector de apartura de ficheros y otro método para mostrar un selector para grabar ficheros. En el aspecto y comportamiento Metal, la única diferencia entre estos dos selectores es el título de la ventana del díalogo y la etiqueta del botón "accept".

La clase tiene un tercer método , **showDialog**, para mostrar un selector de ficheros para una tarea personalizada. Aquí podemos ver el código de **FileChooserDemo2** que muestra el diálogo selector de ficheros para la tarea **Send**.

```
JFileChooser filechooser = new JFileChooser();

int returnVal = filechooser.showDialog(FileChooserDemo2.this,
                                       "Send");
```

El primer argumento del método **showDialog** es el componente padre para el diálogo. El segundo argumento es un **String** que proporciona tanto el título de la ventana de diálogo como la etiqueta del botón"accept".

De nuevo, el selector de ficheros no hace nada con el fichero seleccionado. El programa es responsable de implementar la tarea personalizada para la que se creó el selector de ficheros.

Filtrar la lista de ficheros

Por defecto, un selector de ficheros muestra todos los ficheros y directorios que detecta. Un programa puede aplicar uno o más filtros de ficheros a un selector de ficheros para que el selector sólo muestre algunos de ellos. El selector de ficheros llama al método **accept** del filtro con cada fichero para determinar si debería ser mostrado. Un filtro de ficheros acepta o rechaza un fichero basándose en algún criterio como el tipo, el tamaño, el propietario, etc.

JFileChooser soporta tres clases de filtrado. Los filtros se chequean en el orden aquí listado. Por eso un filtro del segundo tipo solo puede filtrar aquellos ficheros aceptados por el primero, etc.

Friltrado interno
: El filtrado se configura a través de llamadas a métodos específicos de un selector de ficheros. Actualmente el único filtro interno disponible es para los ficheros ocultos. Se llama a **setFileHidingEnabled(true)** para desactivar la selección de ficheros ocultos (como aquellos que empiezan con '.' en sistemas UNIX).

Friltrado controlado por la aplicación
: La aplicación determina los ficheros a mostrar. Se crea una subclase de **FileFilter**, se ejemplariza, y se utiliza el ejemplar como un argumento para **setFileFilter**. El selector de fiheros sólo mostrará los ficheros que acepte el filtro.

Filtrado seleccionable por el usuario

El GUI selector de ficheros proporciona una lista de filtros de la que el usuario puede elegir uno. Cuando el usuario elige un filtro, el selector de ficheros muestra sólo aquellos ficheros que acepte el filtro. **FileChooserDemo2** añade un filtro de ficheros personalizado a su lista de filtros seleccionables del selector de ficheros.

```
filechooser.addChoosableFileFilter(new ImageFilter());
```

El filtro personalizado se implementa en **ImageFilter.java**, como una sublcase de **FileFilter**. La clase **ImageFilter** implementa el método **getDescription** para devolver un string y ponerlo en la lista de filtros seleccionables. Como muestra el siguiente código, **ImageFilter** implementa el método **accept** para aceptar todos los directorios y cualquier fichero que tenga las extensiones ".jpg", ".jpeg", ".gif", ".tif", or ".tiff".

```
public boolean accept(File f) {

    if (f.isDirectory()) {
    return true;
    }

    String s = f.getName();
    int i = s.lastIndexOf('.');

    if (i > 0 &&  i < s.length() - 1) {
    String extension = s.substring(i+1).toLowerCase();
    if (tiff.equals(extension) ||
        tif.equals(extension) ||
        gif.equals(extension) ||
        jpeg.equals(extension) ||
        jpg.equals(extension)) {
            return true;
    } else {
        return false;
    }
    }

    return false;
}
```

Aceptando todos los directorios, este filtro permite al usuario navegar a través del sistema de ficheros. Si se omitieran las líneas en negrita de este método, el usuario se vería limitado al directorio en que se inicializó el selector de ficheros.

Personalizar un Visor de Ficheros

Un selector de ficheros presenta una lista de ficheros para elegir uno. En el aspecto y comportamiento Metal, la lista del selector muestra cada nombre de fichero y mustra un pequeño icono que representa si el fichero es un verdadero fichero o un directorio. Podemos personalizar la **visión** de la lista creando una subclase personalizada de**FileView**, y utilizando un ejemplar de la clase como un argumento al método **setFileView**. El ejemplo utiliza un ejemplar de **ImageFileView** como el visor de ficheros para el selector.

```
filechooser.setFileView(new ImageFileView());
```

ImageFileView muestra un icono diferente para cada tipo de imagen aceptada por el filtro de imágenes descrito anteriormente.

La clase **ImageFileView** sobreescribe los cinco métodos abstractos definidos en **FileView**.

String getTypeDescription(File f)

Devuelve una descripción del tipo de fichero. Aquí podemos ver la implementación e este método en **ImageFileView**.

```
public String getTypeDescription(File f) {
    String extension = getExtension(f);
    String type = null;

    if (extension != null) {
        if (extension.equals("jpeg")) {
            type = "JPEG Image";
        } else if (extension.equals("gif")){
            type = "GIF Image";
        } else if (extension.equals("tiff")) {
            type = "TIFF Image";
        }
    }
    return type;
}
```

Icon getIcon(File f)

Devuelve un icono que representa el fichero o su tipo. Aquí tenemos la implementación de este método en **ImageFileView**.

```
public Icon getIcon(File f) {
    String extension = getExtension(f);
    Icon icon = null;
    if (extension != null) {
        if (extension.equals("jpeg")) {
            icon = jpgIcon;
        } else if (extension.equals("gif")) {
            icon = gifIcon;
        } else if (extension.equals("tiff")) {
            icon = tiffIcon;
        }
    }
    return icon;
}
```

String getName(File f)

Devuelve el nombre del fichero. La mayoría de las implementaciónes de este método deberían responder **null** para indicar que el aspecto y comportamiento debería imaginárselo. Otra implementación común devuelve **f.getName()**.

String getDescription(File f)

Devuelve una descripción del fichero. Una implementación común de este método devuelve **null** para indicar que el aspecto y comportamiento debería imaginárselo.

Boolean isTraversable(File f)

Devuelve si un directorio es atravesable o no. La mayoría de las implementaciones de este método deberían responder **null** para indicar que el aspecto y comportamiento debería imaginárselo. Algunas aplicaciones podrían querer avisar al usuario de no descender a ciertos tipos de directorios porque representan documentos compuestos.

La implementación que hace **ImageFileView** de los métodos **getTypeDescription** y **getIcon** utilizan un método personalizado **getExtension**.

```
private String getExtension(File f) {

    String ext = null;
    String s = f.getName();
    int i = s.lastIndexOf('.');

    if (i > 0 &&  i < s.length() - 1) {
        ext = s.substring(i+1).toLowerCase();
    }
    return ext;
}
```

Proporcionar un acesorio de visionado

El selector de ficheros personalizado en **FileChooserDemo2** tiene un accesorio de visionado. Si el ítem seleccionado es una imagen JPEG, TIFF, o GIF, el accesorio de visionado muestra una pequeña imagen del fichero. Si no lo es, el accesorio de visionado está vacío.

El ejemplo llama al método **setAccessory** para establecer un ejemplar de **ImagePreview** como accesorio de visionado del selector.

```
filechooser.setAccessory(new ImagePreview(filechooser));
```

Cualquier objeto que desciende de **JComponent** puede ser un accesorio de visionado. El componente debería implementar **paint** o **paintComponent**, y tener un tamaño predeterminado que parezca adecuado en el selector de ficheros.

El selector de ficheros dispara un evento de cambio de propiedad cuando el usuario selecciona un ítem de la lista. Por eso, un programa con accesorio de visionado debe registrarse para recibir estos eventos y actualizarse cada vez que la selección cambie. En el ejemplo, el propio objeto **ImagePreview** se registra para estos eventos. Esto mantiene en una sola clase todo el código relacionado con el accesorio de visionado.

Aquí podemos ver la implementación del método **propertyChange** en el ejemplo, que es el método llamado cuando se dispara un evento de cambio de propiedad.

```
public void propertyChange(PropertyChangeEvent e) {
    String prop = e.getPropertyName();
    if (prop == JFileChooser.SELECTED_FILE_CHANGED_PROPERTY) {
        f = (File) e.getNewValue();
        if (isShowing()) {
            loadImage();
            repaint();
        }
    }
}
```

Este método carga la imagen y redibuja el accesorio de visionado, si **SELECTED_FILE_CHANGED_PROPERTY** es la propiedad que ha cambiado.

El API de FileChooser

El API para usar selectores de ficheros se divide en estas gategorías.

Crear y Mostrar un Selector de Ficheros

Método	Propósito
JFileChooser() **JFileChooser(File, FileSystemView)** **JFileChooser(File)** **JFileChooser(FileSystemView)** **JFileChooser(String, FileSystemView)** **JFileChooser(String)**	Crea un ejemplar de JFileChooser.
int showOpenDialog(Component) **int showSaveDialog(Component)** **int showDialog(Component, String)**	Muestra un diálogo modal conteniendo el selector de ficheros.

Navegar por la Lista del Selector de Ficheros

Método	Propósito
void ensureFileIsVisible(File)	Fuerza el fichero indicado a ser visible en la lista de ficheros.
void setCurrentDirectory(File) **File getCurrentDirectory**	Selecciona u obtiene el directorio cuyos ficheros se están mostrando en el selector de ficheros.
void changeToParentDirectory()	Cambia la lista para mostrar el directorio padre del directorio actual.
void rescanCurrentDirectory()	Comprueba el sistema de ficheros y actualiza la lista del selector.

Personalizar el Selector de Ficheros

Método	Propósito
JComponent getAccessory() **void setAccessory(JComponent)**	Selecciona u obtiene el accesorio del selector de ficheros.
void setFileFilter(FileFilter) **FileFilter getFileFilter()**	Selecciona u obtiene el filtro primario del selector de ficheros.
void setFileView(FileView) **FileView getFileView()**	Selecciona u obtiene el visor de ficheros del selector.
FileFilter[] getChoosableFileFilters() **void setChoosableFileFilters(FileFilter[])** **void addChoosableFileFilter(FileFilter)** **boolean removeChoosableFileFilter(FileFilter)** **void resetChoosable(FileFilter)** **FileFilter getAcceptAllFileFilter()**	Selecciona, obtiene o modifica la lista de filtros seleccionables.
void setFileHidingEnabled(boolean) **boolean isFileHidingEnabled()**	Selecciona u obtiene si se muestran los ficheros ocultos.

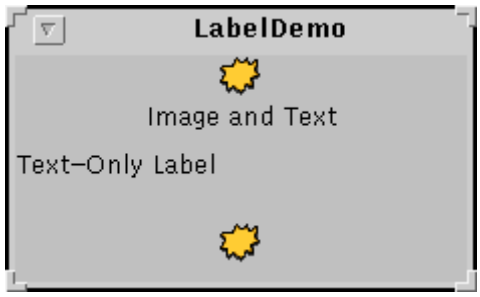
Seleccionar Ficheros y Directorios

Método	Propósito
void setFileSelectionMode(int) **int getFileSelectionMode()** **boolean isDirectorySelectionEnabled()** **boolean isFileSelectionEnabled()**	Selecciona el modo de selección de ficheros. Los valores aceptables son **FILES_ONLY**, **DIRECTORIES_ONLY**, y **FILES_AND_DIRECTORIES**.
void setMultiSelectionEnabled(boolean) **boolean isMultiSelectionEnabled()**	Selecciona u obtiene si se pueden seleccionar varios ficheros a la vez.
void setSelectedFile(File) **File getSelectedFile()**	Selecciona u obtiene el fichero actualmente seleccionado.
void setSelectedFiles(File[]) **File[] getSelectedFiles()**	Selecciona u obtiene los ficheros actualmente seleccionados.

¿Cómo Usar Label?

Con la clase **JLabel**, se puede mostrar texto no seleccionable e imágenes. Si necesitamos crear un componente que muestre un sencillo texto o una imagen, reaccionando opcionalmente a la entrada del usuario, podemos hacerlo utilizando un ejemplar de **JLabel** o de una subclase personalizada de **JLabel**. Si el componente interactivo tiene estado, pobablemente deberíamos utilizar un **button** en vez de una etiqueta.

Aquí podemos ver una imagen de una aplicación que muestra tres etiquetas. La ventana está dividida entres filas de la misma altura, la etieueta de cada fila es lo más ancha posible.



Intenta esto:

1. Compila y ejecuta la aplicación. El código fuente está en **LabelDemo.java**, y la imagen en **middle.gif**.
2. Redimensiona la ventana para poder ver cómo los contenidos de las etiquetas se sitúan con las áreas de dibujo.

   Todos los contenidos de las etiquetas tienen el alineamiento vertical por defecto -- los contenidos están centrados verticalmente en el área de la etiqueta. La etiqueta superior que contiene texto e imagen, tiene alineamiento horizontal centrado. La segunda etiqueta, que sólo tiene texto, tiene alineamiento a la izquierda que es por defecto para las etiquetas de sólo texto. La tercera etiqueta, que contiene sólo una imagen, tiene alineamiento horizontal centrado, que es por defecto para las etiquetas sólo de imágenes.

Abajo puedes ver el código de **LabelDemo.java** que crea las etiquetas del ejemplo anterior.

```
ImageIcon icon = new ImageIcon("images/middle.gif");
. . .
label1 = new JLabel("Image and Text",
                                icon,
                                JLabel.CENTER);
//Set the position of the text, relative to the icon.
label1.setVerticalTextPosition(JLabel.BOTTOM);
label1.setHorizontalTextPosition(JLabel.CENTER);

label2 = new JLabel("Text-Only Label");

label3 = new JLabel(icon);

//Add labels to the JPanel.
add(label1);
add(label2);
add(label3);
```

El API Label

Las siguientes tablas listan los métodos y constructores más utilizados de **JLabel**. Otros métodos están definidos por la clase **Component**. Incluyen **setFont** y **setForeground**. El API se divide en dos categorías.

Seleccionar u Obtener el Contenido de la Etiqueta

Método	Propósito
JLabel(Icon) **JLabel(Icon, int)** **JLabel(String)** **JLabel(String, Icon, int)**	Crea un ejemplar de **JLabel**, inicializándolo para tener texto/imagen/alineamiento especificados. El argumento **int** especifica el alineamiento horizontal del contenido de la etiqueta dentro de su área de dibujo. El alineamiento horizontal debe ser una de las siguientes constantes definidas en el interface **SwingConstants** (que implementa **JLabel**): **LEFT**, **CENTER**, o **LEFT**.

JLabel(String, int) **JLabel()**	
void setText(String) **String getText()**	Selecciona u obtiene el texto mostrado por la etiqueta.
void setIcon(Icon) **Icon getIcon()**	Selecciona u obtiene la imagen mostrada por la etiqueta.
void setDisplayedMnemonic(char) **char getDisplayedMnemonic()**	Selecciona u obtiene la letra que debería ser la tecla alternativa. Esto es muy útil cuando una etiqueta describe un componente (como un campo de texto) que tiene un tecla alternativa pero no puede mostrarla.
void setDisabledIcon(Icon) **Icon getDisabledIcon()**	Selecciona u obtiene la imagen mostrada por la etiqueta cuando está desactivada. Si no se especifica esta imagen, el aspecto y comportamiento crea una manipulando la imagen por defecto.

Ajuste Fina de la Apariencia de la Etiqueta

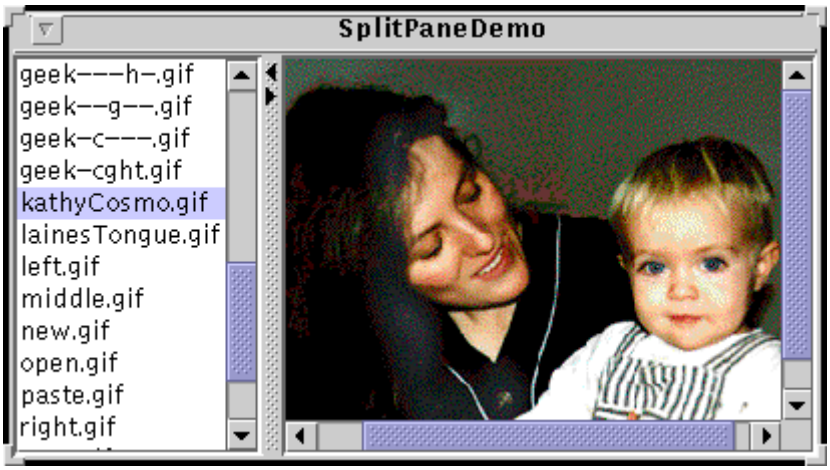
Método	Propósito
void setHorizontalAlignment(int) **void setVerticalAlignment(int)** **int getHorizontalAlignment()** **int getVerticalAlignment()**	Selecciona u obtiene donde debería mostrarse el contenido de la etiqueta. El Interface **SwingConstants** define tres posibles valores para el alineamiento horizontal: **LEFT** (por defecto para etiquetas de sólo texto), **CENTER** (por defecto para etiquetas de sólo imagen), o **LEFT**; y tres para alineamiento vertical: **TOP**, **CENTER** (por defecto), y **BOTTOM**.
void setHorizontalTextPosition(int) **void setVerticalTextPosition(int)** **int getHorizontalTextPosition()** **int getVerticalTextPosition()**	Selecciona u obtiene dónde debería mostrarse el texto del botón con respecto a su imagen. El interface **SwingConstants** define tres posibles valores para posición horizontal: **LEFT**, **CENTER**, y **LEFT** (por defecto); y tres para posición vertical: **TOP**, **CENTER** (por defecto), y **BOTTOM**.
void setIconTextGap(int) **int getIconTextGap()**	Selecciona u obtiene el número de pixels entre el texto de la etiqueta y su imagen.

¿Cómo Usar List?

Un **JList** le presenta al usuario un grupo de ítems para elegir. Los ítems pueden ser cualquier **Object**. Normalmente son un **String**. Un lista puede tener muchos ítems, o podría crecer hasta tenerlos. Cómo la mayoría de las listas se sitúan dentro de paneles desplazables, **JList** es una clase scroll-savvy.

Además de las listas, los siguientes componentes Swing también presentan múltiples ítems seleccionables al usuario: check boxes, combo boxes,menus, radio buttons, y tables. Sólos los checkbox, las tablas, y las listas permiten seleccionar varios ítems a la vez.

Aquí podemos ver una imagen de una aplicación que utiliza **JList** para mostrar los nombres de las imágenes a ver.



Intenta esto:

1. Compila y ejecuta la aplicación. El código fuente está en **SplitPaneDemo.java**. **imagenames.properties** proporciona los nombres de las imágenes para ponerlos en el **JList**.
2. Elige una imagen de la lista. Utiliza las barras de desplazamiento para ver más nombres.

Abajo está el código de **SplitPaneDemo.java** que crea y configura la lista.

```
...where member variables are declared
   this Vector is initialized from a properties file...
static Vector imageList;
...where the GUI is created...
// Create the list of images and put it in a scroll pane
JList listOfImages = new JList(imageList);
listOfImages.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
listOfImages.setSelectedIndex(0);
listOfImages.addListSelectionListener(this);
JScrollPane listScrollPane = new JScrollPane(listOfImages);
```

El código utiliza un objeto **Vector** para proporcionar una lista con los ítems iniciales. También podemos inicializar una lista con un array o con un objeto **ListModel**.

En este ejemplo, el **Vector** contiene strings obtenidas desde un fichero de propiedades. Sin embargo, los valores de la lista pueden ser cualquier **Object**, en cuyo caso el método **toString** de la clase **Object** proporciona el texto a mostrar. Para mostrar un ítem como una imagen u otro valor no-texto, debemos proporcionar una celta personalizada con **setCellRenderer**.

Por defecto, una lista permite que cualquier combinación de ítems sea seleccionada a la vez. Podemos utilizar un valor por defecto diferente utilizando el método **setSelectionMode**. Por ejemplo, **SplitPaneDemo** configura el modo de selección a **SINGLE_SELECTION** (una constante definida por **ListSelectionModel**) para que sólo pueda seleccionarse un ítem de la lista. La siguiente lista describe los tres modos de selección disponibles.

Modo	Descripción	Ejemplo
SINGLE_SELECTION	Sólo un ítem de la lista puede ser seleccionado. Cuando el usuario selecciona un ítem, cualquier ítem anteriormente seleccionado se deselecciona primero.	
SINGLE_INTERVAL_SELECTION	Se pueden seleccionar varios ítems contiguos. Cuando el usuario empieza una nueva selección, cualquier ítem anteriormente seleccionado se deselecciona primero.	
MULTIPLE_INTERVAL_SELECTION	El valor defecto. Se puede seleccionar cualquier combinación de ítems. El usuario debe deseleccionar explícitamente los ítems.	

No importa el modo de selección que utiliza la lista, siempre dispara eventos "list selection" cuando cambia la selección. Se pueden procesar esos eventos añadiendo un Oyente de "list selection" a la lista con el método **addListSelectionListener**.

Un oyente de 'list selection' debe implementar un método : **valueChanged**. Aquí podemos ver el método **valueChanged** para el oyente de **SplitPaneDemo**.

```
public void valueChanged(ListSelectionEvent e) {
    if (e.getValueIsAdjusting())
        return;

    JList theList = (JList)e.getSource();
    if (theList.isSelectionEmpty()) {
        picture.setIcon(null);
    } else {
        int index = theList.getSelectedIndex();
        ImageIcon newImage = new ImageIcon("images/" +
```

```
                                (String)imageList.elementAt(index));
            picture.setIcon(newImage);
            picture.setPreferredSize(new Dimension(newImage.getIconWidth(),
                                                   newImage.getIconHeight() ));
            picture.revalidate();
        }
    }
```

Observa que el método **valueChanged** sólo actualiza la imagen si **getValueIsAdjusting** devuelve **false**. La mayoría de los eventos 'list selection' pueden ser generados desde una simple acción del usuario como una pulsación del ratón. Este programa particular sólo está interesado en el resultado final de la acción del usuario.

ómo la lista esta en modo de selección sencillo, este código puede utilizar **getSelectedIndex** para obtener el índice sólo del índice seleccionado.

JList proporciona otros métodos para Seleccionar u Obtener la Selección cuando el modo de selección permite seleccionar más de un ítem. Por ejemplo, cambiemos el modo de selección de una lista dinámicamente, podemos ver Ejemplos de Manejos de Eventos 'List Selection'.

El API List

Las siguientes tablas listan los métodos y constructores más utilizados de **JList**. Otros métodos útiles están definidos por las clases **JComponent** y **Component**.

Muchas de las operaciones de una lista están manejadas por otros objetos. Por ejemplo, los ítems de la lista están manejados por un objeto ListModel, la selección está manejada por un objeto ListSelectionModel, y la mayoría de los programas ponen una lista en un panel desplazable para menajar el desplazamiento. No tenemos que preocuparnos de la mayor parte de estos objetos auxiliares, porque **JList** los crea cuando son necesarios, y nosotros interactuamos con ellos implicitamente con los métodos de conveniencia de **JList**.

Cómo se ha dicho, el API para utilizar Lists de divide en estas categorías.

Seleccionar Ítems de la Lista

Método	Propósito
JList(ListModel) **JList(Object[])** **JList(Vector)**	Crea una lista con los ítems especificados. El segundo y tercer constructores crean implicitamente un **ListModel**.
void setModel(ListModel) **ListModel getModel()**	Selecciona u obtiene el modelo que contiene el contenido de la lista. Podemos modificar dinámicamente el contenido de la lista llamado a los métodos con el objeto devuelto por **getModel**.
void setListData(Object[]) **void setListData(Vector)**	Selecciona los ítems de la lista. Estos métodos crean implícitamente un **ListModel**.

Manejar la Selección de una Lista

Método	Propósito
void addListSelectionListener(**ListSelectionListener)**	Registra para recibir notificación de los cambios de selección.
void setSelectedIndex(int) **void setSelectedIndices(int[])** **void setSelectedValue(Object, boolean)** **void setSelectedInterval(int, int)**	Configura la selección actual como indicada. Utiliza el método **setSelectionMode** para los rangos de selecciones aceptables. El argumento booleano especifica su la lista debería intentar desplazarse a sí misma para que el ítem seleccionado sea visible.

int getSelectedIndex() **int getMinSelectionIndex()** **int getMaxSelectionIndex()** **int[] getSelectedIndices()** **Object getSelectedValue()** **Object[] getSelectedValues()**	Obtiene información sobre la selección actual.
void setSelectionMode(int) **int getSelectionMode()**	Selecciona u obtiene el modod de selección. Los valores aceptables son. **SINGLE_SELECTION**, **SINGLE_INTERVAL_SELECTION**, o **MULTIPLE_INTERVAL_SELECTION**.
void clearSelection() **boolean isSelectionEmpty()**	Selecciona u obtiene si hay algún ítem seleccionado.
boolean isSelectedIndex(int)	Determina si el índice específicado está seleccionado.

Trabajar con un ScrollPane

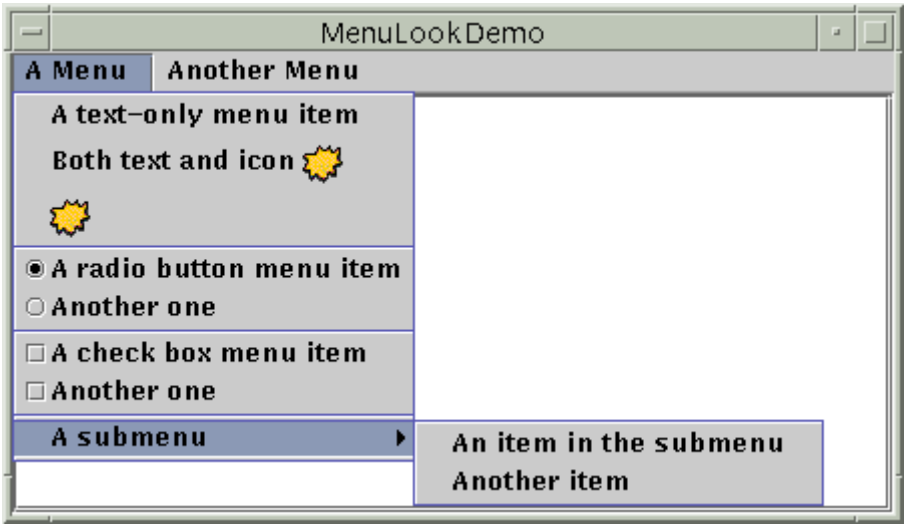
Método	Propósito
void ensureIndexIsVisible(int)	Desplaza para que el índice especificado sea visible dentro del recuadro de la lista.
int getFirstVisibleIndex() **int getLastVisibleIndex()**	Obtiene el índice del primer o el último elemento visible de la lista.
void setVisibleRowCount(int) **int getVisibleRowCount()**	Selecciona u obtiene cúantas filas de la lista son visibles.

¿Cómo Usar Menu?

Un menú proporciona una forma de ahorrar espacio y permitir al usuario elegir una entre varias opciones. Otros componentes con los que el usuario puede hacer una elección incluyen combo boxes, lists, radio buttons, y tool bars. Si alguno de los ítems de un menú realiza una acción que está duplicada en otro ítem de menú o en un botón de una barra de herramientas, además de esta lección deberíamos leer Como usar Actions.

Los menús son únicos en que, por convención, no se sitúan con los otros componentes en el UI. En su lugar, aparecen en una **barra de menú** o en un menú **desplegable**. Una barra de menú contiene uno o más menús, y tiene una posición dependiente de la plataforma -- normalmente debajo de la parte superior de la ventana. Un menú desplegable es un menú que es invisible hasta que el usuario hace una acción del ratón específica de la plataforma, como pulsar el botón derecho del ratón sobre un componente. Entonces el menú desplegable aparece bajo el cursor.

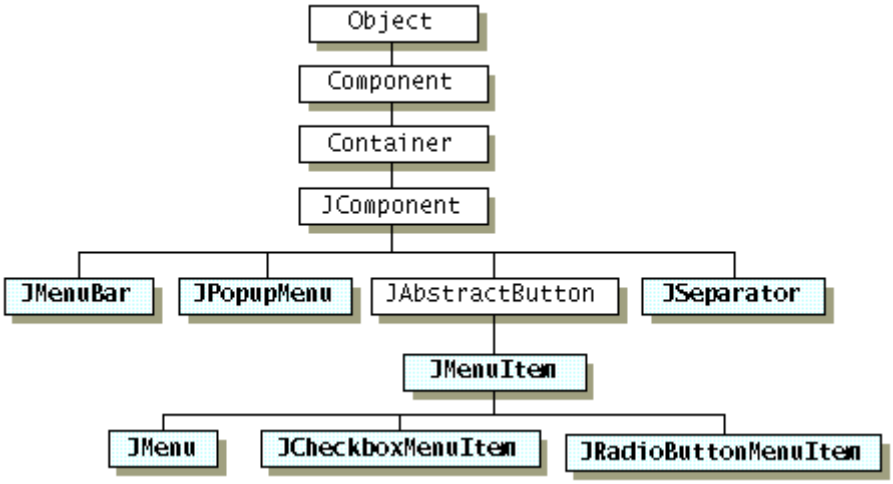
La siguiente figura muestra los componentes Swing que implementan cada parte de un sistema de menús.



El resto de esta sección nos enseña los componentes de menú y nos dice cómo utilizar varias de sus características.

La herencia de componentes Menú

Aquí podemos ver el árbol de herencia de las clases relacionadas con los menús.



Como se ve en la figura, los ítems de menús (incluidos los propios menús) son simples botones. Podríamos preguntarnos como un menú, si es sólo un botón, muestra sus ítems. La respuesta es que cuando se activa un menú, automáticamente trae un menú desplegable que muestra sus ítems.

Crear Menús

Aquí está el código que crea los menús mostrados al principio de esta página. Puedes encontrar el programa completo en **MenuLookDemo.java**. Para ejecutarlo, necesitas tener el fichero de imagen: **images/middle.gif**. Como este código no tiene manejo de eventos, los menús no hacen nada útil, excepto verse como serían. Si ejecutamos el ejemplo, observaremos que a pesar de no tener un manejo de eventos, los menús y submenús aparecen cuando deben, y los checkbox y los botones de radio responden apropiadamente cuando el usuario los elige.

```
//in the constructor for a JFrame subclass:
JMenuBar menuBar;
JMenu menu, submenu;
JMenuItem menuItem;
JCheckBoxMenuItem cbMenuItem;
JRadioButtonMenuItem rbMenuItem;
...
//Create the menu bar.
menuBar = new JMenuBar();
setJMenuBar(menuBar);

//Build the first menu.
menu = new JMenu("A Menu");
menuBar.add(menu);

//a group of JMenuItems
menuItem = new JMenuItem("A text-only menu item");
menu.add(menuItem);
menuItem = new JMenuItem("Both text and icon",
new ImageIcon("images/middle.gif"));
```

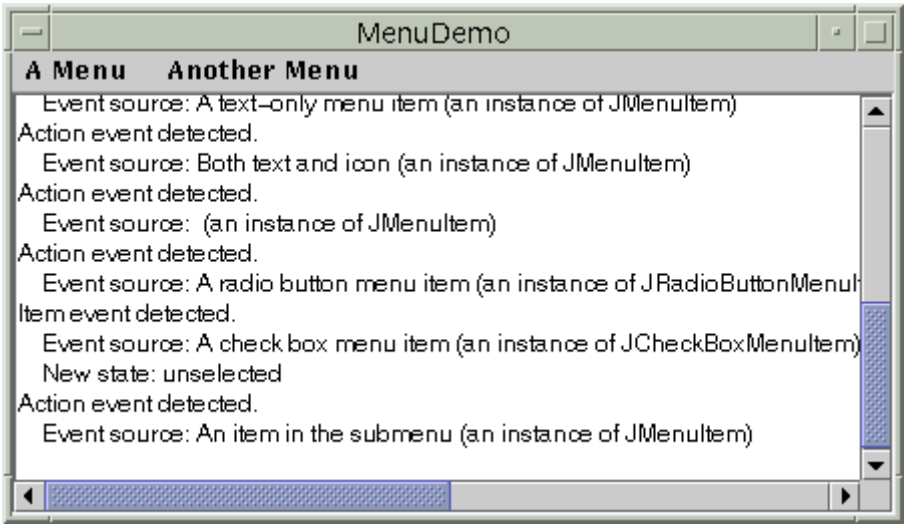
```
menu.add(menuItem);
menuItem = new JMenuItem(new ImageIcon("images/middle.gif"));
menu.add(menuItem);

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

//a group of check box menu items
menu.addSeparator();
cbMenuItem = new JCheckBoxMenuItem("A check box menu item");
menu.add(cbMenuItem);
cbMenuItem = new JCheckBoxMenuItem("Another one");
menu.add(cbMenuItem);

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

//Build second menu in the menu bar.
menu = new JMenu("Another Menu");
menuBar.add(menu);
```

Como se ve en el código, para configurar una barra de menú para un **JFrame**, se utiliza el método **setJMenuBar**. Para añadir un **JMenu** a un **JMenuBar**, se utiliza el método **add(JMenu)**. Para añadir ítems de menú y submenús a un **JMenu**, se utiliza el método **add(JMenuItem)**. Estos métodos y otros más se listan en El API de JMenu.

Manejar Eventos desde Ítems de Menús

Para detectar cuando el usuario selecciona un **JMenuItem**, se puede escuchar por eventos action (igual que se haría para un **JButton**). Para detectar cuando el usuario selecciona un **JRadioButtonMenuItem**, se puede escuchar tanto por eventos action, como por eventos item, como se describio en Cómo usar Radio Buttons. Para **JCheckBoxMenuItem**s, generalmente se escuchan eventos de item, como se describió en Cómo usar CheckBox.

La siguiente figura muestra un programa que añade detección de eventos al ejemplo anterior. El código del programa está en **MenuDemo.java**. Al igual que MenuLookDemo, MenuDemo utiliza el fichero de imagen **images/middle.gif**.



Aquí está el código que implementa el manejo de eventos.

```
public class MenuDemo ... implements ActionListener, ItemListener {
    ...
    public MenuDemo() {
        ./for each JMenuItem instance:
        menuItem.addActionListener(this);

        ./for each JRadioButtonMenuItem:
        rbMenuItem.addActionListener(this);

        ./for each JCheckBoxMenuItem:
        cbMenuItem.addItemListener(this);
        ...
    }

    public void actionPerformed(ActionEvent e) {
        ./Get information from the action event...
        ./Display it in the text area...
    }

    public void itemStateChanged(ItemEvent e) {
        ./Get information from the item event...
        ./Display it in the text area...
```

```
    }
```

Para ejemplos de manejo de eventos action e item, puedes ver las páginas, button, radio button, y check box, así como la lista de ejemplos al final de esta página.

Traer un Menú Desplegable

Para traer un menú desplegable (**JPopupMenu**), debemos registrar un oyente de ratón para cada componente al que debería estar asociado el menú desplegable. El oyente de mouse debe detectar las peticiones del usuario para que aparezca el menú desplegable. Para las plataformas Windows y Motif, el usuario trae un menú desplegable pulsando el botón derecho del ratón mientras el cursor está sobre el coponente adecuado.

El oyente de mouse trae un menú desplegable llamando a **setVisible(true)** sobre el ejemplar apropiado de **JPopupMenu**. El siguiente código, tomado de **PopupMenuDemo.java**, muestra cómo crear y mostrar menús desplegables.

```
./where instance variables are declared:
JPopupMenu popup;

    ./where the GUI is constructed:
    //Create the popup menu.
    popup = new JPopupMenu();
    menuItem = new JMenuItem("A popup menu item");
    menuItem.addActionListener(this);
    popup.add(menuItem);
    menuItem = new JMenuItem("Another popup menu item");
    menuItem.addActionListener(this);
    popup.add(menuItem);

    //Add listener to components that can bring up popup menus.
    MouseListener popupListener = new PopupListener();
    output.addMouseListener(popupListener);
    menuBar.addMouseListener(popupListener);
...
class PopupListener extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        maybeShowPopup(e);
    }

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

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

Los menús desplegables tienen unos pocos detalles interesantes de implementación. Uno es que cada menú tiene un menú desplegable apropiado. Cuando el menú se activa, utiliza su menú desplegable para mostrar sus ítems de menú.

Otro detalle es que un propio menú desplegable utiliza otro componente para implementar la ventana que contiene los ítems del menú. Dependiendo de las circunstancias bajo las que se muestre el menú desplegable, podría implementar su "ventana" utilizando un componente de peso ligero (como un **JPanel**), un componente de peso medio (como un **Panel**), o una ventana de peso pesado (**Window**).

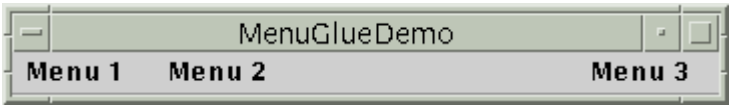
Las ventanas desplegables de peso ligero son más eficientes que las ventanas de peso pesado, pero no funcionan bien si tenemos componentes pesados dentro de nuestro GUI. Especificamente, cuando un área de una ventana desplegable de peso ligero se intersecciona con un componente de peso pesado, el componente de peso pesado se dibuja encima. Esta es una de las razones porla que recomendamos no mezclar componentes de peso ligero y de peso pesado. Si realmente necesitamos utilizar un componente de peso pesado en nuestro GUI, podemos utilizar el método **setLightWeightPopupEnabled** de **JPopupMenu** para desactivar las ventanas desplegables de peso ligero. Para más detalles puedes ver el artículo Mezclar componentes de peso ligero y pesado, de La conexión Swing. (En inglés).

Personalizar la Distribución de un Menú

Como los menús se hacen con componentes ordinarios Swing, podemos personalizarlos fácilmente. Por ejemplo, podemos aañadir cualquier componente de peso ligero a un **JMenu** o **JMenuBar**. Y como **JMenuBar** utiliza **BoxLayout**, podemos personalizar la distribución de la barra de menú añadiéndole componentes invisibles. Aquí tienes un ejemplo que añade un componente glue a una barra de menú, para que el último elemento del menú se sitúe en el lado derecho de la barra de menú.

```
./create and add some menus...
menuBar.add(Box.createHorizontalGlue());
./create the LEFTmost menu...
menuBar.add(LEFTMenu);
```

Aquí podemos ver una imagen del resultado, que podemos duplicar compilando y ejecutando **MenuGlueDemo.java**.



Otra forma de cambiar el aspecto de un menú es cambiar el controlador de distribución que lo controla. Por ejemplo, podemos cambiar el controlador de distribución de la barra de menú del valor por defecto **BoxLayout** de izquierda-derecha, a algo como un **GridLayout**.

También podemos cambiar como un menú activado u otro menú desplegable distribuye sus ítems, como demuestra **MenuLayoutDemo.java**. Aquí podmeos ver una imagen de la distribución de menús que crea MenuLayoutDemo.

El API de JMenu

Las siguientes tablas listan los métodos y constructores más utilizados de **Jmenu**. El API se divide en estas categorías.

Crear y Configurar Barras de Menú

Método	Propósito
JMenuBar()	Crea una barra de menú.
void setJMenuBar(JMenuBar) **JMenuBar getJMenuBar()** (en **JApplet**, **JDialog**, **JFrame**, **JRootPane**)	Selecciona u obtiene la barra de menú de un applet, dialog, frame, o root pane. En las siguientes versiones de Swing y del JDK 1.2, los frames internos también soportarán estos métodos.
void setMenuBar(JMenuBar) **JMenuBar getMenuBar()** (en **JInternalFrame**)	Selecciona u obtiene la barra de menú de un **Frame interno**. En las siguientes versiones de Swing y del JDK 1.2, este método será anulado y deberíamos utilizar **setJMenuBar/getJMenuBar**.

Crear y Rellenar Menús

Método	Propósito
JMenu()	Crea un menú.
JMenuItem add(JMenuItem) **JMenuItem add(Action)** **void add(String)**	Añade un ítem de menú al final del menú. Si el argumento es un objeto **Action**, el menú crea un ítem de menú como se describe en Cómo usar Actions. Si el argumento es un string, el menú crea automáticamente un objeto **JMenuItem** que muestra el texto especificado.
void addSeparator()	Añade un separador la final del menú.
>**JMenuItem insert(JMenuItem, int)** **JMenuItem insert(Action, int)** **void insert(String, int)** **void insertSeparator(int)**	Inserta un ítem de menú o un separador en un menú, en la posición especificada. El primer ítem de menú es la posición 0, el segundo la posición 1, etc. Los argumentos **JMenuItem**, **Action**, y **String** se tratan de la misma forma que en los correspondientes métodos **add**.
void remove(JMenuItem) **void remove(int)** **void removeAll()**	Elimina el ítem o ítems especificados del menú. Si el argumento es un entero, especifica la posición del ítem a eliminar.

Crear y Rellenar Menús Desplegables

Método	Propósito
JPopupMenu() **JPopupMenu(String)**	Crea un menú desplegable. El argumento string opcional especifica el título que el aspecto y comportamiento podría mostrar como parte de la ventana desplegable.
JMenuItem add(JMenuItem) **JMenuItem add(Action)**	Añade un ítem de menú al final del menú desplegable. Si el argumento es un objeto **Action**, el menú crea un ítem como se describe en Cómo usar Actions.
void addSeparator()	Añade un separador al final del menú desplegable.
void insert(Component, int) **JMenuItem insert(Action, int)**	Inserta un ítem de menú en la posición especificada. El primer ítem del menú está en la posición 0, el segundo en la posición 1, etc. El argumento **Component** específica el ítem de menú a añadir. El argumento **Action** es tratado de la misma forma que en el método **add** correspondiente.
void remove(JMenuItem) **void remove(int)** **void removeAll()**	Elimina el ítem o ítems especificados del menú. Si el argumento es un entero, especifica la posición del elemento del menú a eliminar.
static void setDefaultLightWeightPopupEnabled(boolean)	Por defecto, Swing implementa una ventana de menú utilizando un componente de peso ligero. Esto causa problemas su utilizamos componentes de peso pesado en nuestro programa Swing, como se describió en Traer un Menú Desplegable. Para evitar estos problemas, se puede llamar a **JPopupMenu.** **setDefaultLightWeightPopupEnabled(false)** .
void show(Component, int, int)	Muestra el menú desplegable en la posición X,Y (especificada en el orden de los argumentos enteros) en el sistema de coordenadas del componente especificado.

Implementar Ítems de Menú

Método	Propósito
JMenuItem() **JMenuItem(Icon)** **JMenuItem(String)** **JMenuItem(String, Icon)** **JMenuItem(String, int)**	Crea un ítem de menú normal. El argumento icon, si existe, especifica el icono que debería mostrar el ítem de menú. Igualmente el argumento String, especifica el texto que debería mostrar el ítem de menú. El argumento entero especifica el mnemónico de teclado a utilizar. Se puede especifar una de las constantes VK definidas en la clase **KeyEvent**. Por ejemplo, para especificar "a" como el mnemónico, podemos utilizar **KeyEvent.VK_A**.
JCheckBoxMenuItem() **JCheckBoxMenuItem(Icon)** **JCheckBoxMenuItem(String)** **JCheckBoxMenuItem(String, Icon)** **JCheckBoxMenuItem(String, boolean)** **JCheckBoxMenuItem(String, Icon, boolean)**	Crea un ítem de menú que se parece y actúa como un checkbox. Si se especifica un icono, el ítem de menú utiliza el icono en vez del icono por defecto de los checkboxes. El argumento string, si existe, especifica el texto que debería mostrar el ítem de menú. Si se especifica **true** para el argumento booleano, el ítem de menú estará inicialmente seleccionado. De lo contario el íten de menú está desactivado.
JRadioButtonMenuItem() **JRadioButtonMenuItem(Icon)** **JRadioButtonMenuItem(String)** **JRadioButtonMenuItem(String, Icon)**	Crea un ítem de menú que se parece y actúa como un radio buttom. Si se especifica un icono, el ítem de menú utiliza el icono en vez del icono por defecto de los botones de radio. El argumento string, si existe, especifica el texto que debería mostrar el ítem de menú. El ítem de menú está inicialmente desactivado.

void setState(boolean) **boolean getState()**	Seleciona u obtiene el estado de selección de un ítem de menú.
void setEnabled(boolean)	Si el argumento es **true**, activa el ítem de menú, si es **false** lo desactiva.
void setMnemonic(char)	Selecciona la tecla alternativa para seleccionar el ítem de menú sin el ratón.
void setActionCommand(String)	Seleciona el nombre de la acción realizada por el ítem de menú.
void addActionListener(ActionListener) **void addItemListener(ItemListener)**	Añade un oyente de eventos al ítem de menú. Puedes ver Manejar Eventos desde Ítems de Menús para más detalles.
La mayoría de los métodos anteriores son heredados desde **AbstractButton**. Puedes ver El API Button para más información sobre otros métodos útiles proporcionados por **AbstractButton**.	

¿Cómo Usar MonitoProgress?

Una tarea ejecutándose en un programa puede tardar un poco en completarse. Un programa amigable proporciona alguna indicación al usuario sobre lo que puede tardar la tarea y lo que ya lleva realizado.

El paquete Swing proporciona tres clases para ayudar a crear GUIs que monitoricen y muestren el progreso de tareas de larga duración.

JProgressBar
: Una barra de progreso que muestra gráficamente qué cantitad total de la tarea se ha terminado. Puedes ver Cómo usar Progress Bars para más información.

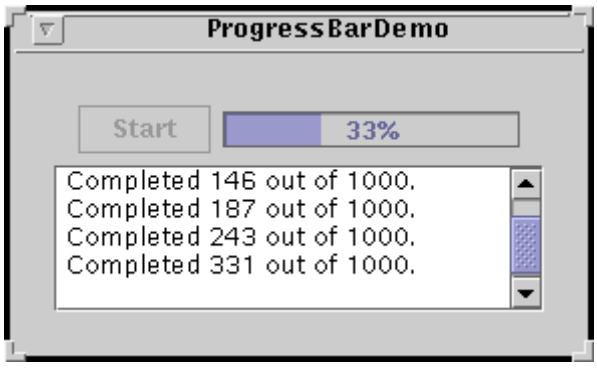
ProgressMonitor
: Un ejemplar de esta clase monitoriza el progreso de una tarea. Si el tiempo enlapsado de la tarea excede un valor especificado en el programa, el monitor trae un diálogo con una descripción de la tarea, una nota de estado, una barra de progreso, un botón **Ok**, y un botón **Cancel**. Puedes ver Cómo usar Progress Monitors para más detalles.

ProgressMonitorInputStream
: Un stream de entrada con un monitor de progreso añadido, que monitoriza la lectura desde el stream. Se utiliza un ejemplar de este stream como cualquier otro stream. Se puede obtener el monitor de progreso del stream llamando a **getProgressMonitor** y configurándolo como se describe en Cómo usar Progress Monitors.

Después de ver una barra de progreso y un monitor de progreso en acción, Decidir si utilizar una Barra o un Monitor de Progreso puede ayudarnos a decidir cuál es el apropiado para nuestra aplicación.

Cómo usar Progress Bars

Aquí podemos ver una imagen de una pequeña aplicación que utiliza una barra de progreso para medir el progreso de una tarea que se ejecuta.



Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente es **ProgressBarDemo.java**. También necesitarás **LongTask.java** y **SwingWorker.java**.
2. Pulsa el botón **Start**. Mira la barra de progreso mientras la tarea progresa. La tarea muestra su salida en el área de texto en la parte inferior de la ventana.

Abajo está el código de **ProgressBarDemo.java** que crea y configura la barra de proceso.

```
...where member variables are delcared...
JProgressBar progressBar;
...
    ...in the constructor for the demo's frame...
    progressBar = new JProgressBar(0, task.getLengthOfTask());
    progressBar.setValue(0);
    progressBar.setStringPainted(true);
```

El constructor usado para crear la barra de progreso selecciona los valores máximo y mínimo de la barra. También se pueden seleccionar estos valores con los métodos **setMinimum** y **setMaximum**. Los valores mínimo y máximo utilizados en este programa son 0 y la longitud de la tarea, lo que es típico de muchos programas y tareas. Sin embarguo, los valores máximo y mínimo de una barra de progreso pueden ser cualquier valor, incluso negativos. El código también selecciona el valor actual de la barra a 0. Los valores mínimo, actual, y máximo deben relacionarse de esta forma.

```
minimum <= current <= maximum
```

Si se intenta seleccionar uno de los valores y el nuevo valor viola la relación, la barra de progreso ajusta uno o más de los otros valores de acuerdo a las reglas establecidas por **BoundedRangeModel** para mantener la relación.

La llamada a **setStringPainted** hace que la barra de progreso muestre un string de porcentaje dentro de sus límites. Por defecto, la cadena indica el porcentaje completo de la barra de progreso. El string de procentaje es el valor devuelto por el método **getPercentComplete** formateado a porcentaje. Otra alternativa es mostrar un string diferente con **setString**.

Se arranca la tarea pulsando el botón **Start**. Una vez que la tarea ha comenzado, un temporizador (un ejemplar de la clase **Timer**) dispara un evento actión cada segundo. Aquí está el método **ActionPerformed** del oyente de ation del temporizador.

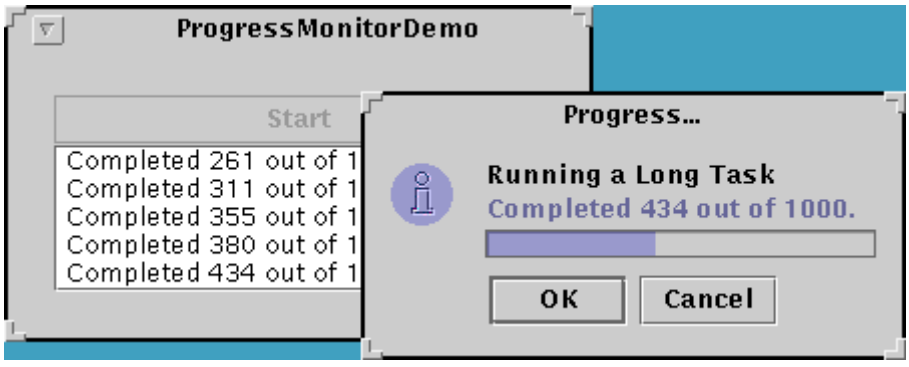
```
public void actionPerformed(ActionEvent evt) {
    progressBar.setValue(task.getCurrent());
    taskOutput.append(task.getMessage() + newline);
    taskOutput.setCaretPosition(taskOutput.getDocument().getLength());
    if (task.done()) {
                Toolkit.getDefaultToolkit().beep();
                timer.stop();
                startButton.setEnabled(true);
                progressBar.setValue(progressBar.getMinimum());
    }
}
```

La línea en negrita obtiene la cantidad de trabajo completada por la tarea y actualiza la barra de progreso con ese valor. Por eso la barra de progreso mide el progreso hecho por la tarea cada segundo, **no** el tiempo enlapsado. El resto del código actualiza la salida, y si la tarea se ha completado, desactiva el temporizador, y resetea los otros controles.

Como se ha mencionado antes, la tarea de larga duración de este programa se ejecuta en un thread separado. Generalmente, es una buena idea aislar una tarea potencialmente de larga duración en su propio thread para que no bloquee el resto del programa. La tarea de larga duración está implementada por **LongTask.java**, que utiliza un **SwingWorker** para asegurarse de que el thread se ejecuta de forma segura dentro de un programa Swing. Puedes ver Usar la clase SwingWorker en Threads y Swing para más información sobre la clase **SwingWorker**.

Cómo usar Progress Monitors

Ahora, reescribamos el ejemplo anterior para utilizar un monitor de progreso en vez de una barra de progreso. Aquí tenemos una imagen del nuevo programa, **ProgressMonitorDemo.java**.



La operación general de este programa es similar al anterior. Se pulsa el botón **Start** para arrancar la misma tarea de larga duración utilizada en el programa anterior. La tarea muestra la salida en el área de texto de la parte inferior de la ventana. Sin embargo, este programa utiliza un monitor de progreso en vez de una barra de progreso.

El ejemplo anterior creaba la barra de progreso al arrancar. En contraste, este programa crea el monitor en el método **actionPerformed** del oyente de action del botón **Start**. Un monitor de progreso no puede utilizarse más de una vez, por eso se debe crear uno nuevo cada vez que se arranca una nueva tarea.

Aquí está la sentencia que crea el monitor de progreso.

```
progressMonitor = new ProgressMonitor(ProgressMonitorDemo.this,
                                      "Running a Long Task",
                                      "", 0, task.getLengthOfTask());
```

El constructor utilizado en este ejemplo inicializa varios parámetros del monitor de progreso.

- El primer argumento proporciona el componente padre del diálogo desplegado para el monitor de progreso.
- El segundo argumento es un **String** que describe la naturaleza de la tarea a monitorizar. Este string se mostrará en el diálogo.
- El tercer argumento es otro **String** que proporciona una nota de estado cambiable. El ejemplo utiliza una cadena vacía porque la nota es actualizada periódicamente cuando la tarea se ejecuta. Si proporcionamos **null** para este argumento, la nota se omite del diálogo. El ejemplo actualiza la nota mientras la tarea se esta ejecutando cada vez que el temporizador dispara un evento action (actualiza el valor actual del monitor al mismo tiempo).
  ```
  progressMonitor.setNote(task.getMessage());
  progressMonitor.setProgress(task.getCurrent());
  ```
- Los últimos dos argumentos proporcionan los valores mínimo y máximo, respectivamente, para la barra de progreso mostrada en el diálogo.

Después el ejemplo crea el monitor de progreso, y lo configura.

```
progressMonitor.setProgress(0);
progressMonitor.setMillisToDecideToPopup(2 * ONE_SECOND);
```

La primera línea selecciona la posición actual de la barra de progreso del diálogo. La segunda indica que el monitor debería desplegar un diálogo si la tarea se ejecuta durante más de dos segundos.

Por el simple echo de que este ejemplo utiliza un monitor de progreso, añade una característica que no estaba presente en la versión del programa que utiliza una barra de progreso. El usuario puede cancelar la tarea pulsando el botón **Cancel** del diálogo. Aquí tenemos el código del ejemplo que chequea si el usuario cancela la tarea o si la tarea sale normalmente.

```
if (progressMonitor.isCanceled() || task.done()) {
    progressMonitor.close();
    task.stop();
    Toolkit.getDefaultToolkit().beep();
    timer.stop();
    startButton.setEnabled(true);
}
```

Observa que el monitor de progreso no cancela por si mismo la tarea. Proporciona el GUI y el API para permitir que el programa lo haga facilmente.

Decidir si utilizar una Barra o un Monitor de Progreso

Se utiliza una **barra de progreso** si.

- Queremos más control sobre la configuración de la barra de progreso. Si estamos trabajando directamente con una barra de progreso, podemos hacer que se muestre verticalmente, podemos proporcionar una cadena para que la muestre, podemos registrar oyentes de cambio, y podemos proporcionarle un modelo de datos personalizado.
- Necesitamos mostrar otros controles o elementos GUI junto con la barra de progreso.
- Necesitamos más de una barra de progreso. Con algunas tareas, necesitamos monitorizar más de un parámetro. Por ejemplo, un programa de instalación podría monitoriazar el espacio de disco utilizado además del número de ficheros que ya han sido instalados.
- Necesitamos reutilizar la barra de progreso. Una barra de progreso puede ser re-utilizada; un monitor de progreso no. El monitor de progreso no tiene un método **reset**, y una vez que **millisToDecideToPopup** ha expirado el monitor ha terminado su trabajo.

Se utiliza un **monitor de progreso** si.

- Si queremos mostrar fácilmente el progreso en un diálogo.
- La tarea a ejecutar es secundaria y el usuario podría no estar interesado en el progreso de la tarea. El monitor de progreso proporciona al usuario una forma para hacer desaparecer el diálogo mientras la tarea se ejecuta.
- Nuestra tarea es cancelable. El monitor de progreso proporciona al usuario una forma de cancelar la tarea. Todo lo que tenemos que hacer es llamar al método **isCanceled** del monitor de progreso para ver si el usuario ha pulsado el botón **Cancel**.
- Nuestra tarea muestra un mensaje corto periódicamente mientras se ejecuta. El diálogo del monitor de progreso proporciona el método **setNote** para que la tarea pueda proporcionar mayor información sobre lo que está haciendo. Por ejemplo, una tarea de instalación podría informar del nombre de cada fichero instalado.
- La tarea podría no tardar mucho tiempo en completarse. Decidimos en que punto una tarea tarda el tiempo suficiente para permitir que el usuario lo sepa. El monitor de progreso no desplegará el diálogo si el la tarea se completa dentro del tiempo especificado.

Si decidimos utilizar un monitor de progreso **y** la tarea que estamos monitorizando lee desde un stream de entrada, se utiliza la clase **ProgressMonitorInputStream**.

El API de ProgressBar

Las siguientes tablas listan los métodos y constructores más utilizados de **JProgressBar**. Otros métodos interesantes están definidos por las clases **JComponent** y **Component**

El API para monitorizador progresos se divide en estas categorías.

Seleccionar u Obtener los Valores/Restricciones de la Barra de Progreso

Método	Propósito
void setValue(int) **int getValue()**	Selecciona u obtiene el valor actual de la barra de progreso. El valor está limitado por los valores máximo y mínimo.
double getPercentComplete()	Obtiene el procentaje terminado por la barra de progreso.
void setMinimum(int) **int getMinimum()**	Selecciona u obtiene el valor mínimo de la barra de progreso.
void setMaximum(int) **int getMaximum()**	Selecciona u obtiene el valor máximo de la barra de progreso.
void setModel(BoundedRangeModel) **BoundedRangeModel getMaximum()**	Selecciona u obtiene el modelo utilizado por la barra de progreso. El modelo establece los valores y restricciones de la barra de progreso. Podemos utilizar este método como alternativa a utilizar los métodos de selección u obtención individuales listados arriba.

Ajuste Fino de la Apariencia de la Barra de Progreso

Método	Propósito
void setOrientation(int) **int getOrientation()**	Selecciona u obtiene si la barra de progreso es vertical u horizontal. Los valores aceptados son **JProgressBar.VERTICAL** o **JProgressBar.HORIZONTAL**.
void setBorderPainted(boolean) **boolean isBorderPainted()**	Selecciona u obtiene si la barra de progreso tiene borde.
void setStringPainted(boolean) **boolean isStringPainted()**	Selecciona u obtiene si la barra de progreso meustra el porcentaje. Por defecto, el valor de la cadena de porcentaje es el valor devuelto por **getPercentComplete** formateado a porcentaje. Podemos cambiar la cadena de porcentaje con **setString**.
void setString(String) **String getString()**	Selecciona u obtiene la cadena de porcentaje.

Configurar un Monitor de Progreso

Método	Propósito
ProgressMonitor(Component, Object, String, int, int)	Crea un monitor de progreso e inicializa el padre del diálogo, el string descriptivo, la nota de estado y los valores mínimo y máximo.
void setMinimum(int) **int getMinimum()**	Selecciona u obtiene el valor mínimo del monitor de progreso.
void setMaximum(int) **int getMaximum()**	Selecciona u obtiene el valor máximo del monitor de progreso.
void setProgress(int)	Actualiza el monitor de progreso.
void setNote(String) **String getNote()**	Selecciona u obtiene la nota de estado. Esta nota se muestra en el diálogo. Para omitirla, se proporciona **null** como el tercer argumento del constructor del monitor.
void setMillisToPopup(int) **int getMillisToPopup()**	Selecciona u obtiene el tiempo después del cual el monitor debería desplegar el diálogo sin importar el estado de la tarea.

void setMillisToDecideToPopup(int) **int getMillisToDecideToPopup()**	Selecciona u obtiene el tiempo después del cual el monitor debería desplegar el diálogo si la tarea no se ha completado.

Terminar el Monitor de Progresos

Método	Propósito
close()	Cierra el monitor de progreso. Esto oculta el diálogo.
boolean isCanceled()	Determina si el usuario ha pulsado el botón **Cancel**.

¿Cómo Usar RadioButton?

Los Botones de Radio son grupos de botones en los que, por convención, sólo uno de ellos puede estar seleccionado. Swing soporta botones de radio con las clases **JRadioButton** y **ButtonGroup**. Para poner un botón de radio en un menú, se utiliza la clase **JRadioButtonMenuItem**. Otras formas de presentar una entre varias opciones son los combo boxes y las listas. Los botones de radio tienen un aspecto similar a los check boxes, pero, por convención, los checkboxes no tienen límites sobre cuantos ítems pueden estar seleccionados a la vez.

Como **JRadioButton** desciende de **AbstractButton**, los botones de radio Swing tienen todas las caracterísitcas de los botones normales. como se explicó en Cómo usar Buttons. Por ejemplo, se puede especificar la imagen mostrada por un botón de radio.

Nota: En Swing 1.0.2, los botones ignoran sus mnemónicos (teclas aceleradoras). Este error se ha corregido en Swing 1.0.3.

Aquí podemos ver una imagen de una aplicación que utiliza cinco botones de radio para elegir qué tipo de mascota mostrar.

Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente es **RadioButtonDemo.java**. También necesitarás cinco ficheros de imágenes: **Bird.gif**, **Cat.gif**, **Dog.gif**, **Rabbit.gif**, y **Pig.gif**.
2. Pulsa el botón 'Dog' o pulsa Alt-d.

   El botón 'Dog' de selecciona, lo que hace que el botón 'Bird' sea deselecciondo. La imagen cambia de un pájaro a un perro. Esta aplicación tiene un oyente de action que escucha todos los botones de radio. Cada vez que el oyente de action recibe un evento, la aplicación muestra la imagen del botón de radio que ha sido seleccionado.

Cada vez que el usuario pulsa un botón de radio, (incluso si ya estaba seleccionado), el botón dispara un **evento action**. También ocurren uno o dos **eventos item** -- uno desde el botón que acaba de ser seleccionado, y otro desde el botón que ha perdido la selección (si existía). Normalmente, las pulsaciones de los botones de radio se manejan utilizando un oyente de action.

Abajo está el código de **RadioButtonDemo.java** que crea los botones de radio en el ejemplo anterior y reaccióna ante las pulsaciones.

```
//In initialization code:
    // Create the radio buttons.
    JRadioButton birdButton = new JRadioButton(birdString);
    birdButton.setMnemonic('b');
    birdButton.setActionCommand(birdString);
    birdButton.setSelected(true);

    JRadioButton catButton = new JRadioButton(catString);
    catButton.setMnemonic('c');
    catButton.setActionCommand(catString);
```

```
    JRadioButton dogButton = new JRadioButton(dogString);
    dogButton.setMnemonic('d');
    dogButton.setActionCommand(dogString);

    JRadioButton rabbitButton = new JRadioButton(rabbitString);
    rabbitButton.setMnemonic('r');
    rabbitButton.setActionCommand(rabbitString);

    JRadioButton teddyButton = new JRadioButton(teddyString);
    teddyButton.setMnemonic('t');
    teddyButton.setActionCommand(teddyString);

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

    // Register a listener for the radio buttons.
    RadioListener myListener = new RadioListener();
    birdButton.addActionListener(myListener);
    catButton.addActionListener(myListener);
    dogButton.addActionListener(myListener);
    rabbitButton.addActionListener(myListener);
    teddyButton.addActionListener(myListener);
...
class RadioListener implements ActionListener ... {
    public void actionPerformed(ActionEvent e) {
        picture.setIcon(new ImageIcon("images/"
                                      + e.getActionCommand()
                                                                  + ".gif"));
    }
}
```

Para cada grupo de botones de radio, se necesita crear un ejemplar de **ButtonGroup** y añadirle cada uno de los botones de radio. El **ButtonGroup** tiene cuidado de desactivar la selección anterior cuando el usuario selecciona otro botón del grupo.

Generalmente se debería inicializar un grupo de botones de radio para que uno de ellos esté seleccionado. Sin embargo, el API no fuerza esta regla -- un grupo de botones de radio puede no tener selección inicial. Una vez que el usuario hace una selección, no existe forma para desactivar todos los botones de nuevo.

El API Radio Button

Puedes ver El API Button para informaión sobre el API de **AbstractButton** del que descienden **JRadioButton** y **JRadioButtonMenuItem**. Los métodos de **AbstractButton** que más se utilizan son **setMnemonic**, **addItemListener**, **setSelected**, y **isSelected**. Las piezas más utilizadas del API de Radio Button se dividen en dos grupos.

Métodos y Constructores más utilizados de ButtonGroups

Método	Propósito
ButtonGroup()	Crea un ejemplar de **ButtonGroup**.
void add(AbstractButton) **void remove(AbstractButton)**	Añade un botón a un grupo, o elimina un botón de un grupo.

Constructores de RadioButton

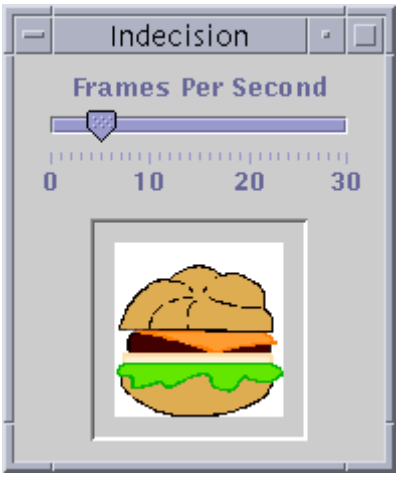
Constructor	Propósito
JRadioButton(String) **JRadioButton(String, boolean)** **JRadioButton(Icon)** **JRadioButton(Icon, boolean)** **JRadioButton(String, Icon)** **JRadioButton(String, Icon, boolean)** **JRadioButton()**	Crea un ejemplar de **JRadioButton**. El argumento string especifica el texto, si existe, que debe mostrar el botón de radio. Similarmente, el argumento, **Icon** especifica la imagen que debe usar en vez la imagen por defecto de un botón de radio para el aspecto y comportamiento. Si se especifica **true** en el argumento booleano, inicializa el botón de radio como seleccionado, sujeto a la aprovación del objeto **ButtonGroup**. Si el argumento booleano esta ausente o es **false**, el botón de radio está inicialmente deseleccionado.
JRadioButtonMenuItem(String) **JRadioButtonMenuItem(Icon)**	Crea un ejemplar de **JRadioButtonMenuItem**. Los argumentos se interpretan de la misma forma que los de los constructores de **JRadioButton**.

JRadioButtonMenuItem(String, Icon)	
JRadioButtonMenuItem()	

¿Cómo Usar Slider?

Se utiliza un **JSlider** para permitir que el usuario introduzca un valor numérico limitado por una valor máximo y un valor mínimo. Mediante la utilización de un Slider en vez de text field, se eliminan errores de entrada.

Aquí tenemos una imagen de una aplicación que utiliza un Slider para controlar la velocidad de una animación.



Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente es **SliderDemo.java**. También necesitarás las 16 imágenes del directorio **example-swing/images** que empiezan con "**burger**".
2. Utiliza el Slider para ajustar la velocidad de animación.

Aquí está el código de **SliderDemo.java** que crea el Slider el programa anterior.

```
JSlider framesPerSecond = new JSlider(JSlider.HORIZONTAL, 0, 30, FPS_INIT);
framesPerSecond.addChangeListener(new SliderListener());
framesPerSecond.setMajorTickSpacing(10);
framesPerSecond.setMinorTickSpacing(1);
framesPerSecond.setPaintTicks(true);
framesPerSecond.setPaintLabels(true);
framesPerSecond.setBorder(BorderFactory.createEmptyBorder(0,0,10,0));
....
//add the slider to the content pane
contentPane.add(framesPerSecond);
```

Por defecto, el espacio para las marcas mayores y menores es cero. Para ver estas marcas, debemos especificar el espaciado de los ticks mayor o menor (o ambos) a un valor distinto de cero y llamar a **setPaintTicks(true)** (llamar sólo a **setPaintTicks(true)** no es suficiente). Para salidas estándard, las etiquetas numéricas en la posición marcas mayores seleccionan el mayor espaciado entre marcas, luego se llama **setPaintLabels(true)**. El programa de ejemplo proporciona la etiqueta para sus deslizadores de esta forma. Sin embargo, las etiquetas del Slider son altamente configurables. Puedes ver un ejemplo en Proporcionar Etiquetas para Deslizadores.

Cuando se mueve el deslizador, se llama al método **stateChanged** del **ChangeListener** del deslizador, cambiando la velocidad de la animación.

```
class SliderListener implements ChangeListener {
    public void stateChanged(ChangeEvent e) {
        JSlider source = (JSlider)e.getSource();
        if (!source.getValueIsAdjusting()) {
                int fps = (int)((JSlider)e.getSource()).getValue();
                if (fps == 0) {
                    if (!frozen) stopAnimation();
                } else {
                    delay = 1000 / fps;
                    timer.setDelay(delay);
                    if (frozen) startAnimation();
                }
        }
    }
}
```

Si movemos el deslizador hasta cero, la animación se para.

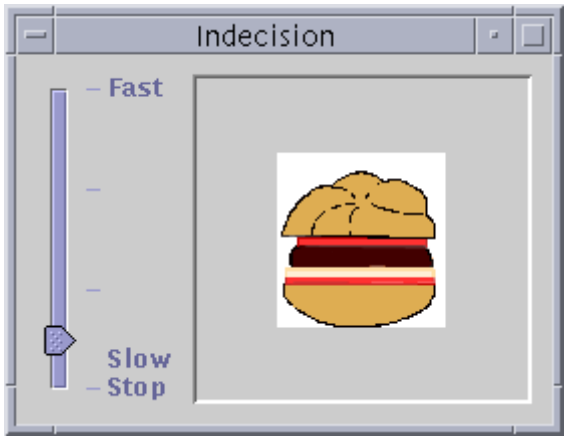
Observa que el método **stateChanged** sólo cambia la velocidad de la animación si **getValueIsAdjusting** devuelve **false**. Muchos eventos change se disparan cuando el usuario mueve el deslizador. Este programa sólo está interesado en el resultado final de la acción del usuario.

Porporcionar Etiquetas para Deslizadores

Para mostrar etiquetas en un deslizador, debemos llamar a **setPaintLabels(true)** y proporcionar un conjunto de etiquetas que indiquen las posiciones y valores para cada etiqueta. Las etiquetas pueden especificarse utilizando una de las siguientes técnicas.

1. Llamar a **setMajorTickSpacing** con un valor distinto de cero. Haciéndolo de esta manera, las etiquetas idendifican el valor de cada marca de pulsación mayor. Esta es la técnica utiliza por **SliderDemo.java**.
2. Crear un **Hashtable** que contenga el valor para cada etiqueta y su posición. Se proporciona el **Hashtable** como un argumento a **setLabelTable**.

SliderDemo2.java, utiliza esta técnica.



Aquí está el código de **SliderDemo2.java** que crea el deslizador.

```
//Create the slider
JSlider framesPerSecond = new JSlider(JSlider.VERTICAL, 0, 30, FPS_INIT);
framesPerSecond.addChangeListener(new SliderListener());
framesPerSecond.setMajorTickSpacing(10);
framesPerSecond.setPaintTicks(true);

//Create the label table
Dictionary labelTable = new Hashtable();
labelTable.put( new Integer( 0 ), new JLabel("Stop") );
labelTable.put( new Integer( 3 ), new JLabel("Slow") );
labelTable.put( new Integer( 30 ), new JLabel("Fast") );
framesPerSecond.setLabelTable( labelTable );

framesPerSecond.setPaintLabels(true);
framesPerSecond.setBorder(BorderFactory.createEmptyBorder(0,0,0,10));
```

Este código crea explícitamente un **Hashtable** y lo rellena con los valores de las etiquetas y sus posiciones. Cada valor de etiqueta debe ser un **Component** y en este programa, son simples etiquetas de texto. También podemos utilizar etiquetas con iconos. Si queremos etiquetas numéricas posicionadas a intervalor específicos, podemos utilizar el método **createStandardLabels** de **JSlider**.

El API Slider

Las siguiente tablas listan los métodos y constructores más utilizados de **JSlider**. Otros métodos interesantes son definidos por las clases **JComponent** y **Component**.

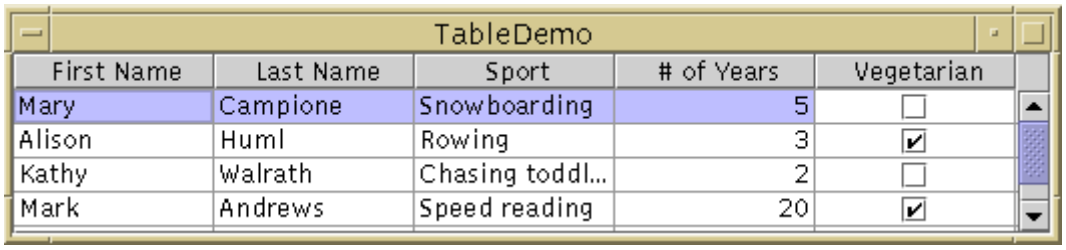
Ajuste fino de la Apariencia del Deslizador

Método	Propósito
void setValue(int) **int getValue()**	Seleciona u obtiene el valor actual del Slider. El marcador del deslizador está en esta posición.
void setOrientation(int) **int getOrientation()**	Seleciona u obtiene la orientación del Slider. Los posibles valores son JSlider.HORIZONTAL o JSlider.VERTICAL
void setInverted(boolean) **boolean getInverted()**	Seleciona u obtiene si el máximo se muestra a la izquierda en un deslizador horizontal o abajo en uno vertical, por lo tanto invierte el rango del deslizador.

void setMinimum(int) **int getMinimum()** **void setMaximum(int)** **int getMaximum()**	Seleciona u obtiene los valores máximos o mínimos del deslizador. Juntos selecionan u obtienen el **rango** del deslizador.
void setMajorTickSpacing(int) **int getMajorTickSpacing()** **void setMinorTickSpacing(int)** **int getMinorTickSpacing()**	Seleciona u obtiene el rango entre marcas mayores y menores. Debemos llamar a **setPaintTicks(true)** para que aparezcan las marcas.
void setPaintTicks(boolean) **boolean getPaintTicks()**	Seleciona u obtiene si se dibujan las marcas en el deslizador.
void setLabelTable(Dictionary) **Dictionary getLabelTable()**	Seleciona u obtiene las etiquetas para el deslizador. Debemos llamar a **setPaintLabels(true)** para que aparezcan las etiquetas. **createStandardLabels** es un método de conveniencia para crear un conjunto de etiquetas estándard.
void setPaintLabels(boolean) **boolean getPaintLabels()**	Seleciona u obtiene si se dibujan las etiquetas de un deslizador. Las etiquetas se seleccionan con **setLabelTable** o configurando el espaciado entre marcas mayores.

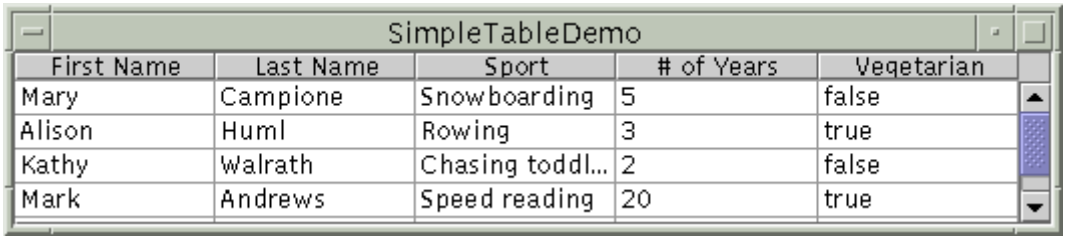
¿Cómo Usar Table?

Con la clase **JTable**, se pueden mostrar tablas de datos, y opcionalmente permitir que el usuario los edite. **JTable** no contiene ni almacena datos; simplemente es una vista de nuestros datos. Aquí podemos ver una tabla típica mostrada en un ScrollPane.



El resto de esta página explica como realizar algunas de las tareas más comunes relacionadas con las tablas. Aquí están los tópicos cubiertos por está página.

Crear una Tabla Sencilla



Intenta esto:

1. Compila y ejecuta la aplicación. el fichero fuente es **SimpleTableDemo.java**.

2. Pulsa sobre la celda que contiene "Snowboarding".

   Se selecciona toda la primera fila, indicando que has seleccionado los datos de Mary Campione. Una especial iluminación indica que la celda "Snowboarding" es editable. Generalmente, se empieza a editar una celda de texto haciendo doble-click en ella.

3. Posiciona el cursor sobre "First Name". Ahora pulsa el botón del ratón y arrástrala hacia la derecha.

   Como puedes ver, los usuarios pueden modificar las columnas en las tablas.

4. Posiciona el cursor justo a la derecha de una columna de cabecera. Ahora pulsa el botón del ratón y arrastralo a derecha o izquierda.

   La columna cambia su tamaño, y las demás columnas se ajustan para rellenar el espacio sobrante.

5. Redimensiona la ventana que contiene la tabla para que sea tan grande como para contener la tabla completa.

   Todas las celdas de la tabla se agrandan, expandiéndose para llenar el espacio extra.

Aquí está el código que implementa la tabla en **SimpleTableDemo.java**.

```
Object[][] data = {
    {"Mary", "Campione",
     "Snowboarding", new Integer(5), new Boolean(false)},
    {"Alison", "Huml",
     "Rowing", new Integer(3), new Boolean(true)},
    {"Kathy", "Walrath",
     "Chasing toddlers", new Integer(2), new Boolean(false)},
    {"Mark", "Andrews",
     "Speed reading", new Integer(20), new Boolean(true)},
    {"Angela", "Lih",
     "Teaching high school", new Integer(4), new Boolean(false)}
};

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

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

El ejemplo SimpleTableDemo utiliza uno de los constructores de **JTable** que aceptan datos directamente.

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

La ventaja de utilizar uno de estos constructores es que es sencillo. Sin embargo, estos constructores también tienen desventajas.

- Automáticamente hacen todas las celdas editables.
- Tratan igual a todos los tipos de datos. Por ejemplo, si una columna tiene datos **Boolean**, los datos pueden mostrarse como un CheckBox en la tabla. Sin embargo, si especificamos los datos como un argumento array o vector del constructor de **JTable**, nuestro dato **Boolean** se mostrará como un string. Se pueden ver estas diferencias en las dos últimas columnas de los ejemplos ateriores.
- Requieren que pongamos todos los datos de la tabla en un array o vector, lo que es inapropiado para algunos datos. Por ejemplo, si estamos ejemplarizando un conjunto de objetos desde una base de datos, podríamos quere pedir los objetos directamente por sus valores, en vez de copiar todos los valores en un array o un vector.

Si queremos evitar estas restricciones, necesitamos implementar nuestro propio modelo de tabla como se describe en Crear un Modelo de Tabla.

Añadir una Tabla a un Contenedor

Es fácil poner una tabla en un ScrollPane. Sólo necesitamos escribir una o dos líneas de código.

```
JScrollPane scrollPane = new JScrollPane(table);
table.setPreferredScrollableViewportSize(new Dimension(500, 70));
```

El ScrollPane obtiene automáticamente las cabeceras de la tabla, que muestra los nombres de las columnas, y los pone en la parte superior de la tabla. Incluso cuando el usuario se desplaza hacia abajo, los nombres de columnas permanecen visibles en la parte superior del área de visión. El ScrollPane también intenta hacer que su área de visión sea del mismo tamaño que el tamaño preferido de la tabla. El código anterior selecciona el tamaño preferido de la tabla con el método **setPreferredScrollableViewportSize**.

Nota: Antes de Swing 1.0.2, el ScrollPane no podía obtener las cabeceras de la tabla a menos que se utilizara el método **JTable.createScrollPaneForTable** para crearlo. Aquí tenemos unos ejemplos de código recomendado, antes

y después de Swing 1.0.2.

```
//1.0.1 code (causes deprecation warning in 1.0.2 and later releases).
scrollPane = JTable.createScrollPaneForTable(table);

//Recommended code (causes missing column names in 1.0.1).
scrollPane = new JScrollPane(table);
```

Si estamos utilizando una tabla sin un ScrollPane, debemos obtener los componentes de cabecera de la tabla y situarlos nosotros mismos. Por ejemplo.

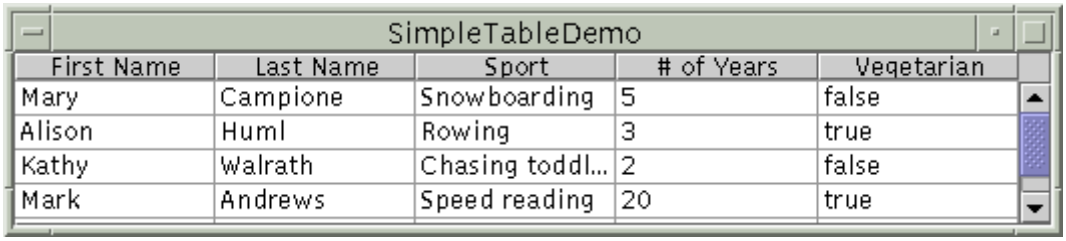
```
container.setLayout(new BorderLayout());
container.add(table.getTableHeader(), BorderLayout.NORTH);
container.add(table, BorderLayout.CENTER);
```

Seleccionar y Cambiar la Anchura de las Columnas

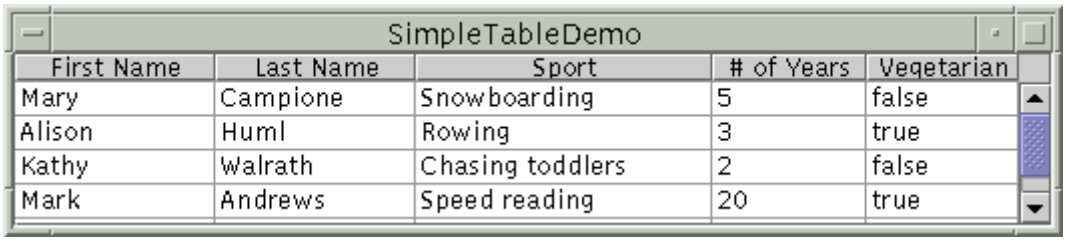
Por defecto, todas las columnas de una tabla empiezan con la misma anchura, y las columnas rellenan automáticamente la anchura completa de la tabla. Cuando la tabla se ensancha o se estrecha (lo que podría suceder cuando el usuario redimensiona la ventana que la contiene) la anchura de todas las columnas cambia apropiadamente.

Cuando el usuario redimensiona una columna, arrastando su borde derecho, todas las demás deben cambiar su tamaño. Por defecto, el tamaño de la tabla permace igual, y todas las columnas a la derecha del punto de arrastre acomodan su tamaño al espacio añadido o eliminado desde la columna situada a la izquierda del punto de arrastre.

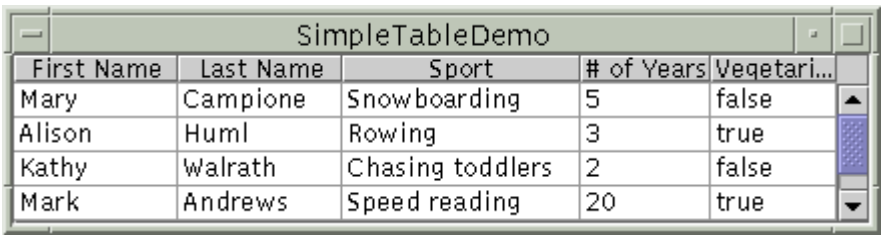
Las siguientes figuras ilustran el comportamiento de redimensionado por defecto.



Inicialmente, las columnas tienen la misma anchura.



Cuando el usuario redimensiona una columna, alguna de las otras columnas debe ajustar su tamaño para que el tamaño de la tabla permanezca igual.



Cuando se redimensiona toda la tabla, todas las columnas se redimensionan.

Para personalizar la anchura inicial de las columnas, podemos llamar al método **setPreferredWidth** con cada una de las columnas de la tabla. Este selecciona tanto las anchuras preferidas de las clumnas como sus anchuras relativas aproximadamente. Por ejemplo, si añadimos el siguiente código a SimpleTableDemo haremos que la tercera columna se mayor que las otras.

```
TableColumn column = null;
for (int i = 0; i < 5; i++) {
    column = table.getColumnModel().getColumn(i);
    if (i == 2) {
        column.setPreferredWidth(100); //sport column is bigger
    } else {
```

```
        column.setPreferredWidth(50);
    }
}
```

Nota: el método **setPreferredWidth** fue primero introducido en Swing 1.1 beta 2. Para versiones anterior debemos utilizar **setMinWidth**, asegurándonos de llamarlo sobre cada columna, (de otro modo, las columnas que no lo hagamos serán muy finas).

Como muestra el código anterior, cada columna de la tabla está representada por un objeto **TableColumn**. Junto a **setPreferredWidth**, **TableColumn** también suministra métodos para obtener la anchura mínima, máxima y actual de una columna. Para un ejemplo de selección de anchura de celdas basada en la cantidad de espacio necesario para mostrar el contenido de las celdas, puedes ver el método **initColumnSizes** en **TableRenderDemo.java**, que se explica en Mayor personalización del Visionado y del manejo de Eventos.

Cuando el usuario redimensiona explícitamente las columnas, los nuevos tamaños no sólo se convierten en la anchura **actual** de la columna, sino que también se convierten en la anchura **preferida**. Si embargo, cuando las columnas se redimensionan como resultado de un cambio de anchura de la tabla, las anchuras preferidas de las columnas no cambian.

Podemos cambiar el comportamiento de redimensionado de una tabla llamando al método **setAutoResizeMode**. El argumento de este método debe ser uno de estos valores (definidos como constantes en **JTable**).

AUTO_RESIZE_SUBSEQUENT_COLUMNS
: Por defecto. Además de redimensionar la columna a la izquierda del punto de arrastre, ajusta los tamaños de todas las columnas situadas a la derecha del punto de arrastre.

AUTO_RESIZE_NEXT_COLUMN
: Ajusta sólo las columnas inmediatas a la izquierda y derecha del punto de arrastre.

AUTO_RESIZE_OFF
: Ajusta el tamaño de la tabla.

Nota: Antes de la versión Swing 1.1 Beta, el modo por defecto era **AUTO_RESIZE_ALL_COLUMNS**. Sin embargo, este modo no es intuitivo por eso se cambió el modo por defecto a: **AUTO_RESIZE_SUBSEQUENT_COLUMNS**.

Detectar Selecciones de Usuario

El siguiente fragmento de código muestra cómo detectar cuando el usuario selecciona una fila de la tabla. Por defecto, una tabla permite que el usuario selecciona varias filas -- no columnas o celdas individuales -- y las filas seleccionadas deben ser contiguas. Utilizando el método **setSelectionMode**, el siguiente código especifica que sólo se puede seleccionar una fila cada vez. Puedes encontrar el programa completo en**SimpleTableSelectionDemo.java**.

```
table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
...
ListSelectionModel rowSM = table.getSelectionModel();
rowSM.addListSelectionListener(new ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
        ListSelectionModel lsm = (ListSelectionModel)e.getSource();
        if (lsm.isSelectionEmpty()) {
            ./no rows are selected
        } else {
            int selectedRow = lsm.getMinSelectionIndex();
            ./selectedRow is selected
        }
    }
});
```

SimpleTableSelectionDemo también tiene código (no incluido en el fragmento anterior) que cambia la orientación de la selección de la tabla. Cambiando un par de valores booleanos, podemos permitir que la tabla acepte selecciones de columnas o de celdas individuales en vez de seleccionar filas.

Para más información y ejemplos de implementación de selecciones, puedes ver Escribir un Oyente de List Selection.

Crear un Modelo de tabla

Como se ha visto, toda tabla obtiene sus datos desde un objeto que implemente el interface **TableModel**.

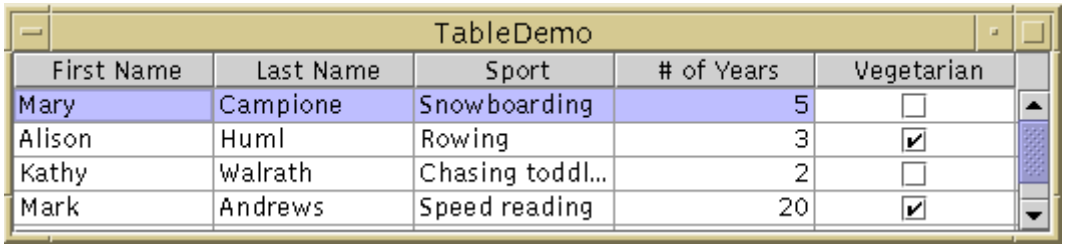
El constructor de **JTable** usado por SimpleTableDemo crea su modelo de tabla con este código.

```
new AbstractTableModel() {
    public String getColumnName(int col) {
        return columnNames[col].toString();
    }
    public int getRowCount() { return rowData.length; }
    public int getColumnCount() { return columnNames.length; }
    public Object getValueAt(int row, int col) {
        return rowData[row][col];
    }
    public boolean isCellEditable(int row, int col) { return true; }
    public void setValueAt(Object value, int row, int col) {
        rowData[row][col] = value;
                fireTableCellUpdated(row, col);
    }
}
```

Cómo muestra el código anterior, implementar un modelo de tabla puede ser sencillo. Generalmente, se implementa en una subclase de la clase **AbstractTableModel**.

Nuestro modelo podría contener sus datos en un array, un vector o un hashtable, o podría obtener los datos desde una fuente externa como una base de datos. Incluso podría generar los datos en tiempo de ejecución.

Aquí está de nuevo una imagen de una tabla implementada por TableDemo, que tiene un modelo de tabla personalizado.



Esta tabla es diferente de la de SimpleTableDemo en estas cosas.

- El modelo de tabla de SimpleTableDemo, habiendo sido creado automáticamente por **JTable**, no es suficentemente inteligente para saber que la columna '# of Years' contiene números (que generalmente deben alinearse a la derecha). Tampoco sabe que la columna 'Vegetarian' contiene un valor booleano, que pueden ser represantados por checkboxes. El modelo de datos personalizado de TableDemo, aunque sencillo, puede determinar los tipos de datos, ayudando a **JTable** a mostrar los datos en el mejor formato.
- En SimpleTableDemo, todas las celdas son editables. En TableDemo, hemos implementado el modelo de tabla personalizado, para que no permita editar la columna 'name' pero, sin embargo, si se pueden editar las otras columnas.

Aquí está el código de **TableDemo.java** que es diferente del código de **SimpleTableDemo.java**. Las partes en negrita indican el código que hace que este modelo de tabla sea diferente del modelo de tabla definido automáticamente por SimpleTableDemo.

```
public TableDemo() {
    ...
    MyTableModel myModel = new MyTableModel();
    JTable table = new JTable(myModel);
    table.setPreferredScrollableViewportSize(new Dimension(500, 70));

    //Create the scroll pane and add the table to it.
    JScrollPane scrollPane = new JScrollPane(table);

    //Add the scroll pane to this window.
    setContentPane(scrollPane);
    ...
}

class MyTableModel extends AbstractTableModel {
    final String[] columnNames = ./same as before...
    final Object[][] data = ./same as before...

    public int getColumnCount() {
        return columnNames.length;
    }

    public int getRowCount() {
        return data.length;
    }

    public String getColumnName(int col) {
        return columnNames[col];
    }

    public Object getValueAt(int row, int col) {
        return data[row][col];
    }

    public Class getColumnClass(int c) {
        return getValueAt(0, c).getClass();
    }

    /*
     * Don't need to implement this method unless your table's
     * editable.
     */
    public boolean isCellEditable(int row, int col) {
        //Note that the data/cell address is constant,
        //no matter where the cell appears onscreen.
        if (col < 2) {
            return false;
        } else {
            return true;
        }
    }

    /*
     * Don't need to implement this method unless your table's
     * data can change.
     */
    public void setValueAt(Object value, int row, int col) {
        ./debugging code not shown...
        ./ugly class cast code for Integers not shown...
        data[row][col] = value;
        ./debugging code not shown...
    }
    ...
```

Detectar Cambios de Datos

Una tabla y su modelo detectan automáticamente si el usuario edita los datos de la tabla. Sin embargo, si los datos cambian por otra razón, debemos realizar unos pasos especiales para indicar a la tabla y a su modelo el cambio de los datos. Si no implementamos un

modelo de tabla, como en SimpleTableDemo, también debemos realizar los pasos especiadl para ver cuando el usuario edita los datos de la tabla.

Para disparar un evento table-model, el modelo llama al método **fireTableRowsInserted**, que está definido por la clase **AbstractTableModel**. Otros métodos **fireXxxx** que define la clase **AbstractTableModel** para ayudarnos a lanzar eventos table-model son **fireTableCellUpdated**, **fireTableChanged**, **fireTableDataChanged**, **fireTableRowsDeleted**, **fireTableRowsInserted**, **fireTableRowsUpdated**, y **fireTableStructureChanged**.

Si tenemos una clase como **SimpleTableDemo** que es una tabla o un modelo de tabla, pero necesita reaccionar a los cambios en un modelo de tabla, necesitamos hacer algo especial para detectar cuando el usuario edita los datos de la tabla. Específicamente, necesitamos registrar un oyente de table-model. Añadiendo el código en negrita del siguiente fragmento hace que SimpleTableDemo reaccione ante los cambios de datos.

```
public class SimpleTableDemo ... implements TableModelListener {
    ...
    public SimpleTableDemo() {
        ...
              model = table.getModel();
        model.addTableModelListener(this);
        ...
    }

    public void tableChanged(TableModelEvent e) {
        ...
        int row = e.getFirstRow();
        int column = e.getColumn();
        String columnName = model.getColumnName(column);
        Object data = model.getValueAt(row, column);

        ./ Do something with the data...
    }
    ...
}
```

Conceptos: Editores de Celdas e Intérpretes

Antes de seguir adelante, necesitamos comprender como las tablas dibujan sus celdas. Podríamos esperar que cada celda de la tabla fuera un componente. Sin embargo, por razones de rendimiento, las tablas Swing no están implementadas de esta forma.

En su lugar, se utiliza un sencillo **intérprete de celdas** para dibujar todas las celdas de una columna. Frecuentemente este intérprete es compartido entre todas las columnas que contienen el mismo tipo de datos. Podemos pensar en el intérprete como un sello de caucho que las tablas utilizan para estampar los datos formateados apropiadamente en cada celda. Cuando el usuario empieza a editar los datos de una celta, un **editor de celdas** toma el control sobre la edición de la celda.

Por ejemplo, todas las celdas de la columna '# of Years' de TableDemo contienen datos numéricos -- específicamente un objeto **Integer**. Por defecto, el intérprete para una columna numérica utiliza un sencillo ejemplar de **JLabel** para dibujar los números apropiados, alineados a la derecha, en las celdas de la columna. Si el usuario empieza a editar una de las celdas, el editor por defecto utiliza un **JTextField** alineado a la derecha para controlar la edición.

Para elegir el intérprete que muestra la celdas de una columna, una tabla primero determina si hemos especificado un ínterprete para esa columna particular. Si no lo hacemos, la tabla llama al método **getColumnClass** del modelo de tabla, que obtiene el tipo de datos de las celdas de la columna. Luego, la tabla compara el tipo de datos de la columna con una lista de tipos de datos para los que hay registrados unos intérpretes. Esta lista es inicializada por la tabla, pero podemos añadirle elementos o cambiarla. Actualmente, las tablas ponen los siguientes tipos en la lista.

- **Boolean** -- interpretado con un checkbox.
- **Number** -- interpretado con una etiqueta alineada a la derecha.
- **ImageIcon** -- interpretado por una etiqueta centrada.
- **Object** -- interpretado por una etiqueta que muestra el valor String del objeto.

Las tablas eligen sus editores de celdas utilizando un algoritmo similar.

Recuerda que si dejamos que una tabla cree su propio modelo, utiliza **Object** como el tipo de cada columna. **TableDemo.java** muestra como especificar los tipos de datos de las columnas con más precisión.

Las siguientes secciones explican cómo pesonalizar el visionado y edición de celdas especificando intérpretes y editores de celdas para columnas o tipos de datos.

Validar el Texto Introducido por el Usuario

En los ejemplos de tablas que hemos visto hasta ahora, el usuario podía introducir cualquier texto en la columna '# of Years'. SimpleTableDemo no comprueba el valor de los datos. El ejemplo TableDemo está ligeramente mejorado en que cuando el usuario hace la edición, el código comprueba si la entrada puede ser pasada a un entero. Sin embargo, TableDemo debe usar un código más ambicioso para convertir el string devuelto por el editor de celdas por defecto en un **Integer**. Si no hace la conversión, el tipo real de los datos podría cambiar de **Integer** a **String**.

Lo que realmente queremos hacer es comprobar la entrada del usuario **mientras** la está tecleando, y hacer que el editor de celdas devuelva un **Integer** en lugar de un string. Podemos conseguir una o estas dos tareas utilizando un campo de texto personalizado para controlar la edición de la celda.

Un campo de texto personalizado, puede chequear la entrada del usuario mientras la está tecleando o después de que haya indicado el final (pulsado la tecla return, por ejemplo). Llamamos a estos tipos de validación, chequeo de pulsación y chequeo de acción, respectivamente.

El siguiente código, tomado de **TableEditDemo.java**, configura un campo de texto con chequeo de pulsación. Las líneas en negrita hacen que el campo de texto sea el editor para todas las celdas que contengan datos del tipo **Integer**.

```
final WholeNumberField integerField = new WholeNumberField(0, 5);
integerField.setHorizontalAlignment(WholeNumberField.LEFT);

DefaultCellEditor integerEditor =
    new DefaultCellEditor(integerField) {
        //Override DefaultCellEditor's getCellEditorValue method
        //to return an Integer, not a String.
        public Object getCellEditorValue() {
            return new Integer(integerField.getValue());
        }
    };
table.setDefaultEditor(Integer.class, integerEditor);
```

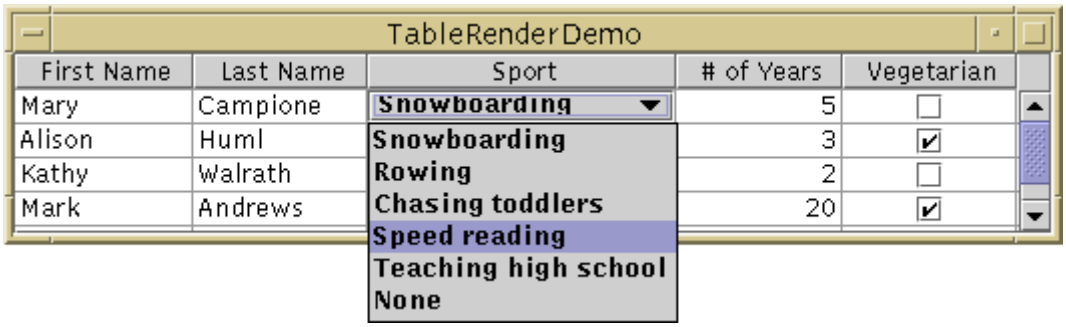
La clase **WholeNumberField** utilizada arriba es una subclase de **JTextField** personalizada que permite al usuario introducir sólo dos dígitos. El método **getValue** devuelve el valor **int** del contenido de **WholeNumberField** Puedes ver Cómo usar TextField para más información sobre **WholeNumberField**. Esta página también proporciona un textfield con validación de propósito general, llamado **DecimalField**, que podemos personalizar para validar cualquier formato de número que le especifiquemos.

Usar un ComboBox como un Editor

Aquí hay un ejemplo de configuración de un ComboBox como editor de celda. Las líneas en negrita del código configuran el ComboBox para editar una columna, en vez de para un tipo de dato específico.

```
TableColumn sportColumn = table.getColumnModel().getColumn(2);
...
JComboBox comboBox = new JComboBox();
comboBox.addItem("Snowboarding");
comboBox.addItem("Rowing");
comboBox.addItem("Chasing toddlers");
comboBox.addItem("Speed reading");
comboBox.addItem("Teaching high school");
comboBox.addItem("None");
sportColumn.setCellEditor(new DefaultCellEditor(comboBox));
```

Aquí hay una imagen el editor ComboBox en uso.



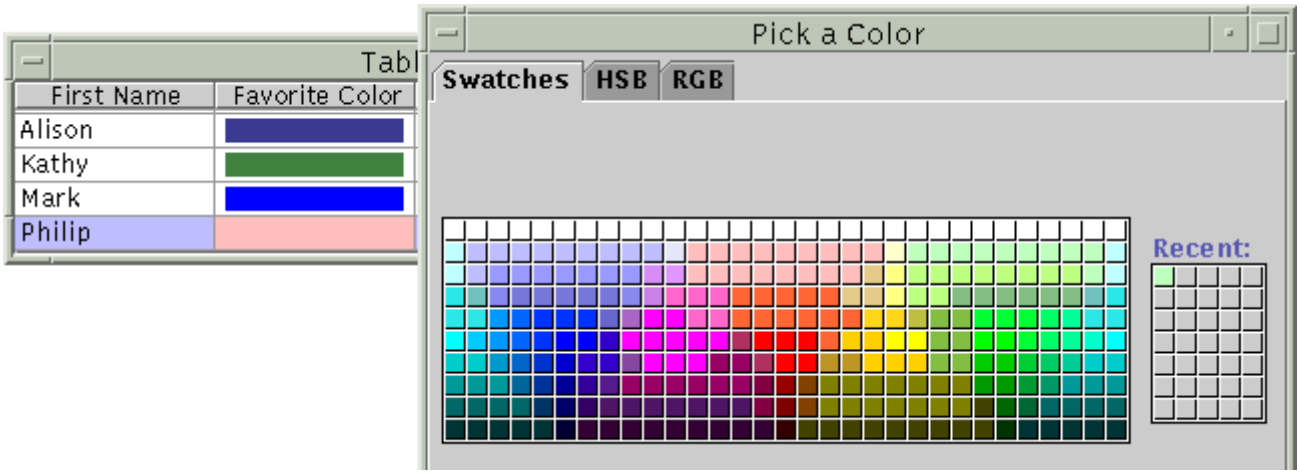
El editor ComboBox está implementado en **TableRenderDemo.java**, que se explica en más detalle en Mayor Personalización de Visionado y de Manejo de Eventos.

Especificar otros Editores

Como se vió en la sección anterior, podemos seleccionar un editor para una columna utilizando el método **setCellEditor** de **Tablecolum**, o para un tipo de datos específico usando el método **setDefaultEditor** de **JTable**. Para ambos métodos, podemos especificar un argumento que implemente el interface **TableCellEditor**. Afortunadamente, la clase **DefaultCellEditor** implementa este interface y proporciona constructores para permitir especificar y editar componentes que sean **JTextField**, **JCheckBox**, o **JComboBox**. Normalmente no tenemos que especificar explícitamente un checkbox como un editor, ya que las columnas con datos **Boolean** automáticamente usan un editor y un intérprete CheckBox.

¿Y si queremos especificar un editor que no sea un textfield, checkbox, o combobox? Bien, como **DefaultCellEditor** no soporta otros tipos de componentes, debemos trabajar un poco más. Necesitamos crear una subclase del editor de componente deseado, y esta subclase debe implementar el interface **TableCellEditor**. Luego configuramos el componente como un editor para un tipo de dato o una columna, utilizando los métodos **setDefaultEditor** o **setCellEditor**, respectivamente.

Aquí hay una imagend e una tabla con un diálogo que sirve, indirectamente, como editor de celda. Cuando el usuario empieza a editar una celda en el columna, 'Favorite Color', un botón, (el verdadero editor de celda) aparece y trae el diálogo, con el que el usuario puede elegir un color diferente.



Podemos encontrar el código fuente en **TableDialogEditDemo.java**. el ejemplo también necesita **WholeNumberField.java**.

Mayor Personalización de Visionado y de Manejo de Eventos

Ya hemos visto como especificar **editores** de celdas. También podemos especificar **intérpretes** para celdas y para cabeceras de columnas. Los intérpretes personalizados permiten mostrar datos de formas personalizadas y especificar texto de ayuda (tooltips) para que lo muestre la tabla.

Aunque los intérpretes determinan cómo se muestran las celdas o cabeceras de columnas, no pueden manejar eventos. Para detectar los eventos que tienen lugar dentro de una tabla, deberíamos elegir la técnica apropiada para ordenar el evento en el que estamos interesados. Para una celda que esta siendo editada, el editor debería procesar eventos. Para detectar selecciones y deselecciones de fila/columna/celda, se utiliza un oyente de selection como se describe en Detectar Selecciones del Usuario. Para editar pulsaciones del ratón en una cabecera de columna, podemos registrar un oyente de mouse en la cabecera de la tabla. (Puedes ver un ejemplo en **TableSorter.java**). Para detectar otros eventos, podemos registrar el oyente apropiado sobre el objeto **JTable**.

Crear un intérprete personalizado puede ser tan sencillo como crear una subclse de un componente existente y luego implementar el único método del interface **TableCellRenderer**. En la figura anterior, el intérprete de color utilizado para la celda "Favorite Color" es una subclase de **JLabel**. Podemos encontrar el código del intérprete en la clase interna de **TableDialogEditDemo.java**. Aquí está el código que registra el ejemplar de **ColorRenderer** como el intérprete por defecto para todos los datos **Color**.

```
table.setDefaultRenderer(Color.class, new ColorRenderer(true));
```

Podemos especificar un intérprete especifico de celda, si queremos. Para hacer esto, nocesitamos definir una subclase de **JTable** que sobreescriba el método **getCellRenderer**. Por ejemplo, el siguiente código hace que la primera celda de la primera columna de la tabla use un intérprete personalizado.

```
TableCellRenderer weirdRenderer = new WeirdRenderer();
table = new JTable(...) {
    public TableCellRenderer getCellRenderer(int row, int column) {
                if ((row == 0) && (column == 0)) {
                    return weirdRenderer;
                }
                // else...
                return super.getCellRenderer(row, column);
    }
};
```

Para añadir tool-tips a las celdas o columnas de cabecera, necesitamos obtener y crear el intérprete de celda o cabecera y luego utilizar el método **setToolTipText** del componente del intérprete. **TableRenderDemo.java** añade tool-tips tanto a las celdas como a la cabecera de la columna 'Sport' con el siguiente código.

```
//Set up tool tips for the sport cells.
DefaultTableCellRenderer renderer =
        new DefaultTableCellRenderer();
renderer.setToolTipText("Click for combo box");
sportColumn.setCellRenderer(renderer);

//Set up tool tip for the sport column header.
TableCellRenderer headerRenderer = sportColumn.getHeaderRenderer();
if (headerRenderer instanceof DefaultTableCellRenderer) {
    ((DefaultTableCellRenderer)headerRenderer).setToolTipText(
            "Click the sport to see a list of choices");
}
```

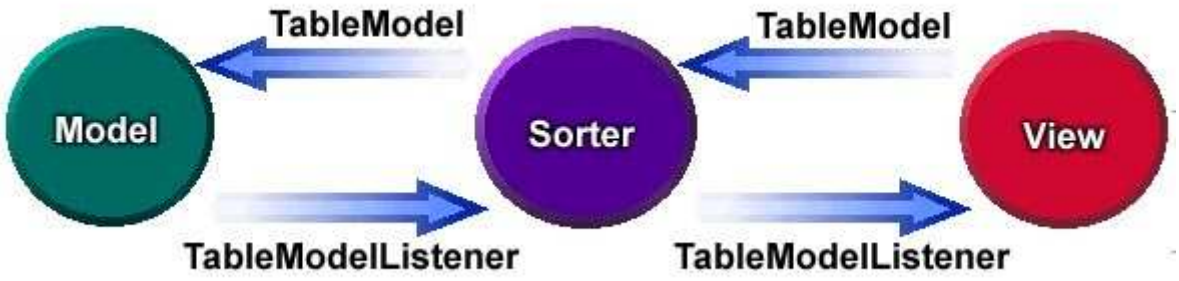
Una interesante característica de TableRenderDemo es cómo determina el tamaño de sus columnas. Por cada columna, TableRenderDemo obtiene los componentes utilizados para intérpretar las celdas y la cabecera de columna. Luego pregunta a los componentes cuánto espacio necesitan. Finalmente, utiliza esa información para seleccionar al anchura de la columna.

```
TableColumn column = null;
Component comp = null;
int headerWidth = 0;
int cellWidth = 0;
Object[] longValues = model.longValues;
```

```
for (int i = 0; i < 5; i++) {
    column = table.getColumnModel().getColumn(i);

    comp = column.getHeaderRenderer().
                  getTableCellRendererComponent(
                      null, column.getHeaderValue(),
                      false, false, 0, 0);
    headerWidth = comp.getPreferredSize().width;

    comp = table.getDefaultRenderer(model.getColumnClass(i)).
                  getTableCellRendererComponent(
                      table, longValues[i],
                      false, false, 0, i);
    cellWidth = comp.getPreferredSize().width;
    ./debugging code not shown...
    column.setPreferredWidth(Math.max(headerWidth, cellWidth));
}

./In the model:
public final Object[] longValues = {"Angela", "Andrews",
                                    "Teaching high school",
                                    new Integer(20), Boolean.TRUE};
```

Ordenación y otras Manipulaciones de Datos

Una forma de realizar una manipulación de datos como la ordenación es utilizar uno o más modelos de tablas especializados (**manipuladores de datos**), además del modelo que proporciona los datos (el modelo de datos). Los manipuladores de datos deberían situarse entre la tabla y el modelo de datos, como muestra la siguiente figura.



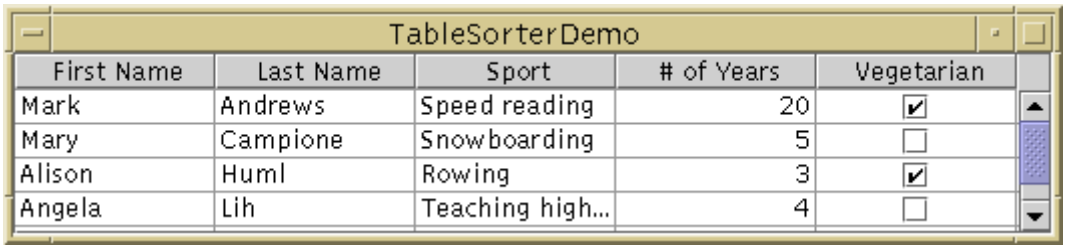
Podemos utilizar las clases **TableMap** y **TableSorter** cuando implementemos nuestro manipulador de datos. **TableMap** implementa **TableModel** y sirve como una superclase para manipuladores de datos.

TableSorter es una subclase de **TableMap** que ordena los datos proporcionados por otro modelo de tabla. También podemos cambiar estas clases, utilizándolas como un base para escribir nuestros propios manipuladores, o utilizarlas tal como son para proporcionar la funcionalidad de ordenación.

Para implementar la ordenación con **TableSort**, necesitamos sólo tres líneas de código. El siguiente listado muestra las diferencias entre TableDemo y su primo ordenado, TableSorterDemo.

```
TableSorter sorter = new TableSorter(myModel); //ADDED THIS
//JTable table = new JTable(myModel);          //OLD
JTable table = new JTable(sorter);             //NEW
sorter.addMouseListenerToHeaderInTable(table); //ADDED THIS
```

El método **addMouseListenerToHeaderInTable** añade un oyente de mouse que detecta pulsaciones sobre las cabeceras de columna. Cuando el oyente detecta un click, ordena las filas básandose en la columna pulsada. Por ejemplo, cuando pulsamos sobre "Last Name", las filas son reordenadas para que la fila con "Andrews" se convierta la primera. Cuando volvemos a pulsar de nuevo la cabecera de columna, las filas se ordenan en orden inverso.



El API Table

Las tablas de esta sección sólo cubren una parte de este API. Para más información puedes ver el API de **JTable** y para distintas clases y paquetes table package. El API para usar tablas se divide en estas categorías.

Clases e Interfaces Relacionados con las Tablas

Clase/Interface	Propósito
JTable	El componente que presenta la tabla al usuario.
JTableHeader	El componente que presenta los nombres de columnas al usuario. Por defecto, la tabla genera este componente automáticamente.
TableModel, **AbstractTableModel**	Respectivamente, el interface que un modelo de tabla debe implementar y la superclase usual para implementaciones de modelos de tabla.
TableCellRenderer, **DefaultTableCellRenderer**	Respectivamente, el interface que un intérprete de celda debe implementar y la implementación más usual.
TableCellEditor, **DefaultCellEditor**	Respectivamente, el interface que debe implementar un editor de celda, y la implementación más usual.
TableColumnModel, **DefaultTableColumnModel**	Respectivamente, el interface que debe implementar un modelo de columna, y su implementación usual. Normalmente no tenemos que tratar directamente con el modelo de columna a menos que necesitemos obtener el modelo de selección de columna u obtener un índice de columna o un objeto.
TableColumn	Controla todos los atributos de una columna de la tabla, incluyendo, redimensionado, anchuras mínima, máxima, preferida y actual; y editor/intérprete opcional específico de la columna.
DefaultTableModel	Un modelo de tabla basado en **Vector** utilizado por **JTable** cuando construimos una tabla sin especificar modelo de datos ni datos.

Crear y Configurar una Tabla

Método/Constructor de JTable	Propósito
JTable(TableModel) **JTable(TableModel, TableColumnModel)** **JTable(TableModel, TableColumnModel, ListSelectionModel)** **JTable()** **JTable(int, int)** **JTable(Object[][], Object[])** **JTable(Vector, Vector)**	Crea una tabla. El argumento opcional **TableModel** especifica el modelo que proporciona los datos de la tabla. Los argumentos opcionales **TableColumnModel** y **ListSelectionModel** permiten especificar el modelo de columna y el modelo de selección. Como una alternativa para especificar un modelo de tabla, podemos suministrar datos y nombres de columnas utilizando un array o un **Vector**. Otra opción es no especificar datos, opcionalmente especificar el número de filas y columnas (ambos enteros) que hayan en la tabla.
void setPreferredScrollableViewportSize(Dimension)	Selecciona el tamaño de la parte visible de la tabla cuando se está viendo dentro de un ScrollPane.
JTableHeader getTableHeader(Dimension)	Obtiene el componente que muestra los nombres de columnas.

Manipular Columnas

Método	Propósito
TableColumnModel getColumnModel() (en **JTable**)	Obtiene el modelo de columna de la tabla.
TableColumn getColumn(int) **Enumeration getColumns()** (en **TableColumnModel**)	Obtiene uno o todos los objetos **TableColumn** de la tabla.
void setMinWidth(int) **void setPreferredWidth(int)** **void setMaxWidth(int)**	Seleciona la anchura mínima, preferida o máxima de la columna.

(en TableColumn)	
int getMinWidth(int) **int getPreferredWidth()** **int getMaxWidth()** **int getWidth()** **(en TableColumn)**	Obtiene la anchura mínima, preferida, máxima o actual de la columna.

Usar Editores e Intérpretes

Métodos	Propósito
void setDefaultRenderer(Class, TableCellRenderer) **void setDefaultEditor(Class, TableCellEditor)** **(en JTable)**	Selecciona el intérprete o editor usado, por defecto, para todas las celdas en todas las columnas que devuelvan el tipo de objetos especificado.
void setCellRenderer(TableCellRenderer) **void setCellEditor(TableCellEditor)** **(en TableColumn)**	Selecciona el intérprete o editor usado para todas las celdas de esta columna.
TableCellRenderer getHeaderRenderer() **(en TableColumn)**	Obtiene el intérprete de cabecera para esta columna, que podemos personalizar.

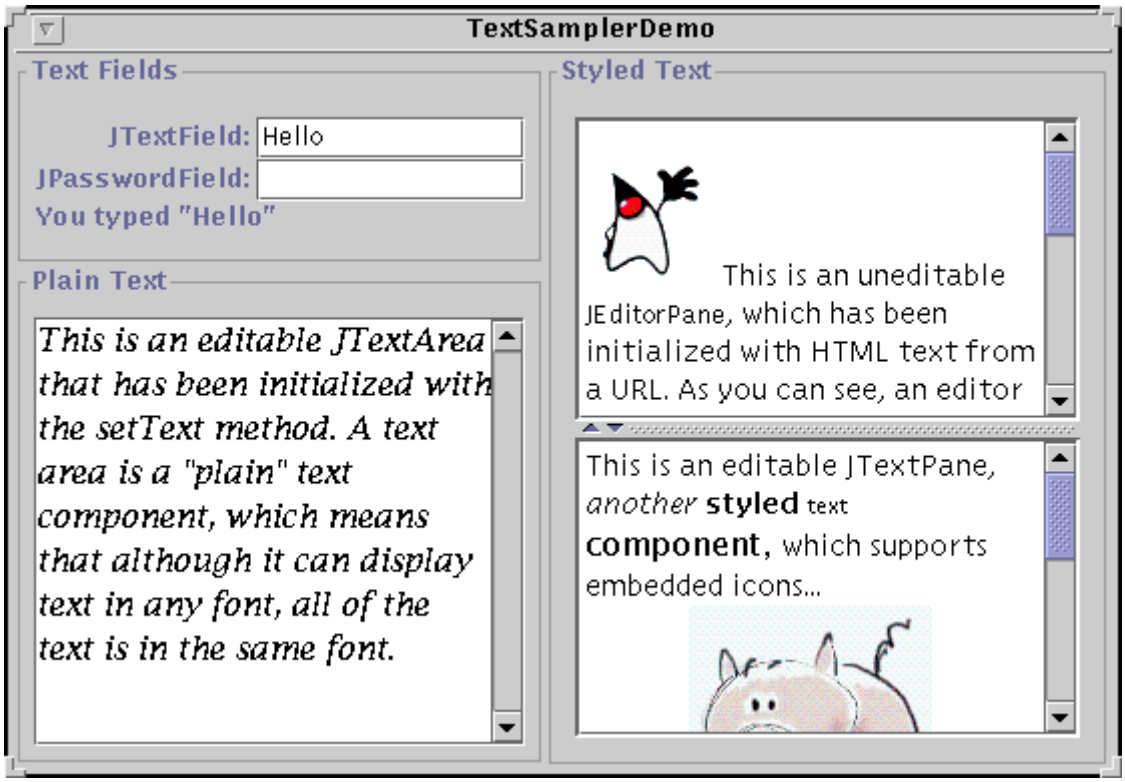
Implementar Selección

Método de JTable	Propósito
void setSelectionMode(int)	Selecciona los intervalos de selección permitidos en la tabla. Los valores válidos están definidos en **ListSelectionModel** como **SINGLE_SELECTION**, **SINGLE_INTERVAL_SELECTION**, y **MULTIPLE_INTERVAL_SELECTION** (por defecto).
void setSelectionModel(ListSelectionModel) **ListSelectionModel getSelectionModel()**	Selecciona u obtiene el modelo usado para controlar las selecciones de filas.
void setRowSelectionAllowed(boolean) **void setColumnSelectionAllowed(boolean)** **void setCellSelectionEnabled(boolean)**	Selecciona la orientación de selección de la tabla. El argumento booleano especifica si está permitido el tipo de selección particular. Por defecto, las selección de filas está permitida, y la de columna y celda no.

¿Cómo Usar Componentes de Texto?

Los componentes de texto muestran algún texto y opcionalmente permiten que el usuario lo edite. Los programas necesitan componentes de texto para tareas dentro delr ango del correcto (introducir una palabra y pulsar Enter) al complejo (mostrar y editar texto con estilos y con imagenes embebidas en un lenguaje asiático). Los paquetes Swing proporcionan cinco componentes de texto y proporcionan clases e interfaces para conseguir los requerimientos más complejos. Sin importar sus diferentes usos capacidades, todos los componentes de texto Swing descienden de la misma superclase, **JTextComponent**, que proporciona una base poderosa y ampliamente configurable para la manipulación de texto.

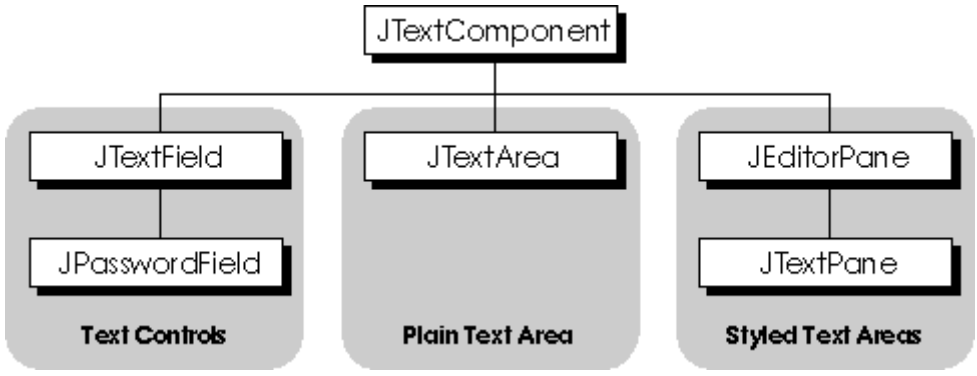
Aquí podemos ver una imagen de una aplicación que muestra cada uno de los componentes de texto Swing



Intenta esto:

1. Compila y ejecuta la aplicación. El código fuente está en **TextSamplerDemo.java**. También necesitarás **TextSamplerDemoHelp.html**, **Pig.gif**, **dukeWaveRed.gif**, y **sound.gif**.
2. Teclea texto en el campo de texto y pulsa Enter. Haz los mismo con el campo Password.
3. Selecciona y edita un texto en el área de texto y en el panel de texto. Usa las teclas especiales de teclado para cortar, cpiar y pegar texto.
4. Intenta editar el texto en el editor pane, que se ha hecho no editable con una llamada a **setEditable**.
5. Mueve por el panel de texto para encontrar un ejemplo de un componente embebido.

La siguiente figura muestra el árbol de **JTextComponent** y sitúa cada clase de componente de texto en uno de los tres grupos.



El siguiente párrafo describe los tres grupos de componentes de texto.

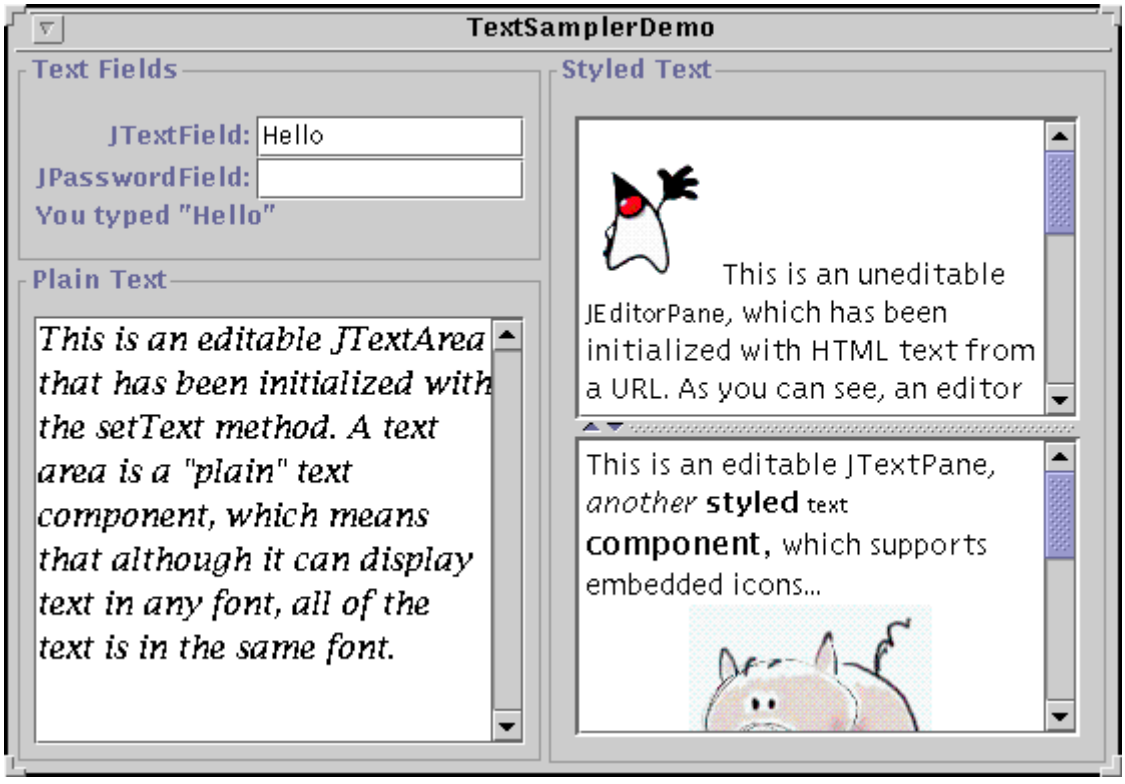
Grupo	Descripción	Clases Swing
Controles de Texto	Conocidos simplemente como campos de texto, los controles de texto pueden mostrar y editar sólo una línea de texto y están basados en action como los botones. Se utilizan para obtener una pequeña cantidad de información textual del usuario y toman algunas acciones después de que la entrada se haya completado.	**JTextField** y su subclase **JPasswordField**
Plano	**JTextArea**, el único componentes de texto plano de Swing, puede mostrar y	**JTextArea**

	editar múltiples líneas de texto. Aqunte un área de texto puede mostrar texto en cualquier fuente, todo el texto está en la misma fuente. Toda la edición de los componentes de texto plano se consigue a través de la manipulación directa del texto con el teclado y el ratón, por esto los componetes de texto plano son más fáciles de configurar y utilizar que los componentes de texto formateado. También, si la longitud del texto es menor de unas pocas páginas, podemos fácilmente utilizar **setText** y **getText** para recuperar o modificar el contenido del componente en una simple llamada a método.	
Formateado	Un componente de texto formateado puede mostrar y editar texto usando más una fuente. Algunos componentes de texto formateado permiten embeber imágenes e incluso componentes. Típicamente se tendrán que hacer más programación para usar y configurar componentes de texto formateado, porque muchas de sus funcionalidades no están disponibles a través de la manipulación directa con el ratón y el teclado. Por ejemplo, para soportar la edición del estilo de texto, tendremos que crear un interface de usuario. Una característica manejable y fácil de usar proporcionada por **JEditorPane** crea 'editor panes' y 'text panes' particularmente poderosos para mostrar información de ayuda no editable: pueden ser cargados con texto formateados desde una URL	**JEditorPane** y su subclase **JTextPane**

El programa **TextSamplerDemo** es extremadamente básico en cómo usa los componentes de texto: simplemente crea cada uno, lo configura y lo añade al marco de su aplicación. Podremos ver cada componente de texto en la siguiente sección que muestra el código usado para crear cada componente, y lo describe brevemente. Estudiando este ejemplo podrás aprender lo básico para crear y utilizar componentes de texto. La mayoría de los programadores y programas tendrán bastante con esta información básica. Sin embargo, esto sólo araña la superficie deL API de texto de Swing. Un iceberg te está esperando. La secciones restantes sobre componentes de texto te ayudarán a navegar por estas aguas.

Ejemplos de Componentes de Texto

Aquí tenemos la imagen de la aplicación **TextSamplerDemo**.



Esta sección muestra el código usado en **TextSamplerDemo** para crear cada componente de texto. Con la información contenida en esta página, aprenderás rápidamente todo lo necesario para incluir componentes de texto en un programa e interactuar con ellos a un nivel básico. Para más información sobre los usos más complejos de varios componentes de texto, puedes ir a la próxima sección, Reglas Generales para usar Componentes de Texto.

Un ejemplo de uso de Text Field

Aquí está el código de **TextSamplerDemo** que crea un **JTextField** y registra un oyente de action sobre él.

```
JTextField textField = new JTextField(10);
textField.setActionCommand(textFieldString);
textField.addActionListener(this);
```

Al igual que con los botones, podemos configurar el comando action de un textfield y registrar un oyente de action sobre él. Aquí está el método **actionPerformed** implementado por el oyente de action del textfield, que simplemente copia el texto del campo a una etiqueta.

```
String prefix = "You typed \"";
...
JTextField source = (JTextField)e.getSource();
actionLabel.setText(prefix + source.getText() + "\"");
```

Para una descripción de los constructores de **JTextField** y el método **getText** usados por esta demostración, puedes ver Cómo usar Text Fields. Esta sección también incluye información y ejemplos de campos de texto personalizados, incluyendo cómo escribir un campo validado.

Un ejemplo de uso de Password Field

JPasswordField es una subclase de **JTextField** que, en vez de mostrar el caracter real tecleado por el usuario, muestra otro caracter como un asterisco '*'. Este tipo de campo es útil para pedir al usuario que introduzca passwords cuando se conecta o para validar su identidad. Aquí está el código de **TextSamplerDemo** que crea el campo passwrod y registra un oyente de action sobre él.

```
JPasswordField passwordField = new JPasswordField(10);
passwordField.setActionCommand(passwordFieldString);
passwordField.addActionListener(this);
```

Este código es similar al usado para crear el campo de texto. El campo password comparte el oyente de action del campo de texto, que usa estas tres líneas de código para copiar el contenido del password en una etiqueta.

```
String prefix = "You typed \"";
...
JPasswordField source = (JPasswordField)e.getSource();
actionLabel.setText(prefix + new String(source.getPassword())
                                + "\"");
```

Observa que este código usa el método **getPassword** para obtener el contenido del campo password en lugar de **getText**. Proporcionar un Password Field explica porqué y proporciona información adicional sobre los campos password. Recuerda que los campos password son campos de texto, por eso la información cubierta en Cómo usar Text Fields también pertenece a los campos de password.

Usar un Text Area

Un área de texto muestra múltiples líneas de texto y permite que el usuario edite el texto con el teclado y el ratón. Aquí está el código de **TextSamplerDemo** que crea su **JTextArea**.

```
JTextArea textArea = new JTextArea(5, 10);
textArea.setFont(new Font("Serif", Font.ITALIC, 16));
textArea.setText(
                "This is an editable JTextArea " +
                "that has been initialized with the setText method. " +
                "A text area is a \"plain\" text component, " +
                "which means that although it can display text " +
                "in any font, all of the text is in the same font."
);
textArea.setLineWrap(true);
textArea.setWrapStyleWord(true);
JScrollPane areaScrollPane = new JScrollPane(textArea);
areaScrollPane.setVerticalScrollBarPolicy(
                                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
areaScrollPane.setPreferredSize(new Dimension(250, 250));
areaScrollPane.setBorder(/*...create border...*/);
```

El constructor usado en este ejemplo para crear un área de texto es similar a los usados para crear un campo de texto y un campo de password. El contructor de área de texto requiere dos argumentos enteros: el número de filas y de columnas del área. Estos números son utilizados para calcular el tamaño preferido del área de texto.

Luego el código selecciona la fuente para el área de texto seguida por el texto inicial del área. Cómo dice el texto inicial del área, aunque un área de texto puede mostrar texto en cualquier fuente, todo el texto tiene la misma fuente.

Las dos siguientes sentencias tratan de la ruptura de líneas. Primero llama a **setLineWrap**, que activa la ruptura de líneas. Por defecto, un área de texto no rompe las líneas. En su lugar mustra el texto en una sóla línea y si el área de texto está dentro de un scroll pane, se permite a sí mismo desplazarse horizontalmente. La segunda es una llamada a **setWrapStyleWord**, que le dice al área de texto que rompa la líneas entre palabras.

El siguiente grupo de código crea un scroll pane y pone el área de texto en él, selecciona el tamaño preferido del scroll pane y establece sus bordes. Normalmente un área de texto está manejada por un scroll pane. Si ponemos un área de texto en un scroll pane, debemos asegurarnos de seleccionar el tamaño preferido del scroll pane en vez del tamaño preferido del área de texto.

Usar un Editor Pane para mostrar Texto desde una URL

JEditorPane es la base para los componentes de texto formateado de Swing. **TextSamplerDemo** usa un editor pane como lo hacen muchos programas: para mostrar información no editable inicializada desde una URL que apunta a un fichero HTML**TextSamplerDemoHelp.html**.

Aquí está el código que crea el editor pane en **TextSamplerDemo**.

```
JEditorPane editorPane = new JEditorPane();
editorPane.setEditable(false);
...create a URL object for the TextSamplerDemoHelp.html file...
try {
    editorPane.setPage(url);
} catch (IOException e) {
    System.err.println("Attempted to read a bad URL: " + url);
}
```

El código usa el constructor por defecto para crear el editor pane, luego llama a **setEditable(false)** para que el usuario no pueda editar el texto. Luego, el código crea el objeto URL, y llama al método **setPage** con él. El método **setPage** abre el recurso apuntado por la URL y se imagina el formato del texto (que en este ejemplo es HTML). Si el texto formateado es conocido, el editor pane se inicializa a sí mismo con el texto encontrado en el URL.

El código que crea la URL no está en el código anterior y es interesante verlo. Aquí está.

```
String s = null;
try {
    s = "file:"
              + System.getProperty("user.dir")
              + System.getProperty("file.separator")
              + "TextSamplerDemoHelp.html";
    URL helpURL = new URL(s);
    /* ...  use the URL to initialize the editor pane  ... */
} catch (Exception e) {
    System.err.println("Couldn't create help URL: " + s);
}
```

Este código utiliza las propiedades del sistema para calcular un fichero URL desde el fichero de ayuda. Por restricciones de seguridad, este código no funcionará en la mayoría de los applets. En su lugar, utilizar el codebase del applet para calcular una URL **http**.

Al igual que las áreas de texto, los editor pane normalmente son manejados por un sroll pane.

```
JScrollPane editorScrollPane = new JScrollPane(editorPane);
editorScrollPane.setVerticalScrollBarPolicy(
                                JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
editorScrollPane.setPreferredSize(new Dimension(250, 150));
```

Observa que se ha seleccionado el tamaño preferido del scroll pane y no el del editor pane.

Un ejemplo de uso de un Text Pane

El componente de texto final en nuestra ruta básica es **JTextPane**, que es una subclase de **JEditorPane**. Aquí está el código de **TextSamplerDemo** que crea e inicializa un text pane.

```
JTextPane textPane = new JTextPane();
String[] initString =
              { /* ...  fill array with initial text  ... */ };

String[] initStyles =
              { /* ...  fill array with names of styles  ... */ };

//Create the styles we need
initStylesForTextPane(textPane);

Document doc = textPane.getDocument();

//Load the text pane with styled text
try {
    for (int i=0; i < initString.length; i++) {
              textPane.setCaretPosition(doc.getLength());
              doc.insertString(doc.getLength(), initString[i],
                                              textPane.getStyle(initStyles[i]));
              textPane.setLogicalStyle(textPane.getStyle(initStyles[i]));
    }
} catch (BadLocationException ble) {
    System.err.println("Couldn't insert initial text.");
}
```

Brevemente, esto código introduce el texto inicial en un array y crea y codifica varios **estilos** - objetos que representan diferentes formatos de párrafos y caracteres -- en otro array. Luego, el código hace un bucle por el array, inserta el texto en el text pane, y especifica el estilo a utilizar para el texto insertado.

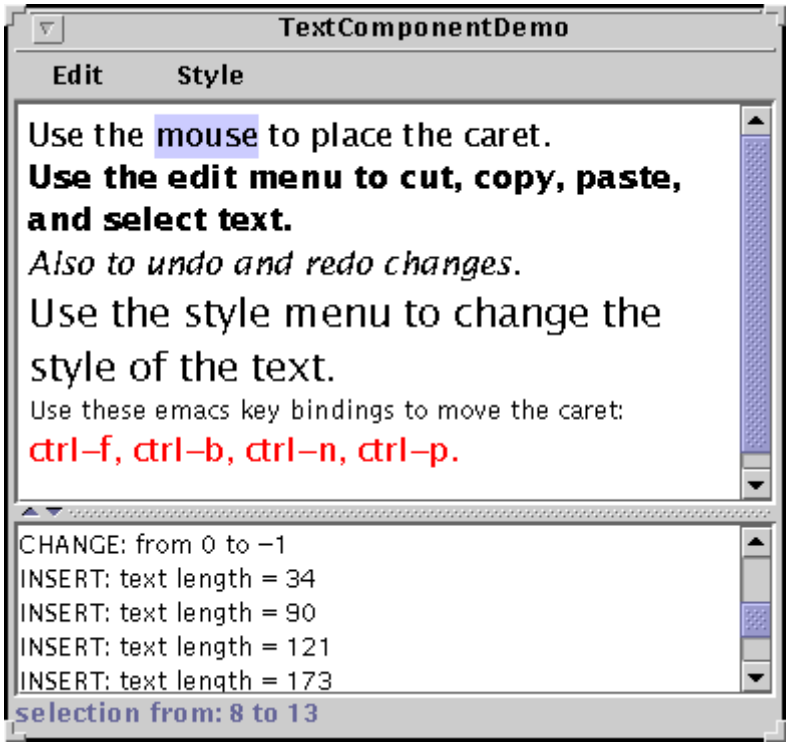
Aunque esto parece un ejemplo interesante y muestra varias características de **JTextPane**, los programas del "mundo real" no suelen inicializar un **JTextPane** de esta forma. En su lugar, un programa de calidad debería aplicar un proceso de atrapamiento, donde un **JTextPane** básico sería utilizado para crear y salvar un documento, que podría entonces ser leido por el **JTextPane** mientras el programa se está desarrollando.

Reglas de Uso de Componentes de Texto

JTextComponent es la base para los componentes de texto swing, y proporciona estas características personalizables para todos sus descendientes.

- Un modelo separado, conocido como document, para manejar el contenido del componente.
- Una vista separada, que se encarga de mostrar el componente en la pantalla. Este tutorial no explica las vistas.
- Un controlador separado, conocido como un editor kit, que puede leer y escribir texto e implementa capacidades de edición con comandos action.
- Mapas de teclas personalizados.
- Soporte para repetir/deshacer infinito.
- Cursores conectables y oyentes de cambios de cursor.

Esta sección utiliza la aplicación mostrada abajo para explorar cada una de estas capacidades. La aplicación demo contiene un **JTextPane**-- una de las subclases de **JTextComponent** que soporta texto formateado, iconos y componentes embebidos -- para ilustrar las capacidades, heredadas por todas las subclases de **JTextComponent**. Para más información específica sobre **JTextPane** pueder ver la página Cómo usar Editor Panes y Text Panes.



El componente de texto superior es el **JTextPane** personalizado. El componente de texto inferior es un **JTextArea**, que sirve como un diario que reporta todos los cambios realizados en el contenido del text pane. La línea de estado de la parte inferior de la ventana informa sobre la localización de la selección o de la posición del cursor, dependiendo de si hay o no texto seleccionado.

Nota: Es fichero fuente de esta aplicación es **TextComponentDemo.java**. También nevesitarás **LimitedStyledDocument.java**.

A través de esta aplicación de ejemplo, aprenderás cómo usar la capacidades de los componentes de texto y cómo personalizarlos. Esta sección cubre los siguientes textos, que puesen ser aplicados a todas las subclases de **JTextComponent**.

Sobre los Documentos

Al igual que muchos otros componentes Swing, un componente de texto separa su contenido de su vista. El contenido de un componente de este es manejado por su **documento**, el cual contiene el texto, soporte para edición, y notifica a los oyente los cambios en el texto. Un documento es un ejemplar de una clase que implementa el interface **Document** o su subinterface **StyledDocument**.

Personalizar un Documento

La aplicación de ejemplo mostrada anteriormente tiene un documento persoanlziado **LimitedStyledDocument**, que limita el número de caracteres que puede contener. **LimitedStyledDocument** es una subclase de **DefaultStyledDocument**, el documento por defecto para **JTextPane**.

Aquí está el código del programa de ejemplo que crea un **LimitedStyledDocument** y lo designa como el documento para le text pane

```
...donde se declaren las variables miembro...
JTextPane textPane;
static final int MAX_CHARACTERS = 300;
    ...en el constructor del frame...
    //Crea el documento para el área de texto
    LimitedStyledDocument lpd = new LimitedStyledDocument(MAX_CHARACTERS);
    ...
    //Crea el text pane y lo configura
    textPane = new JTextPane();
    textPane.setDocument(lpd);
    ...
```

Para limitar los caracteres permitidos en el docuemnto, **LimitedStyledDocument** sobreescribe el método **insertString** de su superclase, que es llamado cada vez que se inserta texto en el documento.

```
public void insertString(int offs, String str, AttributeSet a)
                                                    throws BadLocationException {
    if ((getLength() + str.length()) <= maxCharacters)
        super.insertString(offs, str, a);
    else
        Toolkit.getDefaultToolkit().beep();
}
```

Además de **insertString**, los documentos personalizados también sobreescriben el método **remove**, que es llamado cada vez que se elimina texto de un documento.

Un uso común para un documento personalizado es para crear un campo de texto validado. (un campo cuyo valor es chequeado cada vez que es editado). Para ver dos ejemplos de campos de texto validados pueder ir a Crear un Campo de Texto Validado.

Para más información puedes ver las tablas del API: Clases e Interfaces que Representan Documentos y Métodos útiles para trabajar con Documento.

Escuchar los Cambios de un Documento

Un documento notifica sus cambios a los oyentes interesados. Se utiliza un oyente de Document para reaccionar cuando se inserta o se elimina texto de un documento, o cuando cambia el estilo de alguna parte del texto.

El programa **TextComponentDemo** usa un oyente de Document para actualizar el diario de cambios siempre que ocurra un cambio en el text pane. Esta línea de código registra un ejemplar de **MyDocumentListener** como oyente del **LimitedStyledDocument** usado en el ejemplo.

```
LimitedStyledDocument lpd = new LimitedStyledDocument(MAX_CHARACTERS);
lpd.addDocumentListener(new MyDocumentListener());
```

Aquí está la implementación de **MyDocumentListener**.

```
protected class MyDocumentListener implements DocumentListener {
    public void insertUpdate(DocumentEvent e) {
        update(e);
    }
    public void removeUpdate(DocumentEvent e) {
        update(e);
    }
    public void changedUpdate(DocumentEvent e) {
        //Display the type of edit that occurred
        changeLog.append(e.getType().toString() +
                         ": from " + e.getOffset() +
                         " to " + (e.getOffset() + e.getLength() - 1) +
                         newline);
        changeLog.setCaretPosition(changeLog.getDocument().getLength() - 1);
    }
    private void update(DocumentEvent e) {
        //Display the type of edit that occurred and
        //the resulting text length
        changeLog.append(e.getType().toString() +
                         ": text length = " +
                         e.getDocument().getLength() + newline);
        changeLog.setCaretPosition(changeLog.getDocument().getLength() - 1);
    }
}
```

El oyente de nuestro ejemplo mueatr el timpo de cambio que ha ocurrido y, si está afectada por el cambio, la longitud del texto. Para información general sobre los oyente de Docuement y eventos de Document, puedes ver Cómo escribir un Oyente de Document.

Recuerda que el documento para este text pane limita el número de caracteres permitidos en el decumento. Si intentas añadir texto hasta exceder el límite máximo, el documento bloquea el cambio y no se llamará al método **insertUpdate** del oyente. Los oyentes de Document sólo son notificados si el cambio ha ocurrido realmente.

Algunas veces, podrías estar tentado de cambiar el texto del documento desde dentro de un oyente de Document. Por ejemplo, si tienes un campo de texto sólo debería contener enteros y el usuario introduce algún otro tipo de datos, podrías querer cambiar el texto a **0**. **Sin embargo, nunca se debe modificar el contenido de un componente de texto desde dentro de un oyente de Document.** De hecho, si intentas hacerlo, tu programa se quedará bloqueado. En su lugar proporciona un documento personalizado y sobreescribe los métodos **insert** y **remove**. Crear un Campo de texto validado te enseña como hacerlo.

Sobre los Kits de Edición

Todos los componentes de Texto Swing soportan comandos de edición estándard como cortar, copiar, pegar y la inserción de caracteres. Cada comando de edición está representada e implementado por un objeto action. Esto hace sencillo el asociar un comando con un componente GUI, como un ítem de menú o un botón, y construir un GUI alrededor de un componente de texto.

Un componente de texto usa un objeto **EditorKit** para crear y manejar estas acciones. Además de manejar un conjunto de acciones para un componente de texto, un kit de editor también sabe leer y escribir documentos en un formato particular.

El paquete text de Swing proporciona estos tres kits de editor.

DefaultEditorKit
: Lee y escribe texto sin estilo. Proporciona un conjunto básico de comandos de edición. Los demás kits de editor descienden de este.

StyledEditorKit
: Lee y escribe texto con estilo y proporciona un conjunto de acciones mínimo para texto con estulo. Esta clase es una subclase de **DefaultEditorKit** y es el kit de editor usado por defecto por **JTextPane**.

HTMLEditorKit
: Lee, escribe y edita HTML. Esta es una subclase de **StyledEditorKit**.

La mayoría de los programas no necesitan escribir código que interactúe directamente con los kits de editor porque **JTextComponent** proporciona el API necesario para invocar directamente a las capacidades del kit. Por ejemplo, **JTextComponent** proporicona métodos **read** y **write**, que llaman a los métodos **read** y **write** del kit de editor. **JTextComponent** también proporcionar un método, **getActions**, que devuelve todas las acciones soportadas por un componente. Este método obtiene una lista de acciones desde el kit de editor del componente. Sin embargo, las clases del kit de editor proporciona útiles clases internas y variables de clases que son muy útiles para crear un GUI alrededor de un componente de texto. Asociar Acciones con Ítems de Menú muestra como asociar una acción con un ítem de menú y Asociar Acciones con Pulsaciones de Teclas muestra como asociar una acción con una pulsación de teclas determinadas. Ambas secciones hacen uso de clases manejables o de variables definidas en los kits de editor estándars de Swing.

Asociar Acciones con Ítems de Menú

Como se mencionó anteriormente, podemos llamar al método **getActions** sobre cualquier componente para obtener un array que contenga todas las acciones soportadas por dicho componente. Es conveniente cargar el array de acciones en un **Hashtable** para que nuestro programa pueda recuperar una acción por su nombre. Aquí está el código de **TextComponentDemo** que obtiene las acciones del text pane y las carga dentro de un **Hashtable**.

```
private void createActionTable(JTextComponent textComponent) {
    actions = new Hashtable();
    Action[] actionsArray = textComponent.getActions();
    for (int i = 0; i < actionsArray.length; i++) {
        Action a = actionsArray[i];
        actions.put(a.getValue(Action.NAME), a);
    }
}
```

Y aquí hay un método de conveniencia para recuperar una acción por su nombre desde el hashtable.

```
private Action getActionByName(String name) {
    return (Action)(actions.get(name));
}
```

Puedes utilizar ambos métodos de forma casi literal en tus programas. Sólo tienes que cambiar **actions** por el nombre de tu hashtable.

Ahora, vemaos cómo se crea el ítem de meú **Cut** y como se asocia a la acción de eliminar texto de un componente de texto.

```
protected JMenu createEditMenu() {
    JMenu menu = new JMenu("Edit");
    ...
    menu.add(getActionByName(DefaultEditorKit.cutAction));
    ...
```

Este código obtiene la acción por su nombre usando el método descrito anteriormente y añade la acción al menú. Esto es todo lo que necesitas hacer. El menú y la acción tienen en cuenta todo lo demás. Observaras que el nombre de la acción viene de **DefaultEditorKit**. Este kit proporciona acciones para la edición básica de texto y es la superclase para todos los kits de editor proporcionados por Swing. Por eso sus capacidades están disponibles para todos los componentes de texto a menos que se hayan sobreescrito por una personalización.

Por razones de rendimiento y eficiencia, los componentes de texto comparten acciones. El objeto **Action** devuelto por **getActionByName(DefaultEditorKit.cutAction)** es compartido por el **JTextArea** (no editable) de la parte inferior de la ventana. Esto tiene dos importantes ramificaciones.

- Generalmente hablando, no se deben modificar los objetos **Action** obtenidos de los kits de editor. Si lo hacemos, los cambios afectarán a todos los componentes de texto de nuestro programa.
- Los objetos **Action** pueden operar con otros componentes de texto del programa, quizás más de los esperados. En este ejemplo, incluso aunque no sea editable, el **JTextArea** comparte las acciones con el **JTextPane**. Si no queremos compartir, deberemos ejemplarizar el objeto **Action** nosotros mismos. **DefaultEditorKit** define varias sublcases de **Action** muy útiles.

Configurar el menú **Style** es similar. Aquí está el código que crea y pone el ítem de menú **Bold** en él.

```
protected JMenu createStyleMenu() {
    JMenu menu = new JMenu("Style");
```

```
Action action = new StyledEditorKit.BoldAction();
action.putValue(Action.NAME, "Bold");
menu.add(action);
...
```

StyledEditorKit proporciona sublcases de **Action** para implementar comandos de edición para texto con estilo. Habrás notado que en lugar de obtener la accón del kit de editor, este código crea un ejemplar de la clase **BoldAction**. Asñi, esta acción no será compartida por ningún otro componente de texto, y cambiando su nombre no afectará a ningún otro componente de texto.

Además de asociar una acción con componente GUI, también podemos asociar una acción con una pulsación de teclas Asociar Acciones con Pulsaciones de Teclas muestra como hacerlo.

Puedes ver la tabla de API relacionada con los Comandos de Edición de Texto.

Sobre los Mapas de Teclado

Cada componente de texto tiene uno o más keymaps-- cada uno de los cuales es un ejemplar de la clase **Keymap**. Un keymap contiene una colección de parejas nombre-valor donde el nombre es un **KeyStroke** (pulsación de tecla) y el valor es una **Action**. Cada parej enlaza el keystroke con la acción por lo tanto cada vez que el usuario pulsa la tecla, la acción ocurrirá.

Por defecto, un componente de texto tiene un keymap llamado **JTextComponent.DEFAULT_KEYMAP**. Este keymap contiene enlaces básicos estándars. Por ejemplo, las teclas de flechas están mapeadas para mover el cursor, etc. Se puede modificar o ampliar el kjeymap por defecto de las siguientes formas.

- Añadiendo un keymao personalizado al componente de texto con del método **addKeymap** de **JTextComponent**.
- Añadiendo enlaces de teclas al keymap por defecto con el método **addActionForKeyStroke** de **Keymap**. El Keymap por defecto es compartido entre todos los componentes de texto, utilizalo con precaución.
- Eliminado enlaces de teclas del keymap por defecto con el método **removeKeyStrokeBinding** de **Keymap**. El Keymap por defecto es compartido entre todos los componentes de texto, utilizalo con precaución.

Cuando se resuelve una pulsación a su acción, el componente de texto chequea el keymap en el orden en que fueron añadidos al componente de texto. Así, el enlace para una pulsación específica en un keymap que hayamos añadido a un componente de texto sobreescribe cualquier enlace para la misma pulsación en el keymap por defecto.

Asociar Acciones con Pulsaciones de Teclas

El text pane de **TextComponentDemo** añade cuatro enlaces de teclas al keymap por defecto.

- **CTRL-B** para mover el cursor un caracter hacia atrás
- **CTRL-F** para mover el cursor un caracter hacia adelante
- **CTRL-P** para mover el cursor una línea hacia arriba
- **CTRL-N** para mover el cursor una línea hacia abajo.

El siguiente código añade el enlace de tecla **CTRL-B** al keymap por defecto. El código para añadir las otras tres es similar.

```
//Obtiene el mapa por defecto actual
Keymap keymap = textPane.addKeymap("MyEmacsBindings",
                                                    textPane.getKeymap());

//Ctrl-b para ir hacia atrás un caracter.
Action action = getActionByName(StyledEditorKit.backwardAction);
KeyStroke key = KeyStroke.getKeyStroke(KeyEvent.VK_B, Event.CTRL_MASK);
keymap.addActionForKeyStroke(key, action);
```

Este código primero añade el keymap al árbol de componentes. El método **addKeymap** crea el keymap por nosotros con el nombre y padre proprocionados en la llamada al método. En este ejemplo, el padre es el keymap por defecto del text pane. Luego, el código obtiene la acción de ir hacia atrás del kit de editor y obtiene un objeto **KeyStroke** que representa la secuencia de teclas **CTRL-B**. Finalmente, el código añade la pareja acción y puslación al keymap, y por lo tanto enlaza la tecla con la acción.

Puedes ver el API relacionado en la tabla Enlazar Pulsaciones a Acciones.

Implementar Deshacer y Repetir

Nota: La implementación de deshacer/repetir en **TextComponentDemo** fue copiada directamente del NotePad que viene con Swing. La mayoría de los programadores también podrán copiar esta implementación sin modificaciones.

Implementar Deshacer/repetir tiene dos partes.

Parte 1: Recordar Ediciones "Reversibles"

Para soportar deshacer/repetir, un componente de texto debe recordar cada edición que ha ocurrido sobre él, el orden en que ocurren las ediciones en relación a otras, y que hacer para deshacerlas. El programa de ejemplo usa un manejar de deshacer, un ejemplar de la

clase **UndoManager** del paquere **undo** de Swing, para menjar una lista de ediciones reversibles. El undomanager se crea donde se declaran las variables miembros.

```
protected UndoManager undo = new UndoManager();
```

Ahora veamos como el programa encuentra las ediciones reversibles y las añade al undomanager.

Un documento noticia a los oyentes interesados si en su contenido ha ocurrido una edición reversible. Un paso importante en la implementación de deshacer/repetir es registrar un oynete de 'undoable edit' en el documento del componente de texto. Este código añade un ejemplar de **MyUndoableEditListener** al documento del text pane.

```
lpd.addUndoableEditListener(new MyUndoableEditListener());
```

El oyente usado en nuestro ejemplo añade la edición a la lista del undomanager.

```
protected class MyUndoableEditListener implements UndoableEditListener {
    public void undoableEditHappened(UndoableEditEvent e) {
        //Recuerda la edición y actualiza los menús
        undo.addEdit(e.getEdit());
        undoAction.update();
        redoAction.update();
    }
}
```

Observa que este método actualizad so objetos: **undoAction** y **redoAction**. Estos dosn dos objetos actión añadidos a los ítems de menú **Undo (Deshacer)** y **Redo (Repetir)** menu items, respectivamente. El siguiente paso es ver como se crean los dos ítems de menú y la implementación de las dos acciones.

Para información general sobte oyentes y eventos de 'undoable edit' puedes ver: Cómo escribir un oyente de Undoable Edit.

Parte 2: Implementar los Comandos Deshacer/Repetir

El primer paso de esta parte de implementación es crear las acciones y ponerlas en el menú **Edit**.

```
JMenu menu = new JMenu("Edit");

//Deshacer y repetir son acciones de nuestra propia creacción
undoAction = new UndoAction();
menu.add(undoAction);

redoAction = new RedoAction();
menu.add(redoAction);
...
```

Las acciones deshacer y repetir son implementadas por subclases personalizadas de **AbstractAction**: **UndoAction** y **RedoAction** respectivamente. Estas clases son clases internas de la clases primaria del ejemplo.

Cuando el usuario llama al comando **Undo**, el método **actionPerformed** de **UndoAction** mostrado aquí, obtiene la llamada.

```
public void actionPerformed(ActionEvent e) {
    try {
        undo.undo();
    } catch (CannotUndoException ex) {
        System.out.println("Unable to undo: " + ex);
        ex.printStackTrace();
    }
    update();
    redoAction.update();
}
```

Este método llama al método **undo** del undomanager y actualiza los ítems de menú para reflejar el nuevo estado de deshacer/repetir.

De forma similar, cuando el usuario llama al comando **Redo**, el método **actionPerformed** de **RedoAction** obtiene la llamada.

```
public void actionPerformed(ActionEvent e) {
    try {
        undo.redo();
    } catch (CannotRedoException ex) {
        System.out.println("Unable to redo: " + ex);
        ex.printStackTrace();
    }
    update();
    undoAction.update();
}
```

Este método es similar excepto en que llama al método **redo** de undomanager.

La mayoría del código de las clases **UndoAction** y **RedoAction** está dedicada a habilitar o deshabilitar las acciones de forma apropiada al estado actual, y cmabiar los nombres de los ítems de menú para reflejar la edición a deshacer o repetir.

Escuchar los cambios de cursor o de selección

El programa **TextComponentDemo** usa un oyente de caret para mostrar la posición actual del cursor o, si hay texto selecciónado la extensión de la selección.

El oyente de caret de este ejemplo es también una etiqueta. Aquí está el código que crea la etiqueta, la añade a la ventana, y la hace oyente de caret del text pane.

```
//Crea el área de estado
JPanel statusPane = new JPanel(new GridLayout(1, 1));
CaretListenerLabel caretListenerLabel = new CaretListenerLabel(
                                                                    "Caret Status");
statusPane.add(caretListenerLabel);
...
textPane.addCaretListener(caretListenerLabel);
```

Un oyente de caret debe implemenar un método, **caretUpdate**, que es llamado cada vez que el cursor se mueva o cambie la selección. Aquí está la implementación que **CaretListenerLabel** hace de **caretUpdate**.

```
public void caretUpdate(CaretEvent e) {
    //Obtiene la posición en el texto
    int dot = e.getDot();
    int mark = e.getMark();
    if (dot == mark) {  // no hay selección
        try {
            Rectangle caretCoords = textPane.modelToView(dot);
            //Convierte las coordenadas
            setText("caret: text position: " + dot +
                    ", view location = [" +
                    caretCoords.x + ", " + caretCoords.y + "]" +
                    newline);
        } catch (BadLocationException ble) {
            setText("caret: text position: " + dot + newline);
        }
     } else if (dot < mark) {
        setText("selection from: " + dot + " to " + mark + newline);
     } else {
        setText("selection from: " + mark + " to " + dot + newline);
     }
}
```

Cómo puedes ver, este oyenbte actualiza su etiqueta de texto para reflejar el estado actual del cursor o la selección. El oyente obtiene la información mostrada desde el objeto caret event. Para información general sobre los oyentes y eventos de cursor puedes ver Cómo escribir un oyente de Caret.

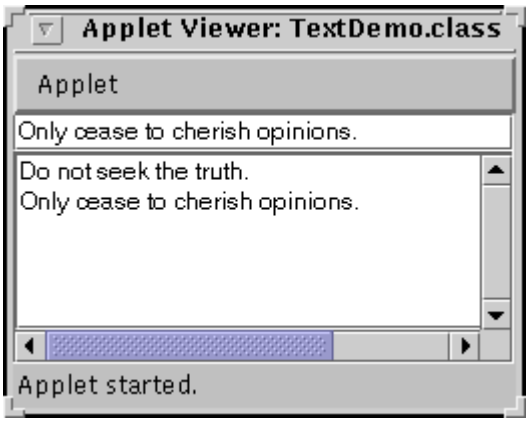
Al igual que los oyenentes de document, un oyente de caret es pasivo. Reacciona a los cambios del cursor o de la selección, pero no cambia el cursor ni la selección. En vez de modidicar el cursor o la seleccíon desde un oyente de caret, deberemos usar un caret personalizado. Para crear un caret peresonalizado, debemos escribir una clase que implemente el interface **Caret**, luego proporcionar un ejemplar de nuestra clase como argumento a **setCaret** sobre un componente de texto.

¿Cómo usar TextField?

Un campo de texto es un control básico que permite al usuario teclear una pequeña cantidad de texto y dispara un evento action cuando el usuario indique que la entrada de texto se ha completado (normalmente pulsando Return). Generalmente se usa la clase **JTextField** para proporcionar campos de texto. Si necesitamos proporcionar un password field -- un campo de texto editable que no muestra los caracteres tecleados por el usuario -- utilizaremos la clase **JPasswordField**. Esta sección explica estos dos campos de texto.

Si queremos un campo de texto que también proporcione un menú de cadenas desde la que elegir una, podemos considerar la utilización de un combo box editable. Si necesitamos obtener más de una línea de texto desde el usuario deberíamos utilizar una de las glases que implementan text area para propósito general.

El Applet siguiente muestra un campo de texto básico y un área de texto. El campo de texto es editable y el áera de texto no lo és. Cuando el usuario pulse Return en el campo de texto, el campo dispara un action event. El applet reacciona al evento copiando el contenido del campo de texto en el área de texto y seleccionando todo el texto del campo de texto.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre la imagen. El Applet aparecerá en una nueva ventana del navegador.

Puedes encontrar el programa fuente en **TextDemo.java**. Aquí está el código de **TextDemo** que crea el campo de texto del applet.

```
textField = new JTextField(20);
textField.addActionListener(this);
...
contentPane.add(textField);
```

El argumento entero pasado al constructor de **JTextField**, **20** en el ejemplo, indica el número de columnas del campo, que es usada junto con la métrica proporcionada por el font actual del campo, para calcular la anchura preferida por el campo. Como varios controladores de disposición ingnoran los tamaños preferidos y como los márgenes, los bordes y otros factores afectan al tamaño del componente, toma este número como una aproximación, no un número absoluto.

Las siguientes líneas de código registran el applet como oyente de action para el campo de texto y añade el campo de texto al panel de contenidos del applet. Aquí está el método **actionPerformed** que meneja los eventos action del campo de texto.

```
public void actionPerformed(ActionEvent evt) {
    String text = textField.getText();
    textArea.append(text + newline);
    textField.selectAll();
}
```

Observa el uso del método **getText** de **jTextField** para recuperar el contenido actual del campo de texto. El texto devuelto por este método no incluye un caracter de nueva línea para la tecla Return que disparó el evento action.

Este ejemplo ilustra usando un campo de texto básico para introducir datos textuales y realizar algunas tareas cuando el campo de teto diapara un evento action. Otros programas, sin embargo, necesitan un comportamiento más avanzado. Una subclase de **JTextComponent**, **JTextField** puede ser configurada y personalizada. Un personalización común es proporcionar un campo de texto cuyos contenidos sean validados. Esta sección cubre los siguientes tópicos de los campos de texto avanzados. Para entender toda la información, necesitas haber comprendido el material presentado en Reglas Generales para el uso de Componentes.

Crear un Text Field Validado

Muchos programas requieren que el usuario introduzca un dato textual de un cierto tipo o formato. Por ejemplo, un programa podría proporcionar un campo de texto para entrar una fecha, un número decimal, o un número de teléfono. Los contenidos de dichos campos como campos de texto deben ser validados antes de ser utilizados para cualquier propósito. Un campo de texto puede ser validado cuando se dispare el evento action o el evento keystroke.

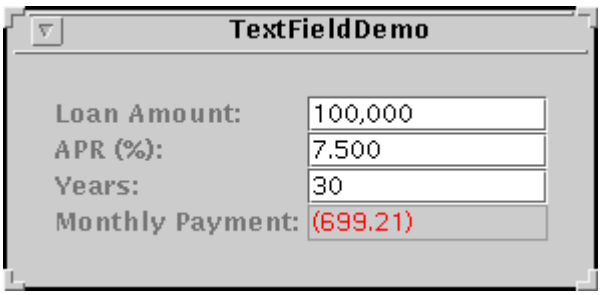
El dato en un campo validado-en-action se chequea cada vez que el campo dispara un evento action (cada vez que el usuario pulsa la tecla Return). Un campo validado-en-action podría, en un momento dado, contener datos no válidos. Sin embargo, el dato será validado antes de ser utilizado. Para crear un campo validado-en-action, necesitamos proporcionar un oyente action para nuestro campo e implementa su método **actionPerformed** de la siguiente forma.

- Usa **getText** para obtener el contenido del campo de texto.
- Evalua el valor devuelto por **getText**.
- Si el valor es válido, realiza cualquier tarea de cálculo que sea requerida. Si el campo es nulo, reporta el error y retorna sin realizar ninguna tarea de cálculo.

El dato en un campo validado-en-pulsación se chequea cada vez que el campo cambia. Un campo validado-en-pulsación nunca puede contener datos no válidos porque cada cambio (pulsación, cortar, copiar, etc.) hace que el dato no válido sea rechazado. Para crear un campo de texto validado-en-pulsación necesitamos proporcionar un documento personalizado para nuestro campo de texto. Si no estás familiarizado con los documentos, puedes ir a Trabajar con el Documento de un Componente de Texto.

Aviso: No use un oyente de document para validación-por-pulsación. El momento en que un oyente de documento es notificado de un cambio, es demasiado tarde, el cambio ya ha tenido lugar. Puedes los dos últimos párafos de Escuchar los Cambios en un Documento para más información

La aplicación mostrada en la siguiente figura tiene tres campos validados-por-pulsación. El usuario introduce información en los tres primeros campos de texto. Cada vez que el usuairo teclea un caracter, el programa valida la entrada y actualiza el resultado del cuarto campo de texto.



Prueba esto:

1. Compila y ejecuta la aplicación. El fichero fuente es **TextFieldDemo.java**. También necesitarás **WholeNumberField.java**, **DecimalField.java**, y **FormattedDocument.java**.
2. Introduce la información en los campos de texto y mira los resultados.

   Si intentas introducir un campo no válido, el programa pitará.

3. Intenta teclear algo en el cuarto campo de texto.

   No puedes, porque no es editable; sin embargo si se puede seleccionar el texto.

4. Redimensiona la ventana.

   Observa como las etiquetas y los campos de texto permanecen alienados. Distribuir Parejas de Etiqueta/campo de Texto te contará más cosas sobre esta característica del programa.

El campo **Years** es un ejemplar de **WholeNumberField.java**, que es una subclase de **JTextField**. Sobreescribiendo el método **createDefaultModel**,**WholeNumberField** establece una subclases **Document** personalizada -- un ejemplar de **WholeNumberDocument** -- como documento para cada **WholeNumberField** creado.

```
protected Document createDefaultModel() {
    return new WholeNumberDocument();
}
```

Aquí está la implementación de **WholeNumberDocument**.

```
protected class WholeNumberDocument extends PlainDocument {

    public void insertString(int offs, String str, AttributeSet a)
                    throws BadLocationException {

        char[] source = str.toCharArray();
        char[] result = new char[source.length];
        int j = 0;

        for (int i = 0; i < result.length; i++) {
                    if (Character.isDigit(source[i]))
                        result[j++] = source[i];
                    else {
                        toolkit.beep();
                        System.err.println("insertString: " + source[i]);
                    }
        }
        super.insertString(offs, new String(result, 0, j), a);
    }
}
```

Esta clase sobreescribe el método **insertString** el cual es llamado cada vez que un string o un caracter va a ser insertado en el documento. La implementación de **WholeNumberDocument** de **insertString** evalua cada caracter a ser insertado dentro dle campo de texto. Si el carácter es un dígito, el documento permite que sea insertado. De otro modo, el método pita e imprime un mensaje de error. Por lo tanto, **WholeNumberDocument** permite los números en el rango 0, 1, 2, ...

Un detalle de implementación interesante esque nuestra clase document personalizada no tiene que sobreescribir el método **remove**. Este método es llamado cada vez que un caracter o grupos de caracteres es eliminado del campo de texto. Como eliminar un dígito de un entero no puede producir un resultado no válido, esta clase no presta atención a las eliminaciones.

Los otros dos campos de texto del ejemplo, así como el campo no editable **Monthly Payment**, son ejemplares de **DecimalField.java**, una subclase personalizada de **JTextField**. **DecimalField** usa un documento personalizado,**FormattedDocument**, que sólo permite que sena intorducidos los datos en un formato particular.

FormattedDocument no tiene conocimiento del formato real de su contenido. En su lugar **FormattedDocument** relga en un formato, una ejemplar de una subclase de **Format**, para aceptar o rechazar el cambio propuesto. El campo de texto que usa el **FormattedDocument** debe especificar el formato que se debe utilizar.

Los campos **Loan Amount** y **Monthly Payment** usan un objeto **NumberFormat** creado de esta forma

```
moneyFormat = NumberFormat.getNumberInstance();
```

El siguiente código crea el formato dle campo de texto **APR**.

```
percentFormat = NumberFormat.getNumberInstance();
percentFormat.setMinimumFractionDigits(3);
```

Como muestra el código, la misma clase (**NumberFormat**) puede soportar diferentes formatos. Además, **Format** y sus subclases son sensitivas a la localidad, por eso un campo decimal, puede hacerse para soportar formatos de otros paises y regiones. Puedes referirte a **Formateando** ien la sección de Internacionalización para información más detallada sobre los formatos.

Aquí está la implementación que **FormattedDocument** hace de **insertString**.

```
public void insertString(int offs, String str, AttributeSet a)
                throws BadLocationException {

    String currentText = getText(0, getLength());
    String beforeOffset = currentText.substring(0, offs);
    String afterOffset = currentText.substring(offs, currentText.length());
    String proposedResult = beforeOffset + str + afterOffset;

    try {
        format.parseObject(proposedResult);
        super.insertString(offs, str, a);
    } catch (ParseException e) {
        Toolkit.getDefaultToolkit().beep();
        System.err.println("insertString: could not parse: "
                                              + proposedResult);
    }
}
```

El método usa el formato para analizar el resultado de la inserción propuesta. Si el resultado se formatea adecuadamente, el método llamada al método **insert** de su superclase para hacer la inserción. Si el resultado no se formatea de la forma decuada, el ordenador pita.

Además de sobreescribir el método **insertString**, **FormattedDocument** también sobreescribe el método **remove**.

```
public void remove(int offs, int len) throws BadLocationException {
    String currentText = getText(0, getLength());
    String beforeOffset = currentText.substring(0, offs);
    String afterOffset = currentText.substring(len + offs,
                                               currentText.length());
    String proposedResult = beforeOffset + afterOffset;

    try {
        if (proposedResult.length() != 0)
            format.parseObject(proposedResult);
        super.remove(offs, len);
    } catch (ParseException e) {
        Toolkit.getDefaultToolkit().beep();
        System.err.println("remove: could not parse: " + proposedResult);
    }
}
```

La implementación que **FormattedDocument** hace del método **remove** es similar a su implementación del método **insertString**. El formato analiza el resultado del cambio propuesto, y realiza la eliminación o no, dependiendo de si el resultado es válido.

Nota: La solución propuesta en este ejmplo no es una solución general para todos los tipos de formatos. Algunos formatos puede ser validados-por-pulsación simplemente llamando al método **parseObject**. Aquí tenemos un ejejmplo que te puede ayudar a entender por qué. Supongamos que tenemos un campo de texto que contiene la fecha "May 25, 1996" y queremos cambiarlo a "June 25, 1996". Deberías selección "May" y empezar a teclear "June". Tan pronto como teclearas la "J", el campo no analizaría porque "J 25, 1996" no es un dato válido, aunque si es un cambio válido. Hay un número de posibles soluciones para fechas y otros tipos de datos cuando un cambio incompleto puede crear un resultado no válido. Se puede cambiar la validación-por-pulsación para que rechace definitivamente todos los cambios no válido (teclear "X" en un campo de fecha, por ejemplo) pero permitir todos los cambios válidos posibles. O cambiar a un campo validado-en-action.

Usar un Oyente de Document en un Campo de Texto

Entonces, si no podemos utilizar un oyente de document para validación de cmapos, ¿para qué podemos utilizarlo? Se usa para oir, pero no interferir, con los cambios del contenido del documento. La calculadora de pagos usa el sigueinte oyente de document para actualizar el pago mensual después de cada cambio.

```
class MyDocumentListener implements DocumentListener {
    public void insertUpdate(DocumentEvent e) {
        update(e);
    }
    public void removeUpdate(DocumentEvent e) {
        update(e);
    }
    public void changedUpdate(DocumentEvent e) {
        // we won't ever get this with a PlainDocument
    }
    private void update(DocumentEvent e) {
        Document whatsup = e.getDocument();
        if (whatsup.getProperty("name").equals("amount"))
            amount = amountField.getValue();
        else if (whatsup.getProperty("name").equals("rate"))
            rate = rateField.getValue();
        else if (whatsup.getProperty("name").equals("numPeriods"))
            numPeriods = numPeriodsField.getValue();
        payment = computePayment(amount, rate, numPeriods);
        paymentField.setValue(payment);
    }
}
```

Este es un uso apropiado para el uso de un oyente de document.

Para información general sobre oyentes de document, puedes ir a la página Cómo Escribir un Oyente de Document.

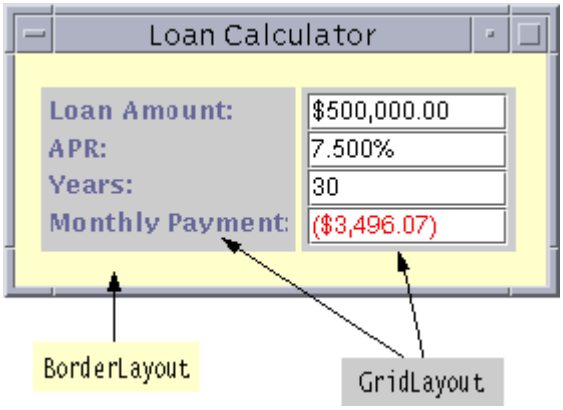
Distribuir Parejas Etiqueta/Campo de Texto

Esta sección describe cómo se an alineado las etiquetas y los campos de texto del ejemplo y requiere algún conocimiento de controladores de distribución.

Las líneas de parejas de etiquetas y campos de texto como los encontradas en la calculadora de pagos son bastante comunes en los paneles que implementan formularios. Aquí está el código que distribuye las etiquetas y los campos de texto.

```
. . .
//distribuye las etiquetas sobre el panel
JPanel labelPane = new JPanel();
labelPane.setLayout(new GridLayout(0, 1));
labelPane.add(amountLabel);
labelPane.add(rateLabel);
labelPane.add(numPeriodsLabel);
labelPane.add(paymentLabel);

//distribuye los campos de texto sobre el panel
JPanel fieldPane = new JPanel();
fieldPane.setLayout(new GridLayout(0, 1));
fieldPane.add(amountField);
fieldPane.add(rateField);
fieldPane.add(numPeriodsField);
fieldPane.add(paymentField);

//Pone los paneles sobre otro panel, las etiquetas a la izquierda,
//los campos de texto a al derecha
JPanel contentPane = new JPanel();
contentPane.setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));
contentPane.setLayout(new BorderLayout());
contentPane.add(labelPane, BorderLayout.CENTER);
contentPane.add(fieldPane, BorderLayout.EAST);

setContentPane(contentPane);
. . .
```

Podrías haberte sorprendido de encontrar que las etiquetas fueran distribuidas sin referencia a los campos de texto, y de hecho, están en un panel diferente, sólo alineados correctamente con ellos. Esto es un efecto colateral de los controladores de distribución usados por el programa.



Como muestra el diagrama, el progra usa dos controladores **GridLayout**, uno para la columna de etiquetas y otro para la columna de campos de texto. **GridLayout** garantiza que todos sus componentes sean del mismo tamaño, por eso todos los campos de texto tienen la mista altura y todas las etiquetas tienen la misma altura.

Para conseguir que las etiquetas y los campos de texto esten alineados, el programa usa un tercer controlador, un **BorderLayout**. con sólo dos componentes, en la izquierda y en el centro, **BorderLayout** garantiza que las columnnas son de la misma altura. Y así se alinean las etiquetas y los campos de texto.

Otra forma de conseguir alinear las etiquetas y los campos de texto es utilizar el más complejo de los controladores de distribución del AWT, el **GridBagLayout**.

Proporcionar un Campo de Password

Swing proporciona la clase **JPasswordField**, una subclase de **JTextField**, que se usa en lugar de un campo de texto cuando el texto introducido por el usuario es una password. Por razones de seguridad, un campo de password no muestra los caracteres que teclea el usuario. En su lugar el campo muestra otro caracter, como un asterisco "*".

El ejemplo **PasswordDemo** descrito en Usar la Clase SwingWorker usa un **JPasswordField**. El programa trae una pequeña ventana para pedirle al usuario que teclee una password.

Aquí está el código de **PasswordDemo** que crea y configura el campo password.

```
JPasswordField password = new JPasswordField(10);
password.setEchoChar('#');

password.addActionListener(showSwingWorkerDialog);
```

Como con los campos de texto, el argumento pasado al constructor **JPasswordField** indica que el campo deberá tener 10 columnas de ancho. Por defecto, un campo password muestra asteriscos "*" por cada caracter tecleado. La llamada a **setEchoChar** lo cambia por el signo de almohadilla "#". finalmente, el código añade una oyente de action al campo password. El método **actionPerformed** del oyente de actuib obtiene la password tecleada por el usuario y la verifica con este código.

```
public void actionPerformed(ActionEvent e) {
    JPasswordField input = (JPasswordField)e.getSource();
    char[] password = input.getPassword();
    if (isPasswordCorrect(password))
        JOptionPane.showMessageDialog(f, worker.get());
    else
        JOptionPane.showMessageDialog(f,
                    new JLabel("Invalid password."));
    }
```

Este método utiliza el método **getPassword** para obtener el contenido del campo. Esto es por lo que **getPassword** devuelve un array de caracteres. La información de passwords no debería ser almacenada no pasada en strings, porque no son seguras.

Un programa que usa un campo de password tipicamente valida la password antes de completar cualquier acción que requiera la password. Este programa llama a un método personalizado, **isPasswordCorrect**, que compara el valor devuelto por **getPassword** con el valor almacenado en un array de caracteres.

El API de Text Field

Las siguientes tablas listan los constructores y métodos más comunmente utilizados de **JTextField**. Otros métodos a los que se podría llamar están definidos en las clases **JComponent** y **Component**. Estos incluyen los métodos **setForeground**, **setBackground**, y **setFont**.

Además, podrías querer llamar a algunos de los métodos definidos en la clase padre de **JTextField**, **JTextComponent**.

El API para usar campos de texto se divide en tres categorías.

Seleccionar u Obtener el Contenido de un Campo de Texto

Método o Constructor	Propósito
JTextField() **JTextField(String)** **JTextField(String, int)** **JTextField(int)** **JTextField(Document, String, int)**	Crea un ejemplar de **JTextField**, inicializando su contenido al texto especificado. El argumento **int** seleccionar el número de columnas. Esto se utiliza para calvlar la anchura preferida del componente y podría no ser el número de columnas realmente mostradas.
void setText(String) **String getText()**	Seleccion u obtiene el texto mostrado por el campo de texto.

Ajuste Fino de la Apariencia de un Campo de Texto

Método o Constructor	Propósito
void setEditable(boolean) **boolean isEditable()**	Selecciona u obtiene si el usuario puede editar el texto del campo del texto.
void setForeground(Color) **Color getForeground()**	Selecciona u obtiene el color del texto en el campo de texto.
void setBackground(Color); **Color getBackground()**	Selecciona u obtiene el color del fondo del campo de texto
void setFont(Font);	Selecciona u obtiene la fuente utilizada por el campo de texto.

Font getFont()	
void setColumns(int); **int getColumns()**	Selecciona u obtiene el número de columnas mostradas por el campo de texto.
int getColumnWidth()	Obtiene la anchura de las columnas del campo de texto. Este valor es establecido implicitamente por la fuente usada.
void setHorizontalAlignment(int); **int getHorizontalAlignment()**	Selecciona u obtiene cómo se alinea el texto horizontalmente dentro de su área. S epuede utilizar **JTextField.LEFT**, **JTextField.CENTER**, y **JTextField.LEFT** como argumentos.

Implementar la Funcionalidad del Campo de Texto

Método o Constructor	Propósito
void addActionListener(ActionListener) **void removeActionListener(ActionListener)**	Añade o elimina un oyente de action.
Document createDefaultModel()	Sobreescribe este método para proporcionar un documento personalizado.

¿Cómo usar EditorPane?

JEditorPane

JTextPane

Lo primero que la mayoría de la gente quiere saber es: ¿Qué diferencia hay entre un editor pane y un text pane? Primero esta sección intenta responder esta cuestión. Luego describe el código en el **TextSamplerDemo** específico de **JTextPane**.

Cada uno de estos kits de editor ha sido registrado con la clase **JEditorPane** y asociado con el formato de texto que el kit entiende. Cuando un fichero es cargado dentro de un editor pane, el panel chequea el formato del fichero con sus kits registrados. Si se encuentra un kit registrado que soporta el formato del fichero, el panel usa el kit para ller el fichero, mostrarlo y editarlo. Así, el editor pane se convierte a sí mismo en un editor para ese formato de texto.

Podemos extender **JEditorPane** para soportar nuestro propio formato de texto creando un editor kit para él, y luego usando **registerEditorKitForContentType** de **JEditorPane** para asociarlo con nuestro formato de texto.

Sumario de Componentes de Texto

El API de texto de Swing es grande. Este tutorial ha mostrado un sencillo ejemplo del uso de cada componente, cubriendo la línea de fundada por **JTextComponent**, y muestra como utilizar esa herencia pra hacer cosas interesantes con los componentes de texto.

El API de Texto

Esta sección proporciona estas tablas de API de los componentes de texto.

Clases de Componentes de Texto Swing

Clase	Descripción
JTextComponent	La superclase abastracta para todos los componentes de texto Swing.
JTextField	Un componente de texto plano, de una sóla línea y opcionalmente editable. Puedes ver Cómo usar Text Fields.
JPasswordField	Un componente de texto plano, de una sóla línea y opcionalmente editable que enmascara su contenido. Puedes ver Proporcionar un campo de Password.
JTextArea	Un componente de texto plano, multi línea y opcionalmente editable.
JEditorPane	Un componente de texto con estilo, multi línea y opcionalmente editable.
JTextPane	Un componente de texto con estilo, multi línea y opcionalmente editable con soporte para atributos.

Métodos de JTextComponent para Seleccionar Atributos

Método	Descripción
void setDisabledTextColor(Color) **Color getDisabledTextColor()**	Selecciona u obtiene el color usado para mostrar el texto cuando el componente está desactivado.
void setOpaque(boolean) **boolean getOpaque()**	Selecciona u obtiene si el componente de texto es completamente opaco.
void setMargin(Insets) **Insets getMargin()**	Selecciona u obtiene el margen entre el texto y el borde del componente de texto.
void setEditable(boolean) **boolean isEditable()**	Selecciona u obtiene si el usuario puede editar el texto del componente de texto.

Convertir Posiciones entre el Modelo y la Vista

Método	Descripción
int viewToModel(Point) (in **JTextComponent**)	Convierte el punto especificado en el sistema de coordenadas de la vista a una posición dentro del texto.
Rectangle modelToView(int) (in **JTextComponent**)	Convierte la posición especificada dentro del texto en un rectángulo en el sistema de coordenadas de la vista.

Clases e Interfaces que Representan Documentos

Clase o Interface	Descripción
Document (un interface)	Define el API que debe ser implementado por todos los documentos.
AbstractDocument (un interface)	Una implementación de una superclase abstracta del interface Document. Esta es la superclase para todos los documentos, proporcionada por el paquete de texto de Swing.
PlainDocument (una class)	Implementa el interface **Document**. El documento por defecto para los componentes de texto plano (text field, password field, y text area). Adicionalmente usada por editor pane y text pane cuando cargan texto plano o texto en un formato desconocido.
StyledDocument (un interface)	Un subinterdace de **Document**. Define el API que deben utilizar todos los documentos que soporten texto con estilo.
DefaultStyledDocument (una class)	Implementa el interface **StyledDocument**. El documento por defecto pàra los componentes de texto con estilo (editor pane y text pane).

Métodos Útiles para Trabajar con Documentos

Método	Descripción
setDocument(Document) **Document getDocument()** (en **JTextComponent**	Selecciona u obtiene el documento de un componente de texto.
Document createDefaultModel() (en **JTextField**	Sobreescribe este método en text field y sus subclases para crear un documento personalizado en lugar de el de por defecto **PlainDocument**. Crear un Campo Validado proporciona un ejemplo de cómo sobreescribir este

	método.
void insertString(int, String, AttributeSet) **void remove(int, int)** (en **Document**)	Estos métodos normalmente son sobreescritos por documentos personalizados. Para ver un ejemplo de un documento personalizado que sobreescribe ambos métodos, puedes ver Crear un Campo Validado.
void addDocumentListener(DocumentListener) **void removeDocumentListener(DocumentListener)** (en **Document**)	Añade o elimina un oyente de document a un documento. Puedes ver Escuchar los cambios en un Document.
void addUndoableEditListener(UndoableEditListener) **void removeUndoableEditListener(UndoableEditlistener)** (en **Document**)	Añade o elimina un oyente de undoable edit a un documento, Los oyentes de Undoable edit se usan en Implementar Deshacer y Repetir.
int getLength() **Position getStartPosition()** **Position getEndPosition()** **String getText(int, int)** (en **Document**)	Métodos de **Document** que devuelven información útil sobre el documento.
Object getProperty(Object) **void putProperty(Object, Object)** (in **Document**) **Dictionary getDocumentProperties()** **void setDocumentProperties(Dictionary)** (un **AbstractDocument**)	Un **Document** matiene un conjunto de propiedades que se pueden manipular con estos métodos. El ejemplo descrito en Usar un Oyente de Document en un Text Field usa una propiedad para nombrar los componentes de texto para compartir el oyente de document y poder identificar el documento de donde viene el evento.

Métodos de JTextComponent para Manipular la Selección Actual

Método	Descripción
String getSelectedText()	Obitene el texto actualmente seleccionado.
void selectAll() **void select(int, int)**	Selecciona todo el texto o selecciona el textgo de un rango.
void setSelectionStart(int) **void setSelectionEnd(int)** **int getSelectionStart()** **int getSelectionEnd()**	Selecciona u obtiene una extensión de la selección actual por índice.
void setSelectedTextColor(Color) **Color getSelectedTextColor()**	Selecciona u obtiene el color del texto seleccionado.
void setSelectionColor(Color) **Color getSelectionColor()**	Selecciona u obtiene el color de fondo del texto seleccionado.

Manipular Cursores y Marcadores de Selección

Interface, Clase, o Método	Descripción
Caret (un interface)	Define el API para objetos que representan un punto de inserción dentro de los documentos.
DefaultCaret (una clase)	El cursor por defecto usad por todos los componentes de texto.
void setCaret(Caret) **Caret getCaret()** (en **JTextComponent**)	Selecciona u obtiene el objeto cursor usado por un componente de texto.
void setCaretColor(Color) **Color getCaretColor()** (en **JTextComponent**)	Selecciona u obtiene el color del cursor.
void setCaretPosition(Position) **void moveCaretPosition(int)** **Position getCaretPosition()** (en **JTextComponent**)	Selecciona u obtiene la posición actual del cursor dentro del documento.
void addCaretListener(CaretListener) **void removeCaretListener(CaretListener)** (en **JTextComponent**)	Añade o elimina un oyente de caret al componente de texto.
Highlighter (un interface)	Define el API para objetos usados para iluminar la selección actual.
DefaultHighlighter (una class)	El iluminador por defecto usado por todos los componentes de texto.
void setHighlighter(Highlighter) **Highlighter getHighlighter()** (un **JTextComponent**)	Selecciona u obtiene el iluminador usado por un componente de texto.

Comandos de Edición de Texto

Clase o Método	Descripción
void cut() **void copy()** **void paste()** **void replaceSelection(String)** (en **JTextComponent**)	Curta, copia y pega texto usando el sistema del portapapeles.
EditorKit (una clase)	Edita, lee y escribe texto en un formato particular.

DefaultEditorKit (una clase)	Una subclase concreta de **EditorKit** que proporciona capacidades de edición de texto básico.
StyledEditorKit (una clase)	Una subclase de **Default EditorKit** que proporciona capacidades adicionales de edición para texto con estilo.
String xxxxAction (en **DefaultEditorKit**)	Los nombres de todas las acciones soportadas por el editor kit por defecto.
BeepAction **CopyAction** **CutAction** **DefaultKeyTypedAction** **InsertBreakAction** **InsertContentAction** **InsertTabAction** **PasteAction** (en **DefaultEditorKit**)	Una colección de clases internas que implementan varios comandos de edición de texto.
AlignmentAction **BoldAction** **FontFamilyAction** **FontSizeAction** **ForegroundAction** **ItalicAction** **StyledTextAction** **UnderlineAction** (en **StyledEditorKit**)	Una colección de clases internas que implementan varios comandos de edición para texto con estilo.
Action[] getActions() (en **JTextComponent**)	Obtiene las acciones soportadas por este componente. Este método obtiene un array de actions desde el editor kit, si el componente usa alguno.

Unir Pulsaciones y Acciones

Interface o Método	Descripción
Keymap (un interface)	Un interface para menejar un conjunto de uniones de teclas. Una unión de tecla está representada por un pareja pulsación/acción.
Keymap addKeymap(nm, Keymap) **Keymap removeKeymap(nm)** **Keymap getKeymap(nm)**	Añade o elimina un mapa de teclado del árbol de mapas. También obtiene el mapa de teclado por su nombre. Observa que estos son métodos de clase. El árbol de mapas de teclado es compartido por todos los componentes de texto.

(en **JTextComponent**)	
void loadKeymap(Keymap, KeyBinding[], Action[]) (en **JTextComponent**)	Añade un conjunto de uniones d eteclas al mapa de teclado especificado. Este es un método de clase.
void setKeymap(Keymap) **Keymap getKeymap()** (en **JTextComponent**)	Selecciona u obtiene el mapa de teclado activo actualmente para un componente de texto particular.
void addActionForKeyStroke(KeyStroke, Action) **Action getAction(KeyStroke)** **KeyStroke[] getKeyStrokesForAction(Action)** (en **Keymap**)	Selecciona u obtiene las uniones pulsación/acción de un mapa de teclado.
boolean isLocallyDefined(KeyStroke) (en **Keymap**)	Obtiene si la pulsación especificada esta unida a una acción en el mapa de teclado.
void removeKeyStrokeBinding(KeyStroke) **void removeBindings()** (en **Keymap**)	Elimina una o todas las uniones de teclas del mapa de teclado.
void setDefaultAction(Action) **Action getDefaultAction()** (en **Keymap**)	Selecciona u obtiene la acción por defecto. Esta acción se dispara si una pulsación no está explícitamente unida a una acción.
Action[] getBoundActions() **KeyStroke[] getBoundKeyStrokes()** (en **Keymap**)	Obtiene un array que contiene todas las uniones de un mapa de teclado.

Leer y Escribir Texto

Método	Descripción
void JTextComponent.read(Reader, Object) **void JTextComponent.write(Writer)** (en **JTextComponent**)	Lee o escribe texto.
void read(Reader, Document, int) **void read(InputStream, Document, int)** (en **EditorKit**)	Lee texto desde un stream a un documento.
void write(Writer, Document, int, int) **void write(OutputStream, Document, int, int)** (en **EditorKit**)	Escribe texto desde un documento a un stream.

API para Mostrar Texto de una URL

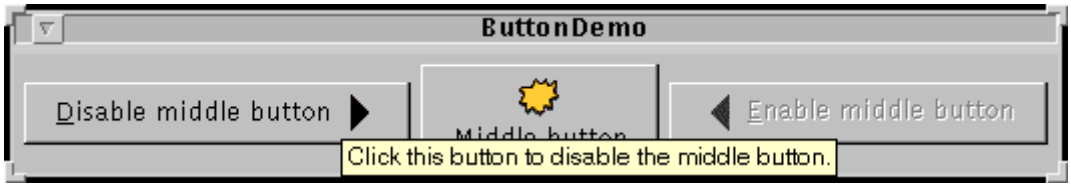
Método o Constructor	Descripción
JEditorPane(URL) **JEditorPane(String)** (en **JEditorPane**)	Crea un editor pane cargado con el texto de la URL especificada.
setPage(URL) **setPage(String)** (en **JEditorPane**)	Carga un editor pane (o text pane) con el texto de la URL especificada.
URL getPage() (en **JEditorPane**)	Obtiene la URL de la página actual de editor pane (o text pane).

¿Cómo usar ToolTip?

Crear un tool tip para cualquier **JComponent** es fácil. Sólo dmeos usar el método **setToolTipText** para configurar un tool tip para el componente. Por ejemplo, para añadir tool tips a tres botones, sólo tenemos que añadir tres líneas de código.

```
b1.setToolTipText("Click this button to disable the middle button.");
b2.setToolTipText("This middle button does nothing when you click it.");
b3.setToolTipText("Click this button to enable the middle button.");
```

Cuando el usuario del programa para el cursor sobre cualquiera de los botones, aparece el tool tip dle botón. Puedes ver esto ejecutando el ejemplo **ButtonDemo**, que se explicó en Cómo usar Buttons. Aquí tenemos una imagen del tool tip que aparece cuando el cursor se para sobre el botón de la izquierda en **ButtonDemo**.



[Por favor, imagínate un cursor sobre el botón. Gracias.]

El API de Tool Tip

La mayoría del API que necesitas para usar tool tips está en **JComponent**, y así lo heredan todos los componentes Swing (excepto los contenedores de alto nivel). Este API se cubre en una tabla más adelante en esta sección.

Más API de tool-tip se encuentra en clases individuales como **JTabbedPane**. Cada página de componente tiene información sobre su API de tool-tip, si existe.

Si quieres evitar o personalizar el manejo por defecto de tooltips, probablemente tendrás que tratar directamente con **JToolTip** o **ToolTipManager**

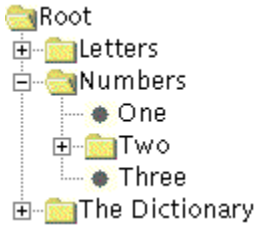
El API de Tool Tip en JComponent

Método	Propósito
setToolTipText(String) (en **JComponent**)	Si el string especificado no es nulo, esté métod registra el componente para tener un tooltip y hace que el tool-tip cuando se muestre tenga el texto especificado. Si el argumento es null, desactiva el tool-tip para este componente.
String getToolTipText() (en **JComponent**)	Devuelve el string que fue especificado anteriormente con **setToolTipText**.
String getToolTipText(MouseEvent) (en **JComponent**)	Por defecto devuelve el mismo valor devuelto por **getToolTipText()**. Componentes multi-parte como **JTabbedPane**, **JTable**, y **JTree** sobreescriben este método para devolver un string asocuado con la posición de ratón. Por ejemplo, cada pestaña en un tabbed pane puede tener un tool-tip diferente.
setToolTipLocation(Point)	Selecciona u obtiene la posición (en el sistema de coordenadas del componente

Point getToolTipLocation() (en **JComponent**)	recibido) donde aparecerá la esquina superior izquierda del tool-tip. El valor por defecto es nulo, lo que le dice a Swing que elija una posición.

¿Cómo usar Tree?

Con la clase **JTree**, se puede mostrar un árbol de datos. **JTree** realmente no contiene datos, simplemente es un vista de ellos. Aquí tienes una imagen de un árbol.



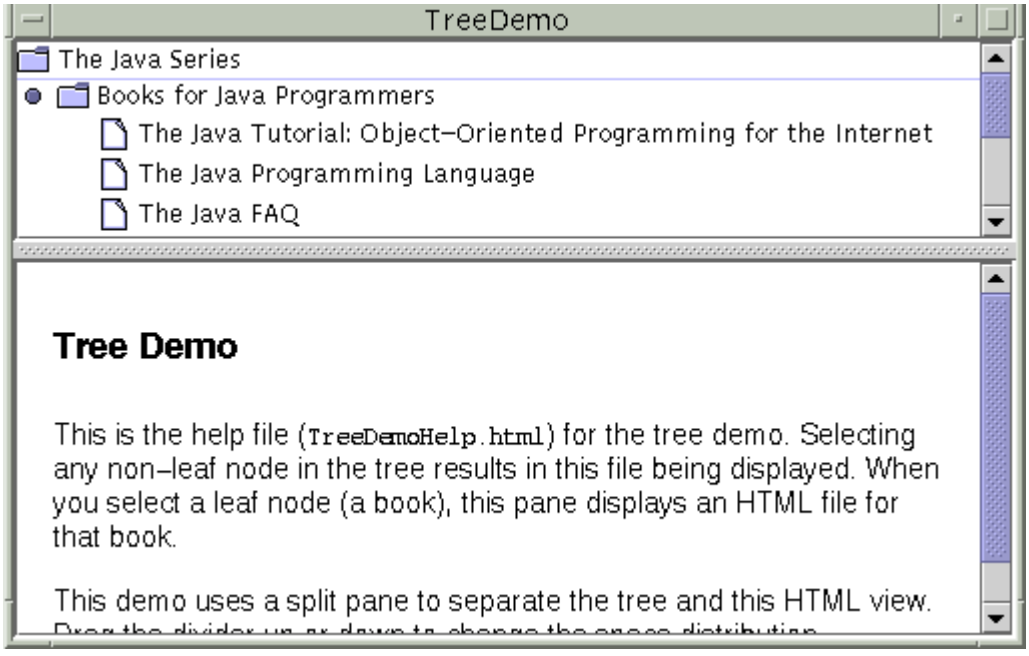
Como muestra la figura anterior, **JTree** muestra los datos verticalmente. Cada fila contiene exactamente un ítem de datos (llamado un nodo). Cada árbol tiene un nodo raíz (llamado Root en la figura anterior, del que descienden todos los nodos. Los nodos que no pueden tener hijos se llaman nodos leaf (hoja). En la figura anterior, el aspecto-y-comportamiento marca los nodos hojas con un círculo.

Los nodos que no sean hojas pueden tener cualquier número de hijos, o incluso no tenerlos. En la figura anterior, el aspecto-y-comportamiento marca los nodos que no son hojas con un carpeta. Normalmente el usuario puede expandir y contraer los nodos que no son hojas -- haciendo que sus hijos sena visibles o invisibles -- pulsando sobre él. Por defecto, los nodos que no son honas empiezan contraidos.

Cuando se inicializa un árbo, se crea un ejemplar de **TreeNode** para cada nodo del árbol, incluyendo el raíz. Cada nodo que no tenga hijos es una hoja. Para hacer que un nodo sin hijos no sea una hoja, se llama al método **setAllowsChildren(true)** sobre él.

Crear un Árbol que Reaccione a las Selecciones

Aquí hay una imagen de una aplicación, en cuya mitad superior se muestra un árbol en un scroll pane.



Intenta esto:

1. Compila y ejecuta la aplicación. El fichero fuente es **TreeDemo.java**.
2. Expande un nodo

Puedes hacer esto pulsando sobre el círculo que hay a la izquierda del ítem.

3. Selecciona un nodo.

   Puedes hacer esto pulsando sobre el texto del nodo o el icono que hay justo a la izquierda. El fichero mostrada en la parte inferior de la ventana muestra un fichero que refleja el estado actual del nodo seleccionado.

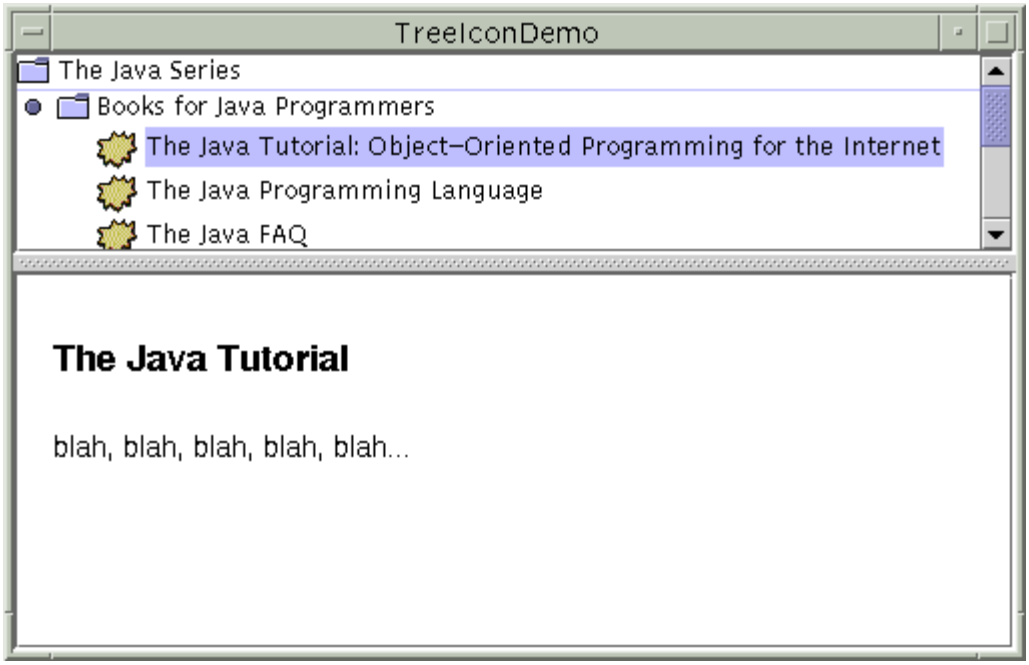
Abajo tenemos el código **TreeDemo.java** que implementa el árbol del ejemplo anterior.

```
public TreeDemo() {
    ...
    //Crea los nodos.
    DefaultMutableTreeNode top = new
                               DefaultMutableTreeNode("The Java Series");
    createNodes(top);

    //Crea un árbol que permite una selección a la vez.
    JTree tree = new JTree(top);
    tree.getSelectionModel().setSelectionMode
            (TreeSelectionModel.SINGLE_TREE_SELECTION);

    //Escucha cuando cambia la selección.
    tree.addTreeSelectionListener(new TreeSelectionListener() {
        public void valueChanged(TreeSelectionEvent e) {
            DefaultMutableTreeNode node = (DefaultMutableTreeNode)
                               (e.getPath().getLastPathComponent());
            Object nodeInfo = node.getUserObject();
            if (node.isLeaf()) {
                BookInfo book = (BookInfo)nodeInfo;
                displayURL(book.bookURL);
            } else {
                displayURL(helpURL);
            }
        }
    });

    //Crea el scroll pane y le añade el árbol.
    JScrollPane treeView = new JScrollPane(tree);
    ...
    //Añade los scroll panes a un split pane.
    JSplitPane splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
                                          treeView, htmlView);
    ...
}

private class BookInfo {
    public String bookName;
    public URL bookURL;

    public BookInfo(String book, String filename) {
        bookName = book;
        ./Set bookURL...
    }

    //Esto es lo que se mostrará en el árbol.
    public String toString() {
        return bookName;
    }
}
...
private void createNodes(DefaultMutableTreeNode top) {
    DefaultMutableTreeNode category = null;
    DefaultMutableTreeNode book = null;

    category = new DefaultMutableTreeNode("Books for Java Programmers");
    top.add(category);

    //Tutorial
    book = new DefaultMutableTreeNode(new BookInfo
        ("The Java Tutorial: Object-Oriented Programming for the Internet",
        "tutorial.html"));
    category.add(book);
    ...
    category = new DefaultMutableTreeNode("Books for Java Implementers");
    top.add(category);

    //VM
    book = new DefaultMutableTreeNode(new BookInfo
        ("The Java Virtual Machine Specification",
         "vm.html"));
    category.add(book);

    //Language Spec
    book = new DefaultMutableTreeNode(new BookInfo
        ("The Java Language Specification",
         "jls.html"));
    category.add(book);
}
```

Personalizar la visualización de un Árbol

Un árbol usa un unico renderizador para dibujar todos sus nodos. Por defecto, este renderizador es un ejemplar de **DefaultTreeCellRenderer**.

Se puede personalizar fácilmente la forma en que **DefaultTreeCellRenderer** dibuja los nodos. Por ejemplo, contiene métodos que permiten seleccionar los iconos usados para los nodos del árbol. Para personalizar el renderizador, sólo debemos crear un ejemplar de **DefaultTreeCellRenderer**, llamar a alguno de sus métodos **setXxx**, y luego hacer que el ejemplar sea el renderizador del árbol. La siguiente figura muestra una aplicación que ha cambiado su renderizador como que use un icono de hoja personalizado.

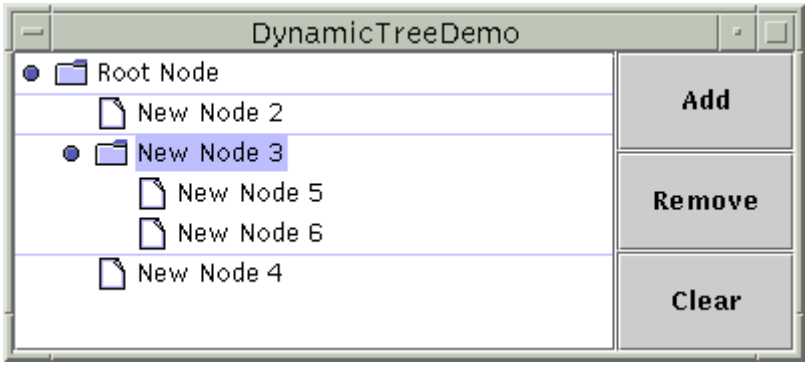


Aquí está el código que seleccion el icono (el programa completo es **TreeIconDemo.java**).

```
DefaultTreeCellRenderer renderer = new DefaultTreeCellRenderer();
renderer.setLeafIcon(new ImageIcon("images/middle.gif"));
tree.setCellRenderer(renderer);
```

Cambiar Dinámicamente un Árbol

La siguiente figura muestra una aplicación que nos permite añadir nodos al árbol de forma dinámica. También podemos editar el texto de cada nodo.



Puedes encontrar el código en **DynamicTreeDemo.java** y **DynamicTree.java**.

Ejemplos de Manejo de Eventos

Aquí un applet que ilustra el manejo de eventos. Sólo contiene un botón que pita cuando se pulsa sobre él.

Esta es una imagen del GUI del applet. Para ejecutar el applet pulsa sobre la imagen. El applet aparecerá en una nueva ventana del navegador.

Puedes encontrar el programa completo en **Beeper.java**. Aquí está el código que implementa el manejo de eventos del botón.

```
public class Beeper ... implements ActionListener {
    ...
    //where initialization occurs:
        button.addActionListener(this);
    ...
    public void actionPerformed(ActionEvent e) {
        ./Make a beep sound...
    }
}
```

¿No es sencillo? La clase **Beeper** implementa el interface **ActionListener**, que contiene un método: **actionPerformed**. Como **Beeper** implementa **ActionListener**, un objeto **Beeper** puede registrar un oyente para los eventos action que genere el botón. Una vez que **Beeper** ha sido registrado usando el método **addActionListener** de **Button**, cada que se pulse el botón se llamará al método **actionPerformed** de **Beeper**.

Un ejemplo más complejo

El modelo de eventos, que hemos visto en su forma más simple en el ejemplo anterior, es bastante poderoso y flexible. Cualquier número de objetos oyentes de eventos puede escuchar todas las clases de eventos desde cualquier número de objetos fuentes de eventos. Por ejmplo, un programa podría crear un oyente por cada objeto fuente. O un programa podría crear un sólo oyente para todos los eventos para todas las fuentes. Incluso un programa puede tener más de un oyente para una sóla clase de evento de una sola fuente de eventos.

El siguiente applet ofrece un ejemplo de uso de múltiples oyentes por objeto. El applet contiene dos fuentes de eventos (ejemplares de **JButton**) y dos oyentes de eventos. Uno de los oyentes (un ejemplar de la clase llamada **MultiListener**) escucha los eventos de los dos botones. Cuando recibe un evento, añade el evento "action command" (el texto de la etiqueta del botón) en la parte superior del área de texto. El segundo oyente (un ejemplar de la clase llamada **Eavesdropper**) esucha los eventos de uno de los botones. Cuando recibe un evento, añade el action command en la parte inferior del área de texto.

Esta es una imagen del GUI del applet. Para ejecutar el applet pulsa sobre la imagen. El applet aparecerá en una nueva ventana del navegador.

Puedes encontrar el programa completo en **MultiListener.java**. Aquí sólo tenemos el código que implementa el manejo de eventos de los botones.

```
public class MultiListener ... implements ActionListener {
    ...
    //where initialization occurs:
        button1.addActionListener(this);
        button2.addActionListener(this);

        button2.addActionListener(new Eavesdropper(bottomTextArea));
    }

    public void actionPerformed(ActionEvent e) {
        topTextArea.append(e.getActionCommand() + newline);
    }
}
```

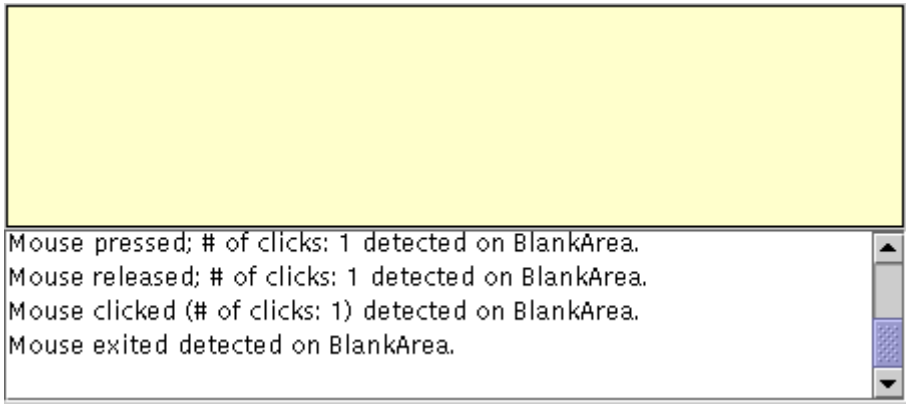
```
class Eavesdropper implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e) {
        myTextArea.append(e.getActionCommand() + newline);
    }
}
```

En el código anterior, tanto **MultiListener** como **Eavesdropper** implementan el interface **ActionListener** y se registran cómo oyentes de action usando el método **addActionListener** de **JButton**. Lás implementaciones que ambas clases hacen del método **actionPerformed** son similares: simplemente añaden el action command del evento al área de texto.

Un Ejemplo de Manejo de Otro Tipo de Evento

Hasta ahora, la única clase de eventos que hemos visto son los eventos action. Echemos un vistazo a un programa que maneja otros tipos de eventos: eventos del ratón.

El siguiente applet muestra un área rectangular y un área de texto, cuando ocurre un evento del ratón -- una pulsación, liberación, entrar o salir -- dentro del área rectangular (**BlankArea**) o su contenedor (**MouseEventDemo**), el área de texto mostrará una cadena describiendo el evento.



Esta es una imagen del GUI del applet. Para ejecutar el applet pulsa sobre la imagen. El applet aparecerá en una nueva ventana del navegador.

Puedes encontrar el programa completo en **MouseEventDemo.java** y **BlankArea.java**. Aquí sólo tenemos el código que implementa el manejo de eventos.

```
public class MouseEventDemo ...  implements MouseListener {
    ...
    //where initialization occurs:
        //Register for mouse events on blankArea and applet
        blankArea.addMouseListener(this);
        addMouseListener(this);
    }

    public void mousePressed(MouseEvent e) {
       saySomething("Mouse pressed; # of clicks: "
                    + e.getClickCount(), e);
    }

    public void mouseReleased(MouseEvent e) {
       saySomething("Mouse released; # of clicks: "
                    + e.getClickCount(), e);
    }

    public void mouseEntered(MouseEvent e) {
       saySomething("Mouse entered", e);
    }

    public void mouseExited(MouseEvent e) {
       saySomething("Mouse exited", e);
    }

    public void mouseClicked(MouseEvent e) {
       saySomething("Mouse clicked (# of clicks: "
                    + e.getClickCount() + ")", e);
    }

    void saySomething(String eventDescription, MouseEvent e) {
        textArea.append(eventDescription + " detected on "
                        + e.getComponent().getClass().getName()
                        + "." + newline);
    }
}
```

Podrás ver el código explicado en Cómo Escribir un Oyente de Ratón, más adelante en esta sección.

Reglas Generales para Escribir Oyentes de Eventos

Cuando el AWT llama a un método oyente de evento, este método se ejecuta en el thread de eventos del AWT. Como todos los demás manejos de eventos y los métodos de dibujo se ejecutan en el mismo thread, un lento método de manejo de eventos puede hacer que el programa parezca no responder y puede relentizar el propio dibujado.

Importante: Asegúrate de que tus métodos de oyentes de eventos se ejecutan rápidamente!

Si necesitas realizar alguna operación lenta como resultado de un evento, hazlo arrancado otro thread (o enviando una petición a otro thread) para realizar la operación.

La siguiente subsección presenta el ascentro de todas las clases evento del AWT, la clase AWTEvent. Depués, Usar adaptadores y Clases Internas para Manejar Eventos AWT te ofrecerá trucos para evitar un código borroso..

La clase AWTEvent

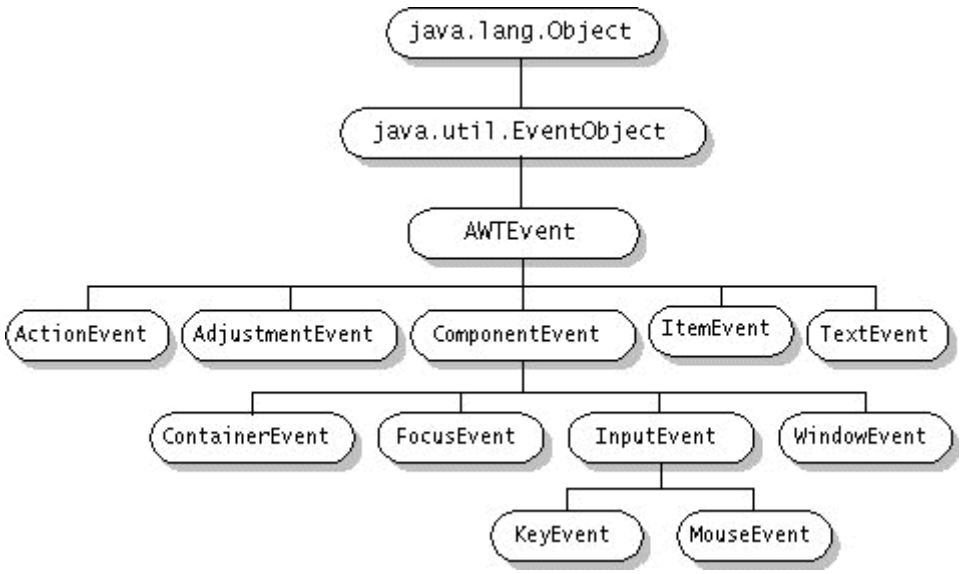
Todo método en un interface oyente de evento del AWT tiene un sólo argumento, un ejemplar de una clase que descienda de la clase java.awt.AWTEvent. Esta clase no define ningún método o API que podamos usar normalmente. Sin embargo, hereda un método muy útil de la clase java.util.EventObject.

Object getSource()
devuelve el objeto que generó el evento.

Observa que el método getSource devuelve un Object. Siempre que sea posible, las subclases de AWTEvent definen métodos similares con tipos de retorno más restrictivos. Por ejemplo, la clase ComponentEvent define un método getComponent que devuelve el Component que generó el evento.

LA figura 68 muestra la herencia de clases de AWTevent. Como se puede ver muchas, pero no todas, las clases de eventos del AWT descienden de la clase ComponentEvent.

El árbol de herencia de las clases de eventos AWT.



Eventos Estandard del AWT

El AWT define cerca de una docena de tipos de oyentes de eventos. Se puede decir los tipos de eventos que puede generar un componente mirando las clases de oyentes de eventos que podemos registrar en él. Por ejemplo, la clase Component define estos métodos de registros de oyentes.

- addComponentListener
- addFocusListener
- addKeyListener
- addMouseListener
- addMouseMotionListener

Por lo tanto, toda clase que descienda de Component soporta oyentes de component, focus, key, mouse y mouse-motion. Sin embargo la mayoría de los ejemplares de Component no generan estos eventos; un componnete genera sólo aquellos eventos para los que los oyentes se han registrado. Por ejemplo, so un oyente de mouse se regostra en un componente particular, y el componente no tiene otros oyentes, entonces el componente sólo generara eventos de mouse.

Usar Adaptadores y Clases Internas para Manejar Eventos

Esta sección explica cómo utilizar adaptadores y clases internas para reducir la sobrecarga del código.

La mayoría de los interfaces de oyentes, al contrario que **ActionListener**, contienen más de un método. Por ejemplo, el interface **MouseListener** contiene cinco métodos:**mousePressed**, **mouseReleased**, **mouseEntered**, **mouseExited**, y **mouseClicked**. Incluso si sólo te importan las pulsaciones, si tu clase implementa directamente **MouseListener**, entonces debes implementar los cinco métodos de **MouseListener**. Aquellos métodos de eventos que no te interesan pueden tener los cuerpos vacíos. Aquí hay un ejemplo.

```
//An example with cluttered but valid code.
public class MyClass implements MouseListener {
    ...
               someObject.addMouseListener(this);
    ...
    /* Empty method definition. */
    public void mousePressed(MouseEvent e) {
    }

    /* Empty method definition. */
    public void mouseReleased(MouseEvent e) {
    }

    /* Empty method definition. */
    public void mouseEntered(MouseEvent e) {
    }

    /* Empty method definition. */
    public void mouseExited(MouseEvent e) {
    }

    public void mouseClicked(MouseEvent e) {
               ./Event handler implementation goes here...
    }
}
```

Desafortunadamente, la colección de cuerpos de métodos vacios resultante puede resultar dura de leer y de mantener. Para ayudarnos a evitar este emborronamiento del código con cuerpos de métodos vacios, el AWT y Swing proporcionan una clase adapter por cada interface de oyente con más de un método. (Manejar Eventos Comunes lista todos los oyentes y sus adaptadores.) Por ejemplo, la clase **MouseAdapter** implmenta el interface **MouseListener**. Una clase adaptador implementa versiones vacías de todos los métodos del interface.

Para usar un adaptador se crea una subclase, en vez de implementar directamente un interface de oyente. Por ejemplo, extendiendo la clase **MouseAdapter**, nuestra clase hereda definiciones de vacías para los métodos que contiene **MouseListener**.

```
/*
 * An example of extending an adapter class instead of
 * directly implementing a listener interface.
 */
public class MyClass extends MouseAdapter {
    ...
               someObject.addMouseListener(this);
    ...
    public void mouseClicked(MouseEvent e) {
               ./Event handler implementation goes here...
    }
}
```

¿Qué pasa si no queremos que nuestras clases de manejo de eventos desciendan de una clase adaptador? Por ejemplo, supongamos que escribimos un applet, y queremos que nuestra subclase **Applet** contenga algún método para manejar eventos de ratón. Como el lenguaje Java no permite la herencia múltiple, nuestra clase no puede descender de las clases **Applet** y **MouseAdapter**. La solución es definir una clase interna -- una clase dentro de nuestra subclase **Applet** -- que descienda de la clase **MouseAdapter**,

```
//An example of using an inner class.
public class MyClass extends Applet {
    ...
               someObject.addMouseListener(new MyAdapter());
    ...
    class MyAdapter extends MouseAdapter {
        public void mouseClicked(MouseEvent e) {
                  ./Event handler implementation goes here...
        }
    }
}
```

Aquí hay otro ejemplo de uso de clases internas.

```
//An example of using an anonymous inner class.
public class MyClass extends Applet {
    ...
               someObject.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
                      ./Event handler implementation goes here...
            }
               });
    ...
    }
}
```

Las clases internas funcionan bien incluso si nuestro manejador de eventos necesita acceder a ejemplares de variables privadas de la clase que la encierra. Siempre que no declaremos una clase interna como **static**, se podrá referir a ejemplares de variables y métodos

como lo hace el resto de código que contiene la clase. Para hacer que una variable local esté disponible para una clase interna solo tenemos que hacer un copia de la variable como una variable local **final**.

Nota: Algunos compiladores 1.1 no permiten a las clases internas acceder a ejemplares de variables privados de la clase que la encierra. Un atajo es eliminar el especificador **private** de la declaración del ejemplar de la variable.

Eventos Generados por Componentes Swing

Esta sección sumariza los eventos que pueden ser generados por componentes Swing, enfocándose en los eventos que manejan típicamente los programas. Los eventos generados por componentes Swing se dividen en tres categorías.

Eventos que todos los componentes Swing pueden generar

Como todos los componentes Swing descienden de la clase **Component** del AWT, todo ellos soportan los siguientes eventos definidos enel AWT.

Component
: Notifica a los oyentes cambios en el tamaño, posición o visibilidad del componente.

Focus
: Notifica a los oyentes que el componente a ganado o perdido la posibilidad de recibir entreda desde el teclado.

Key
: Notifica a los oyentes las pulsaciones de teclas; sólo generado por el componene que tiene el foco del teclado.

Mouse
: Notifica a los oyentes las pulsaciones del ratón y los movimientos de entrada y salida del usuario en el área de dibujo del componente.

Mouse Motion
: Notifica a los oyentes cambios en la posición del cursor sobre el componente.

Aunque todos los componentes Swing descienden de la clase **Container** del aWT, muchos de ellos no son usados como contenedores. Por eso, técnicamente hablando cualquier componente Swing puede generar eventos container, que notifican a los oyentes que se ha añadido o eliminado un componente del contenedor. Sin embargo, hablando en forma real sólo los contenedores como los paneles, marcos, etc., generan eventos container.

Otros Eventos comunes

La siguiente tabla lista los eventos más comunmente manejados que varios componentes Swing pueden generar.

Nota: un asterisco '*' en un cabecera de columna indica un evento definido en el AWT. Todos los otros eventos de la tabla está definidos en **javax.swing.event**.

Componente Swing	action	caret	change	document	internal frame	item*	list selection	undoable edit	window*
ColorSelectionModel (**JColorChooser**'smodelo de seleccion por defecto.) **Nota:** Esta no es una subclase de **JComponent**!			X						
Document (**JTextComponent**'s modelo de datos.) **Nota:** Esta no es una sublcase de **JComponent**!				X				X	
JButton	X		X			X			
JCheckBox	X		X			X			
JComboBox	X					X			
JDialog									X
JEditorPane		X							
JFileChooser	X								
JFrame									X
JInternalFrame					X				
JList							X		
JMenuItem	X		X			X			
JOptionpane									X

JPasswordField	X	X							
JProgressBar			X						
JRadioButton	X		X			X			
JSlider			X						
JTabbedPane			X						
JTextArea		X							
JTextComponent		X							
JTextField	X	X							
JTextPane		X							
JToggleButton	X		X			X			
JViewport			X						
ListSelectionModel (**JList**'s modelo de selección por defecto.) **Nota:** Esta no es una subclase **JComponent**!							X		
Timer **Nota:** Esta not es una subclase **JComponent**!	X								

Eventos no manejados comunmente

Como recordatorio, esta sección lista otros eventos que los componentes Swing pueden generar pero que los programas típicos no necesitan manejar.

Todos los componentes que descienden de la clase **JComponent** pueden generar los eventos descritos en la siguiente lista.

Ancestor
: Un componente genera un evento Ancestor cuando uno de sus contenedores acenstros es añadido o eliminado de un contenedor, es ocultado, visualizado o movido. Este tipo de evento es una implementación detallada y generalmente puede ser ignorado.

Property Change
: Definido en **java.beans** los componentes Swing generan este tipo de eventos porque son compatibles con JavaBeans. Los Beans utilizan los eventos Change para implementar propiedades compartidas.

Vetoable Change
: Definido en **java.beans** los componentes Swing generan este tipo de eventos porque son compatibles con JavaBeans. Los Beans utilizan estos eventos para implementar propiedades restringidas.

La siguiente tabla lista todos los otos eventos definidos en **javax.swing.event** que no se han mencionado anteriormente.

Eventos de Editor de Celdas	Eventos de Teclas de Menú	Eventos de Expansión de Árboles
Hyperlink	Menu	Tree Model
List Data	Popup Menu	Tree Selection
Menu Drag Mouse	Table Model	Tree Will Expand

Manejar Eventos

Esta sección explica cómo podemos escribir un oyente para eventos que puedan ser generados por un componente Swing. Primero ofrece una introducción a los oyentes. Después, cada tipo de oyente se explica en su propia página.

En la tabla que sigue, cada fila describe un grupo de eventos particular correspondiente a un interface oyente. La primer columna ofrece el nombre el interface, con un enlace a la página del tutorial que lo describe. La segunda columna nombra la correspondiente clase adaptador, si existe. La tercera columna indica el paquete en que se definen el interface, la clase event y la case adaptador. La cuarta columna lista los métodos que contiene el interface.

Para ver los tipos de eventos que pueden generar los componentes Swing, puedes volver a la página. Eventos Generados por Componentes Swing.

Interface	Clase Adaptador	Paquete	Métodos
ActionListener	ninguna	**java.awt.event**	**actionPerformed**
CaretListener	ninguna	**javax.swing.event**	**caretUpdate**
ChangeListener	ninguna	**javax.swing.event**	**stateChanged**

ComponentListener	**ComponentAdapter**	**java.awt.event**	**componentHidden** **componentMoved** **componentResized** **componentShown**
ContainerListener	**ContainerAdapter**	**java.awt.event**	**componentAdded** **componentRemoved**
DocumentListener	ninguna	**javax.swing.event**	**changedUpdate** **insertUpdate** **removeUpdate**
FocusListener	**FocusAdapter**	**java.awt.event**	**focusGained** **focusLost**
InternalFrameListener	**InternalFrameAdapter**	**javax.swing.event**	**internalFrameActivated** **internalFrameClosed** **internalFrameClosing** **internalFrameDeactivated** **internalFrameDeiconified** **internalFrameIconified** **internalFrameOpened**
ItemListener	ninguna	**java.awt.event**	**itemStateChanged**
KeyListener	**KeyAdapter**	**java.awt.event**	**keyPressed** **keyReleased** **keyTyped**
ListSelectionListener	ninguna	**javax.swing.event**	**valueChanged**
MouseListener	**MouseAdapter** **MouseInputAdapter** *	**java.awt.event** **javax.swing.event**	**mouseClicked** **mouseEntered** **mouseExited** **mousePressed** **mouseReleased**
MouseMotionListener	**MouseMotionAdapter** **MouseInputAdapter** *	**java.awt.event** **javax.swing.event**	**mouseDragged** **mouseMoved**
UndoableEditListener	none	**javax.swing.event**	**undoableEditHappened**
WindowListener	**WindowAdapter**	**java.awt.event**	**windowActivated** **windowClosed** **windowClosing** **windowDeactivated**

			windowDeiconified **windowIconified** **windowOpened**

***** Swing proporciona la clase **MouseInputAdapter** por conveniencia. Implementa los interfaces **MouseListener** y **MouseMotionListener** haciendo más fácil para nosotros el manejo de ambos tipos de eventos.

Los eventos descritos en la tabla anterior pueden dividirse en dos grupos: eventos de **bajo nivel** y eventos **semánticos**. Los eventos de bajo nivel representan las ocurrencias del sistema windows o entradas de bajo nivel. Claramente, los eventos de ratón y de tecla -- ambos como resultado de la entrada directa del usuario -- son eventos de bajo nivel.

Los eventos component, container, focus, y window también son de bajo nivel. Los eventos Component permite seguir los cambios en la posición, el tamaño y visibilidad del componente. El evento Container permite conocer cuando se añade o elimina cualquier componente a un contenedor particular. El evento Focus indica cuando un componente gana o pierde el **foco del teclado** -- la habilidad de recibir caracteres pulsados en el teclado. Los eventos windows nos informan del estado básico de cualquier ventana, como un **Dialog** o un **Frame**.

Los eventos de ratón se dividen en dos grupos -- mouse motion y mouse -- por eso un objeto puede escuchar eventos de mouse como las pulsaciones sin necesidad de sobrecargar el sistema intentando conocer exactamente los movimientos del ratón, lo que tiende a ocurrir frecuentemente.

Los eventos semánticos incluyen los eventos action, change, document, e item. Estos eventos son el resultado de una interación específica del usuario con un componente expecífico. Por ejemplo, un botón genera un evento action cuando el usuario lo pulsa, y una lista genera un evento action cuando el usuario hace doble clicj sobre uno de sus ítems. Cuando un usuarui selecciona un ítem de entre un grupo de ítems (como una lista), se genera un evento item.

Oyente de Action

Los oyentes de Action son probablemente los más sencillos -- y los más comunes -- manejadores de eventos que para implementar. Se implementa un oyente de action para responder a una indicación del usuario de que alguna ación dependiende de la implementación debería ocurrir.

Cuando un usuario pulsa un button, elige un menu item o pulsa Return en un text field, ocurre un evento actio. El resultado es que se envía un mensaje **actionPerformed** a todos los oyentes de action que estén registrados con un componente en particular.

Métodos de Evento Action

El interface **ActionListener** contiene un sólo método, y no tiene la correspondiente clase adaptadora. Aquí está el único método de **ActionListener**.

void actionPerformed(ActionEvent)
Se le llama justo después de que el usuario informe al componente escuchado de que debería ocurrir una acción.

Ejemplos de Manejo de Eventos Action

Aquí está el código de manejo del evento action de un applet llamado **Beeper**.

```
public class Beeper ...  implements ActionListener {
    ...
    //where initialization occurs:
        button.addActionListener(this);
    ...
    public void actionPerformed(ActionEvent e) {
            Toolkit.getDefaultToolkit().beep();
    }
}
```

Hay algunos ejemplos más de ficheros fuentes que contienen oyentes de action.

- **ShowDocument.java**
- **ButtonDemo.java**
- **DialogDemo.java**
- **ListDialog.java**
- **MenuDemo.java**
- **TextDemo.java**

La clase ActionEvent

El método **actionPerformed** tiene un sólo parámetros, un objeto **ActionEvent**. La clase **ActionEvent** define dos métodos muy útiles.

String getActionCommand()
Devuelve el string asociado con esta acción. La mayoría de los objetos que generan actions soportan un método llamado **setActionCommand** que nos permite seleccionar este string. Si no lo seleccionamos explícitamente, se utiliza el texto mostrado en el componente. Para objetos con múltiples ítems y por lo tanto con múltiples posibles acciones, el comando de acción generalmente es el nombre del ítem seleccionado.

int getModifiers()
Devuelve un entero que representa las teclas modificadores que fueron pulsadas cuando ocurrió el evento action. Se pueden usar las constantes definidas en **ActionEvent** - **SHIFT_MASK**, **CTRL_MASK**, **META_MASK**, y **ALT_MASK** para determinar que teclas fueron pulsadas. Por ejemplo, si el usuario selecciona un ítem de menú con Shift la siguiente expresión será distinta de cero.

```
actionEvent.getModifiers() & ActionEvent.SHIFT_MASK
```

También es útil el método **getSource**, que **ActionEvent** hereda de **EventObject** por medio de **AWTEvent**.

Oyente de Caret

Los eventos de Caret ocurren cuando se mueve el cursor (caret = punto de insercion) en un componente de texto o cuando cambia la selección en un componente de texto. Se puede añadir un oyente de caret a un ejemplar de cualquiera de la subclase de **JTextComponent** con el método **addCaretListener**.

Si nuestro programa tiene un cursor personalizado, podríamos encontrar más conveniente añadir un oyente al objeto caret en vez de la componente de texto al que pertenece. Un cursor genera eventos change en vez de eventos caret, por eso necesitaremos escribir un oyente de change en vez de un puente de caret. Puedes ver Cómo escribir un Oyente de Change para más información.

Métodos de Evento Caret

El interface **CaretListener** sólo tiene un método y por lo tanto no tiene clase adaptadora.

void caretUpdate(CaretEvent)
Se le llama cuando se mueve el cursor de un componente de texto o cuando se modifica la selección en un componente de texto.

Ejemplos de Manejo de Eventos Caret

El ejemplo descrito en How to Use Text Components tiene un oyente de caret que muestra el estado de cursor y de la selección. Podrás encontrar el código fuente en **TextComponentDemo.java**.

La clase CaretEvent

El método **caretUpdate** tiene un sólo parámetro, un objeto **CaretEvent**. Para obtener el componente de texto que generó el evento, se usa el método **getSource** que **CaretEvent** hereda de **EventObject**.

La clase **CaretEvent** define dos métodos muy útiles.

int getDot()
Devuelve la posición actual del cursor. Si hay texto seleccionado, el cursor marca uno de los finales de la selección.

int getMark()
Devuelve el otro final de la selección. Si no hay nada seleccionado, el valor devuelto por este método es igual al devuelto por **getDot**

Oyente de Change

Los eventos Cange ocurren cuando un componente que tiene estado cambia éste. Por ejemplo, una barra deslizadora genera un evento change cuando usuario mueve su cursor.

Métodos de Evento Change

El interface **ChangeListener** tiene sólo un método y por eso no tiene la correspondiente clase adaptadora.

void stateChanged(ChangeEvent)
Se le llama cuando el componente escuchado cambia de estado.

Ejemplos de Manejo de Eventos Change

Aquí el código de manejo de un evento change de una aplicación llamada **SliderDemo**.

```
class SliderListener implements ChangeListener {
    public void stateChanged(ChangeEvent e) {
        JSlider source = (JSlider)e.getSource();
        if (!source.getValueIsAdjusting()) {
```

```
            int fps = (int)((JSlider)e.getSource()).getValue();
            if (fps == 0) {
                if (!frozen) stopAnimation();
            } else {
                delay = 1000 / fps;
                timer.setDelay(delay);
                if (frozen) startAnimation();
            }
        }
    }
}
```

El programa **SliderDemo** se describe en Cómo usar Barras deslizadoras. Puedes encontrar el programa completo en **SliderDemo.java**.

Aquí hay unos cuandos ficheros fuentes que también usan oyentes de change.

- **SliderDemo2.java**
- **ColorChooserDemo.java**
- **ColorChooserDemo2.java**

La clase ChangeEvent

El método **stateChanged** tiene un sólo parámetro: un objeto **ChangeEvent**. Para obtener el componente que generó el evento se usa el método **getSource** que **ChangeEvent** hereda de **EventObject**. LA clase **ChangeEvent** no define métodos adicionales.

Oyente de Component

Un objeto **Component** genera uno o más eventos componente justo después de que el componente haya sido ocultado, hecho visible, movido o redimensionado. Un ejemplo de oyente de component podría estar en una herramienta de construcción de GUI que muestra información sobre el tamaño del componente seleccionado, y que necesita saber cuando cambia el tamaño del componente. Normalmente no se necesita manejar eventos componente para controlar la distribución básica ni el redibujado.

Los eventos component-hidden y component-visible sólo ocurren como resultados de llamadas al método **setVisible** de **Component**. Por ejmplo, una ventana podría ser miniaturizada en un icono iconificada) sin que se generará ningún evento component-hidden.

Métodos de Evento Component

El interface **ComponentListener** y su correspondiente clase adpatador **ComponentAdapter**, contienen cuatro métodos.

void componentHidden(ComponentEvent)
: Se le llama después de ocultar el componente esuchado como resultado de una llamada al método **setVisible**.

void componentMoved(ComponentEvent)
: Se le llama cuando el componente escuchado se mueve en relación a su contenedor. Por ejemplo, si se mueve una ventana, genera un evento component-moved, pero el componente que la contiene no lo genera.

void componentResized(ComponentEvent)
: Se le llama después de que cambie el tamaño del componente escuchado.

void componentShown(ComponentEvent)
: Se le llama después de que componente escuchado se vuelva visible como resultado de una llamada al método **setVisible**.

Ejemplos de Manejo de Eventos Component

El siguiente applet demuestra los eventos component. El applet contiene un botón que trae una ventana (**JFrame**). La ventana contiene un panel que tiene una etiqueta y un checkbox. El checkbox controla si la etiqueta es visible. Cuando abandonamos la página del applet, la ventana desaparece, y reaparece cuando volvemos a la página del applet. El área de texto muestra un mensaje cada vez que la ventana, el panel, la etiqueta o el checkbox generan un evento component.

Esta es una imagen del GUI del applet. Para ejecutarlo, pulsa sobre ella. El applet aparecerá en una nueva ventana de tu navegador.

Prueba esto:

1. **Pulsa el botón llamado "Start playing...".**

   La ventana se desplegará, generando uno o más eventos component-shown y component-moved.

2. **Pulsa el checkbox para ocultar la etiqueta.**

   La etiqueta genera un evento component-hidden.

3. **Pulsa de nuevo el checkbox para mostrar la etiqueta.**

   La etiqueta genera un evento component-shown.

4. **Minimiza y maxima la ventana que contiene la etiqueta.**

   No se obtienen eventos component-hidden ni component-shown. Si queremos ser informados de los eventos de iconificación deberíamos usar un oyente de window.

5. **Cambia el tamaño de la ventana que contiene la etiqueta.**

   Veremos los eventos component-resized (y posiblemente) de los cuatro componentes - etiqueta, checkbox, panel y ventana. Si la controlador de distribución de la ventana y el panel no hacen que cada componente sean tan anchos como sea posible, el panel, la etiqueta y el checkbox podrían no haber sido redimensionados.

El código del applet los puedes encontrar en **ComponentEventDemo.java**. Aquí puedes ver sólo el código relacionado con el manejo de eventos component.

```
public class ComponentEventDemo ... implements ComponentListener {
    ...
    //where initialization occurs:
        aFrame = new JFrame("A Frame");
        ComponentPanel p = new ComponentPanel(this);
        aFrame.addComponentListener(this);
        p.addComponentListener(this);
    ...

    public void componentHidden(ComponentEvent e) {
                displayMessage("componentHidden event from "
                                    + e.getComponent().getClass().getName());
    }

    public void componentMoved(ComponentEvent e) {
                displayMessage("componentMoved event from "
                                    + e.getComponent().getClass().getName());
    }

    public void componentResized(ComponentEvent e) {
                displayMessage("componentResized event from "
                                    + e.getComponent().getClass().getName());
    }

    public void componentShown(ComponentEvent e) {
                displayMessage("componentShown event from "
                                    + e.getComponent().getClass().getName());
    }
}

class ComponentPanel extends JPanel ... {
    ...
    ComponentPanel(ComponentEventDemo listener) {
        ...//after creating the label and checkbox:
        label.addComponentListener(listener);
        checkbox.addComponentListener(listener);
    }
    ...
}
```

La clase ComponentEvent

Cada método de evento Component tiene un sólo parámetro, un objeto **ComponentEvent**. La clase **ComponentEvent** define los siguientes métodos muy útiles.

Component getComponent()
Devuelve el componente que generó el evento. Podemos utilizarlo en lugar del método **getSource**.

Oyente de Container

El eventos container son generados por un **Container** justo después de que se haya añadido o eliminado un componente. Estos eventos son **sólo** para notificación -- no es necesario que esté presente un oyente de container para los componente sean añadidos o eliminados satisfactoriamente.

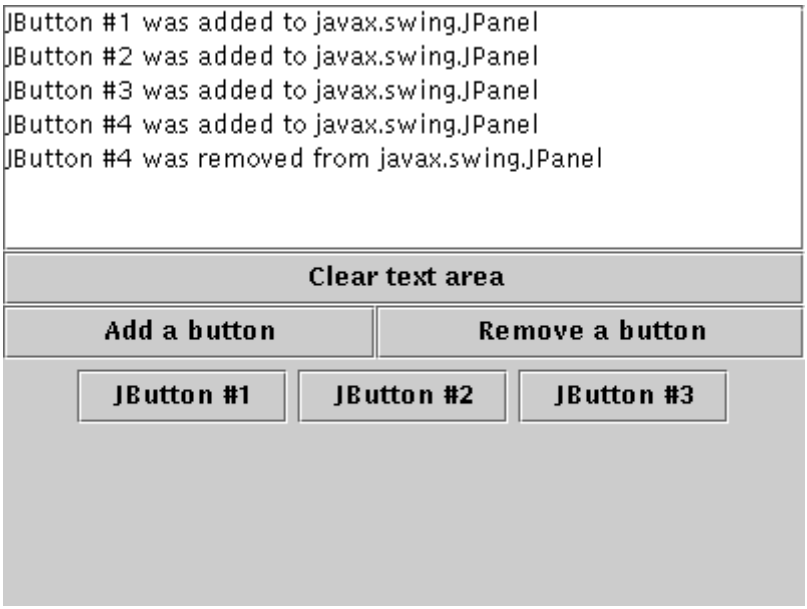
Métodos de Evento Container

El interface **ContainerListener** y su correspondiente clase adaptadora, **ContainerAdapter**, contienen dos métodos.

void componentAdded(ContainerEvent)
Se le llama después de que se la añada un componente al contenedor escuchado.
void componentRemoved(ContainerEvent)
Se le llama después de que se elimine un componente del contenedor escuchado.

Ejemplos de Manejo de Eventos Container

El siguiene applet demuestra los eventos container. Pulsando sobre "Add a button" o "Remove a button", podemos añadir o eliminar componenten de un panel que hay en la parte inferior del applet. Cada vez que se añade o elimina un componente al panel, éste dispara un evento container, y se le notifica a los oyentes del contenedor del panel. El oyente muestra mensajes descriptivos en el área de texto que hay en la parte superior del applet.



Esta es una imagen del GUI del applet. Para ejecutarlo, pulsa sobre ella. El applet aparecerá en una nueva ventana de tu navegador.

Prueba esto:

1. **Pulsa el botón "Add a button".**

   Verás que aparece un botón en la parte inferior del applet. El oyente de container (en este ejemplo, un ejemplar de **ContainerEventDemo**) reacciona ante el resultante evento component-added mostrando "Button #1 was added to java.awt.Panel" en la parte superior del applet.

2. **Pulsa el botón "Remove a button".**

   Esto elimina el último botón añadido al panel, haciendo que el oyente de container reciba un evento de component-removed.

Puedes encontrar el código del applet en ContainerEventDemo.java. Aquí está el código que maneja los eventos container.

```
public class ContainerEventDemo ... implements ContainerListener ... {
    ...//where initialization occurs:
               buttonPanel = new JPanel();
               buttonPanel.addContainerListener(this);
    ...
    public void componentAdded(ContainerEvent e) {
               displayMessage(" added to ", e);
    }

    public void componentRemoved(ContainerEvent e) {
               displayMessage(" removed from ", e);
    }

    void displayMessage(String action, ContainerEvent e) {
               display.append(((JButton)e.getChild()).getText()
                                   + " was"
                                   + action
                                   + e.getContainer().getClass().getName()
                                   + newline);
    }
    ...
}
```

La clase ContainerEvent

Cada método del evento container tiene un sólo parámetro, un objeto **ContainerEvent**. La clase **ContainerEvent** define dos métodos útiles.

Component getChild()
: Devuelve el componente cuya adición o eliminación disparó este evento.

Container getContainer()
: Devuelve el contenedor que generó este evento. Se peude usar en lugar del método **getSource**.

Oyente de Document

Un componente de texto Swing usa un **Document** para contener y editar un texto. Los eventos Document ocurren cuando el contenido de un documento cambia de alguna forma. Se le añade el oyente de Document al documento del componente, en vez de al propio componente.

Métodos de Evento Document

El interface **DocumentListener** contiene estos tres métodos.

void changedUpdate(DocumentEvent)
: Se le llama cuando se modifica el estilo o algo del texto. Este tipo de eventos sólo se generan desde un **StyledDocument**--un **PlainDocument** no genera este tipo de eventos.

void insertUpdate(DocumentEvent)
: Se le llama cuando se inserta texto en el documento escuchado.

void removeUpdate(DocumentEvent)
: Se le llama cuando se elimina texto del documento escuchado.

Ejemplos de Manejo de Eventos Document

Dos ejemplos descritos en otras secciones tienen oyentes de document.

- El descrito en Escuchar los Cambios en un Documento actualiza un diario de cambios cada vez que cambia el texto del documento. El código fuente del ejemplo está en **TextComponentDemo.java**.
- Y el descrito en Usar un Oyente de Document en un Text Field actualiza un valor numérico basado en otros valores introducidos en campos de texto por el usuario. Puedes encontrar el código fuente en **TextFieldDemo.java**.

Ambas sección hacen un importante apunte que merece la pena repetir aquí.

Nunca debemos modificar el contenido de un documento desde dentro de un oyente de document. El programa se podría quedar bloqueado. Para evitarlo, podemos usar un documento personalizado para el componente de texto.

El interface DocumentEvent

Cada método de evento document tiene un sólo parámetros, un ejemplar de una clase que implemente el interface **DocumentEvent**. Típicamente, el objeto pasado a este método será un ejemplar de **DefaultDocumentEvent** que está definido en **AbstractDocument**.

Para obener el documento que generó el evento, podemos usar el método **getDocument** de **DocumentEvent**. Observa que **DocumentEvent** no desciende de **EventObject** como las otras clases de eventos. Por lo tanto, no hereda el método **getSource**.

Además de **getDocument**, la clase **DocumentEvent** proporciona otros tres métodos.

int getLength()
: Devuelve la longitud del cambio.

int getOffset()
Devuelve la posición dentro del documento del primer caracter modificado.
ElementChange getChange(Element)
Devuelve detalles sobre qué elementos del documento han cambiado y cómo. **ElementChange** es un interface definido dentro del interface **DocumentEvent**.
EventType getType()
Devuelve el tipo de cambio que ha ocurrido. **EventType** es una clase definida dentro del interface **DocumentEvent** que enumera los posibles cambios que pueden ocurrir en un document: insertar y eliminar texto y cambiar el estilo.

Oyente de Focus

Muchos componentes --incluso aquellos que operan primariamente con el ratón, como los botones -- pueden operar con el teclado. Parea que una pulsación afecte a un componente, este debe tener el foco del teclado.

Desde el punto de vista del usuario, el componente que tiene el foco del teclado generalmente es más priminente -- con un borde más ancho que el usual, por ejemplo -- y la ventana que contiene el componente también es más priminente que las otras ventanas de la pantalla. Estos aspectos visuales permiten al usuario conocer a qué componente le va a teclear. Al menos un componente del sistema de ventanas tiene el foco del teclado.

Los eventos Focus se generan cuando un componente gana o pierde el **foco del teclado**. El modo exacto en que los componentes ganan o pierden el foco depende del sistema de ventanas. Típicamente, el usuario selecciona el foco pulsando una ventana o componente, haciendo TAB entre componentes, o mediante otra forma de interactuar con el componente, Una vez que el foco está en una ventana (la ventana está **activada**) un programa puede usar el método **requestFocus** de **Component** para requerir que un componente específico tenga el foco.

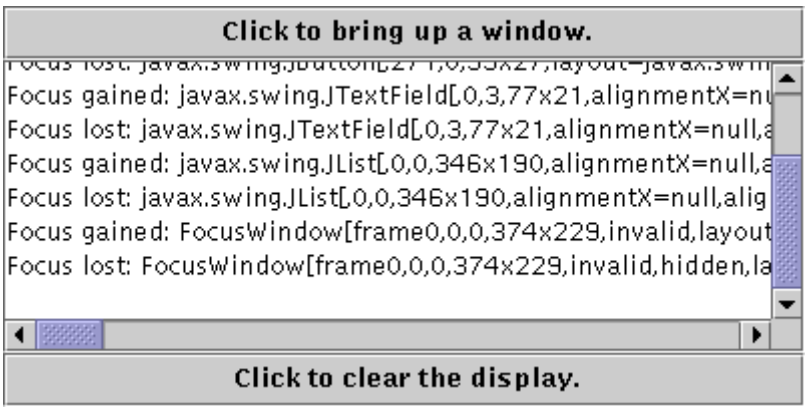
Métodos de Eventos Focus

El interface **FocusListener** y su correspondiente clase adaptadora, **FocusAdapter**, contienen dos métodos .

void focusGained(FocusEvent)
Se le llama después de que el componente escuchado obtenga el foco.
void focusLost(FocusEvent)
Se le llama después de que el componente escuchado pierda el foco.

Ejemplos de Manejo de Eventos Focus

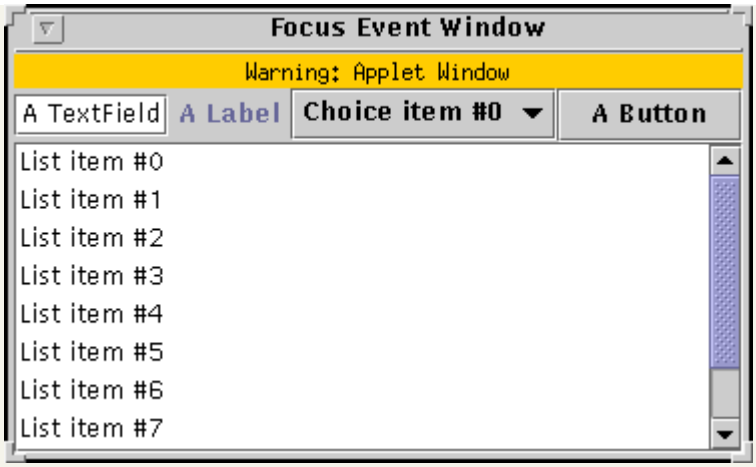
El siguiente applet demuestra los eventos focus. Pulsando sobre el botón superior del applet, se mostrará una ventana que contiene una veriedad de componentes. Un oyente de focus escucha todos los eventos de este tipo incluidos los de la propia ventana (que es un ejemplar de una subclase de **JFrame** llamada **FocusWindow**).



Esta es una imagen del GUI del Applet. Para ejecutarlo, pulsa sobre ella. El applet aparecerá en una nueva ventana de tu navegador.

Prueba esto:

1. **Despliega la ventana pulsando el botón superior del applet.**



si es necesario, pulsa sobre la ventana "Focus Event Window" para su contenido obtenga el foco del teclado. Veras que aparece un mensaje "Focus gained" en el área del applet. La forma en que la ventana obtiene o pierde el foco depende del sistema. Podemos detectar cuando una ventana gana o pierde el foco implementando un oyente de window y escuchando los eventos window activation o deactivation.

2. **Pulsa el botón que hay a la derecha de la ventana, y luego pulsa otro componente, como un campo de texto.**

   Observa que cuando el foco cambia de un componente a otro, el primer componente genera un evento focus-lost antes de que el segundo componente genere un evento focus-gained.

3. **Intenta cambiar el foco pulsao Tab o Shift-Tab.**

   La mayoría de los sistemas permiten usan la tecla Tab para circular a través de los componentes que pueden obtener el foco.

4. **Minimiza la ventana "Focus Event Window".**

   Deberías ver un mensaje "Focus lost" desde el último componente que lo tenía.

Puedes encontrar el código del applet en **FocusEventDemo.java**. Aquí está el código de manejo de eventos.

```
public class FocusEventDemo ... implements FocusListener ... {
                ./where initialization occurs
                window = new FocusWindow(this);
    ...
    public void focusGained(FocusEvent e) {
                displayMessage("Focus gained", e);
    }

    public void focusLost(FocusEvent e) {
                displayMessage("Focus lost", e);
    }

    void displayMessage(String prefix, FocusEvent e) {
                display.append(prefix
                                + ": "
                                + e.getComponent()
                                + newline);
    }
    ...
}

class FocusWindow extends JFrame {
    ...
    public FocusWindow(FocusListener listener) {
                super("Focus Demo Window");
                this.addFocusListener(listener);
                ...
                JLabel label = new JLabel("A Label");
                label.addFocusListener(listener);
                ...
                JComboBox choice = new JComboBox(/* list of items */);
                ...
                choice.addFocusListener(listener);
                ...
                JButton button = new JButton("A Button");
                button.addFocusListener(listener);
                ...
                JList list = new JList(/* list of items */);
                ...
                list.addFocusListener(listener);
    }
}
```

La clase FocusEvent

Cada método de evento focus tiene un sólo parámetro: un objeto **FocusEvent**, La clase **FocusEvent** define el siguiente método.

boolean isTemporary()
Devuelve true si la pérdida del foco es temporal. Necesitaremos utilizar este método si estamos implementando un componente que pueda indicar que obtendrá el foco cuando la ventana lo vuelva a obtener.

El método **getComponent**, que **FocusEvent** hereda de **ComponentEvent**, devuelve el componente que genero el evento de focus.

Oyente de InternalFrame

Los eventos Internal frame son a los **JInternalFrame** lo que los eventos window son a los **JFrame**. Al igual que los eventos window, los eventos internal frame notifican a sus oyentes que la "window" ha sido mostrada por primera vez, ha sido eliminada, iconificada, maximizada, activada o desactivada. Antes de usar este tipo de eventos debemos familiarizarnos con los Oyentes de Window.

Métodos de Evento Internal Frame

El interface **InternalFrameListener** y su correspondiente clase adaptador, **InternalFrameAdapter**, contienen estos métodos.

void internalFrameOpened(InternalFrameEvent)
Se le llama después de el internal frame escuchado se muestre por primera vez.
void internalFrameClosing(InternalFrameEvent)
Se le llama en respuesta a una petición del usuario de que el internal frame esuchado sea cerrado. Por defecto , **JInternalFrame** oculta la ventana cuando el usuario lo cierra. Podemos usar el método **setDefaultCloseOperation** de **JInternalFrame** para especificar otra opción, que puede ser **DISPOSE_ON_CLOSE** o **DO_NOTHING_ON_CLOSE** (ambas definidas en **WindowConstants**, un interface que implementa **JInternalFrame**). O implementando un método **internalFrameClosing** el oyente del internal frame, podemos añadir un comportamiento personalizado (como mostrar un díalogo o salvar datos) para cerrar un internal frame.
void internalFrameClosed(InternalFrameEvent)
Se le llama después de que haya desaparecido el internal frame escuchado.
void internalFrameIconified(InternalFrameEvent)

void internalFrameDeiconified(InternalFrameEvent)

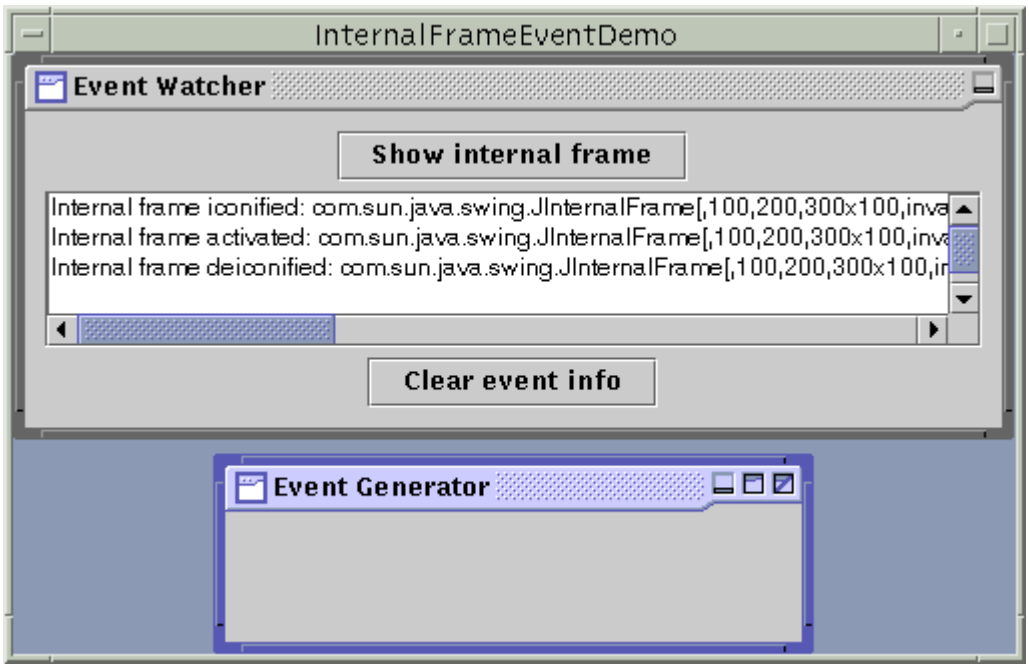
Se les llama después de que el internal frame escuchado sea iconificado o maximizado, respectivamente.
void internalFrameActivated(InternalFrameEvent)

void internalFrameDeactivated(InternalFrameEvent)

Se les llama después de que el internal frame escuchado sea activado o desactivado, respectivamente.

Ejemplos de Manejo de Eventos InternalFrame

La aplicación mostada en la siguiente figura demuestra el uso de eventos internal frame. La aplicación escucha eventos internal frame desde el frame Event Generator, mostrando un mensaje que describe cada evento.



Prueba esto:

1. Compila y ejecuta **InternalFrameEventDemo**. El fichero fuente es **InternalFrameEventDemo.java**.
2. Despliega la ventana Evento Generator pulsando el botón de la parte superior del applet.

   Deberías ver un mensaje "Internal frame opened" en el área de display.

3. Prueba varias cosas para ver que sucede. Por ejemplo, pulsa el Event Generator para activarlo. Pulsa el Event Watcher para desactivar el Event Generator. Pulsa la decoración de la ventana Event Generator para iconificara, maximizarla y cerrarla.

   Puedes ver la página Cómo escribir un Oyente de Window para más información sobre los tipos de eventos que verás.

Aquí está el código que maneja los eventos de internal frame.

```
public class InternalFrameEventDemo ...
                                implements InternalFrameListener ... {
    ...
    protected void createListenedToWindow() {
              listenedToWindow = new JInternalFrame("Event Generator",
                                                                        true,  //resizable
                                                                        true,  //closable
                                                                        true,  //maximizable
                                                                        true); //iconifiable
              listenedToWindow.setDefaultCloseOperation(
                                                        WindowConstants.DISPOSE_ON_CLOSE);
              ...
    }

    public void internalFrameClosing(InternalFrameEvent e) {
              displayMessage("Internal frame closing", e);
    }

    public void internalFrameClosed(InternalFrameEvent e) {
              displayMessage("Internal frame closed", e);
              listenedToWindow = null;
    }

    public void internalFrameOpened(InternalFrameEvent e) {
              displayMessage("Internal frame opened", e);
    }

    public void internalFrameIconified(InternalFrameEvent e) {
              displayMessage("Internal frame iconified", e);
    }

    public void internalFrameDeiconified(InternalFrameEvent e) {
              displayMessage("Internal frame deiconified", e);
    }

    public void internalFrameActivated(InternalFrameEvent e) {
              displayMessage("Internal frame activated", e);
    }

    public void internalFrameDeactivated(InternalFrameEvent e) {
              displayMessage("Internal frame deactivated", e);
    }

    void displayMessage(String prefix, InternalFrameEvent e) {
              String s = prefix + ": " + e.getSource();
              display.append(s + newline);
    }

    public void actionPerformed(ActionEvent e) {
              if (e.getActionCommand().equals(SHOW)) {
                  ...
                  if (listenedToWindow == null) {
                      createListenedToWindow();
                      listenedToWindow.addInternalFrameListener(this);
                      ...
                  }
              }
              ...
    }
}
```

La clase InternalFrameEvent

Cada método de evento internal frame tiene un sólo parámetro: un objeto **InternalFrameEvent**. La clase **InternalFrameEvent** generalmente no define métodos útiles. Para obtener el internal frame que generó el evento, se usa el método **getSource**, que **InternalFrameEvent** hereda de **EventObject**.

Oyente de Item

Los eventos item son generados por componentes que implementan el interface **ItemSelectable**. Estos on componentes que mantienen el estado -- generalmente on/off -- de uno o más ítems. Los componentes Swing que pueden generar estos eventos son checkboxes, checkbox menu items, y comboboxes.

Métodos de Evento Item

El interface **ItemListener** sólo tiene un método y por lo tanto no tiene clase adaptador. Aquí está el método.

void itemStateChanged(ItemEvent)
Se le llama después de que cambie el estado del componente escuchado.

Ejemplos de Manejo de Eventos Item

Aquí tenemos algún código de manejo de eventos item tomado de **ComponentEventDemo.java**.

```
public void itemStateChanged(ItemEvent e) {
    if (e.getStateChange() == ItemEvent.SELECTED) {
        label.setVisible(true);
    } else {
        label.setVisible(false);
    }
}
```

Podrás encontrar más ejemplos de este tipo de oyentes en los siguientes ficheros fuente.

- **CardWindow.java**
- **Converter.java**
- **CheckBoxDemo.java**
- **ComponentEventDemo.java**
- **PopupMenuDemo.java**
- **ScrollDemo.java**

La clase ItemEvent

El método **itemStateChanged** tiene un sólo parámetro, un objeto **ItemEvent**. La clase **ItemEvent** define los siguientes métodos.

Object getItem()
Devuelve el objeto component específico asociado con el ítem cuyo estado ha cambiado. Normalmente es un **String** que contiene el texto del ítem seleccionado. Para evento item generado por un **JComboBox**, es un **Integer** que especifica el índice del ítem seleccionado.
ItemSelectable getItemSelectable()
Devuelve el componente que genero el evento item. Podemos usarlo en lugar del método **getSource**.
int getStateChange()
Devuelve el nuevo estado del ítem. La clase **ItemEvent** define dos estados: **SELECTED** y **DESELECTED**.

Oyente de Key

Los eventos Key informan de cuándo el usuario ha tecleado algo en el teclado. Específiamente, los evento key son generados por el componente que tiene el foco del teclado cuando el usuario pulsa o libera las teclas del teclado. (Para más información sobre el foco del teclado, puedes ver la página Cómo escribir un Oyente de Focus.)

Se pueden notificar dos tipos básicos de eventos key: la pulsación de un caracter Unicode, y la pulsación o liberación de una tecla del teclado. El primer tipo de llama evento key-typed, y el segundo son eventos key-pressed y key-released.

En general, sólo deberíamos manejar los eventos key-typed a menos que necesitemos saber cuando el usuario ha pulsado teclas que no corresponden con caracteres. Por ejmplo, si queremos saber cuando el usario teclea algún caracter Unicode -- siempre como resultado de una pulsación de tecla como 'a' de la pulsación de una secuencia de teclas -- dberíamos manejar eventos key-typed. Por otro lado, si queremos saber cuando el usuario ha pulsado la tecla F1, necesitaremos manejer eventos key-pressed.

Nota: Para generar eventos de teclado, un componente **debe** tener el foco del teclado.

Para hacer que un componente obtenga el foco del teclado debemos seguir estos pasos.

1. Asegurarnos de que el componente puede obtener el foco del teclado. Por ejemplo, en algunos sistemas las etiquetas no pueden obtenerlo.
2. Asegurarnos de que el componente pide el foco en el momento apropiado. Para componentes personalizados, probablemente necesitaremos implementar un **MouseListener** que llame al método **requestFocus** cuandos e pulsa el ratón.
3. Si estamos escribiendo un componente personalizado, implementaremos el método **isFocusTraversable** del componente, para que devuelva true cuando el componente está activado. esto permie al usuario usar Tab para ir a nuestro componente.

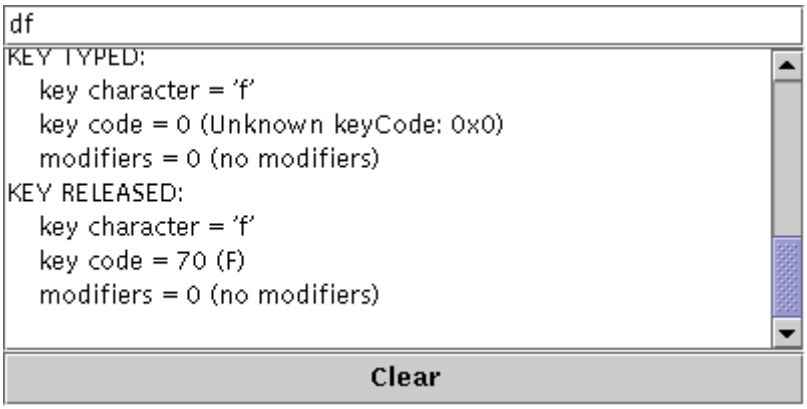
Métodos de Evento Key

El interface **KeyListener** y su correspondiente clase adaptador, **KeyAdapter**, contienen tres métodos.

void keyTyped(KeyEvent)
Se le llama después de que el usuario teclee un caracter Unicode dentro del componente escuchado.
void keyPressed(KeyEvent)
Se le llama después de que el usario pulse una tecla mientras el componente escuchado tiene el foco.
void keyReleased(KeyEvent)
Se le llama después de que el usario libere una tecla mientras el componente escuchado tiene el foco.

Ejemplos de manejo de Eventos Key

El siguiente applet demuestra los eventos key. Consiste en un campo de texto en el que podemos teclear información, seguido de un área de texto que muestra un mensaje cada vez que el campo de texto dispara un evento key. Un botón en la parte inferior del applet nos permite borrar ranto el campo como el área de texto.



Esta es una imagen del GUI del applet. Para ejecutarlo, pulsa sobre ella. El applet aparecerá en una nueva ventana de tu navegador..

Prueba esto:

1. **Pulsa sobre el campo de texto del applet para que obtenga el foco.**
2. **Teclea una 'a' minúscula pulsando y liberando la tecla A del teclado.**

   El campo de texto dispara tres eventos: un key-pressed, un key-typed, y un key-released. Observa que el evento key-typed no tiene información sobre el código de la tecla; los eventos key-typed tampoco tienen información sobre los modificadores.

3. **Pulsa el botón Clear.**

   Deberias hacer esto después de cada uno de los pasos siguientes.

4. **Pulsa y libera la tecla Shift.**

   El campo de texto dispara dos eventos: un key pressed y un key released. El campo de texto no genera ningún evento key-typed porque Shift, por sí misma, no corresponde con ningún carácter.

5. **Teclea una 'A' mayúscula pulsando las teclas Shift y A.**

   Verás los siguientes eventos, aunque quizás no en este orden: key pressed (Shift), key pressed (A), key typed ('A'), key released (A), key released (Shift).

6. **Teclea una 'A' mayúsculas pulsando y liberando la tecla Caps Lock, y luego pulsando la tecla A.**

   Deberías ver los siguientes eventos: key pressed (Caps Lock), key pressed (A), key typed ('A'), key released (A). Observa que la tecla Caps Lock no genera unevento key-released hasta que la pulses y la liberes de neuvo. Lo mismo sucede para otras teclas de estado como Scroll Lock y Num Lock.

7. **Pulsa y mantén la tecla A..**

   ¿Se repite automáticamente? Si es así, verás los mismos resultados que verías si pulsaras y liberaras la tecla A repetidamente.

Puedes encontrar el código del applet en KeyEventDemo.java. Aquí puedes ver el código de manejo de eventos.

```
public class KeyEventDemo ...  implements KeyListener ... {
    ...//where initialization occurs:
               typingArea = new JTextField(20);
               typingArea.addKeyListener(this);
    ...
```

```
    /** Handle the key typed event from the text field. */
    public void keyTyped(KeyEvent e) {
                displayInfo(e, "KEY TYPED: ");
    }

    /** Handle the key pressed event from the text field. */
    public void keyPressed(KeyEvent e) {
                displayInfo(e, "KEY PRESSED: ");
    }

    /** Handle the key released event from the text field. */
    public void keyReleased(KeyEvent e) {
                displayInfo(e, "KEY RELEASED: ");
    }
    ...
    protected void displayInfo(KeyEvent e, String s){
                ...
                char c = e.getKeyChar();
                int keyCode = e.getKeyCode();
                int modifiers = e.getModifiers();
                ...
                tmpString = KeyEvent.getKeyModifiersText(modifiers);

                ...//display information about the KeyEvent...
    }
}
```

La clase KeyEvent

Cada método de evento key tiene un sólo parámetro: un objeto **KeyEvent**. La clase **KeyEvent** define los siguientes métodos.

int getKeyChar()
void setKeyChar(char)
: Obtiene o selecciona el caracter Unicode asociado con este evento.

int getKeyCode()
void setKeyCode(int)
: Obtiene o selecciona el código de tecla asociado con este evento. El código de tecla identifica una tecla particular del teclado que el usuario pulsa o libera. La clase **KeyEvent** define muchas constantes de código de teclas para las más utilizadas. Por ejemplo, **VK_A** especifica la tecla **A**, y **VK_ESCAPE** especifica la tecla ESCAPE.

void setModifiers(int)
: Selecciona el estado de las teclas modificadoras para este evento. Podemos obtener el estados de las teclas modificadores usando el método **getModifiers** de **InputEvent**.

String getKeyText()
String getKeyModifiersText()
: Devuelve una descripción del código de tecla y de la tecla modificadora, respectivamente.

La clase **KeyEvent** hereda muchos métodos de **InputEvent** y **ComponentEvent**. Los siguientes métodos están descritos en La clase MouseEvent.

- **Component getComponent()**
- **void consume()**
- **int getWhen()**
- **boolean isAltDown()**
- **boolean isControlDown()**
- **boolean isMetaDown()**
- **boolean isShiftDown()**
- **int getModifiers()**

Oyente de ListSelection

Los eventos List selection ocurren cuando una selección en una list o una table cambia o acaba de cambiar. Los eventos List selection son disparados por un objeto que implementa el interface **ListSelectionModel**.

Para detectar un evento list selection debemos registrar un oynete con el objeto selection model apropiado. La clase **JList** también ofrece la opción de registrar un oyente sobre la propia lista, mejor que directamente al selection model de la lista.

Métodos de Evento List Selection

El interface **ListSelectionListener** sólo tiene un método y por lo tanto ni tiene la correspondiente clase adaptador. Aquí está el método.

void valueChanged(ListSelectionEvent)
: Se le llama cuando cambia la selección del componente escuchado, también se la llama después de que la selección haya cambiado.

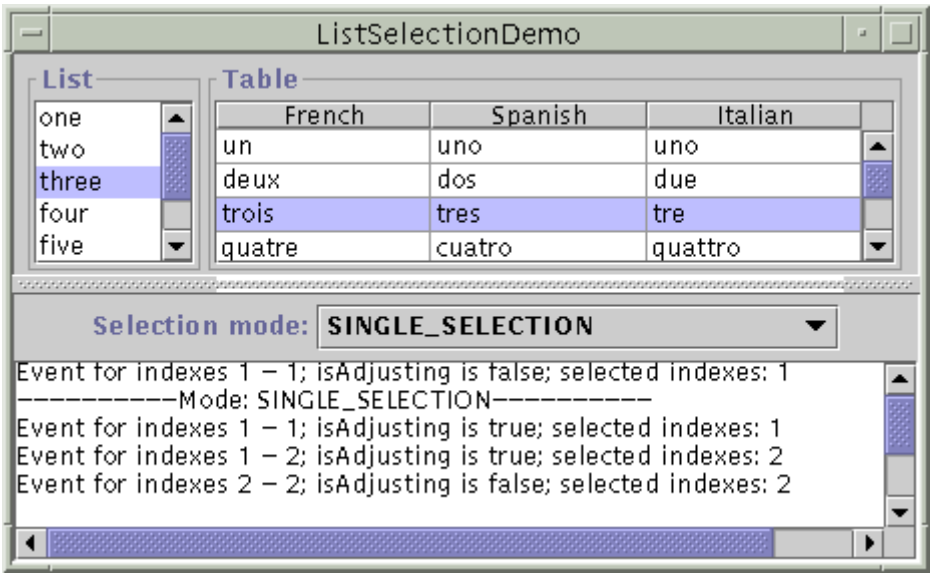
Ejemplos de Manejo de Eventos List Selection

La sección Cómo usar Lists proporciona un ejemplo de un oyente que escucha los eventos de una lista single-selection (no del selection model de la lista).

Esta sección proporciona un ejemplo que mustra cómo escuchar este tipo de eventos en un selection model. El selection model es compartido por un lista y una tabla. Dinámicamente podemos cambiar el modo de selección a uno de los tres modos soportados.

- single selection
- single interval selection
- multiple interval selection

Aquí podmeos ver una imágen del ejemplo ejecutándose.



Prueba esto:

1. Compila y ejecuta la aplicación. El código fuente está en **ListSelectionDemo.java**.
2. Selecciona y deselecciona items de la lista y de la tabla. Los comandos requeridos de ratón y de teclado para seleccionar ítems dependen del "aspecto y comportamiento". Para Metal, pulsa el botón izquierdo delr atón para empezar una selección, usa la tecla Shift para extender una selección contigüa, y usa la tecla Control para extender una selección discontigüa. Arrastrar el ratón mueve o extiende la selección dependiendeo del modo de selección.

Aquí está el código de **ListSelectionDemo.java** que configura el modelo de selección y le añade un oyente.

```
...//where the member variables are defined
JList list;
JTable table;
    ...//in the init method:
    listSelectionModel = list.getSelectionModel();
    listSelectionModel.addListSelectionListener(
                            new SharedListSelectionHandler());
    ...
    table.setSelectionModel(listSelectionModel);
```

Y aquí está el código para el oyente, que funciona para todos los modos de selección posibles.

```
class SharedListSelectionHandler implements ListSelectionListener {
    public void valueChanged(ListSelectionEvent e) {
        ListSelectionModel lsm = (ListSelectionModel)e.getSource();

        int firstIndex = e.getFirstIndex();
        int lastIndex = e.getLastIndex();
        boolean isAdjusting = e.getValueIsAdjusting();
        output.append("Event for indexes "
                      + firstIndex + " - " + lastIndex
                      + "; isAdjusting is " + isAdjusting
                      + "; selected indexes:");

        if (lsm.isSelectionEmpty()) {
            output.append(" <none>");
        } else {
            // Find out which indexes are selected.
            int minIndex = lsm.getMinSelectionIndex();
            int maxIndex = lsm.getMaxSelectionIndex();
            for (int i = minIndex; i <= maxIndex; i++) {
                if (lsm.isSelectedIndex(i)) {
                    output.append(" " + i);
                }
            }
        }
        output.append(newline);
    }
}
```

El método **valueChanged** muestra el primer y último índices reportados por el evento, el valor de la bandera **isAdjusting** del evento, y el indice actualmente seleccionado.

Observa que el primer y útlimo indices reportados por el eventos indican el rango inclusivo de ítems para los que la selección ha cambiado. Si el modo de selección es multiple interval selection algunos ítems dentro del rango podrían no haber cambiado. La bandera **isAdjusting** es **true** si el usuario todavía está manipulando la selección, y **false** si el usuario ha terminado de modificar la selección.

El objeto **ListSelectionEvent** pasado dentro de **valueChanged** indica sólo que la selección ha cambiado. El evento no contiene información sobre la selección actual. Por eso, este método le pide al selection model que se imagine la selección actual.

Nota: La salida de este programa depende de la verión Swing que estemos utilizando. Swing 1.0.x contiene varios bugs y la operación de listas y tablas era inconsistente. Las versiones posteriores de Swing corrigieron estos problemas.

La clase ListSelectionEvent

Cada método de evento list selection tiene un sólo parámetro: un objeto **ListSelectionEvent**. Este objeto le dice al oyente que la selección ha cambiado. Un evento list selecton puede indicar un cambio en la selección de múltiples ítems, discontigüos de la lista.

Para obtener la fuente de un **ListSelectionEvent**, se usa el método **getSource**, que **ListSelectionEvent** hereda de **EventObject**. Si registramos un oyente de list selection directamente sobre una lista, la fuente de cada evento será la propia lista. De otra forma, sería el selection model.

La clase **ListSelectionEvent** define los siguientes métodos.

int getFirstIndex()
: Devuelve el índice del primer ítem cuyo valor de selección ha cambiado. Observa que para selecciones de intervalo múltiple, el primer y último ítems es seguro que han cambiado, pero los ítems que hay entre medias podrían no haberlo hecho.

int getLastIndex()
: Devuelve el índice del último ítem cuyo valor de selección ha cambiado. Observa que para selecciones de intervalo múltiple, el primer y último ítems es seguro que han cambiado, pero los ítems que hay entre medias podrían no haberlo hecho.

int getValueIsAdjusting()
: Devuelve **true** si se está modificando todavía la selección. Muchos oyentes de list selectionsólo están intereados en el estado final de la selección y pueden ignorar eventos cuando este método devuelve **true**.

Oyente de Mouse

Los eventos de Mouse nos cuentan cuando el usuario usa el ratón (o un dispositivo de entrada similar) para interactuar con un componente. Los eventos Mouse ocurren cuando el cursor entra o sale del área de pantalla de un componente, o cuando el usuario pulsa o libera el botón del ratón. Como seguir la pista del movimiento del ratón significa mas sobrecarga del sistema que seguir la pista de los eventos de ratón, los eventos mouse-motion se han separado en otro tipo de oyente (puedes ver Cómo escribir un Oyente de Mouse Motion).

Métodos de Eventos Mouse

El interface **MouseListener** y su correspondiente clase adaptadora, **MouseAdapter**, contienen estos métodos.

void mouseClicked(MouseEvent)
: Llamado justo después de que el usario pulse sobre el componente escuchado.

void mouseEntered(MouseEvent)
: Llamado justo después de que el cursor entre en los límites del componente escuchado.

void mouseExited(MouseEvent)
: Llamado justo después de que el cursor salga de los límites del componente escuchado.

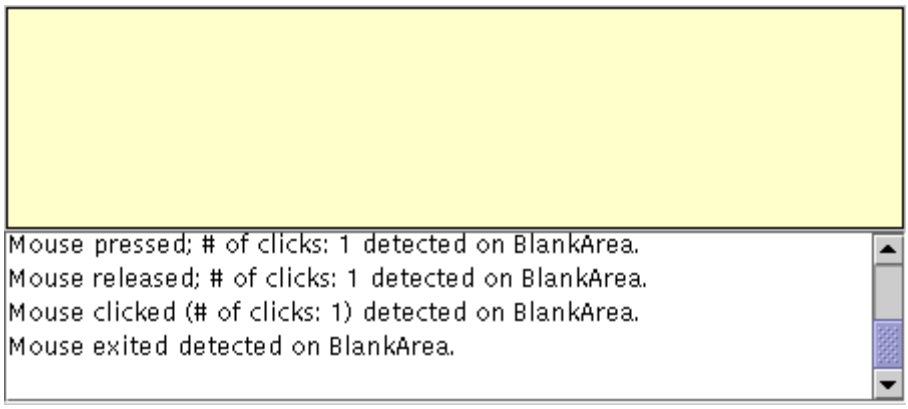
void mousePressed(MouseEvent)
: Llamado justo después de que el usuario pulse un botón del ratón mientras el cursor está sobre el componente escuchado.

void mouseReleased(MouseEvent)
: Llamdo justo después de que el usuario libere un botón del ratón después de una pulsación sobre el componente esuchado.

Una complicación afecta a los evntos mouse-entered, mouse-exited, y mouse-released. Cuando el usuario arrastra (pulsa y mantiene el botón del ratón y luego mueve el ratón), entonces el componente sobre el que estaba el cursor cuando empezó el arrastre es el que recibe todos los subsecuentes eventos de mouse y mouse-motion incluyendo la liberación del botón. Esto significa que ningún otro componente recibera un sólo evento del ratón -- ni siquiera un evento mouse-released -- mientras está ocurriendo el arrastre.

Ejemplos de Manejo de Eventos Mouse

El siguiente applet contiene un oyente de mouse. En la parte superior del applet hay un área vacía, (implementada por una clase llamda **BlankArea**). El oyente de mouse escucha los eventos del **BlankArea** y de su contenedor, que es un ejemplar de **MouseEventDemo**. Cada vez que ocurre un evento de mouse, se muestra un mensaje descriptivo sobre el área blanca. Moviendo el cursor sobre el área blanca y ocasionalmente pulsado algún botón del ratón podemos generar eventos mouse.



Esto es una imagen del GUI del Applet. Para ejecutarlo, pulsa sobre ella. El applet aparecerá en una nueva ventana de tu navegador..

Prueba esto:

1. **Mueve el cursor dentro del rectángulo amarillo de la parte superior del applet.**

   Verás uno o más eventos mouse-entered.

2. **Pulsa y mantén el botón del ratón..**

   Verás un evento mouse-pressed. Podrías ver algún evento extra como un mouse-exited o mouse-entered.

3. **Libera el botón del ratón.**

   Verás une vento mouse-released. Si no has movido el ratón, seguirá un evento mouse-clicked.

4. **Pulsa y mantén el botón del ratón, y arrástralo para el cursor termine fuera del área del applet. Libera el botón del ratón.**

   Verás un evento mouse-pressed, seguido de un evento mouse-exited, seguido por un evento mouse-released. **No** se ha notificado el movimiento del cursor. Para obtener eventos mouse-motion, necesitamos implementar un oyente de mouse-motion.

Puedes encontrar el código del applet en **MouseEventDemo.java** y **BlankArea.java**. Aquí tenemos el código de manejo de eventos del applet.

```
public class MouseEventDemo ... implements MouseListener {
              ...//where initialization occurs:
        //Register for mouse events on blankArea and applet (panel).
        blankArea.addMouseListener(this);
        addMouseListener(this);
    ...

    public void mousePressed(MouseEvent e) {
       saySomething("Mouse pressed; # of clicks: "
                    + e.getClickCount(), e);
    }

    public void mouseReleased(MouseEvent e) {
       saySomething("Mouse released; # of clicks: "
                    + e.getClickCount(), e);
    }

    public void mouseEntered(MouseEvent e) {
       saySomething("Mouse entered", e);
    }

    public void mouseExited(MouseEvent e) {
       saySomething("Mouse exited", e);
    }

    public void mouseClicked(MouseEvent e) {
       saySomething("Mouse clicked (# of clicks: "
                    + e.getClickCount() + ")", e);
    }

    void saySomething(String eventDescription, MouseEvent e) {
        textArea.append(eventDescription + " detected on "
                        + e.getComponent().getClass().getName()
                        + "." + newline);
    }
}
```

Podemos encontrar más ejemplos de oyentes de ratón en los siguientes ficheros fuente.

- **GlassPaneDemo.java**
- **TableSorter.java**
- **AnimatorApplicationTimer.java**

La Clase MouseEvent

Cada método de evento mouse tiene un sólo parámetero: un objeto **MouseEvent**. La clase **MouseEvent** define los siguientes métodos .

int getClickCount()
Devuelve el número de pulsaciones que el usuario ha realizado (incluyendo este evento).
int getX()
int getY()
Point getPoint()
Devuelve la posición (x,y) en la que ocurrió el evento, relativa al componente que generó el evento.
boolean isPopupTrigger()
Devuelve true si el evento mouse debería hacer que apareciera un menú popup. Como los disparadores de menús popup son dependientes de la plataforma, si nuestro programa los usa, deberíamos llamar a **isPopupTrigger** en todos los eventos mouse-pressed y mouse-released geneados por componentes sobre los que el popup pueda aparecer.

La clase **MouseEvent** hereda los siguientes métodos de **ComponentEvent**.

Component getComponent
Devuelve el componente que generó el evento. Podemos usar este método en vez de **getSource**.

La clase **MouseEvent** hereda otros muchos métodos útiles de **InputEvent**.

void consume()
Hace que el evento no sea procesado por el padre del componente. Se podría usar este método para descartar letras tecleadas en un campo de texto que sólo hacepta números.
int getWhen()
Devuelve el momento en que ocurrió el evento.
boolean isAltDown()

boolean isControlDown()

boolean isMetaDown()

boolean isShiftDown()

Devuelven el estado individual de las teclas modificadores en el momento en que se generó el evento.
int getModifiers()
Devuelve el estado de todas las teclas modificadoras y botónes del ratón, cuando se generó el evento. Podemos usar este método para determinar qué botón fue pulsado (o liberado) cuando el evento del ratón fue generado. La clase **InputEvent** define estas constantes para usarlas con el método **getModifiers**: **ALT_MASK**, **BUTTON1_MASK**, **BUTTON2__MASK**, **BUTTON3_MASK**, **CTRL_MASK**, **META_MASK**, y **SHIFT_MASK**. Por ejemplo, la siguiente expresión es verdadera si se pulsó el botón derecho.

```
(mouseEvent.getModifiers() & InputEvent.BUTTON3_MASK)
== InputEvent.BUTTON3_MASK
```

La clase **SwingUtilities** contiene métodos de conveniencia para determinar si se ha pulsado un botón particular del ratón.

static boolean isLeftMouseButton(MouseEvent)

static boolean isMiddleMouseButton(MouseEvent)

static boolean isLEFTMouseButton(MouseEvent)

Oyente de MouseMotion

Los eventos Mouse-motion nos dicen cuando el usuario usa el ratón (u otro dispositivo similar) para mover el cursor sobre la pantalla.

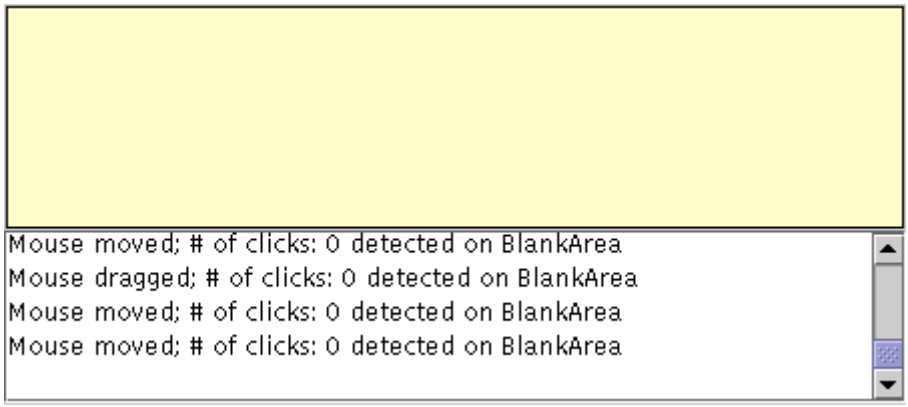
Métodos de Evento Mouse-Motion

El interface **MouseMotionListener** y su correspondiente clase adaptador, **MouseMotionAdapter**, contienen estos métodos.

void mouseDragged(MouseEvent)
Llamado en respuesta a un movimiento del ratón por parte del usuario mientras mantiene pulsa uno de los botones delr atón. Este evento es disparado por el componente que disparó el evento mouse-pressed más reciente, incluso si el cursor ya no está sobre ese componente.
void mouseMoved(MouseEvent)
Llamado en respuesta a un movimiento del ratón por parte del usuario sin ningún botón puslado. El evento es disparado por el eventos que se encuentra actualmente debajo del cursor.

Ejemplos de Manejo de Eventos Mouse-Motion

El siguiente applet contiene un oyente de mouse-motion. Es exactamente igual que el applet de la página Cómo escribir un Oyente de Mouse, excepto en que sustituye **MouseMotionListener** por **MouseListener**, e implementa los método **mouseDragged** y **mouseMoved** en vez de los método del oyente de mouse. Puedes encontrar el código del applet en **MouseMotionEventDemo.java** y en **BlankArea.java**.



Esto es una imagen del GUI del Applet. Para ejecutarlo, pulsa sobre ella. El applet aparecerá en una nueva ventana de tu navegador..

Prueba esto:

1. **Mueve el cursor dentro del rectángulo amarillo de la parte superior del applet.**

   Verás uno o más eventos mouse-moved.

2. **Pulsa y mantén un botón de ratón y muevelo hasta que el cursor se salga del rectángulo amarillo.**

   Verás eventos mouse-dragged.

El siguiente código contiene el manejo de eventos de la clase **RectangleDemo.java**. Esta clase maneja tres clases de eventos: **pulsacion de ratón**, **arrastre de ratón** y **liberación de ratón**. Estos eventos corresponden a los métodos **mousePressed** (de **MouseListener**), **mouseDragged** (de **MouseMotionListener**), **mouseReleased** (de **MouseListener**). Así, esta clase debe implementar tanto **MouseListener** como **MouseMotionListener**. Para evitar tener que manejar demasiados métodos vacios, esta clase no implementar **MouseListener** directamente. En su lugar, extiende MouseAdapter e implementa **MouseMotionListener**.

```
...//where initialization occurs:
    MyListener myListener = new MyListener();
    addMouseListener(myListener);
    addMouseMotionListener(myListener);
...
class MyListener extends MouseAdapter
                 implements MouseMotionListener {
    public void mousePressed(MouseEvent e) {
        int x = e.getX();
        int y = e.getY();
        currentRect = new Rectangle(x, y, 0, 0);
        repaint();
    }

    public void mouseDragged(MouseEvent e) {
        updateSize(e);
    }

    public void mouseMoved(MouseEvent e) {
        //Do nothing.
    }

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

    void updateSize(MouseEvent e) {
        int x = e.getX();
        int y = e.getY();
        currentRect.setSize(x - currentRect.x,
                            y - currentRect.y);
        repaint();
    }
}
```

Métodos de Eventos usados por oyentes de Mouse-Motion

Cada evento mouse-motion tiene un sólo parámetro -- y **no** se llama **MouseMotionEvent**! En su lugar cada evento mouse-motion tiene un método con un argumento **MouseEvent**. Puedes ver la página La clase MouseEvent para más información sobre cómo utilizar objetos **MouseEvent**.

Oyente de UndoableEdit

Los eventos undoable edit ocurren cuando una operación que puede ser reversible ocurre sobre un componente. Actualmente, sólo los componentes de texto pueden generar eventos undoable edit, y sólo indirectamente. El documento del componente genera el evento. Para los componentes de texto, las operaciones undoables incluyen insertar caracteres, borrarlos, y modificar el estilo del texto.

Métodos de eventos Undoable Edit

El interface **UndoableEditListener** tiene un sólo métodos, y por eso no tiene la correspondiente clase adaptadora. Aquí está el método.

void undoableEditHappened(UndoableEditEvent)
: Llamado cuando ocurre un evento undoable sobre el componente escuchado.

Ejemplos de manejo de eventos Undoable Edit

Los programas normalmente escuchan los eventos undoable edit para asistir en la implementación de los comandos "deshacer/repetir". Puedes referirte a Implementar Deshacer/Repetir para ver un ejemplo.

La clase UndoableEditEvent

El método **undoableEditHappened** tiene un sólo parámetros: un objeto **UndoableEditEvent**. Para obtener el documento que generó el evento se usa el método **getSource** que **UndoableEditEvent** hereda de **EventObject**.

La clase **UndoableEditEvent** define un método que devuelve un objeto que contiene información detalladas sobre la edición que ha ocurrido.

UndoableEdit getEdit()
: Devuelve un objeto **UndoableEdit** que representa la edición oucrrida y contiene información sobre los comandos para deshacer o repetir la edición.

Oyente de Window

Los eventos Windows son generados por una **ventana** justo después de que sea abierta, cerrada, iconificada, desiconificada, activada o desactivada. **Abrir** una ventana significa mostrarla por primera vez; **cerrarla** significa eliminarla de la ventana. **Iconificarla** significa sustiuirla por un pequeño icono en el escritorio; **desiconificarla** significa lo contrario. Una ventana es **activada** si uno de sus componentes tiene el foco del teclado; la **desactivación** ocurre cuando la ventana y todos sus componentes pierden el foco del teclado.

El uso más común de los oyentes de windows es cerrar ventanas. Si un programa no maneja los eventos de window-closing, entonces nada sucede cuando el usuario intenta cerrarla. Una aplicación que tenga una sóla ventana podría reaccionar a un evento window-closing saliendo del programa. Otros programas normalmente reaccionarán a los eventos window-closing eliminado la ventana o haciéndola invisible. Puedes ver Cómo crear Frames donde hay un ejemplo de un manejador de eventos window-closing.

Nota: Si utilizamos la clase **JFrame** de swing, podríamos no necesitar escribir un oyente de window. Puedes ver Cómo crear Frames para ma´s información sobre cómo la clase **JFrame** proporciona un manejo automático de los eventos window-closing.

Otros uso común de los oyentes de window es parar los threads y liberar recursos cuando una ventana es iconificada, y arracarlos otra vez cuando es desiconificada. De esta forma, podemos evitarel uso innecesario del procesador o de otros recursos. Por ejemplo, cuando una ventana que contiene una animación es iconificada, debería parar su thread de animación para liberar cualquier buffer. Cuando la ventana es desiconificada se puede arrancar el thread de nuevo y recrear los buffers.

Si queremos ser notificados cuando una ventana se hace visible o se oculta, deberíamos registrar un oyente de component sobre la ventana.

Métodos de evento Window

El interface **WindowListener** y su correspondiente clase adaptadora, **WindowAdapter**, conteniendo estos métodos.

void windowOpened(WindowEvent)
: Llamado justos después de que la vetana escuchadas sea mostrada por primera vez.

void windowClosing(WindowEvent)
: Llamada en respuesta a una petición de usuario de que la ventana escuchada sea cerrada. Para cerrar realmente la ventana, el oyente debería invocar a los métodos **dispose** o **setVisible(false)** de window.

void windowClosed(WindowEvent)

Llamado justo después de que la ventana escuchada sea cerrada.

void windowIconified(WindowEvent)

void windowDeiconified(WindowEvent)

Llamado justo después de que la ventana escuchada sea iconificada o desiconificada, respectivamente.

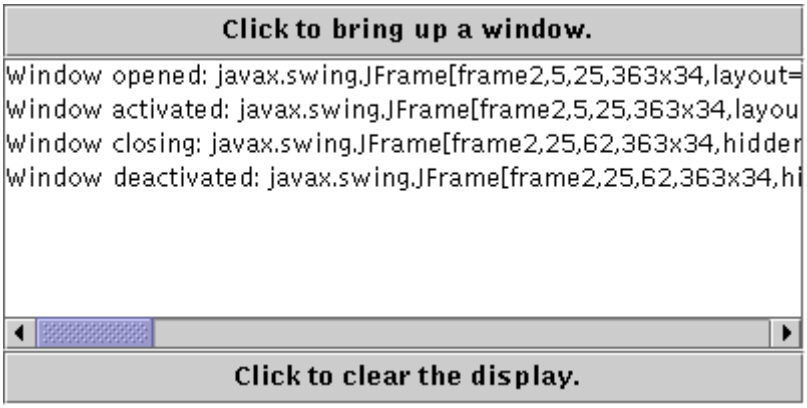
void windowActivated(WindowEvent)

void windowDeactivated(WindowEvent)

Llamado justo después de que la ventana escuchada sea activada o desactivada, respectivamente.

Ejemplos de manejo de eventos de Window

El siguiente applet desmuestra los eventos windows. Pulsando el botón del applet, podrás traer una pequeña ventana. La clase controladora escucha los eventos window de la ventana, mostrando un mensaje siempre que detecte uno. Puedes encontrar el código del applet en **WindowEventDemo.java**.



Esta es una imagen del GUI del applet. Para ejecutar el applet, pulsa sobre ella. El applet aparecera en una nueva ventana del navegador..

Prueba esto:

1. **Trae la ventana Window Demo pulsando el botón de la parte superior del applet.**

   La primera vez que pulses este botón, verás un mensaje "Window opened" en el área de display del applet.

2. **Pulsa sobre la ventana si todavía no tiene el foco.**

   ¿Ves un mensaje "Window activated" en el área de display del applet?

3. **Iconiica la ventana, usando los controles de la propia ventana.**

   Verás un mensaje "Window iconified" en el área de display del applet.

4. **Desiconifica la ventana.**

   Verás un mensaje "Window deiconified" en el área de display del applet.

5. **Cierra la ventana, usando los controles de la ventana.**

   Verás "Window closing" en el área de display del applet. Como el manejador de eventos llama a **setVisible(false)** en vez de **dispose()**, no verás "Window closed".

Aquí tienes el código de manejo de eventos del applet.

```
public class WindowEventDemo ... implements WindowListener {
```

```
     ...//where initialization occurs:
        //Create but don't show window.
        window = new JFrame("Window Event Window");
        window.addWindowListener(this);
        window.getContentPane().add(
                    new JLabel("The applet listens to this window for window events."),
                    BorderLayout.CENTER);
        window.pack();
    }

    public void windowClosing(WindowEvent e) {
        window.setVisible(false);
        displayMessage("Window closing", e);
    }

    public void windowClosed(WindowEvent e) {
        displayMessage("Window closed", e);
    }

    public void windowOpened(WindowEvent e) {
        displayMessage("Window opened", e);
    }

    public void windowIconified(WindowEvent e) {
        displayMessage("Window iconified", e);
    }

    public void windowDeiconified(WindowEvent e) {
        displayMessage("Window deiconified", e);
    }

    public void windowActivated(WindowEvent e) {
        displayMessage("Window activated", e);
    }

    public void windowDeactivated(WindowEvent e) {
        displayMessage("Window deactivated", e);
    }

    void displayMessage(String prefix, WindowEvent e) {
        display.append(prefix
                       + ": "
                       + e.getWindow()
                       + newline);
    }
    ...
}
```

Aquí tienes algunos ficheros fuente que contienen oyentes de window.

- **ComponentEventDemo.java**
- **FlowWindow.java**
- **AnimatorApplicationTimer.java**

La clase WindowEvent

Cada método de evento Window tiene un sólo parámetros: un objeto **WindowEvent**.

Window getWindow()
Devuelve la ventana que generó el evento. Podemos usarlo en lugar del método **getSource**.

Usar Controladores de Distribución

Cada contenedor, tiene un controlador de disposición por defecto -- un objeto que implementa el interface LayoutManager. Si el controlador por defecto de un contenedor no satisface sus necesidades, puedes reemplazarlo fácilmente por cualquier otro. El AWT suministra varios controladores de disposición que van desde los más sencillos (FlowLayout y GridLayout) a los de propósito general (BorderLayout y CardLayout) hasta el ultra-flexible (GridBagLayout) y **BoxLayout**).

Esta lección da algunas reglas generales para el uso de los controladores de disposición, le ofrece una introducción a los controladores de disposición proporcionados por el AWT, y cuenta cómo utilizar cada uno de ellos. En estas páginas encontrarás applets que ilustran los controladores de disposición. Cada applet trae una ventana que se puede redimensionar para ver los efectos del cambio de tamaño en la disposición de los componentes.

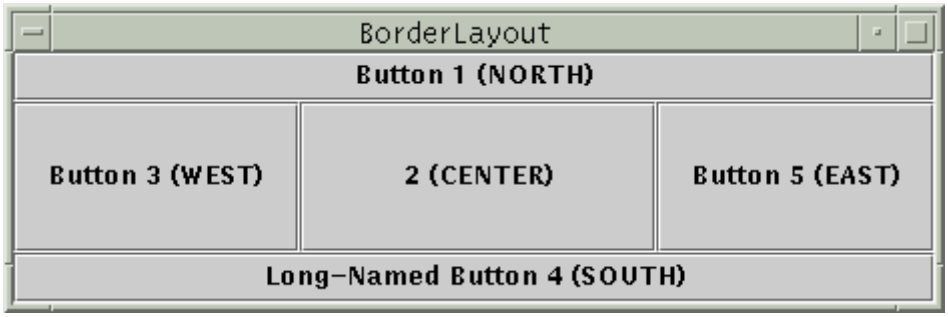
Reglas Generales para el uso de Controladores de Distribución

Esta sección responde algunas de las preguntas más frecuentes sobre los controladores de disposición.

- ¿Cómo puedo elegir un controlador de disposición?
- ¿Cómo puedo crear un controlador de disposición asociado con un contenedor, y decirle que empiece a trabajar?
- ¿Cómo sabe un controlador de disposición los componentes que debe manejar?

Cómo usar BorderLayout

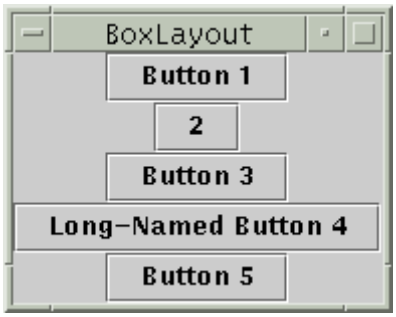
BorderLayout es el controlador de disposición por defecto para todas las ventanas, como Frames y Cuadros de Diálogo. Utiliza cinco áreas para contener los componentes: north, south, east, west, and center (norte, sur, este, oeste y centro). Todo el espacio extra se sitúa en el área central. Aquí tienes un applet que sitúa un botón en cada área.



Esta es una imagen del GUI del Applet. Para ejecutarlo pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

Cómo usar BoxLayout

La clase **BoxLayout** pone los componentes en una sóla fila columna. Respeta las peticiones de máximo tamaño dle componente, y también permite alinearlos. Aqui tienes un applet que pone un conjunto de botones en un columna centrada.



Esta es una imagen del GUI del Applet. Para ejecutarlo pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

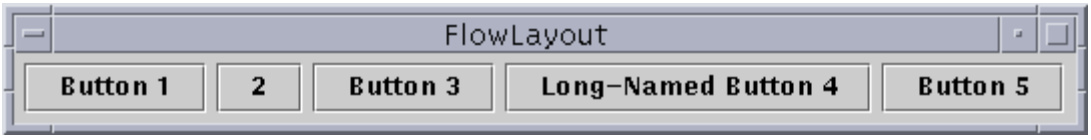
Cómo usar CardLayout

La clase **CardLayout** permite implementar un área que contiene diferentes componentes en diferentes ocasiones. Tabbed panes son componentes Swing intermediarios que propocionan una funcionalidad similar, pero con un GUI predefinido. Un **CardLayout** normalmente está controlador por un combo box, el estado del combo box determina que panel (grupo de componentes) muestra el **CardLayout**. Aquí hay un applet que usa un combo box y **CardLayout** de esta forma.

Esta es una imagen del GUI del Applet. Para ejecutarlo pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

Cómo usar FlowLayout

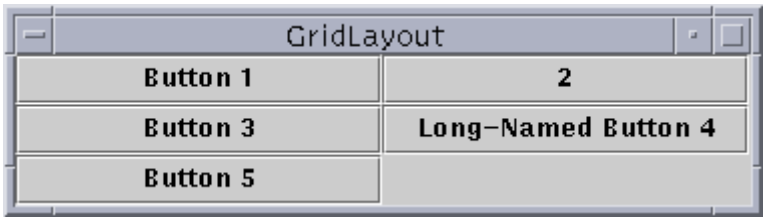
FlowLayout es el controlador por defecto para todos los Paneles. Simplemente coloca los componentes de izquierda a derecha, empezando una nueva línea si es necesario. Los dos paneles en el applet anterior utilizan FlowLayout. Aquí tienes otro ejemplo de applet que utiliza un FlowLayout.



Esta es una imagen del GUI del Applet. Para ejecutarlo pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

Cómo usar GridLayout

GridLayout simplemente genera un razimo de Componentes que tienen el mismo tamaño, mostrándolos en una sucesión de filas y columnas. Aquí tienes un applet que utiliza un GridLayout para controlar cinco botones.



Esta es una imagen del GUI del Applet. Para ejecutarlo pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

Cómo usar GridBagLayout

GridBagLayout es el más sofisticado y flexible controlador de disposición proporcionado por el AWT. Alínea los componentes situándolos en una parrilla de celdas, permitiendo que algunos componentes ocupen más de una celda. Las filas de la parrilla no tienen porque ser de la misma altura; de la misma forma las columnas pueden tener diferentes anchuras. Aquí tiene un applet que utiliza un GridBagLayout para manejar diez botones en un panel.

Esta es una imagen del GUI del Applet. Para ejecutarlo pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

Reglas de Uso de Controladores de Distribución

A menos que se le diga a un contenedor que no utilice un controlador de disposición, el está asociado con su propio ejemplar de un controlador de disposición. Este controlador es consultado automáticamente cada vez que el contenedor necesita cambiar su apariencia. La mayoría de los controladores de disposición no necesitan que los programan llamen directamente a sus métodos.

Cómo elegir un Controlador de Distribución

Los controladores de disposición proporcionados por la plataforma Java tienen diferentes potencias y puntos débiles. Esta sección descubre algunos de los escenarios de distribución más comunes y cómo podrían trabajar los controladores de distribución del AWT en cada escenario. Si ninguno de los controladores del AWT se adapta a nuestra situación, deberiamos; sentirnos libres para utilizar controladores de distribución distribuidos por la red, o escribir el nuestro propio.

Escenario: Necesitamos mostrar un componente en todo el espacio que se pueda conseguir.
Consideramos la utilización de BorderLayout o GridBagLayout. Si utilizamos BorderLayout, necesitarremos poner el componente que necesite más espacio en el centro. Con GridBagLayout, necesitaremos seleccionar las restricciones del componente para que **fill=GridBagConstraints.BOTH**. Otra posibilidad es usar **BoxLayout**, haciendo que el componente con más tamaño especifique unos tamaño preferido y máximo muy grandes

Escenario: Necesitamos mostrar unos pocos componentes en una fila compacta a su tamaño natural.
Consideramos usar un **JPanel** para contener los componentes y usar el controlador por defecto de **JPanel** que es **FlowLayout** o usar **BoxLayout**.
Escenario: Necesitamos mostrar unos pocos componentes del mismo tamaño en filas y columnas.
GridLayout es perfecto para esto.
Escenario: Necesitamos mostrar unos pocos componentes en filas y columnas, posiblemente variando la cantidad de espacio entre ellos, con alineamientos personalizados, o tamaños de componentes personalizados.
BoxLayout es perfecto para esto
Escenario: Tenemos una distribución compleja con muchos componentes
Debemos considerar la utilización de **GridBagLayout** o agrupar los componetes en uno o más **JPanel** para simplificar la distribución. Cada **JPanel** dabería usar un diferente controlador de distribución.

Cómo crear un Controlador de Distribución y Asociarlo con un Contenedor

Como se mencionó en Control de Distribución, por convención, cada contenedor tiene su controlador de distribución. Todos los objetos **JPanel** se inicializan para usar un **FlowLayout**. El panel de contenidos para todos los objetos **JApplet**, **JDialog**, y **JFrame** está inicializado para usar un **BorderLayout**. Otros contendores Swing tienden a tener controladores de distribución muy especializados o, como en el caso de **JLayeredPane**, no tienen controlador de distribución.

Si queremos utilizar el controlador de distribución por defecto de un Contenedor, no tenemos que hacer nada. El constructor de cada Contenedor crea un ejemplar del controlador de distribución e inicializa el Contenedor para que lo utilice.

Para utilizar un controlador de disposición que no sea por defecto, necesitaremos crear un ejemplar de la clase del controlador deseado y luego decirle al contenedor que lo utilice. Normalmente sólo haremos esto para **JPanel** y para paneles de contenido. La siguiente sentencia crea un controlador **BorderLayout** y lo inicializa como controlador de distribución para un panel

```
aJPanel.setLayout(new BorderLayout());
```

Aquí podemos ver un ejemplo de hacer que un objeto **FlowLayout** sea el controlador de distribución para el panel de contenido de un applet.

```
//In a JApplet subclass.
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
```

Reglas del pulgar para usar Controladores de Distribución

Los métodos de **Container** que dan como resultado llamadas a su controlador de distribución son **add**, **remove**, **removeAll**, **doLayout**, **invalidate**, **getAlignmentX**, **getAlignmentY**, **getPreferredSize**, **getMinimumSize**, y **getMaximumSize**. Los métoso **add**, **remove**, y **removeAll** añaden y eliminan componentes de un contenedor; se les puede llamar en cualquier momento. El método **doLayout**, que es llamado como resultado de cualquier petición de dibujado de un contenedor o de una llamada a **validate** sobre el contenedor, requiere que el contenedor se situe y redimensione a sí mosmo y a los componentes que contiene; no se puede llamar al método **doLayout** directamente.

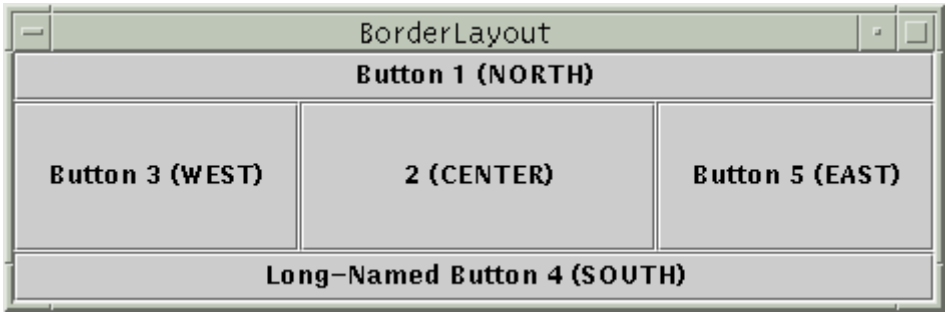
Si cambiamos el tamaño d eun componente, aunque sea indirectamente como cambiando su fuente, el componente debería redimensionarse automáticamente y redibujarse a sí mismo. Si esto no sucediera por alguna razón, deberíamos invocar al método **revalidate** del componente. Esta peticiónes pasada a través del árbol de contenientes hasta que encuentre un contenedor, como un scroll-pane o un contenedor de alto nivel, que no debería verse afecta por el redimensionado del componene. (Esto está determinado por la llamda al método **isValidateRoot** del contenedor.) El contenedor es entonces redistribuido, lo que tiene el efecto de ajustar los componentes revalidados y todos los demás componentes afectados. Después de llamar a **revalidate** sobre un componene se debería llamar a **repaint**.

Los métodos **getAlignmentX** y **getAlignmentY** son llamados por los controladores de distribución para intenrar alinear grupos de componentes. **BoxLayout** es el único controlador de distribución que llama a estos métodos.

Los métodos **getPreferredSize**, **getMinimumSize**, y **getMaximumSize** retornan los tamaños ideal, mínimo y máximo, respectivamente. Los valores devueltos son sólo indicativos, un controlador de distribución puede ignorarlos.

¿Cómo usar BorderLayout?

Aquí hay un applet que muestra un **BorderLayout** en acción.



Esta es una imagen del GUI del applet, para ejecutarlo, pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador."

Como muestra el applet anterior, un **BorderLayout** tiene cinco áreas: north, south, east, west, y center.

Si agrandamos la ventana, el área central obtiene tanto espacio disponible como le sea posible. Las otras áreas se extienden sólo lo necesario para rellenar todo el espacio disponible.

El siguiente código crea el **BorderLayout** y los componentes que maneja. Aquí está el programa completo. El programa funciona desde dentro d eun applet, con la ayuda de **AppletButton**, o como una aplicación.

```
Container contentPane = getContentPane();
//Use the content pane's default BorderLayout.
//contentPane.setLayout(new BorderLayout()); //unnecessary

contentPane.add(new JButton("Button 1 (NORTH)"),
                BorderLayout.NORTH);
contentPane.add(new JButton("2 (CENTER)"),
                BorderLayout.CENTER);
contentPane.add(new JButton("Button 3 (WEST)"),
                BorderLayout.WEST);
contentPane.add(new JButton("Long-Named Button 4 (SOUTH)"),
                BorderLayout.SOUTH);
contentPane.add(new JButton("Button 5 (EAST)"),
                BorderLayout.EAST);
```

Importante: Cuando se añade un componente a un contenedor que usa **BorderLayout**, se especifica la localización específica del componente como uno de los argumentos del método **add**. No esperes que un componente sea añadido al centro, por defecto. Si encontramos que un componente ha desaparecido de un contenedor controlador por un **BorderLayout**, debemos asegurarnos de que hemos especificado la localización del componente y de que no hemos puesto otro componente en la misma localización.

Todos nuestros ejemplos que usan **BorderLayout** especifican el componente como el primer argumento del método **add**. Por ejemplo.

```
add(component, BorderLayout.CENTER)  //preferimos esta forma
```

Sin embargo, podríamos ver el código de otros programas que especifican el componente en segundo lugar. Por ejemplo, esto sería una alternativa al código anterior.

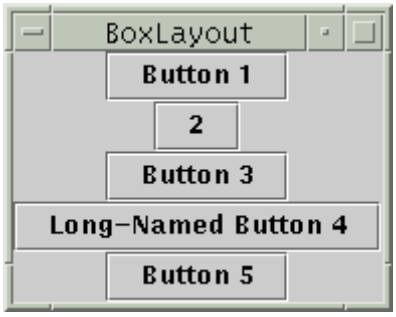
```
add(BorderLayout.CENTER, component)  //valido pero pasado de moda
    o
add("Center", component)             //valido pero propenso a errores
```

Por defecto, un **BorderLayout** no pone espacios entre los componentes que maneja. En el applet anterior, cualquier esparo aparente es el resultado del espacio extra que reserva **JButton** alrededor de su área. Podemos especificar los bordes (en pixels) usando el siguiente constructor.

```
public BorderLayout(int horizontalGap, int verticalGap)
```

¿Cómo usar BoxLayout?

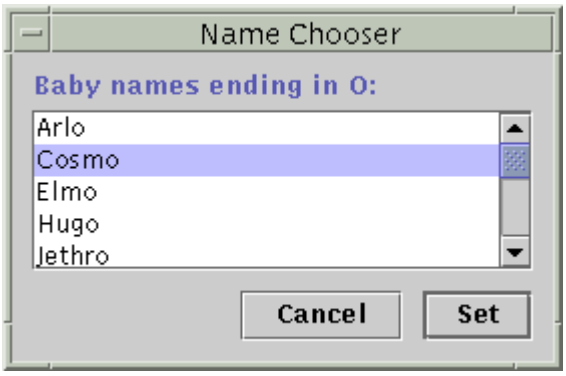
Los paquetes Swing incluyen un controlador de distribución de propósito geneal llamado **BoxLayout**. **BoxLayout** o bien situa los componenes uno encima d eotro (con el primer componente en la parte superior) o los situa en una delgada fila de izquierda a derecha - a nuestra eleción. Podríamos pensar que es una versión más potente de **FlowLayout**. Aquí tenemos una versión de un applet que usa un **BoxLayout** para mostrar una columna de componetes centrada.



Esta es una imagen del GUI del applet, para ejecutarlo, pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

Creando uno o más contenedores de peso ligero que usen **BoxLayout**, podemos conseguir distribuciones más complejas para las que normalmente se usaría **GridBagLayout**. **BoxLayout** también es útil en algunas situaciones donde podríamos considerar la utilización de **GridLayout** o **BorderLayout**. Hay una gran diferencia entre **BoxLayout** y los controladores de distribución existentes en el AWT, y es que **BoxLayout** respeta el tamaño máximo y el alineamiento X/Y de cada componente.

La siguiente figura muestra un GUI que usa dos ejemplares de **BoxLayout**. En la parte superior del GUI un boxlayout de arriba-a-abajo situa una etiqueta sobre un scroll pane. En la parte inferior del GUI, un boclayout de izquierda-a-derecha situa dos botones uno junto al otro. Un **BorderLayout** combina las dos partes del GUI y se asegura que cualquier exceso de espacio será entregado al scroll pane.



El siguiente código, tomado de **ListDialog.java**, disribuye el GUI. Este código está en el constructor del diálogo, que está implementadocomo una subclase de **JDialog**. Las líneas en negrita inicializan los boxlayout y les añaden los componentes.

```
JScrollPane listScroller = new JScrollPane(list);
listScroller.setPreferredSize(new Dimension(250, 80));
listScroller.setMinimumSize(new Dimension(250, 80));
listScroller.setAlignmentX(LEFT_ALIGNMENT);
...
//Lay out the label and scroll pane from top to bottom.
JPanel listPane = new JPanel();
listPane.setLayout(new BoxLayout(listPane, BoxLayout.Y_AXIS));
JLabel label = new JLabel(labelText);
listPane.add(label);
listPane.add(Box.createRigidArea(new Dimension(0,5)));
listPane.add(listScroller);
listPane.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

//Lay out the buttons from left to LEFT.
JPanel buttonPane = new JPanel();
buttonPane.setLayout(new BoxLayout(buttonPane, BoxLayout.X_AXIS));
buttonPane.setBorder(BorderFactory.createEmptyBorder(0, 10, 10, 10));
buttonPane.add(Box.createHorizontalGlue());
buttonPane.add(cancelButton);
buttonPane.add(Box.createRigidArea(new Dimension(10, 0)));
buttonPane.add(setButton);

//Put everything together, using the content pane's BorderLayout.
Container contentPane = getContentPane();
contentPane.add(listPane, BorderLayout.CENTER);
contentPane.add(buttonPane, BorderLayout.SOUTH);
```

La primera línea en negrita crea el boxlayout de ariba-a-abajo y lo inicializada como el controlador de distribución para el **listPane**. Los dos argumentos del constructor de **BoxLayout** son el contenedor que maneja y el eje sobre el que se van a distribuir los componente. Las tres siguiente líneas en negrita añaden la etiqueta y el scroll pane al contenedor, separándolas con un **área rígida** -- un componente invisible de peso ligero usado para añadir espacio entre componentes. En este caso, el área rígida no tiene anchura y pone exactamente 5 pixels entre la etiqueta y el scroll pane. Las áreas rígidas se describen más tarde en Usar componentes invisibles como relleno.

El siguiente bloque de código en negrita crea un boxlyout de izquierda-a-derecha y lo selecciona para contenedor **buttonPane**. Luego el código añade dos botones al contenedor, usando un área rígida de 10 pixels entre los botones. Para hacer que los botones sean situadas en el lado derecho de su contenedor, el primer componente añadido al contenedor es **glue**. Este glue es un componente invisible de peso ligero que crece lo necesario para absrover cualquier espacio extra en su contenedor. Glue se describe en Usar componentes invisibles como relleno.

Como alternativa al uso de componentes invisibles, algunas veces podemos usar bordes vacíos para crear espacio alrededor de los componentes. Por ejemplo, el código anterior usa bordes vacíos de 10 pixels entre todos los lados del dialog y sus contenidos, y entre las dos partes de los contenidos. Los bordes son completamente independientes de los controladores de distribución. Son sólo la forma en que los componentes Swing dibujan sus lados. Para más información puedes ver Cómo usar Borders.

No dejes que te asuste la longitud de esta página. Probablemente podrás usar **BoxLayout** con la información que ua posees. Si tienen problemas o quieres tomar ventaja de la potencia de **BoxLayout** puedes leerlo más tarde.

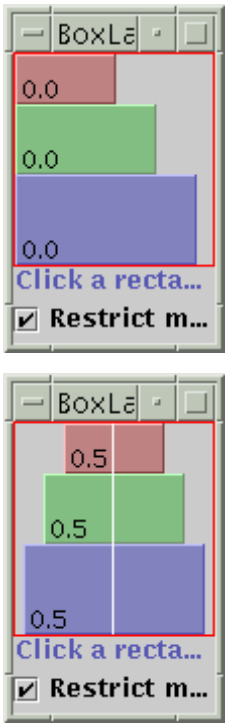
Características de BoxLayout

Como hemos dicho antes, un **BoxLayout** dsiribuye componentes de arriba-a-abajo o de ozquierda-a-derecha. Y esta distribución tiene en cuentra los alineamientos y los tamaño mínimo preferido y máximo de caa componente. En esta sección, hablaremos sobre la distribución de arriba-a-abajo (eje Y). Los mismos conceptos se pueden aplicar a la distribución de izquierda a derecha. Simplemente tenemos que sustituir X por Y, anchura por altura, etc.

Cuando un **BoxLayout** distribuye sus componentes de arriba a abajo, intenta dismiensionarlos a la altura preferida de cada uno de ello. Si la cantidad de espacio vertical no es ideal, el boxlayout intenta ajustar la altura de cada componente para quue los componente llener la cantidad de espacio disponible. Si embargo, los componentes podrían no caber exactamente, ya que **BoxLayout** respeta las alturas mínima y máxima de cada componente. Cualquier espacio extra aparecerá en la parte inferior del contenedor.

Un BoxLayout de arriba-a-abajo intenta hacer que todos sus componentes tengan la misma anchura -- tan anchos como la anchura máxima preferida. Si se fuerza el contenedor a ser más ancho que esto, el boxlayout intenta hacer todos los componentes tan anchos como el contenedor. Si los componentes no son de la misma anchura (debido a las restricciones del tamaño máximo o a que alguno de ellos tiene un alineamiento estricto a la izquierda a derecha), el alineamiento X entra en juego.

El alineamiento X afecta no sólo a la posición relativa de los componentes, sino también a la localización de los componentes (como grupo) dentro del contenedor. Las siguiente figuras ilustran el alineamiento de componentes que tienen anchuras máximas restringidas.

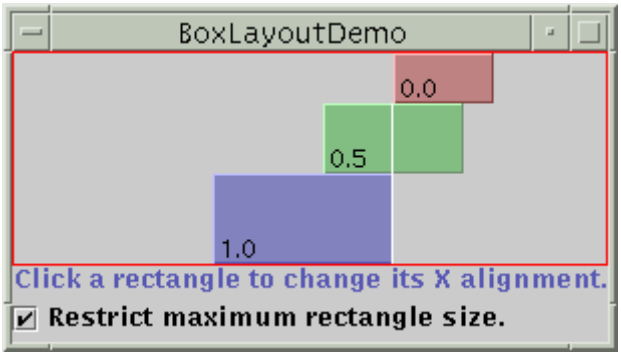


En la primera figura, los tres componentes tienen un alineamiento X de 0.0 (**Component.LEFT_ALIGNMENT**). Esto significa que los lados izquierdos de los componentes deberían estar alineados. Además, significa que los tres componentes deben posicionarse tan a la izquierda de su contenedor como sea posible.

En la segunda figura, los tres componentes tiene un alineamiento X de 0.5 (**Component.CENTER_ALIGNMENT**). Esto significa que los centros de los componentes deben estar alineados, y que los contenedores deberían situarse en el centro horizontal de su contenedor.

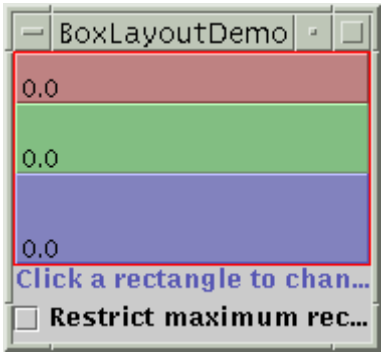
En la tercera figura, los componentes tienen un alineamiento X de 1.0 (**Component.LEFT_ALIGNMENT**). Puedes imaginar lo que esto significa para el alineamiento de los componetes y para su posición relativa dentro del contenedor.

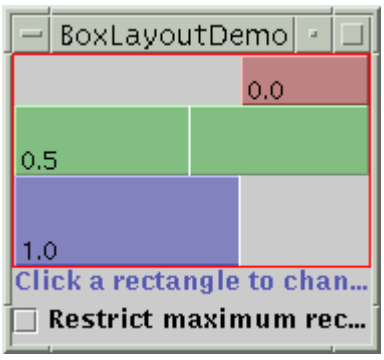
Podríamos preocuparnos por lo que sucederá cuando los tienen tamaños máximo restringidos y diferentes alineamientos X. La siguiente figura muestra un ejemplo de esto.



Como puedes ver, el lado izquierdeo con alineamiento X de 0.0 (**Component.LEFT_ALIGNMENT**) está alineado con el centro del componente con el alineamiento X de 0.5 (**Component.CENTER_ALIGNMENT**), que está alineado con el lado derecho del componente con el alineamiento X de 1.0 (**Component.LEFT_ALIGNMENT**). Los alineamientos mezclados como esto se discuten más adelante en Resolver problemas de Alineamiento.

¿Qué pasa si ningún componente tiene una anchura máxima? Bien, si todos los componentes tienen idéntico alineamiento X, todos los componentes serán tan anchos como su contenedor. Si el alineamiento X es diferente, cualquier componente con alineamiento 0.0 (izquierda) o 1.0 (derecha) serán más pequeños. Todos los componentes con una alineamiento X intermedio (como el centro) serán tan anchos como su contenedor. Aquí tenemos dos ejemplos.





Para conocer mejor BoxLayout, puedes hacer tus propios experimentos con BoxLayoutDemo.

Prueba esto:

1. Compila y ejecuta BoxLayoutDemo. Los ficheros fuentes son **BoxLayoutDemo.java** y **BLDComponent.java**.
2. Pulsa dentro de uno de los rectángulos

   Así es como se cambia el alineamiento X de los rectángulos.

3. Pulsa el check box de la parte inferior de la ventana.

   Esta desactiva las restriciones de tamaño de todos los rectángulos.

4. Agranda la ventana.

Esto hace que el contenedor de los rectángulos sea más grande que la suma de los tamaño máximo preferidos de los rectángulos. El contenedor en un **JPanel** que tiene un borde rojo, por lo que puedes diferenciar donde están los laterales del contenedor.

Usar Componentes Invisibles como Relleno

Cada componente controlador por un BoxLayout se pega a sus vecinos. Si queremos tener espacio entre componentes podemos añadir un borde vacío a uno de los componentes o insertar componentes invisibles para proporcionar espacio. Podemos crear componentes invisibles con la ayuda de la clase **Box**.

La clase **Box** porporciona una clase interna **Box.Filler** que proporciona componentes invisibles. La clase **Box** proporciona métodos de conveniencia para crear los tipos más comunes de rellenos. La siguiente tabla ofrece detalles sobre la creacción de componentes invisibles con **Box** y **Box.Filler**.

Tipo	Restricciones de tamaño		Cómo crearlo
área rígida		XXXXX XXXXX XXXXX	`Box.createRigidArea(size)`
glue	horizontal	<- . ->	`Box.createHorizontalGlue()`
	vertical	^ | . | v	`Box.createVerticalGlue()`
Box.Filler personalizado		(según se especifique)	`new Box.Filler(minSize, prefSize, maxSize)`

Aquí podemos ver como se usa generalmente cada tipo de relleno.

Área Rígida

Se usa cuando queremos un espacio de tamaño fijo entro dos componentes. Por ejmplo, para poner 5 pixels entre dos componentes en un box de izquierda-a-derecha podemos usar este código

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

```
Without rigid area:       With rigid area.
-------------------       -------------------
|XXXXYYY          |       |XXXX     YYY     |
|XXXXYYY          |       |XXXX<--->YYY     |
|XXXXYYY          |       |XXXX     YYY     |
-------------------       -------------------
```

Nota: La clase **Box** proporciona otro tipo de relleno para poner espacio fijo entre componentes: (struts) un puntal vertical u horizontal. Desafortunadamente los puntales no tienen sus anchuras o alturas máximas (para puntales horizontales o verticales, respectivamente). Esto significa que si usamos un box horizontal dentro de un box vertical por ejemplo, el el box horizontal puede algunas veces convertirse demasiado alto. Por esta razón, recomendamos usar áreas rígidas en vez de "struts".

Glue

Se usa para especificar donde debería ir el exceso de espacio en la distribución. Piensa en ello como un goma semi-humeda -- extrechable y expandible, que sólo toma el espacio que los demás componentes no utilizan. Por ejemplo, ponendo un glue horizontal entre dos componentes en un box de izquierda-a-derecha, podemos hacer que todo el espacio extra se quede entre los dos componentes en vez de a la derecha del contenedor. Aquí puedes ver un ejemplo de ello.

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

```
Without glue:             With glue.
-------------------       -------------------
|XXXXYYY          |       |XXXX          YYY|
|XXXXYYY          |       |XXXX<-------->YYY|
|XXXXYYY          |       |XXXX          YYY|
-------------------       -------------------
```

Box.Filler personalizado

Usado para especificar un componente con los tamaños mínimo, preeferido y máximo que querramos. Por ejemplo, para crear algún relleno en un boxlayout de izquierda-a-derecha que ponga al menos 5 pixels entre dos componentes y se asegure de que el contenedor tenga una altura mínima de 100 pixels, podríamos usar este código.

```
container.add(firstComponent);
Dimension minSize = new Dimension(5, 100);
Dimension prefSize = new Dimension(5, 100);
Dimension maxSize = new Dimension(Short.MAX_VALUE, 100);
container.add(new Box.Filler(minSize, prefSize, maxSize));
container.add(secondComponent);
```

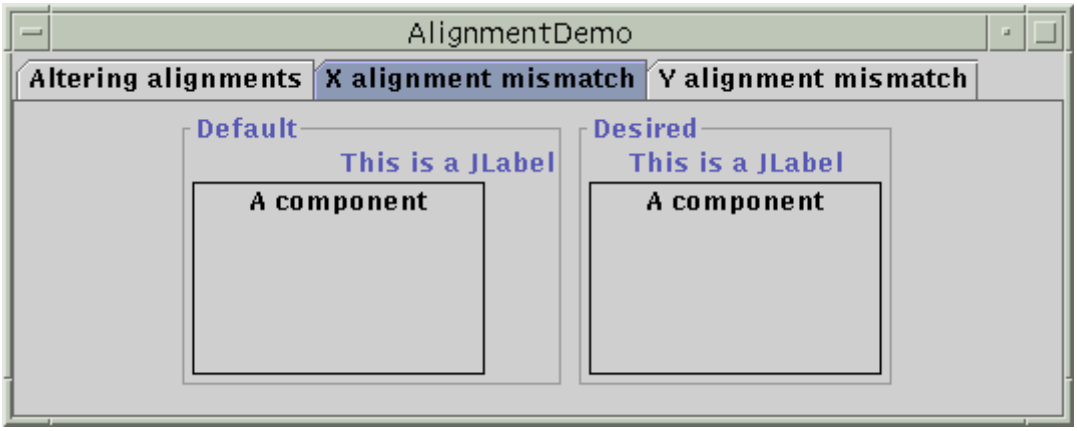
```
Without filler:           With filler.
-------------------       -------------------
|XXXXYYY          |       |        ^        |
|XXXXYYY          |       |XXXX    |    YYY|
|XXXXYYY          |       |XXXX<-------->YYY|
-------------------       |XXXX    |    YYY|
                          |        v        |
                          -------------------
```

Resolver Problemas de Alineamiento

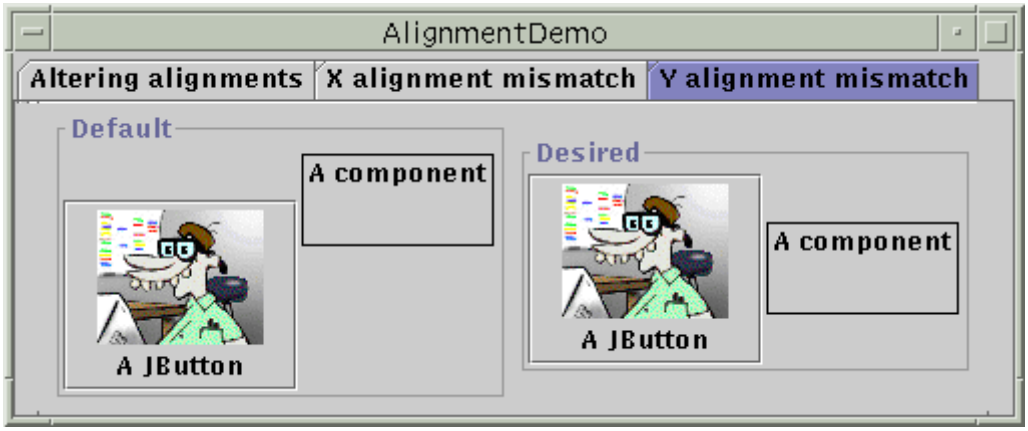
Algunas veces pueden ocurrir dos tipos de problema de alineamiento con **BoxLayout**.

- Un grupo de componentes que tienen el mismo alineamiento, pero queremos cambiar sus alineamientos para que se vean mejor. Por ejemplo, en lugar de tener la parte superior de un grupo de botones de izquierda-a-derecha en un línea, podríamos querer alinear la parte inferior de los botones. Aquí tenemos un ejeplo.

- Dos o más componentes controlados por un **BoxLayout** tienen distinto alineamiento por defecto, lo que hace que estén desalineados. Por ejemplo, como muestra la siguiente imagen, si una etiqueta y un contenedor están en un boxlayout de arriba-a-abajo, el lado izquierda de la etiqueta, por defecto, está alineado con el centro del contenedor.



De forma similar, si un botón y un contenedor están en un boxlayout de izquierda-a-derecha el lado superior del botón, por defecto, está alineado con el centro del contenedor.



En general, todos los componentes controlados por un objeto **BoxLayout** de arriba-a-abajo deberían tener el mismo alineamineto X. Similarmente, todos los componetes controlados por un **Boxlayout** de izquierda-a-derecha deberían tener el mismo alineamiento Y. Podemos seleccionar el alineamiento X de un componente Swing llamando a su método **setAlignmentX**. Una alternativa es sobreescribir el método **getAlignmentX** en un subclase personalizada de la clase del componente. Se puede hacer lo mismo para el alineamiento Y con los métodos **setAlignmentY** y **getAlignmentY**.

Aquí hay un ejemplo, tomado de **AlignmentDemo.java**, de cambio de alineamiento Y de dos botones para la parte inferior de los botones esté alineada.

```
button1.setAlignmentY(Component.BOTTOM_ALIGNMENT);
button2.setAlignmentY(Component.BOTTOM_ALIGNMENT);
```

Como muestra la siguiente tabla, los botones, las etiquetas y los items de menú Swing tienen diferentes valores de alineamiento X que todos los demás componentes. De igual forma, los botones, los items de menú y las barras de herraientas tienen diferentes alineamientis Y que todos los demás componentes.

Componente Swing	Alineamiento X por defecto	Alineamineto Y por defecto
Buttons, menu items	LEFT_ALIGNMENT	TOP_ALIGNMENT
Labels	LEFT_ALIGNMENT	CENTER_ALIGNMENT
Tool bars	CENTER_ALIGNMENT	TOP_ALIGNMENT
Todos los demás componentes Swing	CENTER_ALIGNMENT	CENTER_ALIGNMENT

El programa **AlignmentDemo.java** ofrece ejemplos de cómo corregir los problemas de alineamientos. Normalmente es tan senciilo como forzar a los botones y las etiquetas a que tengan alinamiento centrado. Por ejemplo.

```
label.setAlignmentX(Component.CENTER_ALIGNMENT);
...
button.setAlignmentY(Component.CENTER_ALIGNMENT);
```

Especificar Tamaños de Componentes

Como mencionamos antes, **BoxLayout** presta atención a las peticiones de tamaño mínimo, preferido y máximo del componente. Mientras estemos ajustando la distribución, podríamos necesitar modificar estos tamaños.

Algunas veces la necesidad de ajustar el tamaño es obvia. Por ejemplo, el tamaño máximo de un botón Swing es el mismo que su tamaño preferido. Si queremos que un botón se dibuje más ancho cuando haya espacio disponible, obviamente necesitaremos cambiar el tamaño máximo.

Sin embargo, alguna veces, la necesidad de ajustar el tamaño no es tan obvia. Podríamos obtener resultados inesperdados con un boxlayout, y querrámos saber por qué. En este caso, lo mejor es tratar el problema como si fuera un problema de alineamiento en primer lugar. Si ajustar los alineamientos no ayuda, podríamos tener un problema de tamaños. Lo discutiremos un poco más tarde.

Nota: aunque **BoxLayout** presta atención al tamaño máximo de un componente, la mayoría de los controladores de distribución no lo hacen. Por ejemplo, su ponemos un botón en la parte SOUTH de un **BorderLayout**, el botón probablemente será más ancho que su tamaño preferido. Por otro lado **BoxLayout**, nunca hace un botón tan ancho como su tamaño preferido a menos que especificamente cambiemos su tamaño máximo.

Podemos cambiar los tamaños mínimo, preferido y máximo de dos formas.

- Llamando al método **setXxxSize** apropiado (que está definido por la clase **JComponent**). Por ejemplo.
```
comp.setMinimumSize(new Dimension(50, 25));
comp.setPreferredSize(new Dimension(50, 25));
comp.setMaximumSize(new Dimension(Short.MAX_VALUE,
                                  Short.MAX_VALUE));
```
- Sobreescribiendo el método **getXxxSize** apropiado. Por ejemplo.
```
./in a subclass of a Swing component class:
public void getMaximumSize() {
    size = getPreferredSize();
    size.width = Short.MAX_VALUE; //[PENDING: specify Short or Integer?]
    return size;
}
```

Si tenemos problemas con un boxlayout y hemos resuelto los problemas de alineamiento, el problema podría estar relacionado con el tamaño. Por ejemplo, si el contenedor controlador por el boxlayout está tomando demasiado espacio, entonces uno o más componentes probablemente necesiten tener su tamaño máximo restringido.

Podemos usar dos técnicas para seguir la pista de los problemas de tamaño en un boxlayout.

- Añadir un borde en el lado exterior del componente Swing en cuestión. Esto permitirá ver el tamaño real del componente. Por ejemplo.
```
comp.setBorder(BorderFactory.createCompoundBorder(
                  BorderFactory.createLineBorder(Color.red),
                  comp.getBorder()));
```
- Usar el viejo **System.out.println** para imprimir los tamaños mínimo, preferido y máximo del componente y quizás sus límites.

El API de BoxLayout

Las siguiente tablas listan los métodos y constructores más usados de **BoxLayout** y **Box**. El API para usar BoxLayut se divide en tres categorías.

Crear objetos BoxLayout

Método o Constructor	Propósito
BoxLayout(Container, int)	Crea un ejemplar de **BoxLayout** que controla el **Container** especificado. El argumento entero especifica si los componentes deberían distribuirse de izquierda-a-derecha

	(**BoxLayout.X_AXIS**) o de arriba-a-abajo (**BoxLayout.Y_AXIS**).
Box(int)	Crea un **Box** -- un contenedor de peso ligero que usa **BoxLayout** con el alineamiento especificado (**BoxLayout.X_AXIS** o **BoxLayout.Y_AXIS**). Observa que un **Box** no es un **JComponent** -- está implementado como una subclase de **Container**. Esto hace que sea tan ligero como sea posible, pero se pierde algunas características de **JComponent** como los bordes. Si queremios un sencillo **JComponent** como contenedor, debemos usar **JPanel**.
static Box createHorizontalBox()	Crea un **Box** que distribuye sus componentes de izquierda-a-derecha.
static Box createVerticalBox()	Crea un **Box** que distribuye sus componentes de arriba-a-abajo.

Crear Rellenos

Método o Constructor	Propósito
Component createHorizontalGlue() **Component createVerticalGlue()** **Component createGlue()**	Crea un componente de peso ligero glue.
Component createHorizontalStrut() **Component createVerticalStrut()**	Crea componente de peso ligero "strut" Nosotros recomendamos el uso de áreas rígidas en vez de los struts.
Box.Filler(Dimension, Dimension, Dimension)	Crea un componente de peso ligero con los tamaños mínimo, preferido y máximo especificados (con los argumentos especificados en este orden).

Otros Métodos Útiles

Método	Propósito
void changeShape(Dimension, Dimension, Dimension) (en **Box.Filler**)	Cambia los tamaños mínimo, preferido u máximo del recipiente del objeto **Box.Filler**. La distribución cambiara de forma automática.

¿Cómo usar CardLayout?

Aqui hay un applet que muestra un **CardLayout** en acción.

Esta es una imagen del GUI del applet, para ejecutarlo, pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

Como muestra el applet anterior, la clase **CardLayout** nos ayuda a manejar dos o más componentes, (normalmente ejemplares de **JPanel**) que comparte el mismo espacio. Otra forma de conseguir lo mismo es usar un **JTabbedPane**.

Conceptualmente, cada componente tiene un CardLayout que lo maneja como si jugaran a cartas o las colocaran en una pila, donde sólo es visible la carta superior. Se puede elegir la carta que se está mostrando de alguna de las siguientes formas.

- Pidiendo la primera o la última carta, en el orden en el que fueron añadidas al contenedor.
- Saltando a través de la baraja hacia delante o hacia atrás,
- Especificando la carta con un nombre especifico. Este es el esquema que utiliza el programa de ejemplo. Especificamente, el usuario puede elegir una carta (componente) eligiendo su nombre en una lista desplegable.

Abajo tenemos el código que crea el **CardLayout** y los componentes que maneja. (Aquí está el programa completo. El programa corre como un applet, con la ayuda de **AppletButton**, o como una aplicación.)

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

//Where the container is initialized:
cards = new JPanel();
cards.setLayout(new CardLayout());

...//Create a Panel named p1. Put buttons in it.
...//Create a Panel named p2. Put a text field in it.

cards.add(BUTTONPANEL, p1);
cards.add(TEXTPANEL, p2);
```

Cuando se añaden componentes a un controlador que utiliza un CardLayout, se debe utilizar la forma de dos argumentos del método **add()** de **Container** : **add(String name, Component comp)**. El primer argumento puede ser una cadena con algo que identifique al componente que se está añadiendo.

Para elegir el componente mostrado por el CardLayout, se necesita algún código adicional.

Aquí puedes ver cómo lo hace el applet de esta página.

```
//Where the container is initialized:
. . .
    //Put the JComboBox in a JPanel to get a nicer look.
    String comboBoxItems[] = { BUTTONPANEL, TEXTPANEL };
    JPanel cp = new JPanel();
    JComboBox c = new JComboBox(comboBoxItems);
    c.setEditable(false);
    c.addItemListener(this);
    cp.add(c);
    contentPane.add(cp, BorderLayout.NORTH);
. . .

public void itemStateChanged(ItemEvent evt) {
    CardLayout cl = (CardLayout)(cards.getLayout());
    cl.show(cards, (String)evt.getItem());
}
```

Como se muestra en el código anterior, se puede utilizar el método **show()** de CardLayout para seleccionar el componente que se está mostrando. El primer argumento del método **show()** es el contenedor que controla CardLayout -- esto es, el contenedor de los componentes que maneja CardLayout. El segundo argumento es la cadena que identifica el componente a mostrar. Esta cadena es la misma que fue utilizada para añadir el componente al contenedor.

Abajo tienes todos los métodos de CardLayout que permiten seleccionar un componente. Para cada método, el primer argumento es el contenedor del que CardLayout es el controlador de disposición (el contenedor de las cartas que controla CardLayout).

```
public void first(Container parent)
public void next(Container parent)
public void previous(Container parent)
public void last(Container parent)
public void show(Container parent, String name)
```

Ejemplos que usan CardLayout

Sólo hay un ejemplo en esta lección que use **CardLayout**. **CardWindow.java**. Generalmente nuestros ejemplos usan tabbed panes en lugar de **CardLayout**, ya que los tabbed panes porporcionan un GUI más atractivo con la misma funcionalidad.

¿Cómo usar FlowLayout?

La clase **FlowLayout** proporciona un controlador de distribución muy sencillo, que es usado por defecto por los **JPanel**s. Aquí tenemos un applar que muestra un flowlayout en acción.

Esta es una imagen del GUI del applet, para ejecutarlo, pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

FlowLayout pone los componentes en un fila, domensionados a su tamaño preferido. Si el espacio horizontal del contenedor es demasiado pequeño para poner todos los componentes en un fila, **FlowLayout** usa múltiples filas. Dentro de cada fila, los componentes están centrados (por defecto), alineados a la izquierda o a la derecha como se especifica al crear el **FlowLayout**.

Abajo tenemos el código que crea el **FlowLayout** y los componentes que maneja. (Aquí está el programa completo. El programa corre como un applet, con la ayuda de **AppletButton**, o como una aplicación.)

```
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
contentpane.setFont(new Font("Helvetica", Font.PLAIN, 14));

contentPane.add(new JButton("Button 1"));
contentPane.add(new JButton("2"));
contentPane.add(new JButton("Button 3"));
contentPane.add(new JButton("Long-Named Button 4"));
contentPane.add(new JButton("Button 5"));
```

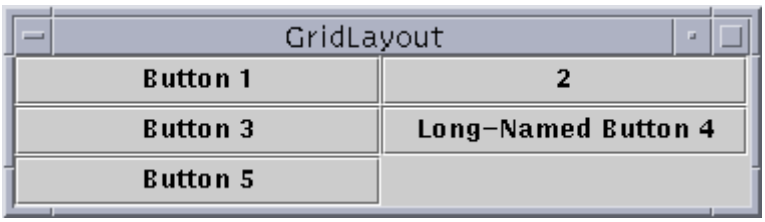
La clase **FlowLayout** tiene tres constructores.

```
public FlowLayout()
public FlowLayout(int alignment)
public FlowLayout(int alignment,
                  int horizontalGap, int verticalGap)
```

El argumento **alignment** debe tener el valor **FlowLayout.LEFT**, **FlowLayout.CENTER**, o **FlowLayout.LEFT**. Los argumentos **horizontalGap** y **verticalGap** especifican el número de pixels entre componentes. Si no especificamos ningún valor, **FlowLayout** utiliza **5** como valor por defecto.

¿Cómo usar GridLayout?

Aquí hay un applet que usa un **GridLayout**.



Esta es una imagen del GUI del applet, para ejecutarlo, pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

Un **GridLayout** sitúa los componentes en una parrilla de celdas. Cada componente toma todo el espacio disponible dentro de su celda, y cada celda es exactamente del mismo tamaño. Si redimensionamos la ventana **GridLayout**, veremos que el **GridLayout** cambia el tamaño de celda para todas sean lo más grande posible, dando el espacio disponible al contenedor.

Abajo tenemos el código quecrea el **GridLayout** y los componentes que maneja. (Aquí está el programa completo. El programa corre como un applet, con la ayuda de **AppletButton**, o como una aplicación.)

```
Container contentPane = getContentPane();

contentPane.setLayout(new GridLayout(0,2));

contentPane.add(new JButton("Button 1"));
contentPane.add(new JButton("2"));
contentPane.add(new JButton("Button 3"));
contentPane.add(new JButton("Long-Named Button 4"));
contentPane.add(new JButton("Button 5"));
```

El constructor le dice a la clase **GridLayout** que cree un ejemplar que tenga dos columnas y tantas filas como sean necesarias. Es uno de los dos constructores de **GridLayout**. Aquí podemos ver los dos juntos.

```
public GridLayout(int rows, int columns)
public GridLayout(int rows, int columns,
                  int horizontalGap, int verticalGap)
```

Al menos uno de los argumentos **rows** y **columns** deben ser distintos de cero. Los argumentos **horizontalGap** y **verticalGap** del segundo constructor permiten especificar el número de pixels entre celdas.

¿Cómo usar GridBagLayout?

Aquí podemos ver un applet que usa un **GridBagLayout**.

Esta es una imagen del GUI del applet, para ejecutarlo, pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

GridBagLayout es el más flexible - y complejo - controlador de disposición porpocionado por la platataforma Java. Como se ve en el applet anterior, un GridBagLayout, sitúa los componentes en una parrilla de filas y columnas, permitiendo que los componentes se espandan más de una fila o columna. No es necesario que todas las filas tengan la misma altura, ni que las columnas tengan la misma anchura. Esencialmente, GridBagLayout sitúa los componentes en celdas en una parrilla, y luego utiliza los tamaños preferidos de los componentes que detemina cómo debe ser el tamaño de la celda.

Si agrandamos la ventana como se vió arriba, observaremos que la fila de abajo, que contiene Button 5, obtiene un nuevo espacio vertical. El nuevo espacio horizontal se divide entre todas las columnas. El comportamiento de redimensionado está basado en pesos que el programa asigna a los componentes individuales en el **GridBagLayout**. También habremos notado que cada componente toda todo el espacio horizontal disponible. Este comportamiento también es especificado por el programa.

La forma en que el programa especifica el tamaño y la posición característicos de sus componetes está especificado por las restricciones de cada componente. Para especificar restricciones, debemos seleccionar las variables de ejemplar en un objeto GridBagConstraints y decírselo al GridBagLayout (con el método **setConstraints()**) para asociar las restricciones con el componente.

Las siguientes páginas explican las restricciones que podemos seleccionar y proporcionan ejemplos...

Especificar Restricciones a GridBagLayout

Abajo está el código que típicamente podremos ver en un contendor que use un **GridBagLayout**.

```
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints c = new GridBagConstraints();

JPanel pane = new JPanel();
pane.setLayout(gridbag);

//For each component to be added to this container:
//...Create the component...
//...Set instance variables in the GridBagConstraints instance...
gridbag.setConstraints(theComponent, c);
pane.add(theComponent);
```

Como podrías haber deducido del ejemplo anterior, se puede reutilizar el mismo ejemplar de GridBagConstraints para varios componentes, incluso si los componentes tienen distintas restricciones.

El GridBagLayout extrae los valores de las restricciones y no vuelve a utilizar el GridBagConstraints.

Sin embargo, se debe tener cuidado de resetar las variables de ejemplar del GridBagConstraints a sus valores por defecto cuando sea necesario.

Podemos seleccionar las siguientes varibles de ejemplar de **GridBagConstraints**.

gridx, **gridy**
: Especifica la fila y la columna de la esquina superior izquierda del componente. La columna más a la izquierda tiene la dirección **gridx=0**, y la fila superior tiene la dirección **gridy=0**. Utiliza **GridBagConstraints.RELATIVE** (el valor por defecto) para especificar que el componente debe situarse a la derecha (para **gridx**) o debajo (para **gridy**) del componente que se añadió al contenedor inmediatamente antes. Recomendamos especificar los valores **gridx** y **gridy** para cada componente, esto tiende a resultar en una distribución más predecible.

gridwidth, **gridheight**
: Especifica el número de columnas (para **gridwidth**) o filas (para **gridheight**) en el área de componente. Esta restricción especifica el número de celdas utilizadas por el componente, no el número de pixels. El valor por defecto es 1. Se utiliza **GridBagConstraints.REMAINDER** para especificar que el componente será el último de esta fila (para **gridwidth**) o columna (para **gridheight**). Se utiliza **GridBagConstraints.RELATIVE** para especificar que el componente es el siguiente al último de esta fila (para **gridwidth**) o columna (para **gridheight**).

  Nota: GridBagLayout no permite que los componentes se expanda múltiples filas a menos que el componente esté en la columna más a la izquierda o especifiquemos valores positivo de **gridx** y **gridy** para componente.

fill
: Utilizada cuando el área del pantalla del componentes es mayor que el tamaño requerido por éste para determinar si se debe, y cómo redimensionar el componente.

  Los valores válidos son **GridBagConstraints.NONE** (por defecto), **GridBagConstraints.HORIZONTAL** (hace que el componente tenga suficiente anchura para llenar horizintalmente su área de dibujo, pero no cambia su altura), **GridBagConstraints.VERTICAL** (hace que el componente sea lo suficientemente alto para llenar verticalmente su área de dibujo, pero no cambia su anchura), y **GridBagConstraints.BOTH** (hace que el componente llene su área de dibujo por completo).

ipadx, **ipady**
: Especifica el espacio interno: cuánto se debe añadir al tamaño mínimo del componente. El valor por defecto es cero. La anchura del componente debe ser al menos su anchura mínima más **ipadx*2** pixels (ya que el espaciado se aplica a los dos lados del componente). De forma similar, la altura de un componente será al menos su altura mínima más **ipady*2** pixels.

insets
: Especifica el espaciado externo del componente -- la cantidad mínima de espacio entre los componentes y los bordes del área de dibujo. Es valor es especificado como un objeto Insets. Por defecto, ningún componente tiene espaciado externo.

anchor
: Utilizado cuando el componente es más pequeño que su área de dibujo para determinar dónde (dentro del área) situar el componente. Los valores válidos (definidos como constantes en **GridBagConstraints**) son **CENTER** (por defecto), **NORTH**, **NORTHEAST**, **EAST**, **SOUTHEAST**, **SOUTH**, **SOUTHWEST**, **WEST**, y **NORTHWEST**.

weightx,**weighty**
: Especificar el peso es un arte que puede tener un impacto importante en la apariencia de los componentes que controla un GridBagLayout. El peso es utiliza para determinar cómo distribuir el espacio entre columnas (**weightx**) y filas (**weighty**); esto es importante para especificar el comportamiento durante el redimensionado.

  A menos que se especifique un valor distinto de cero para **weightx** o **weighty**, todos los componentes se situarán juntos en el centro de su contenendor. Esto es así porque cuando el peso es 0,0 (el valor por defecto) el GidBagLayout pone todo el espacio extra entre las celdas y los bordes del contenedor.

  Generalmente, los pesos son especificados con 0.0 y 1.0 como los extremos, con números entre ellos si son necesarios. Los números mayores indican que la fila o columna del componente deberían obtener más espacio. Para cada columna, su peso está relacionado con el mayor **weightx** especificado para un componente dentro de esa columna (donde cada componente que ocupa varias columnas es dividido de alguna forma entre el número de columnas que ocupa).

  Lo mismo ocurre con las filas para el mayor valor especificado en **weighty**. El espacio extra tiende a irese hacia la columna más a la derecha y hacia la fila más abajo.

La página siguiente explica las oblicaciones en más detalle, explicando cómo trabaja el programa del ejemplo.

Ejemplo de GridBagLayout

De nuevo, aquí está el applet que muestra un **GridBagLayout** en acción

Esta es una imagen del GUI del applet, para ejecutarlo, pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

Abajo tienes el código que crea el GridBagLayout y los componentes que maneja. Aquí tienes el programa completo. El programa puede ejecutarse dentro de un applet, con la ayuda de AppletButton, o como una aplicación.

```
JButton button;
Container contentPane = getContentPane();
GridBagLayout gridbag = new GridBagLayout();
GridBagConstraints c = new GridBagConstraints();
contentPane.setLayout(gridbag);
c.fill = GridBagConstraints.HORIZONTAL;

button = new JButton("Button 1");
c.weightx = 0.5;
c.gridx = 0;
c.gridy = 0;
gridbag.setConstraints(button, c);
contentPane.add(button);

button = new JButton("2");
c.gridx = 1;
c.gridy = 0;
gridbag.setConstraints(button, c);
contentPane.add(button);

button = new JButton("Button 3");
c.gridx = 2;
c.gridy = 0;
gridbag.setConstraints(button, c);
contentPane.add(new JButton("Button 3"));

button = new JButton("Long-Named Button 4");
c.ipady = 40;      //make this component tall
c.weightx = 0.0;
c.gridwidth = 3;
c.gridx = 0;
c.gridy = 1;
gridbag.setConstraints(button, c);
contentPane.add(button);

button = new JButton("Button 5");
c.ipady = 0;       //reset to default
c.weighty = 1.0;   //request any extra vertical space
c.anchor = GridBagConstraints.SOUTH; //bottom of space
c.insets = new Insets(10,0,0,0);  //top padding
c.gridx = 1;       //aligned with button 2
c.gridwidth = 2;   //2 columns wide
c.gridy = 2;       //third row
gridbag.setConstraints(button, c);
contentPane.add(button);
```

Este ejemplo utiliza un ejemplar de **GridBagConstraints** para todos los componetes manejados por el GridBagLayout. Justo antes de que cada componente sea añadido al contenedor, el código selecciona (o resetea a los valores por defecto) las variables apropiadas del objeto GridBagConstraints. Luego utiliza el método **setConstraints()** para grabar los valores obligatorios de ese componente.

Por ejemplo, para hacer que el botón 4 sea extra alto, el ejemplo tiene este código.

```
c.ipady = 40;
```

Y antes de seleccionar las restricciones para el siguiente componente, el códio reseta el valor de **ipady** al vaor por defecto

```
c.ipady = 0;
```

Para mayor claridad aquí tienes una tabla que muestra todas las obligaciones para cada componente manejado por GridBagLayout. Los valores que no son por defecto están marcados en **negrita**. Los valores que son diferentes de su entrada anterior en la tabla están marcados en itálica.

Componente	Restricciones
Todos los componentes	ipadx = 0 **fill = GridBagConstraints.HORIZONTAL**
Button 1	ipady = 0 **weightx = 0.5** weighty = 0.0 gridwidth = 1 anchor = GridBagConstraints.CENTER insets = new Insets(0,0,0,0) **gridx = 0** **gridy = 0**
Button 2	**weightx = 0.5** **gridx = 1** **gridy = 0**
Button 3	**weightx = 0.5** **gridx = 2** **gridy = 0**
Button 4	**ipady = 40** weightx = 0.0 **gridwidth = 3** **gridx = 0** **gridy = 1**
Button 5	ipady = 0 weightx = 0.0 **weighty = 1.0** **anchor = GridBagConstraints.SOUTH** **insets = new Insets(10,0,0,0)** **gridwidth = 2** **gridx = 1** **gridy = 2**

Todos los componentes de este contenedor son tan grandes como sea aposible, dependiendo de su fila y columna. El programa consigue esto selección la variable **fill** de GridBagConstraints a **GridBagConstraints.HORIZONTAL**, dejándola seleccionada para todos los componentes. Si el programa no seleccionar el relleno, los botones serían de su anchura natural, de esta forma.

Este programa tiene cuatro componentes que se espanden varias columnas (buttons 4 y 5). Para hacer más alto el botón 4, le añadimos un borde interno (**ipady**). Para poner espacio entre los botones 4 y 5, usamos insets para añadir un mínimo de 10 pixels sobre el botón 5 y para que el botón 5 se situe en la parte inferior de su celda.

Cuando se agranda la ventana del programa, la anchura de las columnas crece los mismo que la ventana. Este es el resultado de cada componente de la primer fila (donde cada componente tiene la anchura de una columna) tiene **weightx = 1.0**. El valor real del **weightx** de estos componentes no tiene importancia. Lo que importa es que todos los componentes (y así todas las columnas) tienen el mismo peso que es mayor que cero. Si ningún componente manejado por el GridBagLayout tuviera seleccionado **weightx**, cuando se ampliara la anchura del contenedor, los componentes permanecerían juntos en el centro del contenedor de sta forma.

Observamos que alargamos la ventana, la última fila es la que se hace más alta. Es es así porque sólo el botón 5 tiene **weighty** mayor que cero.

Crear un Controlador de Distribución

Nota: Antes de empezar a crear un controlador de disposición personalizado, asegurate de que no existe ningún controlador que trabaje de la forma apropiada. En particular, GridBagLayout y **BoxLayout** son lo suficientemente flexibles para trabajar en muchos casos. También puedes encontrar controladores de distribución desde otras fuentes, como Internet. Finalmente, puedes simplificar la distribución agrupando los componentes en contenedores como en un panel invisible

Para crear un controlador de distribución personalizado, debemos crear una clase que implemente el interface **LayoutManager**. Podemos implementando directamente o implementar su subinterface, **LayoutManager2**.

Todo controlador de distribución debe implementar al menos estos cinco métodos, que son requeridos por el interface **LayoutManager**.

void addLayoutComponent(String, Component)
: Llamado por los métodos **add** de **Container**. Los controladores de distribución que no asocian cadenas con sus componentes generalmente no hacen nada en este métodos.

void removeLayoutComponent(Component)
: Llamado por los métodos **remove** y **removeAll** de **Container**. Los controladores de distribución no hacen nada en este metodo, en su lugar le preguntan por sus componentes al contenedor usando el método **getComponents** de **Container**.

Dimension preferredLayoutSize(Container)
: Llamado por el método **getPreferredSize** de **Container** que a su vez es llamado bajo una gran variedad de circunstancias. Este método debería calcular y devolver el tamaño ideal del contenedor, asumiendo que todos sus componentes serán de sus tamaños preferidos. Este método debe tener en cuenta los bordes internos del contenedor que son devueltos por el método **getInsets**.

Dimension minimumLayoutSize(Container)
: Llamado por el método **getMinimumSize** de **Container**, que a su vez es llamado bajo una gran variedad de circunstancias. Este método debería calcular y devolver el tamaño mínimo del contenedor, asumiendo que todos sus componentes serán de sus tamaños mínimos. Este método debe tener en cuenta los bordes internos del contenedor que son devueltos por el método **getInsets**.

void layoutContainer(Container)

Llamado cuando el contenedor se muestra por primera vez, y cada vez que cambie su tamaño. Este método realmente no dibuja componentes, sólo invoca a los métodos **resize**, **move**, y **reshape** de cada componente, para seleccionar su tamaño y posición. Este método debe tener en cuenta los bordes internos del contenedor que son devueltos por el método **getInsets**. No podemos asumir que se llamara a **preferredLayoutSize** o **minimumLayoutSize** antes de llamar a **layoutContainer**.

Además de la implementación de los cinco métodos anteriores, los controladores de distribución generalmente implementan un constructor público y el método **toString**.

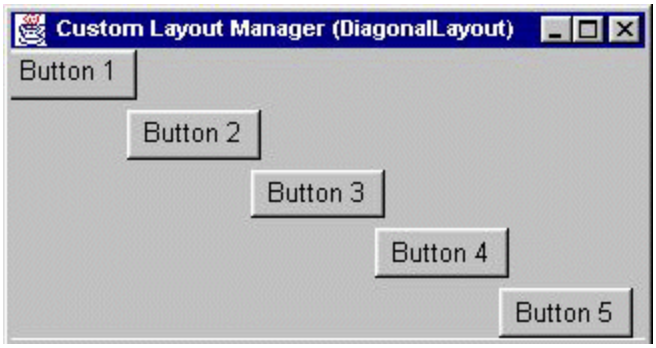
Si deseamos soportar restricciones de componnetes, tamaños máximos o alineamiento, entonces nuestro controlador de distribución debería implementar el interface **LayoutManager2**. Este interface añade cinco métodos a los requeridos por **LayoutManager**.

- **addLayoutComponent(Component, Object)**
- **getLayoutAlignmentX(Container)**
- **getLayoutAlignmentY(Container)**
- **invalidateLayout(Container)**
- **maximumLayoutSize(Container)**

Para más información sobre estos métodos, puedes ver **LayoutManager2** API documentation. También puedes ver el código fuente de **BoxLayout**, para ver como implementa el interface **LayoutManager2**.

Cuando se implementa un controlador de distribución, podríamos queres usan un objeto **SizeRequirements** para ayudarnos a determinar el tamaño y posición de los componentes. Puedes ver el código fuente de **BoxLayout** para ver un ejemplo de uso de **SizeRequirements**.

Aquí tienes el código fuente de DiagonalLayout, un controlador de distribución personalizado que distribuye sus componentes diagonalmente, de izquierda a derecha, con un componente por fila. Aquí tenemos a **DiagonalLayout** en acción.



Esta es una imagen del GUI del Applet. Para ejecutarlo pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

Hacerlo sin Controlador de Distribución

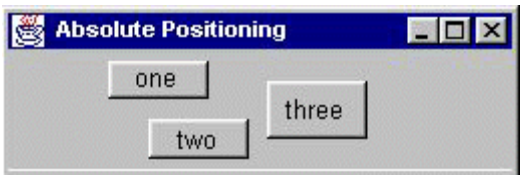
Aunque es posible hacerlo sin un controlador de disposición, se debería utilizar un controlador disposición siempre que sea posible. Los controladores de disposición hacen más sencillo el redimensionado de un contenedor y se ajustan al 'Aspecto y Comportamiendo' dependientes de la plataforma y los diferentes tamaños de las fuentes. También pueden ser reutilizados fácilmente por otros contenedores y otros programas.

Si un contenedor personalizado no será reutilizado ni redimensionado, y controla normalmente los factores dependientes del sistema como el tamaño de las fuentes y la apariencia de los componentes (implementando sus propios controles si fuera necesario), entonces, el posicionamiento absoluto podría tener sentido.

Los Desktop panes, que contienen frames internos, están en esta categoría. El tamaño y posición de los frames internos no depende directamente de los tamaño del desktop pane. El programador determina el tamaño y situación iniciales del frame interno dentro del desktop pane, y luego el usuario puede mover o redimensionar los marcos. En esta situación un controladores de distribución es innecesario.

Otra situación en que el posicionamiento absoluto podría tener sentido es un contenedor personalizado que realice cálculos de tamaño y posición que son particulares del contenedor y que quizás requieran un conocimiento especializado del estado del contenedor. Esta es la situación de los split panes.

Aquí tenemos un applet que muestra una ventana cuyo panel de contenido usa posicionamiento absoluto



Esta es una imagen del GUI del Applet. Para ejecutarlo pulsa sobre ella y el applet aparecerá en una nueva ventana del navegador.

Abajo podemos ver las declaraciones de las variables y la implementación del constructor de la clase window. Aquí puedes ver el programa completo. El programa se puede ejecutar como un applet, con la ayuda de **AppletButton**, o como una aplicación.

```
public class NoneWindow extends JFrame {
    . . .
    private boolean laidOut = false;
    private JButton b1, b2, b3;

    public NoneWindow() {
        Container contentPane = getContentPane();
        contentPane.setLayout(null);

        b1 = new JButton("one");
        contentPane.add(b1);
        b2 = new JButton("two");
        contentPane.add(b2);
        b3 = new JButton("three");
        contentPane.add(b3);

        Insets insets = contentPane.getInsets();
        b1.setBounds(25 + insets.left, 5 + insets.top, 75, 20);
        b2.setBounds(55 + insets.left, 35 + insets.top, 75, 20);
        b3.setBounds(150 + insets.left, 15 + insets.top, 75, 30);
        . . .
    }
    . . .
}
```

Problemas con el Controlador de Distribución

Problema:¿Cómo puedo especificar el tamaño exacto de un componente?

- Primero, asegúrate de que realmente necesitas seleccionar el tamaño exacto del componente. Los componentes estandards tienen distintos tamaños, dependiendo de la plataforma donde se están ejecutando y de la fuente utilizada, por eso normalmente no tiene sentido especificar su tamaño exacto.

  Para los componentes personalizados que tienen contenidos de tamaño fijo (como imágenes), especificar el tamaño exacto tiene sentido. Para ellos, necesitamos sobreescribir los métodos **minimumSize()** y **preferredSize()** del componente para devolver su tamaño correcto.

Nota: No insporta cómo especifiquemos el tamaño de nuestro componente, debemos asegurarnos de que el contenedor de nuestro componente utiliza un controlador de distribución que respeta las peticiones de tamaño del componente. Los controladores **FlowLayout** y **GridBagLayout** usan los tamaños preferidos de los componentes (dependiendo de las restricciones activadas), pero **BorderLayout** y **GridLayout** normalmente lo no hace. El controlador **BoxLayout** generalmente usa el tamaño preferido del componente (aunque los componentes pueden ser más grandes) y es el único controlador de distribución que respeta el tamaño máximo del componente.

Si especificamos un nuevo tamaño para un componente que ya está visible, necesitamos invocar al método **revalidate** sobre él, para asegurarnos de que todo el arbol de herencia también se ha revalidado. Luego llamamos al método **repaint**.

Problema: Mi componente personalizado se dibuja demasiado pequeño.

- ¿Has implementado los métodos **preferredSize()** and **minimumSize()** del componente? Si lo has hecho, ¿devuelven los valores correctos?
- ¿Estás utilizando un controlador de distribución que puede utilizar todo el espacio disponible? Puedes ver Reglas Generales para el Uso de Controladores de Disposición para ver algunas situaciones de elección del controlador de distribución y especificar que utilice el máximo espacio disponible para un componente particular.

¿Cómo usar Action?

Si tenemos dos o más componentes que realizan la misma funcion, podemos considerar la utilización de un objeto **Action** para implementar la función. Un objeto **Action** es un **ActionListener** que no sólo proporciona manejo de eventos, sino que también centraliza el manejo de texto, iconos, y estados de enable de tool bar buttons o itéms de menu. Añadiendo un **Action** a un **JToolBar**, **JMenu**, o un **JPopupMenu**, obtendremos las siguientes características.

- Un nuevo **JButton** (para **JToolBar**) o **JMenuItem** (para **JMenu** y **JPopupMenu**) que se añade automáticamente a la barra de herramientas o menú. El botón o ítem de menú usa automáticamente el icono y el texto especificado por **Action**.
- Un oyente de action registrado (el objeto **Action**) para el botón o el ítem de menú.
- Manejo centralizado para el estado del botón o ítem de menú.

Aquí hay un ejemplo de uso de un **Action** para crear un botón de una barra de herramientas y un ítem de menú que realizan la misma función.

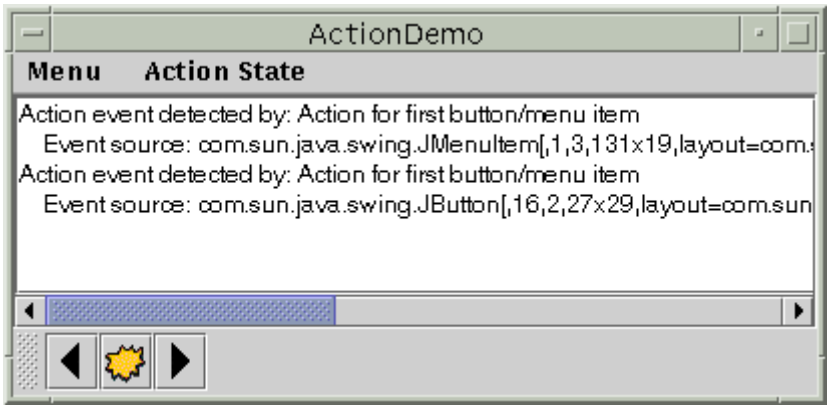
```
Action leftAction = new <a class that implements Action>(...);
JButton button = toolBar.add(leftAction);
JMenuItem menuItem = mainMenu.add(leftAction);
```

Para que un botón o un ítem de menú obtengan todo el beneficio de usar un **Action**, debemos crear el componente usando el método **add(Action)** de **JToolBar**, **JMenu**, o **JPopupMenu**. Actualmente no existe ningún API que permita conectar un objeto **Action** con un componente ya existente. Por ejemplo, aunque podamos añadir un objeto **Action** como oyente de action de cualquier botón, el botón no será notificado cuando la acción esté deshabilitada.

Para crear un objeto **Action**, generalmente se crea una subclase de **AbstractAction** y luego se ejemplariza. En nuestra subclase debemos implementar el método **actionPerformed** para reaccionar de forma apropiada cuando ocurra el evento action. Aquí hay un ejemplo de creacción y ejemplarización de una sublcase **AbstractAction**.

```
leftAction = new AbstractAction("Go left",
                                new ImageIcon("images/left.gif")) {
    public void actionPerformed(ActionEvent e) {
        displayResult("Action for first button/menu item", e);
    }
};
```

Aquí tenemos una imagen de la aplicación que usa actions para implementar características de árbol.



Prueba esto:

1. Compila y ejecuta la aplicación. El fichero fuente es **ActionDemo.java**. También necesitarás tres ficheros de imágenes: **left.gif**, **middle.gif**, y **LEFT.gif**.
2. Elige el ítem superior del menú de la izquierda(Menu->Go left).

   El área d etexto muestra algún texto identificando tanto la fuente del evento como el oyente que lo ha recibido.

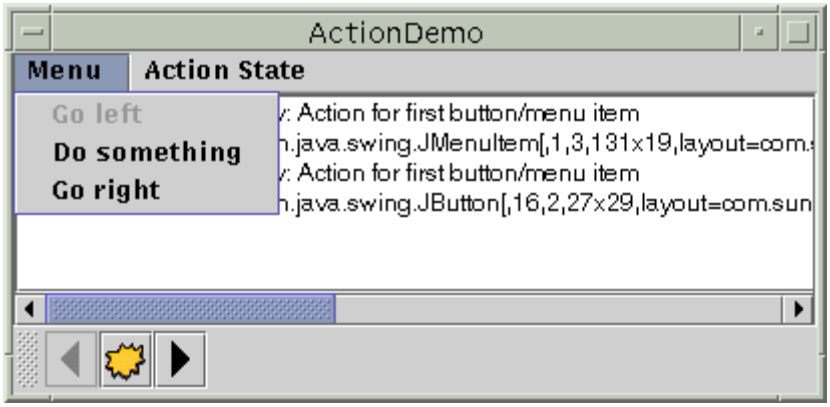
3. Pulsa el botón más a la izquierda de la barra de herramientas.

   De nuevo el área de texto muestra información sobre el evento. Observa que aunque la fuente del evento sea diferente, ambos eventos fueron detectados por el mismo oyente de action: el objeto **Action** con el que se creó el componente.

4. Elige el ítem superior del menú Action State.

   Esto desactiva el objeto **Action** "Go Left", que desactiva sus objetos asociados, el botón y el ítem de menú.

Aquí está lo que ve el usuario cuando la acción "Go left" está desactivada.



Aquí está el código que desactiva la acción "Go left".

```
boolean selected = ...//true if the action should be enabled; false, otherwise
leftAction.setEnabled(selected);
```

Después de crear nuestros componentes usando un **Action**, podríamos necesitar personalizarlos. Por ejemplo, podríamos querer seleccionar el texto del tool-tip del botón. O podríamos personalizar la apariencia de uno de los componentes añadiendo o borrando el icono o el texto. Por ejemplo, **ActionDemo.java** no tiene iconos en sus menús, ni texto en sus botones, ni tool-tips. Aquí está el código que consigue esto.

```
button = toolBar.add(leftAction);
button.setText(""); //an icon-only button
button.setToolTipText("This is the left button");
menuItem = mainMenu.add(leftAction);
menuItem.setIcon(null); //arbitrarily chose not to use icon in menu
```

API de Action

Las siguientes tablas listan los métodos y constructores más usados de **Action**. El API para usar objetos **Action** se divide en dos categorías.

Crear y Usar un action

Método o constructor	Propósito
AbstractAction() **AbstractAction(String)** **AbstractAction(String, Icon)**	Crea un objeto **Action**. A través de los argumentos, podemos especificar el texto y el icono usados en los componentes que controla el action.
void setEnabled(boolean) **boolean isEnabled()**	Selecciona u obtiene si los componetes que controla el actión están activados. Llamanado a **setEnabled(false)** se desactivan todos los componentes que controla. De forma similar, llamando a **setEnabled(true)** se activan todos los componentes del action.

Crear un componente Controlador por un Action

Método	Propósito
JMenuItem add(Action) **JMenuItem insert(Action, int)** (en **JMenu** y **JPopupMenu**)	Crea un objeto **JMenuItem** y lo pone en el menú. Puedes ver la sección Cómo usar Menus para más detalles.
JButton add(Action) (in **JToolBar**)	Crea un objeto **JButton** y lo pone en la barra de herramientas. Puedes ver la sección Cómo usar Tool Bars para más detalles.

Ejemplos que usan Actions

Actualmente, el único ejemplo que usa objetos **Action** es **ActionDemo.java**, que se usa y se describe en esta página.

¿Cómo Soportar Tecnologías Asistivas?

Podrías estar preguntándote que son exactamente las tecnologías asistivas, y por qué debemos tener cuidado. Primero, las tecnologías asistivas existen para permitir usar los ordenadores a las personas con discapacidades. Por ejemplo, si tenemos el síndrome del tunel carpal, podríamos usar tecnologías asistivas para realizar nuestro trabajo sin usar las manos.

Las tecnologías asistivas -- interfaces de voz, lectores de pantallas, dispositivos de entrada alternativos, etc. --- no son sólo útiles para la gente con discapacidades, sino también para la gente que usa los ordenadores fuera de la oficina. Por ejemplo, si están atrapado en un atasco de tráfico, podrías utilizar las tecnologías asistivas para chequear tu e-mail, usando sólo la voz. La accesibilidad a la información puede usarse en otras herramientas, como probadores de GUI automáticos.

Las tecnologías asistivas también graban automáticamente el texto del tool-tip asociado con un componente y puede usarse para describir el componente al usuario.

Aquí hay unas cuantas cosas que podemos hacer para que nuestros programas funcionen también con tecnologías asistivas.

- Usa tool-tips, siempre que tengan sentido.
- Especifica mnemócinos de teclado siempre que sea posible. Debería ser posible usar nuestro programa sólo con el teclado. Trata de olvidar el ratón!
- Siempre que tengas un **JLabel** que muestre un mnemónico para otro componente (como un campo de texto), usa el método **setLabelFor** para que las tecnologías asistivas puedan encontrar el componente asociado con la etiqueta.
- Usa el método **setDescription** para proporcionar una descripción para todos los **ImageIcon** de tu programa.
- Si no proporcionas un tool-tip para un componente, usa el método **setAccessibleDescription** para proporcionar una descripción que las tecnologías asistivas puedan darle al usuario.
- Si un componente no muestra una cadena corta (que sirve como su nombre por defecto), especifica un nombre con el método **setAccessibleName**. Podrías querer hacer esto para botónes que són una imagen, paneles que proporcionan agrupamientos lógicos, áreas de texto, etc.
- Si un grupo de componentes forman un grupo lógico, intenta ponerlos juntos en un componente Swing. Por ejemplo, usa un **JPanel** para contener todos los botones de radio de un grupo de botones de radio.

¿Cómo usar Iconos?

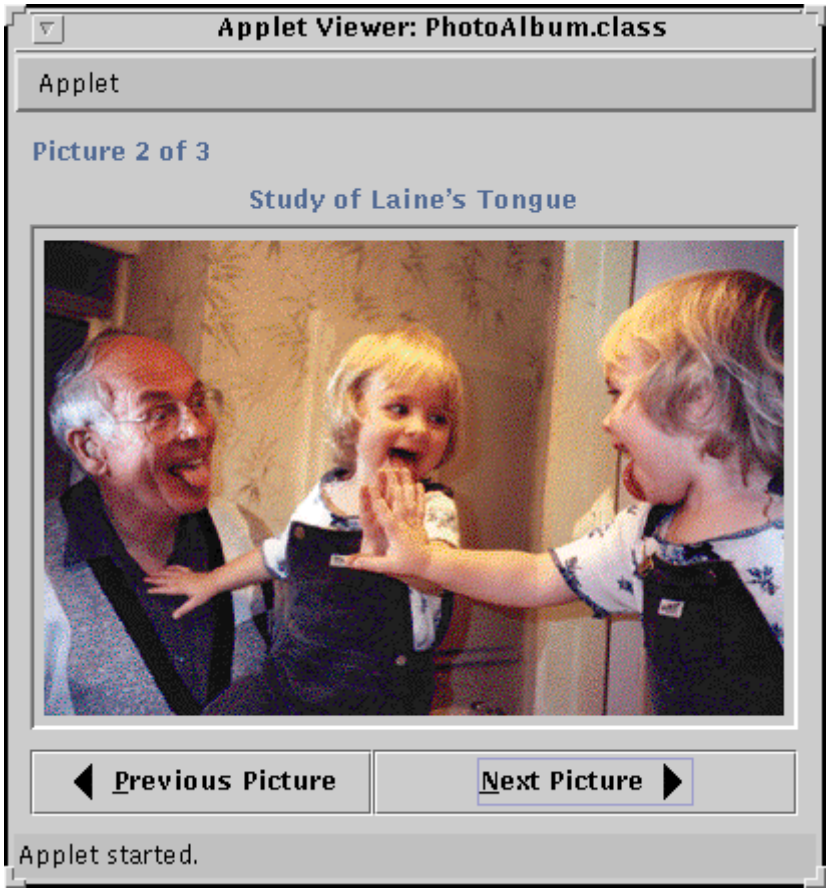
Algunos componentes Swing, como **JLabel** y **JButton**, pueden ser decorados con un icono -- una imagen de tamaño fijo. En Swing, un icono es un objeto que se adhiere al interface **Icon**. Swing proporciona una implementación particularmente útil del interface **Icon**.

ImageIcon, que dibuja un icono desde una imagen JPEG o GIF..

Nota a los programadores AWT: un objeto **ImageIcon** usa **MediaTracker** para cargar la imagen desde un nombre de fichero, una URL u otra fuente.

ImageIcon es una alternativa manejable y fácil de usar para **Image** porque **ImageIcon** maneja la relación con el **MediaTracker** y sigue la pista del estado de la carga de la imagen. Sin embargo, se puede obtner el objeto **Image** desde un **ImageIcon** si necesitamos más control.

Aquí tenemos un applet que usa iconos para dos propósitos diferentes.



Está es una imagen del GUI del Applet, para ejecutarlo, pulsa sobre ella. El applet aparecerá en una nueva ventana del navegador.

Un icono en una etiqueta impelementa el área fotgráfica del applet. El applet tambíen usa iconos para decorar los botones **Previous Picture** y **Next Picture** de la parte inferior de la ventana del applet.

Prueba esto:

1. Ejecuta el applet.

   El applet anterior se ejecuta usando Java Plug-in. Si lo prefieres, puedes ejecutarlo con un navegador o Applet Viewer que soporte JDK 1.1 y Swing. Aquí está el fichero que contiene la etiqueta **<APPLET>** para ejecutar el applet **IconDemoApplet.java**. Para más información sobre la ejecución de applets, puedes ver Ejecutar Applets Swing.

2. Pulsa los botones **Previous Picture** y **Next Picture**, para ver las fotos.
3. Mantén el cursor sobre una foto. Un tool-tip indicará el nombre de la foto actual y su anchura y altura.
4. PAra ver tus propias fotos, modifica los parámetros del applet. Necesitarás proporcionar, el nombre, la anchura y la altura para cada foto. Durante la inicialización, el applet lee sus parámetros y almacena la informaicón en un vector de objetos **Photo**.

Primero veamos el código de **IconDemoApplet.java** que crea e inicializa las felchas usadas en los botones del applet, porque es un código muy sencillo.

```
//create the image icons for the next and previous buttons
ImageIcon nextIcon = new ImageIcon(getURL("images/LEFT.gif"));
ImageIcon previousIcon = new ImageIcon(getURL("images/left.gif"));
...
//use them to create a buttons
previous = new JButton("Previous Picture", previousIcon);
...
next = new JButton("Next Picture", nextIcon);
```

El único argumentos para el constructor del icono es la URL del fichero que contiene la imagen. El método **getURL** añade el code base del applet al nombre del fichero que contiene la imagen del applet. Puedes copiar este método para usarlo en tus applets.

```
protected URL getURL(String filename) {
    URL codeBase = this.getCodeBase();
    URL url = null;

    try {
        url = new URL(codeBase, filename);
    } catch (java.net.MalformedURLException e) {
        System.out.println("Couldn't create image: badly specified URL");
        return null;
    }
    return url;
}
```

La clase **ImageIcon** proporciona otros muchos constructores para crear iconos desde un array de bytes, una imagen o desde un nombre de fichero.

Ahora veamos el código que carga las fotografías.

```
//where the member variables are declared
Vector pictures;
...
                //early in the init method
        pictures = parseParameters();

                //create the image icon for the photo
                Photo first = (Photo)pictures.firstElement();
                ImageIcon icon = new ImageIcon(getURL(first.filename));
        first.setIcon(icon);
                ...
```

Este código crea un **Vector** de objetos **Photo** (en el método **parseParameters** que no se muestra). Cada objeto **Photo** contiene el nombre, el captión, la anchura y la altura de la foto que representa, y después de que la imagen se muestre por primera vez, su icono. El icono para la primera foto es creado en el método **init** del applet y almacenado en el objeto **Photo** apropiado.

Los iconos para las otras fotos se almacenan la primera vez que el usuario las ve. Aquí está el código del método **actionPerformed** de buttons que determina si una imagen ha sido visualizada con anterioridad. Si no, el código crea un nuevo icono y lo almacena.

```
Photo pic = (Photo)pictures.elementAt(current);
icon = pic.getIcon();
if (icon == null) {
    icon = new ImageIcon(getURL(pic.filename));
    pic.setIcon(icon);
}
iconLabel.setText("");
iconLabel.setIcon(icon);
iconLabel.setToolTipText(pic.filename + ": " + icon.getIconWidth() +
                                          " X " + icon.getIconHeight());
```

¿Por qué todo este alboroto para almacenar los iconos? El programa corre mucho más rápido porque los iconos son creados sólo una vez y las correspondientes imágenes también solo se cargan una vez. Si eliminamos explicitamente el almacenamiento de iconos de este prorama, una segunda visualización de un foto parecerá suceder más rápidamente que la primera. Esto implica que algún almacenamiento implícito está sucediendo en la plataforma Java. Sin embargo, esto es un efecto colateral de la implementación y no está garantizado.

El código también selección el tool-tip de la foto: el programa llama a los métodos **getIconWidth** y **getIconHeight** de **IgameIcon** para obtener información sobre el tool-tip. La anchura proporcionadas por el icono son más correctas que las proporcionadas por los parámetros del applet.

El API de Icon

La siguientes tablas listan los métodos y constructores más usados de **ImageIcon**. El API para isar Iconos se divide en tres categorías.

Seleccionar u Obtener la Imagen Dibujada por el Icono

Método o Constructor	Propósito
ImageIcon(byte[]) **ImageIcon(byte[], String)** **ImageIcon(Image)** **ImageIcon(Image, String)** **ImageIcon(String)** **ImageIcon(String, String)** **ImageIcon(URL)** **ImageIcon(URL, String)**	Crea un ejemplar de **ImageIcon**, inicializando su contenido con la imagen especificada. El primer argumento indica la fuente -- imagen, array de bytes, nombre de fichero o URL -- desde la que se debería cargar la imagen del icono. El segundo argumento, cuando existe, proporciona una descripción para la imagen. La descripción es una descripción corta y textual de la imagen que podría ser usada en varias formas, como texto alternativo de la imagen.

void setImage(Image) **Image getImage()**	Selecciona u obtiene la imagen mostrada por el icono.

Seleccionar u Obtener Información sobre el Icono

Método	Propósito
void setDescription(String) **String getDescription()**	Selecciona u obtiene una descripción de la imagen.
int getIconWidth() **int getIconHeight()**	Obtiene el tamaño de la imagen del icono.

Vigilar la Carga de la Imagen del Icono

Método	Propósito
void setImageObserver(ImageObserver) **ImageObserver getImageObserver()**	Selecciona u Obtiene un image observer para la imagen del icono.
int getImageLoadStatus()	Obtiene el estado de la carga de la imagen del icono. El conjunto de valores devueltos por este método está definido por **MediaTracker**.

¿Cómo Seleccionar el Aspecto y Comportamiento?

Si no te importa que aspecto y comportamiento usan tus programas, puedes saltarte está página. Por ejemplo, la mayoría de los programas de esta sección no especifican el aspecto y comportamiento, por lo que podremos ejecutar programas fácilmente sin seleccionar el aspecto y comportamiento.

Cuando un programa no selecciona el aspecto y comportamiento, el controlador del Swing debe imaginarse cual utilizar. Primero chequea si el usuario ha especificado un aspecto y comportamiento preferidos. Si es así, intentan utilizarlo. Si no es así, o el usuario a elegido uno no válido, entonces el UI elige el aspecto y comportamiento Java.

Cómo seleccionar el Aspecto y Comportamiento

Para especificar programáticamente el aspecto y comportamiento, se usa el método **UIManager.setLookAndFeel**. Por ejemplo, el código en negrita del siguiente fragmento hace que el programa use el aspecto y comportamiento Java.

```
public static void main(String[] args) {
    try {
        UIManager.setLookAndFeel(
            UIManager.getCrossPlatformLookAndFeelClassName());
    } catch (Exception e) { }

    new SwingApplication(); //Create and show the GUI.
}
```

El argumento para **setLookAndFeel** es el nombre totalmente cualificado de la subclase apropiado de **LookAndFeel**. Para especificar el aspecto y comportamiento Java, hemos usado el método **getCrossPlatformLookAndFeelClassName**. Si queremos especificar el aspecto y comportamiento nativo para cualquier platadorma en la que el usuario ejecute el programa, usaremos **getSystemLookAndFeelClassName**, en su lugar. PAra especificar un UI, podemos usar el nombre real de la clase. Por ejemplo, si hemos diseñado un programa para que parezca mejor con el aspecto y comportmiento Windows, deberíamos usar este código para seleccionarlo.

```
UIManager.setLookAndFeel(
                "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
```

Aquí podemos ver algunos de los argumentos que podemos usar con **setLookAndFeel**.

UIManager.getCrossPlatformLookAndFeelClassName()
: Devuelve el string para uno de los aspecto-y-comportamiento garantizados que funciona -- el aspecto y comportamiento Java.

UIManager.getSystemLookAndFeelClassName()
: Especifica el aspecto y comportamiento de la plataforma actual. En plataformas Win32, esto especifica el aspecto y comportamiento Windows. En plataforma Mac OS , esto especifica el aspecto y comportamiento Mac OS. En plataformas Sun, especifica el aspecto y comportamiento CDE/Motif.

"javax.swing.plaf.metal.MetalLookAndFeel"

Especifica el aspecto y comportamiento Java. (El nombre código para este aspecto y comportamiento era Metal.) Este string es el valor devuelto por el método **getCrossPlatformLookAndFeelClassName**.

"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
Especifica el aspecto y comportamiento Windows. Actualmente, sólo podemos usar este aspecto y comportamiento en systemas Win32.

"com.sun.java.swing.plaf.motif.MotifLookAndFeel"
Especifica el aspecto y comportamiento CDE/Motif. Este puede ser utilizado sobre cualquier plataforma.

"javax.swing.plaf.mac.MacLookAndFeel"
Especifica el aspecto y comportamiento Mac OS. En el momento de crear este tutor, este aspecto y comportamiento estába en fase bera y no está disponible como parte del JFC 1.1 o JDK 1.2. en su lugar, puedes descargarlo, siguiendo las instrucciones de la JFC Home Page.

No estamos limitados a los argumentos precedentes. Podemos especificar el nombre de cualquier aspecto y comportamiento que esté en nuestro class path.

Cómo elige el UI el Aspecto y Comportamiento

Aquí están los pasos que sigue el controlador del UI para determinar el aspecto y comportamiento cuando se inicializa a sí mismo.

1. Si el programa selecciona el aspecto y comportamiento antes de crear ningún componente, el UI trata de crear un ejemplar de la clase especificada. Si tiene éxito, todos los componentes usarán ese aspecto y comportamiento.
2. Si el programa no ha tenido éxito con el aspecto y comportamiento especificado, antes de crear el primer componente, el UI comprueba si el usuario ha especificado uno en un fichero llamado **swing.properties**. Busca el fichero en el directorio **lib** de la versión de Java. Por ejemplo, si estamos usando el intérprete Java en **javaHomeDirectory\bin**, entonces el fichero **swing.properties** (si existe) está en **javaHomeDirectory\lib**. Si el usuario ha especificado un aspecto y comportamiento, de nuevo el UI intenta ejemplarizar la clase especificada. Aquó hay un ejemplo de los contenidos de un fichero **swing.properties**.

```
3.      # Swing properties
4.
5.      swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

6. Si ni el programa ni el usuario han especificado un aspecto y comportamiento adecuados, el programa usar el aspecto y comportamiento Java.

Cambiar el Aspecto y Comportamiento después de la Arrancada

Se puede cambiar el aspecto y comportamiento con **setLookAndFeel** incluso después de que el GUI del programa sea visible. Para hacer que los componentes existente, reflejen el nuevo aspecto y comportamiento, se llama al método **SwingUtilities updateComponentTreeUI** una vez por cada contenedor de alto nivel. Luego podríamos desear redimensionar cada uno de nuestros contenedores de alto nivel para reflejar los nuevos tamaños de sus componentes. Por ejemplo.

```
UIManager.setLookAndFeel(lnfName);
SwingUtilities.updateComponentTreeUI(frame);
frame.pack();
```

¿Cómo usar Threads?

Esta página ofrece algunos ejemplos de uso de threads relacionados con el API de Swing. Para información concpetual, puedes ver Threads y Swing.

Usar el método invokeLater

Podemos llamar al método **invokeLater** desde cualquier thread para pedir que el thread de despacho de eventos ejecute cierto código. Debemos poner este código en el método **run** de un objeto **Runnable** y especificar ese objeto **Runnable** como el argumento de **invokeLater**. El método **invokeLater** retorna inmediatamente, sin esperar a que el thread de despacho de eventos ejecute el código. Aquí hay un ejemplo de uso de **invokeLater**>

```
Runnable doWorkRunnable = new Runnable() {
    public void run() { doWork(); }
};
SwingUtilities.invokeLater(doWorkRunnable);
```

Usar el método invokeAndWait

El método **invokeAndWait** es exactatamente igual que **invokeLater**, excepto en que no retorna hasta que el thread de despacho de eventos haya ejecutado el código especificado. Siempre que sea posible debemos usar **invokeLater** en vez de **invokeAndWait**. Si usamos **invokeAndWait**, debemos asegurarnos de que el thread que llama a **invokeAndWait** no contiene ningún bloqueo que otros threads podrían necesitar mientras ocurra la llamada. Aquí hay un ejemplo de uso de **invokeAndWait**.

```
void showHelloThereDialog() throws Exception {
    Runnable showModalDialog = new Runnable() {
        public void run() {
            JOptionPane.showMessageDialog(myMainFrame,
                                          "Hello There");
        }
    };
    SwingUtilities.invokeAndWait(showModalDialog);
}
```

De forma similar, un thread que necesite acceso al estado del GUI, como el contenido de un par de campos de texto, podría tener el siguiente código.

```
void printTextField() throws Exception {
    final String[] myStrings = new String[2];

    Runnable getTextFieldText = new Runnable() {
        public void run() {
            myStrings[0] = textField0.getText();
            myStrings[1] = textField1.getText();
        }
    };
    SwingUtilities.invokeAndWait(getTextFieldText);

    System.out.println(myStrings[0] + " " + myStrings[1]);
}
```

Cómo Crear Threads

Si podemos evitarlo, no debemos usar threads. Los threads pueden ser difíciles de usar, y hacen los programas muy duros de depurar. En general, no son necesarior para el trabajo estricto del GUIm como actualizar las propiedades de un componente.

Sin embargo, algunas veces, los threads **son** necesarios. Aquí tenemos algunas situaciones típicas en las que se usan threads.

- Para realizar tareas que llevan mucho tiempo sin bloquear el thread de despacho de eventos (ejemplos de esto pueden ser los cálculos intensivos, o las tareas de inicialización).
- Para realizar una operación de forma repetida, normalmente con periodo de tiempo dererminado entre operaciones.
- Para esperar mensajes de clientes.

Podemos usar dos clases para ayudarnos a implementar threads.

- **SwingWorker**: Crea un thread en segundo plado que ejecuta operaciones que consumen mucho tiempo.
- **Timer**: Crea un thread que ejecuta algún código una o más veces, con un retardo especificado por el usuario entre cada ejecución. Para más información sobre los timers, puedes ver Cómo usar Timers.

Usar la clase SwingWorker

La clase **SwingWorker** está implementada en **SwingWorker.java**, que no está en la versión Swing. **SwingWorker** hace todo el trabajo sucio de implementar un thread en segundo plano. Aunque muchos programas no necesitan este tipo de threads, son muy útiles para realizar tareas que consumen mucho tiempo, y pueden aumentar el rendimiento percibido del programa.

Para usar la clase **SwingWorker**, primero debemos crear una subclase de ella. En la subclase, debemos implementar el método **construct** para que contenga el código que realiza la operación. cuando ejemplarizemos nuestra subclase de **SwingWorker**, **SwingWorker** crea un thread que llama a nuestro método **construct**. Cuando necesitemos el objeto devuelto por el método **construct**, llamaremos al método **get** de **SwingWorker**. Aquí tenemos un ejemplo de uso de **SwingWorker**.

```
...//in the main method:
    final SwingWorker worker = new SwingWorker() {
        public Object construct() {
            return new ExpensiveDialogComponent();
        }
    };

...//in an action event handler:
    JOptionPane.showMessageDialog
        (f, worker.get());
```

Cuando el método **main** del programa crea el objeto **SwingWorker**, inmediatamente se arranca un nuevo thread que ejemplariza **ExpensiveDialogComponent**. El método **main** también construye un GUI que consiste en una ventana con un botón.

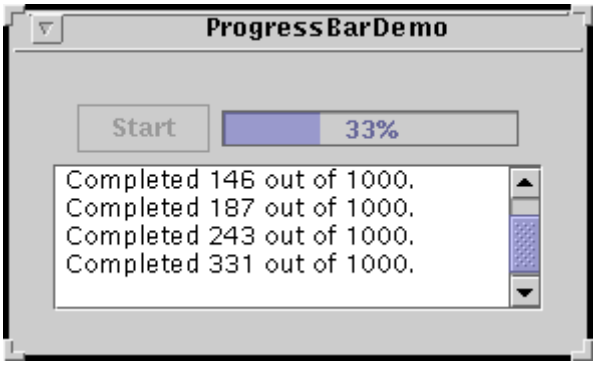
Cuando el usuario pulsa el botón, el programa se bloquea, si es necesario, hasta que se haya crreado **ExpensiveDialogComponent**. Entonces el programa muestra el dialo modal que contiene **ExpensiveDialogComponent**. Puedes encontrar el programa completo en **PasswordDemo.java**. También, el programa de ejemplo propocionado en Cómo Monitorizar el Progreso ejecuta una larga tarea en un theead **SwingWorker**.

¿Cómo usar Timer?

La clase **Timer** dispara uno o más **ActionEvent** después de un retardo especificado. Los temporizadores son útiles en las siguientes situaciones.

1. Hacer algo después de un retardo. Por ejemplo, muchos componentes Swing, como los botones, usan un temporizador para determinar cuando mostrar un tool-tip.
2. Mostrar progresos periódicamente. El prmer ejemplo que siguie, **ProgressBarDemo**, hace esto.
3. Realizar animaciones. Puedes ver Usar un Temporizador para Realizar Animaciones más adelante en esta página.

Veamos un ejemplo de la situación número 2. Aquí hay una imagen de una pequeña aplicación de ejemplo que usa un **Timer** y una barra de progreso para mostrar el progreso de una tarea de larga duración.



Prueba esto:

1. Compila y ejecuta la aplicación. El fichero principal es **ProgressBarDemo.java**.
2. Pulsa el botón **Start**. Mira la barra de progreso mientras la tarea se completa.

Una vez que la tarea ha empezado, el temporizador hace que la barra de progreso se actualice cada segundo hasta que la tarea se haya completado. Aquí está código de **ProgressBarDemo.java** que crea el temporizador y, cuando el usario pulsa el botón **Start**, lo arranca.

```
timer = new Timer(ONE_SECOND, new TimerListener());
. . .
timer.start();
```

Abajo está el código que implementa el oyente de action que es notificado cada vez que el temporizador va a cero.

```
class TimerListener implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
        progressBar.setValue(task.getCurrent());
        if (task.done()) {
                    Toolkit.getDefaultToolkit().beep();
                    timer.stop();
                    startButton.setEnabled(true);
        }
    }
}
```

La línea en negrita del código para el temporizador cuando la tarea se ha completado.

Nota: El método **actionPerformed** definido en el oyente de action del **Timer** es llamado en el thread de despecho de eventos. Esto significa que nunca tenemos que usar el método **invokeLater** sobre él. Para más información sobre el uso de componentes Swing y threads en el mismo programa, puedes ver Threads y Swing.

Usar un Timer para Realizar Animaciones

Aquí hay un ejemplo de uso de un **Timer** para implementar un bucle de animación.

```
public class AnimatorApplicationTimer extends JFrame
                                    implements ActionListener {
    ...//where instance variables are declared:
    Timer timer;

    public AnimatorApplicationTimer(...) {
        ...
        // Set up a timer that calls this
        // object's action handler.
        timer = new Timer(delay, this);
        timer.setInitialDelay(0);
        timer.setCoalesce(true);
        ...
    }

    public void startAnimation() {
        if (frozen) {
            // Do nothing.  The user has
            // requested that we stop
            // changing the image.
        } else {
            //Start (or restart) animating!
            timer.start();
        }
    }

    public void stopAnimation() {
        //Stop the animating thread.
        timer.stop();
```

```
    }

    public void actionPerformed (ActionEvent e) {
        //Advance the animation frame.
        frameNumber++;

        //Display it.
        repaint();
    }
    ...
}
```

Puedes encontrar el programa completo en **AnimatorApplicationTimer.java**.

El API de Timer

Las siguiente tablas listan los métodos y constructores más usado de **Timer**. El API para usar temporizadores de divide en tres categorías.

Ajuste fino de la Operación del Timer

Método o Constructor	Propósito
Timer(int, ActionListener)	Crea un timer inicializado con un retardo y un oyente. Este es el único constructor de **Timer**.
void setDelay(int) **int getDelay()**	Selecciona u obtiene el retardo entre disparos.
void setInitialDelay(int) **int getInitialDelay()**	Selecciona u obtiene el retardo para el disparo inicial.
void setRepeats(boolean) **boolean isRepeats()**	Selecciona u obtiene si el timer se repite.
void setCoalesce(boolean) **boolean isCoalesce()**	Selecciona u obtiene su el timer junta varios disparos pendientes en un único disparo.

Ejecutar el Timer

Método	Propósito
void start() **void restart()**	Activa el timer. **restart** cancela cualquier disparo pendiente.
void stop()	Desactiva el timer.
boolean isRunning()	Obtiene si el timer se está ejecutando.

Escuchar el Disparo del Timer

Método	Propósito
void addActionListener(ActionListener) **void removeActionListener(ActionListener)**	Añade o elimina el oyente de action.

¿Por qué Convertir a Swing?

La razón más fuerte es que Swing ofrece muchos beneficios a los programadores y a los usuarios finales. Entre ellos.

- El rico conjunto de componentes listo-para-usar significa que podemos añadir caracteristicas divertidas a nuestros programas -- botones con imágenes, barras de herramientas, panles con pestañas, display HTML, imagenes en ítems de menú, un selector de color, etc.
- También significa que prodríamos reemplazar algunos componentes personalizados con componentes Swing más extensibles y eficaces.
- Tener modelos separados de datos y estados hace que los componentes Swing sean altamente personalizables y permite compartir datos entres componentes.

- La arquitectura conctable del aspecto y comportamiento swing ofrece una aplica selección de aspectos y comportamientos. Junto al aspecto y comportamiento de la plataforma usual, podemos usar el aspecto y comportamiento Java e incluso aspectos y comportamientos de terceras partes.
- Los componentes Swing tienen soporte interno para accesibilidad, lo que hace que nuestros programas pueden usarse automáticamente con tecnologías asistivas.
- Los componentes Swing continuarán ampliándose en el futuro.

Por lo que la pregunta sería ahora "¿Por qué **no** debo convertir a Swing?"

Es razonable posponer la conversión si pensamos que nuestros usuarios no podrán ejecutar los programas Swing de forma conveniente. Por ejemplo, su nuestro programa es un applet y queremos que todo el mundo pueda usarlo en internet, deberíamos considerar cuandos navegantes del Web tienen navegadores que puedan ejecutar programas Swing. En el momento de escribir esto, la mayoría de los navegadores no tienen soporte Swing interno; los usuarios deben añadirlo descargando e instalando Java Plug-in.

Tienes la opción de actualizarte a Java 2 (JDK 1.2) cuando conviertas a Swing. Sin embargo, no necesitas decidirlo ahora. Los programas escritos en JDK 1.1 y Swing funcionan muy bien en Java 2.

¿Cómo Convertir a Swing?

El primer foco cuando se convierte un programa basado en el AWT 1.1 a Swing es modificar el programa para que use los componentes Swing en vez de los componentes AWT. Esta página explica los pasos para hacer esto. Cada paso se aplica a todos los programas -- aplicaciones y applets -- a menos que se especifique lo contrario.

Paso 1: Guardad una copia del programas basado en el AWT.

Necesitares una copia de todos los ficheros del programa, incluyendo los ficheros **.java** y **.class**. Necesitamos estas copias por varias razones.

- Cualquier usuario que no pueda ejecutar programas Swing necesitará ejecutar el viejo programa AWT.
- Necesitaremos referirinos al código fuente durante el proceso de conversión.
- Después de la conversión, deberemos ejecutar las dos versiones del programa para compararlos.
- Después de la conversión, podemos comparar el código fuente de ambas versiones para aplicar lo que hemos aprendido en otros programas que necesitemos convertir.

Paso 2: Eliminar cualquier sentencia java.awt.

Este paso pone el compilador a trabajar por nosotros. Es útil eliminar todas las importaciones del AWT incluso si nuestro programa todavía utiliza clases AWT -- como sus controladores de distribución -- porque, sin estas sentencias, el compilador generará errores "not found" por cada componente AWT usado en nuestro programa y el número de línea donde es utilizado. Esto ayuda a localizar cada uno de los componentes AWT usado por nuestro programa para que podamos reemplazarlos por su equivalente Swing en el Paso 8, luego añadiremos las importaciones para las clases AWT que realmente necesitemos.

Paso 3: Si nuestro programa es un applet, eliminar cualquier sentencia java.applet

No necesitamos la vieja clase **Applet** ni el paquete **java.applet**, porque nuestros applets Swing será una subclase de la clase **JApplet** de Swing. Por eso debemos eliminar cualquiera de estas sentencias de nuestro programa.

```
import java.applet.Applet;
    o
import java.applet.*;
```

Paso 4: Importar el paquete principal Swing.

Añadir la siguiente sentencia import a nuestro programa.

```
import javax.swing.*;
```

Esto importa todos los componentes Swing además de algunas otras clases Swing. Si queremos podemos ser más meticulosos y añadir una sentencia import por cada clase Swing que utilicemos.

Paso 5: Cuidado con el problemas con los Threads!

Antes de continuar, recuerda este echo importante: **Aunque el AWT es seguro ante los Threads, Swing no lo es.** Debemos tener esto en consideración cuando convirtamos nuestros programas.

Para la mayoría de los programadores esto significa que un programa modifica los componentes Swing sólo desde dentro del thread de eventos del AWT. Los programas típicos modifican componentes desde dentro de los métodos de manejo de eventos y del método paint, que son llamados por el thread de eventos del AWT, por eso modificar un componente en esto métodos es seguro. Si nuestro programa modifica un componente en cualquier otro lugar, debemos tomar las acciones explícitas para hacerlo seguro ante los threads.

Hemos proporcionado dos páginas sobre Swing y los threads. Primero Threads y Swing proporciona una cobertura conceptual. Luego Cómo usar Threads contiene información práctica y ejemplos.

Paso 6: Cambiar cada componente AWT por su equivalente Swing más cercano.

La tabla proporcionada en la sección de recursos, Reemplazos Swing para Componentes AWT, lista cada uno de los componentes AWT y su equivalente Swing más cercano. Podemos usarla como una guía para elegir el reemplazo para cada componente Swing usado en nuestro programa.

En el mejor de los casos, el componente AWT y su reemplazo Swing son compatibles en el código y un simple cambio de nombre es todo lo que requiere. Por ejemplo, para convertir un botón del AWT por un botón Swing, sólo debemos cambiar todas las ocurrencias de **Button** por **JButton** en nuestro programa. Aquí tenemos un pequeño ejemplo de código AWT y su equivalente Swing. El código subrrayado muestra las diferencias entre los dos programas.

Código AWT	Código Swing
`Button button = new Button("A Button");` `button.addActionListener(this);`	`JButton button = new JButton("A Button");` `button.addActionListener(this);`

Estaras contento de aprender que un gran número de componentes Swing tiene el código compatible con su pareja AWT.

En el lado contrario tenemos que no todos los componentes Swing son totalmente compatibles con los componentes AWT. Por eso, para algunos componentes AWT tendremos que re-escribir algún código cuando lo reemplazemos por un componente Swing. Además, podríamos elegir hacer cambios innecesarios para aprovecharnos de las características Swing. Por ejemplo, podríamos querer añadir una imagen a un botón, podríamos querer soportar la accesibilidad llamando al método **setLabelFor** sobre las etiquetas asociadas con otros componentes. Recursos de Conversión tiene más información para ayudarnos con las conversiones que requieren algo más que un simple cambio de nombre y sobre las conversiones opcionales.

Paso 7: Cambiar todas las llamadas a los métodos add y setLayout.

Los programas AWT añaden componentes y seleccionan el controlador de distribución directamente sobre el contenedor de alto nivel (un frame, dialog, o applet). En contraste, los programas Swing añaden componentes y seleccionan el controlador de distribución sobre el **panel de contenido** del contenedor de alto nivel. La primera fila de la siguiente tabla muestra algún código típico AWT para añadir componentes a un frame y seleccionar su controlador de distribución. La segunda y tercera filas muestran dos diferentes conversiones Swing.

Código AWT	`frame.setLayout(new FlowLayout());`
	`frame.add(button);`
	`frame.add(label);`
	`frame.add(textField);`

Conversión Óbvia Swing (No hagas esto)	`frame.getContentpane().setLayout(new FlowLayout());`
	`frame.getContentPane().add(button);`
	`frame.getContentPane().add(label);`
	`frame.getContentPane().add(textField);`

Conversión Eficiente Swing (Haz esto)	`Container contentPane = frame.getContentPane();`
	`contentPane.setLayout(new FlowLayout());`
	`contentPane.add(button);`
	`contentPane.add(label);`
	`contentPane.add(textField);`

Habrás notado que el código Swing de la segunda fila llama a **getContentPane** múltiples veces, lo que es ineficiente si nuestro programa usa muchos componentes. El código Swing de la tercera fila mejora el código, obteniendo el panel de contenido una sóla vez, almacenándolo en una variable, y usando la variable para añadir componentes y seleccionar el controlador de distribución.

Paso 8: Usar el compilador para indicar más cambios necesarios.

Una vez modificado el código fuente como se ha indicado en los pasos anteriores, utilizaremos el compilador para **intentar** compilar nuestro programa.

```
javac MyClass.java
```

Durante este paso, el compilador nos está ayudando a identificar cualquier conversión que nos haya pasado por alto. No podemos esperar que el programa se compile a la primera.

El compilador puede ayudarnos.

- Encontrando cada componente AWT que nos hayamos olvidado de convertir a su equivalente Swing. Si seguimos el paso 2 el compilador mostrará un mensaje de error como este por cada componente AWT que quede en nuestro programa.
  ```
  TextEventDemo.java:23: Class Button not found in type declaration.
                  Button button = new Button("Clear");
                  ^
  ```
- Identificar que clases AWT todavía necesita nuestro programa. Si hemos seguido las instrucciones del paso 2 el compilador mostrará un mensaje de error como este por cada clase AWT que todavía necesitesmos
  ```
  TextEventDemo.java:17: Class BorderLayout not found in type declaration.
                  BorderLayout layout = new BorderLayout();
                  ^
  ```

Las clases AWT que podríamos necesitar en un programa Swing son los controladores de distribución, **Color**, **Dimension**, **Point**, **Insets**, etc.

- Localiza cualquier incompatibilidad de código entre los componentes AWT y si reemplazo Swing. Esto se muestra como un error del compilador sobre métodos no definidos. Por ejemplo, aunque los componentes de texto del AWT tienen un método **addTextListener**, los componentes de Swing no lo tienen. El compilador genera un mensaje de error como este indicando que **JTextField** no tiene dicho método.

```
TextEventDemo.java:22: Method addTextListener(TextEventDemo.
            MyTextListener) not found in class javax.swing.JTextField.
               textField.addTextListener(new MyTextListener("Text Field"));
                                         ^
```

- Eliminar el uso de API "obsoleto". Usamos la bandera del compilador **-deprecation** para obtener detalles sobre las clases o métodos obsoletos usados por nuestro programa y el número de línea donde son utilizados.

```
javac -deprecation MyClass.java
```

Debemos corregir todos los errores devueltos por el compilador hasta que el programa se compile correctamente.

Paso 9: Ejecutar el programa Swing.

Usamoos el intérprete o el Applet Viewer para ejecutar nuestro programa.

```
java MyClass      o     appletviewer MyApplet.html
```

Si hemos olvidado modificar cualquier llamada a **add** o **setLayout**, lo descubriremos durante este paso. Si lo hemos hecho, el sistema runtime muestra un mensaje como este (el mensaje de error para **setLayout** es similar).

```
java.lang.Error: Do not use MultiListener.add() use
                     MultiListener.getContentPane().add() instead
               at javax.swing.JApplet.createRootPaneException(Compiled Code)
               at javax.swing.JApplet.addImpl(Compiled Code)
               at java.awt.Container.add(Compiled Code)
               at MultiListener.init(Compiled Code)
               at sun.applet.AppletPanel.run(Compiled Code)
               at java.lang.Thread.run(Compiled Code)
```

Volvemos sobre el código, buscaremos cualquier **add** o **setLayout**, resolveremos el problema. compilaremos y ejecutaremos el programa de nuevo.

Paso 10: Comparar las versiones Swing y AWT.

Probablemente querramos que los programas sean similares, estén abiertos a las mejoras ofrecidas por swing y cualquier diferencia inherente en los dos GUIs.

Paso 11: Investigar otros componentes Swing.

Podríamos mejorar nuestro UI usando características sólo disponibles en componentes Swing (como las imágenes en botones) o usando componentes Swing más sofisticados no disponibles en el AWT. También podríamos reemplazar un componente escrito por nosotros mismos con un componente del propio Swing o un componente personalizado Swing. Los componentes completamente nuevos en Swing incluyen tablas, trees, color choosers, etc.

Paso 12: Limpieza!

Ahora es el momento de limpiar nuestro código. Si hemos dejado algún código para corregir alguna deficiencia o bug del AWT , es el momento de limpiarlo!

Recursos de Conversión

Te proporcionamos estos recursos para ayudarte a convertir tus programas a Swing.

¿En qué se diferencias los Componentes Swing de los del AWT?
Si no has leído esta página, considera leerla ahora. Proporciona una base útil para la perspectiva del proceso de conversión.

Reemplazos Swing para Componentes AWT
Lista cada uno de los componentes AWT y su reemplazo Swing, y contiene notas que pueden ayudarnos con los cambios más sencillos que tendríamos que hacer en nuestro código. Además, la tabla apunta cuando podríamos querer actualizar a un componente Swing más potente. Por ejemplo, podríamos querer reemplazar un componente de texto AWT con un campo password de Swing en vex de un campo de texto normal.

Trucos de Conversión General
Proporciona información sobre los problemas de conversión generales como la seguridad con los threads, dibujo, etc.

Trucos de Conversión Específicos de Componentes
Proporciona detalles sobre la conversión específica de algunos componentes, como las áreas de texto, que tienden a requerir más trabajo para convertirla. Proporciona pequeños ejemplos donde sea apropiado.

Algunos ejemplos de Conversión
Muestra cómo hemos convertido algunos ejemplos del tutorial. Cada ejemplo es un programa completo -- una aplicación o un applet.

Resolver Problemas Comunes con la Conversión
Describe algunos de los problemas más comunes que nos podríamos encontrar durante la conversión y cómo resolverlos.

Respuestos Swing para Componentes AWT

Usa la siguiente tabla como una guía para elegir un reemplazo Swing para cada uno de los componentes AWT usados en tu programa.

Componente AWT	Equivalente Swing más cercano	Notas
java.applet.Applet	**JApplet**	Los applets AWt y los applets Swing difirien en varias cosas. Puedes ver Convertir Applets.
Button	**JButton**	Un button Swing puede incluir una imagen y/o texto.
Canvas	**JPanel**, **JLabel**, o otro componente Swing apropiado	Nuestra elección depende de para qué utilice el programa el canvas. Puedes ver Convertir Canvas para una explicación de las opciones de conversión
Checkbox	**JCheckBox** o **JRadioButton**	Observa que la 'B' está en mayúsculas en el nombre de la clase Swing y no en el nombre de la clase AWT.
CheckboxMenuItem	**JCheckBoxMenuItem**	Observa que la 'B' está en mayúsculas en el nombre de la clase Swing y no en el nombre de la clase AWT. También observa que los componentes de menús Swing son componentes verdaderos.
Choice	**JComboBox**	Se rellenan de forma diferente un **JComboBox** que un **Choice**. Puedes ver Convertir Choices para más detalles y un ejemplo
Dialog	**JDialog** o **JOptionPane**	Los programas AWT añaden componentes directamente al diálogo y seleccionan directamente el controlador de distribución. En contraste, los programas Swing añaden componente y seleccionan el controlador de distribución sobre el panel de contenidos del **JDialog**.
FileDialog	**JFileChooser**	
Frame	**JFrame**	Los programas AWT añaden componentes directamente al frame y seleccionan directamente el controlador de distribución. En contraste, los programas Swing añaden componente y seleccionan el controlador de distribución sobre el panel de contenidos del **JFrame**.
Label	**JLabel**	Una etiqueta Swing puede incluir una imagen y/o texto. Para soportar accesibilidad, se usa **setLabelFor** para asociar cada etiqueta con el componente al que describe.
List	**JList**	Se rellenan de forma diferente una lista Swing a una lista AWT. Además, normalmente necesitaremos poner una lista Swing en un ScrollPane, mientras que las listas AWT soporta el scrolado directamente. Puedes ver Convertir Lists para información y ejemplos.
Menu	**JMenu**	Los componentes de menús de Swing son componentes verdaderos.
MenuBar	**JMenuBar**	Los componentes de menús de Swing son componentes verdaderos.
MenuItem	**JMenuItem**	Los componentes de menús de Swing son componentes verdaderos.
Panel	**JPanel**	
PopupMenu	**JPopupMenu**	Los componentes de menús de Swing son componentes verdaderos.
ScrollBar	**JScrollPane** o **JSlider** o **JProgressBar**	
ScrollPane	**JScrollPane**	
TextArea	**JTextArea**	Requieren re-escribir algún código para su conversión. Puedes ver Convertir componentes de Texto para más información y ejemplos
TextField	**JTextField**	Para usos sencillos, **JTextField** tiene el código compatible con **TextField**. Si usamos **TextListener** necesitamos modificar nuestro código para usar un tipo distinto de oyente. Si necesitamos un campo de password, usaremos **JPasswordField** en su lugar. Puedes ver Convertir componentes de Texto para más información sobre conversiones no triviales y ejemplos.

Window	**JWindow** o **JToolTip**	

Trucos de Conversión a Swing

Espacio vacío

Al menos con el aspecto y comportamiento por defecto (Java) los componentes Swing tienden a ser más anchos (esto esm contienen menos espacio vacío), Para obtener un aspecto más esparcido. podríamos tener que añadir bordes vacíos a algunos componentes o márgenes a los componentes de texto.

Convertir Código de Dibujo

Habla sobre cómo se debería mover el código de dibujo a paintComponent (y definitivament no a update), y el doble buffer es automático ahora.

Si nuestro código de dibujo crea un título o laterales alrededor del componente, debemos considerar reemplazarlo con un border. Por ejemplo, podemos crear fácilmente una caja alrededor de un grupo de componentes añadiendo los componentes a un **JPanel** y haciendo que el panel tenga un borde.

Si hemos implementadopaintComponent entonces debemos llamar primero a **super.paintComponent**. Si no es así, incluso si el componente es opaco no dibujará automáticamente su fondo.

No debemos dibujar directamente en un **JApplet** porque será cubierto por el panel de contenido automático. En su lugar debemos usar una clase separada para hacer el dibujado y añadirlo al panel de contenidos del applet.

Trucos de Conversión a Específicos de Componentes

Convertir Applets

Cómo se mencióno en el Paso 7: Cambiar las llamadas a los método **add** y **setLayout**, los programas AWT añaden componentes directamente al objeto **Applet** y seleccionan directamente el controlador de distribución del applet. Por otro lado, los applets Swing, añaden componentens y seleccionan el controlador de distribución sobre el panel de contenidos del **JApplet**. Por eso, para convertir applets, debemos hacer que el código fuente cambie lo que se describió en esa sewcción.

Además, mientras que **FlowLayout** es el controlador de distribución por defecto para los applets AWT, **BorderLayout** los es para los applets Swing. Esto tiene dos repercusiones.

1. Si queremos usarun **FlowLayout**, la versión Swing de nuestro programa debe usar **setLayout** sobre el panel de contenido para especificarlo.
2. Si específicamos **BorderLayout** en nuestro applet AWT, podemos eliminar la sentencia **setLayout** de la versión Swing.

No debemos dibujar directamente sobre un **JApplet** (realmente ni sobre cualquier contenedor de alto nivel) porque será cubierto por el panel de contenido del applet. La solución es tener un componente personalizado para hacer el dibujo y añadirlo al panel de contenido. Puedes ver las instrucciones para convertir canvas, en ella encontrarás trucos para elegir una clase de componente personalizado, y cómo mover el código de dibujo al método apropiado.

Convertir Canvas (Componentes Personalizados)

Antes de convertir un componente personalizado, debemos chequear si se puede utilizar un componente estandard de Swing. Por ejemplo, si nuestro componente personalizado simplemente muestra una imagen, quizás algún texto, podríamos usar una Etiqueta Swing. Si nuestro componente personalizado implementa un botón con una imagen, podemos usar un Botón Swing en su lugar. Si hemos implementado un árbol o una tabla, podremos usar un Tree Swing o una tabla Swing.

Si ningún componente Swing tiene la funcionalidad que necesitamos, recomendamos cambiar la superclase de nuestro componente personalizado de **Canvas** a **JPanel**. Luego debemos mover el código de dibujo desde **paint** o **update** a un método llamaod **paintComponent**. En la parte superior de este método, deberíamos insertar **super.paintComponent(g)**. Podremos eliminar todo el código relacionado con el doble buffer, ya que Swing lo proporciona de forma automática.

Otras superclases, como **Component** **JComponent** también podrían ser factibles. Extender **JComponent** en lugar de **Component**/**Container** nos ofrecerá el doble buffer automáticamente. Extender **JPanel** en vez de **JComponent** ofrece dos cosas: dibujo automático del fondo (lo que podemos desactivar usando **setOpaque(false))** además de la habilidad de usar bordes para dibujar los laterales de los componentes.

Convertir Choices

Los programas AWT rellenan un objeto **Choice** usando el método **addItem**. Los programas Swing rellenan un **JComboBox** creando un **Vector**, un array, o un modelo de datos con los ítems iniciales. Aquí hay un ejemplo.

Código AWT	Código Swing

`String choiceprefix = "Choice item #";`
 	
`Choice choice = new Choice();`
`for (int i = 0; i < 10; i++) {`
`  choice.addItem(choiceprefix + i);`
`}`
	`String choiceprefix = "Choice item #";`
	`final int numItems = 10;`
	`Vector vector = new Vector(numItems);`
	`for (int i = 0; i < numItems; i++) {`
	`  vector.addElement(choiceprefix + i);`
	`}`
	`JComboBox comboBox = new JComboBox(vector);`

Convertir Listas

Se necesita poner una lista en un scroll pane, mientrás que en el AWT se obtenía automáticamente. Cuando se rellena una lista, no se puede hacer de la misma forma que se hacía en el AWT. En su lugar, necesitamos crear un vector, un array o un modelo de datos con los ítems iniciales.

Convertir Componentes de Texto

Cuando se convierten áreas de texto AWT, normalmente tendremos que poner el área de texto Swing en un scrol pane y seleccionar el tamaño preferido del scroll pane. Es un poco más difícil imaginarse la relación entre los viejos argumentos de fila y columna y el tamaño preferido en pixels. Esto significa que probablemente dejemos fuera del constructor del área de texto los argumentos de filas y columnas. No debemos olvidarnos de cambiar las llamadas a add para añadir al scroll pane en vez de al área de texto.

Si estamos usando un GridBagLayout debemos asegurarnos de que aplicamos las restricciones al scroll pane y no al área de texto (de otro modo obtendremos un pequeño e inutil scroll pane, que realmente parece que funciona).

Los componentes de texto Swing no tienen un método addTextListener. Debemos convertir el TextEventDemo para suar oyentes de document en su lugar. Esto podría ser un ejemplo útil d euna conversión que no sea 1:1.

Si queremos añadir texto con estilos a nuestro programa, podemos convertir a **JEditorPane** o a **JTextPane** en su lugar.

Algunos Ejemplos de Conversión a Swing

Esta sección proporciona las versiones AWT y Swing de varios programas de ejemplo y habla sobre aspectos interesantes de la conversión.

Convertir ButtonDemoApplet

El primer ejemplo de conversión AWT a Swing es un simple applet que contiene tres botones. Si has leído las sección de los componentes AWT o Swing, el programa te será familiar.

Aquí tenemos dos imágenes. La primera muestra la versión AWT del applet ejecutándose y la segunda la versión Swing.

Versión AWT

Versión Swing

Habrás observado que los programas tienen un aspecto diferente.

1. Los tres botones tienen un borde diferente.
2. El Applet usa diferentes fuentes para el Texto.
3. Los botones en el applet Swing tienen iconos y texto.
4. Los botones en el applet Swing muestran el mnemónico de teclado.
5. El applet Swing es más grande.

Las primeras dos diferencias existen simplemente por que el AWT y el aspecto y comportamiento Java usado en el programa swing dibujan los botones de forma diferente y tienen diferente fuente por defecto. El aspecto y comportamiento Java es el aspecto y comportamiento que los programas Swing usan a menos que el usuario o el programador especifiquen uno diferente.

Las diferencias 3 y 4 son poeque cuando se convirtió el código nosotros elegimos aprovecharnos de la característica de los botones Swing no soportadas por los botones AWT: imágenes y mnemónicos.

La diferencia final es un efecto colateral de las otras cuatro. Si nuestro programa es una pplet, debemos recordad modificar la etiqueta <APPLET> para ajustar cualquier cambio de tamaño en nuestro programa.

Aunque el aspecto de los programas pueda ser diferente, el comportamiento es el mismo. Cuando el usuario pulsa el botón izquierdo, se desactiva el botón central y el propio botón izquierdo, y activa el botón derecho. Cuando el usuario pulsa el botón derecho, se desactiva a sí mismo y activa los botones central e izquierdo.

La siguiente tabla enlaze el código fuente completo y un fichero HTML que contiene una etiqueta <APPLET> de ejemplo, para cada versión del progra,a. Compara el código de la etiqueta <APPLET> para ver las diferencias entre los dos programas

	Código fuente	Etiqueta <APPLET>
AWT	ButtonDemoApplet.java	ButtonDemoApplet.html
Swing	ButtonDemoApplet.java left.gif middle.gif LEFT.gif	ButtonDemoApplet.html

Nota: Observa que si tu navegador no está configurado para ejecutar programas 1.1 o swing, visitar los ficheros **.html** listados en la tabla producirá errores. Nosotros proporcionamos los ficheros para que puedas ver las etiquetas de los applet. Puedes usar shift-click para descargar los ficheros

Convertir AnimatorApplication

AnimatorApplication es una plantilla para programas de animación. Esta implementación particular "anima" una cadena cambiándola periódicamente. El programa puede ser fácilmente modificado para animar imágenes de la misma manera.

Esta sección discute dos soluciones diferentes para convertir el programa de animación. La primera solución utiliza una aproximación minimalista -- el código se cambia sólo lo necesario para que el programa funcione con Swing. La segunda solución es más completa -- cambia el código para ajustar las diferencias de la forma en que dibujan los programas AWT y Swing y se aprovecha de las nuevas clases Swing.

La clase **AnimatorApplication** del AWT desciende de **Frame**. El método **paint** de frame usa **drawString** para dibujar una cadena en el frame. Un thread periódicamente se despierta, cambia la cadena al siguiente valor, y redibuja el frame.

Ambas versiones Swing de la clase **AnimatorApplication** descienden de **JFrame**.

La versión minimalista del programa dibuja de la misma dorma que los hacía la versión del AWT -- llamando al método **drawString**. Sin embargo, en Swing, el código de dibujo pertenece a un método llamado **paintComponent**. Además, como **JFrame** tiene un panel de contenido, el dibujo echo en su método **paintComponent** no tiene efecto (el panel de contenido se dibujará sobre el). Por eso el código de dibujo tiene que moverse fuera de **JFrame**. En su lugar, el programa define una subclase de **JPanel**, **AnimappPanel**, para hacer el dibujado, y sa un ejemplar de **AnimappPanel** como el panel de contenido del **JFrame**.

La segunda solución es más completa. En lugar de crear una subclase de **JPanel** para dibujar la cadena, esta solución usa un **JLabel**, que está especialmente diseñado para dibujar cadenas. Además, la segunda solución usa la clase **Timer** de Swing en lugar de usar un **Thread** que duerme periódicamente.

Aquí están las tres versiones diferentes para compararlas.

	Código fuente
AWT	AnimatorApplication.java
Versión Swing Minimalista	AnimatorApplication.java
Versión completa de Swing	AnimatorApplicationTimer.java

Problemas de Conversión a Swing

Problema: Estoy viendo problemas raros que parecen intermitentes o dependientes del tiempo.

- ¿El thread principal modifica el GUI después de que sea visible? Si es así, mueve el código para que se ejcute antes de que el GUI se muestre, o ejecuta el código que modifica el GUI en un thread de despacho de eventos.
- ¿Tienes tu programa varios threads o se modifica el GUI en respuesta a mensajes de otros programas? Si es así, deberías asegurarte de que todo el código relacionado con el GUI se ejecuta en un thread de despacho de eventos.
- Si tu programa es un applet que implementa los métodos **stop** y **start**, s asegurarte de que todo el código relacionado con el GUI se ejecuta en un thread de despacho de eventos.
- Las sugerencias anteriores asumen que tu problema está causado por código que no es seguro con los threads. Puedes ver Threads y Swing para ver información sobre seguridad de threads, y Cómo usar Threads para información sobre el API que puedes usar para conseguir que tus programas sean seguros ante los threads.

Problema: Mi applet/dialog/frame está negro.

- ¿El applet/frame/dialog realiza dibujo personalziado? Si es así, necesitar mover todo el código de dibujo fuera de las subclases **JApplet**/**JDialog**/**JFrame** y ponerlo dentro de un componente personalizado para añadirlo al panel de contenido. Para más información puedes ver el Paso 8 del plan de conversión al Swing.
- ¿Has seleccionado el panel de contenido del applet/frame/dialog o has añadido componentes al panel de contenido existente. Deberias haberlo hecho. Puedes ver Usar Contenedores de Alto Nivel.

Problema: En la versión Swing de mi programa, el componente list/text ha perdido sus barras de desplazamiento.

- Los componentes de listas y de texto Swing no tienen barras de desplazamiento automáticas. En su lugar, tienes que añadir la lista o el componente de texto a un ScrollPane, como se describe en Trucos de Conversión.

Problema: Aunque estoy usando el mismo código grid bag layout que antes, se ve un componente escrollable pequeño.

- Asegurate de seleccionar las obligaciones del ScrollPane, en vez de las de su cliente.

Problema: No estoy obteniendo los tipos de eventos que experaba para los componentes Swing que estoy usando.

- Leete las secciones de trucos de conversión y de cómo usar el componente que estás usando. Los detalles relevantes del manejo de eventos se cubre en esas secciones.