

Componentes de Swing

Componentes de Swing

Existen muchos componentes en Swing. Vamos a ver algunos de ellos, que pueden ser suficientes para la mayoría de las interfaces.

JTextField

JTextField permite insertar texto. Ya hemos visto **JTextField** con anterioridad. Se puede definir el comportamiento de un **JTextField** ante ciertos eventos:

- **keyTyped**: el evento se lanza cuando se pulsa una tecla correspondiente a un carácter. El método **getText** devuelve la cadena antes de modificar.
- **keyPressed**: el evento se lanza cuando se pulsa cualquier tecla. El método **getText** devuelve la cadena antes de modificar.
- **keyReleased**: el evento se lanza cuando se se suelta cualquier tecla. El método **getText** devuelve la cadena después de ser modificada.

```
txtField.addKeyListener(new KeyListener() {
    @Override
    public void keyTyped(KeyEvent e) {
        System.out.println("-----
-----");
        System.out.println("keyTyped");
        System.out.println("Tecla pulsada: " + e.getKeyChar());
        System.out.println("Contenido del JTextField: " +
txtField.getText());
    }

    @Override
    public void keyPressed(KeyEvent e) {
        System.out.println("-----
-----");
        System.out.println("keyPressed");
        System.out.println("Tecla pulsada: " + e.getKeyChar());
        System.out.println("Contenido del JTextField: " +
txtField.getText());

    }

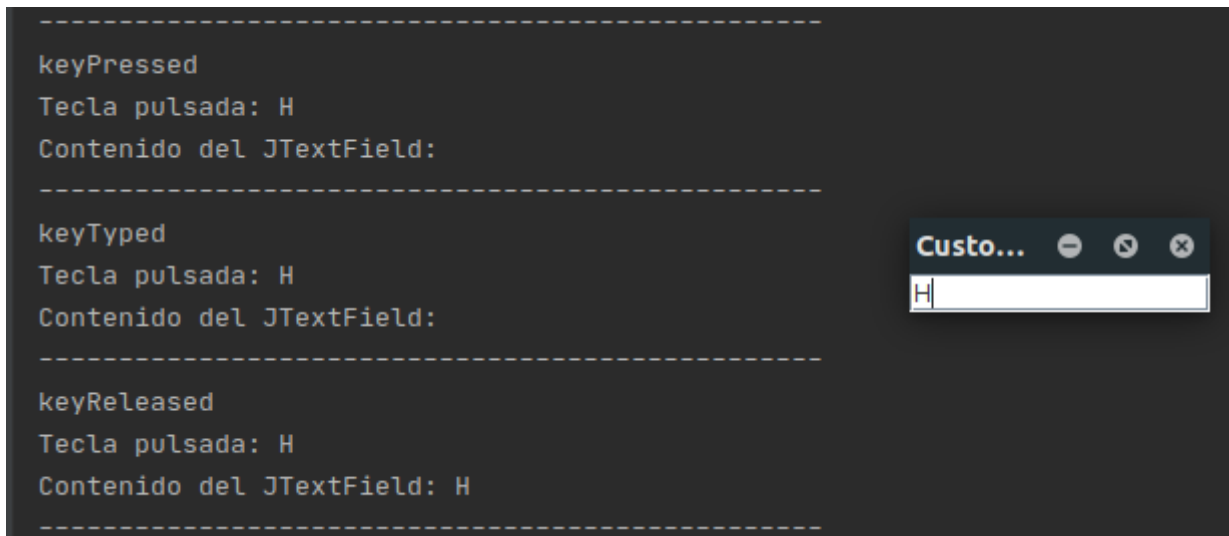
    @Override
```

```

        public void keyReleased(KeyEvent e) {
            System.out.println("-----");
            System.out.println("keyReleased");
            System.out.println("Tecla pulsada: " + e.getKeyChar());
            System.out.println("Contenido del JTextField: " +
txtField.getText());

        }
    });

```



```

-----
keyPressed
Tecla pulsada: H
Contenido del JTextField:
-----
keyTyped
Tecla pulsada: H
Contenido del JTextField:
-----
keyReleased
Tecla pulsada: H
Contenido del JTextField: H
-----

```

JComboBox

Un **JComboBox** es un menú desplegable. En un menú se puede colocar una lista de elementos, y se puede detectar el elemento seleccionado.

Para añadir elementos a un **JComboBox**, utilizamos el método **addItem**. El componente mostrará la versión de texto del objeto insertado.

```

while (lista.hasNext())
    cmbProductos.addItem(lectorFichero.getNext());

```

Para comprobar el elemento seleccionado, usamos el método **getSelectedItem**.

```

MiTipo varName = (MiTipo)comboBox.getSelectedItem();

```

Podemos recoger el evento "selección" mediante un **ActionListener**.

```

String[] opciones = new String[]{"opción1", "opción2", "opción3"};

for (String opcion:Arrays.asList(opciones)){
    cmbBox.addItem(opcion);
}

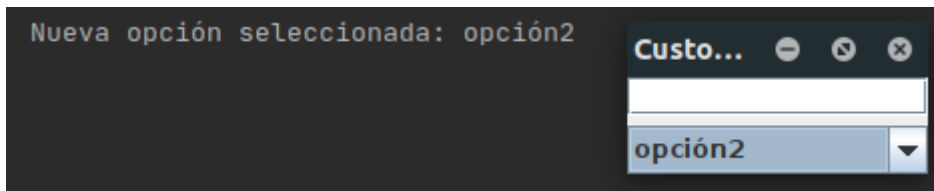
```

```

    }
    cmbBox.setSelectedIndex(0);

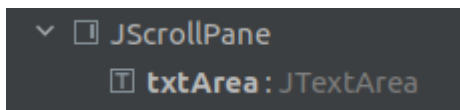
    cmbBox.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            System.out.println("Nueva opción seleccionada: " +
cmbBox.getSelectedItem());
        }
    });
}

```

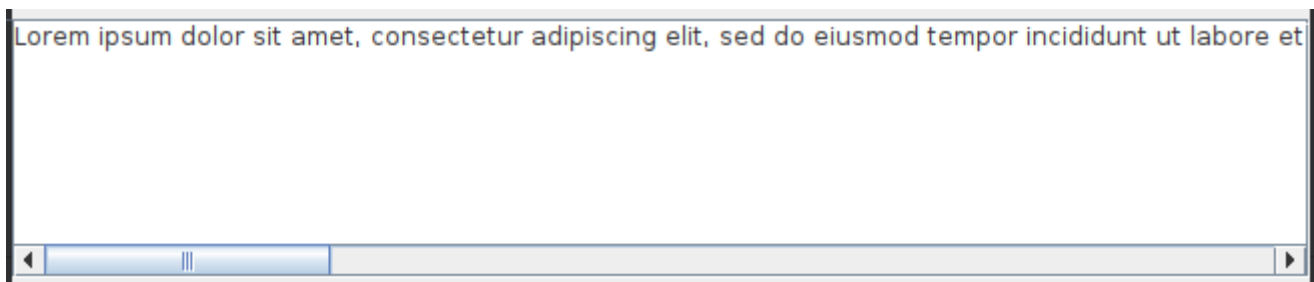


JTextArea

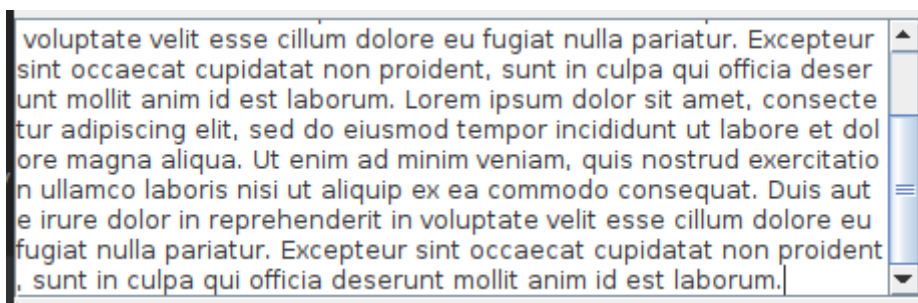
Algunos elementos pueden contener más información de la que pueden mostrar. Por ejemplo, supongamos un **JTextArea** que contiene más área del que puede mostrar. Para estos casos necesitamos colocar el **JTextArea** dentro de un componente **JScrollPane**



De este modo, cuando el contenido supera el tamaño del componente, aparece un scroll:



Si queremos que el Scroll aparezca verticalmente, debemos utilizar el método **setLineWrap**



Para obtener el contenido de un **JTextArea** debemos utilizar el método **getText**.

JList

El componente **JList** nos permite mostrar una lista de elementos de texto. Estos elementos van más allá de simples cadenas de texto, ya que se les puede dar un cierto formato. Además se los puede elegir de manera independiente.

Su funcionamiento es bastante parecido a un JComboBox, aunque da más juego en cuanto la forma de mostrar el contenido.

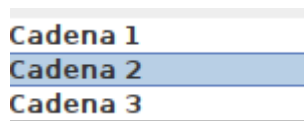
Un **JList** necesita un modelo, que consiste en un patrón **Adapter** de los objetos mostrados. El modelo por defecto de Swing, es suficiente para la mayoría de los casos. Para ilustrar una lista de cadenas de texto:

```
String[] cadenas = new String[]{"Cadena 1", "Cadena 2", "Cadena 3"};

DefaultListModel<String> modelo = new DefaultListModel<>();
for (String cadena:cadenas)
    modelo.addElement(cadena);

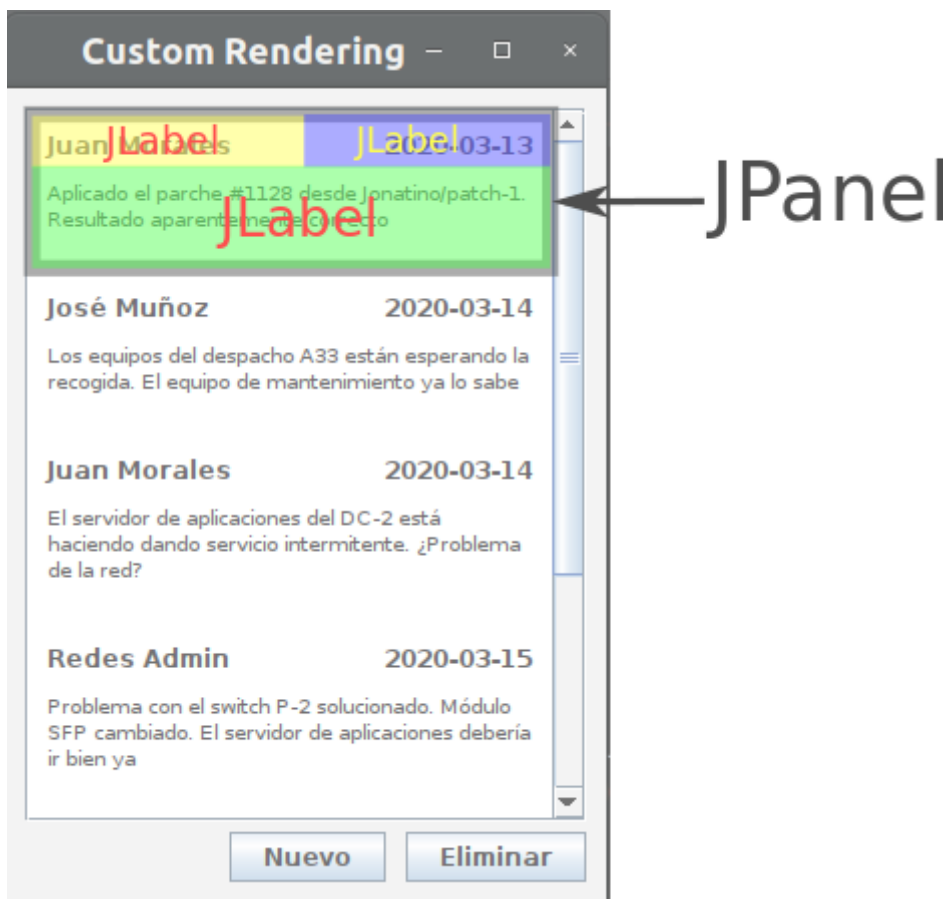
lst.setModel(modelo);
```

El componente resultante permite seleccionar uno de ellos:



Podemos comprobar el elemento seleccionado mediante el método **getSelectedValue()**

En ocasiones, es necesario mostrar elementos más complejos, como se muestra en la siguiente imagen:



El procedimiento para conseguir esto se denomina **CustomRenderer** y resulta algo más complejo. Queda descrito en el apéndice correspondiente al final del documento.

JTable

Mediante una tabla, se pueden mostrar datos de forma tabular. Al igual que las listas, las tablas utilizan un modelo (recuerda que Swing utiliza el patrón MVC). En el modelo, debemos especificar dos cosas:

- Nombres de columna
- Datos para las columnas

Ambas cosas se especifican mediante listas de tipo **Vector**

En el siguiente ejemplo, la tabla contiene cuatro columnas:

- Modelo
- Sección
- Tipo
- Elemento

Además, los datos a mostrar se suministran como un vector de arrays, donde cada array incluye los datos de una fila:

- C1001, Cocina, Silla, sc2011
- S2002, Salón, Mesa, ms1001
- B1011, Baño, Armario, ab4102

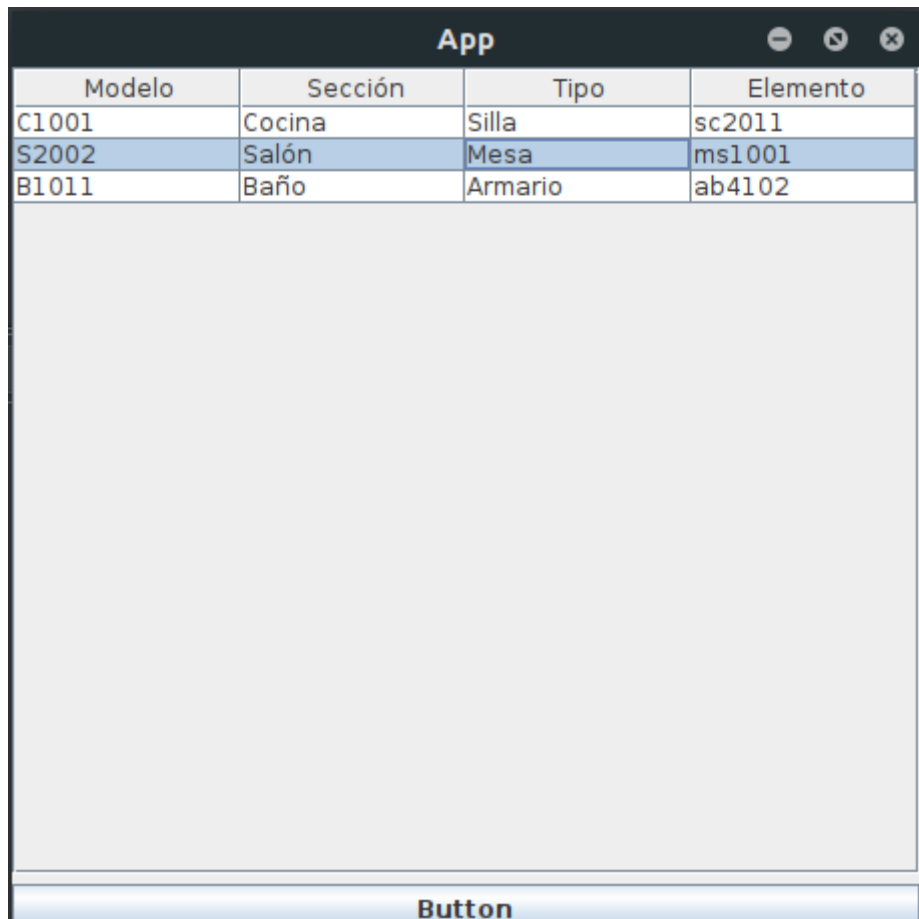
```

Vector<String> columnas = new Vector<>();
columnas.addAll(Arrays.asList(new String[]
{"Modelo", "Sección", "Tipo", "Elemento"}));
Vector<Vector<String>> contenido = new Vector();
contenido.add(new Vector(Arrays.asList(new String[]{"C1001", "Cocina",
"Silla", "sc2011"})));
contenido.add(new Vector(Arrays.asList(new String[]{"S2002", "Salón",
"Mesa", "ms1001"})));
contenido.add(new Vector(Arrays.asList(new String[]{"B1011", "Baño",
"Armario", "ab4102"})));

DefaultTableModel modelo = new DefaultTableModel(contenido, columnas);
tblElementos.setModel(modelo);

```

Es necesario que un componente de tipo **JTable** esté dentro de uno componente **JScrollPane** para que se vean las cabeceras de las columnas.



Modelo	Sección	Tipo	Elemento
C1001	Cocina	Silla	sc2011
S2002	Salón	Mesa	ms1001
B1011	Baño	Armario	ab4102

Button

Por defecto, las celdas son editables. Un cambio sobre el contenido de una celda, afecta a los datos que contiene el modelo. Vamos a ver un ejemplo. Supongamos que cambio el valor **Silla** por **Escritorio**. Si consulto el valor de dicha celda, el valor habrá cambiado. Para consultar el valor de una celda, debemos utilizar el método **getValueAt(fila,columna)** del modelo:

```

modelo.getValueAt(fila,columna)

```

Añadiendo un botón que muestre el contenido de una cierta celda, podemos comprobar esto:

Modelo	Sección	Tipo	Elemento
C1001	Cocina	Escritorio	sc2011
S2002	Salón	Mesa	msl001
B1011	Baño	Armario	ab4102

Mostrar contenido fila 0 columna 2 Valor: Escritorio Cargar datos

El código empleado para esto es el siguiente:

```
btnMostrar.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        int fila = (int) spnFila.getValue();  
        int columna = (int) spnColumna.getValue();  
        txtContenido.setText((String)  
tblElementos.getModel().getValueAt(fila,columna));  
    }  
});
```

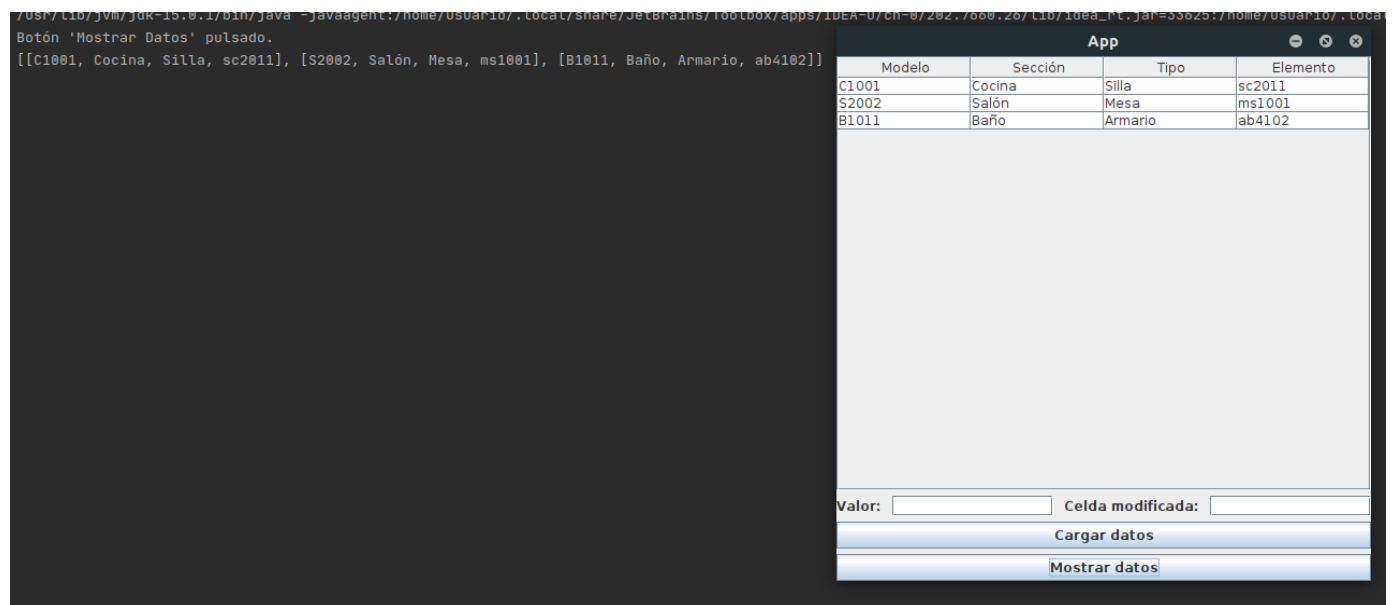
Probablemente queramos automatizar el trabajo sobre los datos de la tabla. Al modificar una celda, se dispara un evento que podemos capturar mediante el listener **TableChanged**. En siguiente ejemplo, se mostrará en el campo de texto "Valor", el contenido de la última celda modificada:


```

        if (column == 1)
            return false;
        else
            return true;
    }
};

```

Para terminar, una vez que quiere recuperar los datos de la tabla, puedo utilizar el método **getDataVector** del modelo.



En ocasiones queremos borrar una fila o insertar una fila. Para insertar una fila, utilizamos el método **addRow** del modelo:

```

modelo.addRow(new Object[]{"", "", "", ""});

```



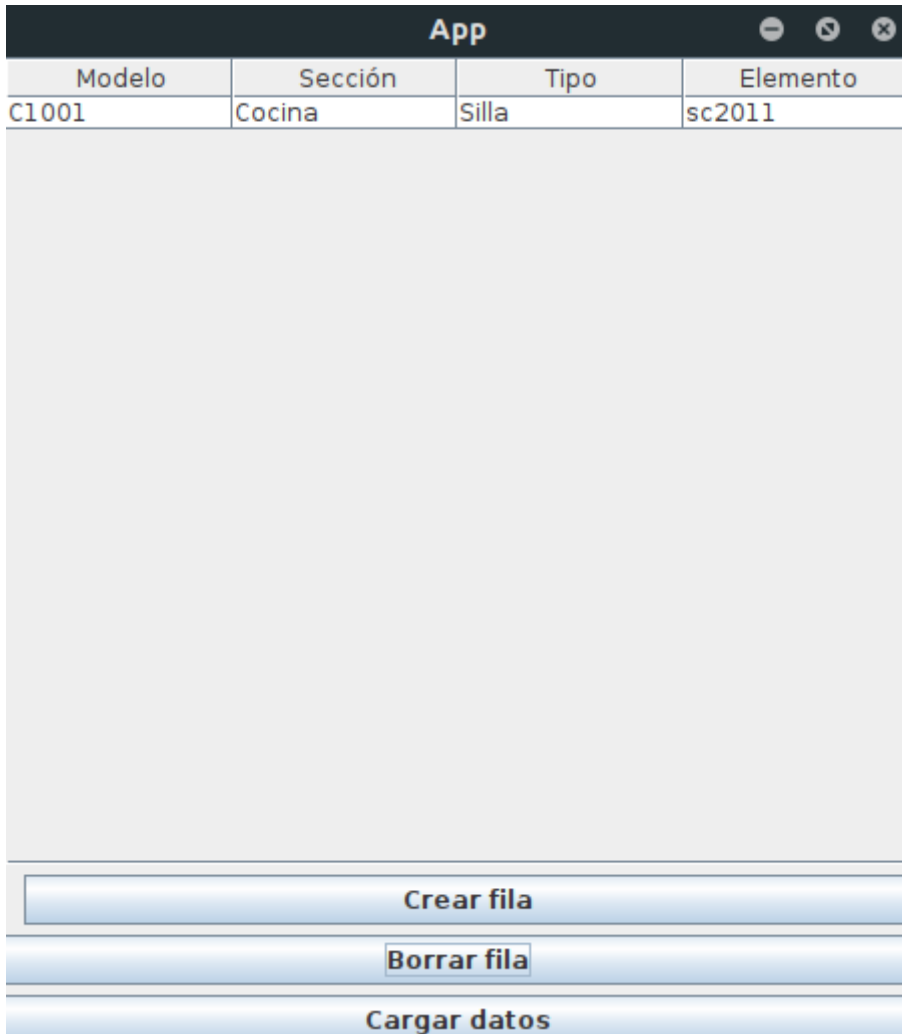
Y para eliminar una fila, debemos en primer conocer la fila/s seleccionada/s mediante el método **getSelectedRows** del objeto **JTable** y a continuación utilizar el método **removeRow**

```

int[] filasSeleccionadas = tblElementos.getSelectedRows();
for (int i = filasSeleccionadas.length-1; i >= 0; i--) {
    System.out.println(filasSeleccionadas[i]);
    modelo.removeRow(filasSeleccionadas[i]);
}

```

Obsérvese que se borran hacia atrás, ya que de otro modo, los índices de las filas posteriores cambian.



Modelo	Sección	Tipo	Elemento
C1001	Cocina	Silla	sc2011

Crear fila

Borrar fila

Cargar datos

Al igual que ocurre con **JList** se puede personalizar más un objeto **JTable**. Por ejemplo, podemos suministrar a la tabla los datos en forma de una lista de nuestros propios objetos. La tabla no entenderá nuestros objetos, pero podemos definir un adaptador, para que traduzca para nuestra tabla la forma de mostrarlos. Para ello debemos construir nuestro propio modelo, como un adaptador de nuestra clase. Este punto tiene mayor complejidad, por lo que se adjunta información en el apéndice **AbstractTableModel**

JDialog

Un diálogo es una ventana lanzada desde otra, y abierta de forma modal, es decir, que hasta cerrarla la ventana llamante queda bloqueada.

Un diálogo es un componente **JDialog**. **JDialog** tiene diferentes constructores, aunque el más común requiere indicar tres cosas:

- El marco (**JFrame**) desde el que se lanza
- El nombre del marco del diálogo
- Si se abrirá de forma modal

```
JFrame marcoPrincipal = (JFrame) SwingUtilities.getRoot((Component)
actionEvent.getSource());
JDialog dialogo = new JDialog(marcoPrincipal, "Crear vendedor", true);
```

Si estamos en un punto en el que no podemos acceder directamente al marco de la ventana que llama al diálogo, podemos obtenerlo utilizando la línea:

```
JFrame marcoPrincipal = (JFrame) SwingUtilities.getRoot((Component)
actionEvent.getSource());
```

Un diálogo necesita componentes. Esto se hace igual que en un marco normal, es decir, mediante un panel con sus componentes:

```
dialogo.add(panelConComponentes1);
```

De entre los componentes añadidos al panel, hay uno especialmente importante: el botón aceptar:

```
JButton boton = new JButton("Aceptar");
boton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // Acciones
        dialogo.dispose();
    }
});
```

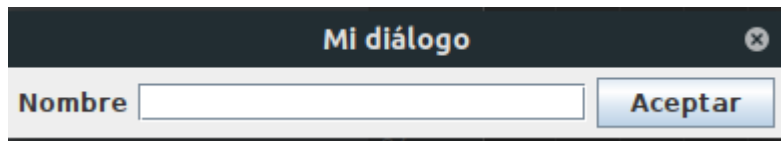
Por ejemplo, supongamos que tenemos una ventana como la siguiente:



En esta ventana hay dos componentes de interés:

- **txtMostrar**: campo de texto no editable
- **btnDialogo**: botón para abrir el diálogo

Al pulsar btnDialogo queremos abrir un diálogo como el siguiente:



En el diálogo hay dos componentes de interés:

- **txtInsertar**: campo de texto editable
- **btnAceptar**: botón para cerrar el diálogo.

En este ejemplo, se crea una ventana de diálogo al pulsar el botón **Abrir diálogo** del panel principal. A continuación se muestra el código en el *listener* del botón de la ventana principal.

```
JFrame marcoPrincipal = (JFrame) SwingUtilities.getRoot((Component)
e.getSource());
JDialog dialogo = new JDialog(marcoPrincipal, "Mi diálogo", true);

// Creación de un panel para el diálogo
JPanel panel = new JPanel();

// Creación de componentes para el diálogo
JLabel lblNombre = new JLabel();
lblNombre.setText("Nombre");

JTextField txtNombre = new JTextField(20);

JButton boton = new JButton("Aceptar");
boton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // Insertar en el campo "txtMostrar" de la ventana padr
        // el valor del campo "txtInsertar" del diálogo
        txtMostrar.setText(lblNombre.getText());
        // Con la siguiente línea el diálogo desaparecerá
        // y la ventana principal recuperará su control
        dialogo.dispose();
    }
});

// Añadido de los componentes al panel
panel.add(lblNombre);
panel.add(txtNombre);
```

```
panel.add(boton);

// Añadido del panel al diálogo y mostrado
dialogo.add(panel);
dialogo.pack();
// En este punto la ventana padre queda bloqueada
dialogo.setVisible(true);
txtDesdeDialogo.setText(txtNombre.getText());
```

Observa esta línea:

```
txtMostrar.setText(lblNombre.getText());
```

En ella se accede a los campos de la ventana principal y el diálogo a la vez, gracias a que están en el mismo ámbito. Pero no es tan fácil cuando la ventana principal y el diálogo están en clases diferentes (tal y como se hace en IntelliJ).

JDialog con IntelliJ

Cuando creamos un diálogo con IntelliJ, el código generado es como el siguiente:

```
public class Dialogo extends JDialog {
    private JPanel contentPane;
    private JButton buttonOK;
    private JButton buttonCancel;

    public Dialogo() {
        setContentPane(contentPane);
        setModal(true);
        getRootPane().setDefaultButton(buttonOK);

        buttonOK.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                onOK();
            }
        });

        buttonCancel.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                onCancel();
            }
        });

        // call onCancel() when cross is clicked
        setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
```

```

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                onCancel();
            }
        });

        // call onCancel() on ESCAPE
        contentPane.registerKeyboardAction(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                onCancel();
            }
        }, KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, 0),
        JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);

        // Añadidos manualmente
        dialog.pack();
        dialog.setVisible(true);

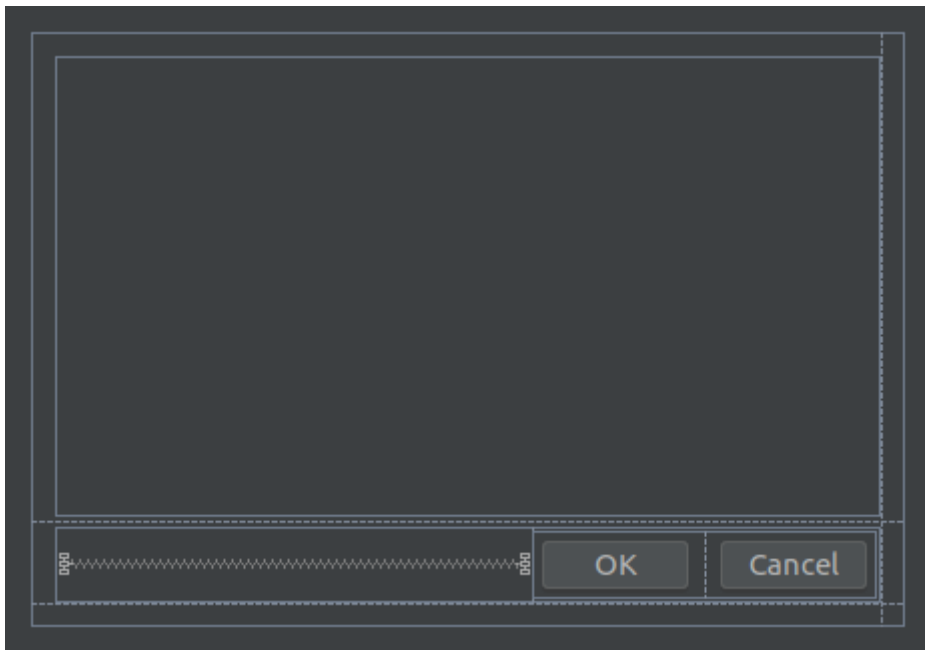
    }

    private void onOK() {
        // add your code here
        dispose();
    }

    private void onCancel() {
        // add your code here if necessary
        dispose();
    }
}

```

En el diálogo podemos añadir los componentes que deseemos, igual que en un formulario.



Las acciones que queramos que haga el diálogo, debemos ubicarlas en el método **onOK**

El problema del ámbito

Desde el diálogo no podemos modificar los campos de la ventana principal. Por eso necesitamos un mecanismo para transmitir la información desde el diálogo hasta la ventana principal.

Existen diferentes alternativas:

Método 1. Pasar la ventana como parámetro al diálogo

En el constructor de la ventana principal:

```
this.estaVentana = this;
```

En la creación del diálogo en la ventana principal:

```
Dialogo dialogo = new Dialogo(estaVentana);
```

En la ventana principal tenemos el siguiente método:

```
public void hacerAlgo(String valor){  
    txtDesdeDialogo.setText(valor);  
}
```

En el método **onOk** del diálogo:

```
private void onOK() {  
    this.ventanaPrincipal.hacerAlgo(txtInsertar.getText());  
    dispose();  
}
```

Método 2. Solicitar la información al diálogo antes de que se cierre.

Otro método para pasar la información desde el diálogo hasta la ventana principal es utilizar un método **get** en el diálogo.

En la ventana principal, el uso del diálogo es como sigue:

```
Dialogo dialogo = new Dialogo();
dialogo.pack();
// Después de la linea siguiente, la ventana principal queda
bloqueada, y el flujo de ejecución detenido
dialogo.setVisible(true);
// Llegamos a este punto sólo cuando el diálogo no es visible
(setVisible(false))
// El diálogo no es visible pero sigue activo, y podemos llamar a sus
métodos
if (dialogo.pulsadoOk())
    txtDesdeDialogo.setText(dialogo.getTexto());
dialogo.dispose();
```

En el diálogo, los métodos **onOK** y **onCancel** son como siguen:

```
private void onOK() {
    // pulsadoOk es un atributo del diálogo inicializado a false
    pulsadoOk = true;
    setVisible(false);
}

private void onCancel() {
    // add your code here if necessary
    setVisible(false);
}
```

En el diálogo definimos los siguientes métodos:

```
public boolean pusladoOk(){
    return pulsadoOk;
}

public String getTexto(){
    return txtInsertar.getText();
}
```

Método 3. Usando un objeto donde almacenar los datos

Pasando un mapa como objeto para tranferir datos:

En la ventana principal, el uso del diálogo es como sigue:

```
Map<String,String> datos = new HashMap<>();
Dialogo dialogo = new Dialogo(datos);
if (datos.containsKey("texto"))
    txtDesdeDialogo.setText(datos.get("texto"));
```

En el constructor del diálogo, hace falta almacenar el mapa:

```
this.datos = datos;
```

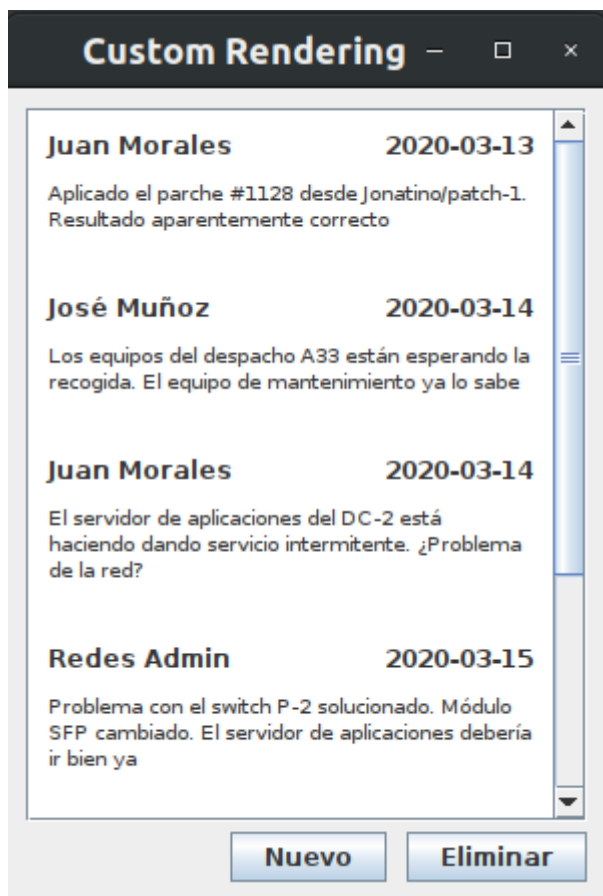
Los métodos **onOK** y **onCancel** del diálogo son:

```
private void onOK() {
    datos.put("texto",txtInsertar.getText());
    dispose();
}

private void onCancel() {
    // add your code here if necessary
    dispose();
}
```

Apéndice: JList CustomRenderer

Supongamos que queremos mostrar un cronograma con las entradas realizadas por diferentes usuarios en un sistema de ticket de un servicio de soport. El esquema de la aplicación queda descrito con la siguiente imagen.



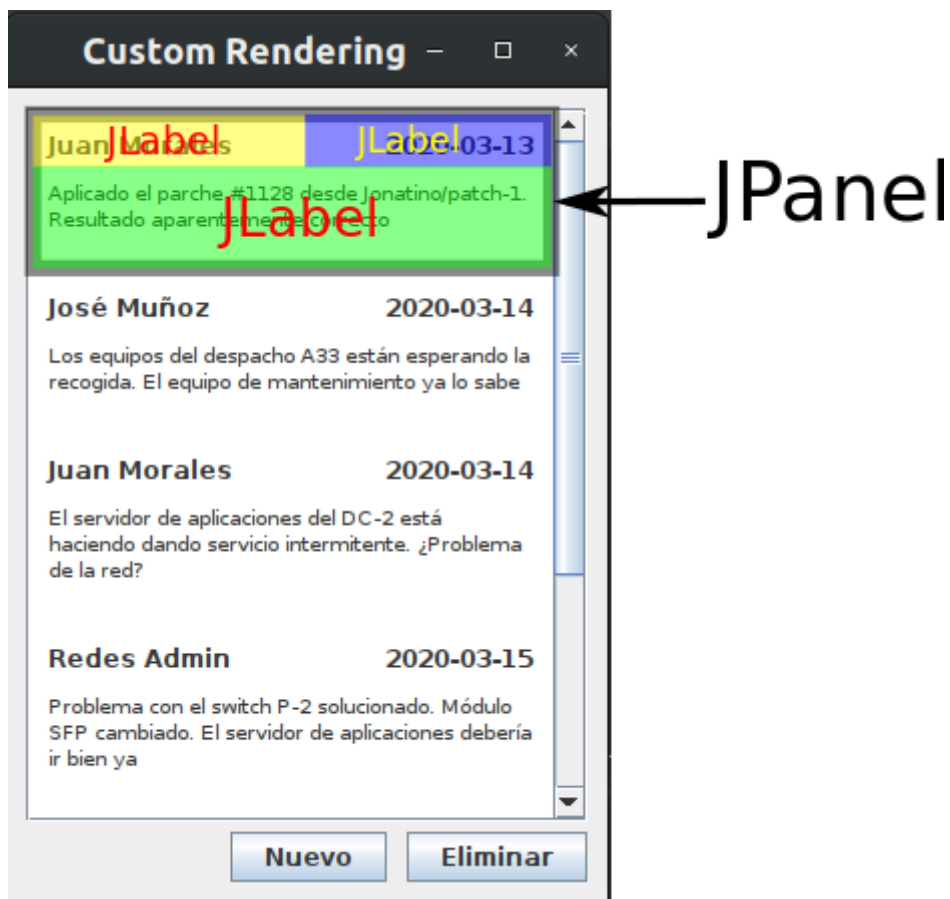
En la imagen se puede ver una lista de entradas, que se corresponden con una clase llamada Item. La clase Item tiene tres atributos:

- autor, de tipo String.
- fecha, de tipo LocalDate.
- texto, de tipo String.

El componente utilizado en la captura es un JList. A priori, un JList sólo puede mostrar Strings. Sin embargo hay una forma de mostrar componentes más complejos, como un JPanel, o cualquier otra cosa. Este comportamiento se consigue mediante lo que se denomina un Renderer.

En el ejemplo actual, se ha implementado un renderer para poder mostrar un componente JPanel que contiene tres componentes JLabel (utilizados para mostrar todo el contenido de un Item).

En la siguiente imagen se puede ver la estructura del componente utilizado.



- Se debe elegir el componente en función de la información a mostrar. Por ejemplo, si sólo se va a mostrar texto, se puede utilizar un simple JLabel.

El componente JPanel debe implementar la interfaz **ListCellRenderer**, e indicar mediante el método **getListCellRendererComponent** de qué forma se mostrará un Item, lo que viene a ser, asignar a cada JLabel su valor.

Item.java (entidad a mostrar en la lista)

```
package com.politecnicomalaga;

import java.time.LocalDate;

public class Item {
    private String autor;
    private LocalDate fecha;
    private String texto;

    public Item(String autor, LocalDate fecha, String texto) {
        this.autor = autor;
        this.fecha = fecha;
        this.texto = texto;
    }

    public String getAutor() {
```

```

        return autor;
    }

    public void setAutor(String autor) {
        this.autor = autor;
    }

    public LocalDate getFecha() {
        return fecha;
    }

    public void setFecha(LocalDate fecha) {
        this.fecha = fecha;
    }

    public String getTexto() {
        return texto;
    }

    public void setTexto(String texto) {
        this.texto = texto;
    }
}

```

VentanaPrincipal.java (ventana donde está el JList)

```

package com.politecnicomalaga;

import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.*;
import java.awt.event.ComponentAdapter;
import java.awt.event.ComponentEvent;
import java.time.LocalDate;

public class VentanaPrincipal {
    private JPanel pnlMain;
    private JButton btnEliminar;
    private JButton btnNuevo;
    private JList lstItems;

    public VentanaPrincipal(){
        Item item1 = new Item("Juan

```

```

Morales", LocalDate.of(2020, 3, 13), "Aplicado el parche #1128 desde
Jonatino/patch-1. Resultado aparentemente correcto");

Item item2 = new Item("José Muñoz", LocalDate.of(2020, 3, 14), "Los
equipos del despacho A33 están esperando la recogida. El equipo de
mantenimiento ya lo sabe");

Item item3 = new Item("Juan Morales", LocalDate.of(2020, 3, 14), "El
servidor de aplicaciones del DC-2 está haciendo dando servicio
intermitente. ¿Problema de la red?");

Item item4 = new Item("Redes
Admin", LocalDate.of(2020, 3, 15), "Problema con el switch P-2 solucionado.
Módulo SFP cambiado. El servidor de aplicaciones debería ir bien ya");

Item item5 = new Item("Juan
Morales", LocalDate.of(2020, 3, 16), "Necesitamos un servidor de despliegue
para la aplicación de mensajería.");

Item item6 = new Item("José Muñoz", LocalDate.of(2020, 3, 16), "¿Qué
configuración debería tener el servidor de despliegue para la aplicación
de mensajería? Enviar a admin@seragul.es");


DefaultListModel<Item> lstModel = new DefaultListModel<>();
lstModel.addElement(item1);
lstModel.addElement(item2);
lstModel.addElement(item3);
lstModel.addElement(item4);
lstModel.addElement(item5);
lstModel.addElement(item6);


lstItems.setModel(lstModel);
lstItems.setCellRenderer(new ItemPanel());


lstItems.setFixedCellWidth(pnlMain.getWidth());

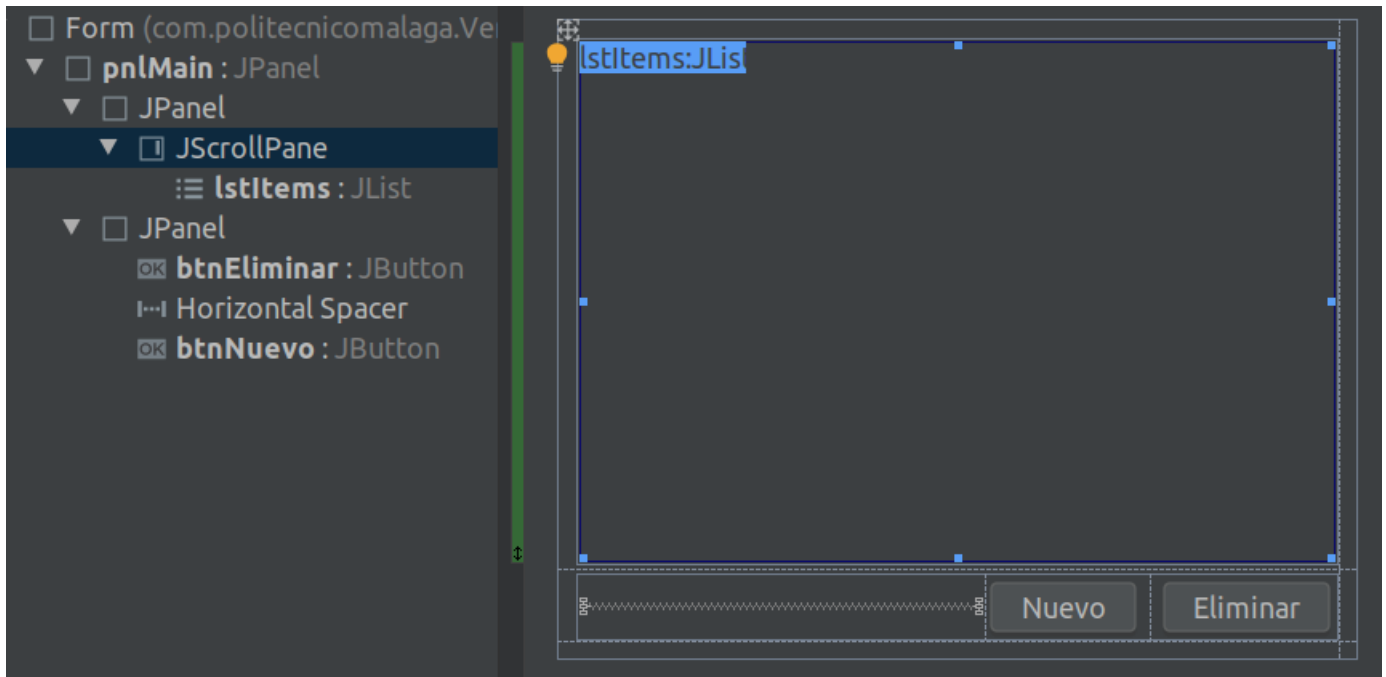

lstItems.addComponentListener(new ComponentAdapter() {
    @Override
    public void componentResized(ComponentEvent e) {
        // Estiramos la celda a 1000 px de altura
        lstItems.setFixedCellHeight(1000);
        // Asignamos el valor -1 para que se recalcule la altura
        lstItems.setFixedCellHeight(-1);
    }
});
}

```

```

public Container getPanel() {
    return pnlMain;
}
}

```



ItemPanel.java (JPanel y Custom Renderer)

```

package com.politecnicomalaga;

import javax.swing.*.*;
import javax.swing.border.Border;
import javax.swing.border.EmptyBorder;
import javax.swing.table.TableCellRenderer;
import java.awt.*.*;

public class ItemPanel extends JPanel implements ListCellRenderer<Item> {

    private JLabel lblAutor;
    private JLabel lblFecha;
    private JTextArea txtAreaTexto;
    public ItemPanel(){
        //setMinimumSize(new Dimension(100,100));
        setLayout(new BorderLayout(10,10));

        lblAutor = new JLabel();
        lblFecha = new JLabel();
        txtAreaTexto = new JTextArea();
        txtAreaTexto.setWrapStyleWord(true);
    }
}

```

```

txtAreaTexto.setLineWrap(true);

Border margenAutorFecha = new EmptyBorder(10,10,0,10);
Border margenTexto = new EmptyBorder(0,10,20,10);
lblAutor.setBorder(margenAutorFecha);
lblFecha.setBorder(margenAutorFecha);
txtAreaTexto.setBorder(margenTexto);

add(lblAutor, BorderLayout.WEST);
add(lblFecha, BorderLayout.EAST);
add(txtAreaTexto, BorderLayout.SOUTH);
}

@Override
public Component getListCellRendererComponent(JList<? extends Item>
jList, Item item, int i, boolean isSelected, boolean cellHasFocus) {

    lblAutor.setText(item.getAutor());
    lblFecha.setText(item.getFecha().toString());
    txtAreaTexto.setText(item.getTexto());
    txtAreaTexto.setSize(jList.getWidth(),Short.MAX_VALUE);

    if (isSelected){
        setBackground(Color.LIGHT_GRAY);
    } else {
        setBackground(Color.WHITE);
    }

    if (cellHasFocus){
        txtAreaTexto.setFont(new Font("SansSerif",Font.BOLD,14));
    } else {
        txtAreaTexto.setFont(new Font("SansSerif",Font.PLAIN,10));
    }

    return this;
}
}

```

App.java

```

package com.politecnicomalaga;

import javax.swing.*.*;

```

```

import java.awt.*;

/**
 * Hello world!
 */
public class App
{
    public static void main( String[] args )
    {
        JFrame frame = new JFrame("Custom Rendering");
        frame.setContentPane(new VentanaPrincipal().getPanel());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Toolkit toolkit = Toolkit.getDefaultToolkit();
        int height = toolkit.getScreenSize().height - 200;
        int width = 300;
        frame.setPreferredSize(new Dimension(width, height));
        frame.setMinimumSize(new Dimension(width,200));
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}

```

AbstractTableModel

AbstractTableModel es un adaptador (ver patrón *Adapter*), donde el cliente es un objeto de tipo `JTable`, y el objeto adaptado es el objeto cuya información se muestra en la tabla.

Aclaración sobre los términos usados en este texto:

- modelo : datos correspondientes a la lógica de negocio, es decir, los objetos cuyos datos se muestran en la tabla.
- adaptador : implementación que hacemos de la clase `AbstractTableModel`.
- tabla : objeto de tipo `JTable` donde se muestran los datos del modelo.

Métodos abstractos de *AbstractTableModel*

Según la documentación de **Oracle**, esta clase tiene tres métodos abstractos (y muchos otros implementados). Estos tres métodos son:

- `public int getRowCount();` devuelve el número de filas que debe mostrar la tabla. Si se está mostrando una lista de elementos, entonces este método devolverá la longitud de la lista.

- `public int getColumnCount();`: devuelve el número de columnas que debe tener la tabla. Lo más recomendable es tener en nuestra clase *adaptador* un array estático con el nombre de las columnas, y devolver la longitud de dicho array.
- `public Object getValueAt(int row, int column);`: devuelve el valor correspondiente a la celda (*row*,*column*).

Además de estos métodos, es recomendable sobrescribir los siguientes:

- [`String getColumnName\(int column\)`](#): devuelve el nombre de la columna número *column*
- [`Class<?> getColumnClass\(int columnIndex\)`](#): devuelve la clase de la columna número *column*
- [`boolean isCellEditable\(int rowIndex, int columnIndex\)`](#): devuelve true cuando queremos que una celda de la tabla (cuya fila y columna son *rowIndex* y *columnIndex*) se pueda editar. En caso contrario devuelve false.
- [`void setValueAt\(Object aValue, int rowIndex, int columnIndex\)`](#): este método está vacío inicialmente, y sólo es necesario implementarlo cuando hay celdas editables en la tabla. Este método se puede utilizar para actualizar los datos del modelo desde la tabla. La idea es la siguiente:
 - Paso 1: se modifica el contenido de una celda en la tabla (JTable) de la ventana. Entonces JTable llama a `setValueAt`.
 - Paso 2: el código dentro de `setValueAt` modifica los datos del modelo.
 - Paso 3: al final de `setValueAt` se debe ejecutar el método `fireTableCellUpdated(rowIndex, columnIndex)`, para hacer saber a la tabla que debe refrescar el contenido de dicha fila/columna.

Además de los métodos anteriores, puede ser útil añadir un método más al adaptador. Este método no está declarado en la clase *AbstractTableModel*:

- **`public void removeRow(int row)`**: este método sirve para modificar el modelo desde la tabla. Su funcionamiento es el siguiente:
 - Paso 1: Cuando se desea eliminar una fila de la tabla (y por tanto en el modelo), se ejecuta el método `removeRow` del adaptador.
 - Paso 2: el método `removeRow` elimina el elemento correspondiente del modelo.
 - Paso 3. finalmente ejecuta el método `fireTableRowsDeleted(row, row)`, para que la tabla refresque el contenido mostrado.

NOTA: El borrado de una fila de la tabla se puede hacer de varias formas:

1. Con un botón borrar: El listener asociado botón ejecuta el método `removeRow` del adaptador.
2. Con un listener para la tabla (ejemplo):

```
tabla.addKeyListener(new KeyAdapter() {
    @Override
    public void keyReleased(KeyEvent e) {
        if (e.getKeyCode() == KeyEvent.VK_DELETE){
```

```
        int confirmaBorrar = JOptionPane.showConfirmDialog( pnlMain,
                                                                "¿Desea borrar la
fila?",
                                                                "Borrado de un
elemento",JOptionPane.WARNING_MESSAGE);
        if (confirmaBorrar==JOptionPane.OK_OPTION){
            tabla.getCellEditor().stopCellEditing(); // evita errores
de borrado (celda bloqueada por edición)
            adaptador.removeRow(tblPersona.getSelectedRow());
        }
    }
}
});
```