Diseño y Desarrollo de Sistemas de Información

Diseño, creación y gestión de interfaces gráficas (vistas) con la librería Swing

- Introducción
- El contenedor *JFrame*
- Componentes básicos
- Gestión de los eventos desde el controlador
- Los componentes *JPanel*, *JTable* y *JDialog*
- Gestión de eventos de ratón
- *Look and Feel*

¿Qué es Swing?

- Swing es una librería de clases que permite crear interfaces gráficas de usuario en Java

- Existen multitud de librerías para diseñar interfaces (por ejemplo, JavaFX, AWT, etc.). En este curso usaremos Swing por ser una librería ligera, estable y estar muy extendida en la comunidad de desarrolladores

- Además, la librería Swing está muy bien integrada en el IDE *NetBeans*

Esto no es un manual de Swing. Únicamente son unas nociones básicas para poder crear aplicaciones de escritorio que interactúen, de forma gráfica, con una base de datos

- Swing posee un enorme número de clases y componentes

- Nosotros estudiaremos una pequeña parte que nos permitirá construir interfaces gráficas de usuario muy completas

- Las interfaces (vistas) las crearemos mediante la herramienta de diseño incluida en *NetBeans*, lo que implica que buena parte del código será generado de forma automática

- Existen dos elementos básicos para la creación de interfaces gráficas de usuario usando Swing:
  - Contenedores. Elementos capaces de albergar otros elementos.
  - Componentes. Elementos que se añaden a los contenedores. También suelen denominarse controles

JAVA SWING

Diseño de la vista

- Para explicar los primeros conceptos de Swing, vamos a crear una vista cuya función será, simplemente, la elección del SGBD con el que trabajará nuestra aplicación

- Deberá tener un aspecto parecido a este

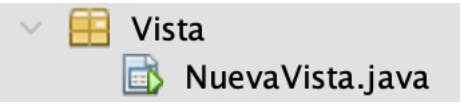
- En el paquete "Vista" seleccionamos un nuevo "Formulario JFrame" (*JFrame Form*). A la clase que se genera le llamaremos vistaLogin.java

> Los *Frames* se utilizan, generalmente, como ventanas independientes de alto nivel. La mayoría de las aplicaciones Swing se crean a partir de este tipo de contenedor

- Una vez elegido el nombre del fichero y la ubicación, aparecerá un panel "vacío" al que podremos ir añadiendo componentes desde la Paleta de Componentes

- En nuestro primer ejemplo, los componentes que tendrá la vista son:
  - Etiqueta (*JLabel*). Para mostrar texto identificativo
  - Botón (*jButton*). Componente que, al ser pulsado, lanzará eventos
  - Lista desplegable (*JComboBox*). Para seleccionar un elemento entre varias opciones

- Al crear un nuevo *JFrame*, *NetBeans* le añade un método **main()**

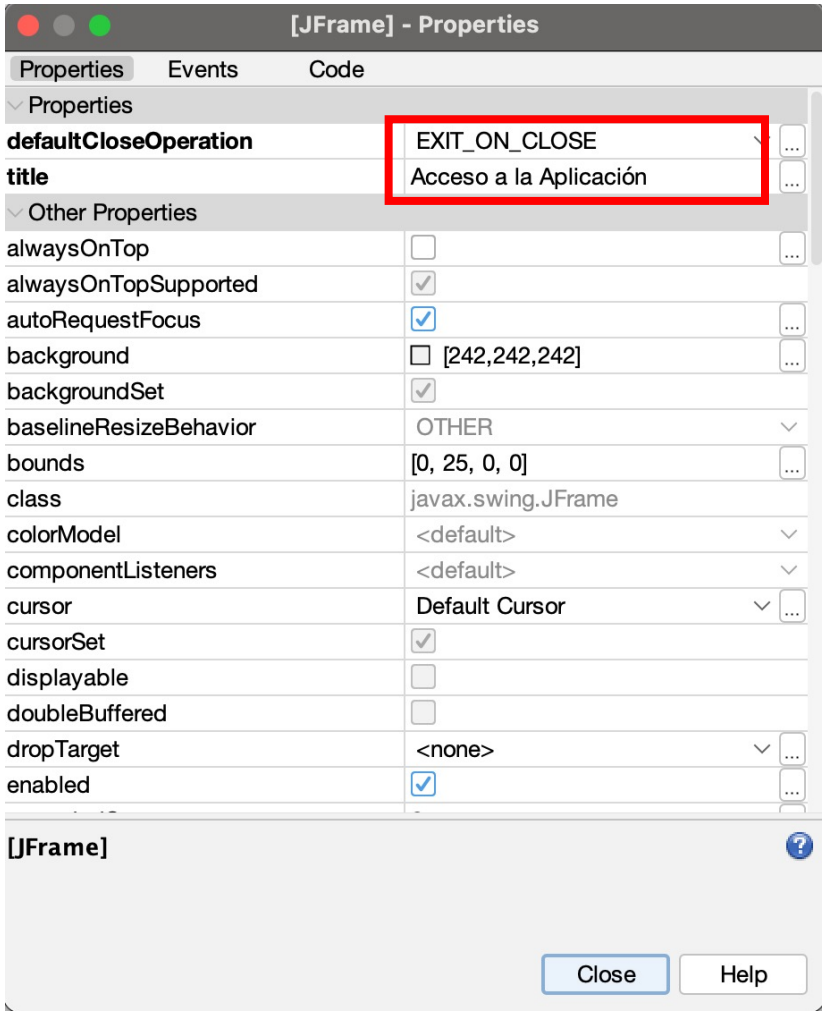


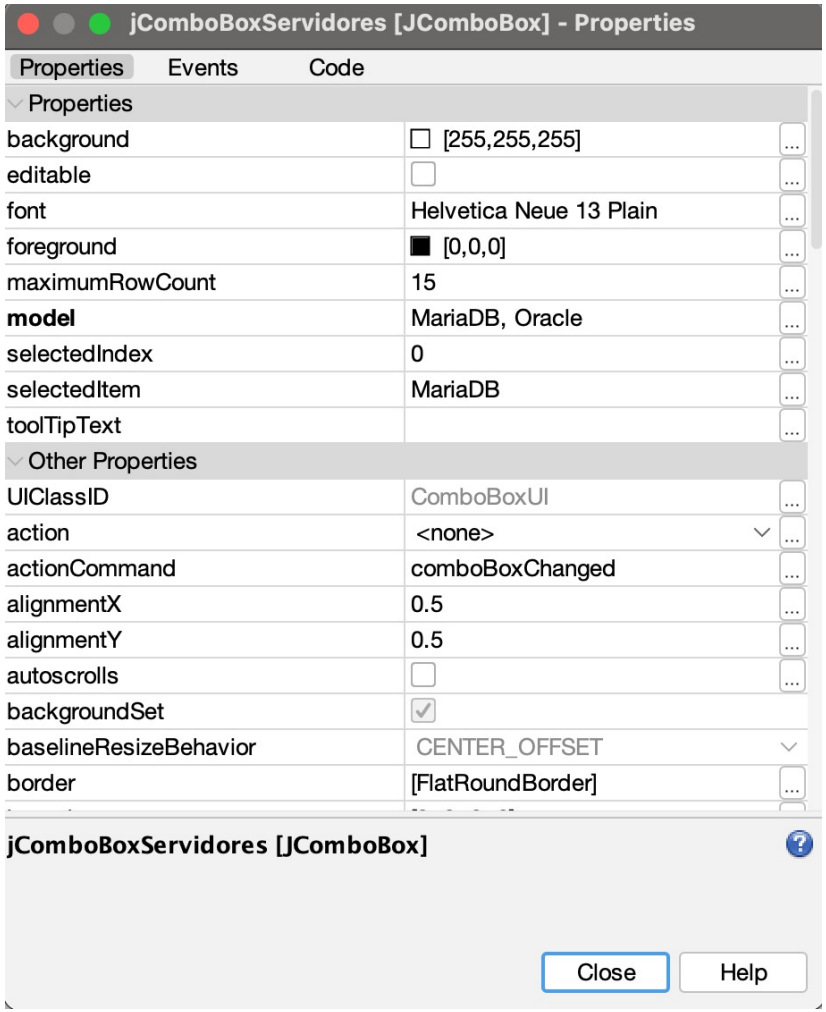

```
/**
 * @param args the command line arguments
 */
public static void main(String args[]) {
    /* Set the Nimbus look and feel */
     Look and feel setting code (optional)

    /* Create and display the form */
    java.awt.EventQueue.invokeLater(new Runnable() {
        public void run() {
            new NuevaVista().setVisible(true);
        }
    });
}
```

- En el proyecto que desarrollemos en este curso eliminaremos esta sección de código puesto que ya tenemos la clase principal en la capa de aplicación

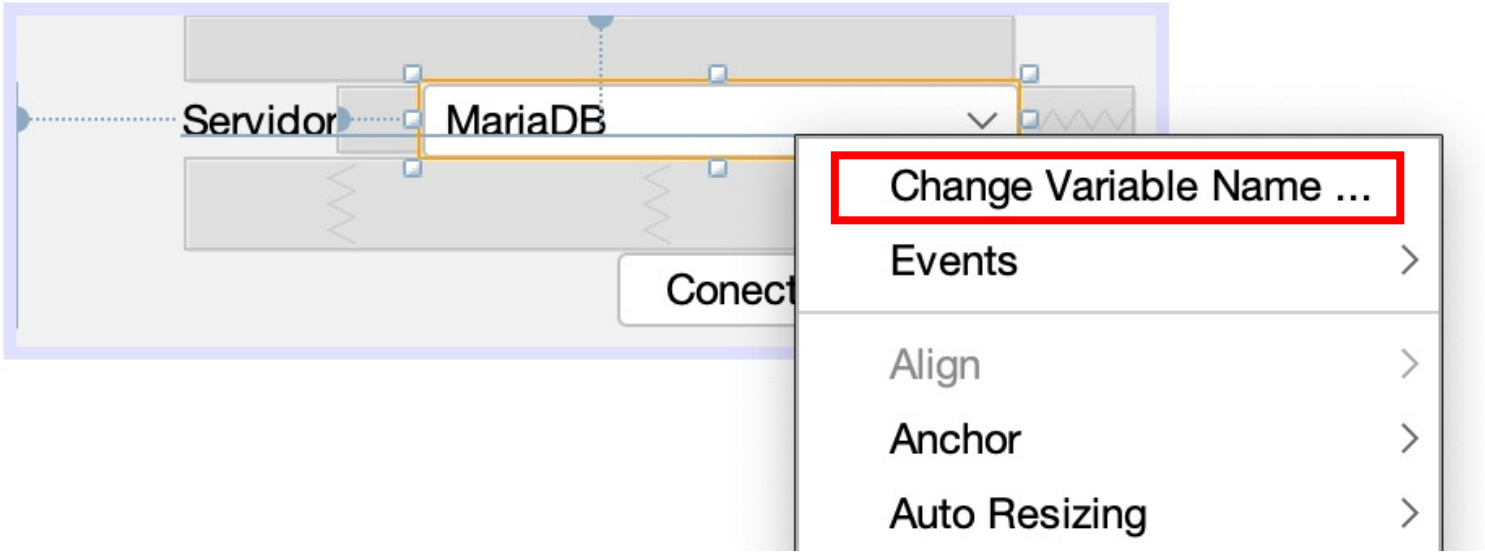
- Una vez situados los componentes, podemos definir y modificar todas sus características y propiedades (que son muchas)



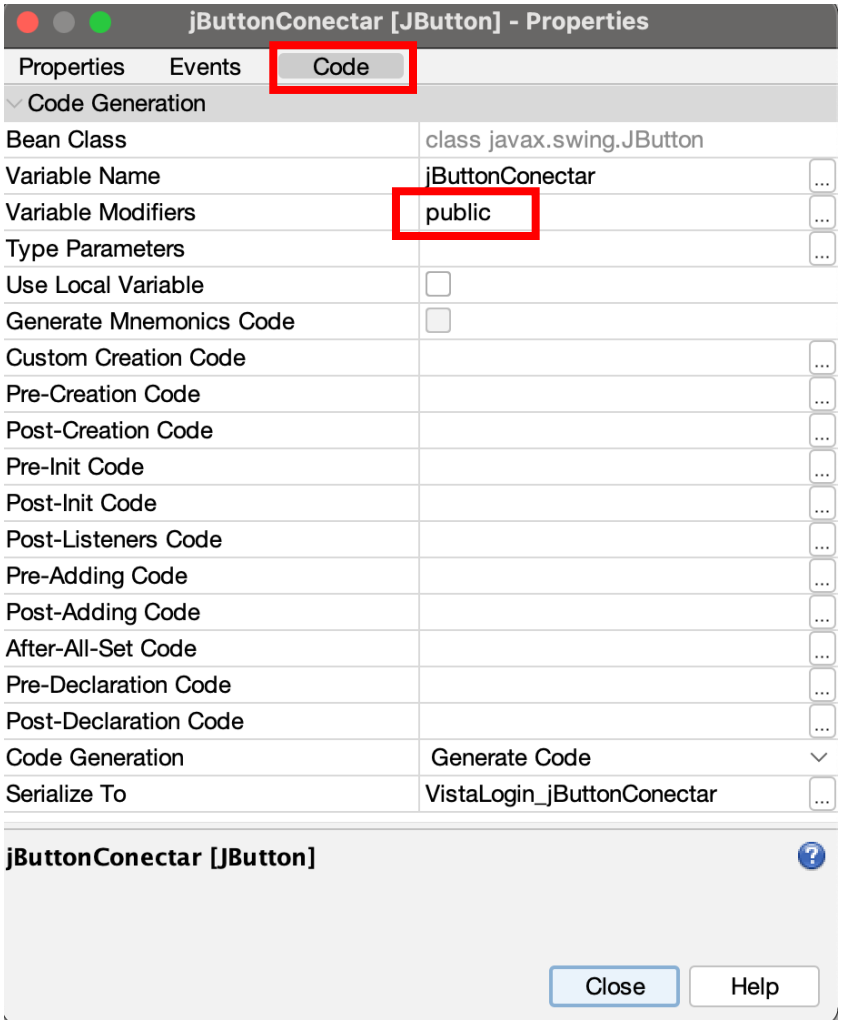


- Una de las principales acciones que debemos hacer es la de asignar al componente un nombre de variable "entendible" ya que lo tendremos que utilizar en el código muy frecuentemente

- Se puede hacer desde el menú de propiedades o desde el menú contextual con el botón derecho



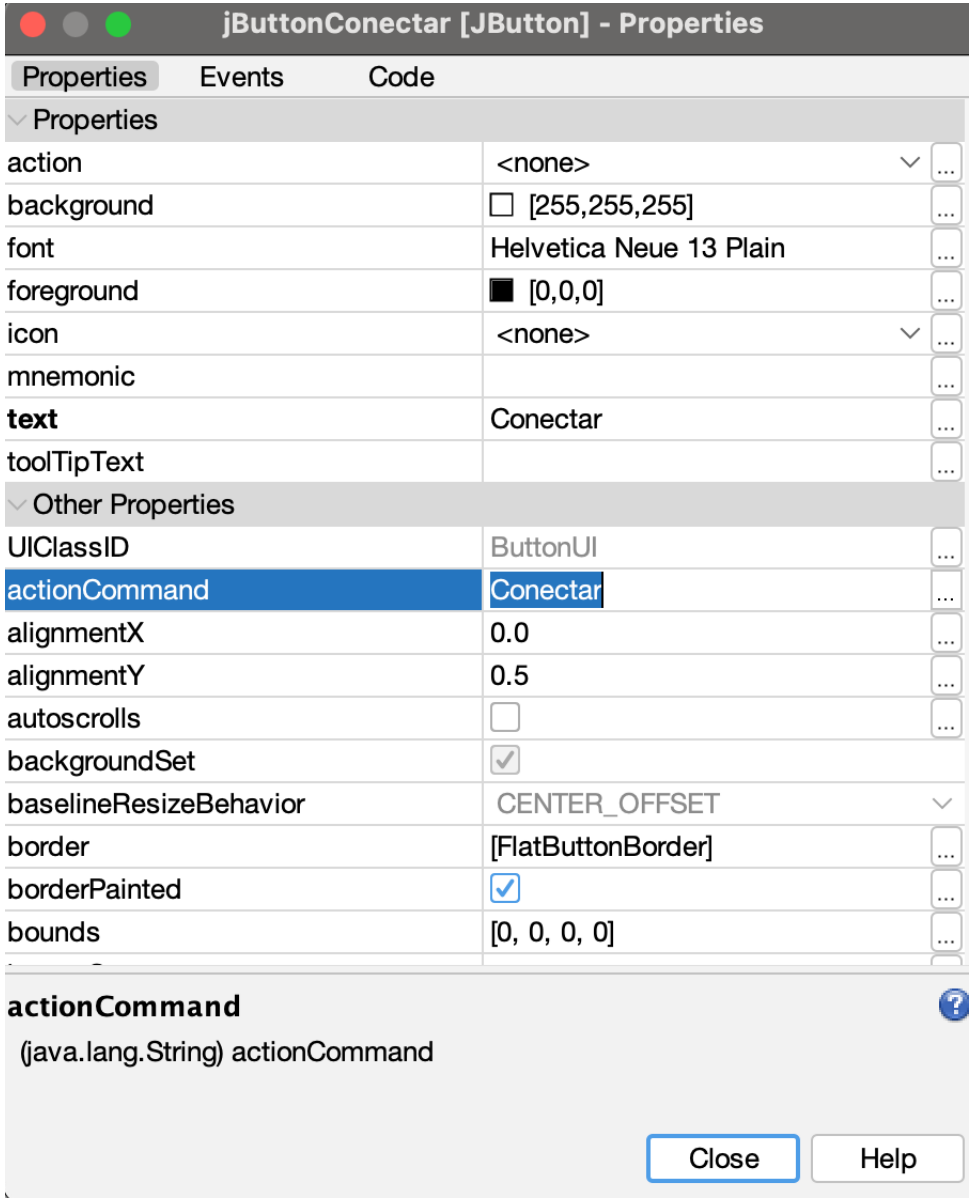
- Para que los componentes sean visibles desde la capa Controlador, debemos hacerlos públicos (por defecto son privados)



- Una propiedad muy importante en el desarrollo de nuestro proyecto es actionCommand, especialmente en aquellos componentes que van a lanzar eventos durante la ejecución de la aplicación

- Esta propiedad se utilizará en el código del controlador para saber el componente que ha lanzado un determinado evento

- Por tanto, debemos ponerle un nombre significativo que lo identifique



- Una vez diseñada la vista, si observamos el código fuente que ha generado *NetBeans* comprobaremos que se han declarado los componentes en una sección cuyo código no se puede modificar

```
// Variables declaration - do not modify
public javax.swing.JButton jButtonConectar;
public javax.swing.JButton jButtonSalirDialogoConexion;
public javax.swing.JComboBox<String> jComboBoxServidores;
private javax.swing.JLabel jLabel3;
// End of variables declaration
```

- Otra parte del código que genera *NetBeans* de forma automática es la función initComponents(), a la que se llama desde el constructor de la vista

```
public VistaLogin() {
    initComponents();
}
```

```
// <editor-fold defaultstate="collapsed" desc="Generated Code">
private void initComponents() {

    jButtonSalirDialogoConexion = new javax.swing.JButton();
    jButtonConectar = new javax.swing.JButton();
    jLabel3 = new javax.swing.JLabel();
    jComboBoxServidores = new javax.swing.JComboBox<>();

    setDefaultCloseOperation(operation: javax.swing.WindowConstants.EXIT_ON_CLOSE);
    setTitle(title: "Acceso a la Aplicación");

    jButtonSalirDialogoConexion.setText(text:"Salir");
    jButtonSalirDialogoConexion.setActionCommand(actionCommand: "SalirDialogoConexion");

    jButtonConectar.setText(text:"Conectar");

    jLabel3.setText(text:"Servidor");

    jComboBoxServidores.setModel(new javax.swing.DefaultComboBoxModel<>(new String[] { "MariaDB", "Oracle" }));
```

Gestión de eventos

- La gestión de los eventos que se producen en las vistas la realizaremos desde los controladores
- Existen varias formas de gestionar los eventos. Una de ellas consiste en implementar en los diferentes controladores el código necesario para la gestión de los eventos que vayamos a utilizar en nuestra aplicación
- En nuestro primer ejemplo sólo vamos a gestionar eventos de tipo ActionListener

```
public class ControladorLogin implements ActionListener {
```

- En el controlador se asignarán los *listener* (escuchadores) a los componentes que lanzan eventos. Suele hacerse con una función de nombre addListeners(), que se llamará desde el constructor de la clase "Controlador"

```
private void addListeners() {
    vLogin.jButtonSalirDialogoConexion.addActionListener(l: this);
    vLogin.jButtonConectar.addActionListener(l: this);
}
```

- Por último, se implementa el método actionPerformed(), de la interfaz actionListener, donde gestionamos los eventos usando la propiedad actionCommand de los componentes

La función **conectarBD()** será similar a la implementada en la práctica, con la particularidad de que la elección del servidor se hace a través de una lista desplegable

```
@Override
public void actionPerformed(ActionEvent e) {

    switch (e.getActionCommand()) {
        case "Conectar":
            conectarBD();
            vLogin.dispose();
            controladorP = new ControladorPrincipal(sessionFactory);
            break;

        case "SalirDialogoConexion":
            vLogin.dispose();
            System.exit(status: 0);
            break;

    }
}
```

Propiedad **actionCommand** del botón "Conectar"

Propiedad **actionCommand** del botón "Salir"

vLogin es un objeto de tipo **vistaLogin**

También se pueden gestionar los eventos usando el nombre de la variable del componente. En ese caso habría que utilizar el método **getSource()** y usar sentencias *if*

Por ejemplo: if (e.getSource() == vLogin.jButtonConectar)

- Siguiendo con el ejemplo, el método constructor de la clase ControladorLogin será algo así

```
public ControladorLogin() {
    vMensaje = new VistaMensajes();
    vLogin = new VistaLogin();

    addListeners();

    vLogin.setLocationRelativeTo(c: null);
    vLogin.setVisible(b: true);

    vLogin.jComboBoxServidores.setSelectedIndex(anIndex:0);
}
```

Usaremos un objeto de tipo vMensaje para mostrar mensajes

Sitúa la ventana en el centro de la pantalla

Muestra la ventana

Se selecciona el primer elemento del ComboBox

- Si la conexión es correcta, se elimina la vista (ventana) de acceso a la aplicación y se instancia un objeto de la clase Controlador, que será el controlador principal de la aplicación

```
@Override
public void actionPerformed(ActionEvent e) {

    switch (e.getActionCommand()) {
        case "Conectar":
            conectarBD();
            vLogin.dispose();
            controladorP = new ControladorPrincipal(sessionFactory);
            break;

        case "SalirDialogoConexion":
            vLogin.dispose();
            System.exit(status: 0);
            break;

    }
}
```

- Los objetos componentes tienen un conjunto grande de métodos para gestionarlos

- Por ejemplo, para capturar un valor y asignarlo a una variable, usaremos los métodos get(), que tendrán un nombre específico dependiendo del componente

```
server = (String) (vLogin.jComboBoxServidores.getSelectedItem());
```

Se almacena en la variable global ***server*** el valor del ítem seleccionado en el ComboBox

Componentes gráficos de la interfaz del proyecto

JFrame con la ventana principal de la aplicación

JMenuBar con componentes de tipo *JMenu*. Cada uno tendrá diversos componentes de tipo *JMenuItem*, que serán los que gestionen los eventos de ratón

Componentes gráficos de la interfaz del proyecto

JPanel con la ventana de la gestión de Monitores

Código	Nombre	DNI	Teléfono	Correo	Fecha Incorporación	Nick
M001	Samuel Sola Vidal	59354777B	663882935	samuel_53@lycos.es	20/08/2010	Robby
M002	Oscar Caro Salcedo	65745956L	777150614	oscar_13@hotmail.co.uk	09/05/2001	Chato
M003	Mercedes Varela Torres	78265588S	745506998	mercedes_30@libero.it	05/07/2003	Nelsa
M004	Arnau Marrero Castellano	91706729W	746720525	arnau_39@teacher.com	20/11/2000	Manny
M005	Jacobo Varela Sola	40079584Z	653233008	jacobo_79@lycos.es	04/07/2019	Waldo
M006	Francisco Antonio Camacho Benito	70784291C	796449086	franciscoantonio_88@email.com	21/07/2016	Yanko
M007	Florentina Cobos Collado	17415823Q	676252092	florentina_47@msn.com	18/03/2019	Fanny
M008	Marco Antonio Cruz León	26130141W	668570734	marcoantonio_24@teacher.com	21/06/2000	Fito
M009	Elisabet Solís Ortiz	30453132M	773861386	elisabet_47@caramail.com	26/02/2005	Ensy
M010	Asunción Alarcón Bartolomé	76855698W	783386243	asuncion_57@unforgettable.com	05/07/2015	Sasha

JPanel con la ventana de la gestión de Socios

Socio	Nombre	DNI	Fecha de Nacimiento	Teléfono	Correo	Fecha de Alta	Cat.
S001	Iria Mosquera Gil	54941721B	31/03/1977	656391774	iria_89@post.com	04/06/2016	A
S002	Jonathan Saez Gracia	46288486C	16/11/1953	782479970	jonathan_03@yahoo.com	24/08/2015	A
S003	María Fernanda Arce Peralta	65298503P	18/04/1994	783908961	mariafernanda_07@journalism.com	22/10/2015	B
S004	Alexis Catalán Frías	56612261S	19/07/1996	603232342	alexis_29@writeme.com	27/02/2004	B
S005	Adolfo Franco Galindo	92325966X	04/03/2002	707592289	adolfo_33@techie.com	17/04/2004	C
S006	José María Garzón Miranda	78504430W	11/08/1976	617707844	josemaria_79@unforgettable.com	21/07/2017	C
S007	Kevin Camacho Guzmán	00174037L	06/10/1945	700702096	kevin_88@unforgettable.com	01/04/2020	D
S008	Rosa Álvarez Crespo	78159605Q	08/06/2001	689981039	rosa_13@aim.com	06/06/2020	D
S009	Virginia de la Fuente Campos	92248499F	15/06/2003	600094259	virginia_89@scientist.com	11/03/2011	E
S010	Juan Pedro Mesa Guzmán	68401554Z	07/06/1983	775855384	juanpedro_25@lycos.co.uk	21/10/2018	E

El contenedor *JPanel*

- *JPanel* es un contenedor que puede albergar componentes gráficos
- Se utilizan, generalmente, para dividir la interfaz de usuario en áreas o secciones permitiendo crear interfaces más complejas y organizadas
- Los usaremos para diseñar las pantallas de gestión de Monitores y Socios, y se añadirán a la ventana principal (*JFrame*) de la aplicación
- Para crear un nuevo *JPanel* solo será necesario indicarlo al crear una nueva clase en el IDE
- Una vez creado, se añadirá a la ventana principal de esta forma:

```
vPrincipal.getContentPane().setLayout(new CardLayout());
vPrincipal.add(comp: pPrincipal);
vPrincipal.add(comp: vMonitor);
vPrincipal.add(comp: vSocio);
```

Usaremos el Layout *CardLayout* para poder tener más de un panel en la misma posición, mostrándolos y ocultándolos según las necesidades de la aplicación

- Este código forma parte del constructor del controlador. Previamente se habrán declarado e instanciado los objetos pPrincipal, vMonitor y vSocio, que serán de tipo *JPanel*

El contenedor *JPanel*

- Para que la primera pantalla de la aplicación aparezca "vacía" hasta que se seleccione alguna opción del menú, crearemos un *JPanel* "vacío", que será el primero que se muestre

```
pPrincipal.setVisible(aFlag: true);
vMonitor.setVisible(aFlag: false);
vSocio.setVisible(aFlag: false);
```

Diseñar una función para intercambiar los paneles

El componente *JTable*

- Un *JTable* es un componente que se utiliza para dibujar tablas

- Una de las formas más cómoda y sencilla de utilizar un *JTable* consiste en instanciar un modelo de datos (por ejemplo, *DefaultTableModel*) y asignárselo aun objeto de tipo *JTable*

- A partir de ese momento, el *JTable* y el modelo estarán asociados. Si se hace algo en el modelo, el *JTable* se "enterará" y se actualizará

```
modeloTablaMonitores = new DefaultTableModel() {
    @Override
    public boolean isCellEditable(int row, int column) {
        return false;
    }
};
vMonitor.jTableMonitores.setModel(dataModel: modeloTablaMonitores);
```

- *modeloTablaMonitores* es una variable de tipo *DefaultTableModel*
- Se sobreescribe el método *isCellEditable* para que las celdas sean no editables
- Se asigna el modelo al componente *JTable* (*jTableMonitores*)

El componente *JTable*

- Para diseñar este componente únicamente lo situaremos en el panel correspondiente. El resto del diseño y su operatividad se realizará con código
- El modelo (en nuestro caso *DefaultTableModel*) tiene métodos para modificar los datos de la tabla que contiene. Por ejemplo, añadir filas o columnas, asignar nombre a las columnas, etc.
- Ejemplo de función para "dibujar" una tabla

jTableMonitores es un objeto de tipo *JTable* que se encuentra en la vista *vMonitor*

```
public void dibujarTablaMonitores(VistaMonitores vMonitor) {
    String[] columnasTabla = {"Código", "Nombre", "DNI",
        "Teléfono", "Correo", "Fecha Incorporación", "Nick"};
    modeloTablaMonitores.setColumnIdentifiers(newIdentifiers: columnasTabla);

    // Para no permitir el redimensionamiento de las columnas con el ratón
    vMonitor.jTableMonitores.getTableHeader().setResizingAllowed(resizingAllowed:false);
    vMonitor.jTableMonitores.setAutoResizeMode(mode:JTable.AUTO_RESIZE_LAST_COLUMN);

    // Así se fija el ancho de las columnas
    vMonitor.jTableMonitores.getColumnModel().getColumn(columnIndex:0).setPreferredWidth(preferredWidth: 40);
    vMonitor.jTableMonitores.getColumnModel().getColumn(columnIndex:1).setPreferredWidth(preferredWidth: 240);
    vMonitor.jTableMonitores.getColumnModel().getColumn(columnIndex:2).setPreferredWidth(preferredWidth: 70);
    vMonitor.jTableMonitores.getColumnModel().getColumn(columnIndex:3).setPreferredWidth(preferredWidth: 70);
    vMonitor.jTableMonitores.getColumnModel().getColumn(columnIndex:4).setPreferredWidth(preferredWidth: 200);
    vMonitor.jTableMonitores.getColumnModel().getColumn(columnIndex:5).setPreferredWidth(preferredWidth: 150);
    vMonitor.jTableMonitores.getColumnModel().getColumn(columnIndex:6).setPreferredWidth(preferredWidth: 60);
}
```

El componente *JTable*

- Ejemplo de función para mostrar los datos de la tabla MONITOR a partir de una lista de monitores

```
public void rellenarTablaMonitores(ArrayList<Monitor> monitores) {
    Object[] fila = new Object[7];
    for (Monitor monitor : monitores) {
        fila[0] = monitor.getCodMonitor();
        fila[1] = monitor.getNombre();
        fila[2] = monitor.getDni();
        fila[3] = monitor.getTelefono();
        fila[4] = monitor.getCorreo();
        fila[5] = monitor.getFechaEntrada();
        fila[6] = monitor.getNick();
        modeloTablaMonitores.addRow(rowData:fila);
    }
}
```

- Ejemplo de función para vaciar el contenido de la tabla MONITOR

```
public void vaciarTablaMonitores() {
    while (modeloTablaMonitores.getRowCount() > 0) {
        modeloTablaMonitores.removeRow(row:0);
    }
}
```

Uniendo todo

- Ejemplo de la parte del controlador principal que solicita los datos de los monitores y los dibuja en su tabla

```
@Override
public void actionPerformed(ActionEvent e) {
    switch (e.getActionCommand()) {
        case "SalirAplicacion":
            vPrincipal.dispose();
            System.exit(status: 0);
            break;

        case "GestionMonitores":
            muestraPanel(panel: vMonitor);
            uTablasM.dibujarTablaMonitores(vMonitor);
            sesion = sessionFactory.openSession();
            tr = sesion.beginTransaction();
            try {
                ArrayList<Monitor> lMonitores = pideMonitores();
                uTablasM.vaciarTablaMonitores();
                uTablasM.rellenarTablaMonitores(monitores: lMonitores);
                tr.commit();
            } catch (Exception ex) {
                tr.rollback();
                vMensaje.Mensaje(C: null, tipoMensaje:"error",
                        "Error en la petición de Monitores\n" + ex.getMessage());
            } finally {
                if (sesion != null && sesion.isOpen()) {
                    sesion.close();
                }
                break;
            }
    }
}
```

Con el objetivo de tener el código limpio y bien estructurado, *dibujarTablaMonitores()*, *vaciarTablaMonitores()* y *rellenarTablaMonitores()* serán métodos de una clase llamada *UtilTablasMonitor.*

En esa misma clase también será muy útil diseñar una función *inicializarTablaMonitores(VistaMonitores vMonitor)* para asignar el modelo al *JTable* que se encuentra en la vista *vMonitor*


```
private ArrayList<Monitor> pideMonitores() throws Exception {
    ArrayList<Monitor> lMonitores = monitorDAO.listaMonitores(sesion);
    return lMonitores;
}
```