IPOO

- Docentes
  - Profesores Adjuntos:
    - Carlos Di Cicco.
    - Diego de la Riva.
  - JTPs:
    - Federico Naso (Junín) .
      Nelson Di Grazia (Pergamino) .
  - Ayudante:
    - Sebastián Sottile (Junín).

Awt y Swing

Swing es un conjunto de clases desarrolladas por primera vez para Java 1.2 (el llamado Java2), para mejorar el anterior paquete que implementaba clases para fabricar interfaces de usuario, el llamado AWT (Abstract Window Tools) que aún se usa bastante en las aplicaciones Java.

Tanto Swing como AWT forman parte de una colección de clases llamada JFC (Java Foundation Classes) que incluyen paquetes dedicados a la programación de interfaces gráficos (así como a la producción multimedia).

Awt y Swing

- Uno de los problemas frecuentes de la programación clásica era como programar interfaces de usuario, ya que esto implicaba tener que utilizar las API propias del Sistema Operativo y esto provocaba que el código no fuera transportable a otros sistemas.

- AWT fue la primera solución a este problema propuesta por Java. Está formada por un conjunto de clases que no dependen del sistema operativo, pero que proponen una serie de clases para la programación de GUIs.

- AWT usa clases gráficas comunes a todos los sistemas operativos gráficos y luego la máquina virtual traduce esa clase a la forma que tenga en el sistema concreto en el que se ejecutó el programa, sin importar que dicho sistema sea un Linux, Mac o Windows. La popularidad de AWT desbordó las expectativas de la propia empresa Sun.

Awt y Swing

- La clave de AWT era el uso de componentes *iguales (peers). Los elementos de los* interfaces AWT dejaban al sistema la responsabilidad de generar realmente los componentes. Eso aseguraba una vista coherente respecto al sistema en el que se ejecutaba el programa.

- El problema es que mientras las GUIs se veian correctamente en Windows y Mac OS, otros sistemas quedaban peor ante la misma aplicación.

- Por ello (y por otros problemas) aparece Swing en la versión 1.2 como parte del JFC (*Java Foundation Classes) que es el kit de clases más importante de Java para las* producciones gráficas.

Awt y Swing

Los problemas de AWT eran:

- AWT tenía problemas de compatibilidad en varios sistemas.
- A AWT le faltaban algunos componentes avanzados (árboles, tablas,...).
- Consumía excesivos recursos del sistema.
- Swing aporta muchas más clases, consume menos recursos y construye mejor la apariencia de los programas. En cualquier caso, AWT no desaparece; simplemente se añade a las nuevas capacidades Swing

Componentes

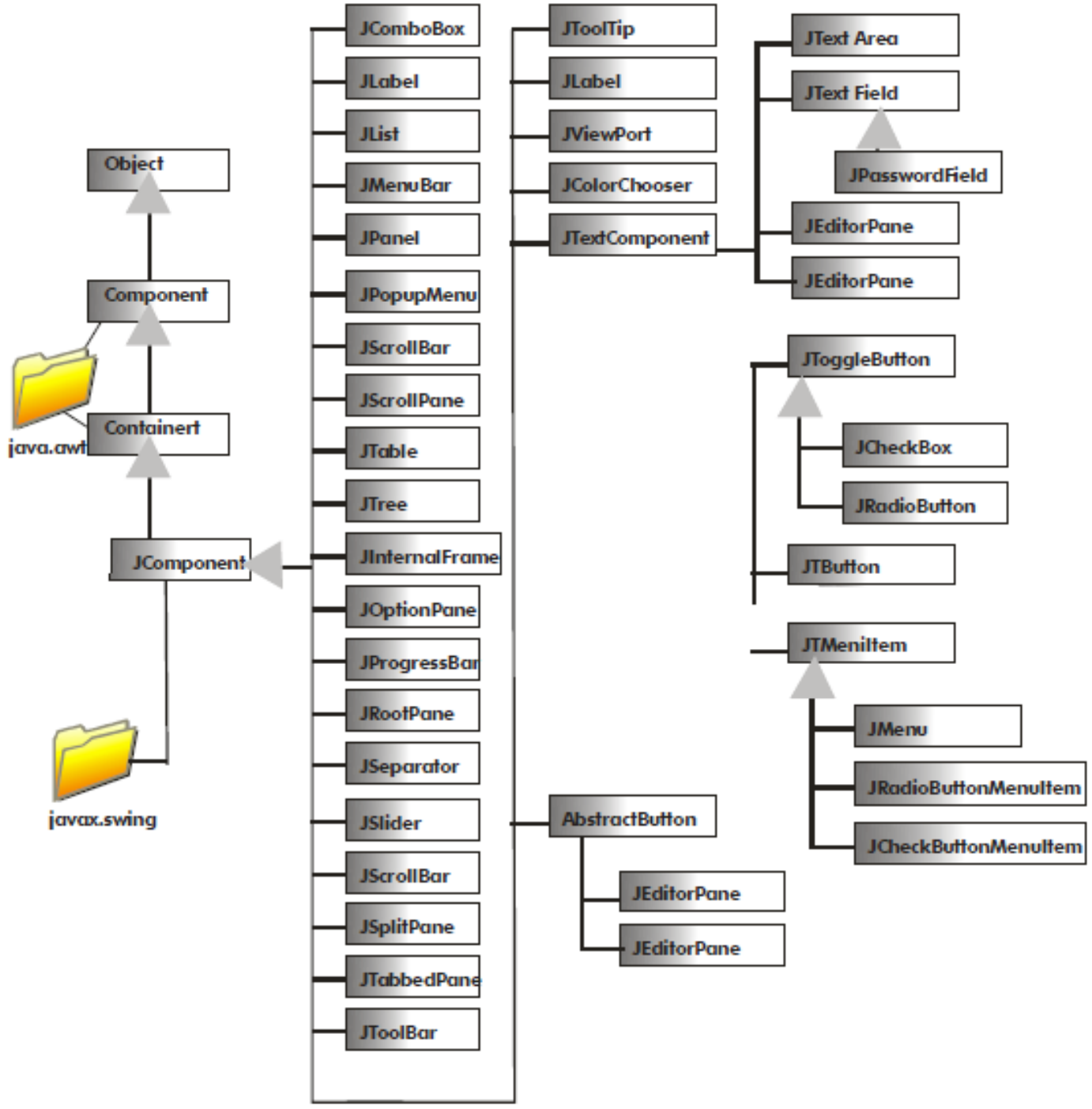
- Los componentes son los elementos básicos de la programación con Swing.

- Todo lo que se ve en un GUI de Java es un componente. Los componentes se colocan en otros elementos llamados contenedores que sirven para agrupar componentes.

- Un administrador de diseño se encarga de disponer la presentación de los componentes en un dispositivo de presentación concreto.

- La clase javax.swing.JComponent es la clase padre de todos los componentes. A su vez, JComponent desciende de java.awt.container y ésta de java.awt.component.

- De esto se deduce que Swing es una extensión de AWT, de hecho su estructura es análoga.


Object
Component
Containert
JComponent
java.awt
javax.swing
JComboBox
JLabel
JList
JMenuBar
JPanel
JPopupMenu
JScrollBar
JScrollPane
JTable
JTree
JInternalFrame
JOptionPane
JProgressBar
JRootPane
JSeparator
JSlider
JScrollBar
JSplitPane
JTabbedPane
JToolBar
JToolTip
JLabel
JViewPort
JColorChooser
JTextComponent
JTextArea
JTextField
JPasswordField
JEditorPane
JEditorPane
JToggleButton
JCheckBox
JRadioButton
JTButton
JTMenuItem
JMenu
JRadioButtonMenuItem
JCheckButtonMenuItem
AbstractButton
JEditorPane
JEditorPane

Componentes

- La clase JComponent posee métodos para controlar la apariencia del objeto. Por ejemplo: la visibilidad, tamaño, posición, tipo de letra, color,... Al dibujar un componente, se le asigna un **dispositivo de presentación.**

- Además posee métodos que controlan el comportamiento del componente. Cuando el usuario ejecuta una acción sobre un componente, entonces se crea un objeto de evento que describe el suceso.

- El objeto de evento se envía a objetos de control de eventos (*Listeners)*. *Los eventos son uno de los pilares de la construcción de Interfaces* de usuario y una de las bases de la comunicación entre objetos.

Componentes peso pesado contra peso ligero

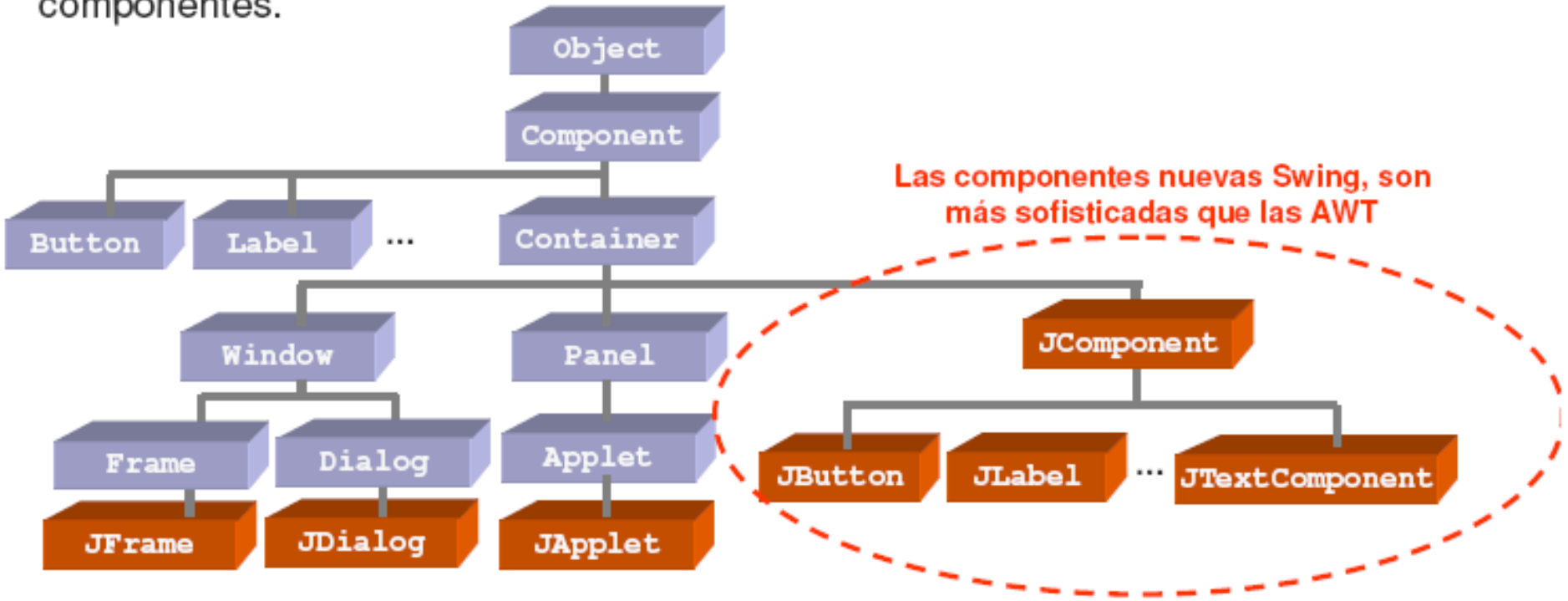
- ¿Por qué no se añadieron a AWT todas estas mejoras y nuevos componentes? Esta pregunta hace que volvamos de nuevo a ver cuál es la diferencia básica en la filosofía de los componentes AWT y Swing.

- Cada componente AWT obtiene su propia ventana de la plataforma (y por lo tanto es como si fuera un control estándar de esa plataforma). En programas extensos, todas estas ventanas ralentizan el rendimiento y consumen gran cantidad de memoria. A tales componentes se les ha denominado “pesos pesados”.

- Los controles Swing, por otro lado, son sencillamente dibujados como imágenes en sus contenedores y no tienen una ventana de la plataforma propia, por lo que usan bastantes menos recursos del sistema. Por este motivo, se les denomina componentes “peso ligero”.

Jerarquía de componentes

Las componentes básicas de Swing son subclases de "Container" (las AWT son subclases de Component) por lo tanto están capacitadas para contener a otras componentes.



Modelo / vista / controlador

- Se trata del modelo fundamental del trabajo con interfaces de usuario por parte de Swing. Consiste en tres formas de abstracción. Un mismo objeto se ve de esas tres formas:

- **Modelo.** Se refiere al modelo de datos que utiliza el objeto. Es la información que se manipula mediante el objeto Swing.
- **Vista.** Es cómo se muestra el objeto en la pantalla.
- **Controlador.** Es lo que define el comportamiento del objeto.

- Por ej un array de cadenas que contenga los meses del año, podría ser el modelo de un cuadro combinado de Windows. Un cuadro combinado es un rectángulo con un botón con una flecha que permite elegir una opción de una lista. La vista de ese cuadro es el hecho de mostrar esas cadenas en ese rectángulo con flecha. Y el controlador es la capa software que permite capturar el clic del ratón cuando apunta a la flecha del control a fin de mostrar y seleccionar el contenido.

Conexión de vistas y modelo

Para el diseño de aplicaciones con interfaces sofisticados se utiliza el patrón de diseño Modelo Vista Controlador (MVC). La lógica de un interfaz de usuario cambia con más frecuencia que los datos que tenemos almacenados y la lógica de negocio. Si realizamos un diseño complicado, es decir, una mezcla de componentes de interfaz y de negocio, entonces cuando necesitemos cambiar la interfaz, tendremos que modificar trabajosamente los componentes de negocio → mayor trabajo y más riesgo de error.

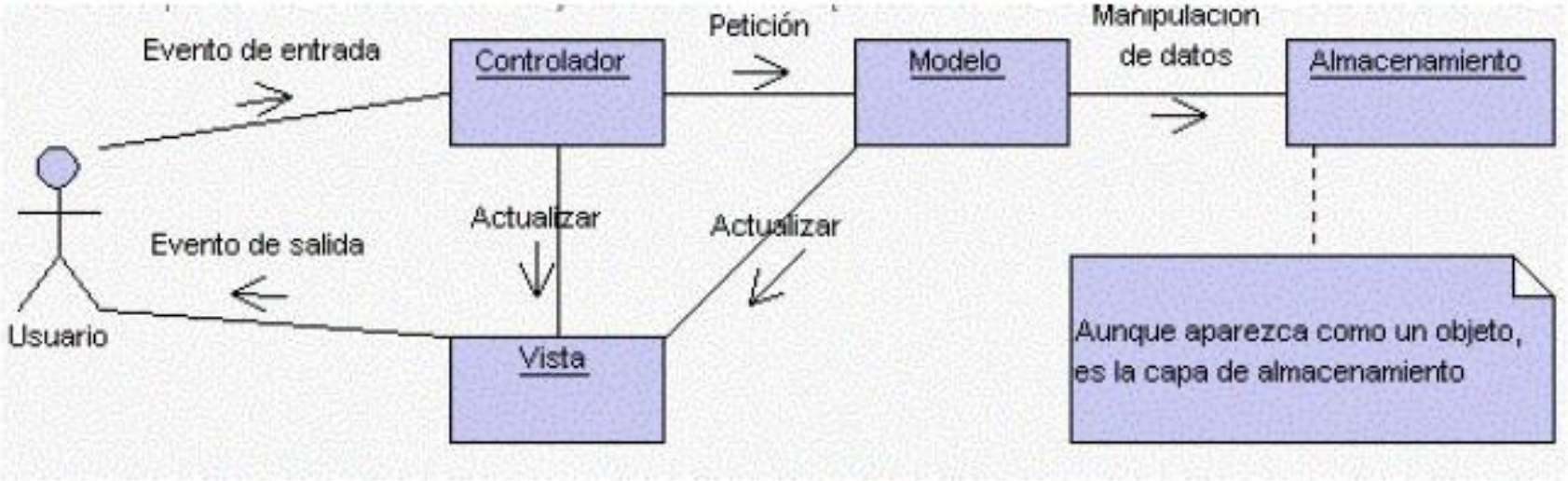
MVC fomenta un diseño que desacople la vista del modelo, con la finalidad de mejorar la reusabilidad. De esta forma las modificaciones en las vistas impactan en menor medida en la lógica de negocio o de datos.

Elementos del patrón:

- **Modelo**: datos y reglas de negocio
- **Vista**: muestra la información del modelo al usuario
- **Controlador**: gestiona las entradas del usuario

Conexión de vistas y modelo

En el siguiente diagrama se puede observar que los eventos de entrada del usuario son procesados por el controlador, quien hace invocaciones al modelo y actuliza la Vista, que será mostrada al usuario.

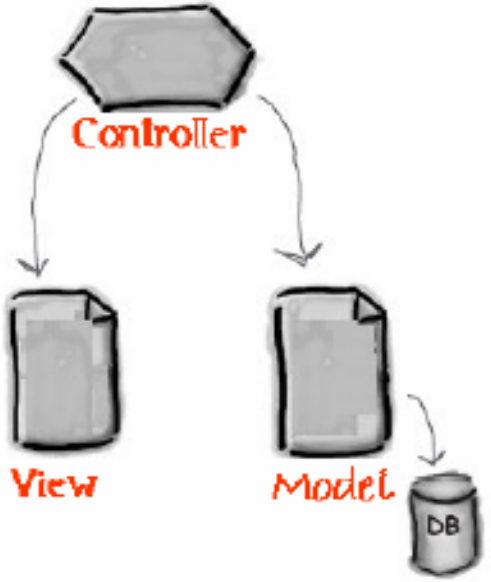


¿Cuál es el rol de cada una de las componentes del MVC?

Conexión de vistas y modelo

Recibe los eventos de entrada (un click, un cambio en un campo de texto, etc.). Contiene reglas de gestión de eventos, del tipo "Si el evento Z, entonces la 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)".

Recibir datos del modelo y los muestra al usuario. Tienen un registro de su controlador asociado. Pueden dar el servicio de "Actualización()", para que sea invocado por el controlador o por el modelo



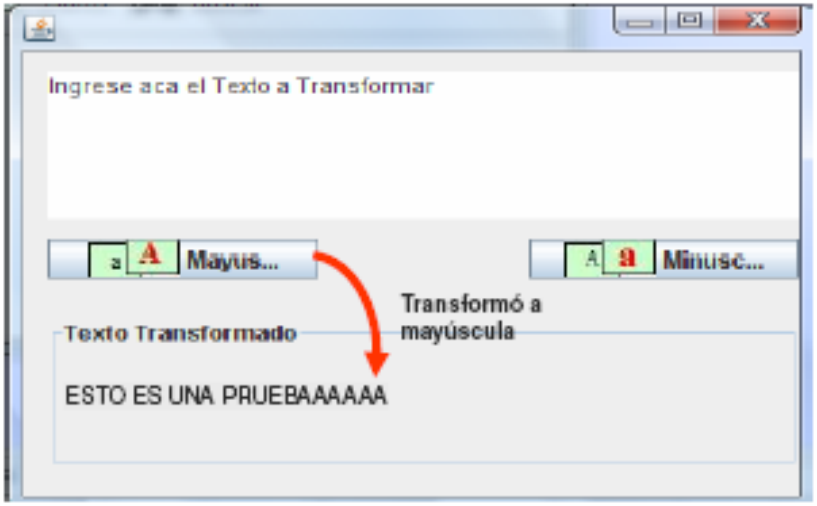
Accede a la capa de almacenamiento de datos. Lo ideal es que el modelo sea independiente del sistema de almacenamiento. Define las reglas de negocio (la funcionalidad del sistema). Lleva un registro de las vistas y controladores del sistema.

Un ejemplo de regla puede ser: "Si la mercadería pedida no está en el almacén, consultar el tiempo de entrega al proveedor".

Conexión de vistas y modelo. Ejemplo

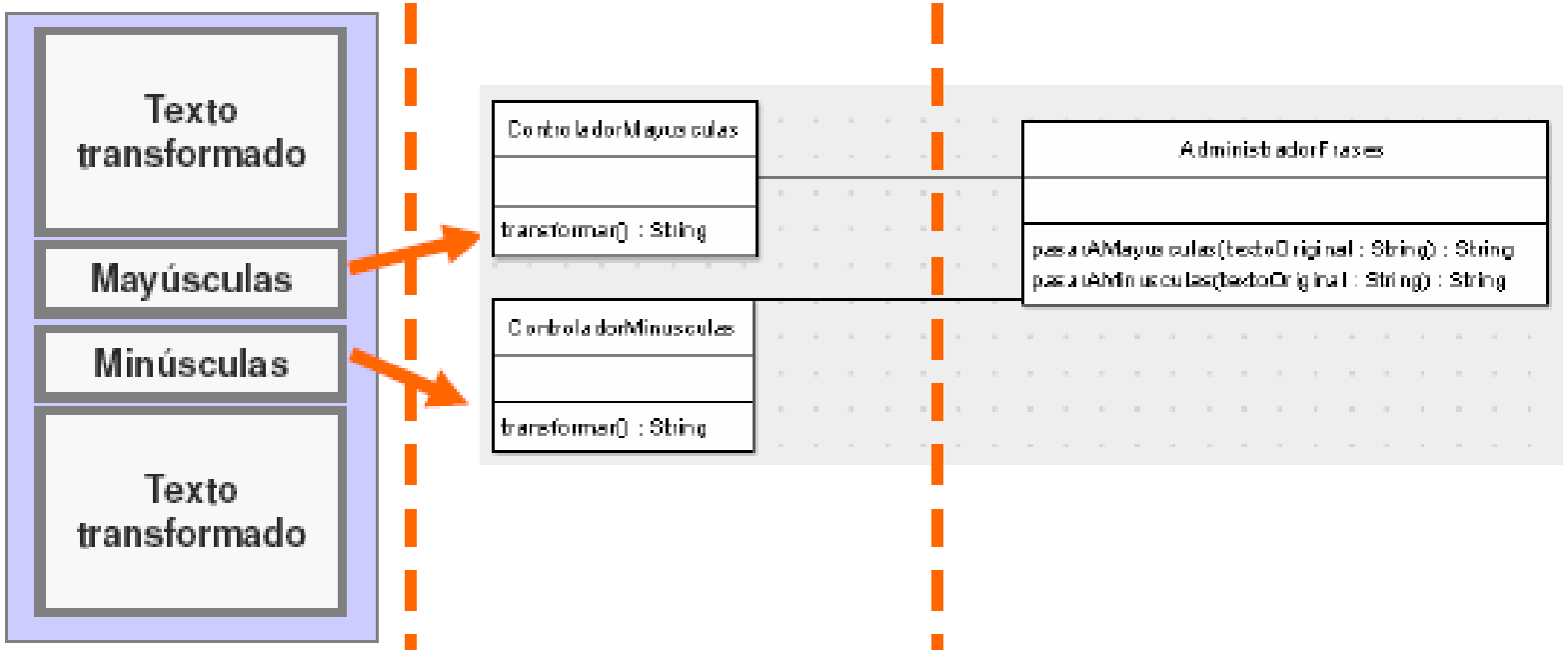
Vamos a realizar un ejemplo para poder entender bien como separar la interfaz del usuario o **vista** de nuestro **modelo** (lógica de negocio) y como conectar estas partes a través de un **controlador** que será el encargado de hacerlo.
El ejemplo se desarrollará usando Eclipse y un plug-in que implementa un editor gráfico para interfaces de usuario gráfica (**GUI**), llamado **Jigloo.** Este plug-in, permite desarrollar la interfaz de usuario arrastrando componentes sobre contenedores y configurando propiedades a través de ventanas y **genera código java automáticamente**.

Nuestro simple ejemplo consistirá en hacer una interfaz gráfica que sea capaz de tomar un texto ingresado por el usuario y darle la opción de poder pasar todo el texto a mayúsculas o pasar todo el texto a minúsculas. Una posible vista es esta:



Conexión de vistas y modelo. Ejemplo

Usando el patrón MVC podemos pensar en dividir nuestra aplicación en las partes :



Vista

Se muestra los datos y permite el ingreso de texto

Controlador

Recibe un evento y lo procesa.

Modelo

Esta clase hará el trabajo de transformar el texto.

JComponent

- La clase JComponent es abstracta, lo cual significa que no puede crear objetos, pero sí es la superclase de todos los componentes visuales (botones, listas, paneles, applets,...) y por ello la lista de métodos es interminable, ya que proporciona la funcionalidad de todos los componentes. Además puesto que deriva de **Component y Container tiene** los métodos de estos, por ello aún es más grande esta lista.

- Algunos son:

métodos de información

método	uso
String **getName()**	Obtiene el nombre del componente
void **setname(**String nombre**)**	cambia el nombre del componente
Container **getParent()**	Devuelve el contenedor que sostiene a este componente

métodos de apariencia y posición

método	uso
void **setVisible(**boolean vis**)**	Muestra u oculta el componente según el valor del argumento sea **true** o **false**
Color **getForeground()**	Devuelve el color de frente en forma de objeto Color
void **setForeGround(**Color color**)**	Cambia el color frontal
Color **getBackground()**	Devuelve el color de fondo en forma de objeto *java.awt.Color*
void **setBackground(**Color color**)**	Cambia el color de fondo
Point **getLocation()**	Devuelve la posición del componente en forma de objeto Point
void **setLocation(**int x, int y**)**	Coloca el componente en la posición x, y
void **setLocation(**Point p**)**	Coloca el componente en la posición marcada por las coordenadas del punto P
Dimension **getSize()**	Devuelve el tamaño del componente en un objeto de tipo *java.awt.Dimension*.
void **setSize(**Dimension d**)**	
void **setSize(**int ancho, int alto**)**	Cambia las dimensiones del objeto en base a un objeto *Dimension* o indicando la anchura y la altura con dos enteros.
void **setBounds(**int x, int y, int ancho, int alto**)**	Determina la posición de la ventana (en la coordenada x, y) así como su tamaño con los parámetros *ancho* y *alto*
void **setPreferredSize(**Dimension d**)**	Cambia el tamaño preferido del componente. Este tamaño es el que el componente realmente quiere tener.

Componentes

Las componentes Swing heredan de Jcomponent la siguiente funcionalidad:

- Bordes: usando el método setBorder() que desplegará la lacomponente
- Doublebuffering: mejora la apariencia de la componentes que sufren cambios contínuos. Todas las componentes Swing son Double-buffering
- Tooltips: usando el método setToolTipText() es posible proveerle de ayuda a usuario de la componente.
- Navegación por teclado: usando el método registerKeyBoardAction() se habilita al usuariorpara que navegue con el teclado en vez de con el mouse. También es posible usar abreviaturas (combinar caracteres con teclas) usando el objeto KeyStroke.
- L&F: cada programa Java tiene un objeto L&F que determina el look&feelde las componentes en ejecución. Es posible elegir el L&F de las componentes Swing invocando al método UIManager.setLookAndFeel()

Contenedores

- Son un tipo de componentes pensados para almacenar y manejar otros componentes. Los objetos JComponent pueden ser contenedores al ser una clase que desciende de Container que es la clase de los objetos contenedores de AWT.

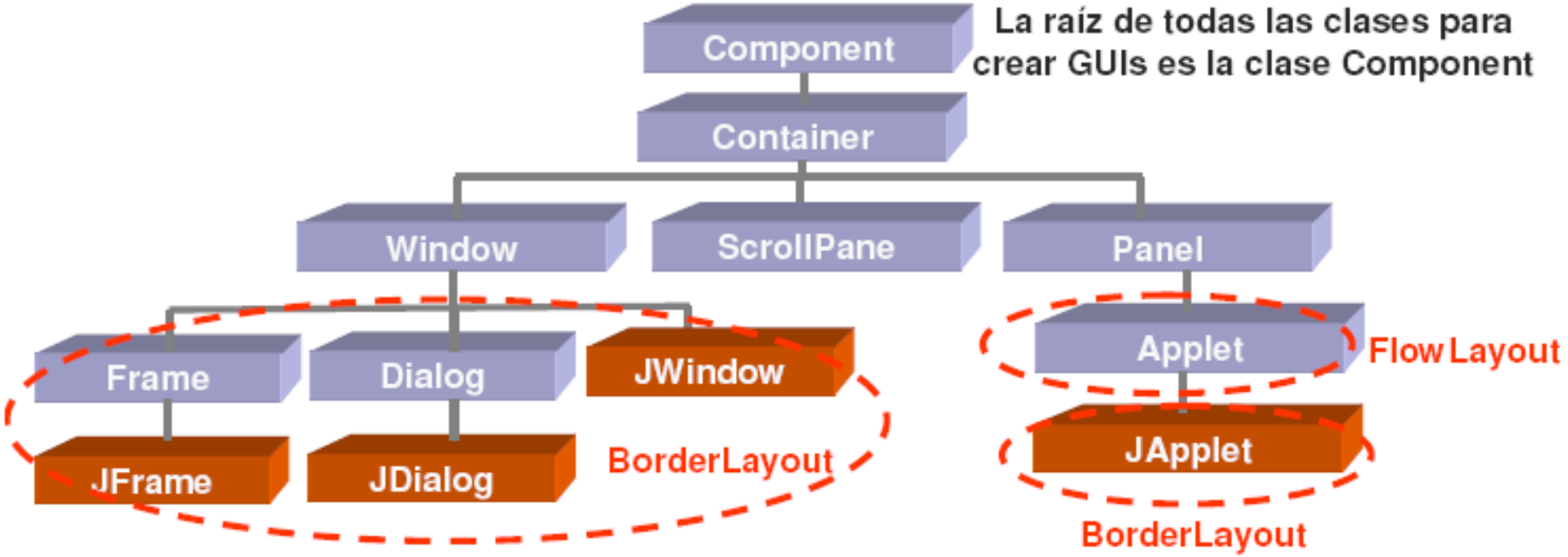
- Para hacer que un componente forme parte de un contenedor, se utiliza el método add. Mientras que el método remove es el encargado de eliminar un componente. Ambos métodos proceden de la clase java.awt.Container

Swing posee algunos contenedores especiales. Algunos son:

- **JWindow. Representa un panel de ventana sin bordes ni elementos visibles.**
- **JFrame. Objeto que representa una ventana típica con bordes, botones de cerrar, etc.**
- **JPanel. Es la clase utilizada como contenedor genérico para agrupar componentes.**
- **JDialog. Clase que genera un cuadro de diálogo.**
- **JApplet. Contenedor que agrupa componentes que serán mostrados en un navegador**

Jerarquía de contenedores

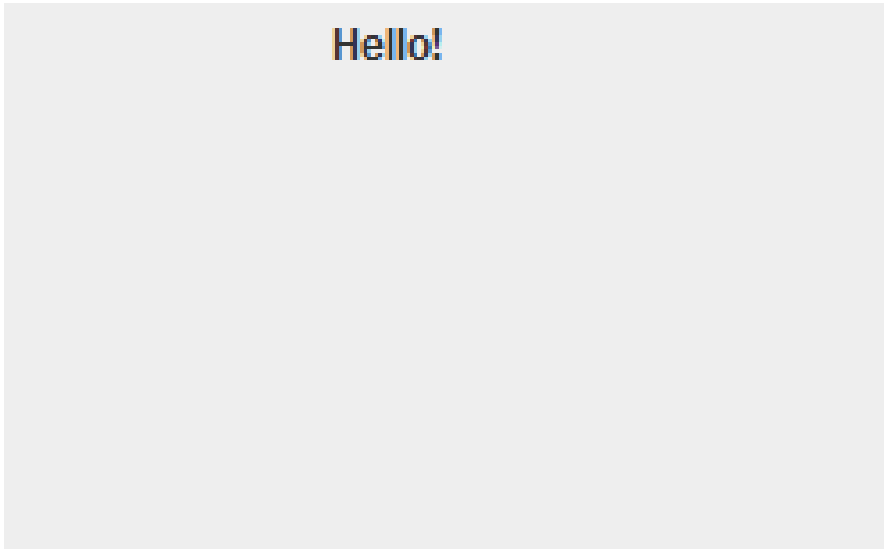
Los contenedores Swing a nivel de ventana, son subclases de los contendores AWT



Los Layouts son los mismos que en AWT con excepción de Applet que cambió de

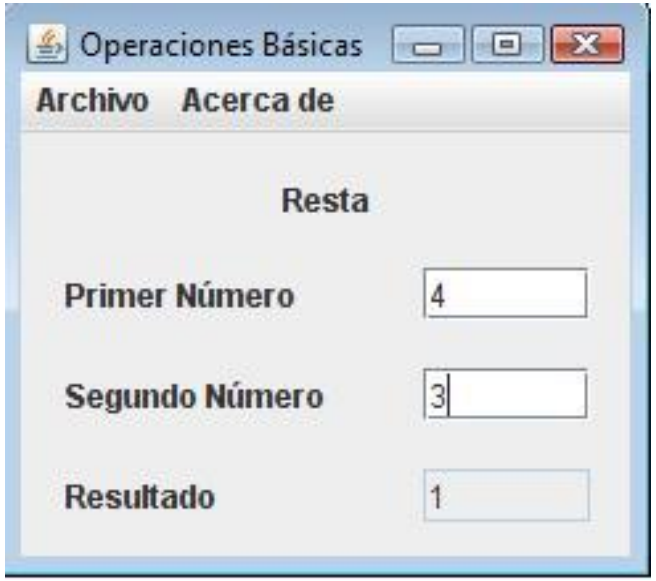
JWindow

Este objeto deriva de la clase java.awt.Window que a su vez deriva de java.awt.Container. Se trata de un objeto que representa un marco de ventana simple, sin borde, ni ningún elemento. Sin embargo son contenedores a los que se les puede añadir información. Estos componentes suelen estar dentro de una ventana de tipo Frame o, mejor, JFrame.



JFrame

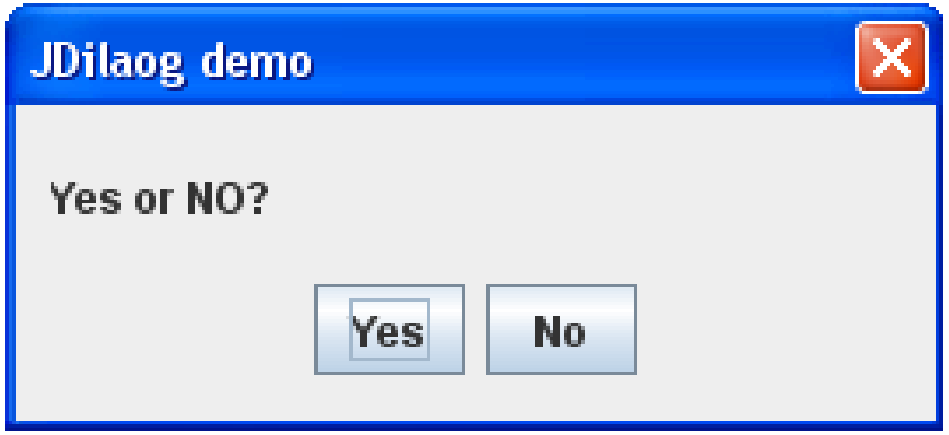
- Los objetos JFrame derivan de la clase Frame que, a su vez deriva, también de la clase Window, por lo que muchos métodos de esta clase son comunes a la anterior. Los objetos JFrame son ventanas completas.




```
Public static void main(Strings[] args)
{
        JFrameframe= new JFrame("IPOO");
        frame.setSize(50,50);
        frame.setVisible(true);
}
```

JDialog

- JDialog deriva de la clase AWT Dialog que es subclase de Window. Representa un cuadro de diálogo que es una ventana especializada para realizar operaciones complejas.



JApplet

- Se utiliza para aplicaciones que se ejecutan en un browser web.

Añadir componentes a las ventanas

- Las clases JDialog y JFrame no permiten usar el método add, como les ocurre a los contenedores normales, por eso se utiliza el método getContentPane() que devuelve un objeto Container que representa el área visible de la ventana. A este contenedor se le llama panel contenedor y sí permite método add.

```
public class prbVentana{
   public static void main(String args[]){
         JFrame ventana=new JFrame("Prueba");
         ventana.setLocation(100,100);
         Container c=ventana.getContentPane();
         c.add(new JLabel("Hola"));
         ventana.pack();
         ventana.setVisible(true);
   }
```

- Este código muestra una ventana ajustada al contenido de una ventana que pone Hola.

Eventos

- En términos de Java, un evento es un objeto que es lanzado por un objeto y enviado a otro objeto llamado *escuchador (listener)*. *Un evento se lanza (o se dispara, fire)* cuando ocurre una determinada situación (un clic de ratón, una pulsación de tecla,...).

- La programación de eventos es una de las bases de Java y permite mecanismos de diseño de programas orientados a las acciones del usuario. Es decir, son las acciones del usuario las que desencadenan mensajes entre los objetos (el flujo del código del programa se desvía en función del evento producido, alterando la ejecución normal).

- En su captura hay que tener en cuenta que hay tres objetos implicados:

Eventos

- **El objeto fuente. Que es el objeto que lanza los eventos**. Dependiendo del tipo de objeto que sea, puede lanzar unos métodos u otros. Por ejemplo un objeto de tipo JLabel (etiqueta) puede lanzar eventos de ratón (MouseEvent) pero no de teclado (KeyEvent). El hecho de que dispare esos eventos no significa que el programa tenga que, necesariamente, realizar una acción. Sólo se ejecuta una acción si hay un objeto escuchando.

- **El objeto escuchador u oyente (*listener*). *Se trata del objeto que recibe el evento producido***. Es el objeto que captura el evento y ejecuta el código correspondiente. Para ello debe implementar una interfaz relacionada con el tipo de evento que captura. Esa interfaz obligará a implementar uno o más métodos cuyo código es el que se ejecuta cuando se dispare el evento.

- **El objeto de evento. Se trata del objeto que es enviado desde el objeto fuente a el escuchador**. Según el tipo de evento que se haya producido se ejecutará uno u otro método en el escuchador.

Escuchadores de eventos. Listeners

- **Cada tipo de evento tiene asociado un interfaz para manejar el evento. A esos interfaces se les llama *escuchadores (Listeners) ya que proporcionan métodos que están a la* espera de que el evento se produzca. Cuando el evento es disparado por el objeto fuente al que se estaba escuchando, el método manejador del evento se dispara automáticamente.**

- **Por ejemplo, el método actionPerformed es el encargado de gestionar eventos del tipo ActionEvent (eventos de acción, se producen, por ejemplo, al hacer clic en un botón). Este método está implementado en la interfaz ActionListener (implementa escuchadores de eventos de acción).**

- **Cualquier clase que desee escuchar eventos (los suyos o los de otros objetos) debe implementar la interfaz (o interfaces) pensada para capturar los eventos del tipo deseado. Esta interfaz habilita a la clase para poder implementar métodos de gestión de eventos.**

- **Por ejemplo; un objeto que quiera escuchar eventos ActionEvent, debe implementar la interfaz ActionListener. Esa interfaz obliga a definir el método ya comentado actionPerformed. *El código de ese método será invocado* automáticamente cuando el objeto fuente produzca un evento de acción.**

Escuchadores de eventos. Listeners

- **Es decir, hay tres actores fundamentales en el escuchador de eventos:**

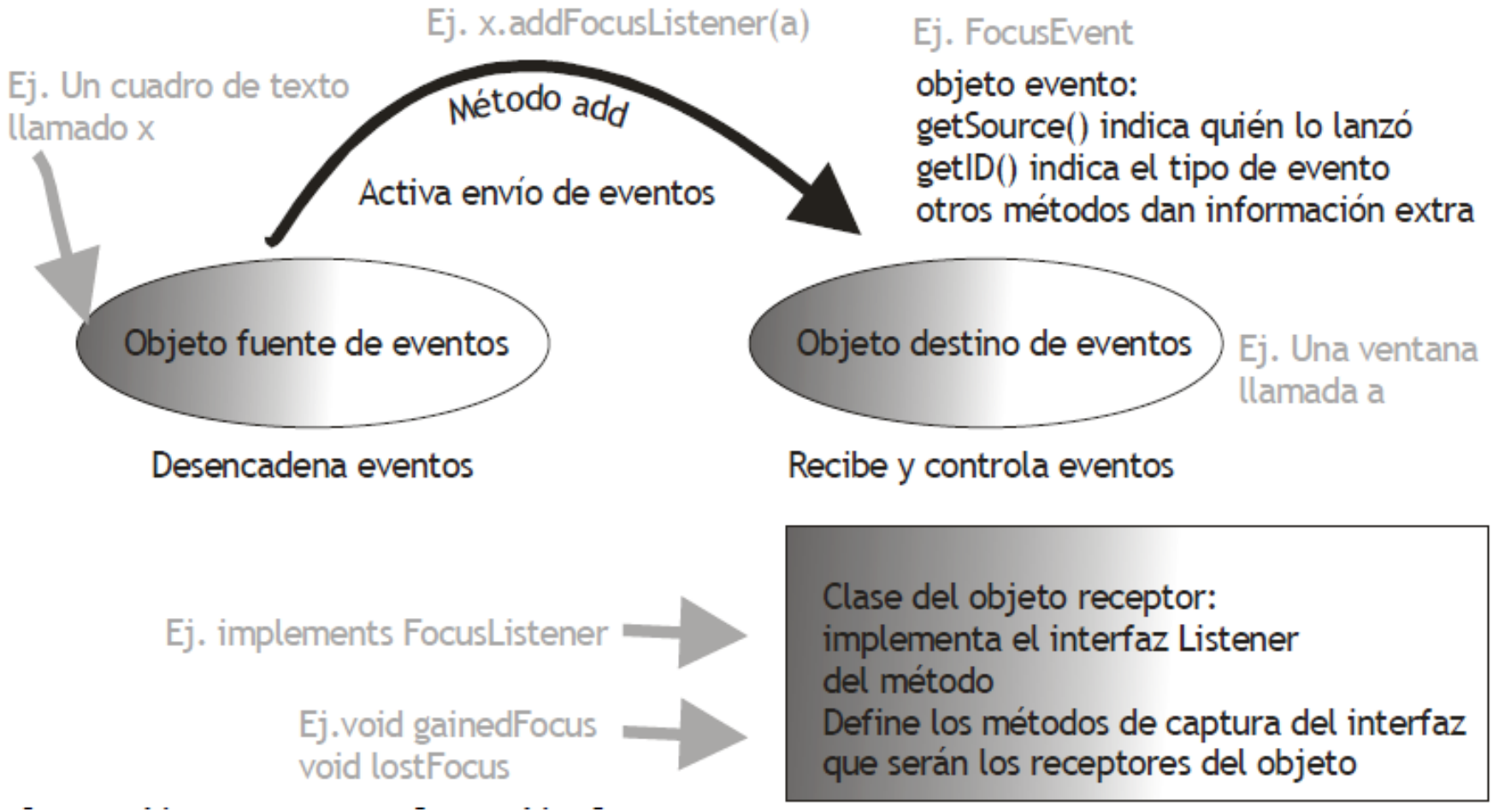
1) **El objeto de evento que se dispara cuando ocurre un suceso.** Por ejemplo para capturar el ratón sería MouseEvent.

2) **El método o métodos de captura del evento (que se lanza cuando el evento se produce).** Pueden ser varios, por ejemplo para la captura de eventos de tipo MouseEvent (evento de ratón) existen los métodos mouseReleased (es invocado cuando se libera un botón del ratón), mousePressed (es invocado cuando se pulsa un botón del ratón), mouseEntered (es invocado cuando el cursor entra en el objeto) y mouseExited (ocurre cuando el ratón sale del objeto).

3) **La interfaz que tiene que estar implementada en la clase que desea capturar ese evento.** En este ejemplo sería MouseListener, que es la que obliga a la clase del escuchador a implementar los cuatro métodos de gestión comentados anteriormente

Sin duda, el más complejo es este último, pero hay que entender que una interfaz lo único que consigue es dar a una clase la facultad de escuchar (*Listen) eventos*.

Proceso de gestión de eventos



Fuentes de eventos

Disparar eventos

- El objeto fuente permite que un objeto tenga capacidad de enviar eventos. Esto se consigue mediante un método que comienza por la palabra add seguida por el nombre de la interfaz que captura este tipo de eventos. Este método recibe como parámetro el objeto escuchador de los eventos.

- Esto es más fácil de lo que parece. Para que un objeto fuente, sea escuchado, hay que indicar quién será el objeto que escuche (que obligadamente deberá implementar la interfaz relacionada con el evento a escuchar). Cualquier componente puede lanzar eventos, sólo hay que indicárselo, y eso es lo que hace el método add.

Fuentes de eventos. Ejemplo

```
public class MiVentana extends JFrame implements ActionListener{
   JButton boton1=new JButton("Prueba");
   //Constructor
   public MiVentana() {
         ...
         boton1.addActionListener(this);//El botón lanza
               //eventos que son capturados por la ventana
         ...
   }
   ...
   public void actionPerformed(ActionEvent e){
         //Manejo del evento
   }
}
```

Fuentes de eventos

Eliminar oyentes

- Hay un método **remove que sirve para que un oyente del objeto deje de escuchar los** eventos.

```
boton1.removeActionListener(this); //La ventana deja de
                                   //escuchar los eventos del botón
```

Objeto de evento.

- Ya se ha comentado que cuando se produce un evento se crea un objeto llamado **objeto de evento. Este objeto es pasado al objeto que está *escuchando los eventos*.**

- Todos los objetos de evento pertenecen a clases que derivan de **EventObject .**

- **Esta** es la superclase de todos los objetos de evento. Representa un evento genérico y en la práctica sólo sirve para definir los métodos comunes a todos los eventos

Objeto de evento.

- Para reducir la cantidad de objetos eventos creados por el modelo, los modelos disparan eventos especiales del tipo *javax.swing.event.ChangeEvent, que contiene solo la fuente del evento.*
- De esta manera el modelo puede reusar el mismo objeto evento para todas las notificaciones de cambio. Son notificaciones lightweigth.
- Las notificaciones lightweigth son usadas para las propiedades del modelo que se modifican frecuentemente.
- Los listeners que escuchan notificaciones lightweigth, implementan la interface ChangeListener.

Adaptadores

- **Para facilitar la gestión de eventos en ciertos casos, Java dispone de las llamadas clases adaptadores. Gracias a ellas, en muchos casos se evita tener que crear clases sólo para escuchar eventos. Estas clases son clases de contenido vacío pero que son muy interesantes para capturas sencillas de eventos.**

- **Todas poseen la palabra adapter en el nombre de clase. Por ejemplo esta es la definición de la clase MouseAdapter:**

```
public abstract class MouseAdapter implements MouseListener
{
    public void mouseClicked(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}
```

Adaptadores

Es una clase que implementa el interfaz MouseListener, pero que no define lo que hace cada método de captura. Eso se suele indicar de manera dinámica:

```
JFrame ventana =new JFrame("prueba");
ventana.setLocation(100,100);
ventana.setSize(300,300);
ventana.setVisible(true);
ventana.addMouseListener(new MouseAdapter(){
            public void mouseClicked(MouseEvent e){
                System.out.println("Hola");
            }});
```

En el ejemplo anterior al hacer clic en la ventana se escribe el mensaje *Hola en la* pantalla.
No ha hecho falta crear una clase para escuchar los eventos. Se la crea de forma dinámica y se la define en ese mismo momento. La única función del adaptador
es capturar los eventos deseados.

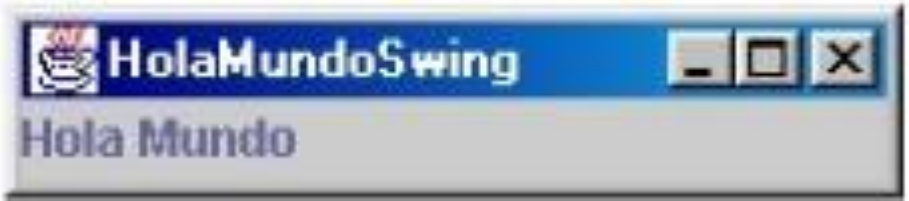
Ejemplo

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

public class HolaMundoSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("HolaMundoSwing");
        JLabel label = new JLabel("Hola Mundo");
        frame.getContentPane().add(label);
        frame.addWindowListener(new WindowAdapter(){
           public void windowClosing(WindowEvent e){
             System.exit(0);
           }
        }
        );

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

El resultado de ese famoso código es esta ventana:



La siguiente tabla muestra algunas de las Interfaces Oyentes con sus correspondientes Clases Adaptadoras y sus métodos en el modelo de eventos de los componentes AWT.

Interfaces Oyentes	Clases Adaptadoras	Métodos de las Interfaces Oyentes
ActionListener	ninguna	actionPerformed(ActionEvent evt)
AdjustmentListener	ninguna	adjustmentValueChanged(AdjustmentEvent evt)
FocusListener	**FocusAdapter**	focusGained(FocusEvent evt) focusLost(FocusEvent evt)
KeyListener	**KeyAdapter**	keyPressed(KeyEvent evt) keyReleased(KeyEvent evt) keyTyped(KeyEvent evt)
MouseListener	**MouseAdapter**	mouseClicked(MouseEvent evt) mouseEntered(MouseEvent evt) mouseExited(MouseEvent evt) mousePressed(MouseEvent evt) mouseReleased(MouseEvent evt)
TextListener	ninguna	textValueChanged(TextEvent evt)
WindowListener	**WindowAdapter**	windowActivated(WindowEvent evt) windowClosed(WindowEvent evt) windowClosing(WindowEvent evt) windowDeactivated(WindowEvent evt) windowDeiconified(WindowEvent evt) windowIconified(WindowEvent evt) windowOpened(WindowEvent evt)

	Evento	Listener	Métodos
Eventos de bajo nivel	ComponentEvent	ComponentListener	componentHidden componentMoved componentResized componentShown
	ContainerEvent	ContainerListener	componentAdded componentRemoved
	FocusEvent	FocusListener	focusGained focusLost
	KeyEvent	KeyListener	keyPressed keyReleased keyTyped
	MouseEvent	MouseListener	mouseClicked mouseEntered mouseExited mousePressed mouseReleased
	MouseMotionEvent	MouseMotionListener	mouseDragged mouseMoved
	WindowEvent	WindowListener	windowActivated windowClosed windowClosing windowDeactivated windowDeiconified windowIconofied windowOpened
Eventos semánticos	ActionEvent	ActionListener	actionPerformed
	AdjustmentEvent	AdjustmentListener	adjustmentValueChanged
	ItemEvent	ItemListener	itemStateChanged
	TextEvent	TextListener	textValueChanged

Eventos en los componentes.

Cómo Implementar un Manejador de Eventos

Todo manejador de eventos requiere tres partes de código.

1. Donde se declara la clase del manejador de eventos, el código implementa un interface de oyente. Por ejemplo.

```
public class MiClase implements ActionListener {
```

2. El código que registra un ejemplar de la clase de manejo de eventos de un oyente sobre uno o más componentes. Por ejemplo.

```
elComponente.addActionListener(intancia de MiClase);
```

3. La implementación de los métodos del interface oyente. Por ejemplo.

```
public void actionPerformed(ActionEvent e) { .../// código que reacciona a la acción ... }
```

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


public class ejemploOyente extends Applet implements ActionListener {
  protected Button botonl;
  protected Label labell;
  protected int contador;


  public void init() {
    botonl = new Button ( "click" );
    botonl.addActionListener( this );
    add( botonl );
    labell = new Label( "clicks: 0");
    add(labell);
  }


    // actionPerformed es el método a que obliga la interfaz.

  public void actionPerformed( ActionEvent e ) {
    Object generador = e.getSource();
    if (generador == botonl) {
      contador++;
      labell.setText( "clicks: "+contador );
    }
  }
}
```

Mensajes hacia el usuario. Clase JOptionPane

- Una de las labores típicas en la creación de aplicaciones gráficas del tipo que sea, es la de comunicarse con el usuario a través de mensajes en forma de cuadro de diálogo.

- Algunos cuadros son extremadamente utilizados por su sencillez (textos de aviso, error, confirmación, entrada sencilla de datos, etc.).

- La clase **JOptionPane deriva de JComponent y es la encargada de crear este tipo** de cuadros. Aunque posee constructores, normalmente se utilizan mucho más una serie de métodos estáticos que permiten crear de forma más sencilla objetos *JOptionPane*.

Cuadros de información

- Son cuadros de diálogo que sirven para informar al usuario de un determinado hecho. Se construyen utilizando los siguientes métodos estáticos:

métodos	uso
static void **showMessageDialog(** Component padre, Object mensaje**)**	Muestra un cuadro de diálogo en el contenedor *padre* indicado con un determinado mensaje
static void **showMessageDialog(** Component padre, Object mensaje, String título, int tipo**)**	Muestra un cuadro de diálogo en el contenedor *padre* indicado con un determinado mensaje, título y tipo.
static void **showMessageDialog(** Component padre, Object mensaje, String título, int tipo, Icon i**)**	Igual que el anterior pero se permite indicar un icono para acompañar el mensaje

Estos son los posibles creadores de este tipo de cuadro. El tipo puede ser una de estas constantes:

- JOptionPane.**INFORMATION_MESSAGE.**
- JOptionPane.**ERROR_MESSAGE.**
- JOptionPane.**WARNING_MESSAGE.** Aviso
- JOptionPane.**QUESTION_MESSAGE.** Pregunta
- JOptionPane.**PLAIN_MESSAGE.** Sin icono

Ejemplo

- JOptionPane.showMessageDialog(**this, "Soy un mensaje normal"**, "Cuadro 1", JOptionPane.INFORMATION_MESSAGE);

El resultado es:

Cuadros de confirmación

- La diferencia con los anteriores reside en que en estos hay que capturar la respuesta del usuario para comprobar si acepta o declina el mensaje. Los métodos estáticos de creación son:

métodos	uso
static int **showConfirmDialog(** Component padre, Object mensaje)	Muestra un cuadro de confirmación en el componente *padre* con el mensaje indicado y botones de **Sí**, **No** y **Cancelar**
static int **showConfirmDialog(** Component padre, Object mensaje, **String** título, int opciones)	Muestra cuadro de confirmación con el título y mensaje reseñados y las opciones indicadas (las opciones se describen al final)
static int **showConfirmDialog(** Component padre, Object mensaje, String título, int opciones, int tipo)	Como el anterior pero indicando el tipo de cuadro (los posibles valores son los indicados en la página anterior) y un icono
static int **showConfirmDialog(** Component padre, Object mensaje, String título, int opciones, int tipo, Icon icono)	Como el anterior pero indicando un icono.

Cuadros de confirmación

- Obsérvese como el tipo de retorno es un número entero; este número representa el botón del cuadro sobre el que el usuario hizo clic. Este valor se puede representar por medio de estas constantes de JOptionPane

- **JOptionPane.NO_OPTION. El usuario no pulsó ningún botón en el cuadro**
- **JOptionPane.CLOSE_OPTION. El usuario cerró sin elegir nada**
- **JOptionPane.OK_OPTION. El usuario pulsó OK**
- **JOptionPane.YES_OPTION. El usuario pulsó el botón Sí**
- **JOptionPane.CANCEL_OPTION. El usuario pulsó el botón Cancelar**

- Estas otras constantes facilitan el uso del parámetro *opciones que sirve para modificar* la funcionalidad del cuadro. Son:

- **JOptionPane.OK_CANCEL_OPTION. Cuadro con los botones OK y Cancelar**
- **JOptionPane.YES_NO_OPTION. Cuadro con botones Sí y No**

Ejemplo

```
int res= JOptionPane.showConfirmDialog(null,
   "¿Desea salir?", "Salir",
   JOptionPane.YES_NO_OPTION,JOptionPane.QUESTION_MESSAGE);


if (res==JOptionPane.YES_OPTION) System.exit(0);
```

Cuadros de diálogo para rellenar entradas

- Son cuadros que permiten que el usuario, desde un mensaje de sistema, rellene una determinada variable.

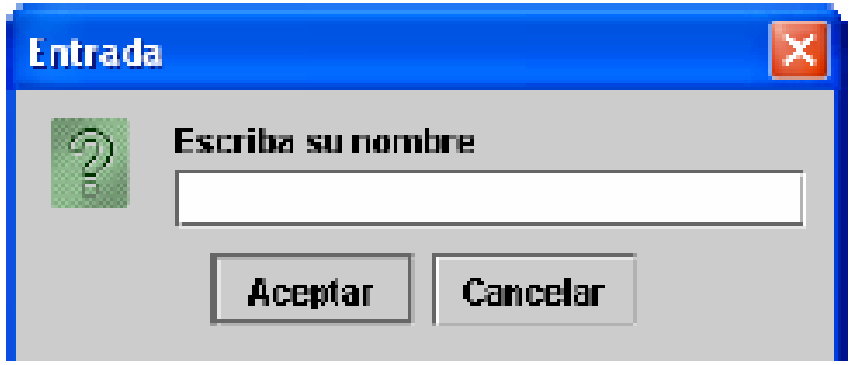
métodos	uso
static String **showInputDialog(**Object mensaje**)**	Muestra un cuadro de entrada con el mensaje indicado
static String **showInputDialog(** Component padre, Object mensaje**)**	Muestra un cuadro de entrada en el componente *padre* con el mensaje indicado
static **String showInputDialog(** Component padre, Object mensaje, String título, int tipo**)**	Muestra cuadro de entrada con el título y mensaje reseñados y el tipo que se indica
static Object **showInputDialog(** Component padre, Object mensaje, String título, int tipo, Icono icono, Object[] selección, Object selecciónInicial**)**	Indica además un icono, selecciones posibles y la selección inicial. El valor devuelto es un objeto **Object**.

Todos los métodos devuelven un String en el que se almacena la respuesta del usuario. En caso de que el usuario cancele el cuadro, devuelve **null** en la cadena a examinar.

Ejemplo

```
String  res= JOptionPane.showInputDialog("Escriba su nombre");
if (res==null) JOptionPane.showMessageDialog(this,"No
escribió"); else nombre=res;
```

El resultado es:



Cuadros de diálogo internos

- Son cuadros que están dentro de un contenedor y no pueden salir fuera de él. Son más ligeros (ocupan menos recursos). Se crean con los mismos métodos y posibilidades que los anteriores, sólo que incorporan la palabra Internal. Funciones de creación:

- **showInternalMessageDialog**. Crea un mensaje normal, pero interno.
  Los parámetros son los mismos que showMessageDialog.

- **showInternalInputDialog**. Crea un mensaje interno de entrada de datos. Los parámetros son los mismos que showInputDialog.

- **showInternalConfirmDialog**. Crea un mensaje interno de confirmación.
  Los parámetros son los mismos que showConfirmDialog.