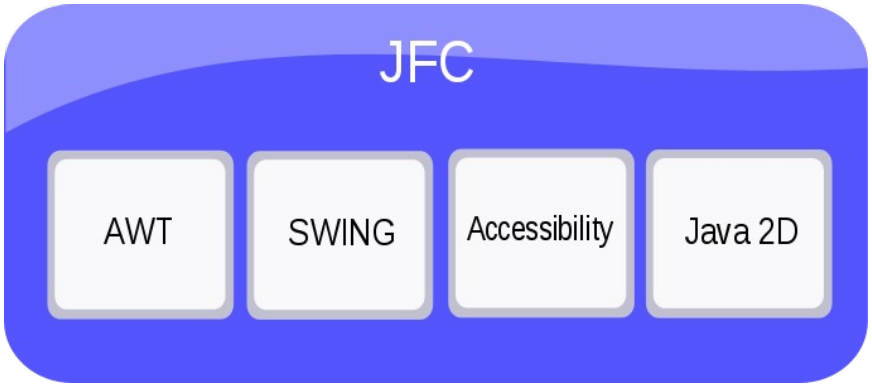
UD9: Desarrollo de interfaces gráficas de usuario.

IES LOS SAUCES – BENAVENTE
CFGS DESARROLLO DE APLICACIONES WEB
PROGRAMACIÓN

Swing

Swing es una biblioteca gráfica para Java.

Pertenece a las JFC (Java Foundation Classes): AWT(Abstract Window Toolkit), Swing y Java 2D.



Está contenida en el paquete *javax.swing*

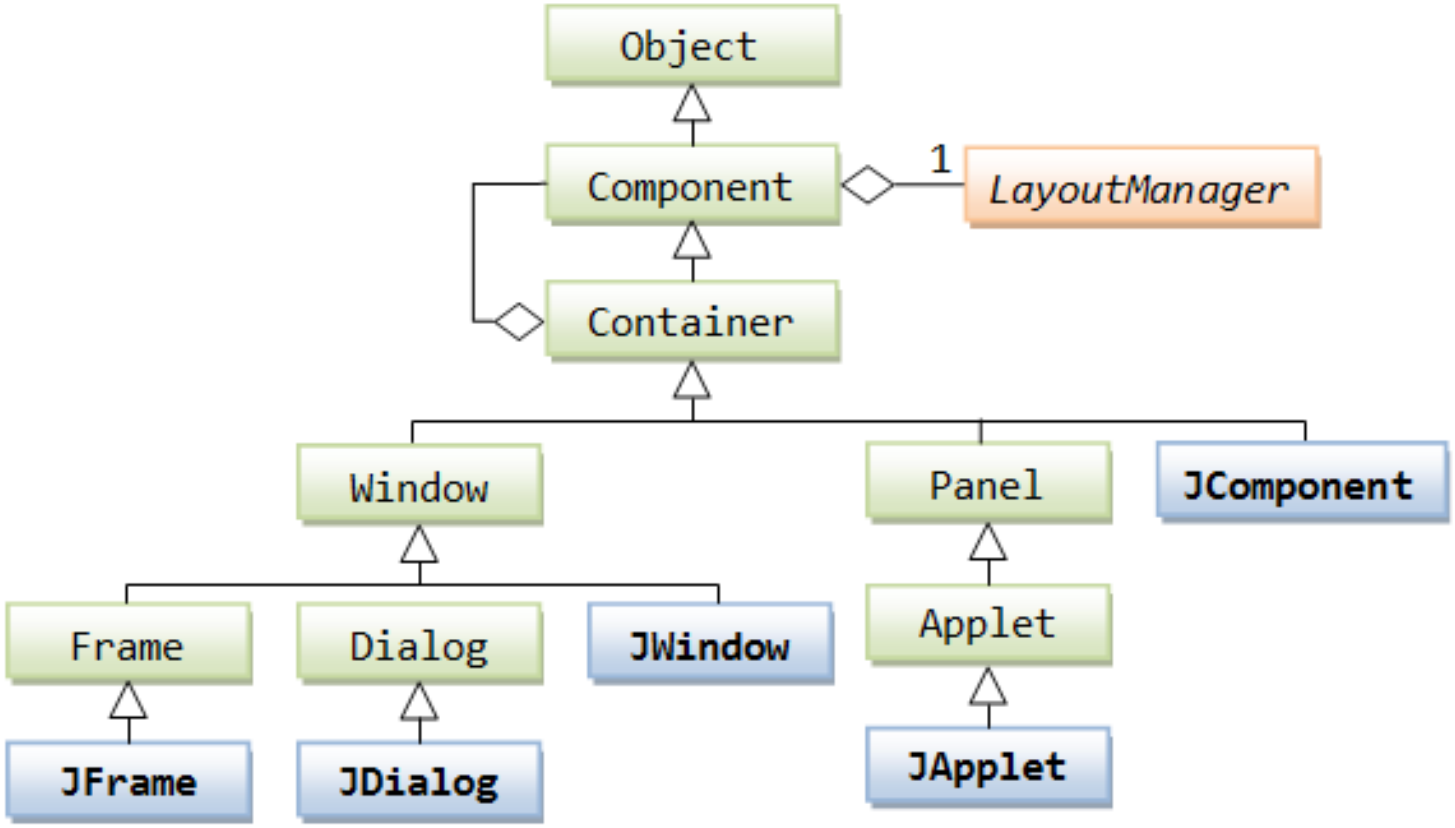
Creada a partir del paquete *java.awt*

Swing

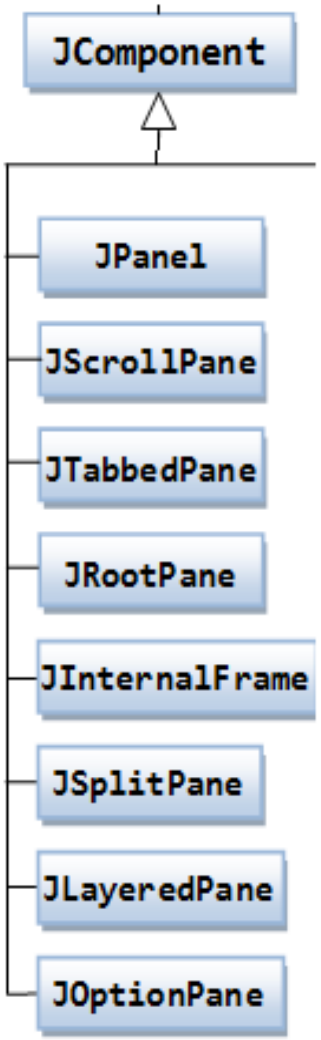
Componentes y contenedores

- Los componentes son el aspecto visible de la interfaz (botones, etiquetas, campos de texto, etc...).
- Se deben situar dentro de algún contenedor.
- Los contenedores almacenan los componentes.
- Los contenedores pueden ser de dos tipos: superiores e intermedios.

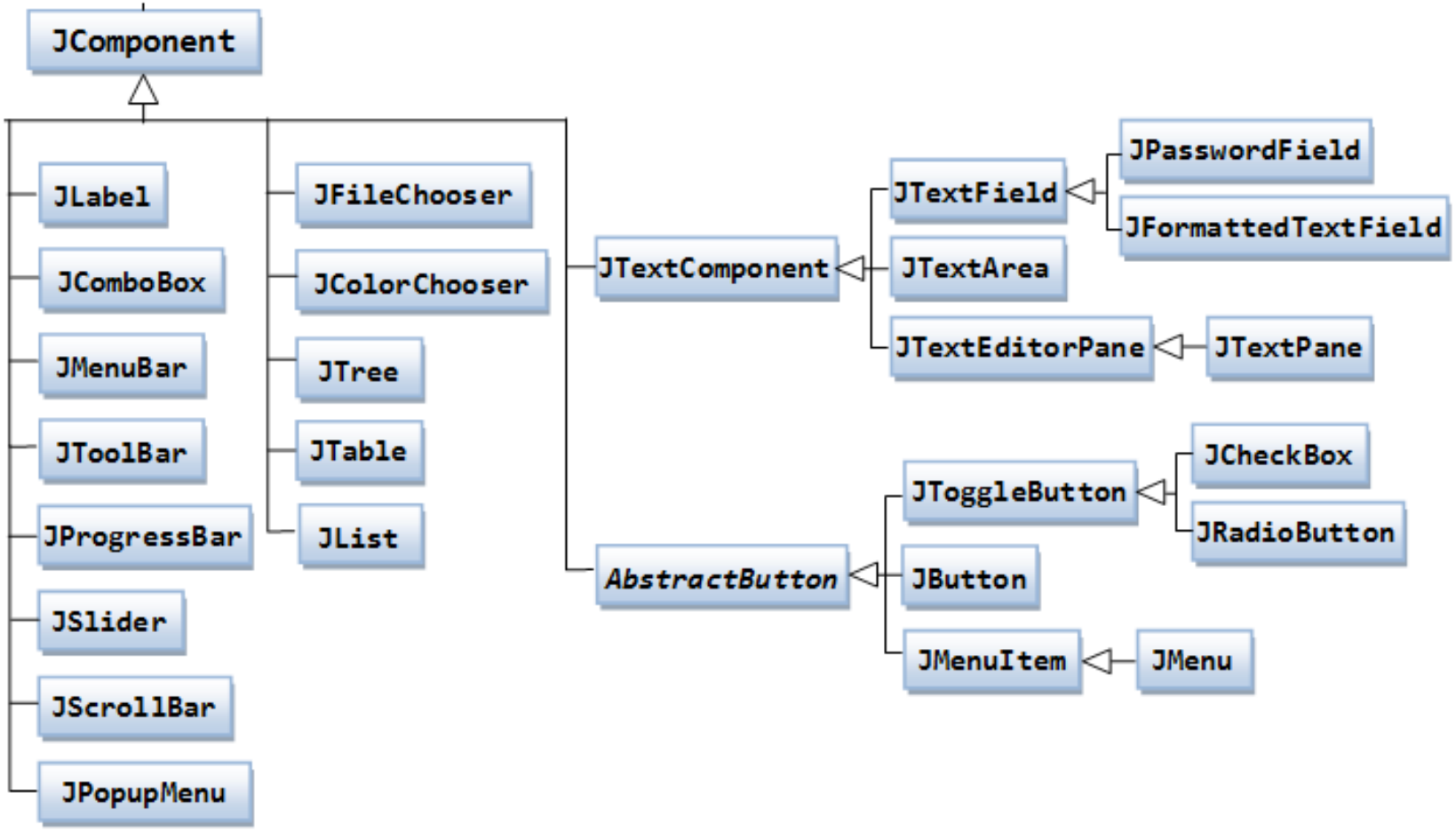
Contenedores superiores



Contenedores intermedios



Componentes



Pasos a seguir

1. Crear un contenedor superior y obtener o fijar un contenedor intermedio
2. Seleccionar un gestor de diseño para el contenedor intermedio
3. Crear los componentes adecuados
4. Añadir los componentes al contenedor intermedio
5. Dimensionar el contenedor superior
6. Mostrar el contenedor superior

Swing

Crear contenedor superior

```
import javax.swing.*;
public class MiVentana extends JFrame{
   public MiVentana(){
      super("Mi ventana");
      setDefaultCloseOperation(EXIT_ON_CLOSE);
   }
}
```

Una vez creado el contenedor superior deberemos obtener o fijar un contenedor intermedio.

Obtener contenedor intermedio

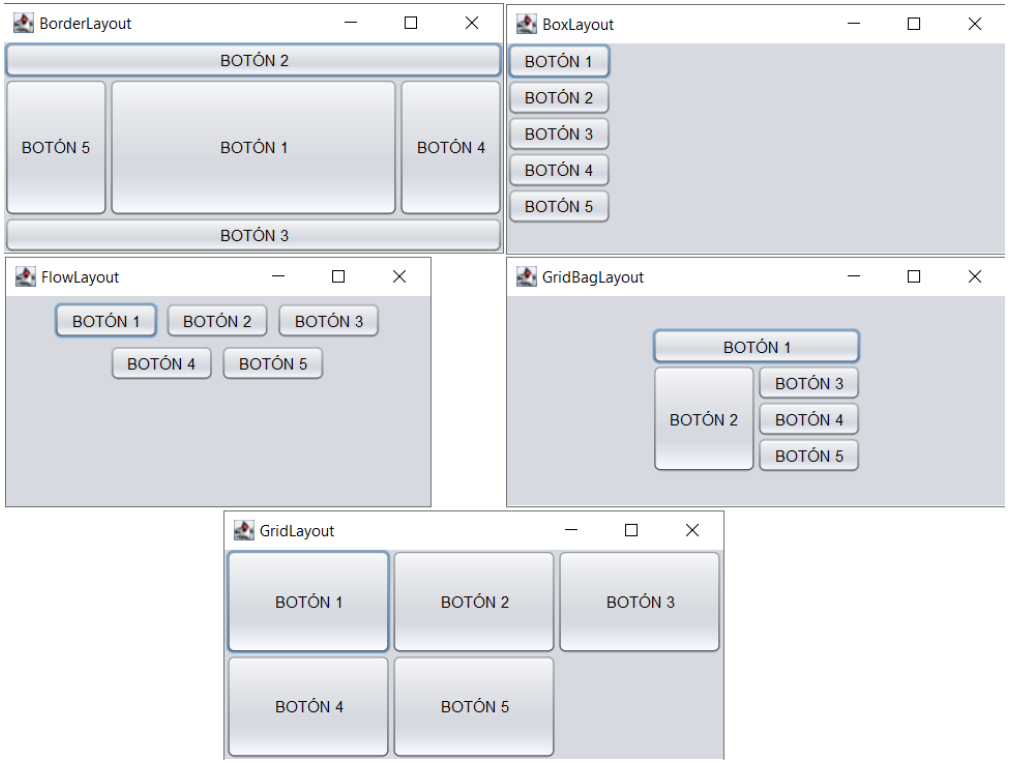
```
import javax.swing.*;
import java.awt.*;
public class MiVentana extends Jframe{
   private Container cp;
   public MiVentana(){
      super("Mi ventana");
      cp=getContentPane();
      setDefaultCloseOperation(EXIT_ON_CLOSE);
   }
}
```

Fijar contenedor intermedio

```
import javax.swing.*;
import java.awt.*;
public class MiVentana extends Jframe{
   private JPanel panel;
   public MiVentana(){
      super("Mi ventana");
      panel=new JPanel();
      setContentPane(panel);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
   }
}
```

Seleccionar un gestor de diseño

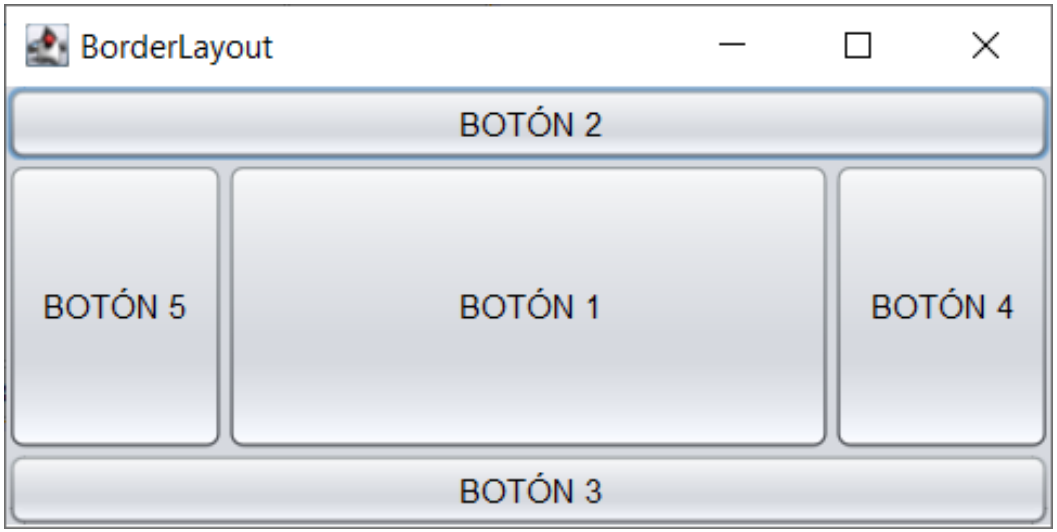
La distribución de los componentes en el contenedor depende del gestor de diseño seleccionado.



BorderLayout

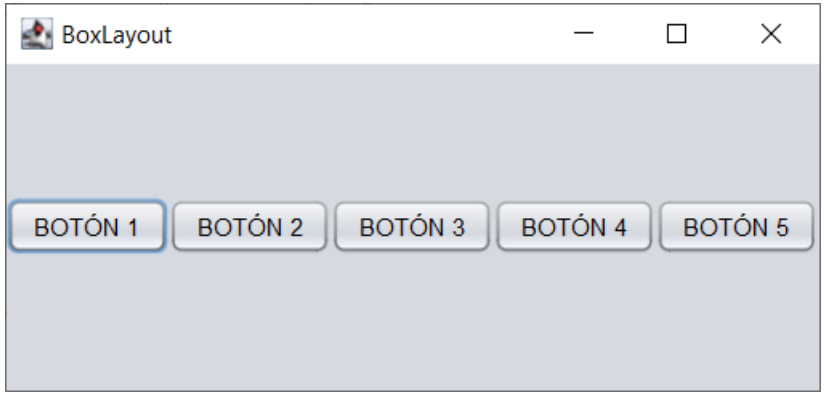
Distribuye los componentes en cinco zonas: norte, sur, este, oeste y centro.

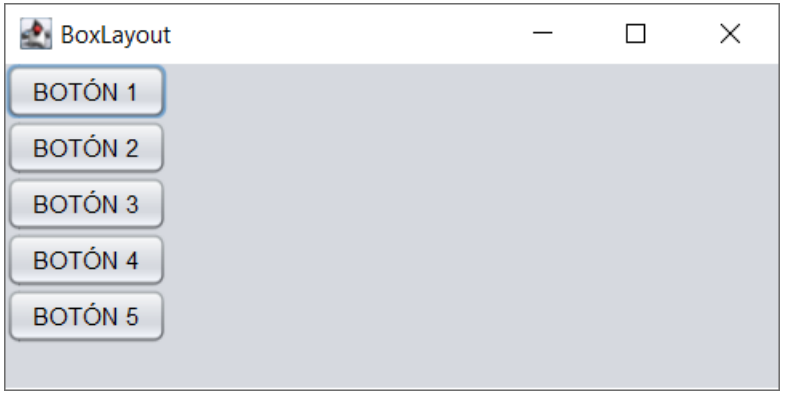
Si a alguna de estas áreas no se le asigna un componente, el área vacía será ocupada por alguna de las áreas restantes.



BoxLayout

Distribuye los componentes en una única fila o columna.

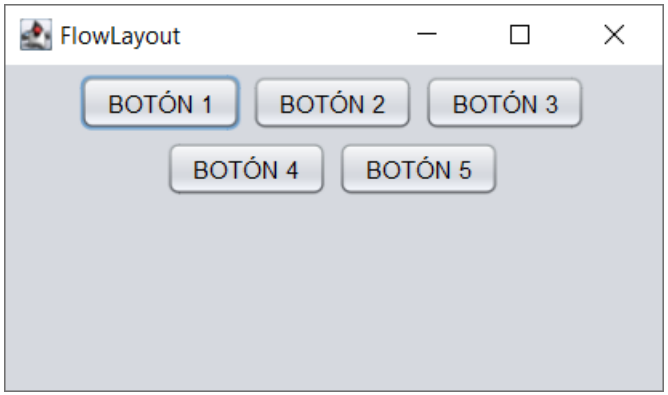




FlowLayout

Sitúa los componentes por filas de izquierda a derecha, cuando completa una fila comienza otra.

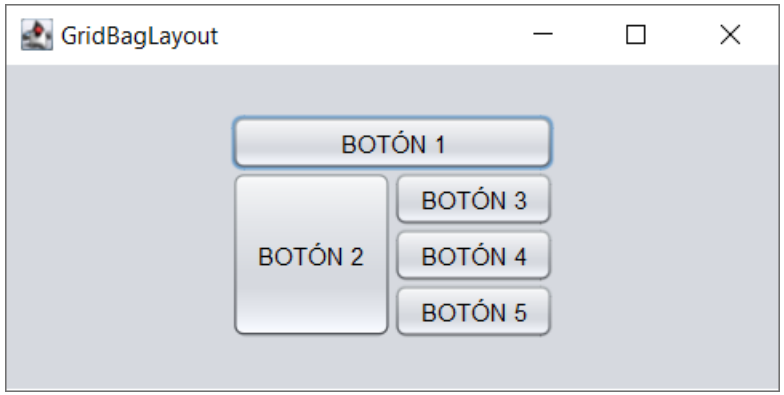
Si se modifica el tamaño del contenedor los componentes se redistribuyen.



GridBagLayout

Divide el contenedor en una rejilla de celdas que pueden ser de distintos tamaños.

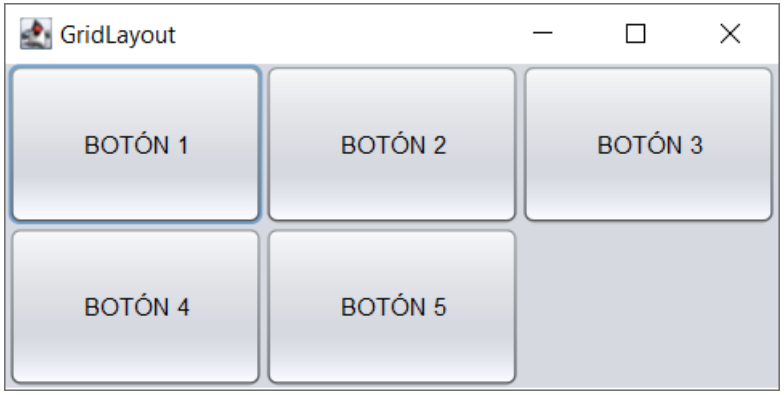
Cada componente puede ocupar una celda entera, una parte de una celda o varias celdas.



GridLayout

Distribuye los componentes en una rejilla de celdas iguales, haciendo que cada componente utilice una celda de la rejilla (se ubican de arriba abajo y de izquierda a derecha).

Debemos indicar el número de filas y columnas que va a presentar.



Crear los componentes adecuados

```
import javax.swing.*;
import java.awt.*;
public class MiVentana extends Jframe{
    private Panel panel;
    private JButton boton;
    public MiVentana(){
        super("Mi ventana");
        panel=new Jpanel();
        setContentPane(panel);
        boton=new JButton(“ACEPTAR”);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}
```

Añadir los componentes al contenedor

```
import javax.swing.*;
import java.awt.*;
public class MiVentana extends Jframe{
    private Panel panel;
    private JButton boton;
    public MiVentana(){
        super("Mi ventana");
        panel=new Jpanel();
        setContentPane(panel);
        boton=new Jbutton(“ACEPTAR”);
        panel.add(boton);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}
```

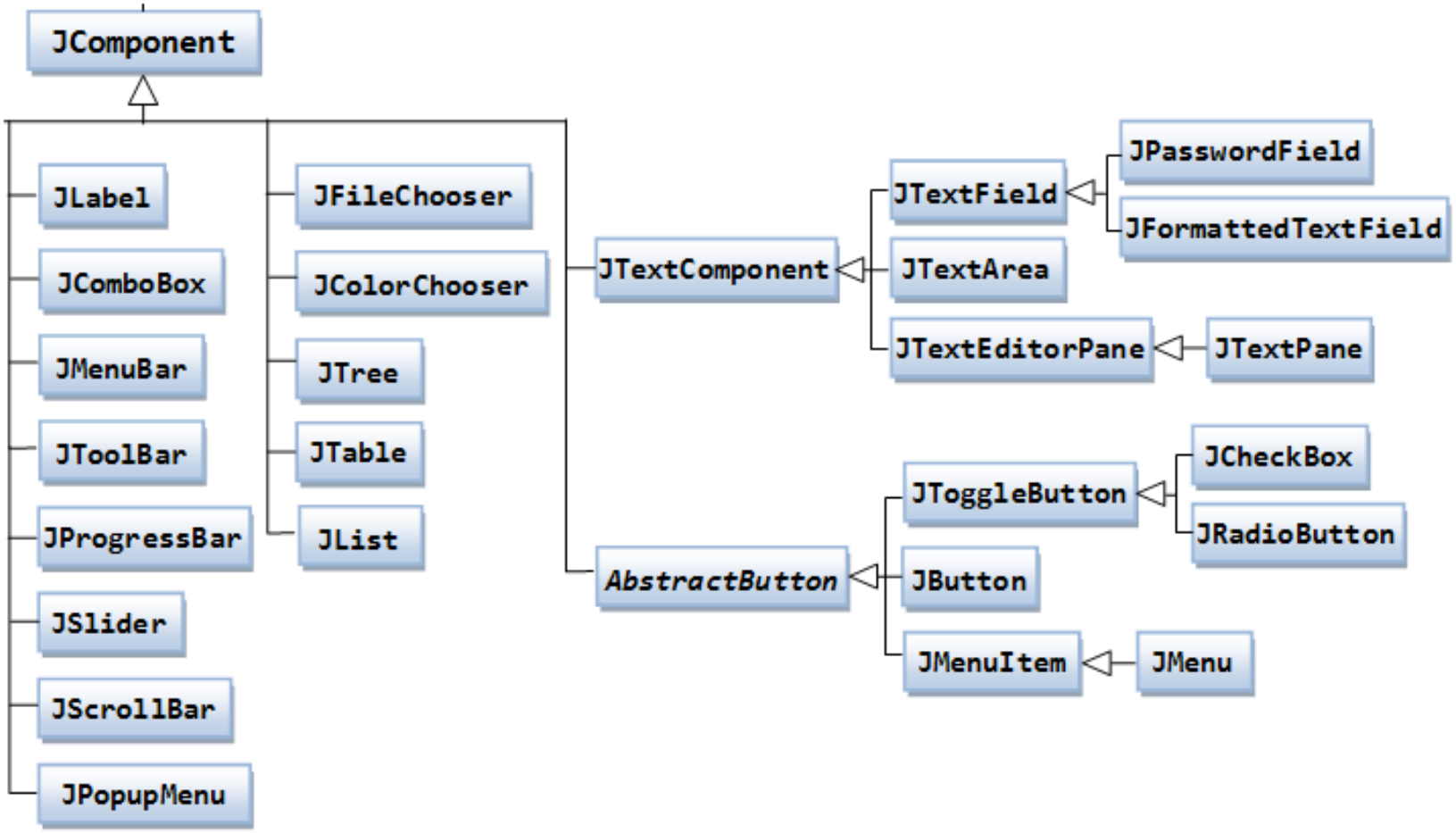
Dimensionar el contenedor superior

```
public class AppVentana{
   public static void main(String... args){
      MiVentana ventana=new MiVentana();
      ventana.setSize(300,200);
   }
}
```

Mostrar el contenedor superior

```
public class AppVentana{
   public static void main(String... args){
      MiVentana ventana=new MiVentana();
      ventana.setSize(300,200);
      ventana.setVisible(true);
   }
}
```

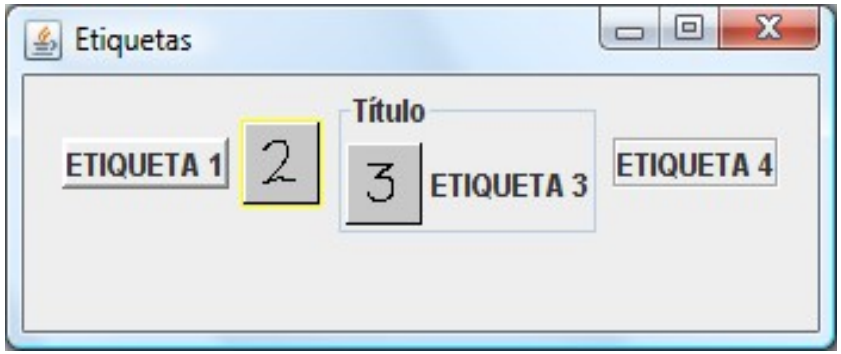
Componentes



Swing

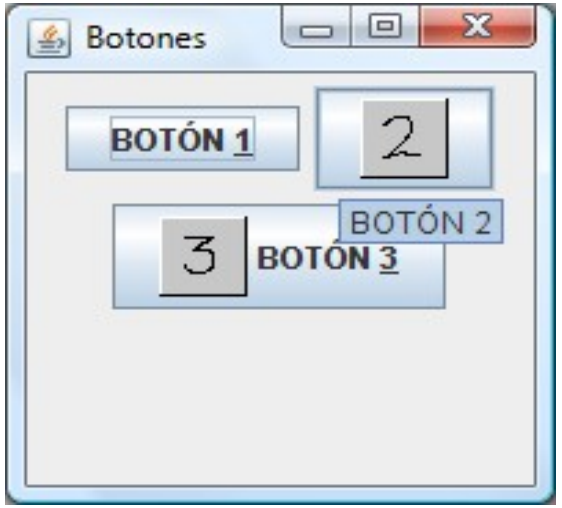
JLabel

Permite mostrar información al usuario mediante texto, imágenes o ambos elementos.



JButton

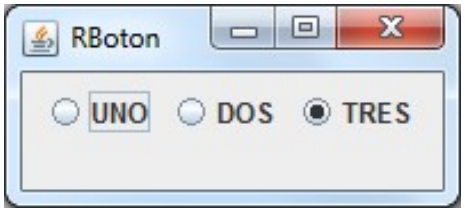
Un botón puede contener texto, imágenes o ambos elementos.



JRadioButton

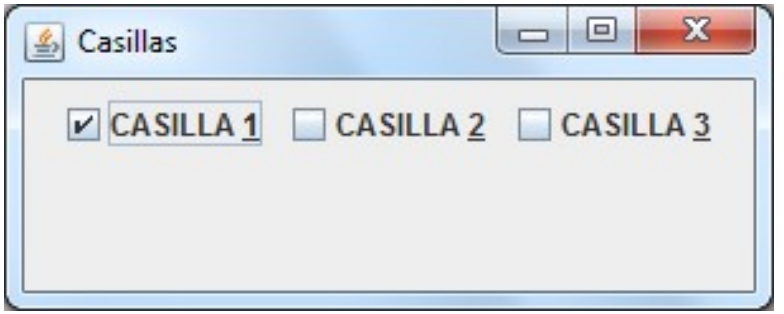
Los botones de opción se suelen encontrar en grupos en los que, por convención, únicamente uno de los botones puede encontrarse seleccionado a un mismo tiempo.

Para agrupar los botones se utiliza la clase ButtonGroup. Crearemos un objeto de la clase ButtonGroup y añadiremos los botones al grupo.



JCheckBox

Las casillas de verificación se suelen agrupar y es posible seleccionar, una, algunas o ninguna de ellas.



JComboBox

Representa una lista desplegable de opciones, que puede ser editable o no.

Cuando el usuario pulsa la lista, el objeto JComboBox muestra un menú de elementos para elegir.

JList

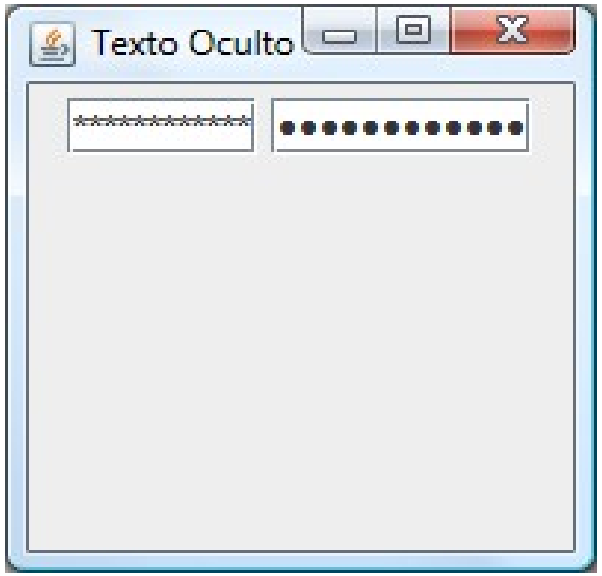
Muestra una lista de elementos para su selección.

JTextField

Permite editar textos en una línea.

JPasswordField

Permite editar textos en una línea. Enmascara el eco.

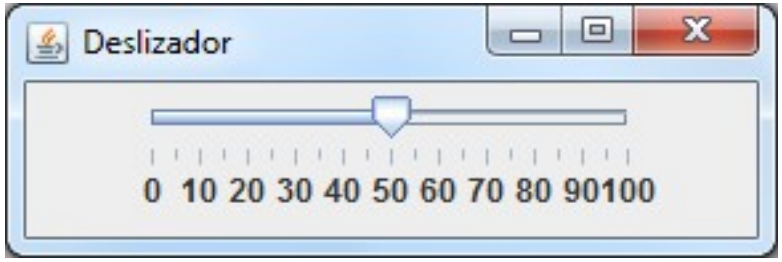


Swing

JSlider

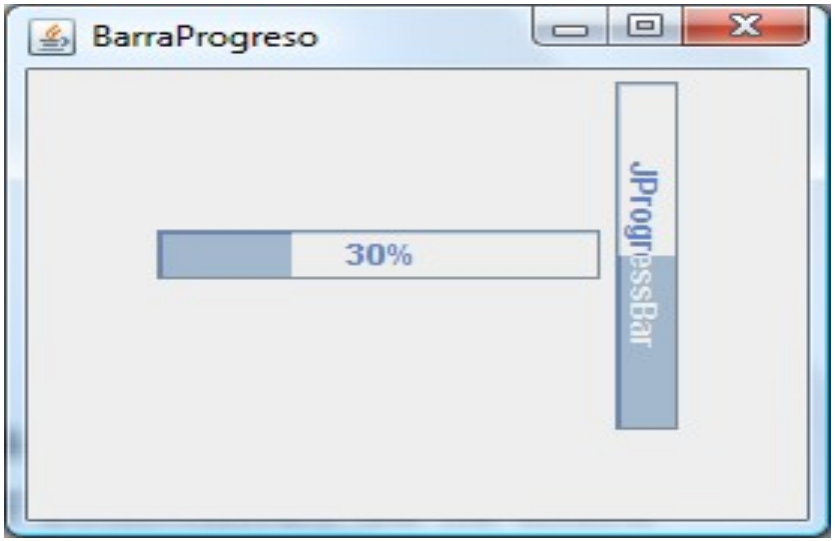
Permite al usuario seleccionar un valor numérico entre un rango determinado.

Se utiliza para restringir los valores que puede ofrecer al usuario y así evitar errores.



JProgressBar

Muestra gráficamente el progreso de una operación determinada.



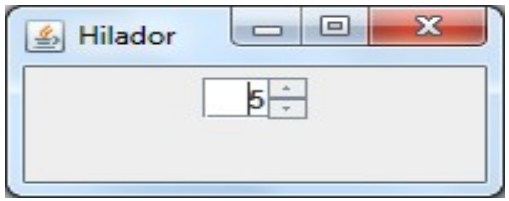
JSpinner

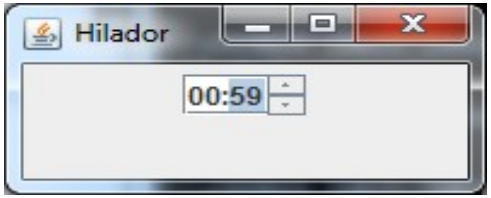
Permite elegir al usuario entre un rango de valores.

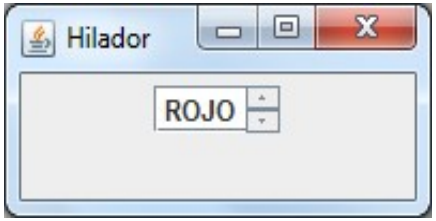
También permite introducir un valor.

Se le debe fijar un modelo (setModel(modelo)):

SpinnerNumberModel SpinnerDateModel SpinnerListModel







Por defecto crea un SpinnerNumberModel con el valor inicial a cero y sin límite inferior ni superior.

JFormattedTextField

Permite especificar el formato de entrada de datos mediante máscaras y sabe aprovechar el resto de especificaciones de formato disponibles en Java para números, fechas, horas, etc.

Permite decidir si se admiten caracteres incorrectos en la entrada o no.

Permite distinguir entre modalidad de edición y modalidad de visualización.

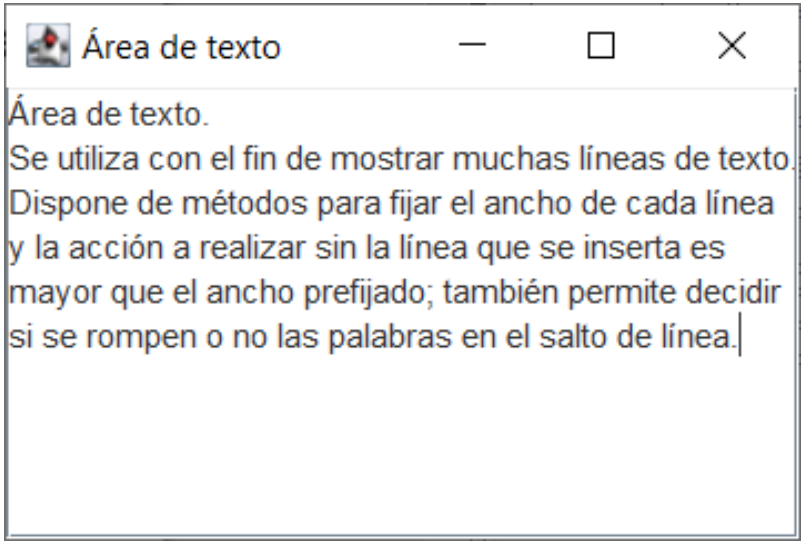
Permite que decidamos qué hacer con el foco si lo que el usuario ha introducido no es correcto.

JTextArea

Se utiliza con el fin de mostrar muchas líneas de texto.

Dispone de métodos para fijar el ancho de cada línea y la acción a realizar si la línea que se inserta es mayor que el ancho prefijado; también permite decidir si se rompen o no las palabras en el salto de línea.

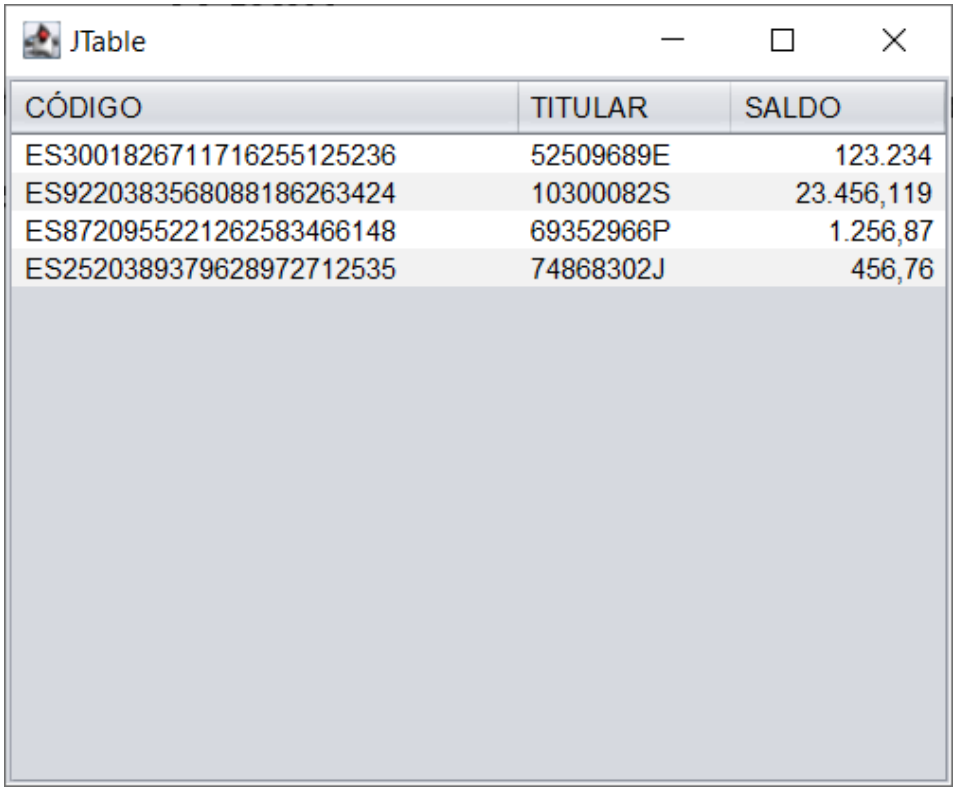
Es conveniente añadirlo a un JScrollPane (se hará en el constructor y no mediante el método add( )).



JTable

Permite visualizar tablas de datos, permitiendo editar los datos de manera opcional.

JTable no contiene los datos, es simplemente una vista de los datos.



JTable

Pasos para añadir un JTable:

1. Crear el objeto JTable

Dos contructores JTable(Object[ ][ ] rowData, Object[ ] columnNames)

JTable(TableModel dm)

DefaultTableModel es una clase que implementa **TableModel** que contiene todos los métodos necesarios para modificar datos en su interior, añadir filas o columnas y darle a cada columna el nombre que se desee.

2. Añadir el objeto JTable a un JScrollPane (en el constructor)

new JScrollPane(tabla)

JTable

Para crear nuestro propio TableModel deberemos crear una clase hija de AbstractTableModel y deberemos redefinir algunos métodos:

```
public Class getColumnClass (int index)
public int getColumnCount()
public String getColumnName (int index)
public int getRowCount()
public Object getValueAt (int rowIndex, int columnIndex)
public boolean isCellEditable (int rowIndex, int columnIndex)
public void setValueAt (Object aValue, int rowIndex, int columnIndex)
```

JTable

Para ordenar la tabla por columnas deberemos crear un objeto de la clase TableRowSorter y a continuación fijar el RowSorter en la tabla.

```
TableRowSorter<TableModel> sorter
    = new TableRowSorter<TableModel>(table.getModel());

tabla.setRowSorter(sorter);
```

Barra de menú

Pasos para la creación de una barra de menú:

1.- Crear una barra de menú (JMenuBar)

A) Crear los menús de la barra (**JMenu**)

B) Crear los elementos de menú (**JMenuItem**)

C) Configurar los elementos de menú

a) Añadirles mnemónicos (***setMnemonic( )***)

b) Añadirles atajos de teclado (***setAccelerator( )***)

D) Añadir los elementos de menú al menú (***add( )***)

E) Añadir separadores de elementos (***addSeparator( )***)

F) Añadir los menús a la barra de menús (***add( )***)

G) Fijar la barra de menús a la ventana (***setJMenuBar( )***)

2.-También podremos crear submenús. Para ello añadiremos un menú (con sus elementos de menú) a otro.

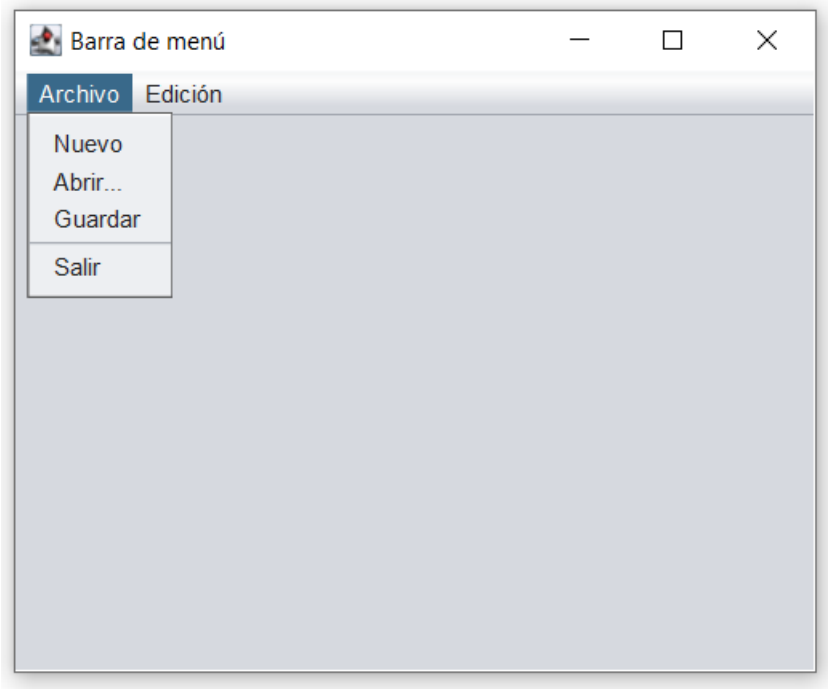
Barra de menú

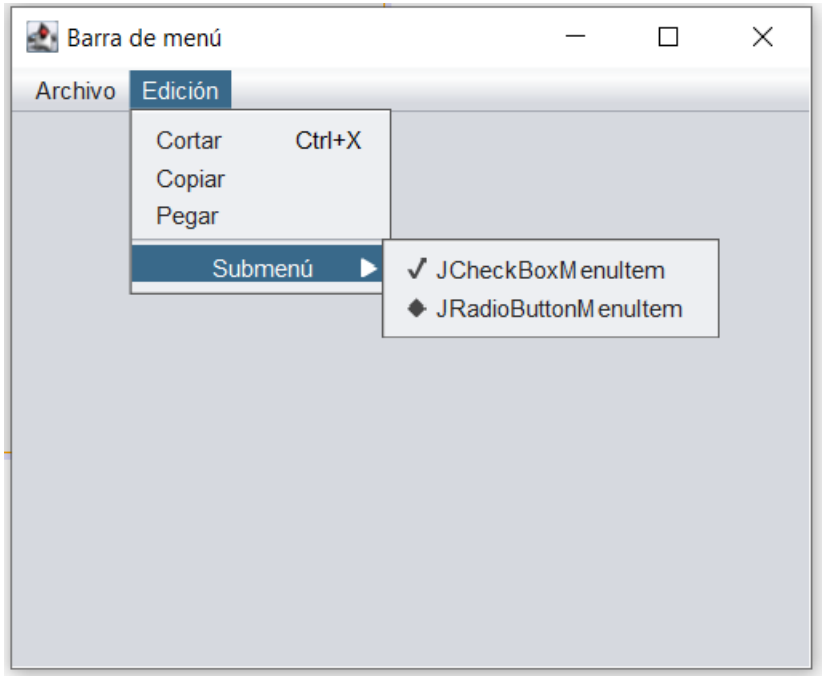
JMenuBar Representa la barra de menú que va a contener los distintos elementos de menú, que serán objetos de la clase JMenu.

JMenu Es una opción de menú determinada, que contiene varios elementos de menú, que serán objetos de la clase JMenuItem, y también puede contener submenús, es decir, objetos de la clase JMenu.

JMenuItem	Es un elemento de menú.
JCheckBoxMenuItem	Elemento de menú que posee casilla de verificación.
JRadioButtonMenuItem	Elemento de menú que posee un botón de opción.
JSeparator	Ofrece una separación entre elementos de menú.

Barra de menú



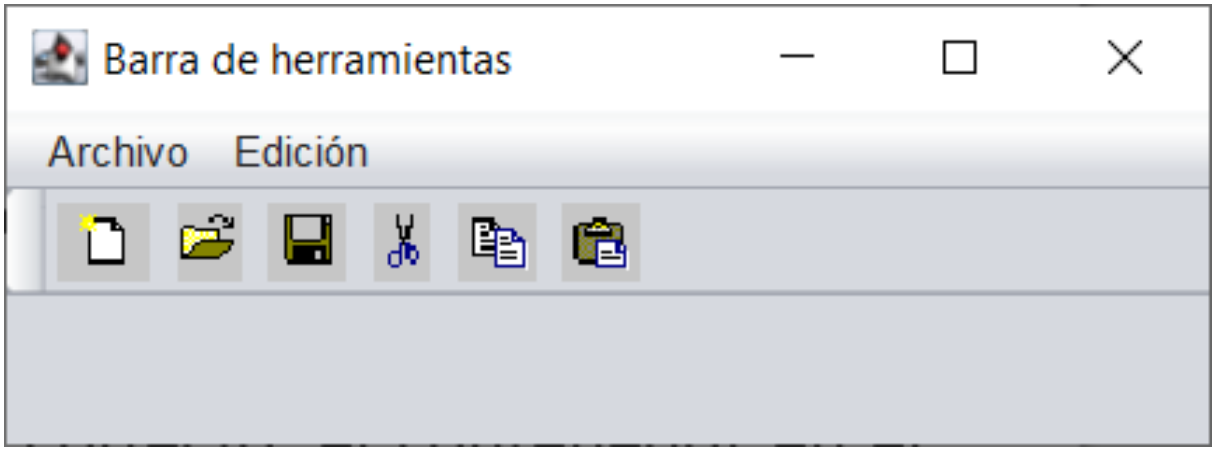


Barra de herramientas

JToolBar Representa una barra de herramientas.

Por defecto el usuario puede arrastrar la barra de herramientas y situarla en los diferentes bordes del contenedor o bien como una ventana independiente.

Para que el funcionamiento de arrastre de la barra sea correcto, el contenedor en el que se sitúa la barra de herramientas debe tener un gestor de diseño BorderLayout. Normalmente la barra de herramientas se añade en el norte del gestor de diseño.



Menú contextual

JPopupMenu Representa un menú contextual.

<u>Pasos en la creación de un JpopupMenu:</u>

Crear un objeto de la clase JPopupMenu

Crear los elementos de menú (JMenuItem)

Configurar los elementos de menú

Añadir los elemento de menú al JPopupMenu

Añadir separadores de elementos

Mostrar el JPopupMenu ***show(Component c , int x , int y))***

Contenedores superiores

JOptionPane	Permite crear sencillos diálogos estándar.
JColorChooser	Ofrece un diálogo estándar para la selección de colores.
JFileChooser	Ofrece un diálogo estándar para la selección de un fichero.
JDialog	Permite crear diálogos personalizados.

Todo diálogo es dependiente de una ventana determinada, si la ventana se destruye también lo harán sus diálogos asociados.

Los diálogos pueden ser modales o no modales.

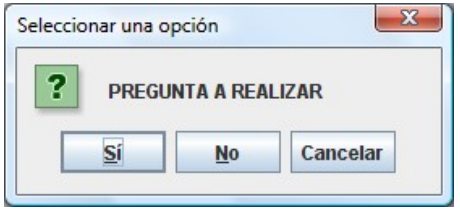
Cuando un diálogo modal se encuentra visible se bloquea la entrada del usuario en todas las demás ventanas del programa.

JOptionPane

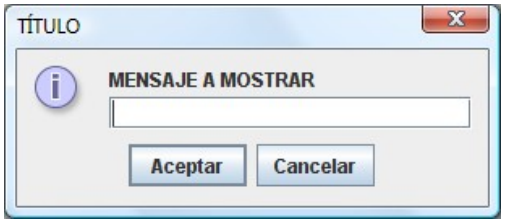
Contiene métodos estáticos que nos permiten crear sencillos diálogos estándar.

```
showMessageDialog(Component parentComponent, Object message)
```


```
showConfirmDialog(Component parentComponent, Object message)
```




```
showInputDialog(Component parentComponent, Object message)
```



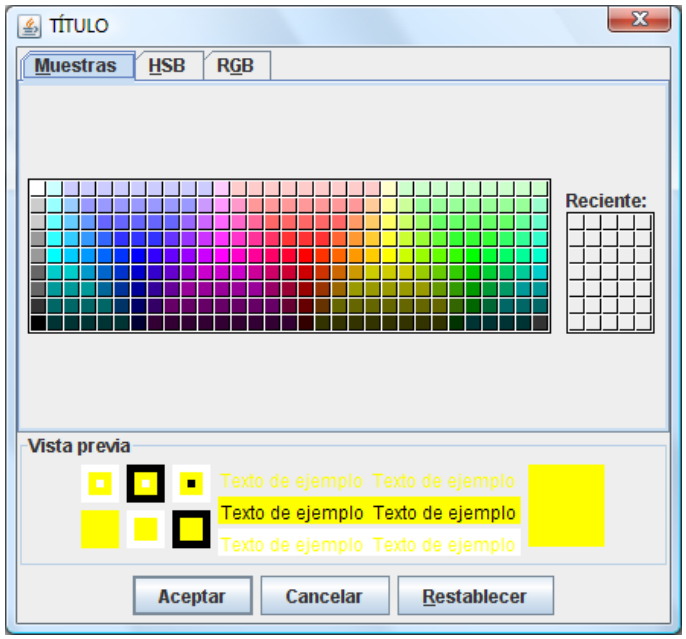
JColorChooser

Permite mostrar de forma sencilla un diálogo que permite seleccionar un color.

Para ello utilizaremos el método estático showDialog( ) de la clase JColorChooser.

```
showDialog(Component padre, String título, Color color)
```

Para el color utilizaremos los atributos estáticos de la clase Color.



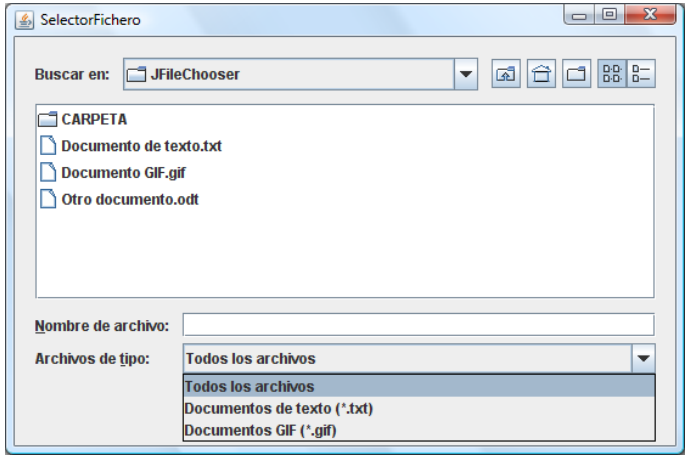
JFileChooser

Nos permiten navegar por el sistema de ficheros y seleccionar un fichero o directorio.

El constructor puede recibir como argumento un objeto File o String para indicar el directorio inicial.

Nos ofrece dos métodos para mostrar los dos tipos de selectores de ficheros distintos, el que se utiliza para abrir ficheros y el que se utiliza para grabar ficheros, estos métodos son:

```
showOpenDialog(Component parent)
showSaveDialog(Component parent)
```



Para obtener el fichero que se ha seleccionado se utiliza el método ***getSelectedFile( )***, que devuelve un objeto de la clase File.

JFileChooser

Si queremos mostrar solamente los archivos de determinado tipo deberemos crear filtros. Podremos añadir varios filtros.

Crearemos una clase hija de **javax.swing.filechooser.FileFilter** e implementaremos los métodos abstractos.

`accept(File f)`	Devuelve true si se acepta el archivo por el filtro.
`getDescription( )`	Devuelve la descripción del filtro.

Para añadir un filtro de archivos a un selector utilizamos el método

`setFileFilter( FileFilter filter)`.

Swing

JFileChooser

```
import javax.swing.*;
import java.io.*;
Import javax.swing.filechooser.FileFilter;
public class Filtro extends FileFilter{
   public boolean accept(File f){
      String n=f.getName();
      boolean b=false;
      if(f.isDirectory() || n.endsWith(".txt"))
         b=true;
      return b;
   }
   public String getDescription(){
      return "Documentos de texto (*.txt)";
   }
}
```

JDialog

Para crear un diálogo, se crea una clase hija de **JDialog**.

El funcionamiento en similar a JFrame.

Se debe obtener el contenedor intermedio (***getContentPane( )***)o crear un contenedor intermedio y fijarlo (***setContentPane( )***).

Se crearán y configuraran los componentes que se quieran añadir.

Se añadirán los componentes al contenedor intermedio.

Se fijará el tamaño y la posición del contenedor superior.

El constructor recibirá el contenedor del que dependa el diálogo y llamará al constructor de la clase padre mediante ***super(Frame propietario, String título, boolean modal)***.

En la ventana en la que queramos utilizar el diálogo deberemos crear un objeto de la clase.

El diálogo se mostrará mediante el método ***setVisible(true)***.

Eventos

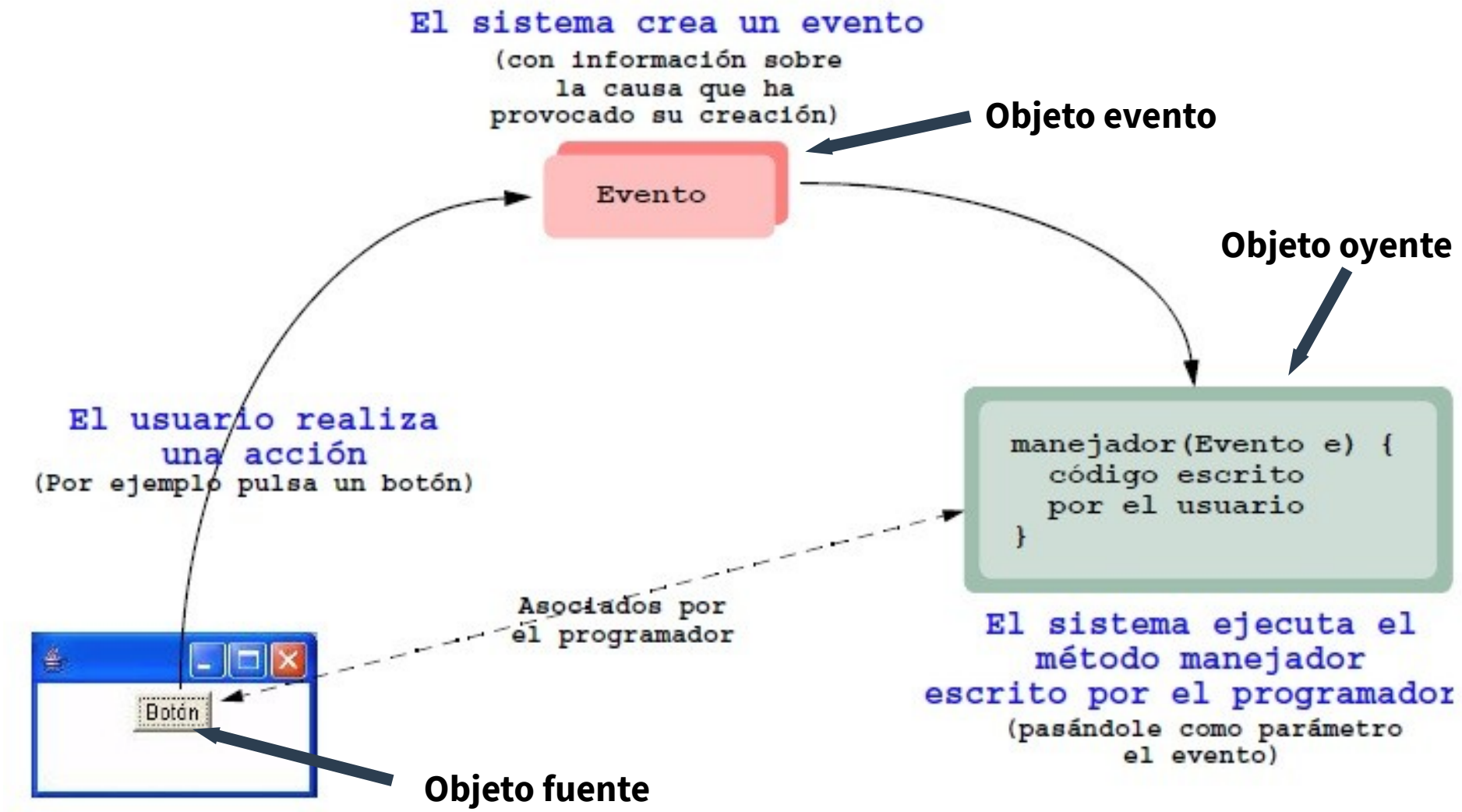
Un evento es un suceso que ocurre en un sistema.

En la interfaz gráfica, un evento es una acción realizada sobre algún componente.

Los eventos se generan en objetos llamados “fuentes” y delegan la responsabilidad de gestionarlos en otros llamados “oyentes”.

En la programación de eventos intervienen tres objetos: *objeto evento*, *objeto fuente* y *objeto oyente*.

Eventos

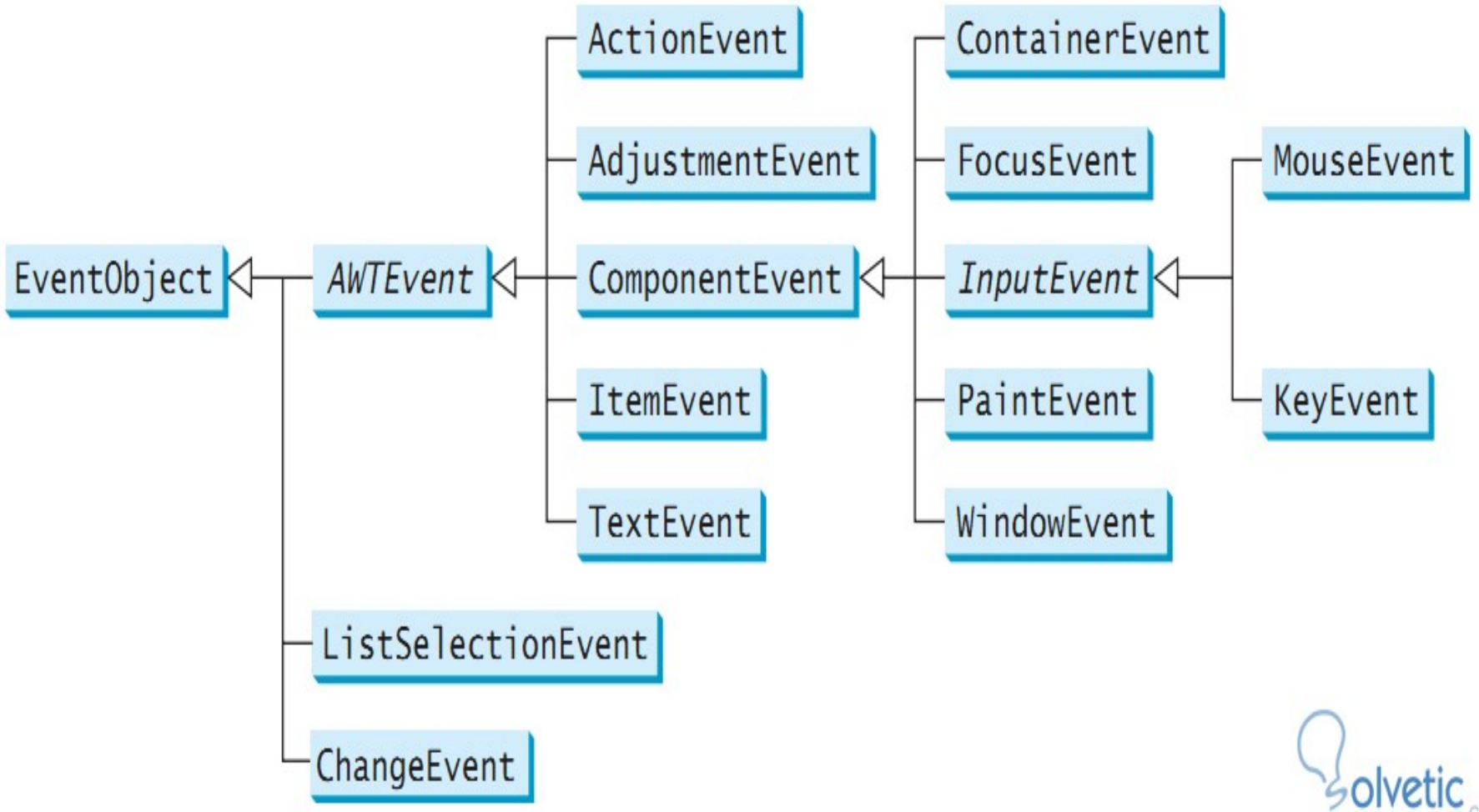

El sistema crea un evento
(con información sobre la causa que ha provocado su creación)
Objeto evento
Evento
Objeto oyente
manejador(Evento e) {
código escrito
por el usuario
}
El usuario realiza una acción
(Por ejemplo pulsa un botón)
Asociados por el programador
El sistema ejecuta el método manejador escrito por el programador
(pasándole como parámetro el evento)
Botón
Objeto fuente

Eventos

El lenguaje Java dentro de su paquete **java.awt.event** nos ofrece una completa jerarquía de clases cuya función es la de representar los distintos tipos de eventos que podemos tratar desde un interfaz de usuario.

El modelo de eventos que ofrece Java es denominado modelo de delegación de eventos (Delegation Event Model). Los diferentes tipos de eventos se encuentran encapsulados dentro de una jerarquía de clases que tienen como raíz a la clase **java.util.EventObject**.

Eventos


EventObject
AWTEvent
ActionEvent
AdjustmentEvent
ComponentEvent
ItemEvent
TextEvent
ListSelectionEvent
ChangeEvent
ContainerEvent
FocusEvent
InputEvent
PaintEvent
WindowEvent
MouseEvent
KeyEvent

Eventos

La idea fundamental del modelo de eventos de Java es que **un evento se propaga desde un objeto "fuente" (source) hacia un objeto "oyente" (listener)** invocando un método en el oyente al que se le pasa como parámetro una instancia de la clase del evento que representa el tipo de evento generado. Este método será el que trate al evento y actúe en consecuencia.

Desde el punto de vista de la jerarquía de clases de Java, un oyente es un objeto que implementa una interface **java.util.EventListener** específica.

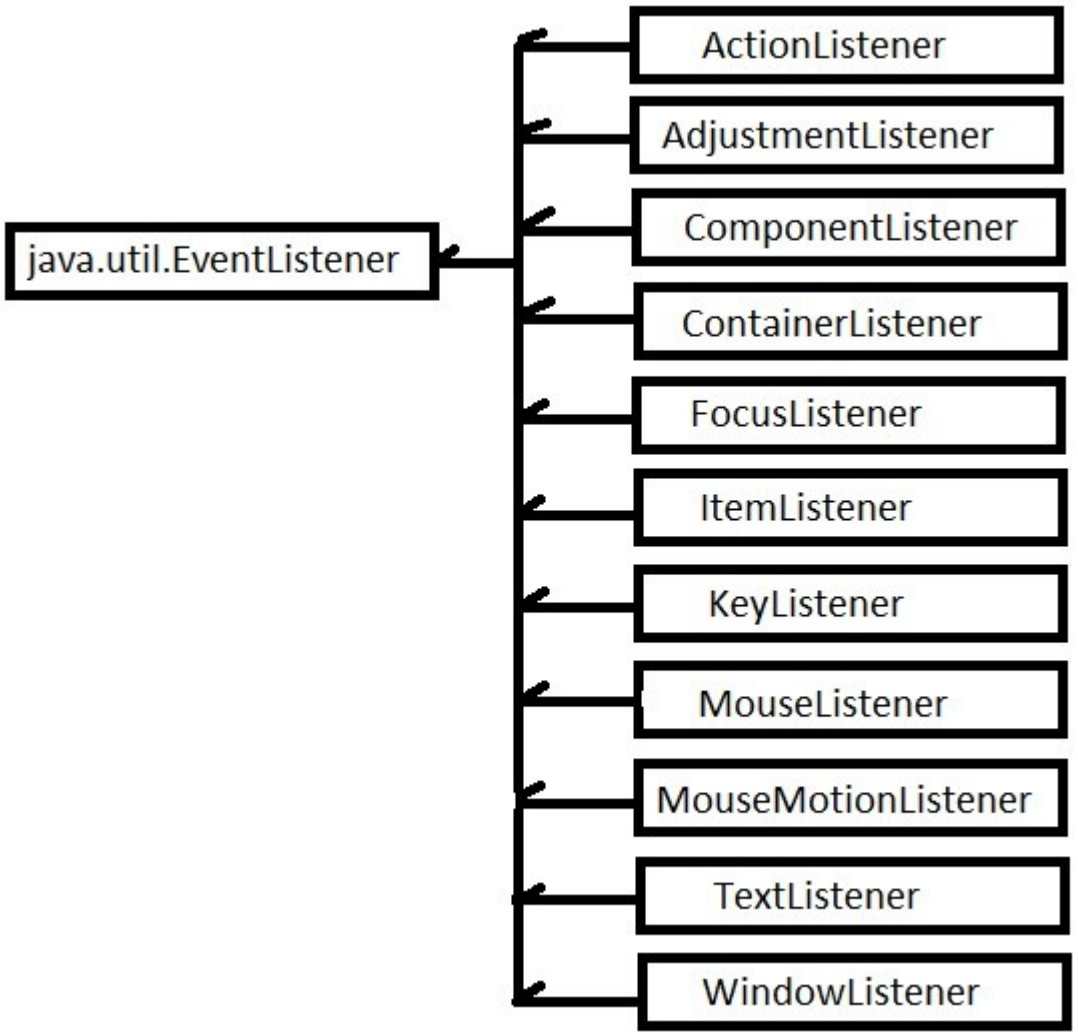
Eventos

La clase **oyente** implementará la interface que se corresponda con el tipo de **evento** que se desea tratar.

Una interface **EventListener** define uno o más métodos que serán invocados por la **fuente** del evento en respuesta a un tipo de **evento** específico.

Las características que debe tener un **oyente** es que debe implementar la interface que se corresponda con el **evento** que desea tratar y debe registrarse como **oyente** de la **fuente** de eventos que va a disparar los eventos a tratar.

Eventos


java.util.EventListener
ActionListener
AdjustmentListener
ComponentListener
ContainerListener
FocusListener
ItemListener
KeyListener
MouseListener
MouseMotionListener
TextListener
WindowListener

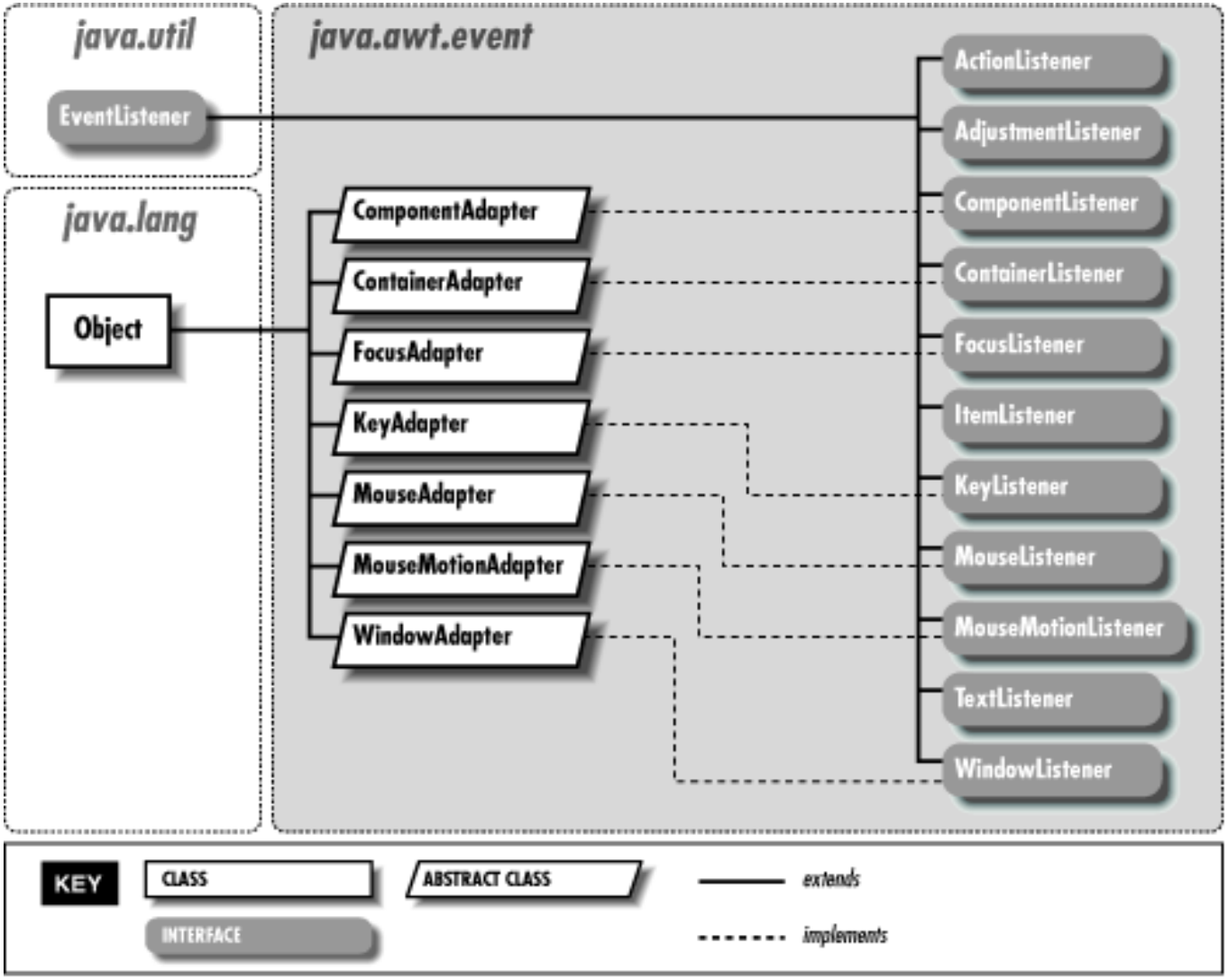
Eventos

Un **oyente** en lugar de implementar una interface, puede utilizar una clase que herede de una **clase adaptadora** de eventos del paquete **java.awt.event**.

Las **clases adaptadoras** permiten sobrescribir solamente los métodos de la interface en los que se esté interesado.

Esto es posible debido a que las **clases adaptadoras** implementan la interface correspondiente y simplemente tienen implementaciones vacías de todos los métodos de la interface **EventListener**.

Eventos


java.util
EventListener
java.lang
Object
java.awt.event
ComponentAdapter
ContainerAdapter
FocusAdapter
KeyAdapter
MouseAdapter
MouseMotionAdapter
WindowAdapter
ActionListener
AdjustmentListener
ComponentListener
ContainerListener
FocusListener
ItemListener
KeyListener
MouseListener
MouseMotionListener
TextListener
WindowListener
KEY
CLASS
ABSTRACT CLASS
INTERFACE
extends
implements

Eventos

Una fuente de eventos es un objeto que origina o "dispara" eventos.

La fuente define los eventos que van a se emitidos ofreciendo una serie de métodos de la forma ***add<Tipo de Evento>( )***, que son usados para registrar oyentes específicos para esos eventos.

A los métodos ***add<Tipo de Evento>( )*** se les pasa por parámetro una instancia del oyente que va a tratar los eventos que genere la fuente.

Eventos

Dentro de la jerarquía de clases de Java hay una serie de clases para representar todos los eventos y otra serie de interfaces que definen una serie de métodos que deben implementar las clases que van a tratar los eventos, es decir, lo que hemos llamado oyentes.

Otras clases que también hemos comentado y que se utilizan dentro del tratamiento de eventos en Java son las clases adaptadoras.

Cada conjunto de eventos tiene asociado una interface, y cada una de estas interfaces declara una serie de métodos para cada uno de los eventos lógicos asociados al tipo de evento de que se trate.

Eventos

Evento	Interface	Componentes
ActionEvent	**ActionListener**	JButton, JTextField, JmenuItem, ...
AdjustmentEvent	**AdjustmentListener**	JscrollBar
ChangeEvent	**ChangeListener**	JSlider
ComponentEvent	**ComponentListener**	* Component *y sus derivados, incluyendo* JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollBar, JTextArea, JTextField
ContainerEvent	**ContainerListener**	Container *y sus derivados, incluyendo* JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame
FocusEvent	**FocusListener**	Component *y sus derivados *.*
KeyEvent	**KeyListener**	Component *y sus derivados *.*
MouseEvent	**MousListener**	Component *y sus derivados *.*
	MouseMotionListener	Component *y sus derivados *.*
WindowEvent	**WindowListener**	Window *y sus derivados, incluyendo* JDialog, JFileDialog, JFrame
ItemEvent	**ItemListener**	JCheckBox, JCheckBoxMenuItem, JComboBox, JRadioButton, JMenu

Eventos

Evento	Métodos
ActionEvent	String getActionCommand( ) Object getSource( )
ChangeEvent	Object getSource( )
ComponentEvent	Component getComponent( )
ContainerEvent	Container getContainer( )
FocusEvent	Component getComponent( ) Component getOppositeComponent( )
KeyEvent	char getKeyChar( ) int getKeyCode( ) int getModifiers( )
MouseEvent	int getButton( ) int getClickCount( ) Point getLocationOnScreen( ) int getX( ) int getY( ) int getModifiersEx( )
WindowEvent	int getNewState( ) int getOldState( )
ItemEvent	Object getItem( ) int getStateChange( )
ListSelectionEvent	int getFirsIndex( ) int getLastIndex( )

Eventos

Interface	Métodos	Clase Adaptadora
ActionListener	actionPerformed(ActionEvent)	
ChangeListener	stateChanged(ChangeEvent)	
ComponentListener	componentHidden(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)	ComponentAdapter
ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)	ContainerAdapter
FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)	FocusAdapter
KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)	KeyAdapter
MouseListener	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)	MouseAdapter
MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)	MouseMotionAdapter

Eventos

Interface	Métodos	Clase Adaptadora
WindowListener	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)	WindowAdapter
ItemListener	itemStateChanged(ItemEvent)	
ListSelectionListener	valueChanged(ListSelectionEvent)	

Eventos

Posibilidades para la creación del oyente:

1. Hacer que la clase que contiene el objeto fuente sea a su vez oyente.

2. Crear la clase oyente como una clase interna (clase definida dentro de otra clase y que tiene acceso a los miembros de la clase que la encierra). La clase interna puede ser anónima.

3. Crear una clase oyente independiente (pasar una referencia a la clase que contiene el objeto fuente en el constructor de la clase oyente).

Eventos

Fuente y oyente

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class EventosFuente extends JFrame implements ActionListener{
    private JPanel panel;
    private JButton b;
    private JScrollPane sp;
    private JTextArea ta;

    public EventosFuente( ){
        super("Eventos Fuente");
        ..........................................
        b.addActionListener(this);
        ..........................................
    }

    public void actionPerformed(ActionEvent ae){
        ta.append("PULSADO "+ae.getActionCommand( )+"\n");
    }
    ….....................................
}
```

Eventos

Con clase interna

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class EventosInterna extends JFrame {
     private JPanel panel;
     private JButton b;
     private JScrollPane sp;
     private JTextArea ta;

     public EventosInterna( ){
          super("Eventos interna");
          ..........................................
          b.addActionListener(new ActionListener(){
               public void actionPerformed(ActionEvent ae){
                    ta.append("PULSADO "+ae.getActionCommand( )+"\n");
               }
          });

          ..............................................
     }

     ….....................................
}
```

Eventos

Con clase oyente

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class EventosExterna extends JFrame {
     private JPanel panel;
     private JButton b;

     public EventosExterna( ){
          super("Eventos externa");
          ..........................................
          b.addActionListener(new Oyente());

          ..........................................
     }

     ….........................................
}

public class Oyente implements ActionListener{
     public void actionPerformed(ActionEvent ae){
               System.out.println("PULSADO "+ae.getActionCommand( ));
     }
}
```

Patrón MVC

El patrón de diseño MVC (modelo-vista-controlador) propone separar los datos de una aplicación, la interfaz de usuario y la lógica de control en tres componentes distintos.

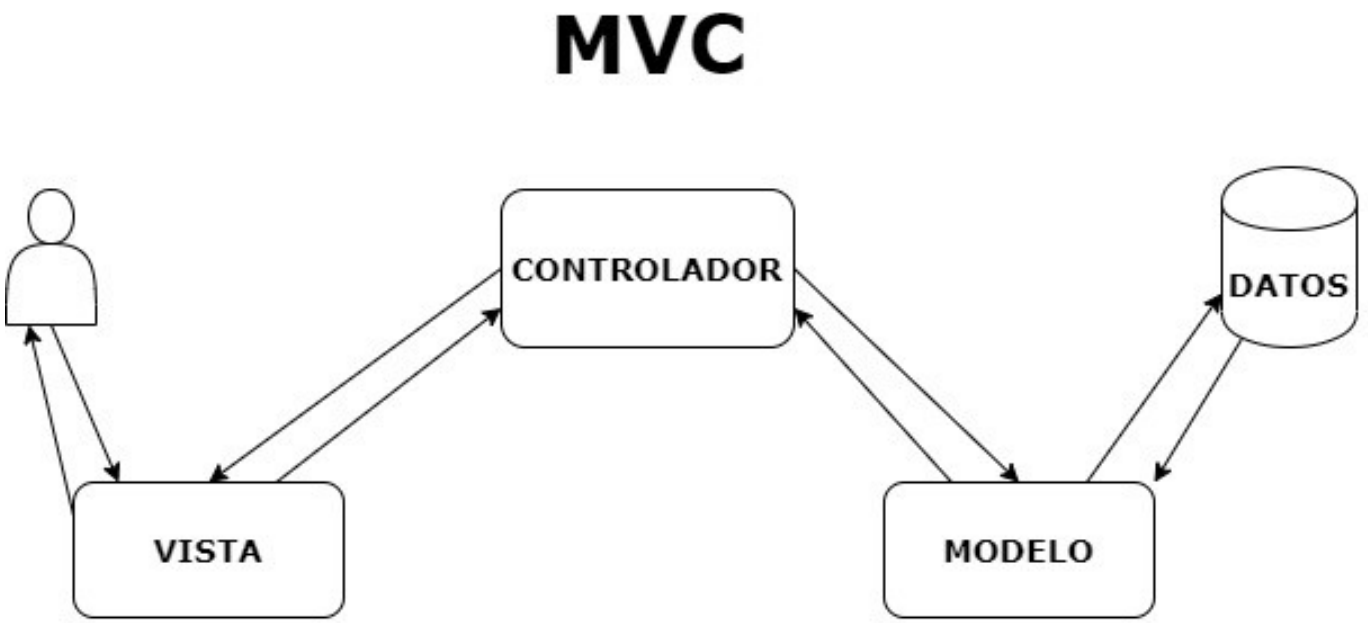
Hay distintas adaptaciones del patrón y distintos frameworks que nos permiten trabajar con el patrón MVC.

Spring Framework es un framework de código abierto para la creación de aplicaciones en Java.

Spring provee de una implementación de patrón MVC.

Patrón MVC

Diagrama que muestra la relación entre el modelo, la vista y el controlador.



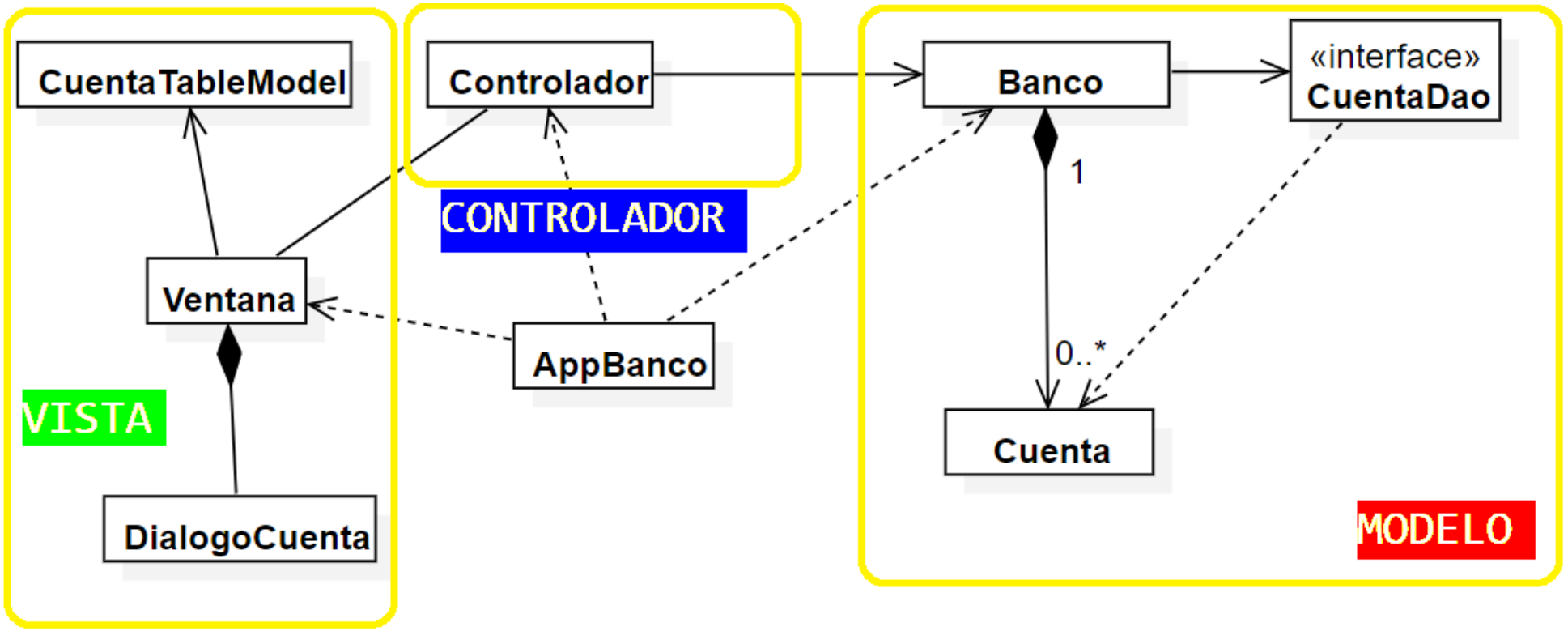
Patrón MVC

La VISTA será la interfaz de usuario. El usuario interactuará con la aplicación mediante la vista y en ella mostraremos la información al usuario.

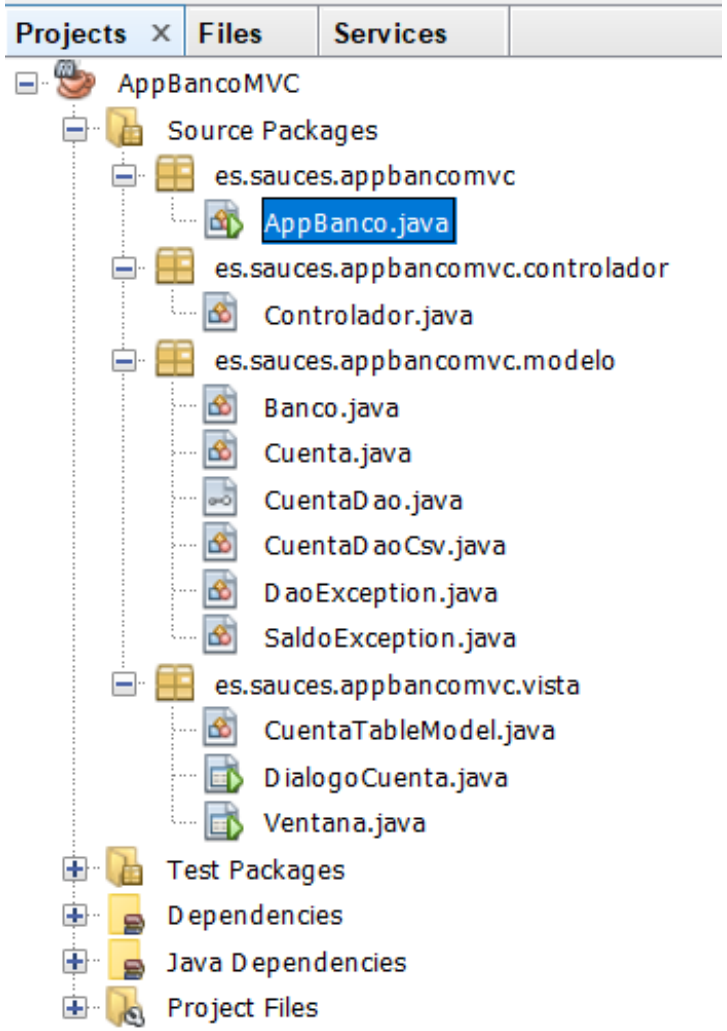
El MODELO tendrá la representación de los datos que maneja la aplicación, la lógica de negocio y los mecanismos de persistencia.

El CONTROLADOR actúa de intermediario entre la vista y el modelo.

Patrón MVC


CuentaTableModel
Controlador
Banco
«interface»
CuentaDao
CONTROLADOR
Ventana
AppBanco
1
0..*
VISTA
Cuenta
DialogoCuenta
MODELO

Patrón MVC


Projects
Files
Services
AppBancoMVC
Source Packages
es.sauces.appbancomvc
AppBanco.java
es.sauces.appbancomvc.controlador
Controlador.java
es.sauces.appbancomvc.modelo
Banco.java
Cuenta.java
CuentaDao.java
CuentaDaoCsv.java
DaoException.java
SaldoException.java
es.sauces.appbancomvc.vista
CuentaTableModel.java
DialogoCuenta.java
Ventana.java
Test Packages
Dependencies
Java Dependencies
Project Files

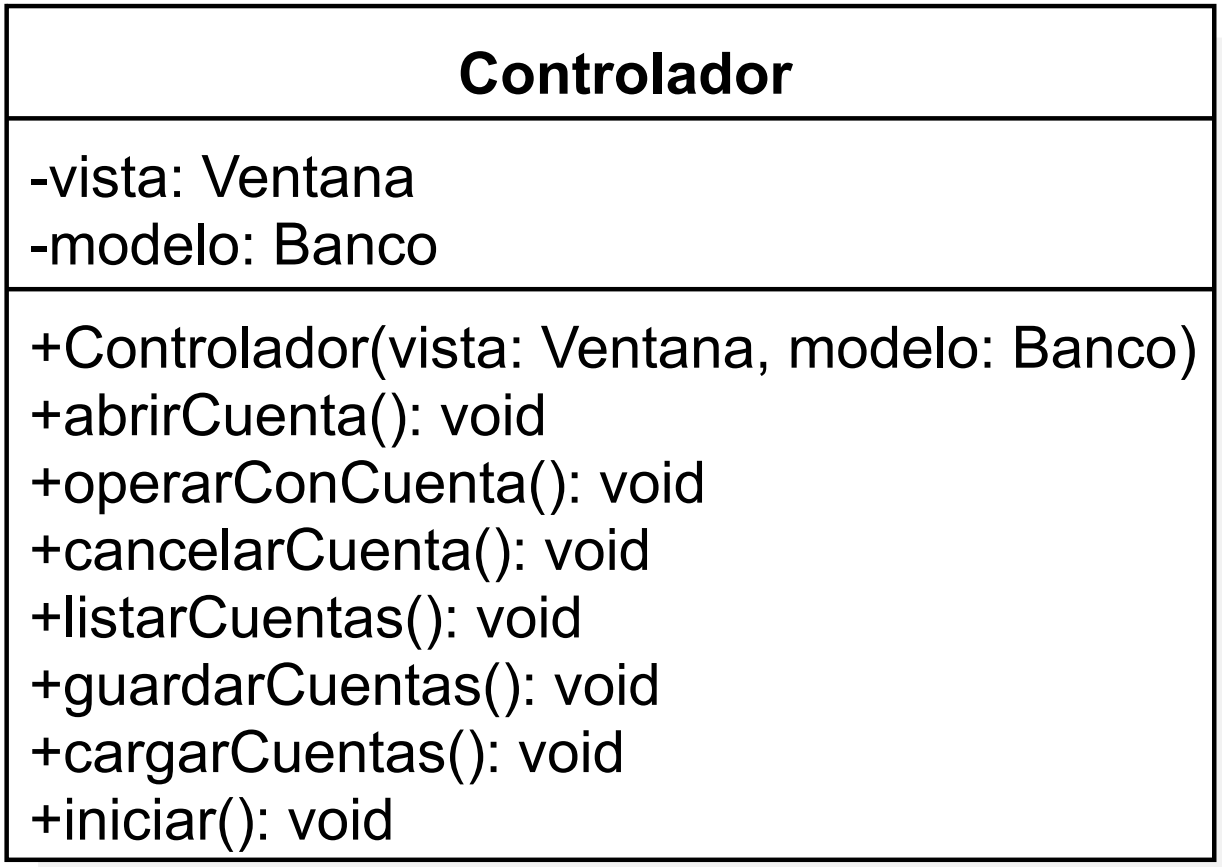
Patrón MVC

Método main

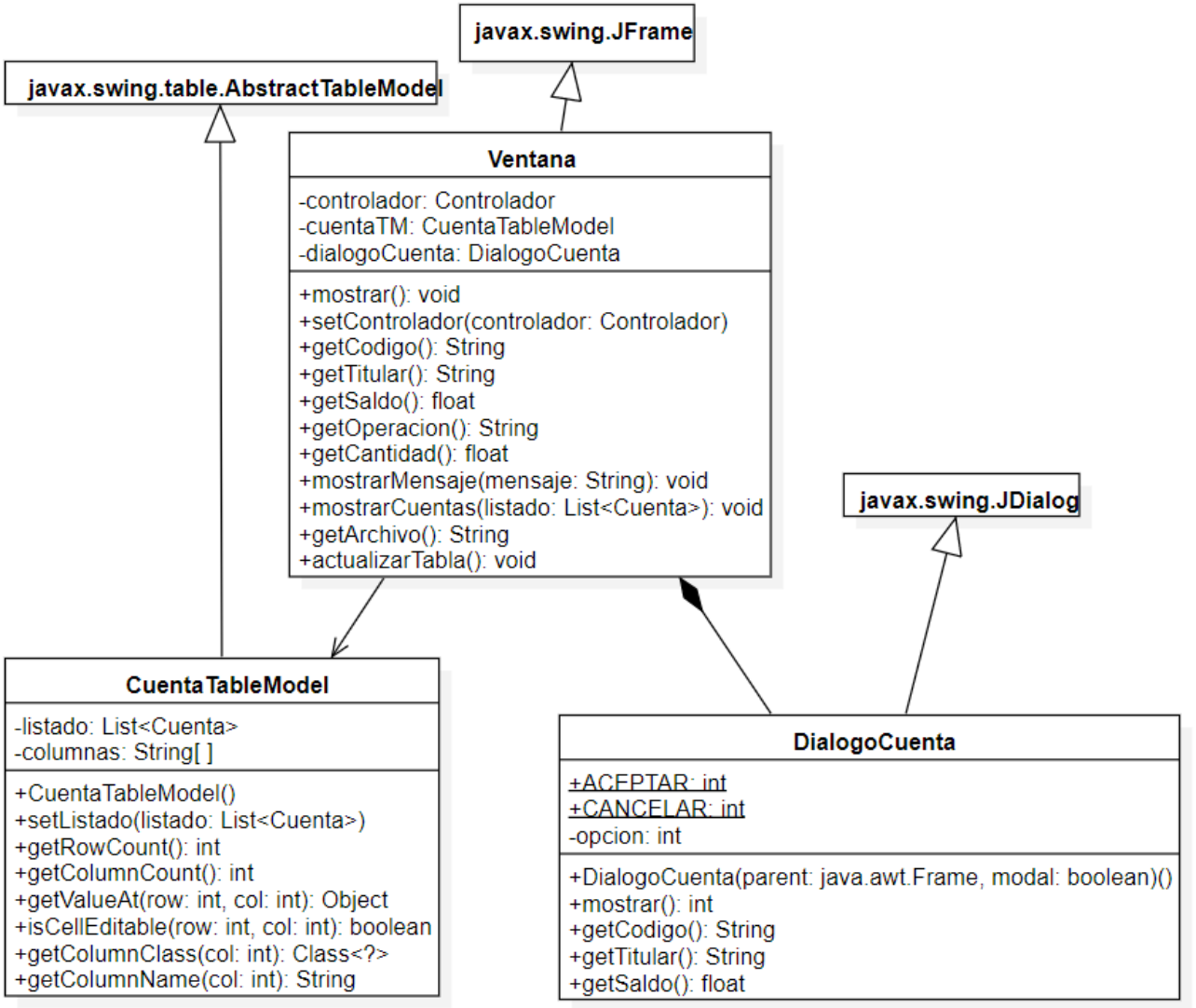
En el método main de la clase AppBancoM deberemos crear el modelo, la vista y el controlador. Fijaremos el controlador a la vista e iniciaremos la aplicación.

```
Banco modelo=new Banco("BANCO SAUCES");
Ventana vista=new Ventana();
Controlador controlador=new Controlador(vista,modelo);
vista.setControlador(controlador);
controlador.iniciar();
```

Patrón MVC


Controlador
-vista: Ventana
-modelo: Banco
+Controlador(vista: Ventana, modelo: Banco)
+abrirCuenta(): void
+operarConCuenta(): void
+cancelarCuenta(): void
+listarCuentas(): void
+guardarCuentas(): void
+cargarCuentas(): void
+iniciar(): void

Patrón MVC


javax.swing.JFrame
javax.swing.table.AbstractTableModel
Ventana
-controlador: Controlador
-cuentaTM: CuentaTableModel
-dialogoCuenta: DialogoCuenta
+mostrar(): void
+setControlador(controlador: Controlador)
+getCodigo(): String
+getTitular(): String
+getSaldo(): float
+getOperacion(): String
+getCantidad(): float
+mostrarMensaje(mensaje: String): void
+mostrarCuentas(listado: List<Cuenta>): void
+getArchivo(): String
+actualizarTabla(): void
javax.swing.JDialog
CuentaTableModel
-listado: List<Cuenta>
-columnas: String[ ]
+CuentaTableModel()
+setListado(listado: List<Cuenta>)
+getRowCount(): int
+getColumnCount(): int
+getValueAt(row: int, col: int): Object
+isCellEditable(row: int, col: int): boolean
+getColumnClass(col: int): Class<?>
+getColumnName(col: int): String
DialogoCuenta
+ACEPTAR: int
+CANCELAR: int
-opcion: int
+DialogoCuenta(parent: java.awt.Frame, modal: boolean)()
+mostrar(): int
+getCodigo(): String
+getTitular(): String
+getSaldo(): float

Patrón MVC

```java
import javax.swing.table.AbstractTableModel;
import java.util.List;
import java.util.ArrayList;
public class CuentaTableModel extends AbstractTableModel{
    private String[] columnas;
    private List<Cuenta> cuentas;

    public CuentaTableModel(){
        this.columnas=new String[]{"CÓDIGO","TITULAR","SALDO"};
        this.cuentas=new ArrayList<>();
    }

    public String[] getColumnas(){
        return columnas;
    }

    public void setColumnas(String[] columnas){
        this.columnas=columnas;
    }

    public List<Cuenta> getCuentas(){
        return cuentas;
    }

    public void setCuentas(List<Cuenta> cuentas){
        this.cuentas=cuentas;
        this.fireTableDataChanged();
    }

    @Override
    public int getRowCount() {
        return cuentas.size();
    }

    @Override
    public int getColumnCount() {
        return columnas.length;
    }

```

Devuelve el número de filas de la tabla

Devuelve el número de columnas de la tabla

Patrón MVC

```
39
40      @Override
41      public Object getValueAt(int rowIndex, int columnIndex) {
42          Cuenta c=cuentas.get(rowIndex);
43          Object o=null;
44          switch(columnIndex){
45              case 0: o= c.getCodigo();
46                  break;
47              case 1: o= c.getTitular();
48                  break;
49              case 2: o= c.getSaldo();
50                  break;
51          }
52          return o;
53      }
54
55      @Override
56      public boolean isCellEditable(int rowIndex, int columnIndex) {
57          return false;
58      }
59
60      @Override
61      public Class<?> getColumnClass(int columnIndex) {
62          Class<?> clase=null;
63          switch(columnIndex){
64              case 0: clase=String.class;
65                  break;
66              case 1: clase=String.class;
67                  break;
68              case 2: clase=Float.class;
69                  break;
70          }
71          return clase;
72      }
73
74      @Override
75      public String getColumnName(int column) {
76          return columnas[column];
77      }
78  }
```

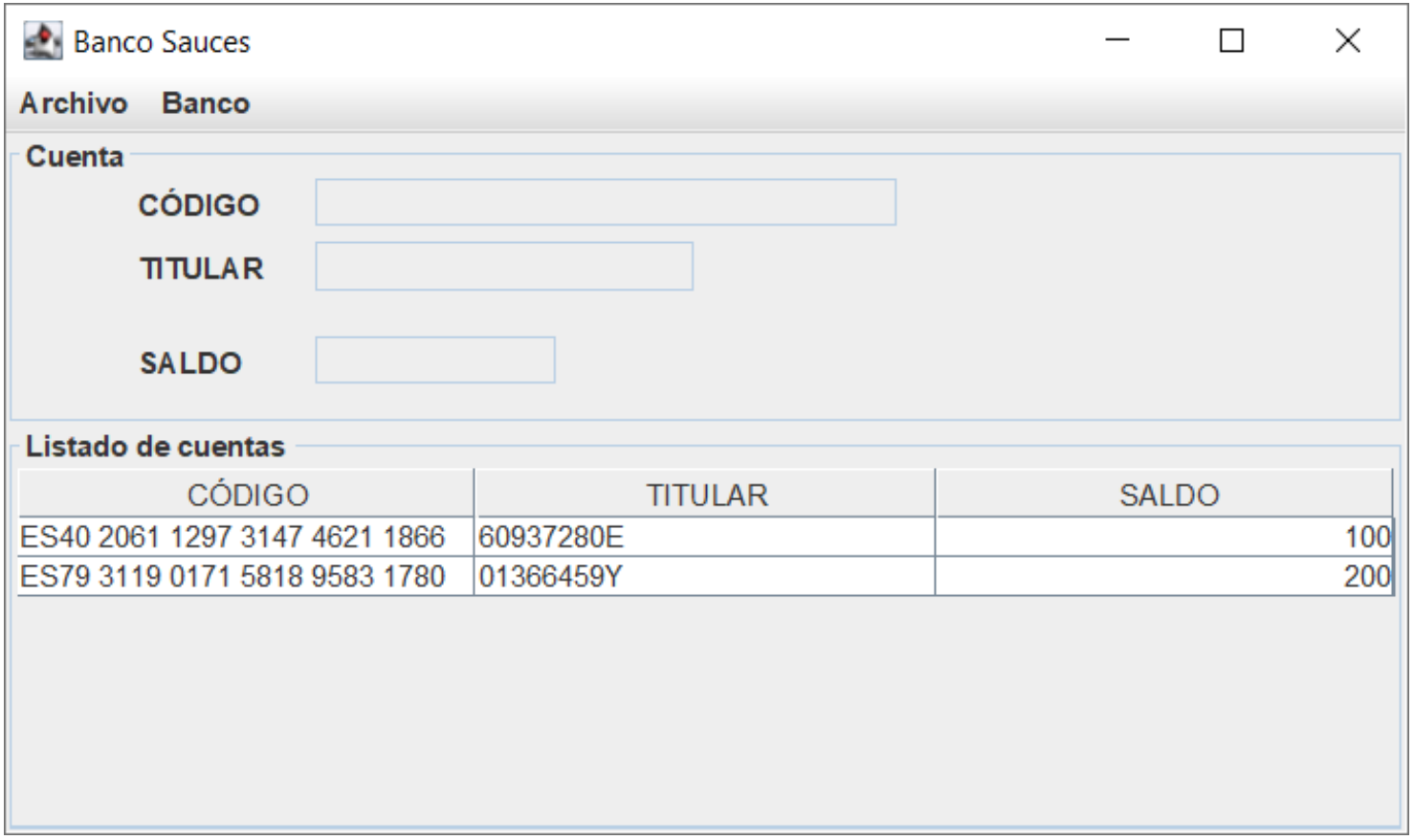
Devuelve el valor de una celda de la tabla

Devuelve si una celda de la tabla es editable

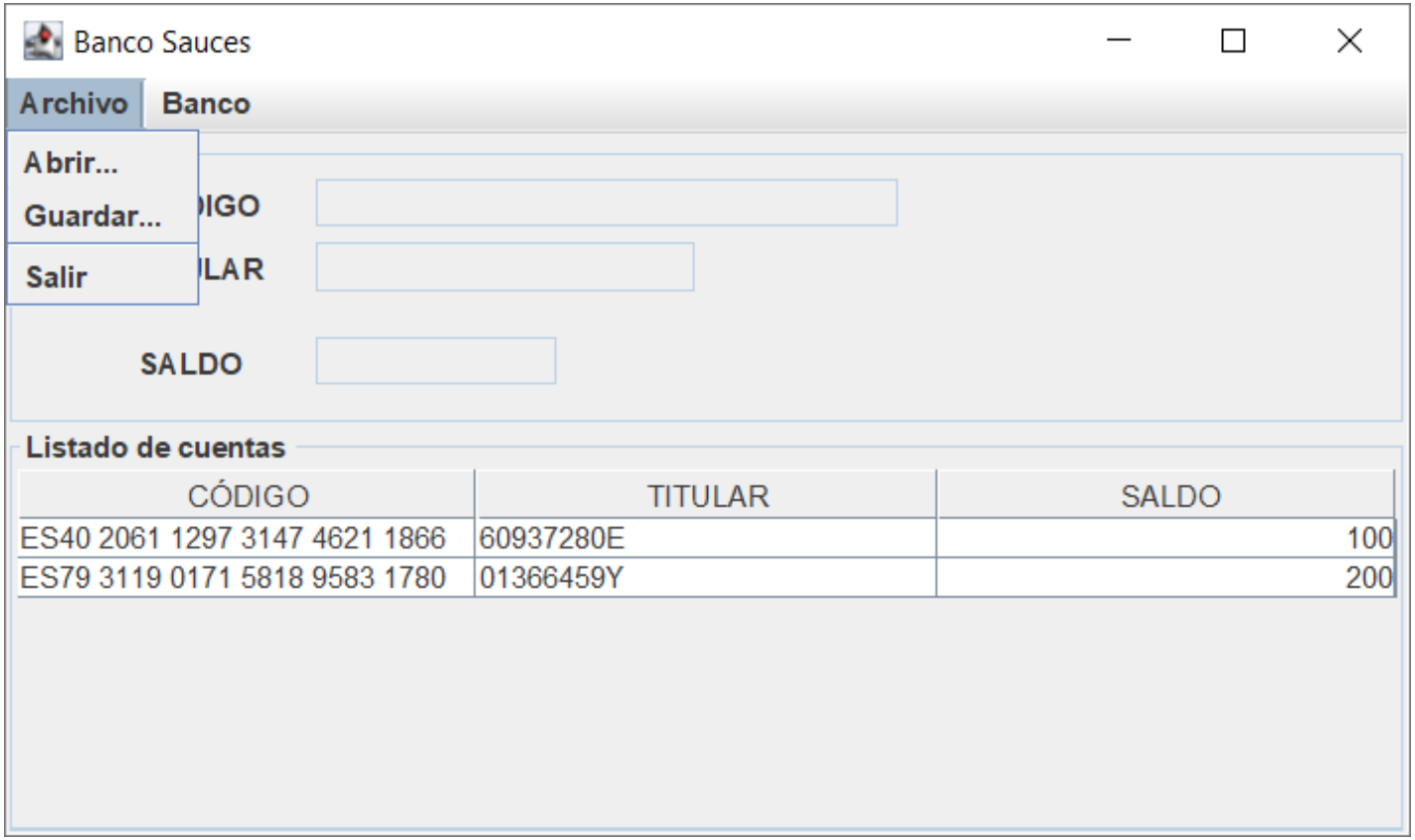
Devuelve el tipo de data que almacena una columna

Devuelve el nombre de una columna

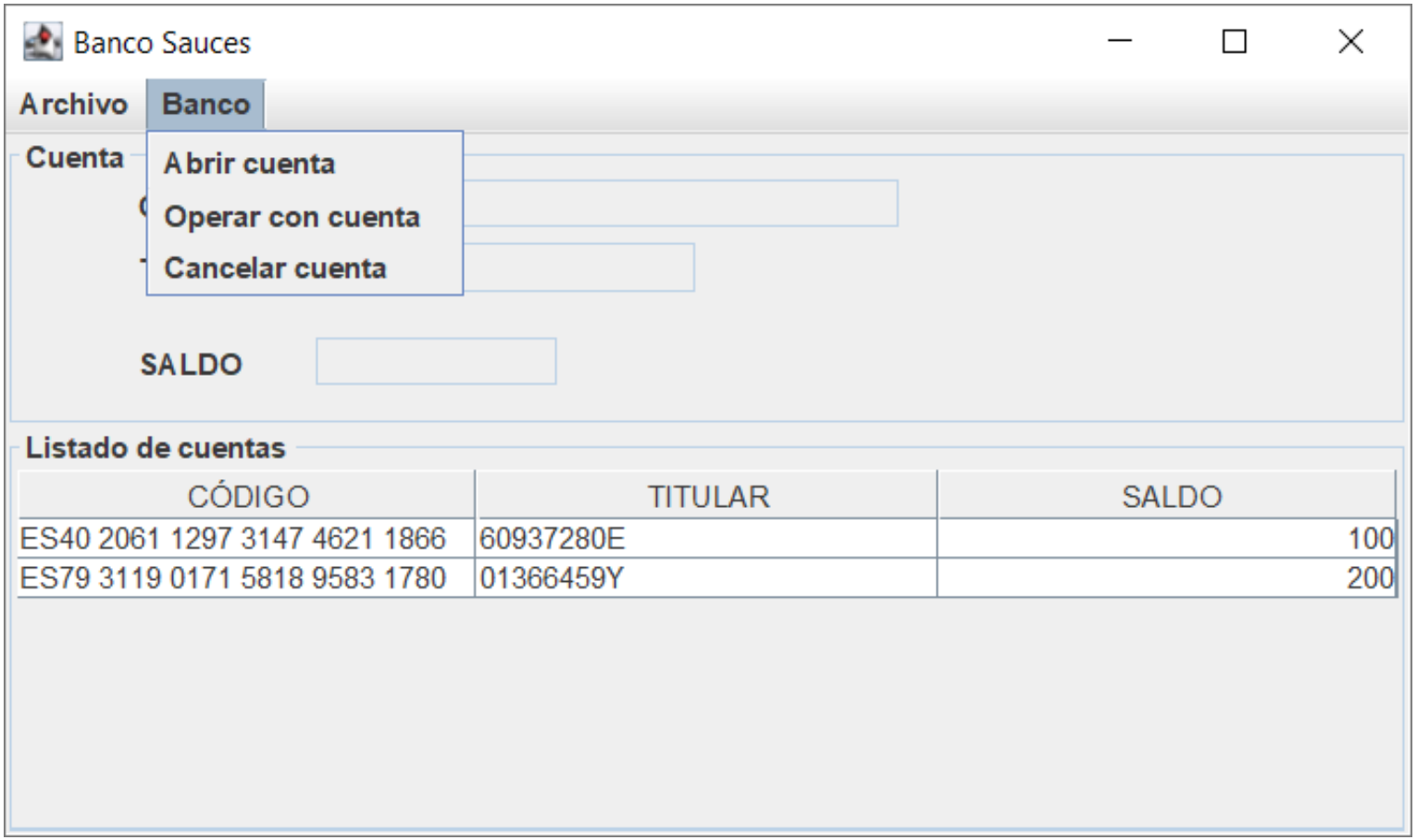
Patrón MVC


Banco Sauces
Archivo
Banco
Cuenta
CÓDIGO
TITULAR
SALDO
Listado de cuentas
CÓDIGO
TITULAR
SALDO
ES40 2061 1297 3147 4621 1866
60937280E
100
ES79 3119 0171 5818 9583 1780
01366459Y
200

Patrón MVC

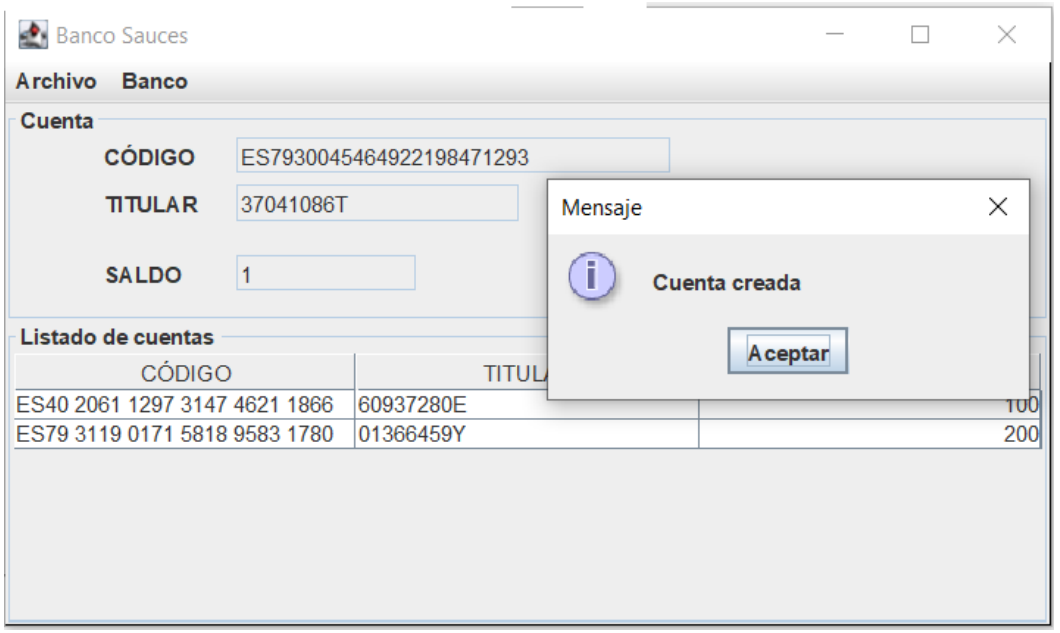

Banco Sauces
Archivo
Banco
Abrir...
Guardar...
Salir
IGO
LAR
SALDO
Listado de cuentas
CÓDIGO
TITULAR
SALDO
ES40 2061 1297 3147 4621 1866
60937280E
100
ES79 3119 0171 5818 9583 1780
01366459Y
200

Patrón MVC


Banco Sauces
Archivo
Banco
Abrir cuenta
Operar con cuenta
Cancelar cuenta
Cuenta
SALDO
Listado de cuentas
CÓDIGO
TITULAR
SALDO
ES40 2061 1297 3147 4621 1866
60937280E
100
ES79 3119 0171 5818 9583 1780
01366459Y
200

Patrón MVC


Crear cuenta
CÓDIGO
ES7930045464922198471293
TITULAR
37041086T
SALDO
1000
ACEPTAR
CANCELAR


Banco Sauces
Archivo
Banco
Cuenta
CÓDIGO
ES7930045464922198471293
TITULAR
37041086T
SALDO
1
Mensaje
Cuenta creada
Aceptar
Listado de cuentas
CÓDIGO
TITULA
ES40 2061 1297 3147 4621 1866
60937280E
100
ES79 3119 0171 5818 9583 1780
01366459Y
200

Patrón MVC

Code Customizer

Component: tablaCuentas Rename...

Initialization code

default code
pre-init
custom prop...
post-init
post-init
post-init
post-init
post-init
post-init
post-init
post-init
post-init

```
tablaCuentas = new javax.swing.JTable();
ctm=new CuentaTableModel();
tablaCuentas.setModel(ctm);
tablaCuentas.getSelectionModel().addListSelectionListener(new ListSelectionListener(){
    public void valueChanged(ListSelectionEvent lse){
        int fila=tablaCuentas.getSelectedRow();
        if(fila>=0){
            tfCodigo.setText((String)tablaCuentas.getValueAt(fila, 0));
            tfTitular.setText((String)tablaCuentas.getValueAt(fila, 1));
            ftfSaldo.setText(tablaCuentas.getValueAt(fila, 2).toString());
        }
    }
});
panelListado.setViewportView(tablaCuentas);
```

Variable declaration code

post-declarat...

```
private javax.swing.JTable tablaCuentas;
private CuentaTableModel ctm;
```

Variable: field Access: private final static transient volatile

Banco Sauces

Archivo Banco

Cuenta

CÓDIGO ES40 2061 1297 3147 4621 1866

TITULAR 60937280E

SALDO 100.0

Listado de cuentas

CÓDIGO	TITULAR	SALDO
ES40 2061 1297 3147 4621 1866	60937280E	100
ES79 3119 0171 5818 9583 1780	01366459Y	200

Internacionalización (i18n)

La internacionalización (i18n) es el proceso de diseño de aplicaciones de tal manera que puedan adaptarse a diferentes idiomas y regiones sin necesidad de realizar cambios en el código.

Algunos aspectos a contemplar:

- Codificación Unicode
- Formatos de fecha
- Formatos de hora
- Formatos de números
- Formatos de moneda

Internacionalización (i18n)

En Java crearemos un archivo de propiedades donde recogeremos todos los mensajes de la aplicación para la configuración por defecto.

mensajes.properties

```
hola=Hola mundo!
```

Luego crearemos un archivo de propiedades por cada idioma y región.

mensajes_es_ES.properties

```
hola=Hola mundo!
```

mensajes_en_GB.properties

```
hola=Hello world!
```

Internacionalización (i18n)

En la aplicación deberemos obtener un objeto de la clase **ResourceBundle**, del paquete **java.util**.

Lo obtendremos a través de su método estático:

getBundle(String baseName)

baseName hace referencia al nombre del archivo de propiedades.

```
java.util.ResourceBundle rb=ResourceBundle.getBundle("mensajes");
```

Los mensajes que aparezcan en la aplicación se deberán especificar mediante las claves del archivo de propiedades.

```
System.out.println(rb.getString("hola"));
```

Internacionalización (i18n)

Dependiendo de la configuración regional establecida (idioma y región) se mostrará un mensaje u otro.

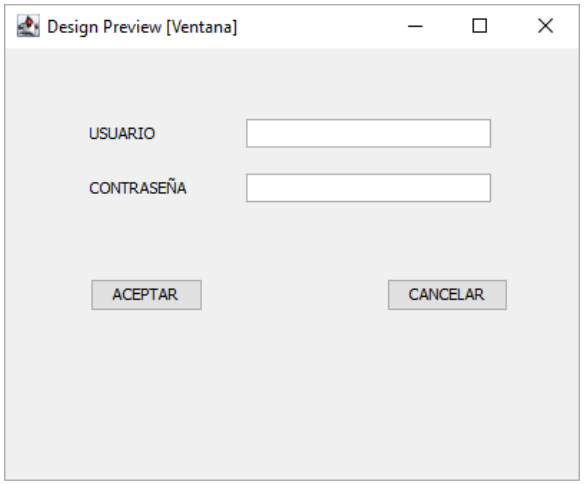
es_ES

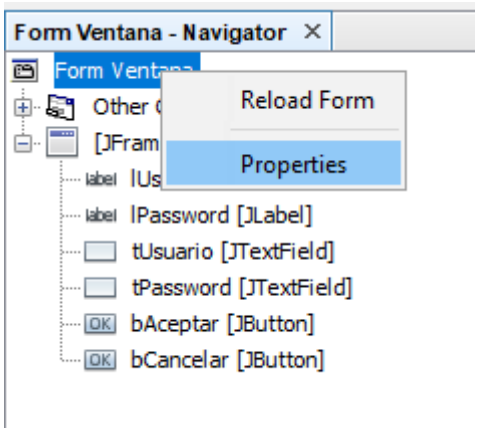
```
Hola mundo!
```

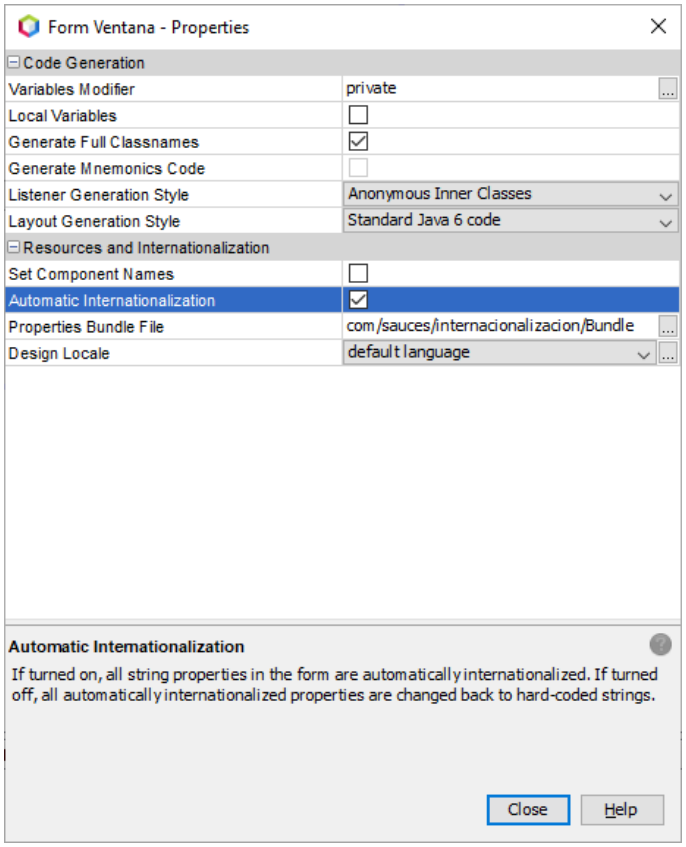
en_GB

```
Hello world!
```

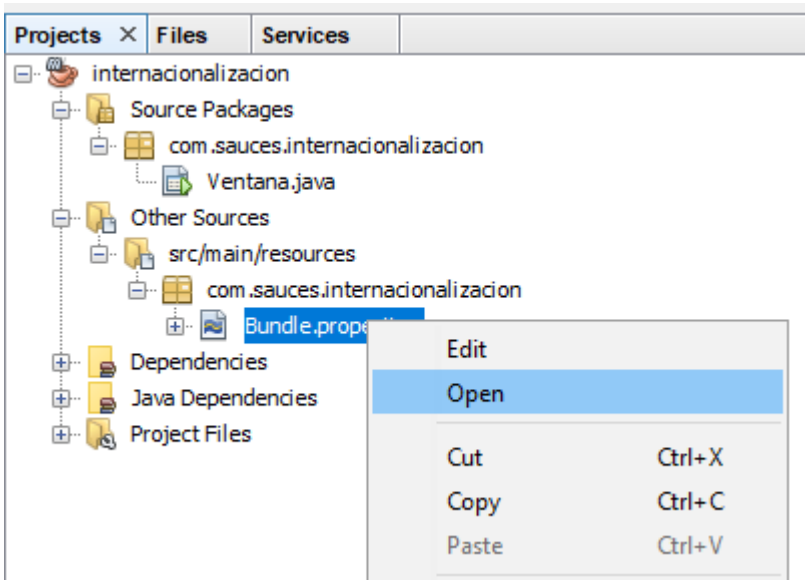
Internacionalización

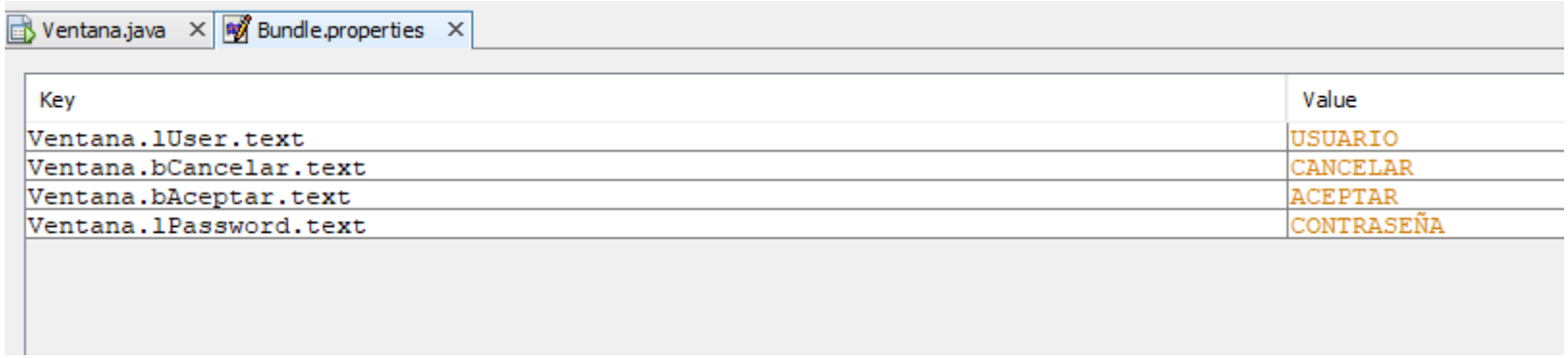




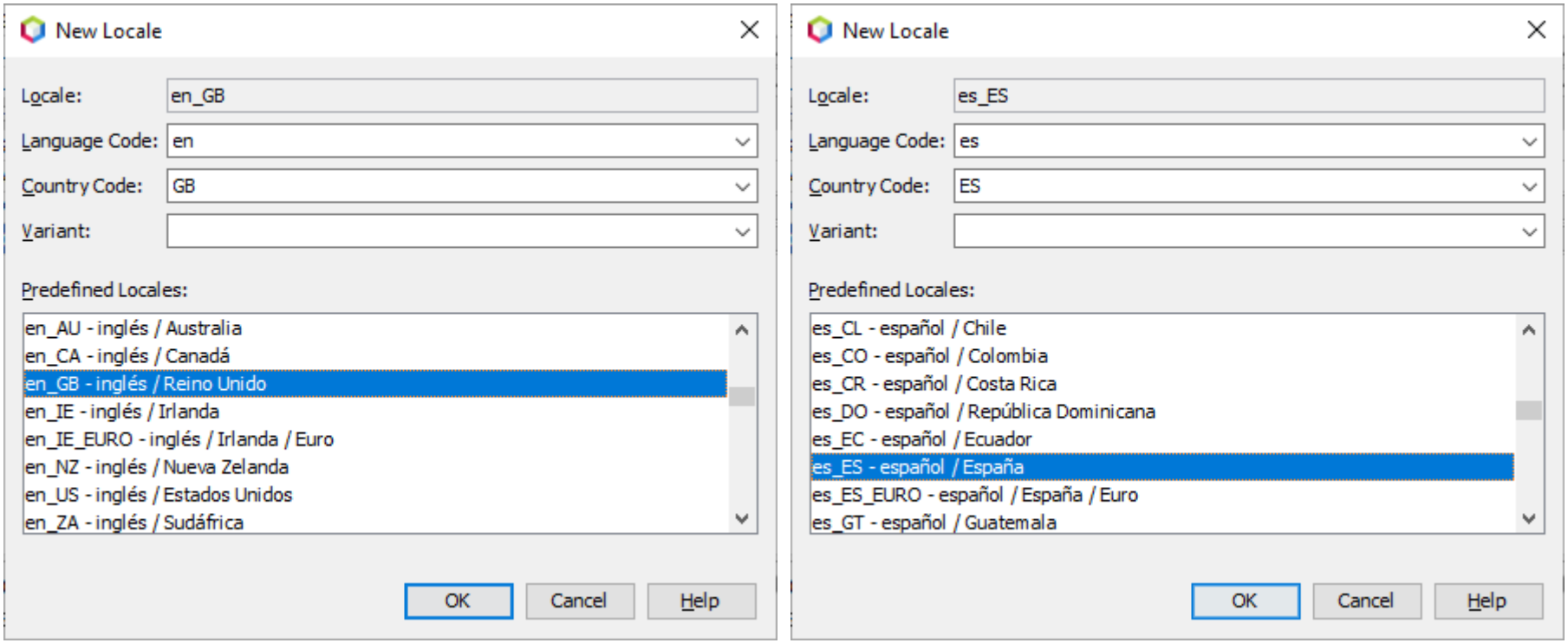


Internacionalización

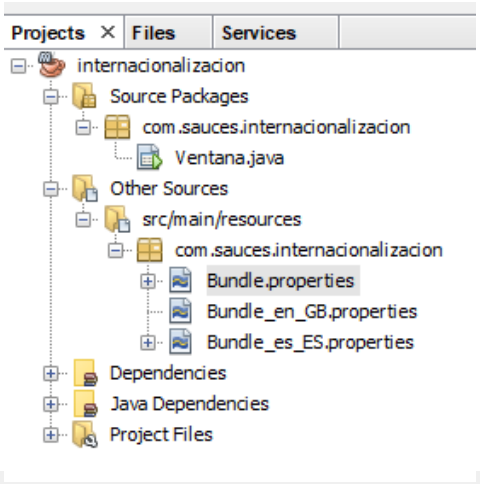




Internacionalización



Internacionalización



Ventana.java | Bundle.properties

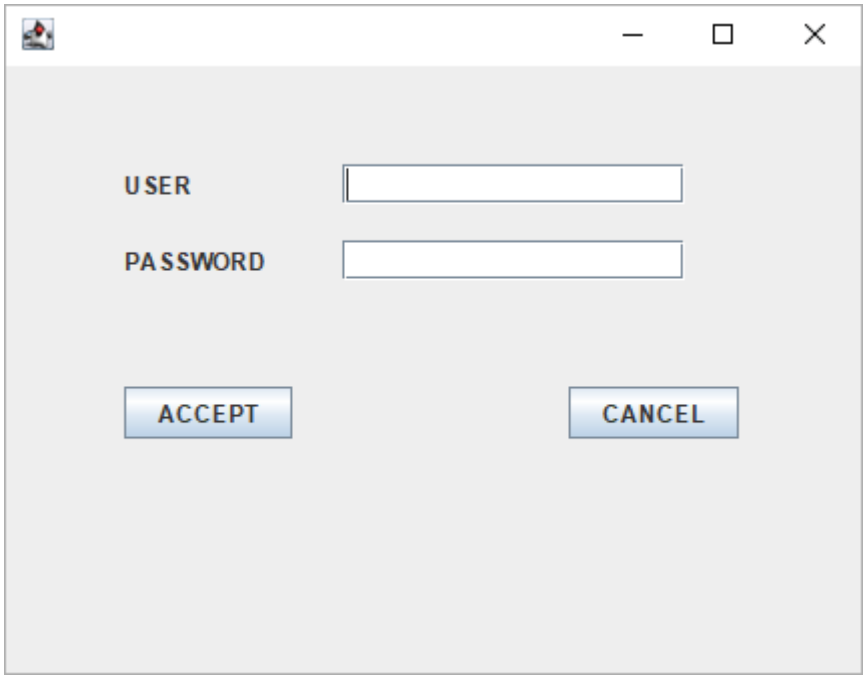
Key	default language	es_ES - español (España)	en_GB - inglés (Reino Unido)
Ventana.lUser.text	USUARIO		
Ventana.bCancelar.text	CANCELAR		
Ventana.bAceptar.text	ACEPTAR		
Ventana.lPassword.text	CONTRASEÑA		

Ventana.java | Bundle.properties

Key	default language	es_ES - español (España)	en_GB - inglés (Reino Unido)
Ventana.lUser.text	USUARIO	USUARIO	USER
Ventana.bCancelar.text	CANCELAR	CANCELAR	CANCEL
Ventana.bAceptar.text	ACEPTAR	ACEPTAR	ACCEPT
Ventana.lPassword.text	CONTRASEÑA	CONTRASEÑA	PASSWORD

Internacionalización

```
Locale.setDefault(new Locale("en_GB"));
```




```
Locale.setDefault(new Locale("es_ES"));
```