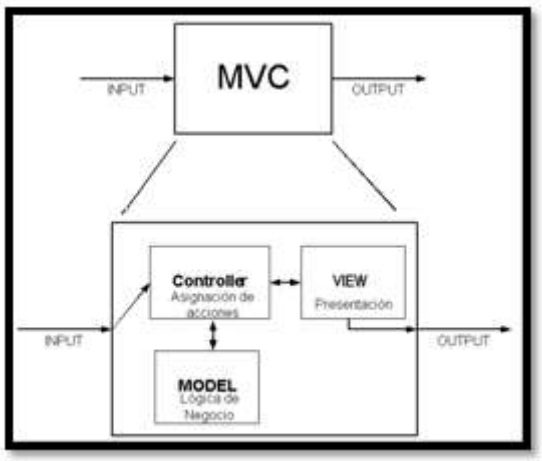
Tema 11: GUI

Patrón Modelo, Vista, Controlador

¿Qué es y en donde se utiliza más frecuentemente el Modelo Vista Controlador?

Modelo Vista Controlador (MVC) es un patrón de arquitectura de software que separa **los datos de una aplicación, la interfaz de usuario, y la lógica de control** en tres componentes distintos. La finalidad del modelo es mejorar la **reusabilidad** por medio del desacople entre la vista y el modelo. Los elementos del patrón son los siguientes:



El modelo es el responsable de:

- Acceder a la capa de almacenamiento de datos. Lo ideal es que el modelo sea independiente del sistema de almacenamiento.

- Define las reglas de negocio (la funcionalidad del sistema). Un ejemplo de regla puede ser: “Si la mercancía pedida no está en el almacén, consultar el tiempo de entrega estándar del proveedor”.

- Lleva un registro de las vistas y controladores del sistema.

- Si estamos ante un modelo activo, notificará a las vistas los cambios que en los datos pueda producir un agente externo (por ejemplo, un fichero que actualiza los datos, un temporizador que desencadena una inserción, etc...).

El controlador es el responsable de:

- Recibe los eventos de entrada (un clic, un cambio en un campo de texto, etc.).

- Contiene reglas de gestión de eventos, del tipo “SI Evento Z, entonces Acción W”. Estas acciones pueden suponer peticiones al modelo o a las vistas. Una de estas peticiones a las vistas puede ser una llamada al método “Actualizar()”. Una petición al modelo puede ser “Obtener_tiempo_de_entrega(nueva_orden_de_venta)”.

Las vistas son responsables de:

- Recibir datos del modelo y mostrarlos al usuario.

- Tienen un registro de su controlador asociado (normalmente porque además lo instancia).

- Pueden dar el servicio de "Actualización()", para que sea invocado por el controlador o por el modelo (cuando es un modelo activo que informa de los cambios en los datos producidos por otros agentes).

¿Qué Ventajas trae utilizar el MVC?

Es posible tener diferentes vistas para un mismo modelo (p.e. representación de un conjunto de datos como una tabla o como un diagrama de barras). Además es posible construir nuevas vistas sin necesidad de modificar el modelo subyacente.
Proporciona un mecanismo de configuración a componentes complejos muchos más tratable que el puramente basado en eventos (el modelo puede verse como una representación estructurada del estado de la interacción).

Flujo que sigue el control en una implementación general de un MVC

Aunque se pueden encontrar diferentes implementaciones de MVC, el flujo que sigue el control generalmente es el siguiente:

- El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace).

- El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (handler).

- El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario). Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.

- El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en el modelo.

- La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

Construcción de GUI en Java

Cualquier lenguaje de programación moderno ofrece herramientas para la construcción de interfaces gráficas de usuario (GUI).Java permite al programador el diseño y programación de interfaces gráficas de usuario de forma rápida y sencilla mediante el uso de las clases contenidas en:

- El paquete de clases AWT (Abstract Window Toolkit).

- El paquete de clases Swing. Swing es una evolución de AWT, ofreciendo más clases y una mayor flexibilidad.

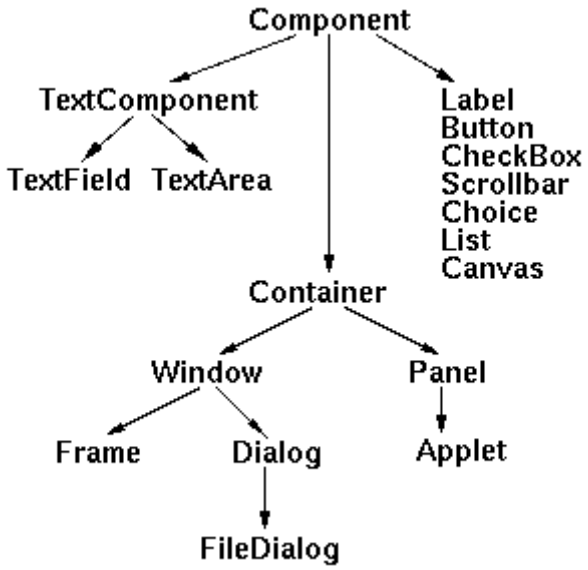
Por cada elemento de AWT existe un elemento en el sistema operativo que lo representa. El resultado final dependerá de este elemento. Sin embargo hay facilidades que algún sistema operativo no tiene por lo que AWT define lo mínimo común. Esto se solucionó con la implementación de Swing, que define lo máximo. Para trabajar con Swing necesitaremos los paquetes (y subpaquetes) de java.awt.* y javax.swing.*.

Elementos de Swing

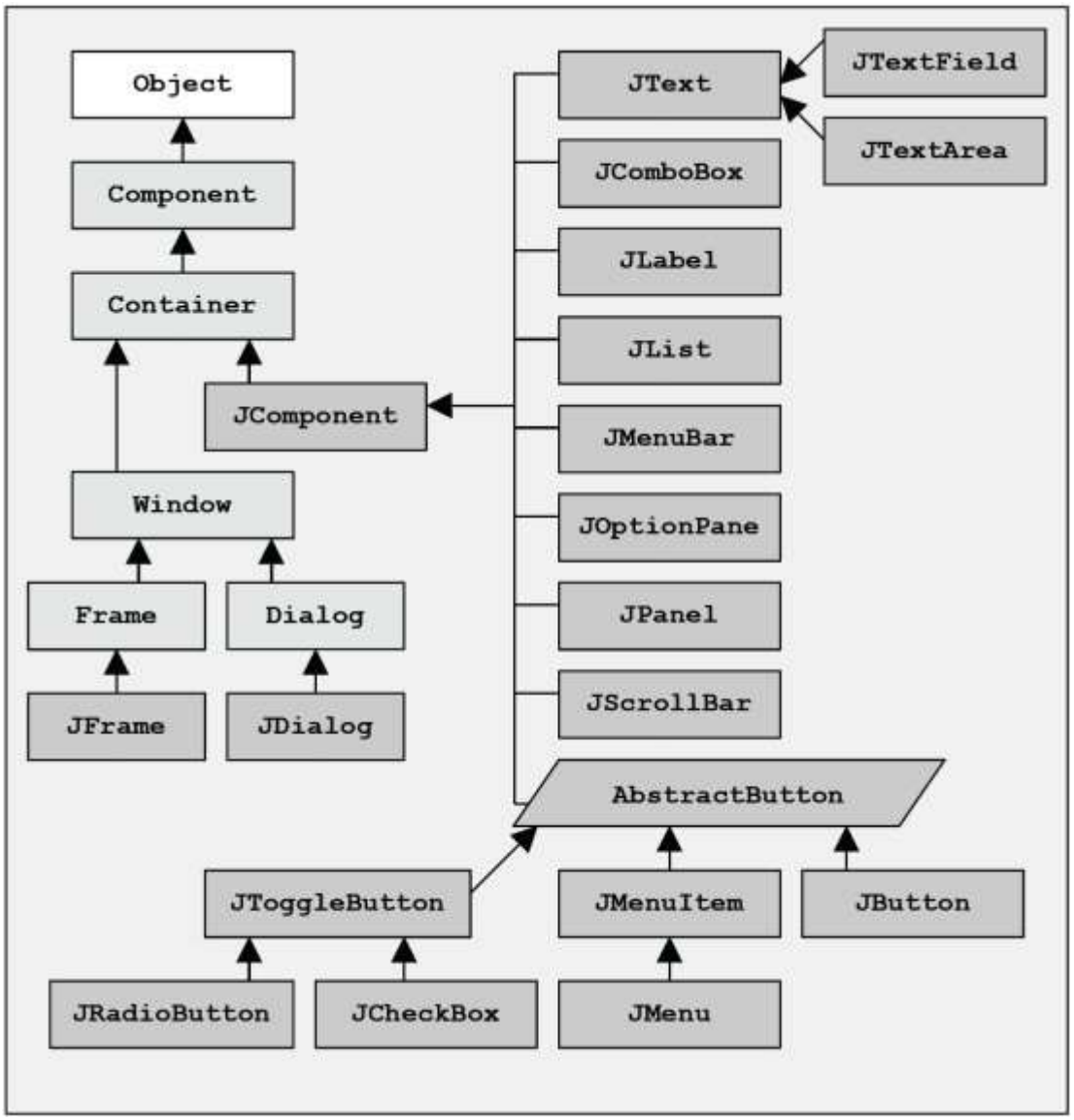
En Swing encontramos componentes y contenedores. Los componentes se sitúan siempre dentro de algún contenedor y son el aspecto visible del interfaz: botones, etiquetas, campos de texto, etc...
Los contenedores pueden a su vez contener a otros contenedores, por lo que nos encontramos contenedores de dos tipos:

- Contenedores de tipo Superior: **JApplet, JFrame, JDialog.**

- Contenedores de tipo Intermedio: **JPanel, JScrollPane, JSplitPane, JTabbedPane, JToolBar** y otros más especializados.



Esquema de la jerarquía de clases de AWT. Son las clases base de Swing.



El diagrama muestra como los componentes Swing (gris oscuro, y su nombre empieza siempre por J) extienden a los AWT. No se ha incluido la rama JApplet. La clase JApplet extiende a la clase Applet de AWT.

En la construcción de un GUI, normalmente seguiremos los siguientes pasos:

I. Crear un contenedor superior y obtener su contenedor intermedio.

II. Seleccionar un gestor de esquemas para el contenedor intermedio.

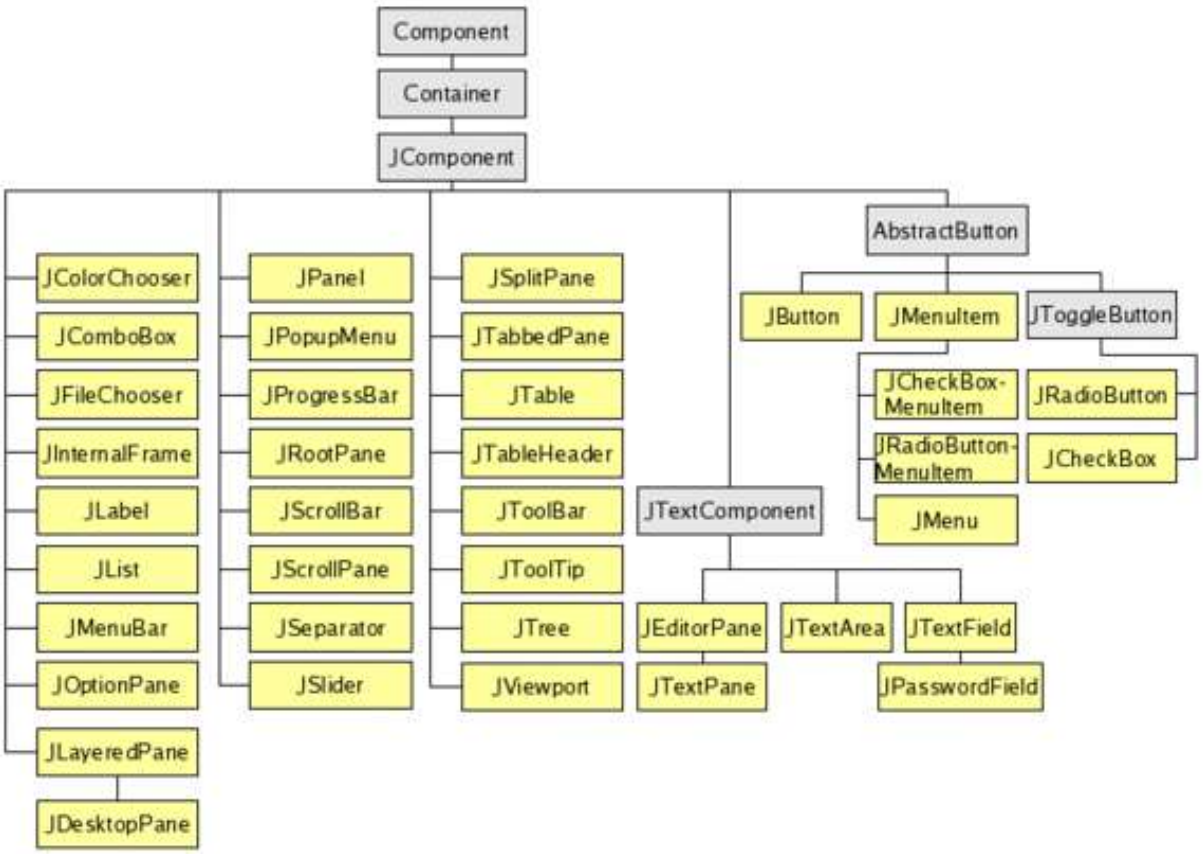
III. Crear los componentes adecuados.

IV. Agregarlos al contenedor intermedio.

V. Dimensionar el contenedor superior.

VI. Mostrar el contenedor superior.

La jerarquía completa de componentes Swing:



Los contenedores intermedios son: **JPanel, JScrollPane, JSplitPane, JTabbedPane** y **JToolBar**. El contenedor más utilizado es JPanel.

I. Crear un contenedor superior

Como de dijo anteriormente, hay tres clases contenedores superiores: **JFrame, JDialog, JApplet**.

- JFrame se usa para aplicaciones corrientes. Son las ventanas de nivel superior con bordes y título. Usa los métodos: setTitle( ), getTitle( ), setIconImage( ) entre otros.

- JApplet se usa exclusivamente para Applets. Un Applet es una aplicación Java incrustada en una página web.

- JDialog se usa para ventanas de diálogo.

Algunos métodos de instancia comunes a los contenedores de nivel superior:
void pack( ), Container getContentPane( ), void setContentPane(Container),
void setJMenuBar(Menu), ...
Tras crear el contenedor superior crearemos él o los **contenedores intermedios**.

• Clases Frame/JFrame
  - Una simple ventana que ofrece iconos para maximizar, minimizar y cerrarla. Se le puede añadir un título.
  - Único contenedor al que se le pueden añadir menús.
  - Podemos crearlo haciendo que nuestra clase extienda a JFrame.

```
import javax.swing.*;
public class Marco extends JFrame {
  public Marco( ) {
  super("Manejando un JFrame.");
  initialize( );
  }
  private void initialize( ) {
  this.setSize(300, 200);
 }
  public static void main(String[] args){
  Marco thisClass = new Marco();
  thisClass.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  thisClass.setVisible(true);
 }
}
```

• Clases Panel/JPanel

  - Contenedores genéricos para agrupar componentes.
  - Clase utilizada para meter contenedores dentro de otros contenedores (dentro de un panel, subpaneles).
  - **En Swing, dentro de la clase JFrame se añade automáticamente un Panel. En AWT no.**

II. Seleccionar un gestor de esquemas para cada contenedor intermedio

Determinan como encajan los componentes a medida que se van añadiendo dentro de los **contenedores intermedios**. Cada contenedor tiene un gestor propio (p.e. JPanel tiene un esquema BorderLayout por defecto).
Los gestores (son clases) existentes son:**FlowLayout, BorderLayout, GridLayout, GriBagLayout, CardLayout, BoxLayout, ...**
Sin embargo podríamos prescindir de utilizar cualquier gestor y colocar los elementos usando el método setPosition( ), aunque esto no es muy recomendable.

Para asignar un gestor de esquemas: **contenedor.setLayout(new FlowLayout( ))**

III. Crear componentes

Cada componente viene determinado por una clase. Hay que crear un objeto de esa clase:

```
JButton botonSi = new JButton("SI");
JButton botonNo = new JButton("NO");
JLabel l = new JLabel("Nombre");
...
```

IV. Agregar componentes al contenedor

Se hace a través del método add( ) de los contenedores:

```
JFrame f = new JFrame("Un ejemplo");  // Creamos contenedor superior
Container cpane = f.getContentPane( );     //  Creamos contenedor intermedio y la
                                           //  añadimos al superior
cpane.setLayout(new FlowLayout( ));    // Seleccionamos gestor de esquema
JButton bSi = new JButton("SI");           // Creamos los componentes
JButton bNo = new JButton("NO");
JLabel l = new JLabel("Nombre");
cpane.add(l);                              // Los añadimos al contenedor intermedio
cpane.add(bSi);
cpane.add(bNo);
```

El orden es importante, ya que se irán colocando en este orden, en función del gestor de esquemas elegido. A un contenedor intermedio también se le pueden agregar otros contenedores intermedios como si fuesen componentes.

V. Dimensionar el contenedor superior

El método a llamar es: **void setSize(int anchura, int altura);** Con este especificamos el tamaño del contenedor superior.

```
JFrame f = new JFrame("Un ejemplo");
....
f.setSize(int anchura, int altura);
```

Una alternativa a utilizar el método **setSize( )** es el método **pack( )**, que calcula el tamaño de la ventana teniendo en cuenta:

- El gestor de esquemas.
- El número y orden de los componentes añadidos.
- La dimensión de los componentes.

En este caso son útiles los métodos:

- void setPreferredSize(Dimension),
- void setMinimunSize(Dimension)
- void setMaximumSize(Dimension)

```
JFrame f = new JFrame("Un ejemplo");
....
f.pack( );
```

VI. Mostrar el contenedor superior

A continuación se debe mostrar el contenedor superior. Para hacerlo visible o invisible se utiliza el método: **setVisible(boolean);**

Este método es válido para mostrar u ocultar componentes y contenedores:

```
JFrame f = new JFrame(“Un ejemplo”);
....
f.setVisible(true);
```

Un ejemplo completo:

```
import java.awt.*;
import javax.swing.*;
class GUI01 {                              //En este caso no hacemos que extienda a JFrame
public static void main(String [ ] args) {
  JFrame f = new JFrame(“Un ejemplo·);
  Container cpane = f.getContentPane( );
  cpane.setLayout(new FlowLayout( ));
  JButton bSi = new JButton(“SI”);
  JButton bNo = new JButton(“NO”);
  JLabel l = new JLabel(“Nombre”);
  cpane.add(l);
  cpane.add(bSi);
  cpane.add(bNo);
  f.pack( );
  f.setVisible(true);
 }
}
```

En este ejemplo las funciones de maximizar y minimizar, cambiar tamaño y mover están operativas. Los botones Si y No ceden cuando se pulsan pero no realizan ninguna acción. Al cerrar la ventana no se cierra la aplicación. Además no tiene el aspecto de una ventana Windows.

```
import java.awt.*;
import javax.swing.*;
public class TestComponentes extends JFrame {

   JLabel jLabel1 = new JLabel( );
   JTextField jTextField1 = new JTextField( );
   JLabel jLabel2 = new JLabel( );
   JTextArea jTextArea1 = new JTextArea( );

   public TestComponentes( ){
   this.setTitle("Título de la ventana.");
   jLabel1.setText("Etiqueta 1");
   jTextField1.setColumns(25);
   jLabel2.setText("Etiqueta 2");
   jTextArea1.setColumns(50);
   jTextArea1.setRows(10);
   // El método getContentPane( ) retorna un contenedor intermedio.
   this.getContentPane( ).setLayout(new FlowLayout( ));
```

```
  this.getContentPane( ).add(jLabel1, null);
  this.getContentPane( ).add(jTextField1, null);
  this.getContentPane( ).add(jLabel2, null);
  this.getContentPane( ).add(jTextArea1, null);
  this.setSize(400, 300);
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 }

  public static void main(String[ ] args){
  Frame frame = new TestComponentes( );

  frame.setVisible(true);
 }
}
```

Controlar el aspecto de la aplicación

1. Iconos

En algunos constructores y métodos aparece un argumento Icon que representa un icono (Icon es una interface). Para cargar un icono desde un fichero:

Icon i = new ImageIcon("c:\\misIconos\\miLogo.gif")

O bien:

ImageIcon i = new ImageIcon("c:\\misIconos\\miLogo.gif")

2. Look and Feel

Existen librerías Look and Feel que nos permiten cambiar el aspecto de la aplicación:

```
public static void main(String[ ] args) {
  try {
        UIManager.setLookAndFeel("Look and feel válido");
       } catch (Exception e) {   }
       //Trabajar normalmente ...
}
```

Algunos posibles Look and Feel:

"javax.swing.plaf.metal.MetalLookAndFeel"
"com.sun.java.swing.plaf.windows.WindowsLookAndFeel"
"com.sun.java.swing.plaf.motif.MotifLookAndFeel"
"javax.swing.plaf.mac.MacLookAndFeel"
"javax.swing.plaf.nimbus.NimbusLookAndFeel"

Gestores de Esquemas

Entendemos por gestores de esquemas a un conjunto de clases que determinan cómo se distribuirán los componentes dentro del contenedor. La mayoría ya estaban definidos en java.awt:

- FlowLayout
- BorderLayout (JPanel **por defecto** dispone de un BorderLayout)
- GridLayout
- GridBagLayout
- CardLayout (Swing propone alternativa)
- BoxLayout (nueva en Swing: javax.swing)

FlowLayout

Los componentes fluyen **de izquierda a derecha y de arriba a abajo**. Su tamaño se ajusta al texto que presentan. Al cambiar el tamaño de la ventana, puede cambiar la disposición.

BorderLayout

Divide el contenedor en 5 partes: NORTH, SOUTH, EAST, WEST y CENTER.
Los componentes se ajustan hasta rellenar completamente cada parte. Si algún componente falta, se ajusta con el resto (menos el centro si hay cruzados).
Para añadir componentes al contenedor se utiliza una versión de add que indica la zona en la que se añade (Constantes definidas en la clase):

```
add(botonSi, BorderLayout.NORTH);
```

```
import javax.swing.*;
import java.lang.*;
import java.awt.*;
import java.awt.event.*;

public class progBorderLayout {

static JFrame ventana= new JFrame();
static JLabel l1 = new JLabel("norte");
static JLabel l2 = new JLabel("sur");
static JLabel l3 = new JLabel("este");
static JLabel l4 = new JLabel("oeste");
static JButton b1 = new JButton("CENTRO");

public static void main(String[ ] args)
{
  ventana.setTitle("BorderLayout");
  ventana.setLayout(new BorderLayout());
  ventana.add(l1,BorderLayout.NORTH);
  ventana.add(l2,BorderLayout.SOUTH);
  ventana.add(l3,BorderLayout.EAST);
  ventana.add(l4,BorderLayout.WEST);
```

```
 ventana.add(b1,BorderLayout.CENTER);
 ventana.setSize(500,500);
 ventana.setVisible(true);
 ventana.addWindowListener( new WindowAdapter( )
  {
      public void windowClosing(WindowEvent e){ System.exit(0); }} );
   };

}
```

GridLayout

Divide al componente en una rejilla (**grid**). En el constructor debemos indicar el número de filas y de columnas.
Los componentes se mantienen de **igual tamaño** dentro de cada celdilla. El orden a la hora de agregar determina la posición (**de izda a drcha y de arriba a abajo**).

```
cpane.setLayout(new GridLayout(2,3));   //2 filas, 3 columnas.
```

```
import javax.swing.*;
import java.lang.*;
import java.awt.*;
import java.awt.event.*;


public class progGridLayout {
static Frame ventana= new Frame();
static Label l1 = new Label("label1");
static Label l2 = new Label("label2");
static Label l3 = new Label("label3");
static Label l4 = new Label("label4");
static Button b1 = new Button("buton1");

public static void main(String[] args)
{
 ventana.setTitle("mi programa");
 ventana.setLayout(new GridLayout(0,3));
 ventana.add(l1);
 ventana.add(l2);
 ventana.add(l3);
 ventana.add(l4);
 ventana.add(b1);
 ventana.setSize(200,200);
 ventana.setVisible(true);
 ventana.addWindowListener( new WindowAdapter()
 {
  public void windowClosing(WindowEvent e){ System.exit(0); }} );
 };
}
```

BoxLayout

Coloca a los componentes a lo largo de un eje. Define dos constantes X_AXIS,Y_AXIS
En el constructor debemos indicar el contenedor y la orientación de los componentes:

BoxLayout(Container, int);

Los componentes **no tienen igual tamaño** (como en GridLayout). Existe la clase Box para facilitar la construcción (es un Container).
El orden a la hora de agregar determina la posición (de izda a drcha y de arriba a abajo):

cpane.setLayout(new BoxLayout(this,BoxLayout.X_AXIS));

```
import javax.swing.*;
import java.lang.*;
import java.awt.*;
import java.awt.event.*;


public class progBoxLayout {
public static void main(String [] args)
        {
                // Se crea la ventana con el BoxLayout
                JFrame v = new JFrame();
                v.getContentPane().setLayout(new
BoxLayout(v.getContentPane(),BoxLayout.Y_AXIS));

                // Se crea un botón centrado y se añade
                JButton boton = new JButton("B");
                boton.setAlignmentX(Component.CENTER_ALIGNMENT);
                v.getContentPane().add(boton);

                // Se crea una etiqueta centrada y se añade
                JLabel etiqueta = new JLabel("una etiqueta centrada");
                etiqueta.setAlignmentX(Component.LEFT_ALIGNMENT);
                v.getContentPane().add(etiqueta);

            JLabel otraEtiqueta = new JLabel("etiqueta ladeada");
                etiqueta.setAlignmentX(Component.RIGHT_ALIGNMENT);
                v.getContentPane().add(otraEtiqueta);
                // Visualizar la ventana
                v.setSize(400,300);
                v.setVisible(true);
                v.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        }
}
```

GridBagLayout

Este controlador de posicionamiento coloca los componentes en filas y columnas, pero permite especificar una serie de parámetros adicionales para ajustar mejor la posición y tamaño de los componentes dentro de las celdas. Y, al contrario que ocurría con el GridLayout, las filas y columnas **no tienen porque tener un tamaño uniforme**.

El controlador utiliza en número de componentes en la fila más larga, el número de filas totales y el tamaño de los componentes, para determinar el número y tamaño de las celdas que va a colocar en la tabla. La forma de visualizarse este conjunto de celdas, puede determinarse a través de una serie de características recogidas en un objeto de tipo **GridBagConstraints**. Estas características, o propiedades, son las que se van a describir a continuación. El objeto GridBagConstraints se inicializa a unos valores de defecto, cada uno de los cuales puede ser ajustado para alterar la forma en que se presenta los componentes dentro del layout.

```
import java.awt.*;
import java.awt.event.*;

public class AwtGBag extends Frame {
   Panel panel;

   public AwtGBag( ) {
      // Estos son los botones que se van a marear
      Button botAceptar,botCancelar;

      // Este es el panel que contiene a todos los componentes
      panel = new Panel();
      panel.setBackground( Color.white );
      add( panel );

      // Se crean los objetos del GridBag y se le asigna este
      // layout al panel
      GridBagConstraints gbc = new GridBagConstraints();
      GridBagLayout gridbag = new GridBagLayout();
      panel.setLayout( gridbag );

      // Se indica que los componentes pueden rellenar la zona
      // visible en cualquiera de las dos direcciones, vertical
      // u horizontal
      gbc.fill = GridBagConstraints.BOTH;
      // Se redimensionan las columnas y se mantiene su relación
      // de aspecto isgual en todo el proceso
      gbc.weightx = 1.0;

      // Se crea la etiqueta que va a servir de título al
      // panel
      Label labTitulo = new Label( "GridBag Layout" );
      labTitulo.setAlignment( Label.CENTER );
      // Se hace que el componente Label sea el único que se
```

```
// sitúe en la línea que lo contiene
gbc.gridwidth = GridBagConstraints.REMAINDER;
// Se pasan al layout tanto el componente Label, como
// el objeto GridBagConstraints que modifica su
// posicionamiento
gridbag.setConstraints( labTitulo,gbc );
// Finalmente se añade la etiqueta al panel. El objeto
// GridBagConstraints de este contenedor pone la etiqueta
// en una línea, la redimensiona para que ocupe toda la
// fila de la tabla, tanto horizontal como verticalmente,
// y hace que las columnas se redimensionen de la misma
// forma cuando la ventana cambie de tamaño
panel.add( labTitulo );

// Ahora se crea uno de los campos de texto, en este
// caso para recoger un supuesto nombre
TextField txtNombre = new TextField( "Nombre:",25 );
// Se hace que el campo de texto sea el siguiente objeto
// después del último que haya en la fila. Esto significa
// que todavía se puede añadir uno o más componentes a
// la fila, a continuación del campo de texto
gbc.gridwidth = GridBagConstraints.RELATIVE;
// Se pasan al layout tanto el campo de texto, como
// el objeto GridBagConstraints que modifica su
// posicionamiento
gridbag.setConstraints( txtNombre,gbc );
// Se añade el campo de texto al panel
panel.add( txtNombre );

// Se crea otro campo de texto, en este caso para recoger
// la direccion del propietario del nombre anterior
TextField txtDireccion = new TextField( "Direccion:",25 );
// Se hace que este campo de texto sea el último
// componente que se sitúe en la fila en que se
// encuentre
gbc.gridwidth = GridBagConstraints.REMAINDER;
// Se pasan al layout tanto el campo de texto, como
// el objeto GridBagConstraints que modifica su
// posicionamiento
gridbag.setConstraints( txtDireccion,gbc );
// Se añade el campo de texto al panel
panel.add( txtDireccion );

// Se crea un área de texto para introducir las cosas
// que quiera el que esté utilizando el programa
TextArea txtComent = new TextArea( 3,25 );
txtComent.setEditable( true );
txtComent.setText( "Comentarios:" );
// Se pasan al layout tanto el área de texto, como
// el objeto GridBagConstraints que modifica su
```

```
    // posicionamiento
    gridbag.setConstraints( txtComent,gbc );
    // Se añade el área de texto al panel
    panel.add( txtComent );
     // Estos son los dos botones de la parte inferior del
    // panel, sobre los que vamos a modificar las
    // propiedades del objeto GridBagConstraints que
    // controla su posicionamiento dentro del panel, para
    // ir mostrando el comportamiento del conjunto ante
    // esos cambios
    botAceptar = new Button( "Aceptar" );
    botCancelar = new Button( "Cancelar" );
     // Hacemos que el botón "Aceptar" no sea el último
    // de la fila y que no pueda expandirse en ninguna
    // dirección
    gbc.fill = GridBagConstraints.NONE;
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    // Se pasan al layout el botón y el objeto
    // GridBagConstraints
    gridbag.setConstraints( botAceptar,gbc );
    // Se añade el botón al panel
    panel.add( botAceptar );
    // Se hace que el botón "Cancelar" sea el último de
    // la fila en que se encuentre
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    // Se hace que su altura no se reescale
    gbc.gridheight = 1;
    // Se pasan al layout el botón y el objeto
    // GridBagConstraints
    gridbag.setConstraints( botCancelar,gbc );
    // Se añade el botón al panel
    panel.add( botCancelar );
     // Se añade el receptor de eventos de la ventana
    // para acabar la ejecución
    addWindowListener( new WindowAdapter() {
       public void windowClosing( WindowEvent evt ) {
          System.exit( 0 );
          }
       } );
    }
public static void main( String args[] ) {
    AwtGBag miFrame = new AwtGBag();

    // Fijamos el título de la ventana y la hacemos
    // visible, con los componentes en su interior
    miFrame.setTitle( "Tutorial de Java, AWT" );
    miFrame.pack();
    miFrame.setVisible( true );
    }
}
```

Otro ejemplo sin comentar:

```
import java.awt.*;
import javax.swing.*;

 public class AwtGBag2  extends JFrame{

    public static void main(String[ ] args) {

        AwtGBag2 gbl = new AwtGBag2();
        Container c = gbl.getContentPane();
        GridBagLayout gridbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        c.setLayout( gridbag );
        gbc.fill = GridBagConstraints.BOTH;
        gbc.weightx = 1.0;
        Button boton0 = new Button( "Botón 0" );
        gridbag.setConstraints( boton0,gbc );
        c.add( boton0 );
        Button boton1 = new Button( "Botón 1" );
        gridbag.setConstraints( boton1,gbc );
        c.add( boton1 );
        Button boton2 = new Button( "Botón 2" );
        gridbag.setConstraints( boton2,gbc );
        c.add( boton2 );
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        Button boton3 = new Button( "Botón 3" );
        gridbag.setConstraints( boton3,gbc );
        c.add( boton3 );
        gbc.weightx = 0.0;
        Button boton4 = new Button( "Botón 4" );
        gridbag.setConstraints( boton4,gbc );
        c.add( boton4 );
        gbc.gridwidth = GridBagConstraints.RELATIVE;
        Button boton5 = new Button( "Botón 5" );
        gridbag.setConstraints( boton5,gbc );
        c.add( boton5 );
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        Button boton6 = new Button( "Botón 6" );
        gridbag.setConstraints( boton6,gbc );
        c.add( boton6 );
        gbc.gridwidth = 1;
        gbc.gridheight = 2;
        gbc.weighty = 1.0;
        Button boton7 = new Button( "Botón 7" );
        gridbag.setConstraints( boton7,gbc );
        c.add( boton7 );
        gbc.weighty = 0.0;
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbc.gridheight = 1;
```

```
        Button boton8 = new Button( "Botón 8" );
        gridbag.setConstraints( boton8,gbc );
        c.add( boton8 );
        Button boton9 = new Button( "Botón 9" );
        gridbag.setConstraints( boton9,gbc );
        c.add( boton9 );
        gbl.setSize(400,400);
        gbl.setVisible(true);
        }

    }
```

GUI complejos

Podemos utilizar un contenedor intermedio en lugar de un componente para agregarlo a otro contenedor intermedio. Este nuevo contenedor intermedio podrá incorporar sus propios componentes y tener su propio gestor de esquemas:

```
JFrame f = new JFrame(“Un ejemplo·);
f.getContentPane(new BorderLayout());
JPanel p = new JPanel();
JButton bp1 = new JButton(“Panel1”);
JButton bp2 = new JButton(“Panel2”);
p.setLayout(new GridLayout(2,1));
p.add(bp1);
p.add(bp2);
f.getContentPane().add(p,BorderLayout.WEST);
...
```

Ejemplo de calculadora:

```
import javax.swing.*;
import java.lang.*;
import java.awt.*;
import java.awt.event.*;
public class progGBL{

   JFrame frame;
   JPanel panelSuperior, panelInferior;
   JButton bt1, bt2, bt3, bt4, bt5, bt6, bt7, bt8, bt9, bt0, btRT, btCE, btCL, btMas,
btMenos, btMul, btDiv, btIgual, btMN, btPunto;
   JTextField pantalla;

   public progGBL(){
      construyePanelSuperior();
      construyePanelInferior();
      construyeVentana();}

   void construyePanelSuperior(){
      panelSuperior = new JPanel ();
```

```
    panelSuperior.setLayout(new FlowLayout());
    pantalla = new JTextField(20);
    panelSuperior.add(pantalla);}

  void construyePanelInferior(){
    panelInferior= new JPanel();
    panelInferior.setLayout(new GridLayout(5,4,8,8));
    bt1=new JButton("1");
    bt2=new JButton("2");
    bt3=new JButton("3");
    bt4=new JButton("4");
    bt5=new JButton("5");
    bt6=new JButton("6");
     bt7=new JButton("7");
    bt8=new JButton("8");
    bt9=new JButton("9");
    bt0=new JButton("0");
    btRT=new JButton("Rtc");
    btCE=new JButton("CE");
    btCL=new JButton("CL");
    btMas=new JButton("+");
    btMenos=new JButton("-");
    btMul=new JButton("x");
    btDiv=new JButton("/");
    btIgual=new JButton("=");
    btMN=new JButton("+/-");
    btPunto=new JButton(".");
    panelInferior.add(btRT);
    panelInferior.add(btCE);
    panelInferior.add(btCL);
    panelInferior.add(btMN);
    panelInferior.add(bt7);
    panelInferior.add(bt8);
    panelInferior.add(bt9);
    panelInferior.add(btDiv);
    panelInferior.add(bt4);
    panelInferior.add(bt5);
    panelInferior.add(bt6);
    panelInferior.add(btMul);
    panelInferior.add(bt1);
    panelInferior.add(bt2);
    panelInferior.add(bt3);
    panelInferior.add(btMenos);
    panelInferior.add(bt0);
    panelInferior.add(btPunto);
    panelInferior.add(btIgual);
    panelInferior.add(btMas);}

  void construyeVentana(){
    frame =new JFrame("Calculadora ");
```

```
        frame.setLayout(new BoxLayout(frame.getContentPane(),BoxLayout.Y_AXIS));
        frame.add(panelSuperior);
        frame.add(panelInferior);
        frame.pack();
        frame.setVisible(true);
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);}

    public static void main(String [] inforux){
        new progGBL();}
}
```

Contenedores intermedios

JScrollPane

Permite hacer scroll a un componente (u otro contenedor intermedio).

Constructores:
JScrollPane JScrollPane(JComponent);
JScrollPane JScrollPane(JComponent,int,int);

Constantes para control del scroll:
VERTICAL_SCROLLBAR_AS_NEEDED
HORIZONTAL_SCROLLBAR_AS_NEEDED
VERTICAL_SCROLLBAR_ALWAYS
HORIZONTAL_SCROLLBAR_ALWAYS
VERTICAL_SCROLLBAR_NEVER
HORIZONTAL_SCROLLBAR_NEVER

```
import javax.swing.*;
public class PruebaJScrollPane {
    public PruebaJScrollPane( ) {
        JFrame ventana = new JFrame("Imagen");
        JScrollPane scroll = new JScrollPane();
        JLabel etiqueta = new JLabel();
        Icon imagen = new ImageIcon ("C:/foto.jpg");
        etiqueta.setIcon (imagen);
        // Se mete el scroll en la ventana
        ventana.getContentPane().add(scroll);
        // Se mete el label en el scroll
        scroll.setViewportView(etiqueta);
        ventana.setSize(200,200);
        ventana.setVisible(true);
        ventana.setDefaultCloseOperation(ventana.EXIT_ON_CLOSE);
    }
    public static void main(String[ ] args) {
        new PruebaJScrollPane();
    }

}
```

JSplitPane

Divide una ventana en dos, de manera vertical u horizontal, con movimiento visible o no.

Constructores (entre otros):
SplitPane(int, JComponent, JComponent)
SplitPane(int, boolean, JComponent, JComponent)

Constantes:
HORIZONTAL_SPLIT VERTICAL_SPLIT

Métodos de instancia:
setOneTouchExpandable(boolean);
setDividerLocation(int);

```
import java.lang.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class progSplitP {
static JFrame ventana= new JFrame();
// abajo se creando con orientacion vertical u horizontal
static JSplitPane panel1 = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
static JLabel jl1 = new JLabel("en split1 label1");
static JLabel jl2 = new JLabel("en split2 label2");

 public static void main(String[] args)
 {
  ventana.setTitle("mi programa");
  ventana.setDefaultCloseOperation(ventana.EXIT_ON_CLOSE);
  //cargando splitpanel panel1 con sus dos componentes
  panel1.add(jl1);
  panel1.add(jl2);
  // cargando la ventana con splitpanel
  ventana.getContentPane().add(panel1,BorderLayout.CENTER);
  ventana.setSize(200,200);
  ventana.setVisible(true);
 }

}
```

JTabbedPane

Permite simular carpetas sobre la ventana.

Constructores (entre otros):
JTabbedPane()
JTabbedPane(int)

Constantes:
TOP, BOTTOM, LEFT, RIGHT.

Métodos de instancia:
addTab(String, Component)
addTab(String, Icon, Component)
addTab(String, Icon, Component, String)
setSelectedIndex(int);

```
import java.awt.*;
import javax.swing.*;

public class progJTabbedP extends JFrame  {

   // configurar GUI
   public progJTabbedP()
   {
      super( "Demo de JTabbedPane " );

      // crear objeto JTabbedPane
      JTabbedPane panelConFichas = new JTabbedPane();

      // establecer pane11 y agregarlo al objeto JTabbedPane
      JLabel etiqueta1 = new JLabel( "panel uno", SwingConstants.CENTER );
      JPanel panel1 = new JPanel();
      panel1.add( etiqueta1 );
      panelConFichas.addTab( "Ficha uno", null, panel1, "Primer panel" );

      // establecer panel2 y agregarlo al objeto JTabbedPane
      JLabel etiqueta2 = new JLabel( "panel dos", SwingConstants.CENTER );
      JPanel panel2 = new JPanel();
      panel2.setBackground( Color.YELLOW );
      panel2.add( etiqueta2 );
      panelConFichas.addTab( "Ficha dos", null, panel2, "Segundo panel" );

      // establecer panel3 y agregarlo al objeto JTabbedPane
      JLabel etiqueta3 = new JLabel( "panel tres" );
      JPanel panel3 = new JPanel();
      panel3.setLayout( new BorderLayout() );
      panel3.add( new JButton( "Norte" ), BorderLayout.NORTH );
      panel3.add( new JButton( "Oeste" ), BorderLayout.WEST );
      panel3.add( new JButton( "Este" ), BorderLayout.EAST );
```

```
      panel3.add( new JButton( "Sur" ), BorderLayout.SOUTH );
      panel3.add( etiqueta3, BorderLayout.CENTER );
      panelConFichas.addTab( "Ficha tres", null, panel3, "Tercer panel" );

      // agregar objeto JTabbedPane al contenedor
      getContentPane().add( panelConFichas );

      setSize( 250, 200 );
      setVisible( true );

   }

   public static void main( String args[] )
   {
      progJTabbedP demoPanelConFichas = new progJTabbedP();
      demoPanelConFichas.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
   }

}
```

JToolBar

Crea un barra de botones. Debe incluirse en un contenedor con BorderLayout. Usualmente contiene botones con iconos.

Constructor

JToolBar()
JToolBar(int) // HORIZONTAL, VERTICAL

Métodos de instancia

addSeparator()
setFloatable(boolean) // float por defecto

```
import javax.swing.*;

public class JToolB extends JFrame {

  JToolBar TBarra=new JToolBar();
  JButton BNuevo=new JButton("Nuevo");
  JButton BAbrir=new JButton("Abrir");
  JButton BCopiar=new JButton("Copiar");
  JButton BCortar=new JButton("Cortar");
  JButton BPegar=new JButton("Pegar");
  JButton BGuardar=new JButton("Guardar");

  public JToolB() {

  //ToolBar
```

```
 TBarra.add(BNuevo);
 TBarra.add(BAbrir);
 TBarra.add(BGuardar);
 TBarra.addSeparator();
 TBarra.add(BCopiar);
 TBarra.add(BCortar);
 TBarra.add(BPegar);
 BGuardar.setToolTipText ("Guardar");
 BNuevo.setToolTipText ("Nuevo");
 BAbrir.setToolTipText ("Abrir");
 BCopiar.setToolTipText ("Copiar");
 BCortar.setToolTipText ("BCortar");
 BPegar.setToolTipText ("Pegar");

 add(TBarra,"North");

 TBarra.setFloatable(false);
 setTitle("Ejemplos JPopUpMenu");
 setSize(800,600);
 setVisible(true);
 }

public static void main (String []args){
 new JToolB();
 }
}
```

Componentes

Todos los componentes Swing tienen los siguientes **métodos heredados** de JComponent:

- Color getBackground()
- void setBackground(Color)
- Graphics getGraphics()
- String getName()
- Toolkit getToolkit()
- void setEnable(boolean)
- void setVisible(boolean)
- void paint(Graphics g)
- void repaint()
- void setBorder()

Además podemos obtrener distintos Bordes:

En javax.swing.borders existen una serie de clases que permiten dar un borde a un componente. Hay nueve clases:

- AbstractBorder
- BevelBorder

- CompountBorder
- EmptyBorder
- EtchedBorder
- LineBorder
- MatteBorder
- SoftBevelBorder
- TitleBorder

Para cambiar el borde de un componente:

public void setBorder(Border);

Por ejemplo:

JButton b = new JButton(“Aceptar”);
b.setBorder(new TitledBorder(“Boton”))

La clase **javax.swing.BorderFactory** tiene métodos de clase para crear bordes.

JButton b = new JButton(“Aceptar”);
b.setBorder(BorderFactory.createTitleBorder(“Boton”))

JButton

Crea botones de pulsación.

Constructores:
JButton( )
JButton(String)
JButton(String,Icon)
JButton(Icon)

Métodos:
String getText()
void setText(String)
...

JLabel

Es una etiqueta con una línea de texto.

Constructores:
JLabel([String,] [Icon,] [int])

Constantes:
LEFT, RIGHT, CENTER.

Métodos de instancia:
String getText( )

void setText(String)
...

JCheckBox

Son marcadores.

Constructores:
JCheckBox([String,] [Icon,] [boolean])

Métodos de instancia:
String getText()
void setText(String)
boolean isSelected()
void setSelected(boolean)
...

```
import java.lang.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class progcheckbox {
static JFrame ventana= new JFrame();
static JPanel p1= new JPanel();
static JPanel p2= new JPanel();
static JTextField jt1=new JTextField(15);
static JButton jb1= new JButton("OK");
static Checkbox cb1=new Checkbox("h",true);
static Checkbox cb2=new Checkbox("m");

public static void main(String[] args)
{
  ventana.setTitle("mi programa");
  ventana.setDefaultCloseOperation(ventana.EXIT_ON_CLOSE);
  ventana.getContentPane().setLayout(new GridLayout(2,0));
  p1.setLayout(new GridLayout(2, 1));
  p1.add(cb1); p1.add(cb2);
//cargando segundo panel con jbutton y jtextfield
  p2.add(jb1); p2.add(jt1);
  ventana.getContentPane().add(p1); ventana.getContentPane().add(p2);
  ventana.pack(); ventana.setVisible(true);
  jb1.addMouseListener( new MouseAdapter()
   { public void mousePressed(MouseEvent e){
     //programando checkbox
     if(cb1.getState()== true ) jt1.setText("HOMBRE");
     if(cb2.getState()== true ) jt1.setText("MUJER"); }} );
     };
}
```

JRadioButtons y ButtonGroup

Botones circulares. Se agrupan de manera que sólo uno esté pulsado. Para agruparlos, se crea una instancia de ButtonGroup y se añaden con add(AbstractButton).

Constructores:
JRadioButtons([String,] [Icon,] [boolean])

Métodos de instancia:
Igual que JCheckBox.

Ejemplo con botones:

```
import java.awt.*;
import javax.swing.*;
class JRbut {
  public static void main(String [] args) {
  JFrame f = new JFrame("Ejemplo de Botones");
  JButton bNorte = new JButton("Norte");
  JLabel lSur = new JLabel("Este es el Sur",
  JLabel.CENTER);
  JCheckBox cEste = new JCheckBox("Este",true);
  JButton bCentro= new JButton("Centro");
  JRadioButton cp1 = new JRadioButton("RB1");
  JRadioButton cp2 = new JRadioButton("RB2",true);
  ButtonGroup gcb = new ButtonGroup();
  gcb.add(cp1);
  gcb.add(cp2);
  JPanel prb = new JPanel();
  prb.setLayout(new GridLayout(2,1));
  prb.add(cp1);
  prb.add(cp2);
  Container contP = f.getContentPane();
  contP.add(bNorte,BorderLayout.NORTH);
  contP.add(lSur,BorderLayout.SOUTH);
  contP.add(cEste,BorderLayout.EAST);
  contP.add(prb,BorderLayout.WEST);
  contP.add(bCentro,BorderLayout.CENTER);
  f.pack();
  f.setVisible(true);
  }
}
```

JTextField

Permite editar texto en una línea.

Constructores
JTextField ([String,] [int])

Métodos
String getText( )
String getText(int,int) // offset y len
void setEditable(boolean)
boolean isEditable( )
...

JPasswordField

Subclase de JTextField que enmascara el eco (con * u otro símbolo).

Método de instancia:
char[ ] getPassword( )

JTextArea

Permite editar texto en un área.

Constructores:
JTextArea ([String,] [int,int])

Métodos:
void append(String)
void insert(String,int)
igual que JTextField
...

Jlist

Muestra una lista de elementos para su selección.

Constructores
JList( ) JList(Object [ ])
JList(Vector) JList(ListModel)

Métodos de instancia
int getSelectedIndex( ) // -1 si no hay
int [ ] getSelectedIndices( )
Object getSelectedValue( )
Object [ ] getSelectedValues( )
boolean isSelectedIndex(int)
boolean isSelectionEmpty([ ])
void setListData(Object)
void setListData(Vector)
void setSelectionMode(int)
get selectionMode( ) ...

Constantes
ListSelectionModel.SINGLE_SELECTION
ListSelectionModel.SINGLE_INTERVAL_SELECTION
ListSelectionModel.MULTIPLE_INTERVAL_SELECTION

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class JLst extends JFrame {
  public static void main(String[] args) {
    new JLst();
  }

  private JList sampleJList;
  private JTextField valueField;

  public JLst() {
    super("Creating a Simple JList");


    Container content = getContentPane();

    // Create the JList, set the number of visible rows, add a
    // listener, and put it in a JScrollPane.
    String[] entries = { "Entrada 1", "Entrada 2", "Entrada 3",
                         "Entrada 4", "Entrada 5", "Entrada 6" };
    sampleJList = new JList(entries);
    sampleJList.setVisibleRowCount(4);
    Font displayFont = new Font("Serif", Font.BOLD, 18);
    sampleJList.setFont(displayFont);
    JScrollPane listPane = new JScrollPane(sampleJList);

    JPanel listPanel = new JPanel();
    listPanel.setBackground(Color.white);
    Border listPanelBorder =
      BorderFactory.createTitledBorder("Muestra JList");
    listPanel.setBorder(listPanelBorder);
    listPanel.add(listPane);
    content.add(listPanel, BorderLayout.CENTER);
    JLabel valueLabel = new JLabel("Última selección:");
    valueLabel.setFont(displayFont);
    valueField = new JTextField("Ninguna", 7);
    valueField.setFont(displayFont);
    JPanel valuePanel = new JPanel();
    valuePanel.setBackground(Color.white);
    Border valuePanelBorder =
      BorderFactory.createTitledBorder("JList Selección");
```

```
    valuePanel.setBorder(valuePanelBorder);
    valuePanel.add(valueLabel);
    valuePanel.add(valueField);
    content.add(valuePanel, BorderLayout.SOUTH);
    pack();
    setVisible(true);
  }
}
```

JComboBox

Permite la selección de un item de entre varios. No está desplegado como Jlist.

Constructores:
JComboBox() JComboBox(Object [])
JComboBox(Vector) JComboBox(ListModel)

Métodos de instancia:
int getSelectedIndex()
Object getSelectedItem()
void setSelectedIndex(int)
boolean isEditable()
void setEditable(boolean)

```
import java.lang.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JComBx {
static JFrame ventana= new JFrame();
static JPanel p1= new JPanel(); static JPanel p2= new JPanel();
static String[] lista={"municipios","Madrid","Villalba","Getafe","Aranjuez"};
static JComboBox municipios = new JComboBox(lista );
static JTextField jt1=new JTextField(15);
static JButton jb1= new JButton("OK");

public static void main(String[] args)
{
  ventana.setTitle("mi programa");
  ventana.setDefaultCloseOperation(ventana.EXIT_ON_CLOSE);
  ventana.getContentPane().setLayout(new GridLayout(2,0));
  //cargando panel1 con combobox y definiendo titulo
  p1.setLayout(new GridLayout(1,0));
  //observar que index cero es el titulo (aunque es un elemento mas)
  municipios.setSelectedIndex(0); p1.add(municipios);
  //cargando segundo panel con jbutton y jtextfield
  p2.add(jb1); p2.add(jt1);
  ventana.getContentPane().add(p1); ventana.getContentPane().add(p2);
  ventana.pack();
```

```
 ventana.setVisible(true);
 jb1.addMouseListener( new MouseAdapter()
 { public void mousePressed(MouseEvent e){
  // la propiedad getselecteditem() regresa un objeto
  jt1.setText(String.valueOf( municipios.getSelectedItem() ) );
  }} );
  };
 }
```

JDialog

Es un elemento de visualización al igual que Frame. Se suele crear y no visualizar hasta que sea necesario. Para que se vea: **setVisible(true)**, para ocultarla **setVisible(false)**. Para eliminarla: **dispose( ).**

Constructor:
JDialog(Frame, String, boolean)

Donde Frame es la ventana padre, String el título y boolean indica si es modal o no.

new JDialog(f,"Ventana modal",true);

```
import java.lang.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JDlg {



 public static void main(String[] args)
 {
  JFrame credi=new JFrame();
  JDialog  dialog = new JDialog(credi, "", true);
  dialog.setTitle("Creditos");
  JTextArea label = new JTextArea("Este Programa es un ejemplo :\n" +
                                                  " del uso de JDialog\n");
  Container contentPane = dialog.getContentPane();
  contentPane.add(label, BorderLayout.CENTER);
  dialog.setSize(new Dimension(300, 150));
  dialog.setVisible(true);
 }
 }
```

JOptionPane

Clase que contiene métodos de clase para crear distintas ventanas de mensajes (modales).

Métodos de clase:
showConfirmDialog(...) Realiza una pregunta de confirmación como Si, No Cancelar
showInputDialog(...) Espera una entrada
showMessageDialog(...) Informa de algo que ha ocurrido
showOptionDialog(...) Unifica las tres anteriores.

Constructor:
showConfirmDialog(Component padre, Object mensaje, String title, int optionType, int messageType, Icon icon)

Hay más constructores para este tipo de ventanas. Ejemplo:

JOptionPane.showConfirmDialog(null,”Esta seguro",”Ventana de Seguridad",JOptionPane.YES_NO_OPTION);

Argumentos de los métodos showXXXDialog(...):
Component padre puede ser null.
Object mensaje usualmente un String.
String título de la ventana.
int tipoOpcion: ERROR_MESSAGE INFORMATION_MESSAGE
WARNING_MESSAGE QUESTION_MESSAGE PLAIN_MESSAGE
int tipoMensaje:DEFAULT_OPTION YES_NO_OPTION
YES_NO_CANCEL_OPTION OK_CANCEL_OPTION
Icon icono: Hay uno por defecto.

```
import javax.swing.JOptionPane;

public class JOptionPaneTest1 {
   public static void main(String[] args) {
      String ans;
      ans = JOptionPane.showInputDialog(null, "Introduzca Millas/Hora:");
      double mph = Double.parseDouble(ans);
      double kph = 1.621 * mph;
      JOptionPane.showMessageDialog(null, "Kilómetros/Hora: = " + kph);

      System.exit(0);
   }
}
```

JFileChooser

Es un elemento de visualización al igual que permite la selección de un fichero.
Manipula nombres de ficheros.

Constructores .
JFileDialog()
JFileDialog(String) // path
JFileDialog(File)

Métodos de instancia
int showOpenDialog(Component)
int showCloseDialog(Component)
File getSelectedFile( )
File [ ] getSelectedFiles( )

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JFileCh extends JPanel implements ActionListener {
  JButton go;

  JFileChooser chooser;
  String choosertitle;

 public JFileCh() {
  go = new JButton("Mostrar directorio:");
  go.addActionListener(this);
  add(go);
  }

 public void actionPerformed(ActionEvent e) {
  int result;

  chooser = new JFileChooser();
  chooser.setCurrentDirectory(new java.io.File("."));
  chooser.setDialogTitle(choosertitle);
  chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
  //
  // disable the "All files" option.
  //
  chooser.setAcceptAllFileFilterUsed(false);
  //
  if (chooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
   System.out.println("getCurrentDirectory(): "
     +  chooser.getCurrentDirectory());
   System.out.println("getSelectedFile() : "
     +  chooser.getSelectedFile());
   }
  else {
   System.out.println("No Selection ");
   }
  }
```

```
  public Dimension getPreferredSize(){
    return new Dimension(200, 200);
    }

  public static void main(String s[]) {
    JFrame frame = new JFrame("");
    JFileCh panel = new JFileCh();
    frame.addWindowListener(
      new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          System.exit(0);
          }
        }
      );
    frame.getContentPane().add(panel,"Center");
    frame.setSize(panel.getPreferredSize());
    frame.setVisible(true);
    }
}
```

Menús

Se pueden añadir a los contenedores superiores. Para añadir un menú:

void setJMenuBar(JMenuBar)

Tres elementos básicos:

- Barra de Menú (JMenuBar)
- Entrada de Menú (JMenu)
- Item de entrada (JMenuItem y ....)

El menú de ayuda se añade a un **JMenuBar** (aún no está implementado)

void setHelpMenu(JMenu)

Un item puede ser a su vez un menú. Para añadir a un JMenuBar una entrada:

void add(JMenu)

Para añadir a un **JMenu**:

void add(JMenuItem)
void add(JMenuItem, MenuShortcut)
void addSeparator()

Para manejar los items y entradas:

void setEnabled(boolean)

boolean isEnabled()

Un CheckBoxMenuItem se puede seleccionar:

boolean getState()
void setState(boolean)

JPopupMenu

Crea menús aislados (en cualquier ventana).Contiene elementos de menú:

add(MenuItem)

Debe activarse en un componente, dada una posición de visualización:

show(JComponent, int x, int y)

```
import java.awt.Component;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.JCheckBoxMenuItem;
import javax.swing.JFrame;
import javax.swing.JMenuItem;
import javax.swing.JPanel;
import javax.swing.JPopupMenu;
import javax.swing.JSeparator;
import javax.swing.JSlider;
import javax.swing.MenuElement;
import javax.swing.MenuSelectionManager;
import javax.swing.border.CompoundBorder;
import javax.swing.border.EmptyBorder;
import javax.swing.border.TitledBorder;
import javax.swing.event.PopupMenuEvent;
import javax.swing.event.PopupMenuListener;

public class PMenu extends JPanel {

  public JPopupMenu popup;
  SliderMenuItem slider;
  int theValue = 0;
  public PMenu() {

    popup = new JPopupMenu();
    slider = new SliderMenuItem();
```

```
  popup.add(slider);
  popup.add(new JSeparator());

  JMenuItem ticks = new JCheckBoxMenuItem("Slider Tick Marks");
  ticks.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
      slider.setPaintTicks(!slider.getPaintTicks());
    }
  });
  JMenuItem labels = new JCheckBoxMenuItem("Slider Labels");
  labels.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
      slider.setPaintLabels(!slider.getPaintLabels());
    }
  });
  popup.add(ticks);
  popup.add(labels);
  popup.addPopupMenuListener(new PopupPrintListener());

  addMouseListener(new MousePopupListener());
}

// Inner class to check whether mouse events are the popup trigger
class MousePopupListener extends MouseAdapter {
  public void mousePressed(MouseEvent e) {
    checkPopup(e);
  }

  public void mouseClicked(MouseEvent e) {
    checkPopup(e);
  }

  public void mouseReleased(MouseEvent e) {
    checkPopup(e);
  }

  private void checkPopup(MouseEvent e) {
    if (e.isPopupTrigger()) {
      popup.show(PMenu.this, e.getX(), e.getY());
    }
  }
}

// Inner class to print information in response to popup events
class PopupPrintListener implements PopupMenuListener {
  public void popupMenuWillBecomeVisible(PopupMenuEvent e) {
  }

  public void popupMenuWillBecomeInvisible(PopupMenuEvent e) {
    theValue = slider.getValue();
```

```
      System.out.println("El valor es ahora " + theValue);
    }

    public void popupMenuCanceled(PopupMenuEvent e) {
      System.out.println("Popup menu está oculto!");
    }
  }

  public static void main(String s[]) {
    JFrame frame = new JFrame("Menu Ejemplo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setContentPane(new PMenu());
    frame.setSize(300, 300);
    frame.setVisible(true);
  }

  // Inner class that defines our special slider menu item
  class SliderMenuItem extends JSlider implements MenuElement {

    public SliderMenuItem() {
      setBorder(new CompoundBorder(new TitledBorder("Control"),
          new EmptyBorder(10, 10, 10, 10)));

      setMajorTickSpacing(20);
      setMinorTickSpacing(10);
    }

    public void processMouseEvent(MouseEvent e, MenuElement path[],
        MenuSelectionManager manager) {
    }

    public void processKeyEvent(KeyEvent e, MenuElement path[],
        MenuSelectionManager manager) {
    }

    public void menuSelectionChanged(boolean isIncluded) {
    }

    public MenuElement[] getSubElements() {
      return new MenuElement[0];
    }

    public Component getComponent() {
      return this;
    }
  }
}
```

Gestión de Eventos

En Java y otros lenguajes actuales existen modelos de eventos definidos. El programa es avisado automáticamente de todas aquellas acciones (eventos) **en los que esté interesado**.

Al diseñar un interfaz gráfico debemos de tener en cuenta que a consecuencia de las acciones del usuario se generarán distintos eventos. Se deben de programar métodos para responder a estos eventos provocados por el usuario.
Un evento es generado por una acción del usuario y está ligado a un componente del GUI. Ejemplos: pulsar una tecla, mover el ratón, cambiar el tamaño de una ventana, cerrarla, minimizarla, pulsar un botón, perder u obtener el foco de un componente, ...

Modelo de eventos en Java

El modelo de eventos de Java distingue dos elementos:

- Fuentes de Eventos: Elementos sobre los que se producen los eventos.
  Ejemplos: un botón, una lista, un panel, ...

- Escuchadores de Eventos: Elementos que reciben las notificaciones de los eventos.

¿Quién puede ser una fuente de eventos?

Cualquier componente de AWT o SWING sobre el que se puedan producir eventos (prácticamente todos).

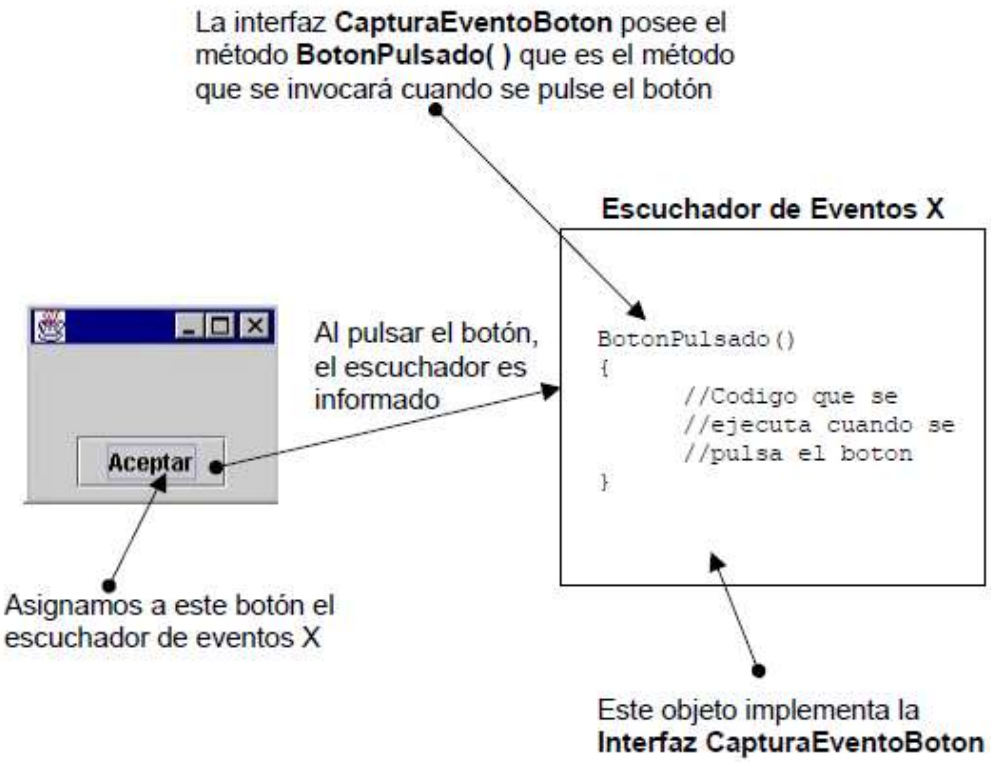
¿Quién puede ser un escuchador de eventos?

Cualquier objeto (perteneciente a una clase) **que implemente alguno de los interfaces definidos en Java para la notificación de los eventos**.

Funcionamiento:

Toda "**Fuente de Eventos**" debe tener asignado un "**Escuchador de Eventos**" que reciba las notificaciones de sus eventos. Cuando se produce un evento sobre la fuente de eventos, su escuchador es informado. Para ello, se invoca el método que el escuchador tenga definido para la notificación de ese tipo de evento. El escuchador, dentro de ese método, tendrá el código necesario para tratar ese evento. ¿Cómo se asigna un escuchador a una fuente de eventos?

<FuenteEvento>.add<EventoListener>(<Escuchador>)



Lista de **algunas** Interfaces y sus métodos:

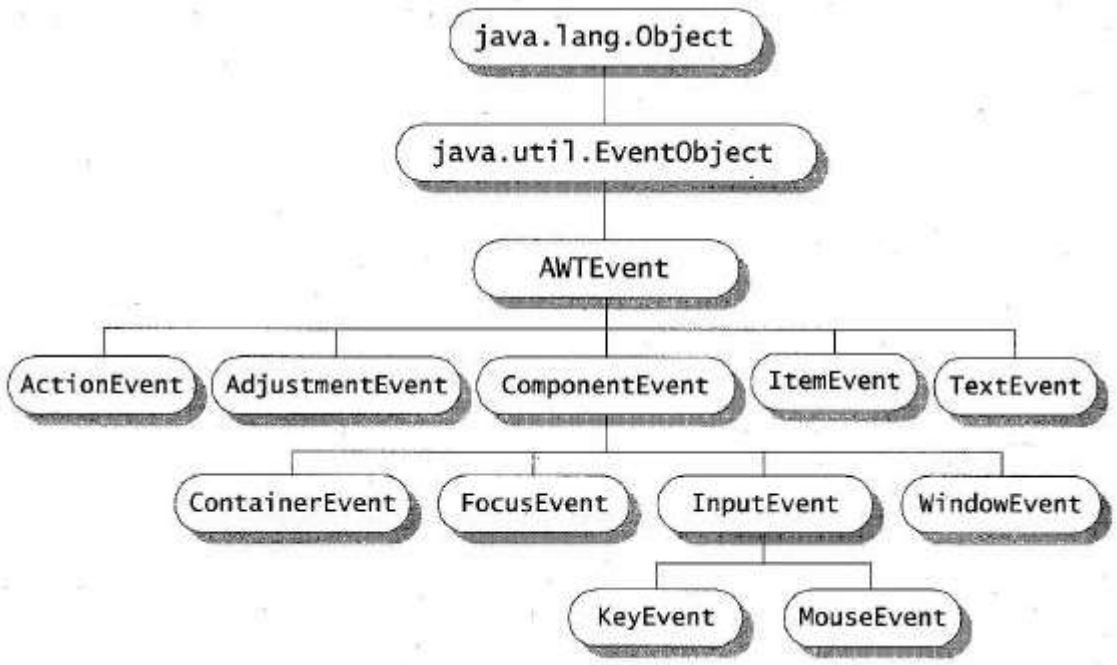
INTERFAZ	MÉTODOS
ActionListener	actionPerformed(ActionEvent)
FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
TextListener	textValueChanged(TextEvent)
KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener	mouseClicked(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent)
MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDeactivated(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)

Interfaces para la notificación de eventos

Son interfaces que contienen métodos que serán invocados cuando se produzca un determinado tipo de evento. Cada interfaz está especializado en capturar un tipo de eventos determinado.Todos estos métodos reciben como parámetro un objeto"Evento" que posee información sobre el evento ocurrido.

Tipos de eventos

Los objetos de tipo "Evento" están organizados en jerarquías de clases de eventos, de la siguiente forma:



La clase padre de todos los eventos es **EventObject** y la de los eventos de naturaleza gráfica es **AWTEvent**. Cuando ocurre un evento, los objetos de tipo "Evento" son pasados como parámetro en los métodos que posee el escuchador(listener).
Cada objeto de tipo "Evento" posee información sobre el evento concreto que se ha producido.Cada tipo de evento posee una serie de métodos que permiten consultar la información del evento ocurrido.

Algunos métodos de la **clase EventObject**:

- **getSource( ):** Devuelve el objeto (fuente de eventos) sobre el que se ha producido el evento.

Algunos métodos de la clase ActionEvent:

- **getModifiers( ):** Devuelve un código que aporta información variada sobre el evento (por ejemplo, indica las combinaciones de teclas pulsadas: SHIFT+ALT).

- **getActionCommand( ):** Devuelve un string que ayuda a determinar el tipo de acción a ejecutar ante el evento. Tiene múltiples usos.

Algunos métodos de la clase **MouseEvent** (más en el API):

- **getModifiers( ):** Devuelve un código que aporta información variada sobre el evento (por ejemplo, indica el botón del ratón pulsado).

- **getX( )** y **getY( ):** Indican la coordenada X e Y donde se ha pulsado el ratón.

- **GetClickCount( ):** Indica el número de clicks seguidos que se han efectuado con el ratón.

Tipos de eventos generados por cada fuente de eventos

Cada componente (fuente de eventos), en función de la acción que se realice sobre él, generará distintos tipos de eventos. Por ejemplo:

Componente	Evento	Hecho que lo genera
Button	ActionEvent	El usuario hace un clic sobre el botón.
Checkbox	ItemEvent	El usuario selecciona o deselecciona el Checkbox
List	ActionEvent	El usuario hace doble click sobre un elementos de la lista
	ItemEvent	El usuario selecciona o deselecciona un elemento de la lista
Component	MouseEvent	El usuario pulsa o suelta un botón del ratón, el cursor del ratón entra o sale o el usuario mueve o arrastra el ratón
	FocusEvent	El componente gana o pierde el foco
	KeyEvent	El usuario pulsa o suelta una tecla
TextField	ActionEvent	El usuario termina de editar el texto (hace un intro)
Window	WindowEvent	La ventana se abre, se cierra, se minimiza, se reestablece o se cierra.

Ejemplo 1

Ventana con un botón y una etiqueta. Cuando se pulse el botón queremos que en la etiqueta se visualice la fecha y hora actual.

Pasos:

- Crear la ventana y todos sus componentes.
- Crear un escuchador que sea capaz de atender a los eventos generados al pulsar un botón.
- Evento generado: ActionEvent
- Interfaz que posee los métodos para atender ese evento: ActionListener
- Asignar ese escuchador al botón.
- Incorporar el código a ejecutar cuando se pulse el botón.

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.util.Date;

class EscuchadorBoton implements ActionListener
{
  JLabel etiqueta;
   public EscuchadorBoton(JLabel etiq)
   {
     t his.etiqueta = etiq;
    }
   public void actionPerformed(ActionEvent e)
   {
     etiqueta.setText("Botón Pulsado: " + new Date( ));
    }
 }

public class VentanaBoton extends JFrame
{
    JPanel panelBoton;
    JLabel etiqueta;
    JButton boton;
  public VentanaBoton( )
  {
    etiqueta = new JLabel( );
    panelBoton = new JPanel( );
    boton = new JButton("Pulsa Aquí");
    panelBoton.add(boton);
    this.getContentPane( ).setLayout(new BorderLayout( ));
    this.getContentPane( ).add(etiqueta,"North");
```

```
    this.getContentPane( ).add(panelBoton,"South");
    EscuchadorBoton escuchador = new EscuchadorBoton(etiqueta);
    boton.addActionListener(escuchador);
    this.setSize(300,100);
    this.setTitle("Ejemplo Sencillo");
    this.setVisible(true);
    this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
  }
  public static void main(String[ ] args)
  {
    new VentanaBoton( );
  }
}
```

Ejemplo 2

A menudo, y puesto que cualquier objeto puede convertirse en un escuchador, en lugar de crear un objeto aparte para el escuchador se pone como escuchador de los eventos generados por los elementos de la ventana a la propia ventana.

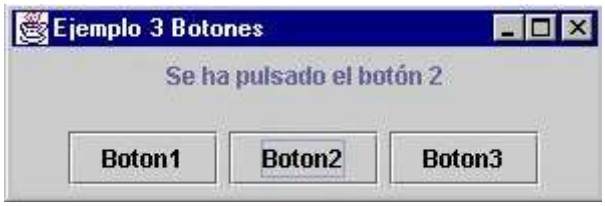
```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.util.Date;

public class VentanaBotonEscuchador extends JFrame implements ActionListener
{
  JPanel panelBoton;
  JLabel etiqueta;
  JButton boton;
  public VentanaBotonEscuchador( )
  {
    etiqueta = new JLabel( );
    panelBoton = new JPanel( );
    boton = new JButton("Pulsa Aquí");
    panelBoton.add(boton);
    this.getContentPane( ).setLayout(new BorderLayout( ));
    this.getContentPane( ).add(etiqueta,"North");
    this.getContentPane( ).add(panelBoton,"South");
    boton.addActionListener(this);
    this.setSize(300,100);
    this.setTitle("Ejemplo Sencillo");
    this.setVisible(true);
    this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
  }
```

```
  public void actionPerformed(ActionEvent e)
  {
    etiqueta.setText("Botón Pulsado: " + new Date( ));
  }
  public static void main(String[ ] args)
  {
    new VentanaBotonEscuchador( );
  }
}
```

Ejemplo 3

Varias fuentes de eventos (del mismo tipo). Tenemos una ventana con tres botones y una etiqueta. Si asignamos a los tres botones un mismo escuchador (la propia ventana), ¿Cómo podemos averiguar dentro del método actionPerformed( ) el botón que se ha pulsado? Solución: consultado la información que viene en el evento.



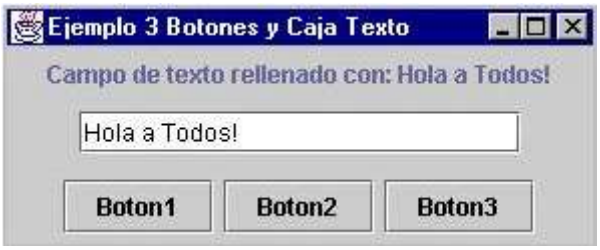

```
import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.util.Date;

public class VentanaBotones extends JFrame implements ActionListener
{
  JPanel panelBotones;
  JPanel panelEtiqueta;
  JLabel etiqueta;
  JButton boton1;
  JButton boton2;
  JButton boton3;
  public VentanaBotones( )
  {
    etiqueta = new JLabel( );
    panelBotones = new JPanel( );
    panelEtiqueta = new JPanel( );
    boton1 = new JButton("Boton1");
    boton2 = new JButton("Boton2");
    boton3 = new JButton("Boton3");
    panelEtiqueta.add(etiqueta);
    panelBotones.add(boton1);
```

```
    panelBotones.add(boton2);
    panelBotones.add(boton3);
    this.getContentPane().setLayout(new BorderLayout( ));
    this.getContentPane().add(panelEtiqueta,"North");
    this.getContentPane().add(panelBotones,"South");
    boton1.addActionListener(this);
    boton2.addActionListener(this);
    boton3.addActionListener(this);
    this.setSize(300,100);
    this.setTitle("Ejemplo 3 Botones");
    this.setVisible(true);
    this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
  }
  public void actionPerformed(ActionEvent e)
  {
    JButton botonPulsado = (JButton)e.getSource( );
    if (botonPulsado == boton1)
    etiqueta.setText("Se ha pulsado el botón 1");
    if (botonPulsado == boton2)
    etiqueta.setText("Se ha pulsado el botón 2");
    if (botonPulsado == boton3)
    etiqueta.setText("Se ha pulsado el botón 3");
  }
  public static void main(String[ ] args)
  {
     new VentanaBotones( );
  }
}
```

Ejemplo 4

Varias fuentes de eventos (de distinto tipo). Tenemos una ventana con tres botones, una caja de texto y una etiqueta. En este caso, ¿cómo averiguamos si el elemento sobre el que se produce el evento es un botón o la caja de texto? Solución: consultando la información del evento (igual que el anterior).




```
public class VentanaBotonesTexto extends JFrame implements ActionListener
{
  ...
  JButton boton1;
  JButton boton2;
  JButton boton3;
  JTextField texto;
```

```
  public VentanaBotonesTexto()
  {
    ...
    boton1.addActionListener(this);
    boton2.addActionListener(this);
    boton3.addActionListener(this);
    texto.addActionListener(this);
    ...
  }
  public void actionPerformed(ActionEvent e)
  {
    Object elemento = e.getSource( );
    if (elemento == boton1)
    etiqueta.setText("Se ha pulsado el botón 1");
    if (elemento == boton2)
    etiqueta.setText("Se ha pulsado el botón 2");
    if (elemento == boton3)
    etiqueta.setText("Se ha pulsado el botón 3");
    if (elemento == texto)
    etiqueta.setText("Campo de texto rellenado con: " + texto.getText( ));
  }
...
}
```

Si en algún momento necesitásemos conocer el tipo al que pertenece alguno de los objetos, podríamos hacer:

```
Object elemento = e.getSource( );
if (elemento instanceof JButton)
{
  System.out.println("El elemento es un botón");
  JButton boton = (JButton)elemento;
}
if (elemento instanceof JTextField)
{
  System.out.println("El elemento es de texto");
  JTextField cajaTexto = (JTextField)elemento;
}
```

Ejemplo 5

En este ejemplo se capturan distintos tipos de eventos:

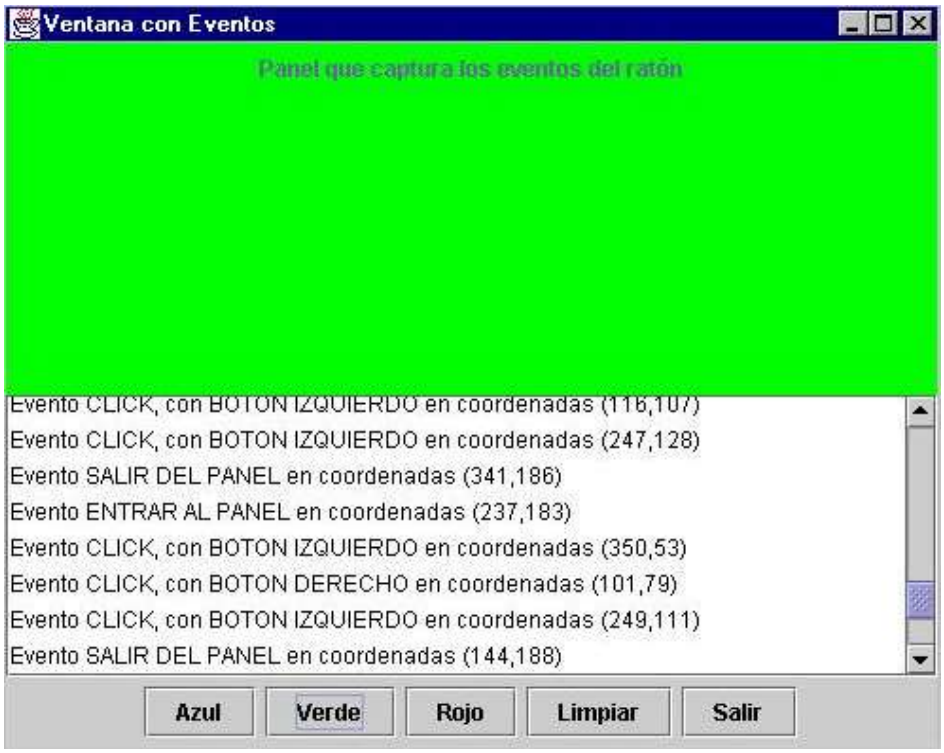
Eventos generados al pulsar botones
- Evento: ActionEvent
- Interfaz: ActionListener
- Acción a realizar: cambiar el color del panel superior

Eventos generados al mover y pulsar el ratón sobre un panel
- Evento: MouseEvent

- Interfaz: MouseListener
- Acción a realizar: Visualizar en una lista el tipo de evento que se ha producido

Eventos generados al cerrar, minimizar, maximizar,… la ventana
- Evento: WindowEvent
- Interfaz: WindowListener
- Acción a realizar: cerrar la ventana y terminar el programa.

Aspecto de la ventana:




```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JList;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
import javax.swing.DefaultListModel;
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.MouseEvent;
import java.awt.event.WindowEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseListener;
import java.awt.event.WindowListener;
```

```
public class VentanaEventos extends JFrame implements ActionListener,
MouseListener, WindowListener{

JPanel panelSuperior;
JPanel panelInferior;
JPanel panelBotones;
JButton botonAzul;
JButton botonVerde;
JButton botonRojo;
JButton botonLimpiar;
JButton botonSalir;
JLabel etiqueta;
JList listaEventos;
DefaultListModel datosLista;
JScrollPane scrollLista;
  public VentanaEventos( )
  {
      panelSuperior = new JPanel( );
      panelInferior = new JPanel( );
      panelBotones = new JPanel( );
      botonAzul = new JButton("Azul");
      botonVerde = new JButton("Verde");
      botonRojo = new JButton("Rojo");
      botonLimpiar = new JButton("Limpiar");
      botonSalir = new JButton("Salir");
      etiqueta = new JLabel("Panel que captura los eventos del ratón");
      datosLista = new DefaultListModel( );
      listaEventos = new JList(datosLista);
      scrollLista = new JScrollPane(listaEventos);
      panelBotones.add(botonAzul);
      panelBotones.add(botonVerde);
      panelBotones.add(botonRojo);
      panelBotones.add(botonLimpiar);
      panelBotones.add(botonSalir);
      panelSuperior.add(etiqueta);
      panelInferior.setLayout(new BorderLayout( ));
      panelInferior.add(scrollLista, "Center");
      panelInferior.add(panelBotones, "South");
      this.getContentPane( ).setLayout(new
      GridLayout(2,1));
      this.getContentPane( ).add(panelSuperior);
      this.getContentPane( ).add(panelInferior);
      // Asignamos escuchadores a todos los elementos que puedan recibir eventos.
      // Ponemos como escuchador de todos ellos a la propia ventana.
    botonAzul.addActionListener(this);
    botonVerde.addActionListener(this);
    botonRojo.addActionListener(this);
    botonLimpiar.addActionListener(this);
    botonSalir.addActionListener(this);
    panelSuperior.addMouseListener(this);
```

```
    this.addWindowListener(this);
    this.setTitle("Ventana con Eventos");
    this.setSize(500,400);
    this.setVisible(true);
    this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
    }

    // Método del interfaz ActionListener

    public void actionPerformed(ActionEvent e)
    {
        JButton botonPulsado = (JButton)e.getSource();
        if (botonPulsado == botonAzul)
        {
          panelSuperior.setBackground(Color.blue);
        }
        if (botonPulsado == botonVerde)
        {
          panelSuperior.setBackground(Color.green);
        }
        if (botonPulsado == botonRojo)
        {
          panelSuperior.setBackground(Color.red);
        }
        if (botonPulsado == botonLimpiar)
        {
          datosLista.clear();
          panelSuperior.setBackground(Color.lightGray);
        }
        if (botonPulsado == botonSalir)
        {
          System.exit(0);
        }
    }

// Los 5 métodos del interfaz MouseListener

public void mouseClicked(MouseEvent e)
    {
        String linea = "Evento CLICK, ";
        if (e.getModifiers() == MouseEvent.BUTTON1_MASK)
        {
          linea = linea + "con BOTON IZQUIERDO ";
        }
        if (e.getModifiers() == MouseEvent.BUTTON2_MASK)
        {
          linea = linea + "con BOTON DEL CENTRO ";
        }
        if (e.getModifiers() == MouseEvent.BUTTON3_MASK)
       {
```

```
            linea = linea + "con BOTON DERECHO ";
           }
            linea = linea + "en coordenadas (" + e.getX( ) + "," + e.getY( ) + ")";
            datosLista.addElement(linea);
    }

    public void mouseEntered(MouseEvent e)
    {
        String linea = "Evento ENTRAR AL PANEL en coordenadas (" + e.getX( ) + ","
        + e.getY( ) + ")";
       datosLista.addElement(linea);
    }

    public void mouseExited(MouseEvent e)
    {
         String linea = "Evento SALIR DEL PANEL en coordenadas (" + e.getX( ) + ","
         +e.getY( ) + ")";
         datosLista.addElement(linea);
    }

    public void mousePressed(MouseEvent e)
    {
        // Este método se invoca cuando se pulsa el botón del ratón.
    }
    public void mouseReleased(MouseEvent e)
    {
        // Este método se invoca cuando se libera el botón del ratón.
    }

//Los 7 métodos del interfaz WindowListener

    public void windowClosing(WindowEvent e)
    {
       System.exit(0);
    }
    public void windowClosed(WindowEvent e){ }
    public void windowIconified(WindowEvent e){ }
    public void windowOpened(WindowEvent e){ }
    public void windowActivated(WindowEvent e){ }
    public void windowDeactivated(WindowEvent e){ }
    public void windowDeiconified(WindowEvent e){ }

  public static void main(String[ ] args)
  {
    new VentanaEventos( );
  }
}
```

Clases adaptadoras de eventos

Muchas interfaces **EventListener** están diseñadas para recibir múltiples clases de eventos, por ejemplo, la interfaz MouseListener puede recibir eventos de pulsación de botón, al soltar el botón, a la recepción del cursor, etc.

La interfaz declara un método para cada uno de estos subtipos. Cuando se implementa una interfaz, es necesario redefinir todos los métodos que se declaran en esa interfaz, incluso aunque se haga con métodos vacíos. En la mayoría de las ocasiones, no es necesario redefinir todos los métodos declarados en la interfaz porque no son útiles para la aplicación.

Por ello, el AWT proporciona un conjunto de **clases abstractas adaptadores (Adapter) que coinciden con los interfaces**. **Cada clase adaptador implementa una interfaz y redefine todos los métodos declarados por la interfaz con métodos vacíos, con lo cual se satisface ya el requerimiento de la redefinición de todos los métodos.**

Se pueden definir clases Receptor extendiendo clases adaptadores, en vez de implementar la interfaz receptora correspondiente. Esto proporciona libertad al programador para redefinir solamente aquellos métodos de la interfaz que intervienen en la aplicación que desarrolla.
De nuevo, hay que recordar que todos los métodos declarados en una interfaz corresponden a los tipos de eventos individuales de la clase de eventos correspondiente, y que el objeto Fuente notifica al Receptor la ocurrencia de un evento de un tipo determinado invocando al método redefinido del interfaz.

Las clases Adaptadores que se definen en el la plataforma Java 2 son las que se indican a continuación:

- java.awt.ComponentAdapter
- java.awt.FocusAdapter
- java.awt.KeyAdapter
- java.awt.MouseAdapter
- java.awt.MouseMotionAdapter
- java.awt.WindowAdapter

Si se compila y ejecuta el siguiente ejemplo, cada vez que se pulse el botón del ratón con el cursor dentro de la ventana, aparecerán las coordenadas en las que se encuentra el cursor, tal como muestra la figura anterior.

```
import java.awt.*;
import java.awt.event.*;

public class EjAdaptadoras1 {
   public static void main( String args[ ] ) {
      // Aqui se instancia un objeto de tipo Interfaz Hombre-Maquina
      IHM ihm = new IHM( );
   }
}
```

```
// Se crea una subclase de Frame para poder sobreescribir el método
// paint(), y presentar en pantalla las coordenadas donsde se haya
// producido el click del ratón
class MiFrame extends Frame {
   int ratonX;
   int ratonY;

   public void paint( Graphics g ) {
      g.drawString( ""+ratonX+", "+ratonY,ratonX,ratonY );
   }
}

class IHM {
   public IHM( ) {
      MiFrame ventana = new MiFrame( );

      ventana.setSize( 300,300 );
      ventana.setTitle( "Clases adaptadoras, ejemplos" );
      ventana.setVisible( true );

      // Se instancia y registra un objeto receptor de eventos
      // para terminar la ejecución del programa cuando el
      // usuario decida cerrar la ventana.
      Proceso1 procesoVentana1 = new Proceso1( );
      ventana.addWindowListener( procesoVentana1 );

      // Se instancia y registra un objeto receptor de eventos
      // que será el encargado de procesar los eventos del raton
      // para determinar y presentar las coordenadas en las que
      // se encuentra el cursor cuando el usuario pulsa el botón
      // del ratón.
      ProcesoRaton procesoRaton = new ProcesoRaton( ventana );
      ventana.addMouseListener( procesoRaton );
   }
}

class ProcesoRaton extends MouseAdapter {
   MiFrame ventanaRef; // Referencia a la ventana

   // Constructor
   ProcesoRaton( MiFrame ventana ) {
      // Guardamos una referencia a la ventana
      ventanaRef = ventana;
   }

   // Se sobreescribe el método mousePressed para determinar y
   // presentar en pantalla las coordenadas del cursor cuando
   // se pulsa el ratón.
   public void mousePressed( MouseEvent evt ) {
```

```
            // Recoge las coordenadas X e Y de la posición del cursor
            // y las almacena en el objeto Frame
            ventanaRef.ratonX = evt.getX( );
            ventanaRef.ratonY = evt.getY( );
            // Finalmente, presenta los valores de las coordenadas
            ventanaRef.repaint( );
        }
    }

    class Proceso1 extends WindowAdapter {
        public void windowClosing( WindowEvent evt ) {
            System.exit( 0 );
        }
    }
```

En el caso más simple, los eventos de bajo nivel del modelo de Delegación de Eventos, se pueden controlar siguiendo los pasos que se indican a continuación:

- Definir una clase Listener, Receptor, para una determinada clase de evento que implemente la interfaz receptora que coincida con la clase de evento, **o extender la clase adaptadora correpondiente**.

- Redefinir los métodos del interfaz receptora para cada tipo de evento específico de la clase evento, para poder implementar la respuesta deseada del programa ante la ocurrencia de un evento. Si se implementa la interfaz receptora, hay que redefinir todos los métodos del interfaz. **Si se extiende la clase adaptadora, se pueden redefinir solamente aquellos métodos que son de interés.**

- Definir una clase Source, fuente, que instancie un objeto de la clase receptor y registrarla para la notificación de la ocurrencia de eventos generados por cada componente específico.

```
import java.awt.*;
import java.awt.event.*;

public class EjAdaptadoras2 {
    public static void main( String args[ ] ) {
        IHM ihm = new IHM( );
    }
}

class MiFrame extends Frame {
    int ratonX;
    int ratonY;

    MiFrame( String nombre ) {
        setTitle( "Clases adaptadoras, Eventos" );
        setSize( 500,500 );
        setName( nombre );
```

```
    }

    public void paint( Graphics g ) {
        // Presenta en pantalla las coordenadas del cursor
        g.drawString( ""+ratonX+", "+ratonY,ratonX,ratonY );
    }
}

class IHM {
    public IHM() {
        // Se crean dos objetos visuales de tipo Frame, se le hace
        // visibles y se les asignan los nombres Frame1 y Frame2
        MiFrame miFrame1 = new MiFrame( "Frame1" );
        miFrame1.setVisible( true );

        MiFrame miFrame2 = new MiFrame( "Frame2" );
        miFrame2.setLocation(500,500);
        miFrame2.setVisible( true );
        Proceso1 procesoVentana1 = new Proceso1( );
        miFrame1.addWindowListener( procesoVentana1 );
        miFrame2.addWindowListener( procesoVentana1 );
        ProcesoRaton procesoRaton = new ProcesoRaton( miFrame1,miFrame2 );
        miFrame1.addMouseListener( procesoRaton );
        miFrame2.addMouseListener( procesoRaton );
    }
}

class ProcesoRaton extends MouseAdapter{
    // Variables para guardar referencias a los objetos
    MiFrame frameRef1,frameRef2;

    // Constructor
    ProcesoRaton( MiFrame frame1,MiFrame frame2 ) {
        frameRef1 = frame1;
        frameRef2 = frame2;
    }

    // Se sobrescribe el metodo mousePressed( ) para controlar la
    // respuesta cuando el ratón se pulse sobre uno de los dos
    // objetos Frame
    public void mousePressed( MouseEvent evt ) {
        if( evt.getComponent( ).getName().compareTo( "Frame1" ) == 0 ) {
            // Recoge las coordenadas X e Y de la posicion del cursor
            // y las almacena en el objeto Frame
            frameRef1.ratonX = evt.getX( );
            frameRef1.ratonY = evt.getY( );
            frameRef1.repaint( );
        }
        else {
            // Recoge las coordenadas X e Y de la posicion del cursor y las almacena en el
```

```
            // objeto Frame
            frameRef2.ratonX = evt.getX( );
            frameRef2.ratonY = evt.getY( );
            frameRef2.repaint( );
        }
    }
}

class Proceso1 extends WindowAdapter {
    public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
    }
}
```

El código del ejemplo es simple, la única parte críptica es la que trata de obtener el nombre del componente visual que ha generado el evento mousePressed().
El método main() instancia un objeto de tipo IHM, que sirve para dos propósitos, por un lado proporcionar la interfaz visual y, por otro, actuar como una fuente de eventos que notificará su ocurrencia a los objetos receptor que se registren con él.

La clase Frame es extendida en una nueva clase llamada MiFrame, para permitir la redefinición del método paint() de la clase; lo cual es imprescindible para presentar las coordenadas en donde se encuentra el cursor sobre el Frame, utilizando el método **drawString( ).**

El constructor de la clase IHM instancia dos objetos de tipo MiFrame y los hace visibles. Cuando son instanciados, se les asignan los nombre Frame1 y Frame2 a través del método setName(). Estos nombres serán los que permitan determinar posteriormente cuál ha sido el objeto que ha generado el evento.

También en ese mismo constructor se instancia a un solo objeto receptor a través del cual se procesarán todos los eventos de bajo nivel que se produzcan en cualquiera de los dos objetos visuales:

```
ProcesoRaton procesoRaton = new ProcesoRaton( miFrame1,miFrame2 );
miFrame1.addMouseListener( procesoRaton );
miFrame2.addMouseListener( procesoRaton );
```

La primera sentencia solamente instancia el nuevo objeto receptor procesoRaton, pasándole las referencias de los dos elementos visuales como parámetro. Las dos sentencias siguientes incorporan este objeto receptor (lo registran) a la lista de objetos receptor que serán automáticamente notificados cuando suceda cualquier evento de ratón sobre los objetos visuales referenciados como miFrame1 y miFrame2, respectivamente. Y ya no se requiere ningún código extra para que se produzca esa notificación.

Las notificaciones se realizan invocando métodos de instancia específicos redefinidos del objeto receptor ante la ocurrencia de tipos determinados de eventos del ratón. Las declaraciones de todos los métodos deben coincidir con todos los posibles eventos del ratón que estén definidos en la interfaz MouseListener, que deben coincidir con los

definidos en la clase MouseEvent. La clase desde la que el objeto receptor es instanciado debe redefinir, bien directa o indirectamente, todos los métodos declarados en el interfaz MouseListener.
Además de registrar el objeto MouseListener para recibir objetos de ratón, el programa también instancia y registra un objeto receptor que monitoriza los eventos de la ventana, y termina la ejecución del programa cuando el usuario cierra uno cualquiera de los objetos visuales.

```
Proceso1 procesoVentana1 = new Proceso1();
miFrame1.addWindowListener( procesoVentana1 );
miFrame2.addWindowListener( procesoVentana1 );
```

La parte más complicada de la programación involucra al objeto receptor del ratón, y aún así, es bastante sencilla. Lo que intenta el código es determinar cuál de los dos elementos visuales ha sido el que ha generado el evento. En este caso el objeto receptor solamente a eventos mousePressed, aunque lo que se explica a continuación se podría aplicar a todos los eventos del ratón y, probablemente, a muchos de los eventos de bajo nivel.
La clase ProcesoRaton (receptor) en este programa extiende la clase MouseAdapter y redefine el método mousePressed( ) que está declarado en el interfaz MouseListener. Cuando es invocado el método mousePressed( ), se le pasa un objeto de tipo MouseEvent, llamado evt, como parámetro.
Para determinar si el objeto que ha generado el evento ha sido Frame1, se utiliza la siguiente sentencia:

```
if( evt.getComponent( ).getName( ).compareTo( "Frame1" ) == 0 ) {
```

El método getComponent( ) es un método que devuelve el objeto donde se ha generado el evento. En este caso devuelve un objeto de tipo Component sobre el cual actúa el método getName( ). Este último método devuelve el nombre del Componente como un objeto String, sobre el cual actúa el método compareTo( ). Este método es estándar de la clase String y se utiliza para comparar dos objetos String. En este caso se utiliza para comparar el nombre del componente con la cadena "Frame1"; si es así, el código que se ejecuta es el que presenta las coordenadas del ratón sobre el objeto visual Frame1; si no coincide se ejecuta el código de la cláusula else que presentará las coordenadas sobre el objeto visual Frame2.

Aunque en el ejemplo anterior se utilizan dos objetos visuales del mismo tipo, no hay razón alguna para que eso sea así, ya que todos los objetos visuales comparten el mismo objeto receptor y son capaces de generar eventos para los que el receptor está registrado.
Actualmente se incorporan eventos de ratón para detectar movimientos de la rueda superior que incorporan la mayoría de estos dispositivos. Por lo tanto, es posible incorporar un receptor de eventos de tipo MouseWheelListener a cualquier componente para que éste reaccione.

El receptor dispone solamente del método **mouseWheelMoved( )** , que recoge un evento de tipo **MouseWheelEvent**. Entre la información que proporciona el evento se puede obtener la cantidad de desplazamiento mediante **getScrollAmount( )** si se está desplazando una unidad o un bloque con **getScrollRotation( )** y, por conveniencia, el número de unidades a desplazar mediante **getUnitsScroll( )**.