Interfases gráficas con Swing (2da. Parte)

Material recopilado y adaptado por:

Elizabeth Izquierdo

2017

Contenido

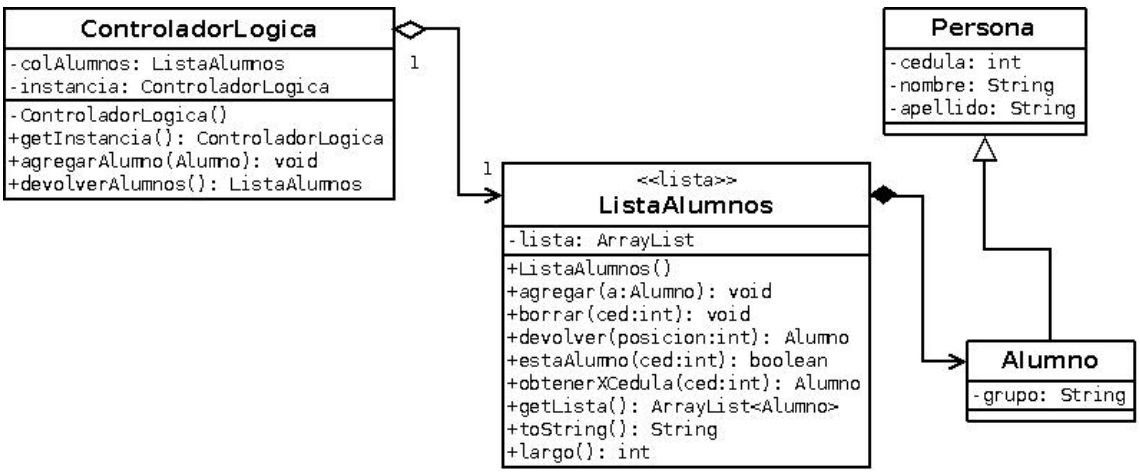
Interfaces gráficas con Swing (2da parte)

Introducción

En este tutorial aprenderemos a realizar aplicaciones con múltiples ventanas. Una ventana "padre" (con JFrame) permitirá desplegar otras ventanas "hijas" (con JDialog). Además, utilizaremos los componentes vistos en el anterior tutorial, junto con otros nuevos como por ejemplo el uso de los componentes para realizar una barra de menú con JMenuBar, Jmenu y JmenuItem. Y como realizar una tabla para poder visualizar una lista de objetos.

Por otra parte, estudiaremos (a nivel introductorio) la programación en capas. Para este caso utilizaremos un paquete que contiene las clases Persona, Alumno y una colección ListaAlumnos (vistas en la Unidad 2) las cuales se ubicarán en la capa **lógica** y cuatro ventanas (un JFrame y tres JDialog) que se ubicarán en la capa de **presentación**. Las capas en este caso las organizaremos en dos paquetes (en otros casos puede haber más de un paquete por capa), por lo tanto veremos una introducción a la interacción entre clases de distintos paquetes.

El diagrama de clases de la capa lógica queda de la siguiente manera:

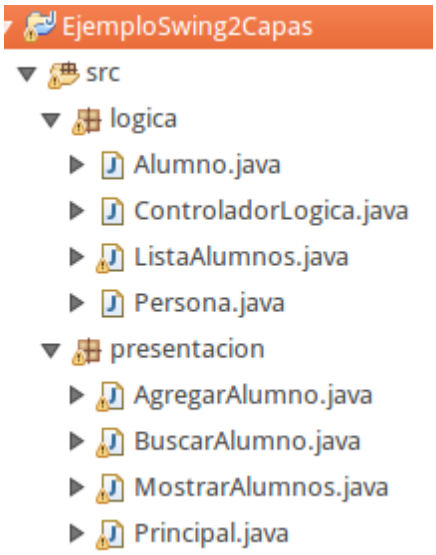


Nota: Los métodos de las clases **Persona** y **Alumno** son los habituales, por ello no se muestran en esta imagen.

Ventana con barra de menú

Armado del proyecto

Creamos un proyecto (de nombre **EjemploSwing2Capas**) con dos paquetes: *presentacion* y *logica*. En el paquete *presentacion* creamos una subclase de JFrame (de nombre **Principal**), junto con tres JDialog (**AgregarAlumno**, **BuscarAlumno** y **MostrarAlumnos**). El paquete *logica*, contendra las clases **Persona**, **Alumno**, **ListaAlumnos y ControladorLogica**. La vista del árbol de proyectos deberá quedar tal cual se ve en la siguiente figura.

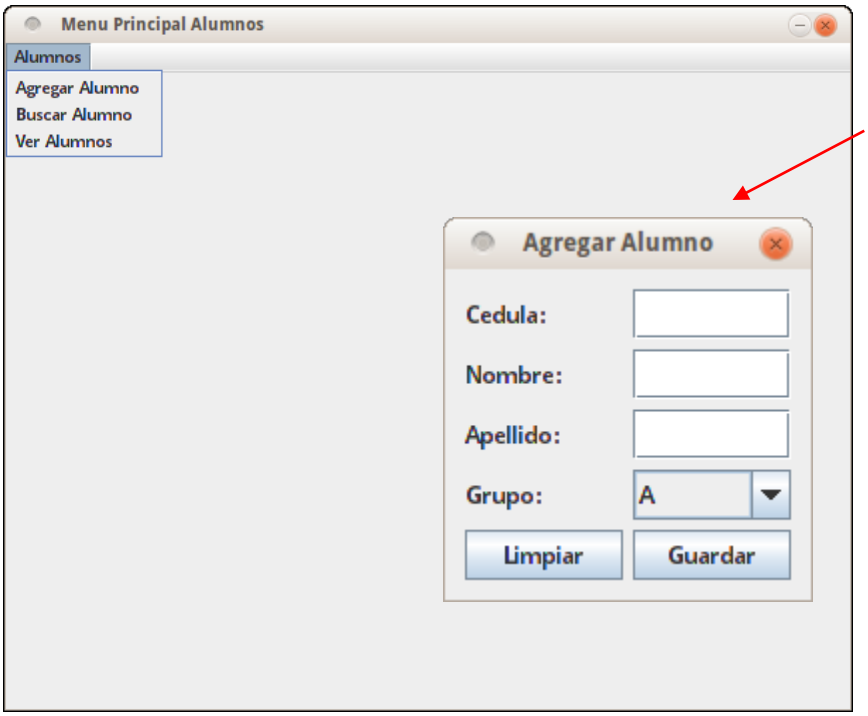


Comencemos con el código de la clase **Principal**:

```
1  /*********************************************************/
2  public class Principal extends JFrame  implements ActionListener{
3
4          //lo único que contendrá este JFrame será la barra de menú
5
6          //*****************************************************
7          private JMenuBar mnbPrincipal = new JMenuBar(); //Barra de menus
8          //*****************************************************
9
10
11   // Atributos privados
12   private  JMenuItem itemAltaAlumno;
13   private JMenuItem itemBuscarAlumno;
14   private  JMenuItem itemVerAlumnos;
15
16   private  JMenu jMenuAlumnos;
17
18
19  public Principal() {
20         setTitle("Menu Principal Alumnos");
21         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22         setResizable(false);
23         this.setBounds(200, 50,600,500);
24         iniciarComponentes();
25   }
26
27   private void iniciarComponentes() {
28
29      mnbPrincipal = new JMenuBar();//barra de menú principal
30      jMenuAlumnos = new  JMenu("Alumnos");//Menú Alumnos
31
32      //opciones de menú Alumnos
33      itemAltaAlumno = new  JMenuItem("Agregar Alumno");
34      itemAltaAlumno.addActionListener(this);
35
36      itemBuscarAlumno = new  JMenuItem("Buscar Alumno");
37      itemBuscarAlumno.addActionListener(this);
38
39
```

```
40          itemVerAlumnos= new  JMenuItem("Ver Alumnos");
41          itemVerAlumnos.addActionListener(this);
42
43          //agregamos los itemMenú  al menú
44          jMenuAlumnos.add(itemAltaAlumno);
45          jMenuAlumnos.add(itemBuscarAlumno);
46          jMenuAlumnos.add(itemVerAlumnos);
47
48          //agregamos el menú Alumnos a la barra de menú
49          mnbPrincipal.add(jMenuAlumnos);
50
51          //colocamos el menú principal en el JFrame
52          setJMenuBar(mnbPrincipal);
53
54      }
55
56          @Override
57          public void actionPerformed(ActionEvent e) {
58                  if (e.getSource()==itemAltaAlumno){
59                          AgregarAlumno altaNac = new AgregarAlumno();
60                          altaNac.setVisible(true);
61                  }
62                  if (e.getSource()==itemBuscarAlumno){
63                          BuscarAlumno busAlu = new BuscarAlumno();
64                          busAlu.setVisible(true);
65                  }
66                  if (e.getSource()==itemVerAlumnos){
67                          MostrarAlumnosTable tablaAlu = new MostrarAlumnosTable();
68                          tablaAlu.setVisible(true);
69                  }
70           }
71
72      public static void main(String args[]) {
73          Principal m=new Principal();
74          m.setVisible(true);
75      }
76
77      /**********************************************************/
```

Con el código anterior tenemos un ventana que contiene una barra de menú con un solo menú, el menú Alumnos; éste menú tiene las opciones que puede ver en la siguiente figura:



Al hacer clic en **Agregar Alumno** aparece la ventana: **Agregar Alumno**.

Su diseño, es simple, y ya analizaremos sus eventos, pero antes observemos la línea 61: **AgregarAlumno altaNac = new AgregarAlumno();**

Cada vez que, en el menú presionamos **Agregar Alumno** se instancia un nuevo objeto de esta clase, para evitar éste hecho y poder reutilizar el mismo objeto todas las veces, aplicaremos un **patrón de diseño**, el **patrón Singleton**. Veamos que es.

Patrones de diseño

Dice Wikipedia: *"Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software (...). Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias".*

Existen cantidades de patrones, dado que el nuestro es un curso introductorio a la Programación Orientada a Objetos, analizaremos dos, **Singleton y Facade**.

Patrón Singleton (instancia única)

Este patrón es uno de los más utilizados y es muy común encontrarlo por toda la bibliografía sobre Java.

Definición: Un Singleton es una clase de la que tan sólo puede haber una única instancia.

Se puede afrontar este problema de varias formas, veremos una que es la siguiente:

Crear el Singleton con un método estático : Esta aproximación lo que hace es hacer privado el constructor de la clase de manera que la única forma de conseguir una instancia de la misma sea con un método estático.

Volviendo a nuestra aplicación, ésta permite crear infinidad de objetos del mismo tipo, cuando en realidad se necesita que: Si el objeto de la ventana no fue creado, entonces crearlo, caso contrario, seguir utilizando la misma instancia (objeto) de la ventana.
Lo que se hace en realidad es crear la ventana mediante un método. Dicho método invoca al constructor si el objeto es null (si no fue creado) caso contrario devuelve la instancia. El constructor es declarado como privado, controlando que la creación del objeto sea mediante el método mencionado. Debemos modificar **Agregar Alumno** y luego **Principal**.

```
1  public class AgregarAlumno extends JDialog implements ActionListener {
2
3          //Aplicando patrón Singleton
4          //El único objeto (instancia) de AgregarAlumno durante toda la ejecución
5          private static AgregarAlumno instancia;
6
7          //resto de atributos
8
9          //constructor privado!!!  ojo
10         private AgregarAlumno(){
11                 //cuerpo del constructor igual que siempre
12         }
13
14         //salvo la primera vez (null), siempre devuelve la misma instancia
15         public static AgregarAlumno getInstancia(){
16
17                 if (instancia==null)
18                         instancia = new AgregarAlumno();
19                 return instancia;
20         }
21
22         //Sigue la clase con sus restantes metodos...
23
24
25
26
```

La clase contiene un objeto de sí misma, que será creado una sola vez (si la instancia es null), luego devuelve siempre la misma ventana. El constructor debe ser privado, ya que no se debe permitir crear el objeto con él mismo, sólo con la invocación del método getInstancia().

En la clase **Principal**, modificamos el evento relacionado con **AgregarAlumno**:

```
1  public void actionPerformed(ActionEvent e) {
2          if (e.getSource()==itemAltaAlumno){
3                  //getInstancia es estático... por lo tanto se invoca por medio de la Clase
4                          AgregarAlumno altaAlu = AgregarAlumno.getInstancia();
5                          altaAlu.setVisible(true);
6          }
7
```

Ahora la diferencia es que tenemos que invocar al método estático **getInstancia()** mediante la clase **AgregarAlumno**, para poder crear un objeto, ya que el constructor es privado.

Nota: la misma idea hay que aplicar al resto de las ventanas, para tener siempre una misma instancia de cada una.

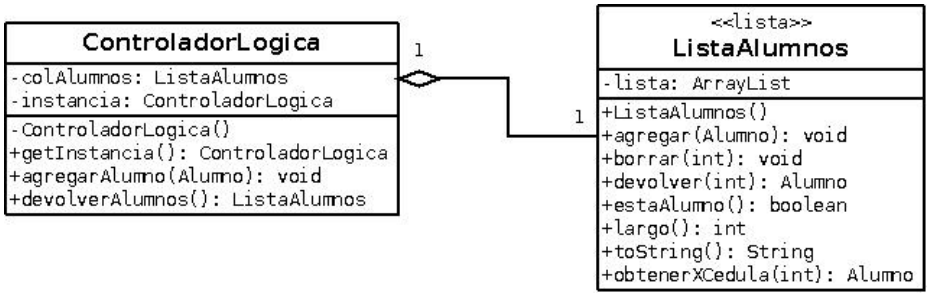
Antes de ver como codificar el evento Guardar en la ventana **Agregar Alumno** veremos otro patrón de diseño el patrón **Facade** (o Fachada), que utilizaremos para comunicarnos desde el paquete *presentación* con el paquete *logica*.

Trabajando en dos capas

Patrón Facade (ControladorLogica)

Dice Wikipedia*: "Es un tipo de patrón de diseño estructural. Viene motivado por la necesidad de estructurar un entorno de programación y reducir su complejidad con la división en subsistemas, minimizando las comunicaciones y dependencias entre estos".*

Facade (en nuestro ejemplo **ControladorLogica**) será una clase que permita encapsular el acceso a la capa lógica, siendo el único punto de comunicación entre las distintas capas, en este caso, la capa lógica y la de presentación.
Sus métodos serán tales que permitan resolver todos los requerimientos del sistema, mientras que sus atributos serán aquellas Colecciones de Objetos que impliquen un punto de partida para la resolución de los requerimientos.
Para asegurarnos de que sea el único punto de comunicación, se aplica el patrón de diseño Singleton a la clase ControladorLogica. Por ello, ésta clase, además de tener como atributo una atributo de tipo ListaAlumnos tiene un atributo de tipo ControladorLogica.



La clase ControladorLogica

```
1   package logica;
2
3   public class ControladorLogica {
4           //atributos
5      private  ListaAlumnos colAlumnos;
6      private static ControladorLogica instancia;          //atributo Singleton
7
8
9      //constructor  privado
10     private ControladorLogica(){
11             colAlumnos=new ListaAlumnos();
12     }
13
14     //Singleton metodo getInstancia
15     public static ControladorLogica getInstancia(){
16             if (instancia==null)
17                     instancia=new ControladorLogica();
18             return instancia;
19             }
20
21
22     // Agrego el Alumno a la coleccion.
23     public  void  agregarAlumno(Alumno a) {
24        colAlumnos.agregar(a);
25     }
26
27     //retorna la lista de alumnos
28     public  ListaAlumnos devolverAlumnos() {
29        return (colAlumnos);
30     }
31  }
32
    /*****************************************************/
```

La clase **ControladorLogica** implementa Singleton. La instancia de ControladorLogica será la que tenga permanentemente la lista de Alumnos cargada en memoria RAM, tanto la operación agregarAlumno como devolverAlumno de la lista de Alumnos, deberán ser invocadas desde el método getInstancia() de ControladorLogica (como ya analizamos en la ventana AgregarAlumno). Una ventaja de aplicar este patrón es que siempre tendremos el control de la colección (o colecciones) con las cuales estemos trabajando, todas en una misma clase.

Ahora sí, estamos prontos para poder entender como codificar el evento del botón **Guardar** de la ventana **Agregar Alumno**

```
1   @Override
2   public void actionPerformed(ActionEvent evento) {
3           if (evento.getSource()==btnGuardar){
4               Alumno alumno1 =new Alumno();
5               try{
6                   //capturamos los datos de la ventana
7                    int auxCed=Integer.parseInt(txtCedula.getText());
8                    String auxNom=txtNombre.getText();
9                    String auxApe=txtApellido.getText();
10                   String auxGrupo= cmbGrupo.getSelectedItem().toString();
11                  //los datos capturados los seteamos en el objeto alumno1
12                   alumno1.setCedula(auxCed);
13                   alumno1.setNombre(auxNom);
14                   alumno1.setApellido(auxApe);
15                   alumno1.setGrupo(auxGrupo);
16                   JOptionPane.showMessageDialog(null, "Datos guardados correctamente
17                           \n","Mensaje ",JOptionPane.INFORMATION_MESSAGE);
18              }catch (Exception e){
19                           System.out.println("el error fue: " +e);
20                           JOptionPane.showMessageDialog(null, "Datos incorrectos, intente de
21                               nuevo  \n","Error",JOptionPane.ERROR_MESSAGE);
22              }
23          //agrego el alumno1 a la lista de alumnos, a la que accedo a través del controlador
24          ControladorLogica.getInstancia().agregarAlumno(alumno1);
25
26      }
```

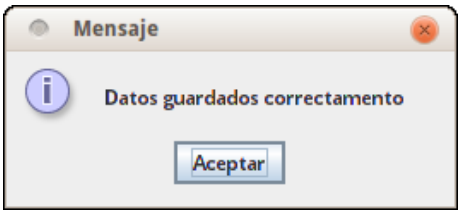
Luego de llenar los datos de la ventana **Agregar Alumno**, cuando se presione el botón **Guardar**, lo que debe suceder es que se guarde un objeto de tipo Alumno en una lista (qué más adelante guardaremos en la base de datos, por ahora solo en memoria).

Una vez que tenemos el objeto alumno1 seteado, lo guardaremos en la lista (línea 24) **ControladorLogica.*getInstancia*().agregarAlumno(alumno1);**

Como ya se mencionó, getInstancia() devuelve la única instancia de la clase. Dicha instancia contiene los métodos declarados en el controlador, en este caso agregarAlumno, agrega un Alumno a la colección estática de Alumnos, ubicada en ControladorLogica.

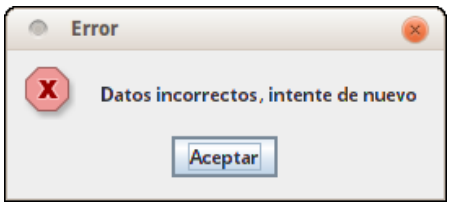
Por otra parte, cuando el usuario ingresa los datos puede cometer algunos errores, por ejemplo ingresar caracteres no numéricos en el campo Cédula, esto hará que nuestro programa termine si no lo colocamos en un bloque try – catch (líneas 5 a 18). [1]

Si el usuario ingresa datos adecuados, aparecerá un cuadro de diálogos como el siguiente



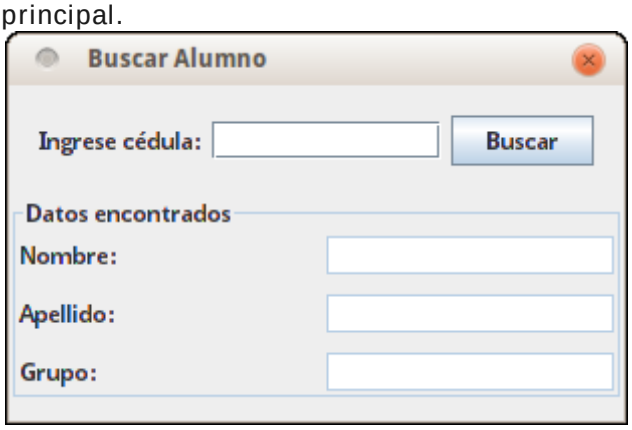
[1] Ampliación de try-catch en apuntes Introducción al manejo de Excepciones

Pero si ocurre una excepción aparecerá este otro cuadro:



que aceptándolo, tendremos chance de solucionar el problema ingresando un número en el campo cédula por ejemplo.

A continuación veamos como implementar el botón **Buscar** de la ventana **Buscar Alumno** que aparece cuando elegimos la opción **Buscar Alumno** del menú principal.



En este caso, el único dato que se ingresa desde el teclado es una cédula. Los JtextField Nombre, Apellido y Grupo están deshabilitados, en ellos aparece los datos correspondientes a la cédula ingresada en caso de existir, sino aparece un cuadro de diálogos informando que dicho Alumno no se encuentra.

```
1   @Override
2   public  void actionPerformed(ActionEvent evento) {
3           ListadoAlumnos listado;
4           Alumno aux=new Alumno();
5           int auxCedula = 0;
6
7           if (evento.getSource()==btnBuscar){
8            //captura la cedula ingresada
9           try{
10              auxCedula=Integer.parseInt(txtCedula.getText());
11          }catch (Exception e1){
12              JOptionPane.showMessageDialog(null, "La cédula debe ser
13                      numérica  \n");
14          }
15          Listado= ControladorLogica.getInstancia().devolverAlumnos();
16          aux=listado.obtenerXCedula(auxCedula);
17          if (aux!=null){
18                txtNombre.setText(aux.getNombre());
19                txtApellido.setText(aux.getApellido());
20                txtGrupo.setText(aux.getGrupo());
21          }else
22                JOptionPane.showMessageDialog(null, "Alumno no encontrado \n");
23          }
24  }
25
26
```

Lo primero que hace es capturar la cedula en una variable, luego (línea 15) obtiene la lista de alumnos a través del ControladorLogica y obtiene el alumno que coincida con la cedula (línea 16).

Si lo encuentra, llena los JtextField Nombre, Apellido y Grupo. Sino informa que no lo encontró.

Creando tablas

Al hacer clic en **Ver Alumnos** de nuestro menú principal aparecerá una tabla en donde en cada fila se verán los datos de cada alumno, como se aprecia en la siguiente imagen.

Para ello lo que se hace es crear una instancia de **MostrarAlumnos**, veamos a continuación su código.

```
1  package presentacion;
2
3  import logica.*; //importamos todas las clases del paquete logica
4
5  import  javax.swing.BorderFactory;
6  import  javax.swing.JDialog;
7  import  javax.swing.JPanel;
8  import  javax.swing.JTable;
9  import  java.awt.BorderLayout;
10
11 import  javax.swing.JScrollPane;
12 import  javax.swing.table.DefaultTableModel;
13
14 public class MostrarAlumnos extends JDialog {
15
16    private JPanel pnlCentral;
17    private DefaultTableModel  miModelo;   // Nuestro modelo para la tabla
18    private JTable tablaAlumnos;
19    private JScrollPane scrPanel;    //barra de desplazamiento horizontal y vertical
20
21    public MostrarAlumnos(){  //constructor
22
23          setTitle("Listado de Alumnos");
24          setLocationRelativeTo(null);
25          setResizable(true);
26          iniciarComponentes();
27          iniciarTabla();
28          ListaAlumnos listado=ControladorLogica.getInstancia().devolverAlumnos();
29
30          //para cada posicion de la fila
31
32
```

```
33          String fila[]=new String[miModelo.getColumnCount()];
34
35           try{// Cada iteración agrega un Alumno a la fila "i" de la tabla
36                for(int i=0;i<listado.largo();i++){
37                     fila[0]=String.valueOf(listado.devolver(i).getCedula());
38                     fila[1]=listado.devolver(i).getNombre();
39                     fila[2]=listado.devolver(i).getApellido();
40                     fila[3]=listado.devolver(i).getGrupo();
41                     miModelo.addRow(fila);
42                }
43                 // Asociar el modelo con la lista.
44                tablaAlumnos.setModel(miModelo);
45          }catch (Exception e) {
46              System.out.println("error"+e);
47          }
48    }//fin constructor
49
50    public  void  iniciarComponentes(){
51
52
53       pnlCentral =new JPanel();
54       pnlCentral.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
55       pnlCentral.setLayout(new BorderLayout());
56
57      /*Instanciamos un scroll y lo agregamos al panel*/
58       scrPanel = new JScrollPane();
59       pnlCentral.add(scrPanel, BorderLayout.CENTER);
60
61       /*Instanciamos una tabla y le asociamos el scroll*/
62       tablaAlumnos = new JTable();
63       scrPanel.setViewportView(tablaAlumnos);
64
65       //agregamos el panel a la zona correspondiendte
66       this.add(pnlCentral, BorderLayout.CENTER);
67       pack();
68    }
69
70     public void  iniciarTabla(){
71
72       /*definimos titulos de la tabla en un array de String*/
73       String titulos[]={"Cedula","Nombre", "Apellido","Grupo"};
74       /*Instanciamos un modelo de tabla con los titulos definidos*/
75       miModelo=new DefaultTableModel(null,titulos);
76      /* la tabla tomará el valor de miModelo */
77      tablaAlumnos.setModel(miModelo);
78    }
79 }
80
81
```

Debemos distinguir dos elementos a la hora de implementar JTable. Por un lado tenemos el JTable, o sea, el componente que se visualiza en pantalla. Por otro lado tenemos el modelo, el cual es el que determina la configuración de la tabla (nombre de columnas, contenido de la fila, entre otros). Esta información se guarda en un objeto del tipo TableModel. La tabla lo que hace es tomar dicho objeto, para visualizar la información.

JTable es una herramienta visual de Java que sirve para poder dibujar tablas, con sus respectivas filas y columnas en donde puedes ingresar el dato que tu desees, como por ejemplo: tu nombre, tu apellido, entre otros.

Para poder usar bien la herramienta **JTable**, es necesario trabajar con otras herramientas que están vinculadas a su desarrollo, para lograr tener un mejor funcionamiento en el momento de implementar una tabla. Estas herramientas son: **DefaultTableModel y JScrollPane**

El **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. Para utilizar DefaultTableModel debemos importarla y luego declararla para luego poder usar la clase JTable, un ejemplo es:

```
DefaultTableModel modelo = new DefaultTableModel();

JTable tabla = new JTable (modelo);
```

El **JScrollPane**, es una clase importada en Swing que admite asociarle una pequeña vista o ventana deslizable o corrediza, que permite solo que se vea una parte de dicho complemento en la tabla. Para poder usarlo, importamos la clase y luego la declaramos, un ejemplo sería:

```
JSrollPane scrollpane = new JScrollPane(tabla);
```

En nuestro código en las líneas 18, 19 y 20 definimos el modelo, la tabla y el scroll, luego en el constructor de la ventana, dentro del método iniciarComponentes, (líneas 58 a 62) Instanciamos un scroll y lo agregamos al panel y instanciamos una tabla y le asociamos el scroll.

Además se llama a un método iniciarTabla (línea 28) que lo que hace es definir los títulos de las columnas en un string (líneas 75 a 77), instanciamos un modelo y luego seteamos la tabla con dicho modelo.

Finalmente obtenemos el listado de alumnos a través del controlador (línea 29) y con un for (por cada objeto de la lista, creamos array de String (de largo, la cantidad de columnas de la tabla) y en cada iteración agrega un Alumno a la fila “i” de la tabla. Agregando cada fila al modelo (línea 40).