TEMA 10. DESARROLLO DE INTERFACES GRÁFICAS DE USUARIO

INDICE

INTERFACES GRÁFICAS Y HERRAMIENTAS DE DISEÑO. SWING

La mayoría de las aplicaciones que se desarrollan hoy en día incluyen un apartado gráfico compuesto por ventanas, botones, ...

Java proporciona la biblioteca gráfica de usuario **Swing** para facilitar el desarrollo de aplicaciones gráficas.

Podemos desarrollar aplicaciones gráficas usando directamente componente proporcionados por Swing o podemos usar complementos que nos faciliten la creación de interfaces gráficas.

CONFIGURACIÓN DE UN IDE PARA CREAR INTERFACES GRÁFICAS. ECLIPSE. WINDOW BUILDER

Eclipse facilita una serie de herramientas que simplifican la creación de aplicaciones gráficas.

Nosotros vamos a usar el Window Builder pero para poderlo utilizar primero lo debemos instalar, si no lo está, y configurar para que se adapte a nuestras necesidades.

Para ello, en primer vamos a https://eclipse.org/windowbuilder/download.php y hacemos clic en la opción correspondiente a nuestra versión.

Al hacerlo iremos a la página de instalación de la versión correspondiente a nuestra versión. En nuestro caso http://download.eclipse.org/windowbuilder/WB/release/R201406251200/4.4/

Seguimos los pasos que se nos indican y, si todo ha ido bien, ya tendremos el Window Builder instalado.

Si queremos modificar las Preferencias del Window Builder podemos consultar el manual de usuario que se encuentra en el siguiente enlace
http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.wb.doc.user%2Fhtml%2Fuserinterface%2Fsource_view.html

EJEMPLO DE WINDOW BUILDER

Para comprobar su correcto funcionamiento vamos a crear un nuevo **JPanel** de nombre **VentanaHola** que tiene un **Layout** de tipo (**absolute**), un **Title** con el valor "**Hola**" y muestra una etiqueta (**JLabel**) con el **text** "Hola Mundo." Centrada en medio del layout.

Para ello vamos a **File -> New -> Other** y seleccionamos **Window Builder -> Swing Designer -> JFrame**.

Escribimos **VentanaHola** en el campo **Name** y pulsamos **Finish**.

Por defecto se nos muestra la vista de código (pestaña **Source**). Para ir a la vista de diseño debemos hacer clic sobre la pestaña **Design** que se encuentra en la parte inferior de la ventana de código.

Si no nos aparecen las pestañas Source y Design en la parte inferior es porque no hemos abierto el fichero con el Window Builder. Para abrir el fichero con el Window Builder vamos a la ventana del Window Explorer, hacemos clic con el botón derecho sobre el fichero que queremos abrir con el Window Builder (ventanaHola.java en micaso) nos situamos sobre la opción **Open With** y seleccionamos **WindowBuilder Editor**. Al hacer esto nos parece en el Workspace una ventana con el código del fichero con las pestañas Source y Design en la parte inferior. Si nos fijamos en el **icono de esa ventana** veremos que en vez de una J dentro de una hoja en blanco aparece una J sobre una ventana.

Una vez en el modo diseño ya podemos añadir componentes al JFrame y modificar sus propiedades.

Aunque sea más sencillo desarrollar aplicaciones gráficas usando el modo diseño es conveniente revisar el código fuente generado ya que en muchas situaciones el asistente de código genera código que provoca problemas.

Una vez configurada la aplicación a nuestro gusto la ejecutamos para comprobar su correcto funcionamiento.

SWING. JFRAME. JDIALOG

La biblioteca de clases Java Swing nos brinda ciertas facilidades para la construcción de interfaces graficas de usuario. Antes de desarrollar una aplicación gráfica es importante conocer a nivel general algunos de los principales componentes que podemos usar.

Cuando vamos a construir aplicaciones gráficas utilizando Java Swing debemos tener al menos un contenedor que será la base para nuestra aplicación, es decir, será el lienzo donde pintaremos los demás componentes.

Normalmente podemos utilizar un componente JFrame o JDialog como contenedor. Este contenedor será la base para nuestra ventana y en su interior colocaremos los componentes gráficos de nuestra aplicación.

Un **JFrame** permite crear una ventana con ciertas características, por ejemplo podemos visualizarla en nuestra barra de tareas, caso contrario de los **JDialog**, ya que estos últimos son Ventanas de Dialogo con un comportamiento diferente, no se puede ver la ventana en la barra de tareas ni posee los botones comunes de maximizar o minimizar.

Los componentes JDialog pueden heredar de componentes JFrame o de otros componentes JDialog mientras que los componentes JFrame sólo pueden heredar de componentes JFrame.

Cuando trabajemos con aplicaciones gráficas es recomendable crear la **ventana principal** de la aplicación usando un **JFrame** y el **resto de ventanas** usando **JDialog** ya que de esa manera hacemos que las ventanas dependan de una única ventana principal.

Independientemente de si usamos JFrame o JDialog debemos usar un contenedor en el que iremos colocando los componentes (por ejemplo un **JPanel**).

Por ejemplo, si queremos crear un JFrame de nombre JFrameHola con title "JFrame Hola" escribimos

```
public class JFrameHola extends JFrame {
 private JPanel contentPane;

 // Creo el JFrame
 public JFrameHola() {
  setTitle("JFrame Hola");
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  setBounds(100, 100, 450, 300);
  contentPane = new JPanel();
  contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
  contentPane.setLayout(new BorderLayout(0, 0));
  setContentPane(contentPane);
 }

// Ejecuto la aplicacion
```

```
 public static void main(String[] args) {
  EventQueue.invokeLater(new Runnable() {
   public void run() {
    try {
     JFrameHola frame = new JFrameHola();
     frame.setVisible(true);
    } catch (Exception e) {
     e.printStackTrace();
    }
   }
  });
 }
}
```

Si en vez de crear un JFrame queremos crear un JDialog de nombre JDialogHola con title "JDialog Hola" escribimos

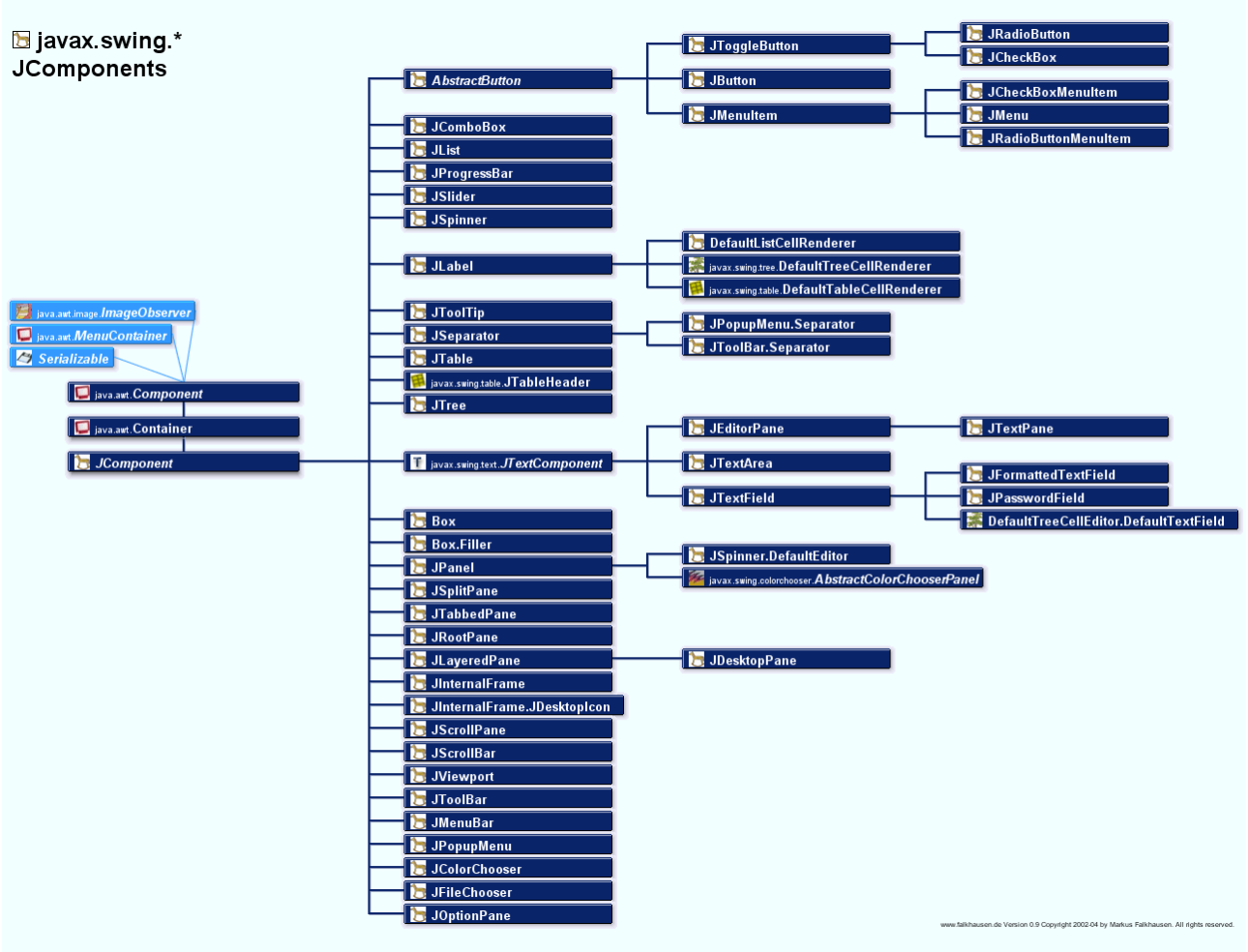
```
public class JDialogHola extends JDialog {
 private final JPanel contentPanel = new JPanel();

 // Creo el JDialog
 public JDialogHola() {
  setTitle("JDialog Hola");
  setBounds(100, 100, 450, 300);
  getContentPane().setLayout(new BorderLayout());
  contentPanel.setLayout(new FlowLayout());
  contentPanel.setBorder(new EmptyBorder(5, 5, 5, 5));
  getContentPane().add(contentPanel, BorderLayout.CENTER);
  {
   JPanel buttonPane = new JPanel();
   buttonPane.setLayout(new FlowLayout(FlowLayout.RIGHT));
   getContentPane().add(buttonPane, BorderLayout.SOUTH);
   {
    JButton okButton = new JButton("OK");
    okButton.setActionCommand("OK");
    buttonPane.add(okButton);
    getRootPane().setDefaultButton(okButton);
   }
   {
    JButton cancelButton = new JButton("Cancel");
    cancelButton.setActionCommand("Cancel");
    buttonPane.add(cancelButton);
   }
  }
 }

// Ejecuto la aplicacion
 public static void main(String[] args) {
  try {
   JDialogHola dialog = new JDialogHola();
   dialog.setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);
   dialog.setVisible(true);
```

```
 } catch (Exception e) {
  e.printStackTrace();
 }
 }
}
```

Si ejecutamos las aplicaciones JFrameHola y JDialogHola podemos observar sus diferencias, siendo la principal que al ejecutar la aplicación con el JFrame el JFrame aparece en la barra de tareas pero al ejecutar la aplicación con el JDialog el JDialog no aparece en la barra de tareas.

SWING. COMPONENTES

La biblioteca Java Swing proporciona diversos componentes gráficos que permiten realizar aplicaciones gráficas con interfaces de usuario amigables.



Cada componente Swing corresponde a una clase en Java. Cuando en una aplicación gráfica queremos usar uno de estos componentes basta con instanciar un objeto de la clase correspondiente. Por ejemplo, si queremos añadir un componente área de texto debemos crear un objeto de la clase JTextArea.

Podemos ver ejemplos de componentes Java Swing en el enlace http://da2i.univ-lille1.fr/doc/tutorial-java/ui/features/components.html

Dentro de Swing podemos diferenciar distintos tipos de componentes.

CONTENEDORES

Un contenedor es el lienzo donde pintaremos nuestros componentes gráficos, existen contenedores principales, entre estos se encuentran JFrame y JDialog pero también existen otros contendedores incluidos dentro de los mencionados.

Existen varios contenedores de alto nivel. Toda aplicación gráfica debe de tener al menos uno de ellos para poder ejecutarse. Los contenedores de alto nivel son

- **JFrame**. Crea una ventana que se puede usar como contenedor principal de la aplicación
- **JDialog**. Crea una ventana de tipo Ventana de diálogo.
- **JApplet**. Crea una ventana dentro de una página web dentro de la cual se ejecutará código Java.

Además, existen varios contenedores de propósito general que son

- **JPanel**. Permite la creación de paneles independientes donde se almacenan otros componentes.
- **JScrollPane**. Permite la vinculación de barras de desplazamiento en un contenedor.
- **JSplitPane**. Permite la creación de un contenedor dividido en 2 secciones.
- **JTabbedPane**. Permite la creación de pestañas, cada pestaña representa un contenedor independiente.
- **JDesktopPane**. Permite crear ventanas dentro de una ventana principal
- **JToolBar**. Permite introducir una Barra de herramientas

COMPONENTES ATÓMICOS

Los componentes atómicos no pueden almacenar otros componentes gráficos, por ejemplo, un JPanel no es Atómico, ya que en él podemos almacenar otros componentes.

- **JLabel**. Permite crear etiquetas, tanto de texto como de imágenes
- **JButton**. Permite crear Botones simples.
- **JCheckBox**. Son casillas de verificación, ideales para selección múltiples.
- **JRadioButton**. Permite presentar opciones de selección de las que sólo se debe seleccionar una.
- **JToggleButton**. Botón que tiene dos estados, presionado o no.
- **JComboBox**. Permite mostrar una lista de elementos como un combo de selección.
- **JScrollBar**. Permite mostrar una barra de desplazamiento. Normalmente se usa en área de texto o paneles donde el contenido es mayor que el tamaño del componente.
- **JSeparator**. Permite separar opciones, es una barra simple.
- **JSlider**. Permite vincular un deslizador en nuestra ventana.
- **JSpinner**. Permite vincular una caja de texto con botones integrados para seleccionar algún valor.
- **JProgressBar**. Establece una barra de progreso.

COMPONENTES DE TEXTO.

Son todos aquellos que nos permiten procesar cadenas de texto, sea como entrada o salida de información.

- **JTextField**. Permite introducir un campo de texto simple.
- **JFormattedTextField**. Permite introducir un campo de texto con formato. Por ejemplo, si definimos que solo recibe números no permitirá letras.
- **JPasswordField**. Campo de texto que oculta los caracteres. Se usa para introducir contraseñas.
- **JTextArea**. Permite crear un área de texto donde el usuario escribirá información o simplemente para mostrar cadenas de texto.
- **JEditorPane**. Permite vincular un área de texto con propiedades de formato.
- **JTextPane**. Similar al anterior, permitiendo otras opciones de formato, colores, iconos entre otros.

COMPONENTES DE MENÚS.

Estos componentes permiten crear opciones de menú en nuestras ventanas, tipo menú principal, como por ejemplo el conocido Inicio, Archivo, Edición etc…

- **JMenuBar**. Permite crear una barra de menús.
- **JMenu**. Permite vincular botones o enlaces que al ser pulsados despliegan un menú principal.
- **JMenuItem**. Botón u opción que se encuentra en un menú.
- **JCheckBoxMenuItem**. Se usa en menús con opciones múltiples.
- **JRadioButtonMenuItem**. Se usa en menús con selección única.
- **JPopupMenu**. Permite crear un menú emergente.

COMPONENTES COMPLEJOS

La biblioteca Java Swing proporciona componentes avanzados que permiten realizar funciones complejas. Por ejemplo, componentes dedicados a obtener gran cantidad de información de una base de datos.

- **JTable**. Permite vincular una tabla de datos con sus respectivas filas y columnas.
- **JTree**. Carga un árbol donde se establece cierta jerarquía visual, tipo directorio.
- **JList**. Permite cargar una lista de elementos, dependiendo de las propiedades puede tenerse una lista de selección múltiple.
- **JFileChooser**. Es un componente que permite la búsqueda y selección de ficheros entre otras.
- **JColorChooser**. Componente que permite cargar un panel selector de color.
- **JOptionPane**. Es un panel que permite agrupar JRadioButton relativos al mismo rango de valores. Si tengo dos rangos de valores de objetos JRadioButton tengo que separarlos mediante un JOptionPane para que pueda seleccionarse una opción en cada uno de los rangos ya que sólo puede haber seleccionada una opción por panel.

LAYOUTS

Java proporciona la clase **Layout** que determina como se distribuyen los componentes dentro de una ventana en aplicaciones gráficas. La clase Layout es la que indica en qué posición van los botones y demás componentes, si van alineados, en forma de matriz, cuáles se hacen grandes al agrandar la ventana, etc. Otra cosa importante que decide el Layout es qué tamaño es el ideal para la ventana en función de los componentes que lleva dentro.

La clase Layout dispone del método **pack**() que aplicado a una ventana hace que el tamaño de la ventana de se adapte hasta el tamaño necesario para que se vea todo lo que tiene dentro.

```
ventana.pack();
```

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

En el enlace http://chuwiki.chuidiang.org/index.php?title=Uso_de_Layouts hay una definición de los distintos tipos de Layouts así como ejemplos de cada uno de ellos.

LAYOUT NULL

Uno de los Layouts más utilizados por la gente que empieza, por ser el más sencillo, es NO usar layout. Somos nosotros desde código los que decimos cada botón en qué posición va y qué tamaño ocupa

```
contenedor.setLayout(null);  // Eliminamos el layout
contenedor.add (boton); // Añadimos el botón
boton.setBounds (10,10,40,20); // Botón en posicion 10,10 con ancho 40 pixels y alto 20
```

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

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

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

FLOWLAYOUT

El FlowLayout es bastante sencillo de usar. Coloca los componente en fila. Hace que todos quepan (si el tamaño de la ventana lo permite). Es adecuado para barras de herramientas, filas de botones, etc.

```
contenedor.setLayout(new FlowLayout());
contenedor.add(boton);
contenedor.add(textField);
contenedor.add(checkBox);
```

BOXLAYOUT

El BoxLayout es como un FlowLayout, pero mucho más completo. Permite colocar los elementos en horizontal o vertical.

```
// Para poner en vertical
contenedor.setLayout(new BoxLayout(contenedor,BoxLayout.Y_AXIS));
contenedor.add(unBoton);
contenedor.add(unaEtiqueta);
```

Para profundizar más podemos visitar el enlace http://java.sun.com/docs/books/tutorial/uiswing/layout/box.html

GRIDLAYOUT

El GridLayout coloca los componentes en forma de matriz (cuadrícula), estirándolos para que tengan todos el mismo tamaño. El GridLayout es adecuado para hacer tableros, calculadoras en que todos los botones son iguales, etc.

```
// Creación de los botones
JButton boton[] = new JButton[9];
for (int i=0;i<9;i++)
  boton[i] = new JButton(Integer.toString(i));

// Colocación en el contenedor
contenedor.setLayout (new GridLayout (3,3));  // 3 filas y 3 columnas
for (int i=0;i<9;i++)
   contenedor.add (boton[i]);  // Añade los botones de 1 en 1.
```

BORDERLAYOUT

El BorderLayout divide la ventana en 5 partes: centro, arriba, abajo, derecha e izquierda.

Hará que los componentes que pongamos arriba y abajo ocupen el alto que necesiten, pero los estirará horizontalmente hasta ocupar toda la ventana.

Los componentes de derecha e izquierda ocuparán el ancho que necesiten, pero se les estirará en vertical hasta ocupar toda la ventana.

El componente central se estirará en ambos sentidos hasta ocupar toda la ventana.

El BorderLayout es adecuado para ventanas en las que hay un componente central importante (una tabla, una lista, etc) y tiene menús o barras de herramientas situados arriba, abajo, a la derecha o a la izquierda.

Este es el **layout por defecto** para los **JFrame** y **JDialog**.

```
contenedor.setLayout (new BorderLayout());
contenedor.add (componenteCentralImportante, BorderLayout.CENTER);
contenedor.add (barraHerramientasSuperior, BorderLayout.NORTH);
contenedor.add (botonesDeAbajo, BorderLayout.SOUTH);
contenedor.add (IndiceIzquierdo, BorderLayout.WEST);
contenedor.add (MenuDerecha, BorderLayout.EAST);
```

Por ejemplo, es bastante habitual usar un contenedor (JPanel por ejemplo) con un FlowLayout para hacer una fila de botones y luego colocar este JPanel en el NORTH de un BorderLayout de una ventana. De esta forma, tendremos en la parte de arriba de la ventana una fila de botones, como una barra de herramientas.

```
JPanel barraHerramientas = new JPanel();
barraHerrameientas.setLayout(new FlowLayout());
barraHerramientas.add(new JButton("boton 1"));
...
barraHerramientas.add(new JButton("boton n"));

JFrame ventana = new JFrame();
ventana.getContentPane().setLayout(new BorderLayout()); // No hace falta, por defecto ya es BorderLayout
ventana.getContentPane().add(barraHerramientas, BorderLayout.NORTH);
ventana.getContentPane().add(componentePrincipalDeVentana, BorderLayout.CENTER);

ventana.pack();
ventana.setVisible(true);
```

GRIDBAGLAYOUT

El GridBagLayout es de los layouts más versátiles y complejos de usar. Es como el GridLayout, pone los componentes en forma de matriz (cuadrícula), pero permite que las celdas y los componentes en ellas tengan tamaños variados.

Es posible hacer que un componente ocupe varias celdas

Un componente puede estirarse o no con su celda

Si no se estira, puede quedar en el centro de la celda o pegarse a sus bordes o esquinas.

Las columnas pueden ensancharse o no al estirar la ventana y la proporición podemos decidirla

Lo mismo con las filas.

En http://www.chuidiang.com/java/layout/GridBagLayout/GridBagLayout.php hay un tutorial más completo de cómo usar este layout.

CARDLAYOUT

El CardLayout hace que los componentes recibidos ocupen el máximo espacio posible, superponiendo unos a otros. Sólo es visible uno de los componentes, los otros quedan detrás. Tiene métodos para indicar cuál de los componentes es el que debe quedar encima y verse.

El CardLayout es el que utiliza el JTabbedPane (el de las pestañas) de forma que en función de la pestaña que pinchemos, se ve uno u otro.

SPRINGLAYOUT

Para los nostálgicos que usaban motif, este layout es muy similar a los attachment de motif.

Se añaden los componentes y para cada uno de ellos tenemos que decir qué distancia en pixel queremos que tenga cada uno de sus bordes respecto al borde de otro componente. Por ejemplo, para decir que el borde izquierdo de una etiqueta está a 5 pixels del panel que la contiene ponemos

```
layout.putConstraint(SpringLayout.WEST, label, 5, SpringLayout.WEST, contentPane);
```

Para decir que el borde derecho de la etiqueta debe estar a 5 pixels del borde izquierdo de un JTextField, ponemos esto

```
layout.putConstraint(SpringLayout.WEST, textField, 5, SpringLayout.EAST, label);
```

Con este layout, cuando estiramos el panel, siempre ceden aquellos componentes más "flexibles". Entre una etiqueta y una caja de texto, la caja de texto es la que cambia su tamaño.

En http://java.sun.com/docs/books/tutorial/uiswing/layout/spring.html hay un tutorial más completo de cómo usar este layout.

EVENTOS. DEFINICIÓN Y TIPOS

Las aplicaciones gráficas se componen de un conjunto de componentes sobre los que se realizan una serie de operaciones. Esas operaciones pueden ser por ejemplo hacer clic sobre un botón. Al realizar esas operaciones se producen **eventos**. Un evento es una situación que se produce dentro de la aplicación.

Para controlar un evento en Java debemos crear métodos que estén a la espera de que se produzca dicho evento. Por ejemplo, si queremos que se controle cuando es pulsado el botón btnAceptar debemos escribir

```
btnAceptar.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
  // accciones a realizar al pulsar el botón
}
});
```

EJEMPLO DE EVENTOS

Para comprobar el funcionamiento de los eventos vamos a crear un nuevo **JFrame** de nombre **VentanaHolaAceptar** que tiene un **Layout** de tipo (**absolute**), un **Title** con el valor "**Evento Clic**" y

muestra una etiqueta (**JLabel**) de nombre **lblTexto** con el **text** "No se ha pulsado Aceptar." Centrada en medio del layout, y un botón (**JButton**) de nombre **btnAceptar**.

Para ello vamos a **File -> New -> Other** y seleccionamos **Window Builder -> Swing Designer -> JFrame**.

Escribimos **VentanaHolaAceptar** en el campo **Name** y pulsamos **Finish**.

Para crear el evento que controle cuando se hace **clic sobre** el botón **btnAceptar** hacemos clic con el botón derecho sobre btnAceptar, seleccionamos la opción **Add Event Handler -> action -> actionPerformed** al hacer esto se generará automáticamente el código para controlar cuando se hace clic sobre btnAceptar.

En nuestro caso queremos que cuando se haga clic sobre el botón el texto de la etiqueta **lblTexto** cambie a "**Ha pulsado Aceptar.**"

Una vez que hemos terminado la aplicación en el modo diseño, revisamos el código fuente generado para comprobar que no haya problemas.

Para finalizar, la ejecutamos para comprobar su correcto funcionamiento.

JLABEL

Los componentes JLabel permiten mostrar un texto. Ese texto se guarda en la propiedad Text.

Para cambiar el texto de un componente JLabel se usa el método **setText** y para obtener el texto el método **getText**.

JBUTTON

Los componentes JButton permiten realizar acciones al hacer clic sobre ellos.

Debemos diferenciar el nombre de la **variable** JButton (**btnAceptar** en nuestro caso) del texto que muestra el botón (propiedad **Text**, **Aceptar** en nuestro caso).

El principal evento que se controla en los componentes JButton es el evento **action** que se produce al hacer clic sobre ellos. Para ello, si se quiere controlar el **clic** sobre un JButton, se crea un nuevo ActionListener dentro del cual se crea un método **actionPerformed** que ejecute el código que queramos ejecutar cuando se haga clic sobre el JButton. Después para completar el proceso se añade el ActionListener al JButton.

Por ejemplo, si queremos que al hacer clic sobre el JButton de nombre **btnAceptar** el texto de la etiqueta **lblTexto** cambie a "**Ha pulsado Aceptar.**" Escribimos

```
btnAceptar.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
  lblTexto.setText("Ha pulsado Aceptar.");
}
});
```

JTEXTFIELD

Los componentes JTextField permiten la introducción de datos. Los datos se leen como Strings de tal modo que si se quieren convertir a otro tipo de dato (Integer, Double, ...) debemos realizar la conversión de String a ese tipo de dato.

El principal evento que se controla en los componentes JTextField es el evento **action** que se produce al pulsar enter en ellos. Su comportamiento es el mismo que en el componente JButton

También se puede querer controlar que tecla se ha pulsado o se está pulsando. Para ello están los eventos de tipo **key** keyPressed, keyReleased, y keyTyped.

JPASSWORDFIELD

Los componentes JPasswordField se comportan como los JTextField pero permiten ocultar los datos que se están introduciendo. La principal diferencia es que los datos se guardan en la propiedad Password como un array de caracteres (char[]).

Para recuperar los datos debemos usar el método getPassword y, si queremos, convertirlos a un String.

Por ejemplo, si tenemos un campo de tipo JPasswordField y de nombre pwfContrasena y queremos guardar su valor en una variable de tipo String y de nombre contrasena escribimos

```
String contrasena = new String(pwfContrasena.getPassword());
```

ACTIONLISTENER COMUN A VARIOS COMPONENTES

En determinadas situaciones varios eventos producen la misma respuesta. Por ejemplo, en el ejemplo de la ventana con el nombre, la contraseña y el botón de Aceptar al hacer clic sobre el botón o al pulsar enter en el campo nombre o en el campo contraseña se ejecuta el mismo código que se encarga de comprobar si los datos son correctos.

En estas situaciones en que varios componentes comparten el mismo comportamiento podemos hacer que compartan también el mismo código lo que simplifica notablemente el código.

Uno de los métodos para hacer que varios componentes compartan el mismo comportamiento consiste en hacer que la clase que contiene los componentes implemente directamente ActionListener.

Por ejemplo, si tenemos una clase de nombre VentanaActionListenerComun y queremos que implemente directamente ActionListener escribimos

```
public class VentanaActionListenerComun extends JFrame implements ActionListener{...}
```

Dentro de la clase debemos de escribir un método **actionPerformed** que contenga el código común a todos los eventos producidos en los componentes

```
public void actionPerformed (ActionEvent e){
 // código común a todos los componentes
}
```

Para terminar debemos especificar en cada componente que vamos a usar el método actionPerformed propio de la clase. En nuestro caso si queremos que el bóton Aceptar y los campos nombre y contraseña usen el método actionPerformed propio de la clase escribimos

```
btnAceptar.addActionListener(this);
txtNombre.addActionListener(this);
pwfContrasena.addActionListener(this);
```

Al hacerlo de esta manera, haciendo que la clase implemente ActionListener, dentro del método actionPerformed podemos acceder a todos los componentes de la clase lo que puede ser útil en muchos de los casos.

ELEMENTO ACTIVO. FOCO. SELECCIONAR CONTENIDO DEL ELEMENTO CON FOCO

Aunque tengamos varios componentes a nuestra disposición solo uno de ellos será el que esté activo en ese momento.

El elemento sobre el que podemos realizar operaciones se dice que tiene el foco. Por ejemplo, los JTextField, necesitan tener el foco para poder escribir en ellos. Si no tienen el foco deberemos hacer que lo tengan haciendo clic sobre ellos o usando el tabulador antes de poder escribir en ellos.

Cuando un elemento recibe el foco se produce un evento de tipo **Focus**.

Un ejemplo típico del uso del evento Focus es hacer que se seleccione todo el texto de un componente al recibir el foco.

Por ejemplo, si queremos que se seleccione todo el texto del **JTextField** de nombre **txtNombre** al recibir el foco escribimos

```
txtNombre.addFocusListener(new FocusListener() {
 public void focusGained(FocusEvent e) {
  txtNombre.select(0, txtNombre.getText().length());
 }

 public void focusLost(FocusEvent e) {
  txtNombre.select(0, 0);
}
});
```

Si queremos que se seleccione todo el texto de un JPasswordField cambio un poco el código ya que la propiedad Password guarda un array de caracteres (char[]) y no un String y debemos convertir el array de caracteres a String.

Por ejemplo, para que se seleccione todo el texto del **JPasswordField** de nombre **pwfContrasena** al recibir el foco escribimos

```
pwfContrasena.addFocusListener(new FocusListener() {
 public void focusGained(FocusEvent e) {
  pwfContrasena.setSelectionStart(0);
 String contrasena = new String(pwfContrasena.getPassword());
 pwfContrasena.setSelectionEnd(contrasena.length());
 }

 public void focusLost(FocusEvent e) {
  pwfContrasena.select(0, 0);
 }
});
```

APLICACIONES MULTIVENTANA. ABRIR UNA VENTANA DESDE EL CODIGO DE OTRA VENTANA.

Muchas aplicaciones constan de varias ventanas que se van creando y destruyendo conforme se van necesitando.

Un caso típico es crear un JFrame desde el código de otro, configurarlo y mostrarlo.

Por ejemplo, podemos tener una ventana de inicio que pida el nombre de usuario y la contraseña y, si los datos son correctos, cree la ventana principal de la aplicación, modifique el título por "Bienvenido" + nombre del usuario, muestre la ventana principal (que en nuestro caso será una ventana de la clase VentanaHola) y cierre la ventana de inicio. Para ello, dentro del método que controla el evento ActionPerformed común escribimos

```
public void actionPerformed (ActionEvent e){
 //defino los datos correctos
 String nombrecorrecto = "1dw3";
 String contrasenacorrecta = "1dw3";
 // compruebo los datos
 // el metodo getPassword de JPasswordField devuelve un char[]
 // para poder usar equals tengo que convertir el char [] a String
 String contrasena = new String(pwfContrasena.getPassword());
 if(nombrecorrecto.equals(txtNombre.getText()) && contrasenacorrecta.equals(contrasena)){
  // si los datos son correctos
  // creo una nueva ventana
  VentanaHola vh = new VentanaHola();
  // le cambio el Title
  vh.setTitle("Bienvenido "+txtNombre.getText());
  // la muestro
  vh.setVisible(true);
  // oculto la ventana de inicio
  // this.setVisible(false);
  // borro de memoria la ventana de inicio
  this.dispose();

 }
 else{
  this.lblTexto.setText("Datos Incorrectos.");
 }
}
```

EJERCICIOS DE INTERFACES GRÁFICAS

1. Crea la clase VentanaHola que tiene un Layout de tipo (absolute), un Title con el valor "Hola" y muestra una etiqueta (JLabel) con el text "Hola Mundo." Centrada en medio del layout.

2. Crea la clase VentanaHolaAceptar que tiene un Layout de tipo (absolute), un Title con el valor "Evento Clic" y muestra una etiqueta (JLabel) de nombre lblTexto con el text "No se ha pulsado Aceptar." Centrada en medio del layout, y un botón (JButton) de nombre btnAceptar.

3. Crea la clase VentanaJTextField que añade a VentanaHolaAceptar un JTextField que inicialmente tiene el valor “Nombre...”. Inicialmente se mostrará en lblTexto el text "Anónimo." Al pulsar enter en el

JTextField o al pulsar el botón se mostrará en lblTexto el text " Bienvenido " y el valor que haya en el JTextField.

4. Crea la clase VentanaJPasswordField que añade a VentanaJTextField un JPasswordField que inicialmente tiene el valor "Password...". Inicialmente se mostrará en lblTexto el text "Anónimo." Al pulsar enter en el JTextField o en el JPasswordField o al pulsar el botón se comprobarán si los datos son correctos. Los datos correctos son "1dw3" como nombre y "1dw3" como contraseña. Si los datos son correctos mostrará en lblTexto el text " Bienvenido " y el valor que haya en el JTextField. Si los datos no son correctos mostrará en lblTexto el text "Datos Incorrectos.".

5. Crea la clase VentanaActionListenerComun que modifica VentanaJPasswordField para que al pulsar enter en el JTextField o en el JPasswordField o al pulsar el botón se ejecute el mismo método. Para ello la clase VentanaActionListenerComun implementará ActionListener en su definición de clase.

6. Crea la clase VentanaActionListenerComunFoco que modifica VentanaActionListenerComunFoco para que al coger el foco el JTextField se seleccione todo su texto y al coger el foco el JPasswordField se seleccione todo su textoCrea la clase FicheroCopiaExcepciones que pide el nombre del fichero origen y el nombre del fichero destino y copia el contenido del fichero origen en el fichero destino, controlando las excepciones que se puedan producir durante el proceso de copia.

7. Crea la clase MultiventanaActionListenerComunFoco que modifica VentanaActionListenerComun para que si los datos de nombre y contraseña son correctos se cree una nueva ventana de tipo VentanaHola con el Title "Bienvenido " y el valor del texto del JTextField, se muestre esa ventana, y se oculte la ventana de inicio.

BARRAS DE MENÚS. JMENUBAR, JMENU, JMENUITEM

Muchas aplicaciones contienen una barra de menús en la que aparecen las operaciones que se pueden realizar.

El componente Java que se usa para crear barras de menús es JMenuBar.

Por ejemplo, si queremos crear una barra de menús escribimos

```
JMenuBar menuBar = new JMenuBar();
```

Dentro de cada barra de menú se pueden crear varios menús. Para crear un menú se usa el componente Java JMenu. Por ejemplo, si queremos crear un menú Archivo en la barra de menús escribimos

```
JMenu mnuArchivo = new JMenu("Archivo");
```

A su vez, dentro de cada menú se pueden crear varias opciones. Para crear una opción se usa el componente Java JMenuItem. Por ejemplo, si queremos crear en el menú Archivo la opción Nuevo escribimos

```
JMenuItem mniNuevo = new JMenuItem("Nuevo");
```

Una vez creada la opción se debe añadir al menú. Por ejemplo, si queremos añadir al menú Archivo la opción Nuevo escribimos

```
mnuArchivo.add(mniNuevo);
```

Una vez creadas todas las opciones del menú añadimos el menú a la barra de menús. Por ejemplo, para añadir el menú Archivo a la barra de menús escribimos

```
menuBar.add(mnuArchivo);
```

Una vez creada la barra de menús se la asignamos a la ventana de nuestra aplicación escribiendo

```
this.setJMenuBar(menuBar);
```

A partir de este momento nuestra aplicación ya cuenta con una barra de menús.

BARRAS DE HERRAMIENTAS. JTOOLBAR

Muchas aplicaciones contienen una barra de herramientas en la que aparecen iconos con las operaciones que se pueden realizar.

El componente Java que se usa para crear barras de herramientas es JToolBar.

Por ejemplo, si queremos crear una barra de herramientas escribimos

```
JToolBar toolBar = new JToolBar();
```

Dentro de cada barra de herramientas se pueden crear varios botones. Para crear un botón se usa el componente Java JButton. La principal característica de los botones de las barras de herramientas es que llevan asociado un icono para identificar la operación que realizan aunque pueden tener también un texto informativo.

Por ejemplo, si queremos añadir un botón para ejecutar la opción Nuevo en la barra de herramientas escribimos

```
JButton btnNuevo = new JButton();
btnNuevo.setText("Nuevo");
btnNuevo.setIcon(new ImageIcon(JEditorPaneTest.class.getResource("/iconos/abrir.gif")));
btnNuevo.setMargin(new Insets(0, 0, 0, 0));
toolBar.add(btnNuevo);
```

Es muy importante que los iconos se encuentren dentro del proyecto. En mi caso los iconos se encuentran en una carpeta de nombre iconos que se encuentra en la raíz del proyecto. Al estar dentro del proyecto podemos ir a la propiedad **icon** del botón **btnNuevo**, hacer clic en **...** y en la ventana que aparece seleccionar **Classpath resource**. Allí buscamos la carpeta iconos y seleccionamos el icono que deseemos.

Además, al estar la carpeta iconos en la raíz del proyecto la ruta es conocida pues para acceder a los archivos que se encuentren en ella basta con poner la ruta **/iconos/archivo**

Una vez creadas todas las opciones de la barra de herramientas la añadimos a nuestra ventana. Por ejemplo, para añadir nuestra barra de herramientas a la parte de arriba de nuestra ventana escribimos

```
contentPane.add(toolBar, BorderLayout.NORTH);
```

A partir de este momento nuestra aplicación ya cuenta con una barra de herramientas.

BARRAS DE ESTADO

Muchas aplicaciones contienen una barra de estado en la que aparece información relativa al estado de la ejecución de la aplicación.

Para crear una barra de estado podemos crear un panel que situaremos en la parte inferior de nuestra ventana. Una vez creado el panel lo dividimos en tantas zonas como deseemos. Una vez creadas las zonas mostramos la información que deseemos en cada una de ellas.

Por ejemplo, para crear una barra de estado consistente en un panel con dos zonas escribimos

```
JPanel barraEstadoPanel = new JPanel();
barraEstadoPanel.setLayout(new BorderLayout());
//creo los mensajes para la barra de estado
estado1 = new JLabel("Estado: ");
estado2 = new JLabel();
barraEstadoPanel.add(estado1, BorderLayout.WEST);
barraEstadoPanel.add(estado2, BorderLayout.CENTER);
```

Una vez creada la barra de estado la tenemos que añadir al contenedor de nuestra ventana. Por ejemplo, para añadir la barra de estado que hemos creado en la parte inferior de nuestra ventana escribimos.

```
contentPane.add(barraEstadoPanel, BorderLayout.SOUTH);
```

JEDITORPANE

Para facilitar el desarrollo de aplicaciones que incorporan un editor de texto Java proporciona el componente JEditorPane.

Si queremos que el componente JEditorPane tenga barras de desplazamiento debemos crear un panel que permita barras de desplazamiento usando el componente JScrollPane indicando que queremos que contenga al editor.

Una vez creado el componente JEditorPane lo tenemos que añadir a nuestra ventana. Por ejemplo, para añadir un JEditorPane con barras de desplazamiento a nuestra ventana escribimos

```
JEditorPane editor = new JEditorPane();
editor.setText("");
// añado una barra de desplazamiento al editor
JScrollPane scrollPaneEditor = new JScrollPane(editor);
// Agrega el editor en el centro del contenedor
contentPane.add(scrollPaneEditor, BorderLayout.CENTER);
```

VENTANAS O CUADROS DE DIÁLOGO PREDEFINIDOS. JOptionPane

Para facilitar el desarrollo de aplicaciones Java proporciona la clase JOptionPane que permite crear unas ventanas o cuadros de diálogo que facilitan la realización de unas determinadas funciones comunes a muchas aplicaciones.

Los métodos que vamos a ver son

- **showMessageDialog**. Muestra una ventana con un mensaje informativo y un botón de Aceptar. No devuelve ningún valor. Un ejemplo es

```
JOptionPane.showMessageDialog(null,(String)"Prueba de Cuadros de Diálogo","Cuadro de Diálogo Mensaje",JOptionPane.INFORMATION_MESSAGE,null);
```

- **showInputDialog**. Muestra una ventana con un campo de texto en el que hay que introducir un valor y dos botones para Aceptar o Cancelar. Devuelve un Objeto, normalmente un String. Un ejemplo es

```
String respuesta = (String)JOptionPane.showInputDialog(null,(String)"Introduzca su Nombre: ","Cuadro de Diálogo de Introducción de Datos",JOptionPane.QUESTION_MESSAGE,null, null, "Sin Nombre");
```

- **showConfirmDialog**. Muestra un mensaje informativo y tres botones Sí, No, y Cancelar. Devuelve un entero con el valor de la opción seleccionada. Un ejemplo es

```
int opcion = JOptionPane.showConfirmDialog(null,(String)"Prueba de Cuadros de Diálogo","Cuadro de Diálogo de Confirmación",JOptionPane.YES_NO_CANCEL_OPTION,JOptionPane.QUESTION_MESSAGE,null);
```

El valor devuelto es un dato de tipo int puede ser alguna de las siguientes constantes simbólicas
YES_OPTION. Si se ha pulsado el botón **Sí** en el cuadro de diálogo.
NO_OPTION. Si se ha pulsado el botón **No** en el cuadro de diálogo.
CANCEL_OPTION. Si se ha pulsado el botón Cancelar en el cuadro de diálogo.
OK_OPTION. Si se ha pulsado el botón **Ok** o **Aceptar** en el cuadro de diálogo.
CLOSED_OPTION. Si se ha cerrado el cuadro de diálogo haciendo clic en el icono X de la parte superior derecha.

- **showOptionDialog**. Muestra una ventana con un mensaje y tantos botones como opciones queramos controlar. Se utiliza para crear ventanas personalizadas. Las opciones válidas se especifican en un array de tipo String. La opción predefinida debe de ser una de las opciones válidas. **Devuelve un entero con la posición de la opción seleccionada dentro del array de opciones**.Un ejemplo es

```
String[] opciones = {"1AS3", "2AS3", "1DW3", "2DW3", "1SM2", "2SM2"};

int opcion = JOptionPane.showOptionDialog(null,"Prueba de Cuadros de Diálogo","Cuadro de Diálogo con OpcionesPersonalizadas",JOptionPane.DEFAULT_OPTION,JOptionPane.QUESTION_MESSAGE,null,opciones,opciones[0]);
```

Los métodos se pueden personalizar en función de las necesidades. No todos los métodos necesitan todos los parámetros. Los parámetros que pueden recibir los métodos son:

- **parentComponent.** A partir de este componente, se intentará determinar cuál es la ventana que debe hacer de padre del JOptionPane. Se puede pasar null, pero conviene pasar, por ejemplo, el botón desde el cual se lanza la acción que provoca que se visualice el JOptionPane. De esta manera, la ventana de aviso se visualizará sobre el botón y no se podrá ir detrás del mismo si hacemos click en otro sitio.
- **Message.** El mensaje a mostrar, habitualmente un String, aunque vale cualquier Object cuyo método toString() devuelva algo con sentido.
- **Title.** El título para la ventana.
- **optionType.** Un entero indicando qué opciones queremos que tenga la ventana. Los posibles valores son las constantes definidas en JOptionPane: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, o OK_CANCEL_OPTION.
- **messageType.** Un entero para indicar qué tipo de mensaje estamos mostrando. Este tipo servirá para que se determine qué icono mostrar. Los posibles valores son constantes definidas en JOptionPane: ERROR_MESSAGE (mensaje de Error), INFORMATION_MESSAGE (mensaje Informativo), WARNING_MESSAGE (mensaje de Advertencia), QUESTION_MESSAGE (mensaje para Preguntar algo), o PLAIN_MESSAGE (mensaje Normal).
- **Icon.** Un icono para mostrar. Si ponemos null, saldrá el icono adecuado según el parámetro messageType. Para poner un icono personalizado por ejemplo el icono **aplicación.png** que se encuentra en la carpeta **iconos** de nuestro proyecto escribimos

```
ImageIcon icono =
new ImageIcon (JOptionPaneShowOptionDialogIcono.class.getResource ("/iconos/aplicacion.png"));
```

- **options**: Un array de objects que determinan las posibles opciones. Si los objetos son componentes visuales, aparecerán tal cual como opciones. Si son String, el JOptionPane pondrá tantos botones como String. Si son cualquier otra cosa, se les tratará como String llamando al método toString(). Si se pasa null, saldrán los botones por defecto que se hayan indicado en optionType.
- **initialValue**: Selección por defecto. Debe ser uno de los Object que hayamos pasado en el parámetro options. Se puede pasar null.

Estas ventanas o cuadros de diálogo predefinidos permanecen en primer plano hasta que se elige una opción o se cierra la ventana. A este **comportamiento** se le denomina **modal** y a los componentes que lo usan se les denomina modales.

EJERCICIOS DE VENTANAS O CUADROS DE DIÁLOGO PREDEFINIDOS. JOptionPane

8. Crea la clase JOptionPaneShowMessageDialog que muestra un cuadro de diálogo con el mensaje "Prueba de Cuadros de Diálogo". El título del cuadro de diálogo será "Cuadro de Diálogo Mensaje". El tipo de mensaje será informativo.

9. Crea la clase JOptionPaneShowConfirmDialog que muestra un cuadro de diálogo con el mensaje "Prueba de Cuadros de Diálogo". El título del cuadro de diálogo será "Cuadro de Diálogo de Confirmación". Las opciones que aparecerán serán Sí, No, y Cancelar. El tipo de mensaje será informativo. El icono que aparecerá será el icono por defecto. Cuando se elija una opción aparecerá otro cuadro de diálogo con un mensaje indicando que opción se ha pulsado. El título de este cuadro de diálogo será "Respuesta" y tendrá sólo un botón de Aceptar.

10. Crea la clase JOptionPaneShowInputDialog que muestra un cuadro de diálogo con el mensaje "Introduzca su Nombre: ". El título del cuadro de diálogo será "Cuadro de Diálogo de Introducción de Datos". El tipo de mensaje será para preguntar. Como valor inicial pondremos "Sin Nombre". Si introduce un valor y pulsa Aceptar, aparecerá otro cuadro de diálogo con el mensaje "Hola " + el valor introducido. Si no se ha pulsado Aceptar no hace nada. El título de este cuadro de diálogo será "Respuesta" y tendrá sólo un botón de Aceptar.

11. Crea la clase JOptionPaneShowOptionDialog que muestra un cuadro de diálogo con el mensaje Seleccione una opción...". El título del cuadro de diálogo será " Cuadro de Diálogo con Opciones Personalizadas". El tipo de mensaje será para preguntar. Las posibles opciones son "1AS3", "2AS3", "1DW3", "2DW3", "1SM2", y "2SM2". Como valor inicial pondremos la primera de las opciones. Si elige una opción, aparecerá otro cuadro de diálogo indicando que opción se ha pulsado. El título de este cuadro de diálogo será "Respuesta" y tendrá sólo un botón de Aceptar.

12. Crea la clase JOptionPaneShowOptionDialogIcono que modifica JOptionPaneShowOptionDialog para que el cuadro de diálogo muestre un icono personalizado.

SELECCIÓN DE ARCHIVOS. JFileChooser

JFileChooser es una clase Java que facilita la selección de un archivo. Muestra una ventana en la que podemos navegar por el sistema de archivos para seleccionar un archivo. El idioma de la ventana dependerá por defecto del idioma del sistema operativo.

Para crear un JFileChooser y mostrar una ventana con el formato para seleccionar un archivo que deseemos abrir escribimos

```
JFileChooser fileChooser = new JFileChooser();
int opcion = fileChooser.showOpenDialog(contentPane);
```

El método **showOpenDialog** devuelve al ejecutarse un valor entero que puede ser uno de los siguientes datos enumerados

- JFileChooser.**APPROVE_OPTION** Si el usuario le ha dado al botón Aceptar
- JFileChooser.**CANCEL_OPTION** Si el usuario le ha dado al botón Cancelar.
- JFileChooser.**ERROR_OPTION** Si ha ocurrido algún Error.

Por ejemplo, para comprobar que opción se ha pulsado escribimos

```
if (opcion == JFileChooser.APPROVE_OPTION){
 // si ha pulsado Aceptar
 lblTexto.setText("Ha elegido el archivo "+fileChooser.getSelectedFile());
}
else if (opcion == JFileChooser.CANCEL_OPTION){
 // si ha pulsado Cancelar
 lblTexto.setText("Ha pulsado Cancelar");
}
else if (opcion == JFileChooser.ERROR_OPTION){
 // si ha producido un Error
 lblTexto.setText("Se ha producido un Error.");
}
```

Para mostrar una una ventana con el formato para seleccionar un archivo que deseemos guardar usamos el método **showSaveDialog** en vez de showOpenDialog.

SELECCIÓN DE COLOR. JColorChooser

JColorChooser es una clase Java que facilita la selección de un color. Muestra una ventana en la que podemos seleccionar un color. El idioma de la ventana dependerá por defecto del idioma del sistema operativo.

Para crear un JColorChooser y mostrar una ventana para cambiar el color de fondo escribimos

```
Color nuevoColor = JColorChooser.showDialog(contentPane, "Elija un Color ...", contentPane.getBackground());
contentPane.setBackground(nuevoColor);
```

SELECCIÓN DE FUENTE. JFontChooser

La clase Java JFontChooser no existe en la jerarquía de clases Java. Sin embargo, un grupo de desarrolladores de código abierto la han creado y permiten descargarla desde el enlace http://sourceforge.jp/projects/jfontchooser/.

En nuestro caso vamos a descargar el archivo **jfontchooser-1.0.5-bin.zip** que contiene el archivo **jfontchooser-1.0.5.jar** que es el que necesitamos.

Para poder usar la clase Java JFontChooser en nuestro proyecto vamos a renombrar el archivo jfontchooser-1.0.5.jar con el nombre **JFontChooser.jar** y lo vamos a copiar en el paquete en el que lo vamos a usar (evaluacion3 en nuestro caso).

Para poderlo usar no basta con copiarlo dentro del paquete también tenemos que añadirlo al Build Path del proyecto.

Para ello, hacemos **clic** con el **botón derech**o sobre el archivo **JFontChooser.jar** que se encuentra en la ventana del **Package Explorer**, en el menú que aparece seleccionamos **Build Path** y dentro de su submenú elegimos **Add to Build Path**. Al hacer esto ya podremos usar la clase Java JFontChooser en nuestras aplicaciones sin problemas.

El método que muestra el JFontChooser es **showDialog** que recibe el contenedor dónde se tiene que mostrar (contentPane en nuestro caso) y de devuelve el valor **JFontChooser.OK_OPTION** si se ha pulsado Aceptar.

El método que devuelve la Fuente que ha sido seleccionada es **getSelectedFont**.

Por ejemplo, para crear un JFontChooser y mostrar una ventana para cambiar la fuente de una etiqueta de nombre lblTexto escribimos

```
int opcion;
Font nuevaFuente;
// muestro el JFontChooser
JFontChooser fontChooser = new JFontChooser();
opcion = fontChooser.showDialog(contentPane);

if (opcion == JFontChooser.OK_OPTION){
 // si se ha pulsado OK
 // cambio la fuente de la etiqueta
 nuevaFuente = fontChooser.getSelectedFont();
 lblTexto.setFont(nuevaFuente);
 // cambio el texto de la etiqueta
 lblTexto.setText("La fuente ha sido cambiada.");
}
```

EJERCICIO EDITOR DE TEXTO. VentanaJEditorPane

Para comprobar todo lo visto hasta ahora vamos a crear una nueva clase Java de nombre **VentanaJEditorPane**. Está clase contará con una barra de menús que contendrá los siguientes menús

Menú **Archivo**. Contará con las siguientes opciones:

- **Nuevo**. Realizará todas las operaciones necesarias para crear un nuevo documento. Si los datos han sido modificados se preguntará al usuario si los desea guardar. El nombre por defecto para los nuevos archivos es "**nuevo.txt**".
- **Abrir**. Realizará todas las operaciones necesarias para cargar un documento de texto en nuestro editor (extensión .**txt**). Si los datos han sido modificados se preguntará al usuario si los desea guardar. Al abrir un nuevo archivo cambiará el título de la ventana por el del nombre del archivo que se ha abierto.
- **Guardar**. Realizará todas las operaciones necesarias para guardar los datos del editor en un archivo de extensión .txt. Los datos se guardarán sólo si han sido modificados.
- **Guardar Como**. Pedirá un nombre de archivo (con extensión .txt) y lo guardará.
- **Salir**. Saldrá de la aplicación. Si los datos han sido modificados se preguntará al usuario si los desea guardar.

Menú **Editar**. Contará con las siguientes opciones:

- **Cortar**. Cogerá el texto seleccionado y lo cortará al portapapeles.

- **Copiar**. Cogerá el texto seleccionado y lo copiará al portapapeles.
- **Pegar**. Pegará el contenido del portapapeles en el editor. Teniendo en cuenta que al pegar un texto cambian los datos del editor.

Menú **Formato**. Contará con las siguientes opciones:

- **Fuente**. Mostrará un JFontChooser para elegir la nueva fuente del texto del editor.
- **Color Texto**. Mostrará un JColorChooser para elegir el nuevo color del texto del editor.

Menú **Ayuda**. Contará con las siguientes opciones:

- **Acerca de**. Mostrará una nueva ventana con información de la aplicación.

También contará con una barra de herramientas (JToolBar) que permitirá ejecutar todas las opciones de los menús. Los iconos de la barra de herramientas se encuentran en una carpeta de nombre iconos que se encuentra en el directorio src por lo que para acceder a ellos desde la aplicación basta con poner la ruta **/iconos/nombre.gif**.

Aunque en principio se puede poner como icono cualquier tipo de imagen en la práctica sólo funcionan bien las imágenes con extensión **.GIF**. Además, para evitar problemas, todos los iconos deben de ser del mismo tamaño (yo he elegido de 32 x 32 pixels). Para convertir los iconos al formato GIF he usado el conversor online que se encuentra en http://www.online-convert.com/es?fl=es
Los iconos me han dado muchísimos problemas. He tenido que ir a la carpeta **bin** del proyecto y **borrar** los **iconos** que había cambiado porque el Eclipse no los actualizaba bien.

Además, he tenido que crear mis propios iconos.gif recortando iconos de varios sitios y redimensionándolos a 32 x 32.

En la parte inferior aparecerá una barra de estado que mostrará el estado del editor. Los valores de estado son "**Nuevo**" cuando se crea un nuevo documento, "**Abierto**" cuando se abre un nuevo documento, "**Modificado**" si los datos del editor han sido modificados, "**Guardado**" si los datos han sido guardados, "Fuente Cambiada" cuando se cambie la fuente, y "Color de Texto Cambiado" cuando se cambie el color del texto del editor.

Para el correcto funcionamiento de la aplicación se crearán los siguientes **métodos**

- **nuevoDocumento**. Realizará las operaciones necesarias para cargar un nuevo documento en el editor.
- **abrirDocumento**. Realizará las operaciones necesarias para cargar un documento en el editor.
- **guardarDocumento**. Realizará las operaciones necesarias para guardar en un documento los datos del editor.
- **guardarComoDocumento**. Realizará las operaciones necesarias para guardar en un documento los datos del editor con un nombre determinado.
- **preguntarSiGuardar**. Muestra una ventana preguntando si se quieren guardar o no los datos. Si el usuario elige la opción Sí los datos se guardan.
- **salir**. Realizará las operaciones necesarias para salir de la aplicación.

El control de las acciones de todos los componentes se realizará a través del método **actionPerformed** de la clase dentro del cual se controlará sobre que componente se ha realizado la acción. Para ello la clase debe implementar el interfaz **ActionListener**.

Para controlar si los datos del editor han sido modificados debemos controlar los eventos de tipo KeyListener. Para ello la clase debe implementar el interfaz **KeyListener**.

EJERCICIO EDITOR DE TEXTO URL. VentanaJEditorPaneUrl

Los componentes JEditorPane se pueden usar para trabajar con páginas web. Para comprobarlo vamos a crear una nueva clase Java de nombre VentanaJEditorPaneUrl que implementa el interfaz **HyperlinkListener** que controla el evento **hyperlinkUpdate**.

Esta clase cuenta con un JEditorPane, que no puede ser modificado, en el que se carga al iniciar la página web http://www.fptxurdinaga.com

Al situar el ratón sobre un enlace cambiará el título del Editor por el del enlace.

CASILLAS DE VERIFICACIÓN. JCheckBox

Java proporciona la clase **JCheckBox** para facilitar el trabajo con casillas de verificación. La principal ventaja de las casillas de verificación es que cuando trabajan en grupo puede haber **varias seleccionadas** a la vez.

Por ejemplo, podemos querer que un texto esté normal, o en cursiva, o en negrita, o en cursiva y negrita.

Para controlar que acción se ha realizado sobre los objetos JCheckBox podemos usar un **ActionListener** con su método **actionPerformed** y controlar sobre que elemento se ha hecho clic o podemos usar un **ChangeListener** con su método **stateChanged** y actualizar el estado usando el método **isSelected**() de cada uno de los objetos JCheckBox.

Por ejemplo, si tenemos una clase de nombre VentanaJCheckBox que deriva de JFrame y que tiene una etiqueta de nombre lblTexto con el valor "Texto de Prueba" y dos JCheckBox de nombre chkCursiva y chkNegrita que hacen que el texto de lblTexto se ponga en negrita o en cursiva escribimos

```
public class VentanaJCheckBox extends JFrame implements ChangeListener {
 private JPanel contentPane;
 private JLabel lblTexto;
 private JCheckBox chkCursiva;
 private JCheckBox chkNegrita;
 ...
 public VentanaJCheckBox() {
 lblTexto = new JLabel("Texto de Prueba");
 contentPane.add(lblTexto);
 ...
 chkCursiva = new JCheckBox("Cursiva");
 chkCursiva.addChangeListener(this);
 contentPane.add(chkCursiva);
 chkNegrita = new JCheckBox("Negrita");
 chkNegrita.addChangeListener(this);
 contentPane.add(chkNegrita);
 ...
 }
 // controlo el cambio de los JCheckBox
 @Override
 public void stateChanged(ChangeEvent arg0) {
  // cojo la fuente actual
  Font nuevaFuente = this.lblTexto.getFont();
  int formato = 0;
  // actualizo la fuente
```

```
  if (this.chkCursiva.isSelected()){
   // si hay que ponerla en cursiva
   formato = formato + Font.ITALIC;
  }
  if (this.chkNegrita.isSelected()){
   // si hay que ponerla en negrita
   formato = formato + Font.BOLD;
  }
  // actualizo el formato de la fuente
  this.lblTexto.setFont(new Font(nuevaFuente.getFamily(), formato, nuevaFuente.getSize()));
 }
}
```

CASILLAS DE OPCIÓN. JRadioButton

Java proporciona la clase **JRadioButton** para facilitar el trabajo con casillas de opción. La principal ventaja de las casillas de opción es que cuando trabajan en grupo **sólo** puede haber **una seleccionada** a la vez. Para ello los componentes JRadioButton se agrupan mediante componentes ButtonGroup de tal modo que dentro un ButtonGroup sólo hay una opción seleccionada.

Por ejemplo, podemos querer que un texto esté de color Negro, Rojo, o Azul.

Para controlar que acción se ha realizado sobre los objetos JRadioButton podemos usar un ActionListener con su método actionPerformed y controlar sobre que elemento se ha hecho clic.

Por ejemplo, si tenemos una clase de nombre VentanaJCheckBoxJRadioButton que es igual que VentanaJCheckBox pero que incluye tres objetos JRadioButton de nombre rbtNegro, rbtRojo, y rbtAzul que están agrupados en ButtonGroup de nombre btgColores y que al hacer clic sobre ellos cambia el color del texto de la etiqueta de nombre lblTexto escribimos

```
public class VentanaJCheckBoxJRadioButton extends JFrame implements ChangeListener,ActionListener {
  ...
 private ButtonGroup btgColores;
 //JRadioButton
 private JRadioButton rbtNegro;
 private JRadioButton rbtRojo;
 private JRadioButton rbtAzul;
 ...
 public VentanaJCheckBoxJRadioButton() {
 ...
  //JRadioButton
  rbtNegro = new JRadioButton("Negro");
  rbtNegro.setFont(new Font("Tahoma", Font.BOLD, 11));
  rbtNegro.setSelected(true);
  rbtNegro.addActionListener(this);
  panel.add(rbtNegro);

  rbtRojo = new JRadioButton("Rojo");
  rbtRojo.setFont(new Font("Tahoma", Font.BOLD, 11));
  rbtRojo.setForeground(Color.RED);
  rbtRojo.addActionListener(this);
  panel.add(rbtRojo);
```

```
 rbtAzul = new JRadioButton("Azul");
 rbtAzul.setFont(new Font("Tahoma", Font.BOLD, 11));
 rbtAzul.setForeground(new Color(0, 0, 255));
 rbtAzul.addActionListener(this);
 panel.add(rbtAzul);

 //agrupo los radio buttons.
 btgColores = new ButtonGroup();
 btgColores.add(rbtNegro);
 btgColores.add(rbtRojo);
 btgColores.add(rbtAzul);
...
}
//controlo el cambio de los JRadioButton
@Override
public void actionPerformed(ActionEvent ae) {
 // compruebo que JRadioButton se ha pulsado
 Object source = ae.getSource();
 if (source == this.rbtNegro){
  // si se ha pulsado negro
  this.lblTexto.setForeground(Color.BLACK);
 }
 else if (source == this.rbtRojo){
  // si se ha pulsado rojo
  this.lblTexto.setForeground(Color.RED);
 }
 else if (source == this.rbtAzul){
  // si se ha pulsado azul
  this.lblTexto.setForeground(Color.BLUE);
 }

 }
}
```

EJERCICIOS DE CASILLAS DE SELECCIÓN.

13. Crea la clase VentanaJCheckBoxJRadioButtonDoble que modifica VentanaJCheckBoxJRadioButton para que también aparezca otro grupo de radio buttons con los valores 12,14, y 16 que permitan cambiar el tamaño de la fuente.

LISTAS DE VALORES. JList

Java proporciona la clase **JList** para facilitar el trabajo con listas de valores. La principal ventaja de las listas de valores es que permiten agrupar opciones y seleccionar una o varias de ellas.

Los componentes JList crean listas genéricas. Si queremos crear una lista de objetos de tipo String lo tendremos que especificar.

Para controlar que objeto u objetos se han seleccionado en la JList podemos usar un **ListSelectionListener** con su método **valueChanged** y controlar que opción u opciones han sido seleccionadas. Para controlar que opción ha sido seleccionada podemos usar el método **getSelectedValue**().

Por ejemplo, si queremos crear una lista de datos de tipo String con las opciones "1AS3", "2AS3", "1DW3", "2DW3" y de nombre lstGrupos en la que cada vez que se selecciona una opción se actualiza el texto de la etiqueta lblTexto con el valor de la opción seleccionada escribimos

```
public class VentanaJList extends JFrame implements ListSelectionListener{
 ...
 // lista de los Grupos
 private JList<String> lstGrupos;
 ...
 public VentanaJList () {
 ...
  // lista de Opciones
  String[] opciones = { "1AS3", "2AS3", "1DW3", "2DW3" };
  lstGrupos = new JList<String>(opciones);
  lstGrupos.addListSelectionListener(this);
  contentPane.add(lstGrupos, BorderLayout.WEST);
 ...
 }

 @Override
 public void valueChanged(ListSelectionEvent lse) {
  // cuando cambia el elemento seleccionado
  // cambio el valor de la etiqueta
  String seleccion = (String) this.lstGrupos.getSelectedValue();
  this.lblTexto.setText(seleccion);
 }
}
```

Para evitar problemas con el WindowBuilder podemos añadir las opciones por defecto a la lista usando un componente del tipo **DefaultListModel**. De este modo el ejemplo anterior quedaría

```
// lista de Opciones
DefaultListModel<String> dlm = new DefaultListModel<String>();
dlm.addElement("1AS3");
dlm.addElement("2AS3");
dlm.addElement("1DW3");
dlm.addElement("2DW3");
lstGrupos = new JList<String>();
lstGrupos.setModel(dlm);
lstGrupos.addListSelectionListener(this);
contentPane.add(lstGrupos, BorderLayout.WEST);
```

LISTAS DE VALORES. SELECCIÓN MÚLTIPLE

La clase **JList** proporciona la posibilidad de que haya uno o varios elementos seleccionados a la vez. Esto viene determinado por el valor de la propiedad **selectionMode**. Si selectionMode tiene el valor ***ListSelectionModel.SINGLE_SELECTION*** solo se puede seleccionar un valor, si tiene el valor ***ListSelectionModel.SINGLE_INTERVAL_SELECTION*** se pueden seleccionar varios valores pero tienen que estar juntos, y si tiene el valor ***ListSelectionModel.MULTIPLE_INTERVAL_SELECTION*** se pueden seleccionar varios valores independientemente de dónde estén.

Por ejemplo, si tenemos la clase VentanaJListSeleccionMultiple que es igual que la clase VentanaJList pero permite la selección múltiple de cualquier grupo de elementos escribimos

```
lstGrupos.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
```

Para probar la selección múltiple vamos a crear una nueva clase de nombre **VentanaJListSeleccionMultiple** que modifica la clase VentanaJList para permitir la selección múltiple. Añade una nueva JList de nombre **lstGruposCopia** que coloca a la izquierda y dos botones de nombre btnCopiarDerecha y btnCopiarIzquierda que copian los elementos seleccionados de la lista de la izquierda (lstGrupos) a la de la derecha (lstGruposCopia) y viceversa. Para ello la clase implementa ActionListener y controla en el método **actionPerformed** que botón ha sido pulsado para realizar una copia u otra.

Para modificar la clase escribimos

```
public class VentanaJListSeleccionMultiple extends JFrame implements ActionListener{
 ...
. private JList<String> lstGrupos;
 private DefaultListModel<String> dlm;
 private JList<String> lstGruposCopia;
 private DefaultListModel<String> dlmCopia;
 ...
 public VentanaJListSeleccionMultiple() {
 ...
  // lista de Opciones
  dlm = new DefaultListModel<String>();
  dlm.addElement("1AS3");
  dlm.addElement("2AS3");
  dlm.addElement("1DW3");
  dlm.addElement("2DW3");
  lstGrupos = new JList<String>();
  lstGrupos.setModel(dlm);
  lstGrupos.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
  contentPane.add(lstGrupos, BorderLayout.WEST);
  // lista de Copia
  lstGruposCopia = new JList<String>();
  dlmCopia = new DefaultListModel<String>();
  lstGruposCopia.setModel(dlmCopia);
  contentPane.add(lstGruposCopia, BorderLayout.EAST);
  // btnCopiarIzquierda
  btnCopiarIzquierda = new JButton("<<");
  btnCopiarIzquierda.addActionListener(this);
  panel.add(btnCopiarIzquierda);
  // btnCopiarDerecha
  btnCopiarDerecha = new JButton(">>");
  btnCopiarDerecha.addActionListener(this);
  panel.add(btnCopiarDerecha);
 }

 // controlo cuando cambia la seleccion de la lista
 @Override
 public void actionPerformed(ActionEvent ae) {
  // cuando pulso un boton
```

```
 Object source = ae.getSource();

 if (source == this.btnCopiarDerecha){
  // si quiero copiar de la lista izquierda a la derecha
  int[]indices = this.lstGrupos.getSelectedIndices();
  int numeroOpciones = indices.length;
  String opcion = null;
  for(int posicion=0;posicion<numeroOpciones;posicion++){
   opcion = dlm.getElementAt(posicion);
   this.dlmCopia.addElement(opcion);
  }
 }
 else if (source == this.btnCopiarIzquierda){
  // si quiero copiar de la lista derecha a la izquierda
  int[]indices = this.lstGruposCopia.getSelectedIndices();
  int numeroOpciones = indices.length;
  String opcion = null;
  for(int posicion=0;posicion<numeroOpciones;posicion++){
   opcion = dlmCopia.getElementAt(posicion);
   this.dlm.addElement(opcion);
  }
 }
}
```

LISTAS DESPLEGABLES. JComboBox

Java proporciona la clase **JComboBox** para facilitar el trabajo con listas desplegables. Una lista desplegable se compone de un cuadro de texto y de una lista de valores. Los valores se pueden seleccionar escribiendo su valor en el cuadro de texto (si la propiedad Editable vale true) o desplegando la lista y haciendo click sobre el valor.

Para comprobar su funcionamiento vamos a crear la clase Java **VentanaJComboBox** que es igual que VentaneJList pero sustituye la JList lstGrupos por la JComboBox cmbGrupos que tiene como valor inicial el elemento que se encuentra en la posición 0 de la lista.

Para modificar la clase escribimos

```
public class VentanaJComboBox extends JFrame implements ActionListener{
 ...
. private JComboBox<String> cmbGrupos;
 ...
 public VentanaJComboBox () {
 ...
  // comboBox de Opciones
  cmbGrupos = new JComboBox<String>();
  cmbGrupos.setToolTipText("Grupos");
  cmbGrupos.addItem("1AS3");
  cmbGrupos.addItem("2AS3");
  cmbGrupos.addItem("1DW3");
  cmbGrupos.addItem("2DW3");
  cmbGrupos.setSelectedIndex(0);
  cmbGrupos.addActionListener(this);
```

```
 contentPane.add(cmbGrupos, BorderLayout.WEST);
}

// controlo cuando cambia la seleccion de la lista
@Override
public void actionPerformed(ActionEvent ae) {
 // cuando cambia el elemento seleccionado
 // cambio el valor de la etiqueta
 //int indiceSelecionado = lse.getFirstIndex();
 String seleccion = (String) this.cmbGrupos.getSelectedItem();
 this.lblTexto.setText(seleccion);

}
}
```

BARRAS DE PROGRESO. JProgressBar

Java proporciona la clase **JProgressBar** para facilitar el trabajo con barras de progreso. Una barra de progreso se utiliza para mostrar información relativa al progreso de una determinada acción (por ejemplo la instalación de una aplicación. Una barra de progreso cuenta con un valor mínimo, un valor máximo, y un valor actual.

Por ejemplo, para crear una JProgressBar con un valor mínimo de 0, un valor máximo de 100, que muestre el valor actual, y que cambie de valor cada segundo escribimos

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JProgressBar;
import javax.swing.SwingUtilities;

public class JProgressBarPrueba extends JPanel{

 JProgressBar pbar;
 static final int MINIMO = 0;
 static final int MAXIMO = 100;

 public JProgressBarPrueba() {
  pbar = new JProgressBar();
  pbar.setMinimum(MINIMO);
  pbar.setMaximum(MAXIMO);
  pbar.setStringPainted(true);
  add(pbar);
 }

 public void updateBar(int newValue) {
  pbar.setValue(newValue);
 }

 public static void main(String args[]) {

  final JProgressBarPrueba it = new JProgressBarPrueba();

  JFrame frame = new JFrame("JProgressBar Prueba");
```

```
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 frame.setContentPane(it);
 frame.pack();
 frame.setVisible(true);

 for (int i = MINIMO; i <= MAXIMO; i++) {
  final int percent = i;
  try {
   SwingUtilities.invokeLater(new Runnable() {
    public void run() {
     it.updateBar(percent);
    }
   });
   java.lang.Thread.sleep(100);
  } catch (InterruptedException e) {
   ;
  }
 }
}
}
```

EJERCICIOS COMPLEMENTARIOS.

14. Crea la clase VentanaJComboBoxJList que modifica VentanaJComboBox para incluya un botón y una JList. Al hacer clic sobre el botón se añadirá el elemento seleccionado de la JComboBox a la JList.

15. Crea la clase VentanaJComboBoxJListFicheros que modifica VentanaJComboBoxJList para que al iniciar el programa se carguen los datos desde el fichero "lista" en el DefaultListModel y al finalizar el programa, si los datos se han modificado, se graben los datos. Sólo se graban los grupos añadidos, el primer elemento de la lista, si es un mensaje informativo, no se graba en el fichero.

16. Crea la clase VentanaJComboBoxJListFicherosBorrar que modifica VentanaJComboBoxJListFicheros añadiendo el botón Del que borra los elementos que estén seleccionados en la lista (si hay algún elemento seleccionado), y el botón Clear que vacía la lista y la deja como al principio (si no está ya vacía).

17. Crea la clase VentanaJComboBoxJListAlumnos que modifica VentanaJComboBoxJList para que incluya una nueva JLabel con el texto "Alumno" y un nuevo JTextField de nombre txtAlumno. La JComboBox contendrá los valores 1AS3, 2AS3, 1DW3, y 2DW3. Inicialmente estará seleccionado 1AS3. Cada valor tendrá asociado un DefaultListModel por lo que habrá que crear los DefaultListModel de nombre dlm1as3, dlm2as3, dlm1dw3, y dlm2dw3. Cada DefaultListModel contará con el valor inicial del grupo al que hace referencia (por ejemplo el valor inicial de dlm1as3 será 1AS3). La lista tendrá asociado el DefaultListModel correspondiente al grupo que esté seleccionado en la JComboBox y, cuando cambie el grupo seleccionado, cambiará también el DefaultListModel asociado a la lista. Al hacer clic sobre el botón se añadirá el valor del JTextField al DefaultListModel asociado a la lista en ese momento.

18. Crea la clase VentanaJComboBoxJListAlumnosFicheros que modifica VentanaJComboBoxJListAlumnos para que al iniciar el programa se carguen los datos desde uno o varios ficheros en los DefaultListModel y al finalizar el programa, si los datos se han modificado, se graben los datos. El nombre del grupo no se graba en el fichero.

19. Crea la clase VentanaJProgressBar que modifica VentanaHolaAceptar para incluya una barra de progreso que va de 0 a 100 y muestra su valor. Al pulsar el botón btnIniciar empezará a funcionar la barra de progreso que irá actualizando su valor cada 100ms.

20. Crea el paquete chatGrafico y copia dentro de él las clases del chat cliente / servidor. Añade las clases necesarias y modifica las existentes para que el chat tenga un interfaz gráfico.