



UNIDAD 8

APLICACIONES CONTROLADAS POR EVENTOS

1. INTRODUCCIÓN.
2. PRIMEROS PASOS.
 - 2.1. Frames Y Paneles.
 - 2.2. Otros Elementos.
 - 2.3. Dibujar Sobre los Componentes.
3. EVENTOS.
 - 3.1. Eventos de Ratón.
 - 3.2. Eventos Timer.
 - 3.3. Eventos de Teclado.
 - 3.4. Eventos de Componentes.
 - 3.5. Eventos de Foco.
4. ALGUNOS COMPONENTES DE LA GUI.
 - 4.1. Componentes Básicos.
 - 4.2. Layouts Manager.
 - 4.3. Paneles.
 - 4.4. Listas, Combos, Spinners y Sliders.
 - 4.5. Menús, Acciones y Barras de Herramientas.
 - 4.6. Diálogos Predefinidos.
5. IMÁGENES Y RECURSOS.
6. DISEÑO DE INTERFACES DESDE EL IDE.
7. EJERCICIOS.

BIBLIOGRAFÍA:

- Java, Cómo Programar (10ª Ed.) Pearson. Paul y Harvey Deitel (2016).
- Introduction to Programming Using Java (7ª Ed.). David J. Eck (2014)
- Java Swing (2º Ed.). Matthew Robinson. (2007)
- Referencia de Java SE7.



UNIDAD 8. Aplicaciones Controladas por Eventos.

- Guía del plugin de eclipse Windows Builder.

8.1. INTRODUCCIÓN.

Los programas GUI se diferencian de los tradicionales en que son **conducidos por eventos**: hay sucesos del mundo, como las acciones de un usuario que presiona un botón o una tecla, que generan **eventos** y el programa debe dar respuesta estos eventos. En un programa ya no hay certeza de qué sentencias se ejecutarán primero ni en qué orden, depende de los eventos que aparezcan. Hacer un programa, ahora consiste además en pensar como hacer métodos que respondan a los eventos que te interesen.

Cada elemento de la interfaz GUI e incluso los eventos son objetos, un color es un objeto, un tipo de letra es un objeto, etc. En Java programar una aplicación GUI es programación orientada a objetos.

El toolkit original usado en Java fué **AWT** (**Abstract Windowing Toolkit**) que aportaba una interfaz común de componentes GUI para varios sistemas operativos. Usaba un modelo de eventos muy simple que **no necesitaba objetos listener**. Pero este modelo se abandonó pronto en favor del toolkit **Swing** en Java 1.2 como una alternativa mejorada de AWT, una gran variedad de componentes gráficos y mejor estructura lógica.

Nota: En el IDE Eclipse debes incluir **requires java.desktop;** en el fichero module-info.java de un proyecto.

Hace poco se introdujo otro toolkit llamado **JavaFX** que es similar a Swing pero usa un conjunto de clases diferentes. Con Java 8, JavaFX se ha vuelto la opción preferida para realizar aplicaciones GUI. JavaFX es compatible con Swing y puede usar sus componentes. Nosotros aprenderemos a usar Swing, aunque aún se utilizan algunos elementos



UNIDAD 8. Aplicaciones Controladas por Eventos.

de AWT. El salto a JavaFX es muy sencillo.

8.2 PRIMEROS PASOS.

8.2.1. FRAMES Y PANELES.

Hay varias formas de utilizar GUI. La más básica sería usar diálogos (ventanas modales o no, predefinidas en Java).

EJEMPLO 1: un programa que muestra un diálogo.

```
import javax.swing.JOptionPane;

public class Ejemplo1 {

    public static void main(String[] args){
        // Primer argumento es la ventana padre, null el sistema
        JOptionPane.showMessageDialog(null, "Hola Mundo!");
    }
}
```

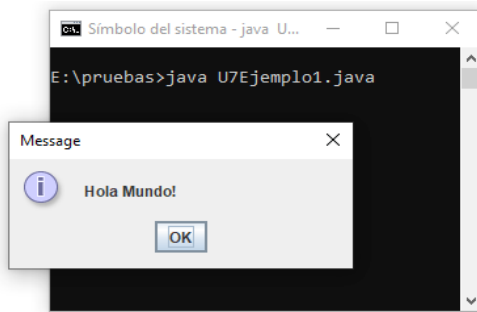


Figura 1: Diálogo mostrado desde programa de consola.

Si queremos una aplicación GUI con nuestra propia ventana, debemos definirla. Una ventana de alto nivel (top-level), es decir, que **no está contenida dentro de otra**, en Java se llama **frame** (marco). La librería AWT tiene una clase llamada **Frame**, para este elemento. La versión de esta clase de Swing se llama **JFrame** y extiende a la clase **Frame**. Es una de las pocas clases donde Swing no dibuja nada (los componentes



UNIDAD 8. Aplicaciones Controladas por Eventos.

no los dibuja en un canvas), es el sistema de ventanas del usuario el que se encarga de dibujar las decoraciones (barra de título, sus botones de maximizar, minimizar y cerrar, icono, etc.)

Nota: cuidado!! La mayoría de clases de Swing comienzan por "J" (JButton, JFrame, etc.). AWT tiene clases equivalentes (Button, Frame, etc.). Ten cuidado en no confundirlas.

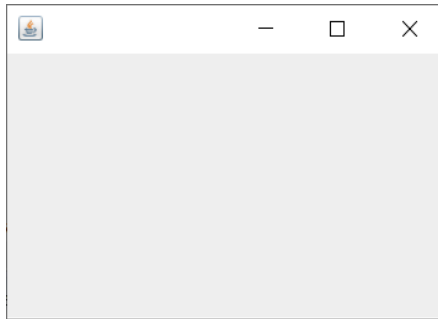


Figura 2: SimpleFrame derivado de JFrame.

Las clases de Swing están en el paquete **javax.swing**. Cuando aparece **javax** significa que es un paquete de extensión, no un paquete **core**.

Por defecto un frame tiene un tamaño de 0x0 pixels. Para tener nuestra propia ventana creamos una subclase de JFrame, en el ejemplo la llamamos SimpleFrame cuyo constructor fija un tamaño de 300 x 200 pixels. Esta es la única diferencia con un JFrame. En el método **main()** de la clase U7Ejemplo2 se construye un objeto de la clase SimpleFrame y lo hacemos visible.

EJEMPLO 2: Crear nuestra propia ventana de alto nivel.

```
// Fichero U7Ejemplo2.java
package simpleFrame; // Comenta la línea para ejecutar desde consola

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



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
public class U7Ejemplo2 {

    public static void main( String[] args) {
        SimpleFrame f = new SimpleFrame();
        f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        f.setVisible(true);
    }
}

class SimpleFrame extends JFrame {
    private static final int ANCHO = 300;
    private static final int ALTO = 200;

    public SimpleFrame() { // constructor
        setSize(ANCHO, ALTO);
    }
}
```

Luego, el programa define lo que pasará cuando el usuario cierre el frame. En este caso queremos que el programa acabe. Esto lo hacemos en la sentencia:

```
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

Si tu programa tiene varios frames abiertos a la vez, quizás no quieras este comportamiento. Por defecto, cuando el usuario cierra un frame, este se oculta, pero el programa no termina. También puedes cerrar el frame y que el programa siga funcionando.

Construir un frame no significa que se vea, comienza su vida siendo invisible, esto permite al programador añadir componentes y retocarlo antes de que sea visible. Para hacerlo visible ejecutamos:

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

Hay dos cuestiones técnicas que deben resolverse en cada programa Swing. La primera es que **todos los componentes deben configurarse para que un thread les envíe o despache (dispatch) eventos**, es el thread de control el que pasa los eventos a los componentes de la



UNIDAD 8. Aplicaciones Controladas por Eventos.

interfaz. Aunque el ejemplo 2 no crea un thread separado para la parte gráfica, es decir, no inicializa la GUI en el **event dispatch thread** sino en el thread principal. Es aceptable, pero esta aproximación es menos eficiente y más propensa a errores. La probabilidad de que ocurra un error es baja, pero es preferible hacer las cosas bien, aunque el código parezca más complicado. En el ejemplo 3 hacemos la GUI en otro thread.

EJEMPLO 3: Crear ventana de alto nivel propia en thread separado.

```
// Fichero U7Ejemplo3.java
package simpleFrame; // Comenta la línea para ejecutar desde consola

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

public class U7Ejemplo3 {
    public static void main( String[] args) {
        EventQueue.invokeLater(
            new Runnable() {
                public void run() {
                    SimpleFrame f= new SimpleFrame();
                    f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
                    f.setVisible(true);
                } // método run de Runnable
            } // clase y Objeto Runnable, es una clase anónima!!
        ); // llamada a EventQueue.invokeLater()
    }
}

class SimpleFrame extends JFrame {
    private static final int ANCHO = 300;
    private static final int ALTO = 200;

    public SimpleFrame() { // constructor
        setSize( ANCHO, ALTO );
    }
}
```

El código que ejecuta sentencias dentro del **event dispatch thread** es:

```
EventQueue.invokeLater(
    new Runnable() { public void run(){ /* sentencias... */ } }
);
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

Después de ejecutar las sentencias de inicialización, el método `main()` acaba. Observa que **si tienes otro thread, aunque se acabe `main()`, el programa no finaliza, solo finaliza el thread principal**. El event `dispatch thread` continúa ejecutándose hasta que se termine, o bien se cierre el frame, o bien se llame al método **`System.exit(int)`**.

Como puedes ver en la figura, la barra de título y las decoraciones de los bordes, las esquinas de cambio de tamaño, etc. no las dibuja Swing, por tanto no tendrán la misma apariencia en Windows, en GTK o en MacOS. Swing dibuja todo lo que hay dentro del frame, que en este programa solo es poner un color de relleno por defecto.

Nota: Puedes eliminar las decoraciones del frame ejecutando el método **`frame.setUndecorated(true)`**; Te animo a probarlo. ;-)

El contenido que se muestra en un `JFrame` se llama su **content pane**. (Además de su content pane, un `JFrame` también puede tener una barra de menús). Un `JFrame` básico ya tiene un content pane vacío, en el que puedes añadir otros componentes. Incluso puedes cambiar el content pane que tiene:

`window.setContentPane(nuevo_content);`

Un `JFrame` es un **componente contenedor**, **sirve para contener a otros**. Todos los objetos asociados (contenidos) en un `JFrame` son gestionados por su único hijo, una instancia de **`JRootPane`** (un simple contenedor). Cada `JFrame` contiene un `JRootPane`, que es accesible (lo podemos obtener) a través del método **`getRootPane()`**. La figura 2 muestra la jerarquía de un **`JFrame`** y su **`JRootPane`** (las líneas de la figura indican la relación "tiene un").

Cada **`JRootPane`** contiene varios componentes identificados por su



UNIDAD 8. Aplicaciones Controladas por Eventos.

nombre: `glassPane` (por defecto un `JPanel`), `layeredPane` (un `JLayeredPane`), `contentPane` (por defecto un `JPanel`) y `menuBar` (un `JMenuBar`).

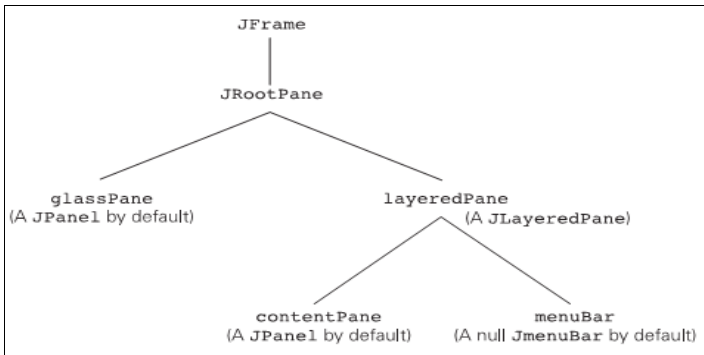


Figura 3: Diagrama de qué contiene a qué.

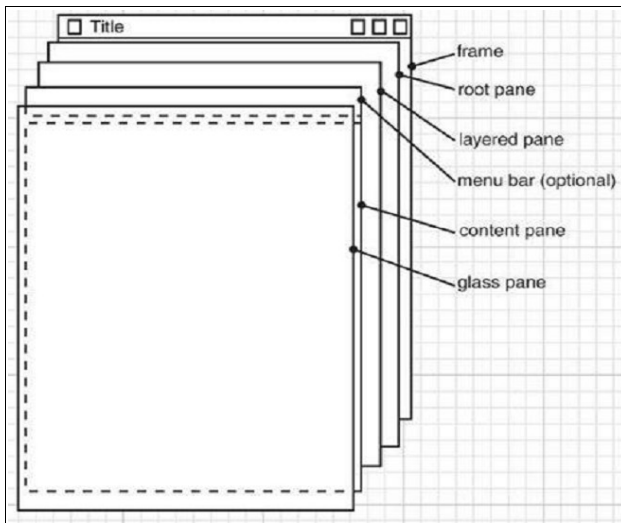


Figura 4: Distribución de los Pane del JRootPane de un JFrame.

El `glassPane` se inicializa a un `JPanel` no opaco (transparente) que se sitúa encima del **`JLayeredPane`** como se aprecia en la figura 4. Este componente se usa mucho cuando necesitas interceptar eventos del



UNIDAD 8. Aplicaciones Controladas por Eventos.

ratón para mostrar un cursor sobre el frame o redirigir el foco de la aplicación. Aunque el glassPane puede ser cualquier componente, por defecto es un JPanel. Puedes cambiarlo con `setGlassPane(componente)`.

MÉTODOS Y PROPIEDADES DE UN JFRAME

Si consultas las operaciones que podemos pedir a un JFrame, verás que no son muchas. Pero hay algo que debes tener en cuenta cuando programes en lenguajes orientados a objetos, usando clases que tienen relaciones de herencia: muchos de los métodos y propiedades los heredan y no aparecerá en su documentación si la consultas. Por eso es muy importante conocer la jerarquía de clases que tiene. Vamos a ver qué cosas se le pueden pedir (a él de forma directa o a sus ancestros).

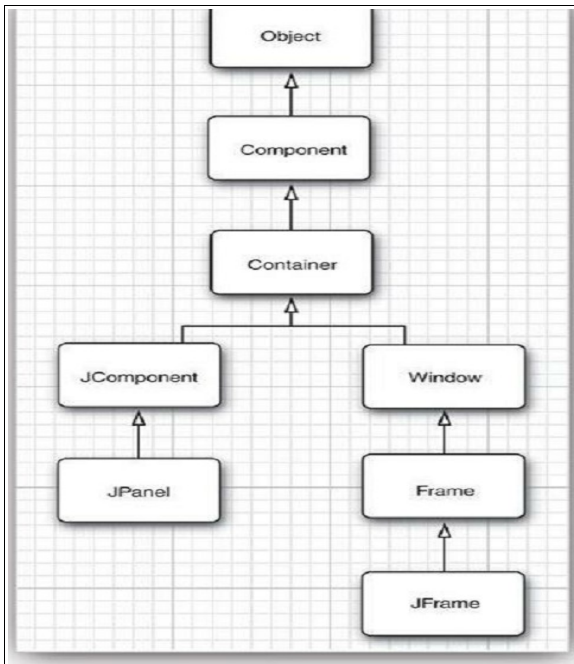


Figura 5: Jerarquía de clases de JFrame y JPanel.



UNIDAD 8. Aplicaciones Controladas por Eventos.

POSICIÓN Y TAMAÑO.

Para cambiar la posición donde aparece cualquier componente, puedes ejecutar el método:

```
setLocation( x, y );
```

que hará que su esquina superior izquierda se posicione en esas coordenadas. La coordenada (0,0) es la esquina izquierda de la pantalla.

La clase **Component** tiene otro método que permite además de posicionar, cambiar las dimensiones:

```
setBounds( x, y, ancho, alto);
```

Alternativamente, puedes dar al sistema gestor de ventanas del sistema operativo, la capacidad de elegir donde aparece la ventana ejecutando esta llamada antes de hacer visible la ventana:

```
setLocationByPlatform(true);
```

***Nota:** Para un frame, las coordenadas se refieren a la pantalla. Para otros componentes se refieren al componente que los aloja.*

DIMENSIONES DE LA PANTALLA

A veces es interesante que el programa pueda averiguar las dimensiones de la pantalla donde se ejecuta, para que pueda tomar decisiones sobre tamaños y posiciones de sus frames. Si haces una llamada al método estático **getDefaultToolkit()** de la clase **Toolkit** obtienes un objeto **Toolkit** (es un intermediario para interactuar con el sistema gestor de ventanas).

A este objeto le puedes pedir ejecutar el método **getScreenSize()** que te devuelve el tamaño de la pantalla en un objeto **Dimension**. Ejemplo:



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
Toolkit kit = Toolkit.getDefaultToolkit();  
Dimension pantalla = kit.getScreenSize();  
int ancho = pantalla.width;  
int alto = pantalla.height;
```

Nota: si escribes un programa que se va a ejecutar en varias pantallas, con varios tamaños y resoluciones, utiliza las clases **GraphicsEnvironment** y **GraphicsDevice** (esta te permite además ejecutar el programa en modo pantalla completa).

java.awt.Component 1.0

- **isVisible()** y **setVisible(boolean b)**: salvo los de alto nivel (como JFrame), por defecto suelen ser visibles.
- **setSize(int width, int height)** 1.1: cambia dimensiones.
- **setLocation(int x, int y)** 1.1: mueve el componente a las coordenadas (x,y) del contenedor o de la pantalla (top-level).
- **setBounds(int x, int y, int width, int height)** 1.1 :mueve y redimensiona.
- **Dimension getSize()** y **setSize(Dimension)** 1.1

java.awt.Window

- **toFront()**: muestra la ventana delante de cualquier otra.
- **toBack()**: mueve la ventana al fondo de la pila de ventanas del escritorio.
- **isLocationByPlatform()** y **setLocationByPlatform(boolean)** si el gestor de ventanas es quien elige la posición donde aparece.

java.awt.Frame

- **isResizable()** y **setResizable(boolean)**: Si la propiedad es true, el usuario puede redimensionar el Frame.
- **String getTitle()** y **setTitle(String)**: el texto que aparece en la barra de títulos del Frame.
- **Image getIconImage()** y **setIconImage(Image)**: el icono que



UNIDAD 8. Aplicaciones Controladas por Eventos.

aparece en el frame.

- **isUndecorated()** y **setUndecorated(boolean)** si es true, el frame se muestra sin decoraciones. Debe llamarse antes de mostrar el frame.
- **getExtendedState()** y **setExtendedState(int)** el estado extendido puede ser alguna de las siguientes constantes estáticas de Form: `NORMAL`, `ICONIFIED`, `MAXIMIZED_HORIZ`, `MAXIMIZED_VERT` y `MAXIMIZED_BOTH`.

java.awt.Toolkit

- **static Toolkit getDefaultToolkit()**: devuelve el toolkit.
- **Dimension getScreenSize()**: dimensiones de pantalla.

javax.swing.ImageIcon 1.2

- **ImageIcon(String filename)**: construye un icono a partir de una imagen almacenada en un fichero.
- **Image getImage()**: obtiene la imagen del icono.

EJERCICIO 1: ¿Qué hacen estas sentencias?

```
JFrame window = new JFrame("GUI Test");  
window.setContentPane(content);  
window.setSize(250, 100);  
window.setLocation(100, 100);  
window.setVisible(true);
```

MOSTRAR INFORMACIÓN DENTRO DE UN COMPONENTE

Si quieres mostrar un mensaje de texto en la ventana, dibujar directamente sobre el frame no es una buena idea. No está pensado para usarlo de esta forma. Ya has visto que es un contenedor complejo que tiene otros contenedores integrados. Así que dibujas en otros elementos que añades al frame.



UNIDAD 8. Aplicaciones Controladas por Eventos.

Lo que nos interesa ahora del frame es su **content pane**. Añades componentes al content pane de un frame f con sentencias como esta:

```
// A partir de Java 5 estas sentencias equivalen a: f.add(c);
Container cP = f.getContentPane();
Component c = ...; // Creas un componente o usas uno existente
cP.add(c);
```

Hasta Java SE 1.4, el método **add()** de JFrame estaba definido para lanzar una excepción con el mensaje: “Do not use JFrame.add(). Use JFrame.getContentPane().add() instead”. Pero a partir de Java SE 5.0, el método JFrame.add() en vez de reeducar a los programadores, pasa el componente a su content pane, así que ahora puedes usar: **f.add(c);**

Como ahora solo queremos un sencillo componente donde dibujar un texto, definimos una clase que extienda a **JComponent** y sobrescribimos su método **paintComponent(Graphics g)**.

El parámetro **Graphics** de paintComponent(g) es un objeto que recuerda una configuración de como dibujar (tipo de letra, color, etc.).

Nota: *Graphics es similar a un device context de la API Win32 de Windows o un graphics context de X11.*

```
class MiComponente extends JComponent {
    public void paintComponent(Graphics g) {
        // sentencias para dibujar...
    }
}
```

Cada vez que una ventana necesita redibujarse, no importa el motivo, el manejador de eventos de la aplicación se lo notifica al componente. Esto causa que los métodos **paintComponent()** de todos los componentes se ejecuten.

Nunca llames desde un programa a un método paintComponent(). Lo



UNIDAD 8. Aplicaciones Controladas por Eventos.

llamarán automáticamente cuando sea necesario (cambio en las dimensiones de la ventana, maximizar, minimizar, un menú emergente lo ha tapado, se solapa con otra ventana, etc.)

Nota: Si necesitas forzar el repintado de la pantalla, llama al método **repaint()** nunca al método **paintComponent()**.

La clase Graphics tiene el método **drawString()** con esta sintaxis:

```
g.drawString( texto, x, y);
```

Si queremos el texto "Esto es GUIDAM" en las coordenadas (75, 100), significa que el primer carácter de la cadena comienza en el pixel (75,100) del contenedor:

```
class UnTexto extends JComponent {
    public static final int MSJ_X = 75;
    public static final int MSJ_Y = 100;
    public void paintComponent(Graphics g) {
        g.drawString("Esto es GUIDAM", MSJ_X, MSJ_Y);
    }
}
```

Por último, un componente podría tener unas dimensiones por defecto para informar a los usuarios, sobreescribiendo el método **getPreferredSize()** y devolviendo un objeto **Dimension**:

```
class UnTexto extends JComponent {
    public static final int MSJ_X = 75;
    public static final int MSJ_Y = 100;
    public static final int ANCHO = 300;
    public static final int ALTO = 200;
    public void paintComponent(Graphics g) {
        g.drawString( "Esto es GUIDAM", MSJ_X, MSJ_Y);
    }
    public Dimension getPreferredSize() {
        return new Dimension( ANCHO, ALTO );
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

Cuando llenes un frame con uno o más componentes y quieras ajustar el tamaño al preferido, llama a **pack()** en vez de a **setSize()**:

```
class Ejemplo extends JFrame {
    public Ejemplo() {
        add( new Componente() );
        pack();
    }
}
```

Nota: muchos programadores en vez de extender *JComponent*, prefieren extender *JPanel*. *JPanel* está pensado para contener a otros componentes aunque se pueda pintar sobre él. La única diferencia es que un panel es opaco y por tanto responsable de pintar todos sus pixels. La forma más sencilla de hacerlo es llamar al método **paintComponent()** de la superclase.

```
class Ejemplo extends JPanel {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        // Tu propio código para dibujar en el JPanel...
    }
}
```

EJEMPLO 4: programa gráfico que muestra un mensaje de texto.

```
package mensaje; // Comentar la línea si ejecutas desde consola

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

public class U7Ejemplo3 {
    public static void main(String[] args) {
        EventQueue.invokeLater(
            new Runnable() {
                public void run() {
                    JFrame f = new MensajeFrame();
                    f.setTitle("Esto es GUIDAM");
                    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    f.setVisible(true);
                }
            }
        );
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

    };
}

class MensajeFrame extends JFrame {
    // private static final long serialVersionUID = 1L;
    public MensajeFrame() {
        add( new MensajeComponente() );
        // equivale a: getContentPane().add(new MensajeComponente() );
        pack();
    }
}

class MensajeComponente extends JComponent {
    public static final int MENSAJE_X = 75;
    public static final int MENSAJE_Y = 100;
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;
    public void paintComponent(Graphics g) {
        g.drawString("Esto es GUIDAM" , MENSAJE_X, MENSAJE_Y);
    }
    public Dimension getPreferredSize() {
        return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }
}

```

javax.swing.JFrame

- **Container** **getContentPane()**: devuelve el objeto content pane.
- **Component** **add(Component c)**: añade c al content pane.

java.awt.Component

- **void** **repaint()**: solicita un redibujado del componente.
- **Dimension** **getPreferredSize()**: devuelve el tamaño preferido del componente.

javax.swing.JComponent

- **void** **paintComponent(Graphics g)**: indica que cosas se dibujan.

java.awt.Window

- **void** **pack()**: redimensiona la ventana según el tamaño preferido de los componentes.



UNIDAD 8. Aplicaciones Controladas por Eventos.

8.2.2. OTROS ELEMENTOS.

COMPONENTES Y LAYOUTS

Si usas **JPanel** como lo que es (un contenedor), podrás añadirle otros **componentes** (objetos de la GUI). Java tiene muchas clases que definen componentes. Salvo los de alto nivel (como los JFrames) deben añadirse a un contenedor antes de que puedan aparecer en pantalla. Si queremos añadir dos componentes (cada uno referenciado con su variable) a un **JPanel** que se referencia con la variable contenedor:

```
contenedor.add(displayPanel, BorderLayout.CENTER);  
contenedor.add(okBoton, BorderLayout.SOUTH);
```

El contenedor puede ser un objeto de tipo **JPanel** que se convierta en el content panel de la ventana. El primer componente que se añade es un displayPanel que muestra un mensaje. El segundo es un botón referenciado en la variable okBoton (de tipo JButton).

El segundo parámetro es una clase "BorderLayout" y tiene constantes que definen como se ordenan los componentes visuales dentro del contenedor. Cuando se añaden componentes al contenedor, se necesita un criterio para decidir como aparecen dentro de él. Normalmente se usa un objeto **layout manager** que implementa una política sobre como ordenarlos. Diferentes layouts implementan diferentes políticas. La clase **BorderLayout** es uno de esos tipos y se puede fijar con la sentencia:

```
contenedor.setLayout( new BorderLayout() );
```

Así que el botón se añade en la posición **BorderLayout.SOUTH** y eso significa que va a la parte de abajo del panel y la posición **BorderLayout.CENTER** hace que ocupe toda la zona que no rellena el botón (más adelante veremos con más detalle los layout managers).



UNIDAD 8. Aplicaciones Controladas por Eventos.

Y esa es la técnica general para ir creando interfaces gráficas: crear un contenedor, establecer un layout manager e ir añadiendo componentes. Algunos componentes son contenedores, así que son contenidos y a su vez pueden contener otros componentes. De esta manera es posible diseñar complejas interfaces.

EVENTOS Y LISTENERS

La estructura de contenedores y componentes define como es la apariencia de la interfaz (como se ve), pero no indica nada de su comportamiento.

Las GUI's son conducidas por eventos, así que el programa espera que se genere algún evento por la acción del usuario (o por alguna otra causa). Cuando aparece un evento, el programa responde ejecutando un método manejador del evento. Para definir el comportamiento del programa debes escribir métodos manejadores de eventos que respondan a los eventos que te interesen.

La técnica más usada en Java para manejar eventos es usar **listeners de eventos**. Un listener es un objeto que incluye uno o más métodos **manejadores de eventos**. Cuando un evento es detectado por otro objeto, como un botón o un menú, se lo indican al listener, quien responde ejecutando el correspondiente método manejador. Un evento se genera o se detecta en un objeto y el listener tiene la responsabilidad de responder a él.

El propio evento es otro objeto, que almacena la información sobre el tipo de evento, cuando ha ocurrido, donde, etc. La división de responsabilidades hace más fácil organizar programas grandes:



UNIDAD 8. Aplicaciones Controladas por Eventos.

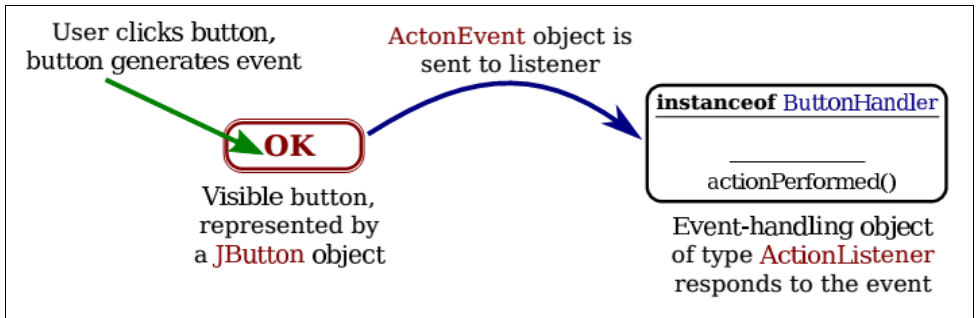


Figura 6: Elementos que definen el comportamiento.

A modo de ejemplo, mira la figura 6 donde aparece un botón con el texto "Ok". El usuario hace clic en él, el objeto detecta el evento generado. El evento se representa como un objeto de la clase **ActionEvent**. Se dice que el origen del evento es el botón, donde se genera. El objeto listener es un objeto de la clase **ButtonHandler**, que se define como una clase anidada dentro de los programas (normalmente):

```
private static class ButtonHandler implements ActionListener {
    public void actionPerformed( ActionEvent e ) {
        System.exit(0);
    }
}
```

Si el listener estuviese definido como se ve en las sentencias de arriba, el programa acabaría. La interfaz **ActionListener** obliga a implementar un solo método que es **actionPerformed()**.

Hay un paso más para que todo funcione, hay que decirle al botón que tiene un listener al que mandarle eventos y eso lo hace esta sentencia en la que la variable listener es una referencia a un objeto de la clase **ButtonHandler**:

```
okButton.addActionListener( listener );
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

8.2.3. DIBUJAR SOBRE LOS COMPONENTES.

En Java, los componentes GUI son subclases de la clase `java.awt.Component`. Los componentes usados en Swing son subclases de `javax.swing.JComponent`, subclase a su vez de `java.awt.Component`. Cada componente es responsable de dibujarse a él mismo. Como ya tiene programado como debe dibujarse, lo hará cuando sea necesario de forma automática.

Si te interesa dibujar sobre un componente, debes definir que quieres dibujar. Por tanto debes crear tu propia clase derivada de la del componente y definir el método que se encarga de dibujar. Puede ser una subclase de un `JPanel` que tiene el método `public void paintComponent(Graphics g)` para dibujarse. Cuando sea necesario que el componente se dibuje, se llama este método de forma automática, como ya se ha comentado.

Solo puedes dibujar en un componente usando un objeto `Graphics` del componente. Para obtener este objeto gráfico de un componente, lo podemos hacer de dos maneras:

- cogerlo del parámetro que pasa el sistema dentro del método `paintComponent(Graphics g) {...}`.
- Ejecutar el método `Graphics g = componente.getGraphics();`
Si `g` es un objeto creado a partir de este método, debes liberar la memoria que ocupa tan pronto como no lo vayas a usar, así liberas recursos del sistema operativo. Ej: `g.dispose();`

El método `paintComponent()` de la clase `JPanel` por defecto simplemente rellena el panel con el color de fondo. Al definir una subclase de un `Jpanel`, antes de hacer tus dibujos, lo llamas para que se ejecute. Así que la mayoría de métodos `paintComponent()` que sobreescribas de un `Jpanel` tendrán este formato:



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
public void paintComponent(Graphics g) {  
    super.paintComponent(g);  
    . . . // Dibujar tu contenido en el componente.  
}
```

Recuerda!! Si dibujando otro componente (estando en su método `paintComponent()` por ejemplo) ves que necesitas redibujar otro, **nunca llames a su método `paintComponent()`, llama a su método `repaint()`.**

COORDENADAS

La pantalla del ordenador está formada por una rejilla de pequeños cuadrados luminosos llamados pixels. El color de cada uno puede cambiarse de forma individual.

Cada uno de estos pixels puede referenciarse mediante sus coordenadas en la pantalla. Unas coordenadas son dos enteros (x, y). El pixel de la esquina superior izquierda tiene de coordenadas la (0,0). A medida que te desplazas a la derecha, el valor de x aumenta. A medida que bajas en la pantalla, el valor de y aumenta.

Todo componente ocupa una zona cuadrada de la pantalla. La figura 7 muestra un trozo de la pantalla, el trozo que ocupa un componente. Es como una pequeña pantalla, que tiene su propio sistema de coordenadas. Para saber las dimensiones de un componente, puedes llamar a los métodos `getWidth()` y `getHeight()`. Es bueno no asumir nada sobre su tamaño y averiguarlo justo antes de dibujar en su superficie. Ej:

```
public void paintComponent(Graphics g) {  
    super.paintComponent(g);  
    int ancho= getWidth(); // ancho del componente  
    int alto= getHeight(); // alto del componente  
    // Dibujar sabiendo su tamaño...
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

}

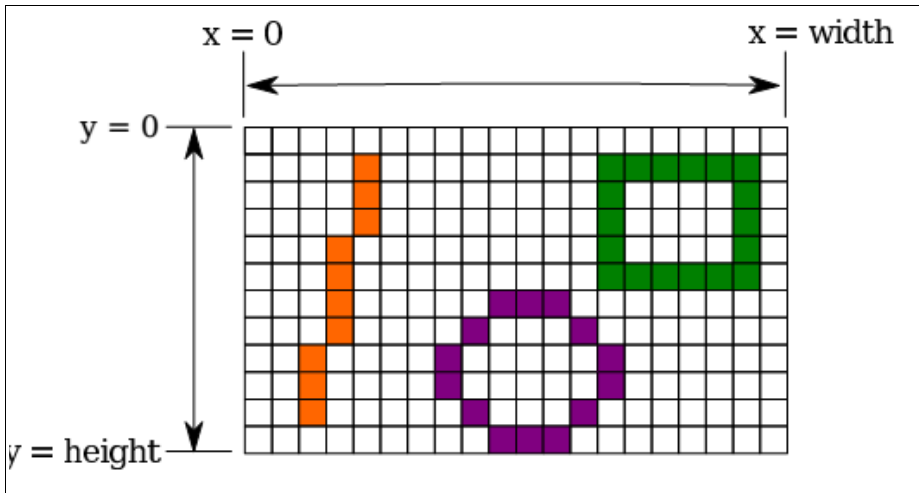


Figura 7: Rejilla de píxeles de un componente GUI.

COLORES

Java puede trabajar con el sistema de colores RGB (rojo, verde y azul). Cada color es un objeto de la clase `java.awt.Color`. Puedes construir un nuevo color indicando el valor de rojo, verde y azul en un constructor:

```
Color unColor = new Color(r,g,b);
```

Uno de los constructores tiene valores enteros entre 0 y 255. En el otro son flotantes en el rango 0.0F a 1.0F. La clase tiene constantes que definen los colores más comunes: `Color.WHITE`, `Color.BLACK`, `Color.RED`, `Color.GREEN`, `Color.BLUE`, `Color.CYAN`, `Color.MAGENTA`, `Color.YELLOW`, `Color.PINK`, `Color.ORANGE`, `Color.LIGHT_GRAY`, `Color.GRAY` y `Color.DARK_GRAY`.

Otro sistema de colores es el HSB (hue, saturación y brillo). Hue es el color básico (del rojo al naranja y el resto de colores del arcoiris). El



UNIDAD 8. Aplicaciones Controladas por Eventos.

brillo es la luminosidad. Y un color completamente saturado es un color de tono puro. Si la saturación baja, es como quitarle blanco al color. En Java, los valores hue, saturación y brillo son siempre valores de tipo float entre 0.0F y 1.0F. Para crear un color con HSB:

```
Color unColor = Color.getHSBColor(h,s,b);
```

Por ejemplo, para crear un color aleatorio podemos usar:

```
Color azar = Color.getHSBColor((float)Math.random(), 1.0F, 1.0F);
```

Una de las propiedades del objeto Graphics es el color en el que se dibuja. Puedes cambiarlo con `g.setColor(c)`, Ej:

```
g.setColor(Color.GREEN);
```

Para saber el color que está usando actualmente: `g.getColor()`.

Cada componente tiene asociado un color de fondo. Cuando creas un nuevo objeto gráfico para un componente, el color de dibujo (foreground) y de fondo (background) se establece al color de fondo del componente.

Para cambiar los colores de dibujo y de fondo de un componente c, se usan los métodos `c.setForeground(c)` y `c.setBackground(c)`.

FUENTES

Una fuente representa un tamaño y estilo de letra. En Java, una fuente se identifica mediante un nombre, un estilo y un tamaño. Las fuentes disponibles varían, aunque hay 4 que siempre tendrás:

- **"Serif"**: tiene una pequeña decoración en cada letra, como una pequeña línea en la base de la i.
- **"SansSerif"**, sin adornos en las letras (sin serifs).
- **"Monospaced"**, todas las letras tienen la misma anchura.



UNIDAD 8. Aplicaciones Controladas por Eventos.

- “Dialog”, la usada en las cajas de diálogo.

Para encontrar las fuentes que tienes disponibles en un sistema puedes llamar a los métodos `getAvailableFontFamilyNames()` de la clase `GraphicsEnvironment`. Devuelve un array de strings con los nombres de las fuentes disponibles. Para obtener un objeto de la clase `GraphicsEnvironment` usa su método static `getLocalGraphicsEnvironment()`. Ejemplo:

```
import java.awt.*;  
  
public class ListaFuentes {  
    public static void main(String[] args) {  
        String[] nombreFuentes = GraphicsEnvironment  
                                .getLocalGraphicsEnvironment()  
                                .getAvailableFontFamilyNames();  
        for(String fN : nombreFuentes)  
            System.out.println(fN);  
    }  
}
```

El **estilo de una fuente** se indica usando constantes simbólicas definidas en la clase `Font`. Hay 3 valores que se mezclan sumándolos o con operaciones OR a nivel de bits:

- `Font.PLAIN`
- `Font.ITALIC`
- `Font.BOLD`
- No hay subrayado (debes usar `FontMetrics`).

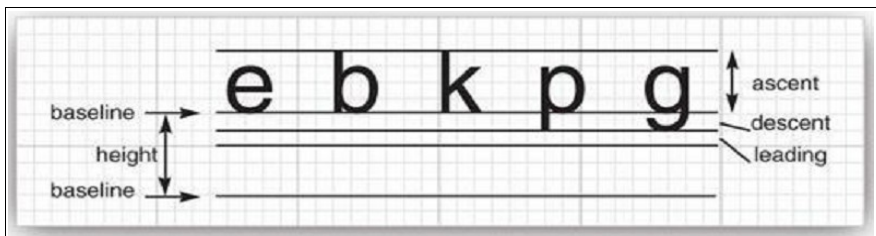


Figura 8: Elementos que definen el tamaño de una fuente.



UNIDAD 8. Aplicaciones Controladas por Eventos.

El tamaño de una fuente es un entero. Los tamaños típicos van desde 9 hasta 36, aunque pueden utilizarse tamaños mayores. El tamaño es igual a la altura de la letra más alta, aunque es un poco más complicado de calcular. Por defecto el tamaño es 12.

Java usa la clase `java.awt.Font` para representar fuentes. Puedes construir una nueva indicando nombre, estilo y tamaño al constructor. Para mostrar un texto con la fuente sansserif usando negrita y 14 de tamaño:

```
Font ssb14 = new Font("SansSerif", Font.BOLD, 14);
g2.setFont(ssb14);
String mensaje = "Hola mundo";
g2.drawString( mensaje, 75, 100);
```

EJEMPLO 5: Escribir un texto centrado.

En vez de dibujar el texto en cualquier posición, lo vamos a centrar. Para hacerlo, necesitamos saber la anchura y la altura que va a ocupar el texto en pixels cuando se dibuje. Estas dimensiones dependen de 3 factores:

- La fuente usada.
- La cadena de texto a dibujar
- El dispositivo en el que se dibuja el texto.

Para conocer las características del dispositivo (la pantalla) llamamos al método `getFontRenderContext()` de la clase `Graphics2D`. Devuelve un objeto de la clase `FontRenderContext`. Pasando el objeto al método `getStringBounds()` de la clase `Font` puedes saber lo que ocupa el texto.

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D r = f.getStringBounds(mensaje, context);
```

Devuelve un rectángulo que envuelve al string. Para interpretar las dimensiones del rectángulo debes conocer algunos términos que



UNIDAD 8. Aplicaciones Controladas por Eventos.

aparecen en la figura 8. Por ejemplo, la **baseline** es la línea horizontal imaginaria donde se apoya la base de la mayoría de las letras (la "e" por ejemplo). El **ascent** es la distancia vertical entre la baseline hasta la parte alta de letras como la "b" o la "k," o una letra en mayúsculas. El **descent** es la distancia desde la baseline hasta el **descender**, que es la parte baja de letras como la "p" o la "g".

Leading es el espacio vertical entre el descent de una línea y el ascent de la siguiente línea. La altura de la fuente será la distancia entre sucesivas baselines (descent + leading + ascent). Y la anchura es la distancia horizontal que ocupa el texto en el dispositivo. No todas las letras tienen el mismo ancho en todas las fuentes.

El rectángulo tiene su origen en la baseline del string y la coordenada alta de la cadena es negativa. Puedes obtener el ancho y el alto así:

```
double stringAncho = r.getWidth();
double stringAlto = r.getHeight();
double ascent = -r.GetY();
```

Si necesitas saber el descent o el **leading**, usa el método **getLineMetrics()** de la clase **Font**. Devuelve un objeto de la clase **LineMetrics**, que tiene los métodos necesarios. Ej:

```
LineMetrics m = f.getLineMetrics(mensaje, context);
float descent = m.getDescent();
float leading = m.getLeading();
```

El siguiente código usa la información recopilada para dibujar el texto centrado en el componente:

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D r = f.getStringBounds(mensaje, context);
// ( x, y) = esquina izquierda del texto
double x = (getWidth() - r.getWidth() ) / 2;
double y = (getHeight() - r.getHeight() ) / 2;
// sumar el ascent
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
double ascent = -bounds.getY();  
double baseY = y + ascent;  
g2.drawString( mensaje, (int)x, (int) baseY);
```

Cada contexto gráfico tiene una fuente actual. Puedes cambiarla con el método `g.setFont()`. Y consultarla con `g.getFont()`.

Cada componente también tiene una fuente por defecto y los mismos métodos anteriores. Cuando creas el contexto gráfico de un componente, la fuente que tiene se copia de la que tenga el componente.

DIBUJAR FIGURAS

La clase `Graphics` incluye una gran cantidad de métodos para dibujar figuras. Se usan las coordenadas (x,y) de las figuras para indicar su posición. El color de dibujo es el del contexto y se puede cambiar con `setColor()`. Si dibujas fuera de los límites del componente, se ignora. En los ejemplos suponemos que `g` es un contexto gráfico creado.

- `g.drawString(String str, int x, int y)` — dibuja el texto en (x,y) donde y es la baseline del texto.
- `g.drawLine(int x1, int y1, int x2, int y2)` — Dibuja una línea desde el punto (x1,y1) hasta el punto (x2,y2). El lápiz usado se extiende un pixel a la derecha y abajo de las coordenadas.
- `g.drawRect(int x, int y, int w, int h)` — Dibuja un rectángulo que une la esquina superior izquierda (x,y) y tiene w-1 pixel de ancho y h-1 pixels de alto. El lápiz es el mismo que las líneas. Así que si quieres poner un marco al componente, debes dibujar: `g.drawRect(0, 0, getWidth()-1, getHeight()-1)`; o los bordes inferiores no aparecen.
- `g.drawOval(int x, int y, int w, int h)` — Dibuja un óvalo inscrito en el rectángulo que indicas.



UNIDAD 8. Aplicaciones Controladas por Eventos.

- `g.drawRoundRect(int x, int y, int w, int h, int xdiam, int ydiam)` — Dibuja un rectángulo con las esquinas redondeadas. El arco de las esquinas lo definen `xdiam` e `ydiam` (diámetro del arco).
- `g.draw3DRect(int x, int y, int w, int h, boolean raised)` — Dibuja un rectángulo con efecto 3D, que se eleva o se hunde en el fondo. Usa como sombra un color más oscuro. Con algunos colores el efecto no está conseguido.
- `g.drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)` — Dibuja un arco, una parte de un óvalo. La parte del óvalo que se dibuja va desde `startAngle` hasta `arcAngle` grados.
- `g.fillRect(int x, int y, int w, int h)` — Dibuja un rectángulo relleno (no se incluye el pixel extra de la versión no rellena), así que para rellenar todo el componente, harías la llamada `g.fillRect(0, 0, getWidth(), getHeight());`
- `g.fillOval(int x, int y, int w, int h)` — Dibuja un óvalo relleno.
- `g.fillRoundRect(int x, int y, int w, int h, int xdiam, int ydiam)` — Dibuja un rectángulo relleno.
- `g.fill3DRect(int x, int y, int w, int h, boolean raised)` — Dibuja un rectángulo 3D relleno.
- `g.fillArc(int x, int y, int w, int h, int startAngle, int arcAngle)` — Dibuja un arco relleno.

MOSTRAR IMÁGENES

Suponiendo que la imagen a mostrar está almacenada en un fichero (también la podría generar el programa), podemos usar las clases `Image` e `ImageIcon` (hay más posibilidades):

```
Image img = new ImageIcon(nombreFichero).getImage();
```

Ahora, la variable `img` contiene una referencia al objeto que encapsula los datos de la imagen. Puedes mostrarla usando el método `drawImage()`



UNIDAD 8. Aplicaciones Controladas por Eventos.

de la clase `Graphics`.

```
public void paintComponent(Graphics g) {  
    // Otras sentencias. . .  
    g.drawImage(img, x, y, null);  
}
```

Si queremos hacer un enlosado de un área, podemos usar el método `copyArea()` para ir copiando una y otra vez un trozo del área del componente a otro lado:

```
for(int i = 0; i * imgAncho <= getWidth(); i++)  
    for(int j = 0; j * imgAlto <= getHeight(); j++)  
        if(i + j > 0)  
            g.copyArea(0,0,imgAncho,imgAlto,i*imgAncho,j*imgAlto);
```

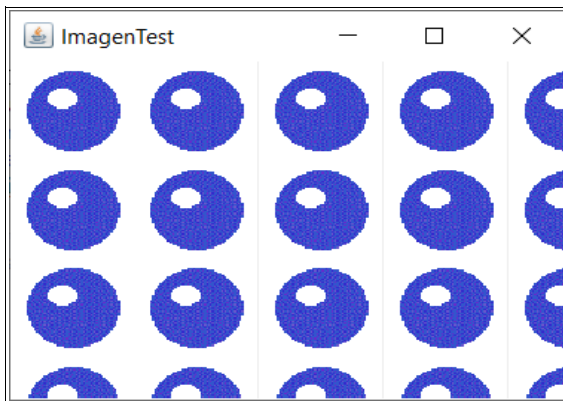


Figura 9: (Enlosado) Rellenar el área de un componente con una imagen.

EJEMPLO 6: programa que rellena la ventana con círculos azules.

```
package imagen; // Comentar para ejecutar desde consola  
  
import java.awt.*;  
import javax.swing.*;  
  
public class U7Ejemplo5 {  
    public static void main(String[] args) {  
        EventQueue.invokeLater(  
            new Runnable() {
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

        public void run() {
            JFrame frame = new ImageFrame();
            frame.setTitle("ImagenTest");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
        }
    }
);
}
}

class ImageFrame extends JFrame {
    private static final long serialVersionUID = 1L;

    public ImageFrame() {
        add( new ImageComponent() );
        pack();
    }
}

class ImageComponent extends JComponent {
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;
    private Image imagen;

    public ImageComponent( ) { // Haz tu una bola en el paint
        imagen = new ImageIcon("bola-azul.png").getImage();
    }

    public void paintComponent(Graphics g) {
        if (image == null) return;
        int iAncho = imagen.getWidth(this);
        int iAlto = imagen.getHeight(this);
        // dibujar la imagen en la esquina superior izq.
        g.drawImage(imagen, 0, 0, null);
        // Copiar ese área al resto
        for(int i = 0; i * iAncho <= getWidth(); i++)
            for(int j = 0; j * iAlto <= getHeight(); j++)
                if(i + j > 0)
                    g.copyArea(0, 0, iAncho, iAlto, i*iAncho, j*iAlto);
    }

    public Dimension getPreferredSize() {
        return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }
}

```

Listado de métodos de Graphics para mostra imágenes:



UNIDAD 8. Aplicaciones Controladas por Eventos.

java.awt.Graphics 1.0

- `boolean drawImage(Image img, int x, int y, ImageObserver observer);` Dibuja una imagen sin escalar a tamaño original. La función podría acabar antes de que la imagen esté dibujada. El parámetro `observer` indica el progreso del dibujo.
- `boolean drawImage(Image img, int x, int y, int w, int h, ImageObserver observer);` Dibuja una imagen escalada para que se ajuste a la región que indicas (w ancho y h alto).
- `void copyArea(int x, int y, int w, int h, int dx, int dy);` copia un área de la pantalla a otra zona de destino (dx,dy).

GRAPHICS 2D

En Java 1.0, la clase **Graphics** tenía métodos para dibujar líneas, rectángulos, elipses, etc. Pero con limitaciones (no puede cambiar la textura ni rotar las figuras).

Java 1.2 introdujo la librería **Java 2D**, que implementa operaciones gráficas más potentes. Para poder utilizarla debes obtener un objeto de la clase `Graphics2D` que es una subclase de `Graphics`. Incluso en Java 2, los métodos como `paintComponent()` reciben un objeto que no es `Graphics2D`. Simplemente debes usar un cast de tipos:

```
public void paintComponent(Graphics g) {  
    Graphics2D g2 = (Graphics2D) g;  
    // A partir de ahora, g2 es más potente...  
}
```

La librería **Java 2D** organiza las figuras geométricas a la manera de la orientación a objetos. Hay clases para líneas, rectángulos y elipses que implementan la interfaz `Shape`:

- `Line2D`
- `Rectangle2D`
- `Ellipse2D`



UNIDAD 8. Aplicaciones Controladas por Eventos.

Nota: Java 2D soporta figuras más complejas como arcos, curvas cúbicas y cuadráticas y rutas generales.

Para dibujar una figura, primero creas un objeto de esa clase y llamas al método `draw()` de la clase `Graphics2D`. Ej:

```
Rectangle2D r = . . . ;  
g2.draw(r);
```

Las figuras de Java 2D no usan coordenadas enteras, usan valores en punto flotante (lo que da más precisión: puedes indicar tamaños en mm que luego se convierten a pixels). Usa flotantes de precisión simple de forma interna. Pero obliga a hacer casts de tipos. Por ejemplo:

```
float f = 1.2; // Error, es un double por defecto
```

La solución es añadir un sufijo F ó f a la constante:

```
float f = 1.2F; // Ok
```

Ahora esta otra:

```
Rectangle2D r = . . . ;  
float f = r.getWidth(); // Error
```

Corregida:

```
float f = (float) r.getWidth(); // Ok
```

Como esto se convierte en un suplicio, al final han implementado dos versiones de cada figura, una para los programadores espabilados con float, y otra para el resto de nosotros que nos comemos en el código estos detalles. Las clases `Rectangle2D.Float` y `Rectangle2D.Double` extienden a `Rectangle2D`. Ejemplo:

```
Rectangle2D.Float r1= new Rectangle2D.Float(10.0F,25.0F,22.5F,20.0F);  
Rectangle2D.Double r2= new Rectangle2D.Double(10.0,25.0,22.5,20.0);
```




UNIDAD 8. Aplicaciones Controladas por Eventos.

Pero no hay ningún problema en que uses directamente Rectangle2D:

```
Rectangle2D r1 = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D r2 = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

Entrar en profundidades con Graphics2D está fuera del objetivo de este módulo (al menos sabemos que existe si lo necesitamos). Pero un par de usos sencillos si que voy a comentarte. El primero es esta sentencia que activa el **antialiasing** (una mejora de los dibujos de texto y figuras que disminuye el efecto de pixelado):

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON );
```

El otro comando interesante es modificar el lápiz con el que se pinta:

```
g2.setStroke( new BasicStroke(ancho_del_trazo) );
```

EJEMPLO 7: programa que use un JPanel como superficie de dibujo. Dibujaremos un texto sobre un fondo negro. Luego copiaremos el texto en colores elegidos al azar. Usaremos 5 fuentes distintas, con diferentes tamaños y estilos:

```
package fuentes; // Comenta si ejecutas desde consola

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

public class RandomStringPanel extends JPanel {
    private static final long serialVersionUID = 1L;
    private String txt; // Mensaje a mostrar
    private Font font1, font2, font3, font4, font5; // Fuentes

    public RandomStringPanel() {
        this(null); // Llama al otro constructor
    }

    /**
     * Constructor
     * @param texto El mensaje a mostrar. Si es null "Java!"
     */
    public RandomStringPanel(String texto) {
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
txt = texto;
if (txt == null) txt = "Java!";
font1 = new Font("Serif", Font.BOLD, 14);
font2 = new Font("SansSerif", Font.BOLD + Font.ITALIC, 24);
font3 = new Font("Monospaced", Font.PLAIN, 30);
font4 = new Font("Dialog", Font.PLAIN, 36);
font5 = new Font("Serif", Font.ITALIC, 48);
setBackground(Color.BLACK);
}

public void paintComponent(Graphics g) {
    super.paintComponent(g); // Es un JPanel, hay que rellenar
    Graphics2D g2 = (Graphics2D)g; // suavizar líneas
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON );
    int ancho = getWidth();
    int alto = getHeight();
    for(int i = 0; i < 25; i++) {
        int fNum = (int)(5 * Math.random()) + 1;
        switch (fNum) {
            case 1: g.setFont(font1);
                    break;
            case 2: g.setFont(font2);
                    break;
            case 3: g.setFont(font3);
                    break;
            case 4: g.setFont(font4);
                    break;
            case 5: g.setFont(font5);
                    break;
        } // switch
        // Poner color
        float h = (float) Math.random();
        g.setColor( Color.getHSBColor(h, 1.0F, 1.0F) );
        // Seleccione posición
        int x,y;
        x = -50 + (int)(Math.random()*(ancho + 40) );
        y = (int) (Math.random()*(alto + 20) );
        // Dibuja texto
        g.drawString(txt, x, y);
    } // for
} // paintComponent()
} // clase
```

¿DONDE ESTÁ EL MAIN() DEL PROGRAMA?

Si compilas este código, no falla, pero tampoco se ejecuta porque no



UNIDAD 8. Aplicaciones Controladas por Eventos.

encuentra el `main()` definido.

Otro problema es que un `JPanel` no puede existir por sí mismo, necesita estar contenido en otro `JPanel` o una ventana (**`JFrame`**). Debes añadir un método **`main()`** a otra clase `RandomString` para que haga este trabajo:

```
package fuentes;  
  
import javax.swing.JFrame;  
  
public class RandomString {  
    public static void main(String[] args) {  
        JFrame window = new JFrame("Java!");  
        RandomStringPanel content = new RandomStringPanel();  
        window.setContentPane(content);  
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        window.setLocation(120,70);  
        window.setSize(350,250);  
        window.setVisible(true);  
    }  
}
```

El método `main()` no es parte de la clase panel. Aunque es posible incluir a `main()` como parte de la clase panel. Es posible ejecutar el panel como un programa con la ventaja de usar solo un fichero.

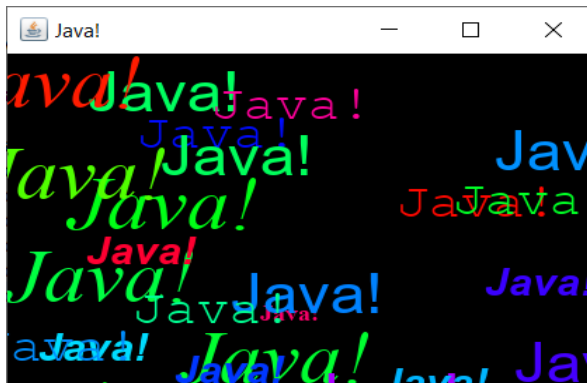


Figura 10: Resultado de Ejecutar el ejemplo.



UNIDAD 8. Aplicaciones Controladas por Eventos.

8.3. EVENTOS.

Los eventos son la parte central de la programación de las GUI. Son los elementos que indican las sentencias que hay que ejecutar y se representan mediante objetos. Cuando se genera, el sistema recava toda la información relevante y construye un objeto que la encapsula. Diferentes tipos de eventos están representados por diferentes objetos de clases distintas. Por ejemplo si pulsas un botón con el ratón, la clase del evento generado es **MouseEvent**. El objeto contiene información de donde se ha originado el evento (**source**, el botón donde se ha clicado), las coordenadas (x,y) del punto donde ha ocurrido el clic, la hora exacta en que ha sucedido y qué botón del ratón se ha presionado. Cuando el usuario pulsa una tecla del teclado, el evento creado es de la clase **KeyEvent**.

El trabajo de quien programa en Java, es implementar métodos para ejecutarlos en respuesta a estos eventos. Aunque en realidad, hay un gran procesamiento debajo desde que el usuario pulsa la tecla hasta que algún método se ejecuta como respuesta. No es necesario que lo conozcas al dedillo, pero sí comprenderlo: hay un método ejecutándose que tiene un bucle:

```
Mientras <el programa esté en ejecución> Haz:  
    <Esperar al siguiente evento>  
    <Llamar al método que se encarga de responder>
```

Este bucle se llama el **bucle de eventos**. Cada programa GUI tiene uno. En Java no tienes que escribirlo, es parte del sistema. Pero en C++ por ejemplo, si tendrías que hacerlo (salvo que uses un framework que lo haga por tí).

MANEJO DE EVENTOS

Para que un evento tenga un efecto, primero hay que ser capaz de



UNIDAD 8. Aplicaciones Controladas por Eventos.

detectarlo (escucharlo). Escuchar eventos es lo que hacen los objetos llamados **event listeners**. Deben contener métodos de instancia para manejar los eventos que escuchan. Por ejemplo, si un objeto es de la clase `MouseEvent`, debe tener el siguiente método:

```
public void mousePressed(MouseEvent evt) { . . . }
```

El cuerpo del método define como responde el objeto cuando es notificado de que se ha pulsado un botón del ratón. El parámetro `evt` tiene información que puede utilizar el listener para decidir la respuesta a dar.

Los métodos que necesita implementar un event listener del ratón se indican en la **interfaz `MouseListener`**. Muchos eventos están asociados a componentes GUI. Por ejemplo, si el usuario presiona un botón del ratón, el componente asociado es sobre el que el usuario ha realizado clic.

Antes de que un objeto listener pueda detectar eventos originados en un componente, **hay que registrar (asociar) el listener con el componente**. Por ejemplo, si un objeto `MouseListener` llamado `mL`, necesita oír eventos asociados al componente referenciado por la variable `comp`, hay que ejecutar:

```
comp.addMouseListener( mL );
```

El método **`addMouseListener()`** es un método de instancia de la clase `Component`. Las clases de eventos como `MouseEvent` y las interfaces de listener como `MouseListener`, se definen en el package **`java.awt.event`**. Así que si necesitas trabajar con eventos debes importarlo. Por resumir lo que hay que hacer para usar eventos:

1. Importar especificaciones: `import java.awt.event.*;` (o de forma individual).



UNIDAD 8. Aplicaciones Controladas por Eventos.

2. Declarar algunas clases que implementen la interfaz listener adecuada, como `MouseListener`;
3. Aportar definiciones de los métodos de la interfaz.
4. Registrar un objeto de esa clase con el que genere los eventos llamando a `addMouseListener()` del componente.

Cualquier objeto puede actuar como un event listener si implementa la interfaz adecuada. Puede escuchar los eventos que él mismo genere. Un panel escucha los eventos de los componentes que tiene en su interior. Algunos programadores prefieren usar clases anónimas internas para definir los objetos listeners.

8.3.1. EVENTOS DEL RATÓN.

MOUSEEVENT Y MOUSELISTENER

La interfaz **MouseListener** especifica estos 5 métodos:

```
public void mousePressed(MouseEvent evt);  
public void mouseReleased(MouseEvent evt);  
public void mouseClicked(MouseEvent evt);  
public void mouseEntered(MouseEvent evt);  
public void mouseExited(MouseEvent evt);
```

El método `mousePressed()` se llama tan pronto como el usuario presiona hacia abajo uno de los botones del ratón y `mouseReleased()` cuando lo suelta. Estos dos son los más usados. Si no quieres responder a alguno de ellos, al implementarlos dejas el cuerpo vacío, aunque deben aparecer.

El método `mouseClicked()` se llama si el usuario pulsa un botón y lo libera sin haber movido el ratón (esto dispara las 3 rutinas en realidad). Los métodos `mouseEntered()` y `mouseExited()` se llaman cuando el cursor entra y abandona la superficie de un componente.



UNIDAD 8. Aplicaciones Controladas por Eventos.

EJEMPLO 8: Vamos a añadir al programa de las fuentes aleatorias, que el panel se redibuje de nuevo cuando hagamos clic en él.

Un mouse listener debe detectar eventos de ratón y cuando detecte el evento **mousePressed** responde llamando al método **repaint()** del panel. Lo vamos a poner en una clase pública separada en su propio fichero por claridad, aunque no es lo común:

```
package fuentes;

import java.awt.Component;
import java.awt.event.*;
/**
 * El objeto Repinta es un MouseListener que responde a
 * mousePressed llamando a repaint()
 */
public class Repinta implements MouseListener {
    public void mousePressed(MouseEvent evt) {
        Component source = (Component)evt.getSource();
        source.repaint(); // repintar
    }
    public void mouseClicked(MouseEvent evt) { } // Vacíos pero aparecen
    public void mouseReleased(MouseEvent evt) { }
    public void mouseEntered(MouseEvent evt) { }
    public void mouseExited(MouseEvent evt) { }
}
```

Ahora debemos asociar un objeto del listener con el componente a escuchar. Esto se puede hacer en el método **main()** por ejemplo:

```
Repinta listener = new Repinta(); // Crea objeto MouseListener
panel.addMouseListener(listener); // Registrarlo
```

Aunque en el ejemplo hemos escrito la clase **Repinta** para usarla con la clase **RandomString**, la clase **Repinta** no hace ninguna referencia a ella. ¿Cómo es posible? Porque el método **mousePressed()** mira el evento que es quien le dice el origen del mismo, si usamos la clase con otro componente funcionará igual de bien. Esta es la clave del diseño modular, tener cosas independientes que puedan unirse juntas y



UNIDAD 8. Aplicaciones Controladas por Eventos.

funcionar.

EJEMPLO 9: Ahora versionamos el programa del ejemplo para hacer lo mismo pero la clase Repinta la copiamos dentro de la clase RandomString como una clase estática anidada y funcionará igual de bien.

```

import java.awt.Component;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.JFrame;
/**
 * Muestra ...
 */
public class RandomString {
    public static void main(String[] args) {
        JFrame window = new JFrame("Haga clic para Redibujar!");
        RandomString content = new RandomString();
        content.addMouseListener( new Repaint() );
        window.setContentPane(content);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setLocation(120,70);
        window.setSize(350,250);
        window.setVisible(true);
    }
    private static class Repinta implements MouseListener {
        public void mousePressed(MouseEvent evt) {
            Component source = (Component)evt.getSource();
            source.repaint();
        }
        public void mouseClicked(MouseEvent evt) { }
        public void mouseReleased(MouseEvent evt) { }
        public void mouseEntered(MouseEvent evt) { }
        public void mouseExited(MouseEvent evt) { }
    }
} // Clase Repinta
} // clase RandomString

```

DATOS DEL EVENTO MOUSEEVENT

Si evt es el parámetro que reciben los listeners, puedes saber las coordenadas del ratón cuando se produjo el evento llamando a **evt.getX()** y **evt.getY()**. Las coordenadas se expresan en el sistema de coordenadas del componente, no de la pantalla.



UNIDAD 8. Aplicaciones Controladas por Eventos.

El usuario puede mezclar teclas mientras usa el ratón. Las teclas modificadoras pueden ser: Shift, Control, Alt y Meta. Para detectar estas pulsaciones puedes usar los métodos: `evt.isShiftDown()`, `evt.isControlDown()`, `evt.isAltDown()` y `evt.isMetaDown()`.

Para saber cuál de los botones del ratón ha pulsado, o liberado (izquierdo, centro o derecho) puedes ejecutar el método `evt.getButton()`, que devuelve una de estas constantes: `MouseEvent.BUTTON1` (izquierdo), `MouseEvent.BUTTON2` (centro) o `MouseEvent.BUTTON3` (derecho). Para los eventos en que no hay pulsación (entrar y salir) devolverá `MouseEvent.NOBUTTON`. Si se pulsa el botón derecho, se considera true la pulsación de la tecla Meta.

EJEMPLO 10: Usa un `JPanel` para que si haces clic con el botón izquierdo, se dibuje un rectángulo de color rojo centrado en el punto clicado. Si haces clic con el derecho, se dibuja un óvalo azul en la posición donde has hecho clic. Si pulsas la tecla mayúscula cuando haces clic, se borrará toda la pantalla.

En el código, es el propio panel el que responde a sus eventos, el panel es un mouse listener.

```
package eventos;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/**
 * Una demo de eventos del ratón: MouseEvents. Se dibujan
 * figuras sobre un fondo negro cuando el usuario hace clic
 * en el panel. Rectángulos rojos si usa el botón izquierdo
 * óvalos azules si usa el derecho o barra la pantalla si
 * pulsa la tecla mayúscula. Así programado el dibujo
 * se pierde si se redibuja (cambia de tamaño, etc)
 */
public class SimpleMouse extends JPanel implements MouseListener {
    public static void main(String[] args) {
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
JFrame window = new JFrame("Simple Mouse");
SimpleSMouse content = new SimpleMouse();
window.setContentPane(content);
window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
window.setLocation(120,70);
window.setSize(450,350);
window.setVisible(true);
}
//-----
/**
 * constructor que pone fondo negro y registra el listener
 * con sigo mismo.
 */
public SimpleMouse() {
    setBackground(Color.BLACK);
    addMouseListener(this);
}
/**
 * Detectar pulsaciones
 */
public void mousePressed(MouseEvent evt) {
    if ( evt.isShiftDown() ) {
        repaint();
        return;
    }
    int x = evt.getX();
    int y = evt.getY();
    Graphics g = getGraphics(); // Graphics del panel para dibujar
    if ( evt.isMetaDown() ) { // botón derecho
        g.setColor(Color.BLUE); // interior azul
        g.fillOval( x - 30, y - 15, 60, 30 );
        g.setColor(Color.BLACK); // El borde negro
        g.drawOval( x - 30, y - 15, 60, 30 );
    }
    else { // botón centro o izquierdo
        g.setColor(Color.RED);
        g.fillRect( x - 30, y - 15, 60, 30 );
        g.setColor(Color.BLACK);
        g.drawRect( x - 30, y - 15, 60, 30 );
    }
    g.dispose(); // ya no usamos el contexto
} // mousePressed

// Los siguientes 4 para cumplir con la interfaz
public void mouseEntered(MouseEvent evt) { }
public void mouseExited(MouseEvent evt) { }
public void mouseClicked(MouseEvent evt) { }
public void mouseReleased(MouseEvent evt) { }
} // clase
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

Nota: violamos la regla de dibujar solo dentro del método `paintComponent()` del componente. Las consecuencias son tener que fabricarnos y liberar un contexto gráfico para dibujar y que perdemos lo dibujado ante cualquier redibujado automático. Prueba a mover una ventana sobre la de la aplicación.

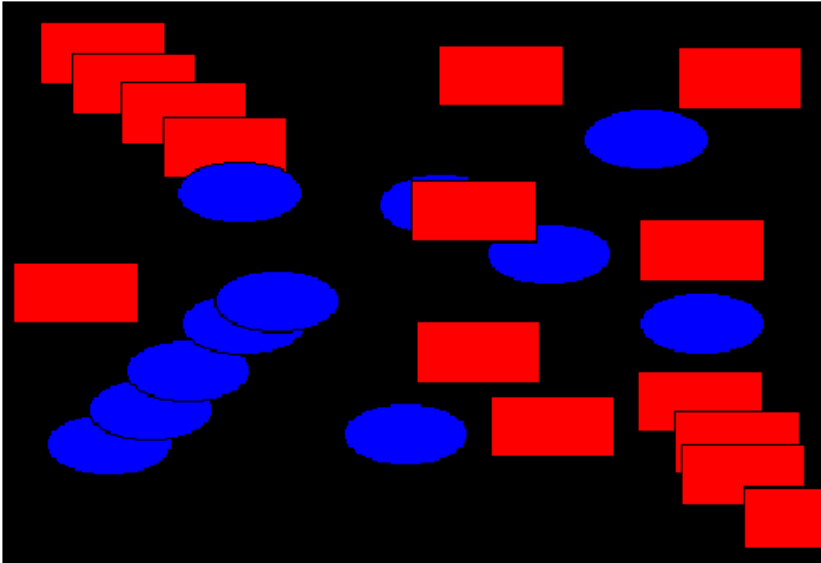


Figura 10: Ejecución del programa de ejemplo.

MOVER Y ARRASTRAR

Mientras el ratón se mueve genera eventos que el SO detecta y los usa para mover el cursor por la pantalla. Un programa también puede escucharlos y responder a ellos. El motivo más usado es implementar **drag and drop** (arrastrar y soltar).

Los métodos para responder al movimiento del ratón se definen en la interfaz **MouseListener**. Define dos métodos:



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
public void mouseDragged(MouseEvent evt);  
public void mouseMoved(MouseEvent evt);
```

El método **mouseDragged()** es llamado si el ratón se mueve mientras uno de sus botones esta pulsado. Si se mueve cuando no hay botones pulsados se llama a **mouseMoved()**. El parámetro evt es un objeto **MouseEvent** y contiene los datos que ya hemos comentado. Como se generan muchos, puede ser ineficiente escucharlos si no vas a hacer nada con ellos, por eso se definen en una interfaz separada del resto de eventos del ratón.

Si quieres que tu programa los escuche, debes tener un objeto que implemente la interfaz **MouseMotionListener** y registrar el objeto con los componentes que te interesen llamando al método **addMouseMotionListener()**. Muchas veces ese objeto también implementará **MouseListener**.

Para implementar dragging debes utilizar 3 eventos: **mousePressed()**, **mouseDragged()** y **mouseReleased()**. La operación comienza con la pulsación de un botón, continuará mientras el ratón es arrastrado y acabará cuando el botón se libere. El método **mouseDragged()** se llamará muchas veces, así que tendrás que ayudarte de variables de instancia para recordar por ejemplo las coordenadas donde comenzó toda la operación, o la vez anterior que el método se llamó (antX y antY) de tipo **int**. También se utiliza a menudo una variable booleana que indique cuando estás procesando una operación de dragging. Es necesaria porque no todas las llamadas a **mousePressed()** significan el inicio de una operación de arrastre, así que en los métodos **mouseDragged()** y **mouseReleased()** esa variable te ayuda a saber si estás procesando una operación de arrastre o es una falsa alarma.

EJEMPLO 11: Esqueleto que implementa dragging.



UNIDAD 8. Aplicaciones Controladas por Eventos.

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

public class Drag implements MouseListener, MouseMotionListener {
    private int iniX, iniY;        // Donde comienza el drag&drop
    private int antX, antY;        // Posición anterior
    private boolean dragging;      // true si está arrastrando
    // otras variables que se necesiten...

    public void mousePressed(MouseEvent evt) {
        if ( queremos_comenzar_arrastre ) {
            dragging = true;
            iniX = evt.getX(); // Recordar donde comienza
            iniY = evt.getY();
            antX = iniX;       // Recordar recientes
            antY = iniY;
            // Otras operaciones...
        }
    }

    public void mouseDragged(MouseEvent evt) {
        if( dragging == false )    // Comprobar si hay arrastre
            return;                // un falso dragging
        int x = evt.getX();        // Posición actual
        int y = evt.getY();
        // ...Procesar movimiento desde (antX,antY) hasta (x,y)...
        antX = x;                  // Actualizar pos. anterior
        antY = y;
    }

    public void mouseReleased(MouseEvent evt) {
        if( dragging == false )    // comprobar si hay operación
            return;                // No hay
        dragging = false;         // Se acaba el dragging (soltar)
        // ...Otro procesamiento (comprobar donde sueltas, etc.) ...
    }
}
```

EJEMPLO 12: vamos a hacer un programa de dibujo con el ratón. Esto exige implementar dragging.

Las coordenadas que se usan son las que pueda tener el panel, obtenidas mediante `getWidth()` y `getHeight()`. Redimensionar complica más las cosas. El área de dibujo se extiende desde `y = 3` hasta `y = alto - 3` verticalmente y desde `x = 3` hasta `x = ancho - 56` horizontalmente.



UNIDAD 8. Aplicaciones Controladas por Eventos.

Estos números se necesitan para interpretar de forma correcta el significado de un clic. Porque hay un borde gris alrededor de los cuadros que permiten elegir el color de dibujo. Este borde tiene 3 pixels de ancho y los rectángulos de color tienen 50 x 50 pixels, que sumados a los 3 de cada lado nos dan 56 pixels de ancho desde el borde derecho del panel y el alto varía con el tamaño de la ventana.



Figura 11: Ejecución del programa de ejemplo 2.

El cuadro blanco etiquetado con "CLEAR" es el último cuadro, así que tendremos una región con 7 cuadrados de 50 x algo y 2 bordes grises en cada lado. Para saber cuando se clicke en uno de los cuadrados hay que hacer cálculos. Pero si la ventana puede crecer y quieres ocupar toda la zona vertical disponible, las cuentas hay que cambiarlas. Para evitarlo, lo que haremos es que aumenten de tamaño todos los cuadros



UNIDAD 8. Aplicaciones Controladas por Eventos.

salvo el cuadro blanco de borrar, así simplificamos cálculos.

La altura de cada rectángulo será: $\text{colorEspacio} = \text{alto} - 56$. El tope del rectángulo n-ésimo está en $(N * \text{colorEspacio} + 3)$ pixels debajo del tope del panel (el primer panel es el 0). Para dibujar el N-ésimo rectángulo con esta sentencia:

```
g.fillRect(width - 53, N * colorEspacio + 3, 50, colorEspacio -3);
```

El ratón hace 3 cosas: elige un color, borra el dibujo y dibuja una curva. Solamente la última obliga a implementar dragging. Se decide cuando hay dragging en el método **mousePressed()** comprobando donde se ha hecho clic, la (x,y) le dirá si hay o no dragging. Si ha clicado en CLEAR se borra la pantalla y si clicla en un color se selecciona el color, en otro caso, se inicia una operación de dragging. Para calcular en qué botón ha hecho clic, se divide la coordenada y entre colorEspacio.

```
package pintar;

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

/**
 * @version 1. 2019
 * @author Un señor
 */
public class MiniPaint {

    public static void main(String[] args) {
        EventQueue.invokeLater(
            new Runnable() {
                public void run() {
                    JFrame frame = new Dibujo();
                    frame.setTitle("Ejemplo de dragging");
                    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    frame.setSize(300,240);
                    frame.setVisible(true);
                }
            }
        );
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

}

/**
 * Un frame con un componente
 */
class Dibujo extends JFrame {
    private static final long serialVersionUID = 1L;

    public Dibujo() {
        Componente c1 = new Componente();
        addMouseListener( (MouseListener) c1 );
        addMouseMotionListener( (MouseMotionListener) c1 );
        add(c1); // Añadimos como contentPane del JFrame a c1
    }
}

/**
 * Un componente que permite dibujar en el
 */
class Componente extends JComponent implements MouseListener,
MouseMotionListener {
    private static final long serialVersionUID = 1L;
    private static final int DEFAULT_WIDTH = 300;
    private static final int DEFAULT_HEIGHT = 200;
    public int iAncho, iAlto, antX, antY, colorEspacio;
    public boolean dragging= false;
    public Graphics g1;
    public Color colorActual = Color.BLACK;

    public void paintComponent(Graphics g) {
        iAncho = getWidth();
        iAlto = getHeight();
        colorEspacio = (iAlto - 53) / 7;
        // Dibujar el marco
        g.setColor(Color.GRAY);
        g.drawRect(0, 0, iAncho-1, iAlto-1);
        g.drawRect(1, 1, iAncho-3, iAlto-3);
        g.drawRect(2, 2, iAncho-5, iAlto-5);
        g.fillRect(iAncho - 56, 0, 56, iAlto);
        // dibujar los cuadrados con los colores
        for(int i= 0; i< 7; i++) {
            switch(i) {
                case 0: g.setColor(Color.BLACK);
                        break;
                case 1: g.setColor(Color.RED);
                        break;
                case 2: g.setColor(Color.GREEN);
                        break;
                case 3: g.setColor(Color.BLUE);

```




UNIDAD 8. Aplicaciones Controladas por Eventos.

```
        break;
    case 4: g.setColor(Color.CYAN);
        break;
    case 5: g.setColor(Color.PINK);
        break;
    case 6: g.setColor(Color.YELLOW);
        break;
    }
    g.fillRect(iAncho-53, i * colorEspacio + 3, 50, colorEspacio-3);
}
g.setColor(Color.WHITE);
g.fillRect(iAncho - 53, 7 * colorEspacio + 3, 50, 50);
g.setColor(Color.BLACK);
g.drawString("BORRA", iAncho- 50 , 7 * colorEspacio + 28);
}

public Dimension getPreferredSize() {
    return new Dimension(DEFAULT_WIDTH, DEFAULT_HEIGHT);
}

public void cambiaColor(int x, int y) {
    if( x > iAncho - 53) { // Elige color o borrar
        int color = (y - 21) / colorEspacio;
        switch(color) {
            case 0: colorActual= Color.BLACK;
                break;
            case 1: colorActual= Color.RED;
                break;
            case 2: colorActual= Color.GREEN;
                break;
            case 3: colorActual= Color.BLUE;
                break;
            case 4: colorActual= Color.CYAN;
                break;
            case 5: colorActual= Color.PINK;
                break;
            case 6: colorActual= Color.YELLOW;
                break;
            case 7: repaint();
                break;
            default: colorActual = Color.BLACK;
        }
    }
    else {
        dragging= true;
        antX = x;
        antY = y;
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

public void mousePressed(MouseEvent evt) {
    int x = evt.getX();
    int y = evt.getY();
    cambiaColor(x,y);
} // mousePressed

public void mouseReleased(MouseEvent evt) {
    if( dragging == false ) // comprobar si hay operación
        return;           // No hay
    dragging = false;       // Se acaba el dragging
    if(g1 != null) g1.dispose();
    g1 = null;
}

public void mouseDragged(MouseEvent evt) {
    if( dragging == false ) return; // Comprobar si hay arrastre
    int x = evt.getX();             // Posición actual
    int y = evt.getY();
    // Procesar movimiento desde (antX,antY) hasta (x,y)
    if(g1 == null) {
        g1= getGraphics();
        g1.setColor(colorActual);
    }
    // No marranear los colores ni los bordes
    if(x > iAncho-56) x = iAncho-56; else if (iAncho < 3) iAncho= 3;
    if( y <3) y=3; else if( y > iAlto-3) y= iAlto-3;
    g1.drawLine(antX , antY, x, y);
    antX = x; // Actualizar pos. anterior
    antY = y;
} // mouseDragged

public void mouseClicked(MouseEvent evt) { }
public void mouseEntered(MouseEvent evt) { }
public void mouseExited(MouseEvent evt) { }
public void mouseMoved(MouseEvent evt) { } // la otra interfaz
}

```

MANEJADORES ANÓNIMOS Y CLASES ADAPTADORAS

Es muy común usar **clases internas anónimas** para definir listeners. Para crear un objeto listener:

```

new MouseListener() {
    public void mousePressed(MouseEvent evt) { . . . }
    public void mouseReleased(MouseEvent evt) { . . . }
}

```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
public void mouseClicked(MouseEvent evt) { . . . }  
public void mouseEntered(MouseEvent evt) { . . . }  
public void mouseExited(MouseEvent evt) { . . . }  
}
```

Esta sentencia hace las dos cosas (define una clase anónima y crea el objeto). Si esto se pasa como parámetro en la llamada al método **addMouseListener()** se registra como el listener del componente:

```
component.addMouseListener(  
    new MouseListener() {  
        public void mousePressed(MouseEvent evt) { . . . }  
        public void mouseReleased(MouseEvent evt) { . . . }  
        public void mouseClicked(MouseEvent evt) { . . . }  
        public void mouseEntered(MouseEvent evt) { . . . }  
        public void mouseExited(MouseEvent evt) { . . . }  
    }  
);
```

Para evitar la tediosa tarea de definir todos los métodos de la interfaz, Java tiene **clases adaptadoras**. Una clase adaptadora implementa la interfaz de un listener aportando definiciones vacías de los métodos de la interfaz. Para usarla debes hacer una subclase suya y en la subclase defines solo los métodos que vas a usar, estando el resto ya definidos en la clase.

La clase adaptadora **MouseAdapter** implementa tanto la interfaz **MouseListener** como **MouseMotionListener**. Por ejemplo, si quieres un listener que solo responda al evento de pulsación:

```
component.addMouseListener(  
    new MouseAdapter() {  
        public void mousePressed(MouseEvent evt) { . . . }  
    }  
);
```

EJEMPLO 13: Para ver como funciona en un ejemplo real, este código:

```
import java.awt.Component;
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.JFrame;

public class Ejemplo {
    public static void main(String[] args) {
        JFrame window = new JFrame("Strings Aleatorios");
        RandomString content = new RandomString();
        content.addMouseListener(
            new MouseAdapter() {
                public void mousePressed(MouseEvent evt) {
                    Component origen = (Component)evt.getSource();
                    origen.repaint();
                }
            }
        );
        window.setContentPane(content);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setLocation(100,75);
        window.setSize(300,240);
        window.setVisible(true);
    }
}
```

8.3.2 EVENTOS DE TIMER.

No todos los eventos los genera el usuario. Los eventos también los pueden generar objetos que se programan para ello, y sus eventos los aprovechan otros. Un ejemplo es la clase **javax.swing.Timer**. Un **Timer** genera eventos a intervalos regulares. Pueden utilizarse para controlar una animación o realizar otra tarea a intervalos regulares de tiempo.

TIMERS Y ANIMACIONES

Un objeto **Timer**, por defecto genera una secuencia de eventos con un intervalo fijo de tiempo entre ellos (también es posible que genere un único evento transcurrido cierto tiempo, en este caso es como una alarma). Cada evento pertenece a la clase **ActionEvent**. Un objeto que quiera escuchar estos eventos debe implementar la interfaz **ActionListener**, que solo tiene un método:



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
public void actionPerformed(ActionEvent evt);
```

Como no tiene sentido tener un **Timer** sin un listener que lo escuche, el listener se indica en el constructor del Timer como un parámetro además del intervalo de tiempo:

```
timer = new Timer( milisegundos_retardo, listener );
```

El tiempo es un **int** y el tipo del listener es **ActionListener**. El método **actionPerformed()** del listener se ejecutará. Para que comience a generar eventos, debes llamar a su método **start()** y si quieres que deje de enviar eventos llamas al método **stop()**.

EJEMPLO 14: Usar un Timer para controlar una animación. El programa irá mostrando varias imágenes cada cierto tiempo. Muestra imágenes creadas al azar cada vez que se llama al método **paintComponent()** y responde al evento llamando a **repaint()**:

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

public class Arte extends JPanel {
    /**
     * Redibuja el panel controlado por un Timer cada 4 segundos
     */
    private class Repinta implements ActionListener {
        public void actionPerformed(ActionEvent evt) {
            repaint(); // Llamar a repintar
        }
    }
    /**
     * El constructor crea el Timer
     */
    public Arte() {
        Repinta accion = new Repinta();
        Timer timer = new Timer(4000, accion);
        timer.start();
    }
    /**
     * El método paintComponent() pinta el panel
     */
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

public void paintComponent(Graphics g) {
    Color c = Color.getHSBColor(1.0F, 0.0F, (float)Math.random());
    g.setColor(c);
    g.fillRect(0, 0, getWidth(), getHeight());
    int arte = (int)(3 * Math.random());
    switch (arte) {
        case 0: for(int i = 0; i < 500; i++) {
            int x1 = (int)(getWidth() * Math.random());
            int y1 = (int)(getHeight() * Math.random());
            int x2 = (int)(getWidth() * Math.random());
            int y2 = (int)(getHeight() * Math.random());
            Color cH = Color.getHSBColor(
                (float)Math.random(), 1.0F, 1.0F);
            g.setColor(cH);
            g.drawLine(x1,y1,x2,y2);
        }
        break;
        case 1: for(int i = 0; i < 200; i++) {
            int centerX = (int)(getWidth() * Math.random());
            int centerY = (int)(getHeight() * Math.random());
            Color cH = Color.getHSBColor((float)Math.random(),
                1.0F, 1.0F);
            g.setColor(cH);
            g.drawOval(centerX - 50, centerY - 50, 100, 100);
        }
        break;
        case 2: for(int i = 0; i < 25; i++) {
            int centerX = (int)(getWidth() * Math.random());
            int centerY = (int)(getHeight() * Math.random());
            int size = 30 + (int)(170*Math.random());
            Color cC = new Color( (int)(256*Math.random()),
                (int)(256*Math.random()),
                (int)(256*Math.random()) );
            g.setColor(cC);
            g.fillRect(centerX - size/2, centerY - size/2,
                size, size, true);
        }
        break;
    }
}
}

```

Observa que la clase Repinta es una clara candidata a convertirse en anónima interna (y nos ahorramos definir la clase):

```

Timer timer = new Timer(4000,
    new ActionListener() {
        public void actionPerformed(ActionEvent evt){
            repaint();
        }
    }
);

```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
    }  
  }  
);
```

8.3.3. EVENTOS DEL TECLADO.

En Java, las acciones del usuario generan eventos asociados a un componente de la GUI. ¿Pero qué pasa con el teclado? Igual, cuando pulsa una tecla, el evento generado se asocia con un componente ¿Con cuál? La GUI usa la idea de un foco (simil del teatro cuando en una escena se le quiere dar protagonismo a uno de los personajes, se le pone un foco que lo ilumina). En toda GUI, uno de los componentes tiene ese foco. A ese componente se le asocian los eventos del teclado.

Es buena idea que la GUI de al usuario feedback para que el usuario sepa cuál es el componente con el foco. Por ejemplo, si el componente es un área capaz de editar textos, el cursor parpadeando es el indicador del foco. A otros componentes se les dibuja un borde que lo rodea, para destacarlos de alguna forma visual.

Si **comp** es un componente y quieres que tenga el foco, puedes llamar al método **comp.requestFocusInWindow()**, y si la ventana donde está comp no es la activa, debes usar el método **requestFocusInWindow()**.

Cuando el usuario hace clic sobre un componente, el componente normalmente se queda con el foco. Si pulsa la tecla **tab** se mueve el foco de un componente a otro (esto es automático, no necesita programarse). Sin embargo hay controles que no reciben el foco, hay que forzarlo, por ejemplo los paneles, para ganar el foco al pulsar el ratón debes escuchar el evento y reclamarlo por programa: **mousePressed()** -> **requestFocusInWindow()**.

Los eventos de teclado pertenecen a la clase **KeyEvent**. Un objeto que necesite escuchar eventos **KeyEvent** debe implementar la interfaz



UNIDAD 8. Aplicaciones Controladas por Eventos.

KeyListener. Además el objeto debe estar registrado en el componente con una llamada al método **comp.addKeyListener(listener)**. Donde comp es el componente que genera los eventos de teclado (cuando tiene el foco) y listener el objeto que implementa la interfaz KeyListener. Es posible que componente y listener sean el mismo objeto (se escucha a sí mismo).

La interfaz **KeyListener** declara los siguientes métodos:

```
public void keyPressed(KeyEvent evt);  
public void keyReleased(KeyEvent evt);  
public void keyTyped(KeyEvent evt);
```

Java diferencia entre las teclas que pulsas y los caracteres que tecleas. Hay muchas teclas en el teclado como letras, números que al pulsarlas generan un carácter. Otras son teclas modificadoras como mayúscula, teclas de función, etc. que no están asociadas a caracteres al pulsarlas. Otras veces para generar una letra hay que pulsar más de dos teclas (alt + may + tecla).

Hay 3 tipos de **KeyEvent**. Los tipos se asocian con presionar un tecla, liberarla y teclear un carácter. Si el usuario presiona una tecla se llama a **keyPressed()**, cuando la libera se llama a **keyReleased()** y si se teclea un carácter se llama al método **keyTyped()**. Observa que **pulsar una tecla puede generar de dos a 5 eventos**:

pulsas control -> keyPressed + keyreleased;

pulsas 'E' -> keyPressed de may + keyPressed de 'E' +

keyreleased de 'E' + keyreleased de may + keytyped.

Es práctico separar los eventos en 2 secuencias: la pulsación y liberación por un lado y el tecleo por otro. Detectar el carácter pulsado a partir de pulsaciones y liberaciones es complejo. Detectar si hay una tecla de estado pulsada con keyTyped no es posible. Así que



UNIDAD 8. Aplicaciones Controladas por Eventos.

cada secuencia tiene su utilidad. Suele ocurrir que al presionar una tecla, se genere más de una llamada a `keyPressed()` y solo una llamada a `keyReleased()`. La correspondencia no tiene por qué ser 1 a 1.

Cada tecla en el teclado tiene un código numérico entero. Cuando se llama a los métodos `keyPressed()` o `keyReleased()`, el evento `evt` contiene ese código que puedes obtener llamando al método `evt.getKeyCode()`. En vez de tener que memorizar una table numérica, java aporta constantes simbólicas para cada tecla:

<code>KeyEvent.VK_SHIFT</code>	tecla mayúscula
<code>KeyEvent.VK_LEFT</code>	flecha izquierda
<code>KeyEvent.VK_RIGHT</code>	flecha derecha
<code>KeyEvent.VK_UP</code>	flecha arriba
<code>KeyEvent.VK_DOWN</code>	flecha abajo ...

Las letras VK vienen de "Teclado virtual" porque en realidad cada teclado tiene sus propios códigos, pero Java para unificarlos los traduce a códigos de un teclado virtual (VK).

En el caso del evento del método `keyTyped`, querrás saber el carácter que se ha tecleado. Lo puedes obtener llamando al método `evt.getKeyChar()`.

EJEMPLO 15: Un programa que mueva un rectángulo en las 4 direcciones (Arriba, abajo, izquierda y derecha), cuando el usuario pulse 'R', 'V', 'A' o 'N' cambia el color del cuadrado a rojo, verde, azul o negro respectivamente. Habrá que tener en cuenta que al ser el área de dibujo un `JPanel`, no recibe el foco. Cuando lo tenga, el programa le pone un borde de color cyan.

```
package eventos;

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



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
/**
 * Demo de foco y eventos de teclado.
 * En un panel se dibuja un cuadrado que puede moverse (flechas)
 * y cambiar color (teclas R, G, B, K). El panel no coge el foco.
 * Hay que pasárselo e indicarlo con un recuadro
 */
public class TeclasDemo extends JPanel {

    /**
     * Crea ventana con TeclasDemo */
    public static void main(String[] args) {
        JFrame window = new JFrame("Demo de Teclado y Foco");
        TeclasDemo content = new TeclasDemo();
        window.setContentPane( content );
        window.setSize(400,400);
        window.setLocation(100,100);
        window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        window.setVisible(true);
        content.requestFocusInWindow();
    }

    //-----

    /**
     * Clase anidada para listener de teclado y foco */
    private class Listener implements KeyListener, FocusListener {

        /**
         * El panel gana el foco -> repaint() con borde. */
        public void focusGained(FocusEvent evt) {
            repaint(); // borde de color cyan
        }

        /**
         * El panel pierde foco -> repaint() borde gris y msj */
        public void focusLost(FocusEvent evt) {
            repaint(); // Sin borde cyan
        }

        /**
         * Teclea carácter -> Comprobar si cambia color */
        public void keyTyped(KeyEvent evt) {
            char c = evt.getKeyChar(); // El carácter
            if (c == 'A' || c == 'a') {
                cColor = Color.BLUE;
                repaint();
            }
            else if (c == 'V' || c == 'v') {
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
        cColor = Color.GREEN;
        repaint();
    }
    else if (c == 'R' || c == 'r') {
        cColor = Color.RED;
        repaint();
    }
    else if (c == 'N' || cc == 'n') {
        cColor = Color.BLACK;
        repaint();
    }
} // keyTyped()

/**
 * Presiona tecla -> mover panel sin salirse */
public void keyPressed(KeyEvent evt) {
    int k = evt.getKeyCode(); // código de tecla
    if(k == KeyEvent.VK_LEFT) { // izquierda
        cIzq -= 8;
        if (cIzq < 3) cIzq = 3;
        repaint();
    }
    else if (k == KeyEvent.VK_RIGHT) { // derecha
        cIzq += 8;
        if(cIzq > getWidth() - 3 - ANCHO_CUADRADO)
            cIzq = getWidth() - 3 - ANCHO_CUADRADO;
        repaint();
    }
    else if(k == KeyEvent.VK_UP) { // arriba
        cTope -= 8;
        if (cTope < 3) cTope = 3;
        repaint();
    }
    else if (k == KeyEvent.VK_DOWN) { // abajo
        cTope += 8;
        if(cTope > getHeight() - 3 - ANCHO_CUADRADO)
            cTope = getHeight() - 3 - ANCHO_CUADRADO;
        repaint();
    }
} // keyPressed()

    public void keyReleased(KeyEvent evt) { } // Necesaria
} // clase Listener

// -----

private static final int ANCHO_CUADRADO = 50; // longitud
private Color cColor; // Color del cuadrado
private int cTope, cIzq; // Coordenadas esquina sup. izq.
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
/**
 * El constructor fija posición inicial, color y foco */
public TeclasDemo() {
    cTop = 100; // posición inicial
    cIzq = 100;
    cColor = Color.RED; // color inicial
    setBackground(Color.WHITE);
    Listener listener= new Listener(); // listener tecl y foco
    addKeyListener(listener); // escuchar teclado
    addFocusListener(listener); // escuchar foco
} // constructor

/**
 * Dibujar */
public void paintComponent(Graphics g) {
    super.paintComponent(g); // panel, hay rellenar
    // borde
    if (hasFocus())
        g.setColor(Color.CYAN);
    else
        g.setColor(Color.LIGHT_GRAY);
    int ancho = getSize().width; // anchura del panel
    int alto = getSize().height; // altura del panel
    g.drawRect(0,0,ancho-1,alto-1);
    g.drawRect(1,1,ancho-3,alto-3);
    g.drawRect(2,2,ancho-5,alto-5);
    // Dibuja cuadrado
    g.setColor(cColor);
    g.fillRect(cIzq, cTope, ANCHO_CUADRADO, ANCHO_CUADRADO);
    // Imprimir mensaje
    g.setColor(Color.MAGENTA);
    if (hasFocus()) {
        g.drawString("Flechas para mover",7,20);
        g.drawString("K, R, G, B Cambian color",7,40);
    }
    else
        g.drawString("Clic para activar",7,20);
} // fin paint

} // clase
```

8.3.4 EVENTOS DE COMPONENTES.

Los componentes pueden generar varios tipos de eventos. Cuando un componente genera un evento, por ejemplo un botón se pulsa, suele generar un evento de tipo `ActionEvent`, aunque realmente hay más:



UNIDAD 8. Aplicaciones Controladas por Eventos.

Tipo Evento	Listener	Cuándo ocurre
ActionEvent	ActionListener	Al pulsar un botón o elegir un elemento de una lista o un menú
AdjustmentEvent	AdjustmentListener	Emitido por un Scrollbar
ComponentEvent	ComponentListener	Al mover un componente o cambiar su tamaño o su visibilidad
ContainerEvent	ContainerListener	Se añaden más componentes o se eliminan
ItemEvent	ItemListener	Cuando un elemento es seleccionado o no
TextEvent	TextListener	Cambios en el texto de un objeto
WindowEvent	WindowListener	Cuando una ventana cambia su estado

En el caso de los ActionEvent, se gestionan en un ActionListener, que te obligan a implementar el método:

```
public void actionPerformed(ActionEvent e);
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

EJEMPLO 16: Identificar el componente que genera el evento porque varios componentes comparten el mismo manejador de eventos.

```
public class BotonPulsado extends JFrame {
    private static final long serialVersionUID = 1L;
    private JLabel etiqueta;
    private JButton b1, b2, b3;

    public BotonPulsado(){
        super("¿Qué botón se Pulsa?");
        getContentPane().setLayout( new FlowLayout() );
        b1 = new JButton("Botón 1");
        b2 = new JButton("Botón 2");
        b3 = new JButton("Botón 3");
        etiqueta = new JLabel("");
        add(b1);
        add(b2);
        add(b3);
        add(etiqueta);
        ActionListener aL = new BotonPulsadoListener();
        b1.addActionListener(aL);
        b2.addActionListener(aL);
        b3.addActionListener(aL);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        setSize(400,300);
        setVisible(true);
    }

    private class BotonPulsadoListener implements ActionListener{
        @Override
        public void actionPerformed(ActionEvent e) {
            etiqueta.setText("Ha pulsado el botón " +
                            e.getActionCommand() );
        }
    }

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



UNIDAD 8. Aplicaciones Controladas por Eventos.

8.3.5. EVENTOS DEL FOCO.

En Java, los objetos son notificados cuando cambia el foco con eventos de tipo **FocusEvent**. Un objeto que quiera escuchar los cambios debe implementar la interfaz **FocusListener**. Esta interfaz declara dos métodos:

```
public void focusGained(FocusEvent evt);  
public void focusLost(FocusEvent evt);
```

Además debes usar el método **addFocusListener(Listener)** para asociar el objeto Listener con el componente. Cuando un componente gana el foco, llama al método **focusGained()** del Listener registrado y cuando lo pierde llama a **focusLost()**. Un componente puede comprobar si tiene el foco llamando a la función **hasFocus()**.

EJEMPLO 17: Haremos un juego en el que un submarino negro debe ser atacado por un barco azul con bombas de color rojo, el jugador debe alcanzar al submarino moviéndose a derecha e izquierda. Así usaremos un poco todo lo que hemos visto sobre eventos. Comentamos algunas cosas antes del listado.

Las flechas izquierda y derecha mueven el barco y sueltas la carga de profundidad pulsando la flecha abajo. Si la carga toca al submarino lo hundes. Si la carga llega a la parte baja de la pantalla, aparece una nueva pegada al barco.

Tanto el barco, como el submarino y la carga de profundidad son objetos. Cada uno definido en una clase anidada dentro de la clase del panel donde se dibuja. Las variables bar, bom y sub referencian los objetos de estas clases.



UNIDAD 8. Aplicaciones Controladas por Eventos.

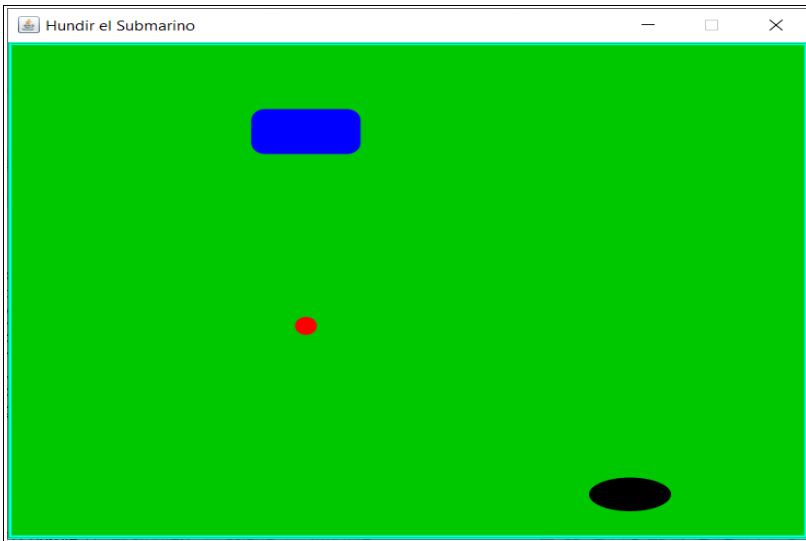


Figura 14: Imagen del juego (Destacan los gráficos, ¿verdad? ;-).

¿Cuál es el estado del programa? ¿Qué cambios hay de un tiempo a otro en la apariencia o el comportamiento del programa?

- Las posiciones del barco, el submarino y la bomba $\rightarrow (x,y)$.
- A veces la bomba acierta y otras no \rightarrow booleana cayendo.
- El submarino va hacia izq o der. \rightarrow booleana aIzquierda
- A veces el submarino explota \rightarrow booleana explotando.
- Una explosión es como una animación, necesita varios frames. Mientras dura la explosión el sub parece diferente en cada frame y el tamaño de la explosión aumenta. Una variable entera en el submarino indica el nº de frame a mostrar en la animación \rightarrow sub.exploNumFram.

¿Cómo cambia el estado? Algunas variables cambiarán automáticamente porque la acción es controlada por un Timer (la posición del submarino). El barco y la bomba tendrán un método



UNIDAD 8. Aplicaciones Controladas por Eventos.

llamado **actualizaEstado()** que modifica el estado para el siguiente frame. El action listener llamará a:

```
bar.actualiza();  
bom.actualiza();  
sub.actualiza();
```

Y luego a **repaint()**, para que refleje el estado en el dibujo. Si la bomba está cayendo, su posición y debe variar, si la bomba impacta en el submarino las variable de sub explotando se pone a true y cayendo a false. Cada cierto tiempo el submarino cambia de dirección su movimiento, cambia aIzquierda. Su método de actualización tiene estas líneas donde cambia la dirección al azar:

```
if ( Math.random() < 0.04 )  
    aIzquierda = ! aIzquierda; // 1/25 de probabilidad
```

Además de estos cambios automáticos, también las pulsaciones de teclas cambian el estado de este mundo virtual. El método que se encarga de escuchar estas pulsaciones es:

```
public void keyPressed(KeyEvent evt) {  
    ...
```

Lee el código e intenta comprender como funciona y hunde ese molesto submarino...

```
package eventos;  
  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
/**  
 * Hundir el submarino. Resumen de eventos */  
public class Hundir extends JPanel {  
    private static final long serialVersionUID = 1L;  
  
    public static void main(String[] args) {
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

JFrame window = new JFrame("Hundir el Submarino");
Hundir content = new Hundir();
window.setContentPane(content);
window.setSize(600, 480);
window.setLocation(100,100);
window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
window.setResizable(false); // Usuario no cambia tamaño
window.setVisible(true);
}

//-----
private Timer timer; // Timer que controla la animación
private int ancho, alto; // Tamaño del panel
private Barco bar; // bar,bomb,sub: objetos del juego
private Bomba bom;
private Submarino sub;

/**
 * constructor -> fondo, timer y listeners
 */
public Hundir() {
    setBackground( new Color(0, 200, 0) );
    ActionListener accion = new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            if (bar != null) {
                bar.actualiza();
                bom.actualiza();
                sub.actualiza();
            }
            repaint();
        }
    };
    timer = new Timer(30, accion); // cada 30 ms -> 33/segundo
    addMouseListener(
        new MouseAdapter() { // pide el foco
            public void mousePressed(MouseEvent evt){
                requestFocus();
            }
        }
    );
    addFocusListener(
        new FocusListener() {
            public void focusGained(FocusEvent evt) {
                timer.start();
                repaint();
            }
            public void focusLost(FocusEvent evt) {
                timer.stop();
                repaint();
            }
        }
    );
}

```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

    }
}
);
addKeyListener(
    new KeyAdapter() {
        public void keyPressed(KeyEvent evt) {
            int k = evt.getKeyCode(); // tecla pulsada
            if (k == KeyEvent.VK_LEFT) { // Barco a izq.
                bar.centroX -= 15;
            }
            else if (k == KeyEvent.VK_RIGHT) { // Barco derecha
                bar.centroX += 15;
            }
            else if (k == KeyEvent.VK_DOWN) { // Suelta bomba
                if ( bom.cayendo == false )
                    bom.cayendo = true;
            }
        }
    }
);
} // constructor

/**
 * Dibujar el estado
 */
public void paintComponent(Graphics g) {
    super.paintComponent(g); // rellenar color
    Graphics2D g2 = (Graphics2D)g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
    if (bar == null) {
        ancho = getWidth();
        alto = getHeight();
        bar = new Barco();
        sub = new Submarino();
        bom = new Bomba();
    }
    if ( hasFocus() )
        g.setColor(Color.CYAN);
    else {
        g.setColor(Color.BLACK);
        g.drawString("CLIC PARA ACTIVAR", 20, 30);
        g.setColor(Color.GRAY);
    }
    g.drawRect(0,0,ancho-1,alto-1); // borde
    g.drawRect(1,1,ancho-3,alto-3);
    g.drawRect(2,2,ancho-5,alto-5);
    // Dibujar elementos de la escena
    bar.draw(g);

```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
        sub.draw(g);
        bom.draw(g);
    } // paintComponent()

    /**
     * Definir el barco
     */
    private class Barco {
        int centroX, centroY; // Posición del centro del barco
        Barco() { // Constructor
            centroX = ancho/2;
            centroY = 80;
        }
        void actualiza() {
            if (centroX < 0)
                centroX = 0;
            else if (centroX > ancho)
                centroX = ancho;
        }
        void draw(Graphics g) {
            g.setColor(Color.BLUE);
            g.fillRoundRect(centroX-40, centroY-20, 80, 40, 20, 20);
        }
    } // Barco

    /**
     * Define la bomba
     */
    private class Bomba {
        int centroX, centroY; // Posición
        boolean cayendo; // true, si está soltada
        Bomba() { // Constructor
            cayendo = false;
        }
        void actualiza() {
            if (cayendo) {
                if (centroY > alto) { // No ha dado al submarino
                    cayendo = false;
                }
                else if (Math.abs(centroX - sub.centroX) <= 36 &&
                    Math.abs(centroY - sub.centroY) <= 21) { //Blanco
                    sub.explotando = true;
                    sub.numFrame = 1;
                    cayendo = false;
                }
                else { // se mueve 10 pixels.
                    centroY += 10;
                }
            }
        }
    }
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
    }
}

void draw(Graphics g) { // Dibujar la bomba
    if ( ! cayendo ) {
        centroX = bar.centroX;
        centroY = bar.centroY + 23;
    }
    g.setColor(Color.RED);
    g.fillOval(centroX - 8, centroY - 8, 16, 16);
}

} // Bomba

/**
 * Submarino
 */
private class Submarino {
    int centroX, centroY; // Posición
    boolean aIzquierda; // true si se mueve hacia izquierda
    boolean explotando; // Está explotando
    int numFrame; // Nº de frame de explosión
    Submarino() { // Constructor
        centroX = (int)(ancho*Math.random());
        centroY = alto - 40;
        explotando = false;
        aIzquierda = (Math.random() < 0.5);
    }
    void actualiza() {
        if (explotando) { // Se necesitan 15 frames
            numFrame++;
            if (numFrame == 15) {
                centroX = (int)(ancho*Math.random());
                centroY = alto - 40;
                explotando = false;
                aIzquierda = (Math.random() < 0.5);
            }
        }
        else { // Moverse
            if (Math.random() < 0.04) {
                aIzquierda = ! aIzquierda;
            }
            if (aIzquierda) { // 5 pixels a la izquierda
                centroX-= 5;
                if (centroX <= 0) {
                    centroX = 0;
                    aIzquierda= false;
                }
            }
        }
        else {
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

        centroX+= 5;
        if(centroX > ancho) {
            centroX = ancho;
            aIzquierda= true;
        }
    }
}

void draw(Graphics g) {
    g.setColor(Color.BLACK);
    g.fillOval(centroX - 30, centroY - 15, 60, 30);
    if (explotando) {
        g.setColor(Color.YELLOW);
        g.fillOval(centroX-4 * numFrame, centroY-2 * numFrame,
            8 * numFrame, 4 * numFrame);
        g.setColor(Color.RED);
        g.fillOval(centroX - 2 * numFrame, centroY - numFrame/2,
            4 * numFrame, numFrame);
    }
}
} // Submarino
} // Hundir

```

8.4. COMPONENTES DE LA GUI.

Un programa GUI usa componentes estándar como botones, barras de desplazamiento (scroll bars), cajas de entrada de texto, menús, etc. Estos componentes ya están preparados para que los uses: saben dibujarse a sí mismos, saben manejar los detalles del procesamiento de eventos del ratón y del teclado que les afecte, etc. También son capaces de generar eventos de la clase `java.awt.event.ActionEvent`.

Los componentes son subclases de **JComponent**, vamos a mirar **algunos de los métodos que todos comparten**:

- `c.getWidth()` y `c.getHeight()` devuelven el tamaño del componente (no debes usarlos en el constructor, si no cuando estén dibujados).
- `c.setEnabled(b)` y `c.isEnabled()` activa/desactiva o consulta el



UNIDAD 8. Aplicaciones Controladas por Eventos.

estado del componente. Al desactivarlo cambia su apariencia y el usuario no puede interactuar con él.

- **c.setVisible(b)** muestra/oculta al componente.
- **c.setFont(font)** cambia la fuente usada.
- **c.setBackground(color)** y **c.setForeground(color)** fija sus colores de fondo y dibujo.
- **c.setOpaque(b)** por defecto, solo los JLabel no son opacos. El color de fondo lo herdan de su contenedor.
- **c.setToolTipText(string)** fija el string indicado como nota explicativa "tool tip" del componente (se muestra al dejar el puntero del ratón sobre el componente unos segundos sin moverlo).
- **c.setPreferredSize(size)** fija el tamaño con el que debería mostrarse. El parámetro es un objeto **java.awt.Dimension**, que tiene dos variables de nombre **width** y **height**. Las llamadas suelen ser "**setPreferredSize(new Dimension(100,50))**". Lo usan los layout managers pero no siempre lo pueden respetar.

Usar un componente es una tarea de varios pasos: debes crear el objeto con su constructor, añadirlo a un contenedor, crear un listener para oír los eventos que genere y en algunos casos tener una variable con la que referenciarlos para poder hacer cosas desde el programa.

Hay muchos componentes, algunos sencillos, otros complejos. La manera más eficaz de aprenderlos es programar con ellos. Nosotros veremos algunos, pero si alguna vez necesitas alguno en concreto te aconsejo que vayas a libros especializados o la documentación oficial (<https://docs.oracle.com/javase/tutorial/uiswing/components/index.html>)

8.4.1. COMPONENTES BÁSICOS.



UNIDAD 8. Aplicaciones Controladas por Eventos.

ETIQUETAS DE TEXTO (JLabel)

JLabel es el componente más sencillo, su utilidad es mostrar una línea de texto. El texto no puede ser modificado por el usuario pero si por programa. El constructor ya indica el texto a mostrar:

```
JLabel mensaje = new JLabel("Hola mundo!");
```

Otro constructor también indica donde se posiciona el texto, si hay espacio extra. Los valores de alineamiento posibles son: **JLabel.LEFT**, **JLabel.CENTER** y **JLabel.RIGHT**. Ej:

```
JLabel menaje = new JLabel("Hola Mundo!", JLabel.CENTER);
```

Puedes incluir una imagen leyendo un fichero .gif o .jpg y .png y pasarla al constructor del JLabel:

```
ImageIcon img = new ImageIcon("avion.gif");  
JLabel label = new JLabel("Esto vuela", img, JLabel.RIGHT);
```

Puedes cambiar la imagen con **setIcon(img)**; y puedes quitarla si el parámetro es null o puedes mostrarla desactivada con **setDisabledIcon()**. Puedes cambiar el texto con el método **setText(string)**:

```
mensaje.setText("Adiós mundo!");
```

JLabel tiene las propiedades **labelFor** y **displayedMnemonic** que es un carácter que cuando se pulsa junto con la tecla ALT (ejemplo ALT+R), provoca que el componente de **labelFor** reclame el foco y se active. Es como **un atajo de teclado** para ir a un componente etiquetado por un JLabel. Estas propiedades tienen getters y setters. Ejemplo:

```
JLabel msj = new JLabel("Hola mundo!", JLabel.CENTER);  
msj.setForeground(Color.RED);  
msj.setBackground(Color.BLACK);  
msj.setFont(new Font("Serif", Font.BOLD, 18));
```




UNIDAD 8. Aplicaciones Controladas por Eventos.

```
msj.setOpaque(true);
```

BOTONES (JButton)

Elementos de la clase **JButton** que al pulsarlos haciendo clic o con el teclado, generan un evento. Para acceder a más información utiliza el enlace: <https://docs.oracle.com/javase/tutorial/uiswing/components/button.html>

Aspectos básicos a tener en cuenta:

- **Constructores:** Tiene uno que acepta un string con el texto que se va a mostrar. Ej: `bContinuar = new JButton("Continuar")`.
- **Eventos:** al pulsarlo genera un evento **ActionEvent**. Que se envía a un **ActionListener**.
- **Listeners:** Un objeto que escuche los eventos del botón debe implementar la interfaz **ActionListener** que tiene un solo método `public void actionPerformed(ActionEvent evt)`;
- **Registrar los listeners:** El Action Listener debe asociarse con el botón `boton.addActionListener(objActionListener)`
- **Eventos:** En `actionPerformed(evt)` el evento `evt` contiene información que se recupera con `evt.getActionCommand()` y `evt.getSource()`.
- **Métodos del Componente:** Hay muchos métodos (tanto de la clase **JButton** como heredados), tendrás que mirar la documentación de la API siempre. Por ejemplo `boton.setText("Hola")` cambia el texto y `boton.setActionCommand("sgb")` cambia la acción.

EJEMPLO 18: creación de un componente botón y asociarle una acción.

```
JButton bPulsar = new JButton();
ActionListener act = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Swing es potente!!");
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
};  
bpulsar.addActionListener(act);
```

Se puede asociar un icono al botón o mediante el constructor o mediante el método **setIcon()** como un **JLabel**. Podemos asignar iconos para los estados **selected**, **pressed**, **rollover** y **disabled** con los métodos: **setDisabledSelectedIcon()**, **setPressedIcon()**, **setRolloverIcon()**, **setRolloverSelectedIcon()** y **setSelectedIcon()**.

También es posible asociar directamente un atajo de teclado (**mnemonic**) a un botón para pulsarlo llamando al método **boton.setMnemonic('R')**; y si el carácter aparece en el texto del botón, se subraya con **setDisplayMnemonicIndex()**.

JToggleButton

La clase **javax.swing.JToggleButton** es un tipo de botón que implementa la propiedad **selected** (puede estar seleccionado o no) y de ella heredan las clases **JCheckbox** y **JRadioButton**. Puedes consultar el estado con **boton.isSelected()** y cambiarla con **boton.setSelected()**.

ButtonGroup

La clase **javax.swing.ButtonGroup** sirve para agrupar los botones de tipo **JToggleButton**. Al agruparlos, solamente uno puede estar seleccionado al mismo tiempo.

```
package componentes;  
  
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
class Agrupar extends JFrame {  
    public Agrupar() {  
        super("Agrupar botones");  
        getContentPane().setLayout(new FlowLayout());  
        ButtonGroup bG = new ButtonGroup();  
        char ch = (char) ('1' + k);
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

    for(int k = 0; k < 4; k++) {
        JToggleButton b = new JToggleButton("Boton " + ch, k==0);
        b.setMnemonic(ch);
        b.setEnabled(k < 3);
        b.setToolTipText("Este es el botón " + ch);
        b.setIcon(new ImageIcon("bola_negra.gif"));
        b.setSelectedIcon(new ImageIcon("bolaRoja.gif"));
        b.setRolloverIcon(new ImageIcon("bolaVerde.gif"));
        b.setRolloverSelectedIcon(new ImageIcon("bolaAzul.gif"));
        getContentPane().add(b);
        BG.add(b);
    }
    pack();
}

public static void main(String args[] {
    Agrupar frame = new Agrupar();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}
}

```

JCheckBox y JRadioButton

Un **JCheckBox** es un componente con dos estados: seleccionado y no seleccionado. El usuario puede cambiar el estado haciendo clic en él. Tiene una etiqueta que se usa en el constructor:

```
JCheckBox muestraHora = new JCheckBox("Mostrar hora actual");
```

El programa puede leer y cambiar el estado de un **JCheckBox** llamado **cb**, usando los métodos **cb.isSelected()** y **cb.setSelected(boolean)**.

Un checkbox genera un evento de tipo **ActionEvent** cuando el usuario le cambia el estado y puedes detectarlo y responder a ese cambio. Para ello debes crear un listener y registrarlo en el checkbox con el método **cb.addActionListener()** (pero no genera evento si es el programa quien cambia el estado con **setSelected()**). Si quieres cambiar el estado y generar evento usa **cb.doClick()**.

Puedes usar **evt.getSource()** dentro del método **actionPerformed()**



UNIDAD 8. Aplicaciones Controladas por Eventos.

para saber qué elemento (si el mismo Listener escucha a varios checkbox) y responder a ambos desde el mismo listener sin tener que hacer 20 listeners para 20 checkbox. Ejemplo:

```
public void actionPerformed(ActionEvent evt) {  
    Object source = evt.getSource();  
    if(source == cb1) { // El primer checkbox de una GUI  
        boolean estado = cb1.isSelected();  
        // responder...  
    }  
    else if (source == cb2) {  
        boolean estado = cb2.isSelected();  
        // responder...  
    }  
}
```

Otra alternativa es usar **evt.getActionCommand()** para recuperar el comando de la acción asociada con la fuente del evento (para un JCheckBox es su etiqueta). Los checkbox se agrupan como los botones con estado, puedes marcar varios checkbox de un grupo. Ej: (deportes favoritos: Tenis, fútbol y basket).



Los JRadioButton son similares, pero cuando se agrupan solamente uno de ellos puede estar seleccionado. Por eso se usan para representar opciones incompatibles, al contrario de los checkbox. Ej: Sexo: hombre, mujer (no puedes ser las dos cosas a la vez).



En cualquier caso, ambos se usan igual que los togglebutton.

JTextField and JTextArea

JTextField y JTextArea son subclases de la clase abstracta



UNIDAD 8. Aplicaciones Controladas por Eventos.

JTextComponent y por tanto comparten muchas propiedades y métodos. Representan componentes que contienen texto que puede ser modificado por el usuario. Un **JTextField** contiene una sola línea y un **JTextArea** puede tener varias. Pueden estar en modo solo lectura.

El método de instancia **setText(String)** cambia el texto del componente y **getText()** lo recupera (devuelve String). Para cambiar si permites o no modificar el texto al usuario usas **setEditable(boolean)**.

Por defecto no hay separación entre el texto de los componentes y su borde. Con **setMargin(java.awt.Insets)** lo defines. El parámetro son 4 enteros que definen márgenes en pixels para los lados arriba, izquierda, abajo y derecha. Ej:

```
tC.setMargin( new Insets(5,5,5,5) );
```

JTextField tiene un constructor que permite indicar el nº de caracteres visibles en el texto **JTextField(int columnas)**. Eso se usa para calcular la anchura preferida del texto. Otros constructores no lo usan:

```
public JTextField(String contents);  
public JTextField(String contents, int columnas);
```

Constructores de **JTextArea**:

```
public JTextArea()  
public JTextArea(int filas, int columnas)  
public JTextArea(String contents)  
public JTextArea(String contents, int filas, int columnas)
```

La clase **JTextArea** añade algunos métodos más. Por ejemplo: **append(másTexto)**, añade un string al contenido actual (al usar **append()** o **setText()** los saltos de línea usados son el carácter '\n') y **setLineWrap(boolean)**, si es true las líneas muy largas continúan en la siguiente línea del componente.



UNIDAD 8. Aplicaciones Controladas por Eventos.

Si quieres barras de desplazamiento en componentes que tengan mucho texto, debes añadirlas de forma manual (no es automático). Debes poner al componente dentro de otro de la clase **JScrollPane**. Las barras a partir de ahora, aparecen solamente cuando son necesarias. Ejemplo:

```
JTextArea inputArea = new JTextArea();  
JScrollPane scroller = new JScrollPane( inputArea );
```

Cuando el usuario pulsa return dentro de un **JTextField** se genera un evento **ActionEvent** sin embargo en un **JTextArea** no se genera.

La propiedad **lineCount** de un **JTextArea** indica cuantas líneas de texto contiene. (una línea son caracteres hasta llegar a un salto '\n'). Podemos saber la distancia hasta el salto en una línea con el método **getLineEndOffset()**, la distancia con el primer carácter con **getLineStartOffset()**, y el número de línea para un desplazamiento con **getLineOfOffset()**.

Podemos establecer el número de filas y columnas visibles con **setRows()** y **setColumns()**, y podemos averiguarlo con **getRows()** y **getColumns()**.

JPasswordField

Es una subclase de **JTextField** cuya única diferencia es que esconde el texto que se escribe mostrando asteriscos (ideal para passwords) y que el texto se lee con **getPassword()**.

JEditorPane

La clase **javax.swing.JEditorPane** es un componente multitexto capaz de mostrar y editar varios tipos de texto como RTF y HTML.



UNIDAD 8. Aplicaciones Controladas por Eventos.

EJEMPLO 19: Mostrar un diálogo que permita rellenar los datos personales de un usuario. Debes usar el código de `DialogLayout.java` en el paquete `dl` (en la página 91 tienes el código).

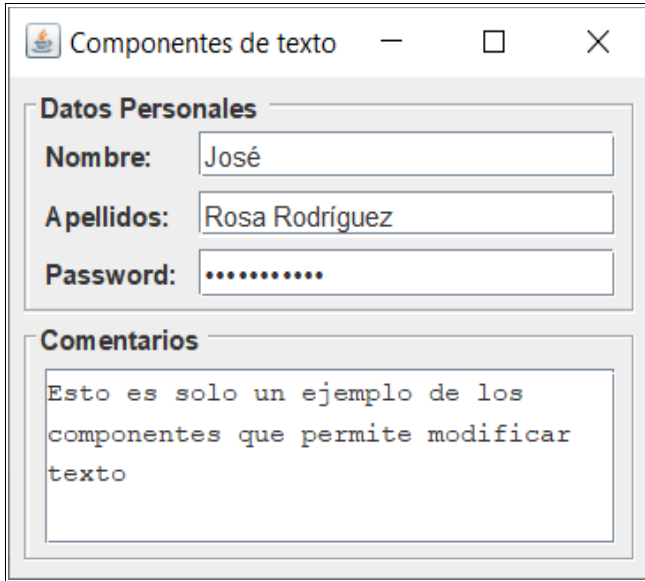


Figura 15: Ejecución del ejemplo.

```
package componentes;

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

public class TextoCompo extends JFrame {
    private static final long serialVersionUID = 1L;
    protected JTextField m_nombre;
    protected JTextField m_apellidos;
    protected JPasswordField m_password;
    protected JTextArea m_comentario;

    public TextoCompo() { // constructor
        super("Componentes de texto");
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

Font ms = new Font("Monospaced", Font.PLAIN, 12);
JPanel pp = new JPanel( new BorderLayout(0,0) );
JPanel p = new JPanel( new DialogLayout() );
//p.setBorder( new JLabel("Nombre:") );
p.add( new JLabel("Nombre:") );
m_nombre = new JTextField(20);
p.add(m_nombre);
p.add(new JLabel("Apellidos:"));
m_apellidos = new JTextField(20);
p.add(m_apellidos);
p.add(new JLabel("Password:"));
m_password = new JPasswordField(20);
m_password.setFont(ms);
p.add(m_password);
p.setBorder(new CompoundBorder(
    new TitledBorder(new EtchedBorder(), "Datos Personales"),
    new EmptyBorder(1, 5, 3, 5))
);
pp.add(p, BorderLayout.NORTH);
m_comentario = new JTextArea("", 4, 30);
m_comentario.setFont(ms);
m_comentario.setLineWrap(true);
m_comentario.setWrapStyleWord(true);
p = new JPanel(new BorderLayout());
p.add(new JScrollPane(m_comentario));
p.setBorder(
    new CompoundBorder(
        new TitledBorder(new EtchedBorder(), "Comentarios"),
        new EmptyBorder(3, 5, 3, 5))
);
pp.add(p, BorderLayout.CENTER);
pp.setBorder(new EmptyBorder(5, 5, 5, 5));
getContentPane().add(pp);
pack();
}

public static void main(String[] args) {
    JFrame frame = new TextoCompo();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}
}

```

TEXTOS CON PLANTILLAS

La clase `javax.swing.JFormattedTextField` es un componente de Swing que extiende a `JTextField` y añade soporte para formatos



UNIDAD 8. Aplicaciones Controladas por Eventos.

personalizados que se definen mediante máscaras.

MaskFormatter (subclase de **DefaultFormatter**) se diseñó para definir Strings que se ajusten a una plantilla (sigan un formato). El formato se define mediante otro String que hace de máscara. La máscara se pasa al constructor o al método **setMask()**. En la máscara están permitidos los siguientes caracteres:

- #: representa cualquier carácter numérico que case con **Character.isDigit()**.
- ': carácter de escape.
- U: cualquier carácter, las minúsculas se convierten a mayúsculas, validado por **Character.isLetter()**.
- L: cualquier carácter, las mayúsculas se transforman en minúscula, validado por **Character.isLetter()**.
- A: cualquier carácter o número, validado por **Character.isLetter()** o **Character.isDigit()**.
- ?: cualquier carácter, validado por **Character.isLetter()**.
- *: cualquier carácter.
- H: cualquier carácter hexadecimal (0-9, a-f o A-F)
- Cualquier otro carácter que aparezca en la máscara es fijo.

Por ejemplo, la plantilla de un CIF: "#####-U"

Cuando se intenta rellenar un string que debe seguir una máscara, puedes cambiar el carácter que indica que en esa posición se necesita teclear algo, por defecto es el espacio en blanco pero se puede cambiar con **setPlaceholderCharacter()**.

EJEMPLO 19: Dos campos, uno con un valor de dinero y otro con una fecha a los que se impone un formato. Se necesita :

```
package mascarar;
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
import java.awt.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.*;
import dl.*;

class Mascaras extends JFrame {
    private static final long serialVersionUID = 1L;

    public Mascaras() {
        super("Texto con Formato");
        JPanel p = new JPanel( new DialogLayout() );
        p.setBorder( new EmptyBorder(10, 10, 10, 10) );
        p.add( new JLabel("Cantidad de Euros:") );
        NumberFormat nf= NumberFormat.getCurrencyInstance(Locale.FRANCE);
        JFormattedTextField ftImporte = new JFormattedTextField(nf);
        ftImporte.setColumns(10);
        ftImporte.setValue( 100 );
        p.add(ftImporte);
        p.add(new JLabel("Fecha de transacción:") );
        DateFormat fd= new SimpleDateFormat("MM/dd/yyyy");
        JFormattedTextField ftFecha = new JFormattedTextField(fd);
        ftFecha.setColumns(10);
        ftFecha.setValue( new Date() );
        p.add(ftFecha);
        JButton btn = new JButton("OK");
        p.add(btn);
        getContentPane().add(p, BorderLayout.CENTER);
        pack();
    }

    public static void main( String args[] ) {
        Mascaras mainFrame = new Mascaras();
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainFrame.setVisible(true);
    }
}
```

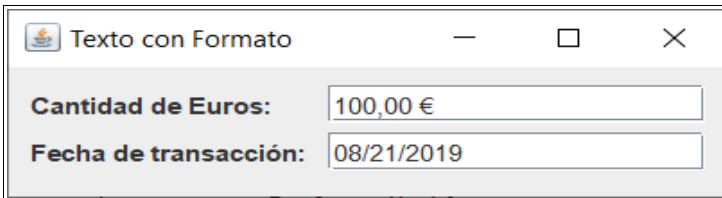


Figura 16: Ejecución de texto con formato.



UNIDAD 8. Aplicaciones Controladas por Eventos.

Usar formatos con verificadores

Puedes combinar el uso de formatos con la clase **InputVerifier** que automáticamente traslada el foco a aquellos datos que no cumplan el formato (pero se sale ya del objetivo de este tema).

8.4.2. LAYOUTS (DISEÑOS).

Otro aspecto importante al hacer programas con GUI es la distribución de los componentes en la ventana. Cuando el contenedor puede cambiar de tamaño hay que decidir como se deben mover, si pueden crecer o disminuir de tamaño, etc.

Ya hemos usado componentes que son los objetos visibles de la GUI, también objetos contenedores (contienen a otros), por ejemplo el content pane de un JFrame que puede ser un JPanel.

Como un JPanel es un contenedor, puedes añadirle otros JPanel, así que la anidación de componentes es posible.



Figura 17: Anidación de Contenedores.



UNIDAD 8. Aplicaciones Controladas por Eventos.

En la figura 17, un panel (rosa) contiene otros dos paneles (azules), y cada uno de ellos a su vez contienen a otros (en gris). De mantener estos diseños se encargan los objetos de la clase **LayoutManager**.

Cada contenedor tiene un layout manager por defecto y un método para cambiarlo llamado **setLayout()**. Los componentes se añaden llamando al método **add()** del contenedor. Hay varias versiones del método **add()** según la versión de layout que use el contenedor.

FLOW LAYOUT

Alinea los componentes del contenedor en una fila. El tamaño de cada componente coincide con su "preferred size." Cuando en una fila ya no caben más, los mueve a la siguiente fila. **Un JPanel tiene un FlowLayout por defecto.**

Puede alinear los componentes a la derecha, izquierda o centro y pueden indicarse huecos horizontales y verticales.

Con el constructor **FlowLayout()**, los componentes de cada fila están centrados tanto horizontal como verticalmente y con huecos de 5 pixels. El constructor **FlowLayout(int align, int hgap, int vgap)** es otra alternativa y los valores de align puede ser **FlowLayout.LEFT**, **FlowLayout.RIGHT** y **FlowLayout.CENTER**.

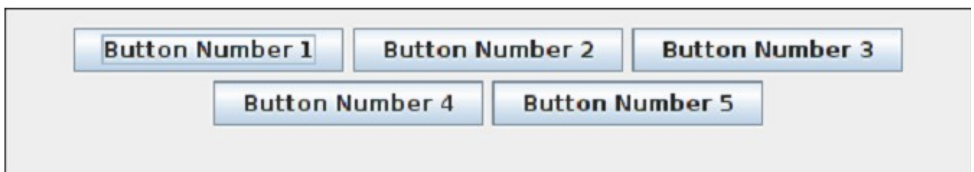


Figura 18: Distribución de componentes de FlowLayout.

BORDER LAYOUT



UNIDAD 8. Aplicaciones Controladas por Eventos.

Muestra un gran espacio central con hasta cuatro componentes más pequeños rodeándolo. Si el contenedor `c` está usando un **BorderLayout**, un componente `comp` se añade al contenedor con la sentencia **`c.add(comp, lugar);`** donde `lugar` puede ser **`BorderLayout.CENTER`**, **`BorderLayout.NORTH`**, **`BorderLayout.SOUTH`**, **`BorderLayout.EAST`** o **`BorderLayout.WEST`**.

No siempre tendrá los 5 componentes, aunque es muy raro que no tenga componente central. Los tamaños los decide así: norte y sur (si hay) se muestran en su altura preferida, pero el ancho rellena la anchura del contenedor. Este y oeste se muestran con sus anchuras preferidas, pero la altura rellena el contenedor (sin contar altura de norte y sur claro). Por último, el centro se queda con el resto.

Cada componente se comporta mejor o peor en algunas posiciones de este layout, por ejemplo un slider horizontal o un campo de texto (pensados para dirección horizontal) se comportan bien en el norte y sur, pero son poco agradables en el este o en el oeste.

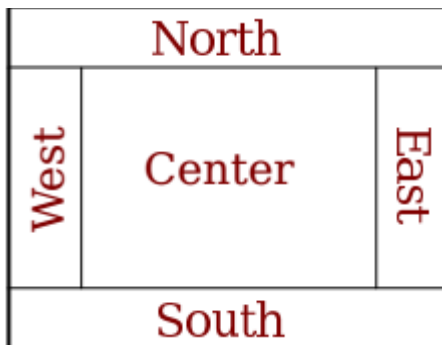


Figura 19: Componentes de un BorderLayout.

El constructor por defecto es **`BorderLayout()`** y no deja espacio entre los componentes. Otro es **`BorderLayout(int hh, int hv)`** que deja huecos horizontales `hh` y verticales `hv`.



UNIDAD 8. Aplicaciones Controladas por Eventos.

Es el layout por defecto de un JFrame por ejemplo.

GRIDLAYOUT

Deja cada componente en una rejilla de rectángulos definidos en filas y columnas de igual tamaño. Para añadir componentes simplemente se indica el componente a añadir. Se van añadiendo rellenando primero las filas y luego las columnas, por ejemplo: `c.add(comp1)`; irá a la fila 1, columna 1 (rectángulo #1 en la figura 20).

El constructor es `GridLayout(filas,columnas)`, y si quieres tener huecos horizontales (hh) o verticales (hv) usas `GridLayout(f,c,hh,hv)`.

#1	#2	#3
#4	#5	#6
#7	#8	#9
#10	#11	#12

Figura 20: Divisiones de un GridLayout.

Si pones a cero el número de columnas, el creará las necesarias, y puedes hacer lo mismo con el número de filas. Si defines un grid layout de 1×3 , es similar a un flow layout de una sola línea. Ejemplo:

```
JPanel barra = new JPanel();
barra.setLayout( new GridLayout(1,3) );
// El 3 se ignora y equivale a GridLayout(1,0)
// Para tener huecos: GridLayout(1,0,5,5)
...
barra.add(boton1);
barra.add(boton2);
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
barra.add(boton3);
```

BORDES

Además de huecos se pueden definir bordes entre los componentes. La clase `javax.swing.BorderFactory` tiene un gran número de métodos estáticos para crear bordes de los objetos. Por ejemplo la función `BorderFactory.createLineBorder(Color.BLACK)` devuelve un objeto borde de un pixel de color negro bordeando al componente. Este objeto se añade al componente `c` con el método `c.setBorder(BorderFactory.createLineBorder(Color.BLACK));`

Ejemplos de bordes (hay muchos más):

- `BorderFactory.createEmptyBorder(top, left, bottom, right)` — no se dibuja nada. Los parámetros son espacio con las fronteras del componente.
- `BorderFactory.createLineBorder(color, grosor)` — línea alrededor de un color y un grosor.
- `BorderFactory.createMatteBorder(top, left, bottom, right, color)` — es como el de línea pero con un grosor distinto para cada arista.
- `BorderFactory.createEtchedBorder()` — crea un borde que parece estar marcado en el contenedor.
- `BorderFactory.createLoweredBevelBorder()` — efecto 3D que deja al componente como hundido en el contenedor.
- `BorderFactory.createRaisedBevelBorder()` — similar pero elevado sobre el contenedor.
- `BorderFactory.createTitledBorder(title)` — borde con título.



UNIDAD 8. Aplicaciones Controladas por Eventos.



Figura 21: Algunos ejemplos de bordes.

EJEMPLO 20: Una calculadora sencilla con un grid layout de 4x1 y 3 pixels de huecos, Mira el código para fijarte en los detalles.

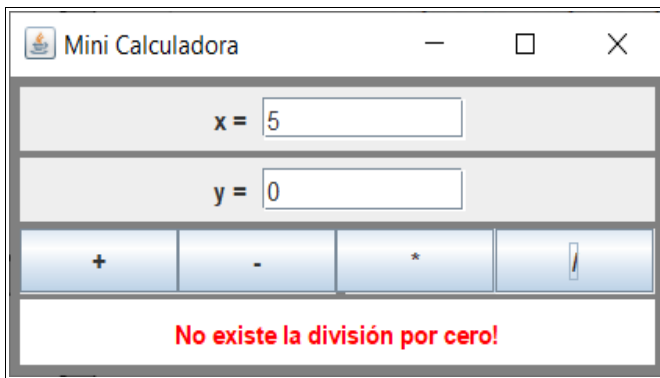


Figura 22: El programa del ejemplo.

```
package calculadora;

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

/**
 * Probar layouts, campos de texto y botones
 */
public class Calc extends JPanel implements ActionListener {

    /**
```




UNIDAD 8. Aplicaciones Controladas por Eventos.

```

* Hace posible ejecutar el programa
*/
public static void main(String[] args) {
    JFrame window = new JFrame("Mini Calculadora");
    Calc content = new Calc();
    window.setContentPane(content);
    window.pack(); // Tamaños a tamaño preferido.
    window.setLocation(100,100);
    window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    window.setVisible(true);
}

//-----

private JTextField xInput, yInput; // Cajas de entrada.
private JLabel resp; // La respuesta es un JLabel

public Calc() {
    setBackground(Color.GRAY);
    setBorder( BorderFactory.createEmptyBorder(5,5,5,5) );
    xInput = new JTextField("0", 10);
    xInput.setBackground(Color.WHITE);
    yInput = new JTextField("0", 10);
    yInput.setBackground(Color.WHITE);
    // Paneles para contener las cajas, etc.
    JPanel xPanel = new JPanel();
    xPanel.add( new JLabel(" x = ") );
    xPanel.add(xInput);
    JPanel yPanel = new JPanel();
    yPanel.add( new JLabel(" y = ") );
    yPanel.add(yInput);
    // Panel para 4 botones
    JPanel bPanel = new JPanel();
    bPanel.setLayout(new GridLayout(1,4));
    JButton plus = new JButton("+");
    plus.addActionListener(this);
    bPanel.add(plus);
    JButton minus = new JButton("-");
    minus.addActionListener(this);
    bPanel.add(minus);
    JButton producto = new JButton("*");
    producto.addActionListener(this);
    bPanel.add(producto);
    JButton divide = new JButton("/");
    divide.addActionListener(this);
    bPanel.add(divide);
    // Fondo y texto de la respuesta
    resp = new JLabel("x + y = 0", JLabel.CENTER);
    resp.setForeground(Color.red);
}

```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

resp.setBackground(Color.white);
resp.setOpaque(true);
// Definir el layout
setLayout( new GridLayout(4,1,3,3) );
add(xPanel);
add(yPanel);
add(bPanel);
add(resp);
} // fin constructor

/**
 * Al clicar un botón se hacen los cálculos
 */
public void actionPerformed(ActionEvent evt) {
    double x, y; // The numbers from the input boxes.
    // Obtener x (texto a double)
    try {
        String xStr = xInput.getText();
        x = Double.parseDouble(xStr);
    }
    catch (NumberFormatException e) {
        // The string xStr is not a legal number.
        resp.setText("Dato no legal en x");
        xInput.requestFocusInWindow();
        return;
    }
    // Obtener y (texto a double)
    try {
        String yStr = yInput.getText();
        y = Double.parseDouble(yStr);
    }
    catch (NumberFormatException e) {
        resp.setText("Dato no legal para y.");
        yInput.requestFocusInWindow();
        return;
    }
    // Realizar operación según comando de acción del botón
    String op = evt.getActionCommand();
    if (op.equals("+"))
        resp.setText( "x + y = " + (x+y) );
    else if (op.equals("-"))
        resp.setText( "x - y = " + (x-y) );
    else if (op.equals("*"))
        resp.setText( "x * y = " + (x*y) );
    else if (op.equals("/")) {
        if (y == 0)
            resp.setText("No existe la división por cero!");
        else
            resp.setText( "x / y = " + (x/y) );
    }
}

```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
    }  
  } // actionPerformed()  
} // clase
```

USAR UN NULL LAYOUT

Puedes poner un layout nulo (no usar ninguno) en un contenedor. Tu programa es responsable de definir tamaños preferidos y posiciones de los componentes. Para hacerlo, pasas un null como en **contenedor.setLayout(null)**. Esto tiene ventajas (apariencia fácil de controlar en tiempo de diseño y fija en ejecución) y desventajas (distintas resoluciones de los dispositivos afectan a la apariencia y usabilidad: puedes cortar componentes, texto, etc.).

EJEMPLO 21: definir el diseño de la figura 21 sin layout manager. Observa el código.



Figura 23: El programa del ejemplo.



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

package ajedrez;

/**
 * Demo de null layout
 */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Ajedrez extends JPanel implements ActionListener {

    /**
     * Main */
    public static void main(String[] args) {
        JFrame window = new JFrame("Demo de Null Layout");
        Ajedrez content = new Ajedrez();
        window.setContentPane(content);
        window.pack(); // tamaño preferido
        window.setResizable(false); //Tamaño fijo
        window.setLocation(100,100);
        window.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        window.setVisible(true);
    }

    //-----

    Tablero t; // tablero de ajedrez
    JButton cambiaTama; // Two buttons.
    JButton nuevaPartida;
    JLabel msj; // feedback al usuario
    int clickCount; // Contar clics

    /**
     * constructor */
    public Ajedrez() {
        setLayout(null); // sin layout
        setBackground(new Color(0,120,0));
        setBorder( BorderFactory.createEtchedBorder() );
        setPreferredSize( new Dimension(350,240) );
        t = new Tablero();
        add(t);
        nuevaPartida = new JButton("Nueva partida");
        nuevaPartida.addActionListener(this);
        add(nuevaPartida);
        cambiaTama = new JButton("Cambia Tamaño");
        cambiaTama.addActionListener(this);
        add(cambiaTama);
        msj = new JLabel("Clic \"Nueva partida\" para comenzar.");
        msj.setForeground( new Color(100,255,100) );
    }
}

```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

msj.setFont(new Font("Serif", Font.BOLD, 14));
add(msj);
// pos y tamaño con setBounds()
t.setBounds(20,20,164,164);
nuevaPartida.setBounds(210, 60, 120, 30);
cambiaTama.setBounds(210, 120, 120, 30);
msj.setBounds(20, 200, 330, 30);
}

/**
 * Responder a un clic */
public void actionPerformed(ActionEvent evt) {
    String buttonText = evt.getActionCommand();
    clickCount++;
    if (clickCount == 1)
        msj.setText("Primer clic: \"" + buttonText +
                    "\" clicado.");
    else
        msj.setText("Clic " + clickCount + ": \"" + buttonText
                    + "\" fue clicado.");
}

/**
 * muestra tablero */
private static class Tablero extends JPanel {
    public Tablero() {
        setPreferredSize( new Dimension(164, 164) );
    }
    public void paintComponent(Graphics g) {
        g.setColor(Color.BLACK);
        g.drawRect(0,0,getSize().width-1, getSize().height-1);
        g.drawRect(1,1,getSize().width-3, getSize().height-3);
        // dibujar cuadros
        for (int row = 0; row < 8; row++) {
            for (int col = 0; col < 8; col++) {
                if ( row % 2 == col % 2 )
                    g.setColor(Color.LIGHT_GRAY);
                else
                    g.setColor(Color.GRAY);
                g.fillRect(2 + col*20, 2 + row*20, 20, 20);
            }
        }
    }
} // tablero
} // clase

```

PERSONALIZAR LAYOUTS



UNIDAD 8. Aplicaciones Controladas por Eventos.

Si alguna vez necesitas un layout que no existe, te lo puedes fabricar. Por ejemplo vamos a diseñar uno que asocie un JLabel a cada componente como su etiqueta.

```
package dl;

import java.awt.*;
import java.util.*;

public class DialogLayout implements LayoutManager {
    protected int m_divider = -1;
    protected int m_hGap = 10;
    protected int m_vGap = 5;
    public DialogLayout() {}
    public DialogLayout(int hGap, int vGap) {
        m_hGap = hGap;
        m_vGap = vGap;
    }
    public void addLayoutComponent(String name, Component comp) {}
    public void removeLayoutComponent(Component comp) {}
    public Dimension preferredLayoutSize(Container parent) {
        int divider = getDivider(parent);
        int w = 0;
        int h = 0;
        for (int k=1 ; k<parent.getComponentCount(); k+=2) {
            Component comp = parent.getComponent(k);
            Dimension d = comp.getPreferredSize();
            w = Math.max(w, d.width);
            h += d.height + m_vGap;
        }
        h -= m_vGap;
        Insets insets = parent.getInsets();
        return new Dimension(divider+w+insets.left+insets.right,
            h+insets.top+insets.bottom);
    }

    public Dimension minimumLayoutSize(Container parent) {
        return preferredLayoutSize(parent);
    }

    public void layoutContainer(Container parent) {
        int divider = getDivider(parent);
        Insets insets = parent.getInsets();
        int w = parent.getWidth() - insets.left - insets.right -
            divider;
        int x = insets.left;
        int y = insets.top;
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

        for (int k=1 ; k<parent.getComponentCount(); k+=2) {
            Component comp1 = parent.getComponent(k-1);
            Component comp2 = parent.getComponent(k);
            Dimension d = comp2.getPreferredSize();
            comp1.setBounds(x, y, divider-m_hGap, d.height);
            comp2.setBounds(x+divider, y, w, d.height);
            y += d.height + m_vGap;
        }
    }

    public int getHGap() { return m_hGap; }
    public int getVGap() { return m_vGap; }
    public void setDivider(int divider) {
        if (divider > 0)
            m_divider = divider;
    }
    public int getDivider() { return m_divider; }
    protected int getDivider(Container parent) {
        if (m_divider > 0)
            return m_divider;
        int divider = 0;
        for (int k=0 ; k<parent.getComponentCount(); k+=2) {
            Component comp = parent.getComponent(k);
            Dimension d = comp.getPreferredSize();
            divider = Math.max(divider, d.width);
        }
        divider += m_hGap;
        return divider;
    }
    public String toString() {
        return getClass().getName() +
            "[hgap=" + m_hGap + ",vgap=" + m_vGap + ",divider="
            + m_divider + "]";
    }
}

```

8.4.3. PANELES.

JTABBEDPANE

La clase *javax.swing.JTabbedPane* es simplemente una pila de componentes. Cada layer tiene un componente que normalmente es un contenedor. La extensión *Tab* se usan para mover al frente uno de los contenedores y pueden tener asignado un texto, un icono (y su



UNIDAD 8. Aplicaciones Controladas por Eventos.

desactivado), colores de fondo y de dibujo y un tooltip.

Para añadir un componente a un tabbed pane, usas el método sobrecargado **add()**, que crea un nuevo tab seleccionable y reorganiza los otros tabs si es necesario. También puedes usar los métodos **addTab()** e **insertTab()**. El método **remove()** acepta un componente como parámetro y lo elimina del tab asociado.

Los tab asociados pueden residir en norte, sur, este u oeste del pane content. El lugar se indica con el método **setTabPlacement()** que acepta una de las constantes ya mencionadas.

Puedes indicar cuando los tabs deben dividirse en varias filas de tabs, o formar una fila desplazable. La política de distribución se aplica con el método **setTabLayoutPolicy()** que puede aceptar como parámetros los dos valores siguientes: **JTabbedPane.WRAP_TAB_LAYOUT** y **JTabbedPane.SCROLL_TAB_LAYOUT**. La figura 24 ilustra las posibilidades de distribución de los tab.



Figura 24-a. opción SCROLL_TAB_LAYOUT y alineación TOP.



UNIDAD 8. Aplicaciones Controladas por Eventos.



Figura 24-b. opción `WRAP_TAB_LAYOUT` y alineación `TOP`.

Con los métodos `getSelectedIndex()` y `setSelectedIndex()` manejas el tab activo y su componente asociado con `getSelectedComponent()` y `setSelectedComponent()`.

A un tabbed pane puedes añadir uno o más `ChangeListeners` para escuchar eventos de cambios en el tab activado, lo que genera eventos `ChangeEvent` que puedes procesar en el método `stateChanged()` del listener. También generan eventos `PropertyChangeEvent` cuando la posición o el modelo de seleccionar tabs cambia.

EJEMPLO 22: vamos a permitir o denegar el acceso a un tab en un tab pane de 2 tabs. En el primero hay una lista de empleados, si la lista está vacía o no hay empleado seleccionado, no deja acceder al segundo. De lo contrario accedes al segundo donde modificas sus datos.

```
package tabpanned;

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



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
public class TPDemo extends JFrame {
    public static final int LIST_TAB = 0;
    public static final int DATA_TAB = 1;
    protected Persona[] m_emp = {
        new Persona("Juanito", "Soria", "111-111111"),
        new Persona("Silvia", "Gil", "222-222222"),
        new Persona("Dora", "Exploradora", "333-333333"),
        new Persona("Pepa", "Pig", "444-444444"),
        new Persona("Bob", "Esponja", "000-123456")
    };

    protected JList m_list;
    protected JTextField m_nom;
    protected JTextField m_ape;
    protected JTextField m_tel;
    protected JTabbedPane m_tab;
    public TPDemo() {
        super("Demo de TabPanned");
        JPanel p1 = new JPanel( new BorderLayout() );
        p1.setBorder( new EmptyBorder(10, 10, 10, 10) );
        m_list = new JList(m_emp);
        m_list.setVisibleRowCount(4);
        JScrollPane sp = new JScrollPane(m_list);
        p1.add(sp, BorderLayout.CENTER);
        MouseListener mlst = new MouseAdapter() {
            public void mouseClicked(MouseEvent evt) {
                if( evt.getClickCount() == 2 )
                    m_tab.setSelectedIndex(DATA_TAB);
            }
        };
        m_list.addMouseListener(mlst);
        JPanel p2 = new JPanel( new DialogLayout() );
        p2.setBorder( new EmptyBorder(10, 10, 10, 10) );
        p2.add( new JLabel("Nombre:") );
        m_nom = new JTextField(20);
        p2.add(m_nom);
        p2.add( new JLabel("Apellidos:") );
        m_ape = new JTextField(20);
        p2.add(m_ape);
        p2.add(new JLabel("Teléfono:"));
        m_tel = new JTextField(20);
        p2.add(m_tel);
        m_tab = new JTabbedPane();
        m_tab.addTab("Empleados", p1);
        m_tab.addTab("Datos Personales", p2);
        m_tab.addChangeListener(new TabChangeListener());
        JPanel p = new JPanel();
        p.add(m_tab);
        p.setBorder(new EmptyBorder(5, 5, 5, 5));
        getContentPane().add(p);
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
        pack();
    }

    public Persona getSelectedPersona() {
        return (Persona)m_list.getSelectedValue();
    }
    public static void main(String[] args) {
        JFrame frame = new TPDemo();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    class TabChangeListener implements ChangeListener {
        public void stateChanged(ChangeEvent e) {
            Persona sp = getSelectedPersona();
            switch(m_tab.getSelectedIndex()) {
                case DATA_TAB: if (sp == null) {
                                m_tab.setSelectedIndex(LIST_TAB);
                                return;
                            }
                                m_nom.setText(sp.m_nom );
                                m_ape.setText(sp.m_ape);
                                m_tel.setText(sp.m_tel);
                                break;
                case LIST_TAB: if (sp != null) {
                                sp.m_nom = m_nom.getText();
                                sp.m_ape = m_ape.getText();
                                sp.m_tel = m_tel.getText();
                                m_list.repaint();
                            }
                                break;
            }
        }
    }
}

class Persona{
    public String m_nom;
    public String m_ape;
    public String m_tel;
    public Persona(String nombre, String apes, String telef) {
        m_nom = nombre;
        m_ape = apes;
        m_tel = telef;
    }
    public String toString() {
        String str = m_nom + " " + m_ape;
        if (m_tel.length() > 0) str+= " (" + m_tel + ")";
        return str.trim();
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
} // clase anidada Persona
} // clase TPDemo
```

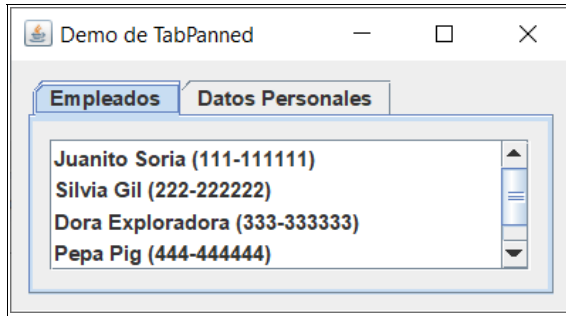


Figura 25. Un tab panned con dos tabs.

JSCROLLPANE

La clase **javax.swing.JScrollPane** es muy simple. Cualquier componente o contenedor que pongas dentro se podrá desplazar con barras. Su constructor:

```
JScrollPane jsp = new JScrollPane(etiqueta);
```

Algunos componentes usan un JScrollPane internamente para mostrar su contenido, como JComboBox y JList. Y los componentes de texto multilínea deben de estar dentro de uno porque no tienen la capacidad de desplazarse por su contenido por ellos mismos.

EJEMPLO 23: Cargar una imagen dentro para verla usando las barras. Es muy lento porque solo se desplaza un pixel cada pulsación. Para solucionar esto, hay que acceder a las barras de desplazamiento con los métodos **getXScrollbar()** y **setXScrollbar()** donde XX puede ser **HORIZONTAL** o **VERTICAL**.

```
package scroll;

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



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
public class Demo extends JFrame {  
    public Demo() {  
        super("Demo de JScrollPane");  
        ImageIcon ii = new ImageIcon("cara.jpg");  
        JScrollPane jsp = new JScrollPane( new JLabel(ii) );  
        getContentPane().add(jsp);  
        setSize(300,250);  
        setVisible(true);  
    }  
  
    public static void main(String[] args) {  
        new Demo();  
    }  
}
```



Figura 26: Ejemplo de JScrollPane.

Java 1.4 añade soporte para las ruedas del ratón en un JScrollPane. La interfaz **MouseWheelListener** y la clase **MouseWheelEvent** se encargan de la implementación y el método **addMouseWheelListener()** permite la asociación. Puedes desactivar y reactivar este soporte mediante los métodos **setWheelScrollingEnabled()**. Pero si no necesitas cambiar el comportamiento por defecto, no tendrás que usarlos.



UNIDAD 8. Aplicaciones Controladas por Eventos.

La interfaz abstracta **`javax.swing.ScrollPaneConstants`** permite indicar políticas de uso de las barras usando los métodos **`setVerticalScrollBarPolicy()`** y **`setHorizontalScrollBarPolicy()`** que aceptan una de estas constantes:

- `HORIZONTAL_SCROLLBAR_AS_NEEDED`
- `HORIZONTAL_SCROLLBAR_NEVER`
- `HORIZONTAL_SCROLLBAR_ALWAYS`
- `VERTICAL_SCROLLBAR_AS_NEEDED`
- `VERTICAL_SCROLLBAR_NEVER`
- `VERTICAL_SCROLLBAR_ALWAYS`

Por ejemplo, forzar que la barra vertical siempre esté visible aunque no haga falta y ocultar la horizontal:

```
jsp.setHorizontalScrollBarPolicy(  
    ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);  
jsp.setVerticalScrollBarPolicy(  
    ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
```

JVIEWPORT

La clase **`javax.swing.JViewport`** es la realmente responsable en un `ScrollPane` de mostrar una región visible de un componente. Podemos obtener o asignar el viewport con los métodos **`setView()`** y **`getView()`**. Podemos controlar cuanto muestra cambiando el tamaño con el método **`setExtentSize()`** al que le pasamos un objeto `Dimension`. También podemos indicar el origen en el que debe comenzar a mostrar con **`setViewPosition()`** (estas coordenadas son las que cambian al pulsar las barras del `JScrollPane`).

La clase **`javax.swing.ScrollPaneLayout`** gestiona el comportamiento de un `JScrollPane`. Un `JScrollPane` puede contener hasta 9 componentes y su `ScrollPaneLayout` se asegura de que se posicionen correctamente. Los 9 componentes son:

- Un `JViewport` con el componente principal a desplazar.



UNIDAD 8. Aplicaciones Controladas por Eventos.

- Un JViewport que se usa para cargar una nueva fila.
- Un JViewport que se usa para cargar la siguiente columna.
- 4 componentes (uno en cada esquina).
- Dos JScrollBars para desplazamientos.

Para asignar un componente a una esquina del JScrollPane usas su método **setCorner()** que acepta un string y un componente de parámetros. El string sirve para identificar la esquina.

Para asignar los JViewports como cabeceras de fila y columna se usan los métodos **setRowHeader()** y **setColumnHeader()** del JScrollPane. Podemos evitar tener que crearlos nosotros mismos si le pasamos directamente el componente de la fila o columna con **setRowHeaderView()** o **setColumnHeaderView()**.

EJEMPLO 24: Como JScrollPane se usa con frecuencia para mostrar imágenes, es habitual usar algún tipo de regla como cabecera de fila y columna. EL ejemplo muestra una marca cada 30 pixels y la dibujo en función de la posición mostrada.

```
package viewport;

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

public class VDemo extends JFrame {
    public VDemo() {
        super("Demo JScrollPane ");
        ImageIcon ii = new ImageIcon("cara.jpg");
        JScrollPane jsp = new JScrollPane(new JLabel(ii));
        JLabel[] esquinas = new JLabel[4];
        for(int i=0; i<4; i++) {
            esquinas[i] = new JLabel();
            esquinas[i].setBackground(Color.yellow);
            esquinas[i].setOpaque(true);
            esquinas[i].setBorder(
                BorderFactory.createCompoundBorder(
                    BorderFactory.createEmptyBorder(2,2,2,2),
                    BorderFactory.createLineBorder(Color.red, 1)
                )
            );
        }
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

        );
    }

JLabel rowheader = new JLabel() {
    Font f = new Font("Serif",Font.ITALIC | Font.BOLD,10);

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Rectangle r = g.getClipBounds();
        g.setFont(f);
        g.setColor(Color.red);
        for (int i = 30-(r.y % 30);i<r.height;i+=30) {
            g.drawLine(0, r.y + i, 3, r.y + i);
            g.drawString("" + (r.y + i), 6, r.y + i + 3);
        }
    }

    public Dimension getPreferredSize() {
//return new Dimension(25, (int)this.getPreferredSize().getHeight() );
        return new Dimension(25, 0 );
    }
};

rowheader.setBackground(Color.yellow);
rowheader.setOpaque(true);

JLabel columnheader = new JLabel() {
    Font f = new Font("Serif",Font.ITALIC | Font.BOLD,10);

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Rectangle r = g.getClipBounds();
        g.setFont(f);
        g.setColor(Color.red);
        for (int i = 30-(r.x % 30);i<r.width;i+=30) {
            g.drawLine(r.x + i, 0, r.x + i, 3);
            g.drawString("" + (r.x + i), r.x + i - 10, 16);
        }
    }

    public Dimension getPreferredSize() {
//return new Dimension( (int)this.getPreferredSize().getWidth(), 25);
        return new Dimension( 0, 25);
    }
};

columnheader.setBackground(Color.yellow);
columnheader.setOpaque(true);

```




UNIDAD 8. Aplicaciones Controladas por Eventos.

```

jsp.setRowHeaderView(rowheader);
jsp.setColumnHeaderView(columnheader);
jsp.setCorner(JScrollPane.LOWER_LEFT_CORNER, esquinas[0]);
jsp.setCorner(JScrollPane.LOWER_RIGHT_CORNER, esquinas[1]);
jsp.setCorner(JScrollPane.UPPER_LEFT_CORNER, esquinas[2]);
jsp.setCorner(JScrollPane.UPPER_RIGHT_CORNER, esquinas[3]);
getContentPane().add(jsp);
setSize(400,300);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true);
}

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

```

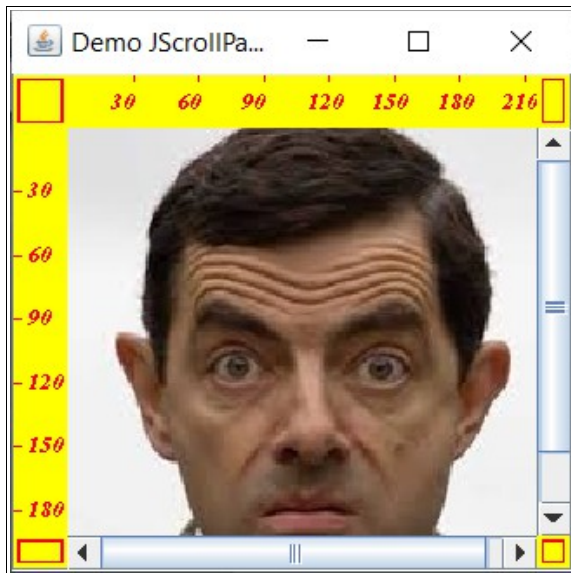


Figura 27: Cambiar elementos del JScrollPane.

JSPLITPANE

La clase `javax.swing.JSplitPane` divide los paneles para permitir a los



UNIDAD 8. Aplicaciones Controladas por Eventos.

usuarios cambiar dinámicamente el tamaño de dos o más componentes que se muestran al lado (separados por una frontera en el mismo panel). Un divisor puede ser arrastrado con el ratón para cambiar el tamaño de uno y otro, sin cambiar el tamaño total.

Un ejemplo muy usado es la combinación de un árbol y una tabla separados por un divisor común. El encargado de ofrecer esta posibilidad es **JSplitPane** que maneja dos componentes separados por un divisor horizontal o vertical. Los componentes se añaden en el constructor o con los métodos **setXXComponent()** (donde XX puede ser Left, Right, Top, o Bottom). Podemos establecer la orientación con **setOrientation()**.

El tamaño del divisor (la única parte visible) se cambia con **setDividerSize()**, y su posición con **setDividerLocation()** que acepta una posición absoluta, o proporcional si es un double. Tiene también una propiedad **oneTouchExpandable** que si es true, muestra dos flechas dentro del divisor.

EJEMPLO 25: Manipular tamaño de 4 paneles colocados en 3 JsplitPanes:

```
package split;

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

public class Demo extends JFrame {
    public Demo() {
        super("Ejemplo SplitPanel");
        setSize(400, 400);
        Component c11 = new SimplePanel();
        Component c12 = new SimplePanel();
        JSplitPane spLeft = new JSplitPane(
            JSplitPane.VERTICAL_SPLIT, c11, c12);
        spLeft.setDividerSize(8);
        spLeft.setDividerLocation(150);
        spLeft.setContinuousLayout(true);
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

Component c21 = new SimplePanel();
Component c22 = new SimplePanel();
JSplitPane spRight = new JSplitPane(
    JSplitPane.VERTICAL_SPLIT, c21, c22);
spRight.setDividerSize(8);
spRight.setDividerLocation(150);
spRight.setContinuousLayout(true);
JSplitPane sp = new JSplitPane(
    JSplitPane.HORIZONTAL_SPLIT, spLeft, spRight);
sp.setDividerSize(8);
sp.setDividerLocation(200);
sp.setResizeWeight(0.5);
sp.setContinuousLayout(false);
sp.setOneTouchExpandable(true);
getContentPane().add(sp, BorderLayout.CENTER);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setVisible(true);
}
public static void main(String argv[]) { new Demo(); }
}

class SimplePanel extends JPanel {
    public Dimension getPreferredSize() {
        return new Dimension(200, 200);
    }
    public Dimension getMinimumSize() {
        return new Dimension(40, 40);
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.black);
        Dimension sz = getSize();
        g.drawLine(0, 0, sz.width, sz.height);
        g.drawLine(sz.width, 0, 0, sz.height);
    }
}

```



UNIDAD 8. Aplicaciones Controladas por Eventos.

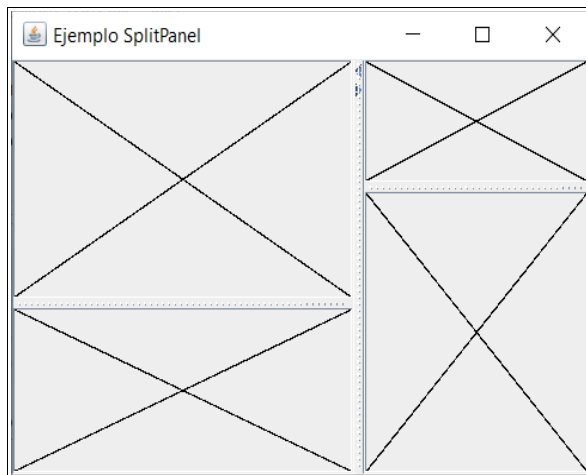


Figura 28: Sencillo Ejemplo de un Splitpane.

8.4.4. LISTAS, COMBOS, SPINNERS Y SLIDERS.

JCOMBOBOX

La clase `javax.swing.JComboBox` representa un componente formado por dos partes:

- Un menú desplegable (pop-up) subclase de `JPopupMenu` que suele ser `javax.swing.plaf.basic.BasicComboPopup` que contiene un `JList` en un `JScrollPane`.
- Un botón que actúa de contenedor para un editor de texto y un botón de flecha que se usa para desplegar el menú.

El `JList` usa un modelo `ListSelectionModel` `SINGLE_SELECTION`. Tiene muchos constructores. El constructor por defecto crea una lista vacía, también podemos pasar datos como un array unidimensional, o un `Vector`, o una implementación de la interfaz `ComboBoxModel`.

Como siempre, podemos usar la implementación por defecto del



UNIDAD 8. Aplicaciones Controladas por Eventos.

ListCellRenderer y **ComboBoxEditor**, o personalizar el comportamiento. La visualización por defecto muestra cada elemento como un string definido por el método **toString()** del objeto (salvo que sea un icono y se renderiza como el icono de un **JLabel**). No se puede interactuar con el componente dibujado.

Usa **ListDataEvents** para informar de cambios en el estado de su modelo de lista desplegable. **ItemEvents** y **ActionEvents** también se generan si la selección actual cambia. Puedes atacar **ItemListeners** y **ActionListeners** para capturarlos.

Puede mostrar u ocultar la lista desplegable usando **showPopup()** y **hidePopup()**. Y puede mostrarlo de dos formas que se eligen con **setLightWeightPopupEnabled()**.

También define una interfaz interna llamada **KeySelectionManager** que declara el método **selectionForKey(char aKey, ComboBoxModel aModel)**, que devuelve el índice del elemento de la lista que debe seleccionarse cuando sea visible la lista.

La interfaz **javax.swing.ComboBoxModel** tiene los métodos **setSelectedItem()** y **getSelectedItem()**.

EJEMPLO 26: muestra información de modelos de coches en un combo y cuando seleccionas uno, se rellenan sus diferentes versiones y te muestra precio y ofertas.

```
package combo;

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



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
public class ComboDemo extends JFrame {
    public ComboDemo() {
        super("ComboBox [Comparar Coches]");
        getContentPane().setLayout( new BorderLayout() );
        Vector cars = new Vector();
        Car maxima = new Car("Maxima", "Nissan",
                               new ImageIcon("nissan.png"));
        maxima.addModelo("GXE", 21499, 19658, "3.0L V6 190-hp");
        maxima.addModelo("SE", 23499, 21118, "3.0L V6 190-hp");
        maxima.addModelo("GLE", 26899, 24174, "3.0L V6 190-hp");
        cars.addElement(maxima);
        Car accord = new Car("Accord", "Honda",
                               new ImageIcon("honda.png"));
        accord.addModelo("LX Sedan", 21700, 19303, "3.0L V6 200-hp");
        accord.addModelo("EX Sedan", 24300, 21614, "3.0L V6 200-hp");
        cars.addElement(accord);
        Car camry = new Car("Camry", "Toyota",
                               new ImageIcon("toyota.png"));
        camry.addModelo("LE V6", 21888, 19163, "3.0L V6 194-hp");
        camry.addModelo("XLE V6", 24998, 21884, "3.0L V6 194-hp");
        cars.addElement(camry);
        Car lumina = new Car("Lumina", "Chevrolet",
                               new ImageIcon("chevrolet.png"));
        lumina.addModelo("LS", 19920, 18227, "3.1L V6 160-hp");
        lumina.addModelo("LTZ", 20360, 18629, "3.8L V6 200-hp");
        cars.addElement(lumina);
        Car taurus = new Car("Taurus", "Ford",
                               new ImageIcon("ford.png"));
        taurus.addModelo("LS", 17445, 16110, "3.0L V6 145-hp");
        taurus.addModelo("SE", 18445, 16826, "3.0L V6 145-hp");
        taurus.addModelo("SHO", 29000, 26220, "3.4L V8 235-hp");
        cars.addElement(taurus);
        Car passat = new Car("Passat", "Volkswagen",
                               new ImageIcon("volkswagen.png"));
        passat.addModelo("GLS V6", 23190, 20855, "2.8L V6 190-hp");
        passat.addModelo("GLX", 26250, 23589, "2.8L V6 190-hp");
        cars.addElement(passat);
        getContentPane().setLayout(new GridLayout(1, 2, 5, 3));
        CarPanel pl = new CarPanel("Modelo Base", cars);
        getContentPane().add(pl);
        CarPanel pr = new CarPanel("Comparar con", cars);
        getContentPane().add(pr);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pl.selectCar(maxima);
        pr.selectCar(accord);
        setResizable(false);
        pack();
        setVisible(true);
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
        public static void main(String argv[]) {
            new ComboDemo();
        }
    }

    class Car {
        protected String nomb;
        protected String fabricante;
        protected Icon img;
        protected Vector modelo;
        public Car(String nombre, String fabricante, Icon img) {
            nomb = nombre;
            this.fabricante = fabricante;
            this.img = img;
            modelo = new Vector();
        }

        public void addModelo(String nombre, int MSRP, int invoice, String
motor) {
            Trim trim = new Trim(this, nombre, MSRP, invoice, motor);
            modelo.addElement(trim);
        }
        public String getNombre() { return nomb; }
        public String getFabricante() { return fabricante; }
        public Icon getIcon() { return img; }
        public Vector getTrims() { return modelo; }
        public String toString() { return fabricante + " " + nomb; }
    }

    class Trim {
        protected Car m_parent;
        protected String m_nombre;
        protected int m_MSRP;
        protected int m_oferta;
        protected String m_motor;
        public Trim(Car parent, String name, int MSRP, int invoice,
String engine) {
            m_parent = parent;
            m_nombre = name;
            m_MSRP = MSRP;
            m_oferta = invoice;
            m_motor = engine;
        }
        public Car getCar() { return m_parent; }
        public String getName() { return m_nombre; }
        public int getMSRP() { return m_MSRP; }
        public int getInvoice() { return m_oferta; }
        public String getEngine() { return m_motor; }
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

    public String toString() { return m_nombre; }
}

class CarPanel extends JPanel {
    protected JComboBox m_cbCars;
    protected JComboBox m_cbVersion;
    protected JLabel m_lImg;
    protected JLabel m_lblMSRP;
    protected JLabel m_lOferta;
    protected JLabel m_lMotor;
    public CarPanel(String title, Vector cars) {
        super();
        setLayout(new BorderLayout(this, BorderLayout.Y_AXIS));
        setBorder(new TitledBorder(new EtchedBorder(), title));
        JPanel p = new JPanel();
        p.add(new JLabel("Modelo:"));
        m_cbCars = new JComboBox(cars);
        ActionListener lst = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Car car = (Car)m_cbCars.getSelectedItem();
                if (car != null)
                    showCar(car);
            }
        };
        m_cbCars.addActionListener(lst);
        p.add(m_cbCars);
        add(p);
        p = new JPanel();
        p.add(new JLabel("Modelo:"));
        m_cbVersion = new JComboBox();
        lst = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Trim trim = (Trim)m_cbVersion.getSelectedItem();
                if (trim != null)
                    showTrim(trim);
            }
        };
        m_cbVersion.addActionListener(lst);
        p.add(m_cbVersion);
        add(p);
        p = new JPanel();
        m_lImg = new JLabel();
        m_lImg.setHorizontalAlignment(JLabel.CENTER);
        m_lImg.setPreferredSize(new Dimension(140, 80));
        m_lImg.setBorder(new BevelBorder(BevelBorder.LOWERED));
        p.add(m_lImg);
        add(p);
        p = new JPanel();
        p.setLayout(new GridLayout(3, 2, 10, 5));
    }
}

```




UNIDAD 8. Aplicaciones Controladas por Eventos.

```

        p.add(new JLabel("MSRP:"));
        m_lblMSRP = new JLabel();
        p.add(m_lblMSRP);
        p.add(new JLabel("Oferta:"));
        m_lOferta = new JLabel();
        p.add(m_lOferta);
        p.add(new JLabel("Motor:"));
        m_lMotor = new JLabel();
        p.add(m_lMotor);
        add(p);
    }
    public void selectCar(Car car) { m_cbCars.setSelectedItem(car); }
    public void showCar(Car car) {
        m_lImg.setIcon( car.getIcon() );
        if (m_cbVersion.getItemCount() > 0)
            m_cbVersion.removeAllItems();
        Vector v = car.getTrims();
        for (int k=0; k<v.size(); k++)
            m_cbVersion.addItem( v.elementAt(k) );
        m_cbVersion.grabFocus();
    }

    public void showTrim(Trim trim) {
        m_lblMSRP.setText(trim.getMSRP()+ "€" );
        m_lOferta.setText(trim.getInvoice() + "€");
        m_lMotor.setText( trim.getEngine() );
    }
}

```

JLIST

La clase **javax.swing.JList** es un componente básico que permite seleccionar uno o más elementos de una lista. Tiene dos modelos:

- **ListModel**, almacena los datos en una lista.
- **ListSelectionModel**, maneja la selección de 3 modos diferentes.

JList también soporta personalizar la visualización implementando la interfaz **ListCellRenderer**. (puedes usar su implementación por defecto, **DefaultListCellRenderer**) o crear la tuya. Con la que tiene muestra cada elemento como un string usando el método `toString()` de los objetos, salvo que sea un icono en cuyo caso usa un **JLabel**. Ten en cuenta que **ListCellRenderer** devuelve un **Component**, pero no es

UNIDAD 8. Aplicaciones Controladas por Eventos.

interactivo y se usa solo para visualizar. Por ejemplo si usas un JCheckBox como visualización, no podrás consultar si está marcado o no. Jlist no soporta edición como un combobox.

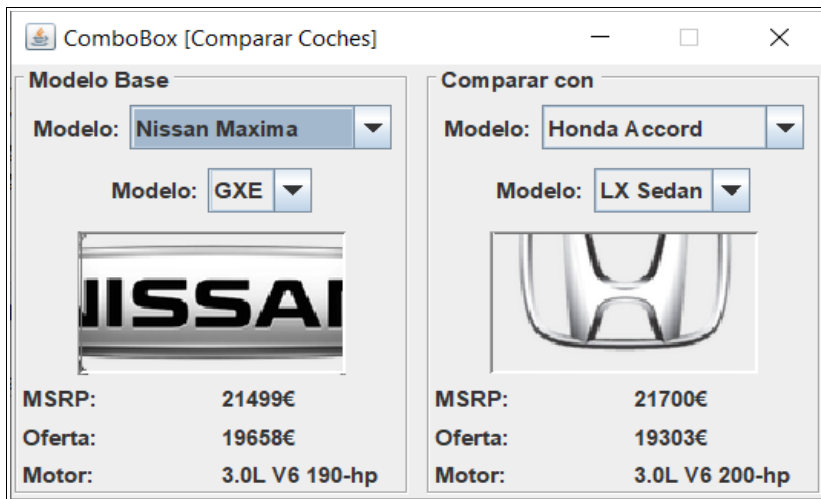


Figura 29: Usar combobox.

Podemos usar el constructor por defecto o pasar una lista de datos como un array unidimensional, como un **Vector**, o como una implementación de la interfaz **ListModel**. La última variante da mayor control sobre la visualización, podemos asignar datos a un JList usando tanto el método **setModel()** o una sobrecarga de **setListData()**.

JList no ofrece acceso directo a sus elementos, podemos acceder a su **ListModel** para acceder a sus datos. Sin embargo si da acceso a sus datos seleccionados implementando los métodos **ListSelectionModel**.

El método **getNextMatch()** devuelve el índice del siguiente elemento de la lista que comience por el prefijo que se le pasa en un string. También usa un índice donde comenzar y una dirección



UNIDAD 8. Aplicaciones Controladas por Eventos.

(`Position.Bias.Forward` o `Position.Bias.Backward`).

JList mantiene los colores de fondo y dibujo y el visualizador por defecto `DefaultListCellRenderer`, los usa para resaltar las celdas seleccionadas. Puedes cambiarlos con `setSelectedForeground()` y `setSelectedBackground()`. El resto de elementos se dibuja con los colores por defecto o los indicados con `setForeground()` y `setBackground()`.

JList implementa la interfaz `Scrollable` pero no lo hace directamente, necesita estar dentro de un `JScrollPane`. La propiedad `visibleRowCount` indica cuantas celdas deben estar visibles si está colocado en un scroll pane. Por defecto es 8 y se puede indicar con `setVisibleRowCount()`. El método `ensureIndexIsVisible()`, fuerza a que el elemento indicado en el parámetro (un índice) sea visible.

Por defecto la altura y anchura de las celdas corresponden a los elementos más altos y anchos. Se puede cambiar este comportamiento indicando un ancho y alto fijo con `setFixedCellWidth()` y `setFixedCellHeight()`.

Otra forma de controlar el ancho y alto de las celdas es el método `setPrototypeCellValue()` que acepta un parámetro `Object` y lo usa para calcular `fixedCellWidth` y `fixedCellHeight`.

A partir de Java 1.4 hay dos nuevos layouts, quedando un total de 3:

- `VERTICAL`: modo mono columna (por defecto).
- `VERTICAL_WRAP`: las celdas fluyen en columnas y es desplazable horizontalmente.
- `HORIZONTAL_WRAP`: las celdas fluyen en filas y las listas se desplazan verticalmente.



UNIDAD 8. Aplicaciones Controladas por Eventos.

El método **locationToIndex()** devuelve el índice de una celda en determinado punto (coordenadas de la lista). -1 si el punto no está en una celda de la lista. Ejemplo para localizar la celda clicada con el ratón:

```
myJList.addMouseListener(  
    new MouseAdapter() {  
        public void mouseClicked(MouseEvent e) {  
            if (e.getClickCount() == 2) {  
                int cellIndex = myJList.locationToIndex(e.getPoint());  
                // We now have the index of the double-clicked cell.  
            }  
        }  
    }  
);
```

La interfaz abstracta **javax.swing.ListModel** aporta los métodos:

- **getElementAt()** para recuperar el elemento de la posición pasada como un Object.
- **getSize()** que devuelve el número de ítems de la lista.
- Contiene dos métodos para informar a los ListDataListeners de adiciones, eliminaciones y cambios en el modelo.

La clase abstracta **javax.swing.AbstractListModel** representa una implementación parcial de la interfaz ListModel. Deja pendiente la implementación de getElementAt() y getSize().

La clase **javax.swing.DefaultListModel** representa una implementación de la interfaz ListModel. Extiende la clase AbstractListModel y usa **java.util.Vector** para almacenar sus datos.

La interfaz abstracta **javax.swing.ListSelectionModel** describe el modelo para seleccionar los elementos de la lista. Define 3 modos de selección: única, única contigua, varios intervalos contiguos. Una selección se define como un rango de índices, o un conjunto de rangos,



UNIDAD 8. Aplicaciones Controladas por Eventos.

o una lista elementos.

El comienzo de un rango seleccionado se denomina **anchor**, y el último elemento del rango se denomina **lead**. El índice más bajo seleccionado se denomina **minimum**, y el mayor **maximum**. Cada uno de estos índices representan propiedades del **ListSelectionModel**. Cuando no hay elementos seleccionados el minimum y el maximum deberían ser -1, y el anchor y el lead mantienen su valor más reciente hasta que no haya nuevas selecciones.

Para cambiar el modo de selección usa el método **setSelectionMode()**, pasando una de las constantes: **MULTIPLE_INTERVAL_SELECTION**, **SINGLE_INTERVAL_SELECTION** O **SINGLE_SELECTION**.

La clase **javax.swing.DefaultListSelectionModel** es la implementación por defecto de la interfaz **ListSelectionModel**. Define métodos para disparar eventos **ListSelectionEvents** cuando hay cambios en un rango de selección.

La interfaz abstracta **javax.swing.ListCellRenderer** describe un componente que se usa para visualizar una lista de elementos (items) y una implementación por defecto llamada **DefaultListCellRenderer**.

La interfaz abstracta **javax.swing.event.ListDataListener** define 3 métodos para entregar **ListDataEvents** cuando se añaden, eliminan o cambian elementos en el **ListModel**: **intervalAdded()**, **intervalRemoved()** y **contentsChanged()**.

La clase **javax.swing.event.ListDataEvent** representa un evento que se genera cuando hay cambios en las listas del **ListModel**. Incluye el origen del evento, los índices mayor y menor elementos afectados y el tipo de evento que puede ser: **CONTENTS_CHANGED**, **INTERVAL_ADDED** e



UNIDAD 8. Aplicaciones Controladas por Eventos.

INTERVAL_REMOVED. Con **getType()** averiguas el tipo de evento.

La interfaz abstracta **javax.swing.event.ListSelectionListener** describe un listener que escucha cambios en las listas del ListSelectionModel. Declara el método **valueChanged()** que acepta un ListSelectionEvent.

La clase **javax.swing.event.ListSelectionEvent** representa un evento del ListSelectionModel cuando hay cambios en una de sus selecciones. Es idéntica a ListDataEvent, salvo que los índices significan donde se ha producido el cambio en el modelo de selección en vez de en el modelo de datos.

EJEMPLO 27: Muestra una lista de estados de EEUU usando un array de strings que tienen el formato <abreviatura de 2 caracteres> <tabulador> <nombre completo del estado> <tabulador> <capital>

```
package listasusa;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class ListaEstadosUSA extends JFrame {
    protected JList m_lista;
    public ListaEstadosUSA() {
        super("Demo de Listas");
        setSize(500, 240);
        String [] estados = { "AK\tAlaska\tJuneau", "AL\tAlabama\tMontgomery",
                              "AR\tArkansas\tLittle Rock", "AZ\tArizona\tPhoenix",
                              "CA\tCalifornia\tSacramento", "CO\tColorado\tDenver",
                              "CT\tConnecticut\tHartford", "DE\tDelaware\tDover",
                              "FL\tFlorida\tTallahassee", "GA\tGeorgia\tAtlanta",
                              "HI\tHawaii\tHonolulu", "IA\tIowa\tDes Moines",
                              "ID\tIdaho\tBoise", "IL\tIllinois\tSpringfield",
                              "IN\tIndiana\tIndianapolis", "KS\tKansas\tTopeka",
                              "KY\tKentucky\tFrankfort", "LA\tLouisiana\tBaton Rouge",
                              "MA\tMassachusetts\tBoston", "MD\tMaryland\tAnnapolis",
                              "ME\tMaine\tAugusta", "MI\tMichigan\tLansing",
                              "MN\tMinnesota\tSt.Paul", "MO\tMissouri\tJefferson City",
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

        "MS\tMississippi\tJackson",      "MT\tMontana\tHelena",
        "NC\tNorth Carolina\tRaleigh",   "ND\tNorth Dakota\tBismarck",
        "NE\tNebraska\tLincoln",          "NH\tNew Hampshire\tConcord",
        "NJ\tNew Jersey\tTrenton",        "NM\tNew Mexico\tSantaFe",
        "NV\tNevada\tCarson City",         "NY\tNew York\tAlbany",
        "OH\tOhio\tColumbus",              "OK\tOklahoma\tOklahoma City",
        "OR\tOregon\tSalem",               "PA\tPennsylvania\tHarrisburg",
        "RI\tRhode Island\tProvidence",    "SC\tSouth Carolina\tColumbia",
        "SD\tSouth Dakota\tPierre",         "TN\tTennessee\tNashville",
        "TX\tTexas\tAustin",               "UT\tUtah\tSalt Lake City",
        "VA\tVirginia\tRichmond",          "VT\tVermont\tMontpelier",
        "WA\tWashington\tOlympia",         "WV\tWest Virginia\tCharleston",
        "WI\tWisconsin\tMadison",           "WY\tWyoming\tCheyenne"
    };

    m_lista = new JList(estados);
    JScrollPane ps = new JScrollPane();
    ps.getViewPort().add( m_lista );
    getContentPane().add(ps, BorderLayout.CENTER);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
}

public static void main(String argv[]) {
    new ListaEstadosUSA();
}
}

```

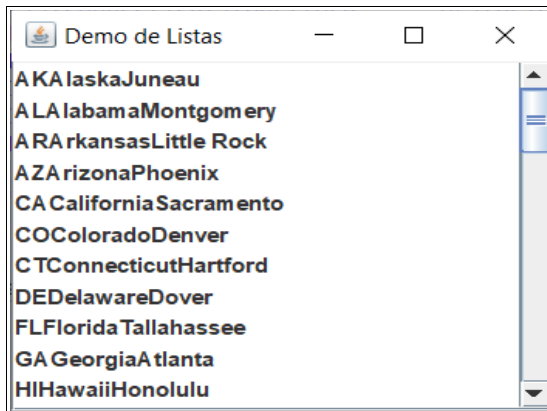


Figura 29: Usar JList.

JSPINNER

Consiste en un área de entrada de texto (por defecto un `JTextField`) y dos pequeños botones con flechas hacia arriba y abajo a la derecha.



UNIDAD 8. Aplicaciones Controladas por Eventos.

Presionando los botones o usando las teclas de arriba y abajo aumentas el valor de una secuencia ordenada de valores. Similar a un `JList` y `JComboBox` pero donde no se necesita desplegar o mostrar los items.

Los items se mantienen en instancias de **`SpinnerModel`** que puedes obtener o establecer con **`setModel()`**/**`getModel()`**. El valor actual se puede cambiar tecleándolo y pulsando enter. Hay implementaciones para los tipos de datos más comunes: **`SpinnerDateModel`**, **`SpinnerListModel`** y **`SpinnerNumberModel`**. El constructor y **`setModel()`** están diseñados para que cambie de editor según los datos usados.

EJEMPLO1: Seleccionar un valor entero desde 0 hasta infinito.

```
package spinner1;

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

class SpinnerDemo1 extends JFrame {
    public SpinnerDemo1() {
        super("Spinner de Números");
        JPanel p = new JPanel();
        p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
        p.setBorder(new EmptyBorder(10, 10, 10, 10));
        p.add(new JLabel("Selecciona un entero: "));
        SpinnerModel model = new SpinnerNumberModel(0, 0, null, 2);
        JSpinner spn = new JSpinner(model);
        p.add(spn);
        getContentPane().add(p, BorderLayout.NORTH);
        setSize(400,75);
    }

    public static void main( String args[] ) {
        SpinnerDemo1 mainFrame = new SpinnerDemo1();
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainFrame.setVisible(true);
    }
}
```




UNIDAD 8. Aplicaciones Controladas por Eventos.

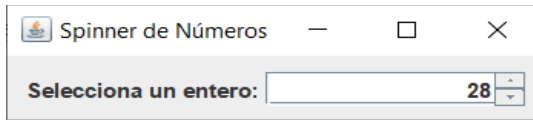


Figura 30: Usar un `SpinnerNumberModel`.

EJEMPLO 28: Un spinner de fechas con intervalos `Calendar.DAY_OF_MONTH`.

```
package spinner2;

import java.awt.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.*;

class SpinnerDemo2 extends JFrame {
    public SpinnerDemo2() {
        super("Spinner de fechas");
        JPanel p = new JPanel();
        p.setLayout(new BorderLayout(p, BorderLayout.X_AXIS));
        p.setBorder(new EmptyBorder(10, 10, 10, 10));
        p.add(new JLabel("Selecciona una fecha: "));
        SpinnerModel model = new SpinnerDateModel(new Date(), null,
            null, //Maximum value - not set
            Calendar.DAY_OF_MONTH // Step
        );
        JSpinner spn = new JSpinner(model);
        p.add(spn);
        getContentPane().add(p, BorderLayout.NORTH);
        setSize(400,75);
    }

    public static void main( String args[] ) {
        SpinnerDemo2 mainFrame = new SpinnerDemo2();
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainFrame.setVisible(true);
    }
}
```

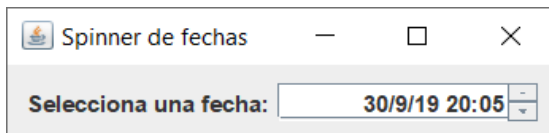


Figura 31: Usar un `SpinnerDateModel`.



UNIDAD 8. Aplicaciones Controladas por Eventos.

JSlider

Una forma visual de seleccionar un valor entero de entre un rango posible desplazando una barra:

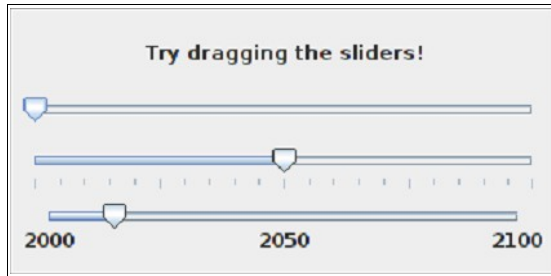


Figura 32: Diferentes sliders.

El constructor más usado indica el comienzo y final del rango de valores:

```
public JSlider(int min, int max, int valor);
```

Si se usa el constructor por defecto se usan los valores 0, 100 y 50. Por defecto es horizontal, si quieres vertical puedes ejecutar `setOrientation(JSlider.VERTICAL)`.

El valor actual puede leerse con `getValue()` y puede cambiarse desde el programa con `setValue(n)`.

Si quieres responder cuando hay un cambio de valor, registras un listener en el slider. Los JSliders no generan eventos `ActionEvent` si no `ChangeEvent` definidos en `javax.swing.event` no en `java.awt.event`. Debes definir un objeto que implemente la interfaz `ChangeListener` y que declara el método `addChangeListener()`:

```
public void stateChanged(ChangeEvent evt)
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

Si quieres saber si es el usuario el que ha generado el evento, puedes llamar al método del slider `getValueIsAdjusting()` que devuelve true si es el usuario quien ha movido la barra.

Para poner marcas a las barras hay que seguir dos pasos: indicar el intervalo entre marcas e indicar las marcas que se quieren. Hay dos tipos (mayores y menores) y con `setMinorTickSpacing(i)` indicas que debe haber una marca menor cada *i* unidades y de forma similar `setMajorTickSpacing(i)`. Y por último, hay que llamar a `setPaintTicks(true)`. Ejemplo:

```
slider2 = new JSlider();
slider2.addChangeListener(this);
slider2.setMajorTickSpacing(25);
slider2.setMinorTickSpacing(5);
slider2.setPaintTicks(true);
```

También puedes poner etiquetas de forma similar a las marcas:

```
sldr.setLabelTable( sldr.createStandardLabels(i) );
```

donde *i* es un entero indicando el espacio entre las etiquetas. Por último llamar a `setPaintLabels(true)`. Ej:

```
slider3 = new JSlider(2000,2100,2014);
slider3.addChangeListener(this);
slider3.setLabelTable( slider3.createStandardLabels(50) );
slider3.setPaintLabels(true);
```

8.4.4. MENÚS, TOOLBARS Y ACCIONES.

MENUS Y BARRAS DE MENUS

Los elementos de un menú se representan con la clase `JMenuItem` empaquetada en `javax.swing`. Los items de menú se usan como los botones. El constructor indica el texto que contiene:

```
JMenuItem miRellenar = new JMenuItem("Rellenar");
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

Puedes añadir un `ActionListener` a un `JMenuItem` llamando a `addActionListener()`. El método `actionPerformed()` del action listener se ejecuta cada vez que el usuario escoja la opción del menú. Puedes cambiar el texto del item con `setText(String)` y puedes activarlo y desactivarlo con `setEnabled(boolean)`.

Un menú se respresenta con la clase `JMenu`. Tiene un nombre que se indica en el constructor y se añaden items con el método `add(JMenuItem)`. Ejemplo de construcción de un menú donde listener es una variable de tipo `ActionListener`:

```
JMenu mtools = new JMenu("Tools");           // Crea un menú
JMenuItem miDraw = new JMenuItem("Draw");     // Crea item
miDraw.addActionListener(listener);           // Asocia listener
mTools.add(miDraw);                           // Añade item a menu
JMenuItem miErase = new JMenuItem("Erase");   // Crea item
miErase.addActionListener(listener);          // Añade listener
mTools.add(miErase);                          // Añade item a menu
:
```

Cuando se crea un menú, puede añadirse a una barrra de menú representada por la clase `JMenuBar`. Su constructor no tiene parámetros y con `add(JMenu)` se añaden menús. El nombre de cada menú aparece en la barra de menús. Ejemplo:

```
JMenuBar menuBar = new JMenuBar();
menuBar.add(mControl);
menuBar.add(mColor);
menuBar.add(mTools);
```

El paso final para definir el menú es añadir la barra de menús a una ventana como un `JFrame`. La barra de menús es otro de los componentes de un frame (no está en el content pane). El `JFrame` tiene el método `setMenuBar(JMenuBar)` para poner la barra de menú que luego puedes obtener con `getMenuBar()`. Ej:



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
MosaicDrawController controller = new MosaicDrawController();
MosaicPanel content = controller.getMosaicPanel();
window.setContentPane( content );
JMenuBar menuBar = controller.getMenuBar();
window.setJMenuBar( menuBar );
```

Hay otros tipos de items definidos como subclases de JMenuItem, como **JCheckBoxMenuItem**, que representa un item que tiene dos estados (seleccionado y no seleccionado) y se usa igual que un JCheckBox. Ej:

```
JMenuItem miRejilla = new JCheckBoxMenuItem("Ver Rejilla");
miRejilla.addActionListener(listener);
miRejilla.setSelected(true);
mControl.add(miRejilla);
```

En **actionPerformed()** del listener se usan sentencias similares a esta para obtener el nombre del origen del evento:

```
String comando = evt.getActionCommand();
```

Si el item de menu es un **JCheckBoxMenuItem**, debe comprobar su estado para saber como responder, en este caso puede usar **evt.getSource()** que devuelve un Object, al que debe hacer un cast de tipos. Ej:

```
if (comando.equals("Ver Rejilla")) {
    JCheckBoxMenuItem cb = (JCheckBoxMenuItem)evt.getSource();
    VerRejilla = cb.isSelected();
}
```

Un menú también puede contener separadores que agrupen varios menu items. Un JMenu tienen un método de instancia **addSeparator()**.

Un menú también puede contener un submenú (en realidad es un JMenu normal). El nombre del submenú aparecerá como si fuera un item del menú principal. Para añadir el submenú simplemente lo pasas en la llamada: **menu.add(submenu)**.



UNIDAD 8. Aplicaciones Controladas por Eventos.

EJEMPLO 29: Crear un sencillo editor de texto que contenga una barra de menús con opciones. Observa el uso de las opciones de menús.

```
package editortexto;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

public class Editor extends JFrame {
    public static final String FONTS[] = { "Serif", "SansSerif",
                                           "Courier" };

    protected Font m_fonts[];
    protected JTextArea m_editor;
    protected JMenuItem[] m_fontMenus;
    protected JCheckBoxMenuItem m_bold;
    protected JCheckBoxMenuItem m_italic;
    protected JFileChooser m_chooser;
    protected File m_currentFile;
    protected boolean m_textChanged = false;
    public Editor() {
        super("Demo de Menús I");
        setSize(450, 350);
        m_fonts = new Font[FONTS.length];
        for (int k=0; k<FONTS.length; k++)
            m_fonts[k] = new Font(FONTS[k], Font.PLAIN, 12);
        m_editor = new JTextArea();
        JScrollPane ps = new JScrollPane(m_editor);
        getContentPane().add(ps, BorderLayout.CENTER);
        JMenuBar menuBar = createMenuBar();
        setJMenuBar(menuBar);
        m_chooser = new JFileChooser();
        try {
            File dir = (new File(".")).getCanonicalFile();
            m_chooser.setCurrentDirectory(dir);
        } catch (IOException ex) {}
        updateEditor();
        newDocument();
        WindowListener wndCloser = new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                if (!promptToSave()) return;
                System.exit(0);
            }
        };
    }
};
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

        addWindowListener(wndCloser);
    }

    protected JMenuBar createMenuBar() {
        final JMenuBar menuBar = new JMenuBar();
        JMenu mFile = new JMenu("Fichero");
        mFile.setMnemonic('f');
        JMenuItem item = new JMenuItem("Nuevo");
        item.setIcon(new ImageIcon("New.gif"));
        item.setMnemonic('n');
        item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_N,
                                                    InputEvent.CTRL_MASK));
        ActionListener lst = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (!promptToSave()) return;
                newDocument();
            }
        };
        item.addActionListener(lst);
        mFile.add(item);
        item = new JMenuItem("Abrir...");
        item.setIcon(new ImageIcon("Open.gif"));
        item.setMnemonic('a');
        item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O,
                                                    InputEvent.CTRL_MASK));
        lst = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (!promptToSave()) return;
                openDocument();
            }
        };
        item.addActionListener(lst);
        mFile.add(item);
        item = new JMenuItem("Guarda");
        item.setIcon(new ImageIcon("Save.gif"));
        item.setMnemonic('g');
        item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S,
                                                    InputEvent.CTRL_MASK));
        lst = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (!m_textChanged) return;
                saveFile(false);
            }
        };
        item.addActionListener(lst);
        mFile.add(item);
        item = new JMenuItem("Guarda como...");
        item.setIcon(new ImageIcon("SaveAs.gif"));
        item.setMnemonic('c');
    }

```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        saveFile(true);
    }
};
item.addActionListener(lst);
mFile.add(item);
mFile.addSeparator();
item = new JMenuItem("Salir");
item.setMnemonic('x');
lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
};
item.addActionListener(lst);
mFile.add(item);
menuBar.add(mFile);
ActionListener fontListener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        updateEditor();
    }
};
JMenu mFont = new JMenu("Fuentes");
mFont.setMnemonic('o');
ButtonGroup group = new ButtonGroup();
m_fontMenus = new JMenuItem[FONTS.length];
for (int k=0; k<FONTS.length; k++) {
    int m = k+1;
    m_fontMenus[k] = new JRadioButtonMenuItem(m + " " +
                                                FONTS[k]);

    m_fontMenus[k].setSelected(k == 0);
    m_fontMenus[k].setMnemonic('1' + k);
    m_fontMenus[k].setFont(m_fonts[k]);
    m_fontMenus[k].addActionListener(fontListener);
    group.add(m_fontMenus[k]);
    mFont.add(m_fontMenus[k]);
}
mFont.addSeparator();
m_bold = new JCheckBoxMenuItem("Negrita");
m_bold.setMnemonic('b');
Font fn = m_fonts[1].deriveFont(Font.BOLD);
m_bold.setFont(fn);
m_bold.setSelected(false);
m_bold.addActionListener(fontListener);
mFont.add(m_bold);
m_italic = new JCheckBoxMenuItem("Italic");
m_italic.setMnemonic('i');
fn = m_fonts[1].deriveFont(Font.ITALIC);

```




UNIDAD 8. Aplicaciones Controladas por Eventos.

```
m_italic.setFont(fn);
m_italic.setSelected(false);
m_italic.addActionListener(fontListener);
mFont.add(m_italic);
menuBar.add(mFont);
return menuBar;
}

protected String getDocumentName() {
    return m_currentFile == null ?
        "Sin título" : m_currentFile.getName();
}

protected void newDocument() {
    m_editor.setText("");
    m_currentFile = null;
    setTitle(" [" + getDocumentName() + "]" );
    m_textChanged = false;
    m_editor.getDocument().addDocumentListener(new UpdateListener());
}

protected void openDocument() {
    if (m_chooser.showOpenDialog(Editor.this) !=
        JFileChooser.APPROVE_OPTION)
        return;
    File f = m_chooser.getSelectedFile();
    if (f == null || !f.isFile()) return;
    m_currentFile = f;
    try {
        FileReader in = new FileReader(m_currentFile);
        m_editor.read(in, null);
        in.close();
        setTitle(" ["+getDocumentName()+"]");
    }
    catch (IOException ex) {
        showError(ex, "Error leyendo fichero " + m_currentFile);
    }
    m_textChanged = false;
    m_editor.getDocument().addDocumentListener(new UpdateListener());
}

protected boolean saveFile(boolean saveAs) {
    if (saveAs || m_currentFile == null) {
        if (m_chooser.showSaveDialog(Editor.this) !=
            JFileChooser.APPROVE_OPTION)
            return false;
        File f = m_chooser.getSelectedFile();
        if (f == null) return false;
        m_currentFile = f;
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
        setTitle(" ["+getDocumentName()+"]");
    }
    try {
        FileWriter out = new FileWriter(m_currentFile);
        m_editor.write(out);
        out.close();
    }
    catch (IOException ex) {
        showError(ex, "Error guardando fichero "+ m_currentFile);
        return false;
    }
    m_textChanged = false;
    return true;
}

protected boolean promptToSave() {
    if (!m_textChanged)
        return true;
    int result = JOptionPane.showConfirmDialog(this,
        "Guardar cambios a " + getDocumentName() + "?",
        "Editor", JOptionPane.YES_NO_CANCEL_OPTION,
        JOptionPane.INFORMATION_MESSAGE);
    switch (result) {
        case JOptionPane.YES_OPTION: if (!saveFile(false)) return false;
                                     return true;
        case JOptionPane.NO_OPTION:  return true;
        case JOptionPane.CANCEL_OPTION: return false;
    }
    return true;
}

protected void updateEditor() {
    int index = -1;
    for (int k=0; k<m_fontMenus.length; k++) {
        if (m_fontMenus[k].isSelected()) {
            index = k;
            break;
        }
    }
    if (index == -1) return;
    if (index==2) { // Courier
        m_bold.setSelected(false);
        m_bold.setEnabled(false);
        m_italic.setSelected(false);
        m_italic.setEnabled(false);
    }
    else {
        m_bold.setEnabled(true);
        m_italic.setEnabled(true);
    }
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```

    }
    int style = Font.PLAIN;
    if (m_bold.isSelected()) style |= Font.BOLD;
    if (m_italic.isSelected()) style |= Font.ITALIC;
    Font fn = m_fonts[index].deriveFont(style);
    m_editor.setFont(fn);
    m_editor.repaint();
}

public void showError(Exception ex, String msj) {
    ex.printStackTrace();
    JOptionPane.showMessageDialog(this, msj, "Editor",
                                JOptionPane.WARNING_MESSAGE);
}

public static void main(String argv[]) {
    Editor frame = new Editor();
    frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    frame.setVisible(true);
}

class UpdateListener implements DocumentListener {
    public void insertUpdate(DocumentEvent e) {
        m_textChanged = true;
    }

    public void removeUpdate(DocumentEvent e) { m_textChanged = true; }

    public void changedUpdate(DocumentEvent e) {
        m_textChanged = true;
    }
}
}

```

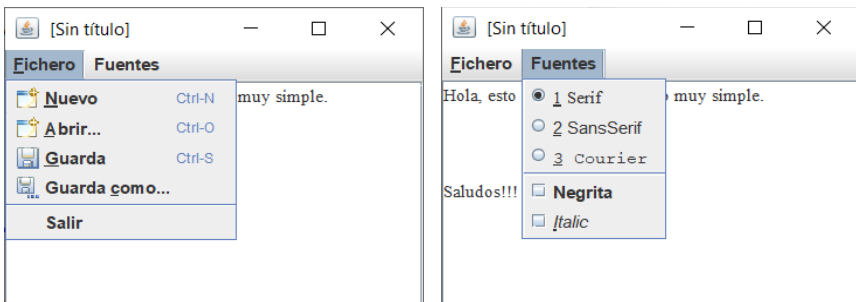


Figura 33: Resultado de crear los menús del ejemplo.



UNIDAD 8. Aplicaciones Controladas por Eventos.

TOOLBARS

La clase **JToolBar** implementa una barra de herramientas (una barra con botones). Swing aporta la interfaz **Action** para simplificar la creación de los items de un menú. Al implementar esta interfaz se encapsula el conocimiento de qué hacer cuando se pulsa una opción de menú o un botón de una toolbar y se encapsula también como visualizar el componente.

Podemos crear tanto un item de menu como un botón de la toolbar a partir de una instancia de **Action**, conservando la sincronización y consistencia entre menús y toolbars de una manera más sencilla.

EJEMPLO 30: Hacer los siguientes cambios al ejemplo anterior para añadir acciones y toolbars. Subrayado en amarillo las partes nuevas del código:

```
package editortexto;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

public class Editor extends JFrame {
    // Código sin cambios aquí...
    protected JToolBar m_toolBar;
    protected JMenuBar createMenuBar() {
        final JMenuBar menuBar = new JMenuBar();
        JMenu mFile = new JMenu("Fichero");
        mFile.setMnemonic('f');
        ImageIcon iconNew = new ImageIcon("New.gif");
        Action actionNew = new AbstractAction("New", iconNew) {
            public void actionPerformed(ActionEvent e) {
                if (!promptToSave())
                    return;
                newDocument();
            }
        };
        JMenuItem item = new JMenuItem(actionNew);
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
item.setMnemonic('n');
item.setAccelerator(KeyStroke.getKeyStroke(
    KeyEvent.VK_N, InputEvent.CTRL_MASK));
mFile.add(item);
ImageIcon iconOpen = new ImageIcon("Open.gif");
Action actionOpen = new AbstractAction("Abrir...", iconOpen) {
    public void actionPerformed(ActionEvent e) {
        if (!promptToSave())
            return;
        openDocument();
    }
};
item = new JMenuItem(actionOpen);
item.setMnemonic('a');
item.setAccelerator(KeyStroke.getKeyStroke(
    KeyEvent.VK_O, InputEvent.CTRL_MASK));
mFile.add(item);
ImageIcon iconSave = new ImageIcon("Save16.gif");
Action actionSave = new AbstractAction("Guarda", iconSave) {
    public void actionPerformed(ActionEvent e) {
        if (!m_textChanged)
            return;
        saveFile(false);
    }
};
item = new JMenuItem(actionSave);
item.setMnemonic('g');
item.setAccelerator(KeyStroke.getKeyStroke(
    KeyEvent.VK_S, InputEvent.CTRL_MASK));
mFile.add(item);
ImageIcon iconSaveAs = new ImageIcon("SaveAs16.gif");
Action actionSaveAs = new AbstractAction("Guarda como...",
    iconSaveAs) {
    public void actionPerformed(ActionEvent e) {
        saveFile(true);
    }
};
item = new JMenuItem(actionSaveAs);
item.setMnemonic('c');
mFile.add(item);
mFile.addSeparator();
Action actionExit = new AbstractAction("Exit") {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
};
item = mFile.add(actionExit);
item.setMnemonic('x');
menuBar.add(mFile);
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
m_toolBar = new JToolBar("Comandos");
JButton btn1 = m_toolBar.add(actionNew);
btn1.setToolTipText("Nuevo texto");
JButton btn2 = m_toolBar.add(actionOpen);
btn2.setToolTipText("Abre fichero");
JButton btn3 = m_toolBar.add(actionSave);
btn3.setToolTipText("Guarda fichero");
// Código sin cambiar...
getContentPane().add(m_toolBar, BorderLayout.NORTH);
return menuBar;
}
// Código sin cambiar...
}
```

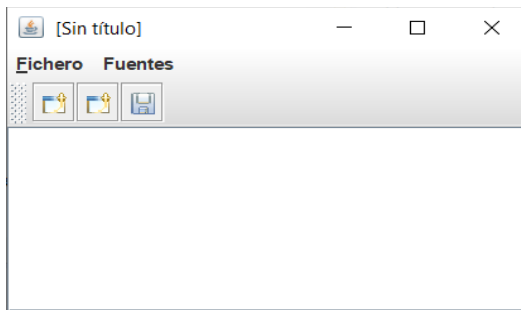


Figura 34: Resultado de crear acciones y toolbar de Ficheros.

7.4.6. DIÁLOGOS.

Un diálogo es un tipo de ventana que se utiliza normalmente para un único y corto propósito de interacción con el usuario. Por ejemplo, mostrar un mensaje, hacer una pregunta, permitir seleccionar un fichero, un color, etc. En Swing, una caja de diálogo se representa con la clase **JDialog** o una de sus subclases.

Un JDialog es muy similar a un JFrame, es una ventana separada. Pero no es completamente independiente. Cada diálogo debe estar asociado a un frame (o a otro diálogo), llamado su **ventana padre**. Si no indicas el padre, se crea un frame invisible para que lo tenga.

Las cajas de diálogo pueden ser **modales** o **no modales**. En caso de ser



UNIDAD 8. Aplicaciones Controladas por Eventos.

modales, el padre queda bloqueado (el usuario no puede interactuar con él) hasta que el diálogo modal se cierre. Los modales son fáciles de usar y son muy comunes.

DIÁLOGOS PREDEFINIDOS

Swing tiene muchos métodos con diálogos prefabricados, por ejemplo:

```
Color JColorChooser.showDialog(Component padre, String titulo,  
                                Color colorinicial);
```

Para elegir un color, al hacer la llamada aparece el diálogo y permite al usuario elegir un color. Cuando se pulsa el botón "OK", el diálogo se cierra y el color seleccionado se devuelve. Si se pulsa el botón "Cancel" o se cierra la ventana de alguna otra forma se devuelve null. Es un diálogo modal así que bloquea la interacción hasta que se cierra.

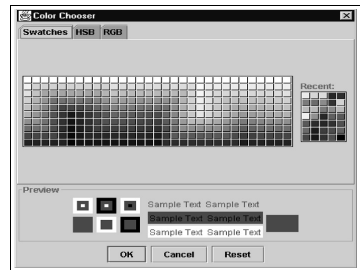


Figura 35: *JcolorChooser.showDialog()*.

JOPTIONPANE

La clase **JOptionPane** es una clase estática (no se instancia para utilizarla). Nos provee un conjunto de ventanas de dialogo para mostrar mensajes al usuario. Ya sean informativos, advertencias, errores, confirmaciones... O incluso tenemos la posibilidad de solicitar la introducción de un dato.

Soporta 4 tipos de diálogos predefinidos: **Message**, **Confirm**, **Input** y **Option**. La encontramos dentro del paquete **javax.swing**. Ejemplo:

```
import javax.swing.JOptionPane;
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
public class Main {  
    public static void main(String[] args){  
        JOptionPane.showMessageDialog(null, "Java!!");  
    }  
}
```

Las diferentes ventanas de dialogo que tiene son:

- **JOptionPane.showMessageDialog()** permite mostrar un mensaje.
- **JOptionPane.showInputDialog()** permite la entrada de datos.
- **JOptionPane.ConfirmDialog()** permite hacer preguntas con varias confirmaciones. Por ejemplo: Sí, No, Cancelar.
- **JOptionPane.showOptionDialog()** engloba los anteriores.

Antes de ver cada tipo, hay que comprender los elementos que tiene. En la figura 36, podemos ver las diferentes partes. Tenemos que escoger un método y ver que parámetros necesitamos rellenar y guiarnos con la información de más abajo.



Figura 36: Partes de un diálogo JOptionPane.

Tipos de iconos de un JOptionPane

Aunque los iconos se pueden personalizar, vamos a explicar los que tenemos por defecto y las diferentes maneras de llamarlos.



UNIDAD 8. Aplicaciones Controladas por Eventos.





Icon	Code	IDE Value
No icon	JOptionPane.PLAIN_MESSAGE	-1
	JOptionPane.ERROR_MESSAGE	0
	JOptionPane.INFORMATION_MESSAGE	1
	JOptionPane.WARNING_MESSAGE	2
	JOptionPane.QUESTION_MESSAGE	3

Figura 37: tipos de iconos de JOptionPane.

Para completar la estructura mira la figura 38 que describe las áreas.

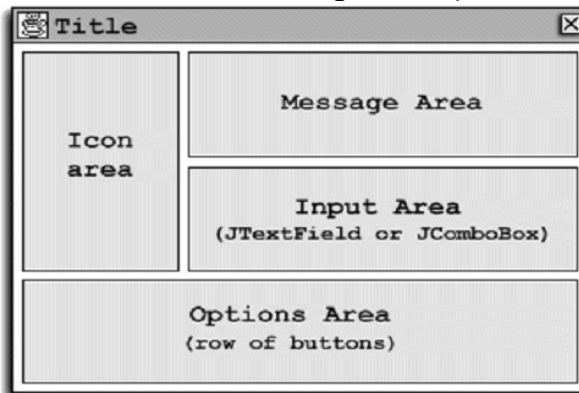


Figura 38: Áreas de los Componentes de JOptionPane.

Recapitulando, para crear un **JOptionPane** necesitamos aportar alguno de los siguientes elementos al método estático **showXXDialog()** donde XX puede ser alguno de los tipos predefinidos:

- **Un componente padre:** si es un Frame, aparece centrado con respecto a él. Si es null, aparece centrado en la pantalla.
- **Un mensaje** de tipo Object para mostrarlo. Normalmente es un



UNIDAD 8. Aplicaciones Controladas por Eventos.

string que puede tener varias líneas separadas por “\n”. También se suele utilizar:

- Icon: se muestra en un JLabel.
 - Component: se coloca en el área de mensajes.
 - Object[] : se coloca en una columna vertical.
 - Object: se usa su método toString().
- **Tipo de mensaje:** un int que puede ser una de las constantes de JOptionPane: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` o `PLAIN_MESSAGE`. Se usa para personalizar el diálogo y hace aparecer el icono apropiado.
 - **Opción:** un int que puede tomar los valores de JOptionPane: `DEFAULT_OPTION`, `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION` o `OK_CANCEL_OPTION`. Indica los botones que se muestran en el panel (en el área Options).

Nota: tras llamar a alguno de los métodos `showXXDialog()` se devuelve un valor similar a opción que indica el botón que se ha pulsado para cerrar el diálogo: `CANCEL_OPTION`, `CLOSED_OPTION`, `NO_OPTION`, `OK_OPTION` y `YES_OPTION`. Ten en cuenta que `CLOSED_OPTION` solamente se devuelve si el diálogo se cierra desde la barra de título.

- **Icono:** se muestra un icono en la parte izquierda del diálogo (área Icon). Si no se indica ninguno de forma explícita, se usa uno según el tipo de mensaje (salvo que el tipo sea `PLAIN_MESSAGE`).
- **Un array de opciones Objects.** Se pueden indicar un array de opciones de tipo Object mostrados al final del panel (área Options). También se puede indicar usando el método `setOptions()`. Normalmente contiene un array de strings que se



UNIDAD 8. Aplicaciones Controladas por Eventos.

muestran como `JButtons`.

- **Valor inicial:** un valor `Object` indica la opción que recibe el foco en principio.
- **Título:** el título de la ventana `JDialog` o `JInternalFrame`.

Los siguientes métodos estáticos de **`JOptionPane`** permiten crear `JDialogs`:

- **`showMessageDialog()`**: muestra un diálogo con un mensaje y un botón OK, no devuelve nada. Hay 3 métodos sobrecargados para indicar al menos el componente padre, el mensaje, un título, el tipo de mensaje y el icono.

```
import javax.swing.JOptionPane;
```

```
public class Main {  
    public static void main(String[] args){  
        JOptionPane.showMessageDialog(null, "Java!!");  
    }  
}
```

```
// Ejemplo con 4 parámetros
```

```
import javax.swing.JOptionPane;
```

```
public class Main {  
    public static void main(String[] args){  
        JOptionPane.showMessageDialog(null, "Msj sin icono", "Java!!", -1);  
        JOptionPane.showMessageDialog(null, "Msj ERROR_MESSAGE",  
            "Java!!", 0);  
        JOptionPane.showMessageDialog(null, "Msj INFORMATION_MESSAGE",  
            "Java!!", 1);  
        JOptionPane.showMessageDialog(null, "Msj WARNING_MESSAGE",  
            "Java!!",  
                JOptionPane.WARNING_MESSAGE);  
        JOptionPane.showMessageDialog(null, "Msj QUESTION_MESSAGE",  
            "Java!!", JOptionPane.QUESTION_MESSAGE);  
    }  
}
```

- **`showConfirmDialog()`**: muestra un diálogo con varios botones y devuelve un entero con el botón pulsado. Hay 4 métodos



UNIDAD 8. Aplicaciones Controladas por Eventos.

sobrecargados que permiten indicar un padre, un mensaje, el título, el tipo de opción y el icono.

```
import javax.swing.JOptionPane;

public class Main {
    public static void main(String[] args) {
        int dYNC = JOptionPane.showConfirmDialog(null,
                                                "¿Sale del programa?",
                                                "Java!!",
                                                JOptionPane.YES_NO_CANCEL_OPTION,
                                                JOptionPane.WARNING_MESSAGE);

        //0=yes, 1=no, 2=cancel
        if( dYNC == 0 ) { System.out.println("Ha pulsado Yes"); }
        else if( dYNC == 1 ) { System.out.println("Ha pulsado No"); }
        else if( dYNC == 2 ) { System.out.println("Ha pulsado Cancel"); }
    }
}
```

- **showInputDialog()**: muestra un mensaje para permitir leer un string tecleado por el usuario (si el componente es una caja de texto) o un Object si el componente del diálogo es una lista o un combo. Hay 4 métodos sobrecargados para indicar el padre, el mensaje, el título, el tipo de opción, el icono, array de posibles selecciones y el elemento inicialmente seleccionado. Siempre aparecen dos botones, el de OK y el de Cancel.

```
import javax.swing.JOptionPane;

public class Main {
    public static void main(String[] args) {
        String m = JOptionPane.showInputDialog(null, "Su nombre");
        System.out.println( m );
    }
}
```

- **showOptionDialog()**: este método permite personalizar un poco más que los anteriores el diálogo. Devuelve un índice con la opción escogida del array de Objects de opciones proporcionado o un tipo de opción si no se indican los Object.



UNIDAD 8. Aplicaciones Controladas por Eventos.

Solo hay un método donde hay que indicar el padre, el mensaje, el título, el tipo de opción, el tipo de mensaje, icono, array de opciones y una opción `Object` con el foco.

```
import javax.swing.JOptionPane;

public class Main {
    public static void main(String[] args) {
        String[] botones = {"Boton A", "Boton B", "Boton C"};
        int = JOptionPane.showOptionDialog(null,
                                           "Pulsa un boton:",
                                           "Java!!",
                                           JOptionPane.DEFAULT_OPTION,
                                           JOptionPane.QUESTION_MESSAGE,
                                           null,
                                           botones, botones[0] );

        if( d == 0 ) { System.out.println("Opcion A"); }
        else if( d == 1 ) { System.out.println("Boton B"); }
        else if( d == 2 ) { System.out.println("Boton C"); }
    }
}
```

Para crear `JOptionPanes` contenidos en `JInternalFrames` en vez de `JDialogs`, podemos usar los métodos `showInternalConfirmDialog()`, `showInternalInputDialog()`, `showInternalMessageDialog()` y `showInternalOptionDialog()`. Funcionan igual que los otros pero el padre será un `JDesktopPane` (o un descendiente suyo).

Nota: *los diálogos internos no son modales.*

Otra opción es crear nosotros mismos un `JOptionPane`, como en este ejemplo:

```
JOptionPane pane = new JOptionPane(...);
pane.setXX(...);
JDialog dialog = pane.createDialog(padre, titulo);
dialog.show();
// Procesar el resultado
Object result = pane.getValue();
```



8.5. IMAGENES Y RECURSOS.

Java incluye clases y subrutinas usadas para leer los datos de las imágenes desde ficheros, mover esos datos y mostrarlos.

IMAGE y BUFFEREDIMAGE

La clase `java.awt.Image` representa una imagen almacenada en la memoria del ordenador. Hay dos grandes tipos de imágenes, el primero representa una leída desde el exterior (como un fichero) y el segundo tipo es una creada por el propio programa (usando un contexto gráfico pero que no es visible).

Una imagen de ambos tipos puede copiarse sobre la pantalla o sobre un canvas fuera de la pantalla (memoria RAM) usando los métodos definidos en la clase **Graphics**. Lo normal es hacerlo en el método `paintComponent()` de un `JComponent`. Supongamos que `g` es el objeto `Graphics` que se pasa como parámetro del método `paintComponent()`, y que `img` es un objeto de tipo `Image`. Entonces, la sentencia:

```
g.drawImage(img, x, y, this);
```

dibuja la imagen en el área rectangular cuya esquina izquierda comienza en las coordenadas (x,y) del componente. El cuarto parámetro se usa por motivos técnicos (en nuestro caso a null).

***Nota:** el cuarto parámetro es de tipo `ImageObserver` y solo se necesita cuando puede que no estén disponibles los datos de la imagen al hacer la llamada a `drawImage()` al tener que descargarse el fichero imagen por la red. Cualquier `JComponent` puede actuar como un `ImageObserver`.*

Hay unas cuantas variaciones del método `drawImage()`. Por ejemplo, es posible escalar las imágenes para que ocupen cierta anchura y altura:



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
g.drawImage(img, x, y, ancho, alto, imageObserver);
```

Otras versiones permiten dibujar solamente una parte de la imagen, por ejemplo:

```
g.drawImage(img, dest_x1, dest_y1, dest_x2, dest_y2,  
source_x1, source_y1, source_x2, source_y2, imageObserver);
```

Los enteros source definen el rectángulo a copiar de la imagen, los parámetros dest indican en qué rectángulo dibujarlos del contexto gráfico g.

EJEMPLO 31: si un juego de cartas necesita tener las imágenes de todas las cartas, puede usar una sola imagen con todas ellas y si tienen las mismas dimensiones, puede ir dibujando la que necesite en cada momento. Esta imagen es del proyecto Gnome desktop, <http://www.gnome.org>, dibujar una sola carta exige coger un trozo de la imagen y copiarla a otro lugar (el destino).

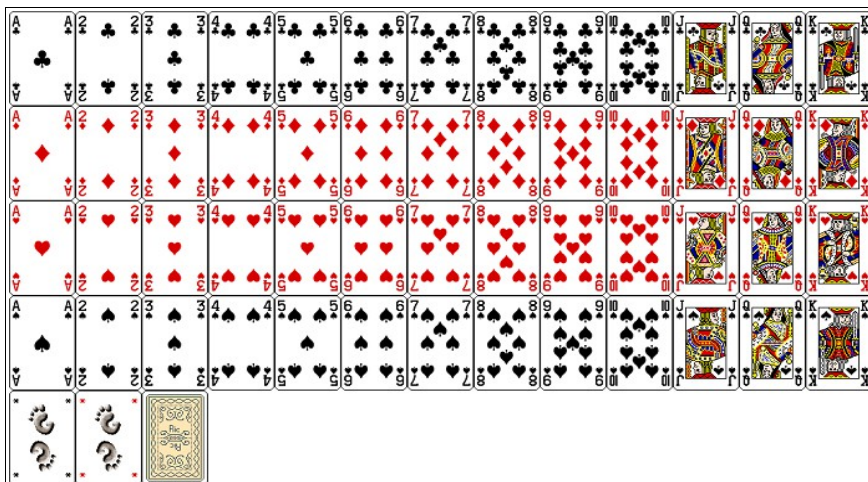


Figura 36: Copiar trozos de imagen.



UNIDAD 8. Aplicaciones Controladas por Eventos.

En este trozo de código, suponiendo que cada carta tenga 79x123 pixels de tamaño, usamos el valor de cada carta y el palo (trebol, rombos, corazones, comodines y el anverso para elegir el trozo a copiar:

```
/**
 * Dibujar una carta
 */
public void dibujaCarta(Graphics g, Carta c, int x, int y) {
    int cx;    // coordenada x
    int cy;    // coordenada y
    if (c == null) {
        cy = 4 * 123;    // anverso
        cx = 2 * 79;
    }
    else {
        cx = (c.getValor() - 1) * 79;
        switch ( c.getPalo() ) {
            case Carta.TREBOL: cy = 0;
                                break;
            case Carta.DIAMANTES: cy = 123;
                                break;
            case Carta.CORAZONES: cy = 2 * 123;
                                break;
            default: // picas
                    cy = 3*123;
                    break;
        }
    }
    g.drawImage(imgBaraja, x, y, x+79, y+123, cx, cy, cx+79, cy+123, this);
}
```

Además de usar imágenes cargadas desde ficheros, un programa puede trabajar con imágenes dibujadas en la memoria, en un canvas representado en la clase **java.awt.image.BufferedImage**, (una subclase de Image) que se construye con el constructor **BufferedImage(int width, int height, int imageType)** donde el tipo de imagen puede ser alguna de las constantes predefinidas e indica como se representa el color de cada pixel:

- **BufferedImage.TYPE_INT_RGB**, cada pixel es un color RGB (componentes rojo, verde y azul con valores 0 a 255).



UNIDAD 8. Aplicaciones Controladas por Eventos.

- **BufferedImage.TYPE_INT_ARGB** un RGB con transparencia.
- **BufferedImage.TYPE_BYTE_GRAY** escala de grises.

Si OSC es de tipo **BufferedImage**, el método **OSC.createGraphics()** devuelve un **Graphics2D** que puedes usar para dibujar en la imagen. Ej:

```
/**
 * crea un canvas off-screen para rellenarlo con el color actual.
 */
private void creaOSC() {
    OSC = new
    BufferedImage(getWidth(),getHeight(),BufferedImage.TYPE_INT_RGB);
    Graphics osg = OSC.createGraphics();
    osg.setColor(fillColor);
    osg.fillRect(0, 0, getWidth(), getHeight() );
    osg.dispose();
}
```

Se utiliza **OSC.createGraphics()** para obtener un contexto gráfico para dibujar en la imagen. Observa que este contexto se destruye (dispose) al final, es una buena idea hacerlo en cuanto no lo necesites.

Este código tiene un problema: si se llama en el constructor, la anchura y la altura obtenida con `getWidth()` y `getHeight()` devolverá un valor cero, y no crearemos una imagen del tamaño correcto. Para solucionarlo, puedes llamarlo en el método **paintComponent()** la primera vez que sea llamado:

```
public void paintComponent(Graphics g) {
    if(OSC == null) createOSC();
    g.drawImage(OSC, 0, 0, null);
    if ( dragging && SHAPE_TOOLS.contains(currentTool) ) {
        g.setColor(currentColor);
        putCurrentShape(g);
    }
}
```

Un uso típico de los canvas fuera de pantalla es implementar el **doble**



UNIDAD 8. Aplicaciones Controladas por Eventos.

buffering, donde la imagen fuera de la pantalla es una copia exacta de la imagen de la pantalla, y si hay que cambiarla los cambios se realizan fuera de la pantalla y luego se copia el resultado (mejora las animaciones).

TRABAJAR CON PIXELS

Si necesitas explorar o hacer cambios a los pixels de una imagen, usar una **BufferedImage** es la solución porque puedes usar dos métodos para leer y cambiar cada uno de sus píxeles:

- **image.getRGB(x,y)** — devuelve un entero que codifica el color del pixel en las coordenadas (x,y) de la imagen, donde se cumple que $0 \leq x < \text{image.getWidth}()$ y $0 \leq y < \text{image.getHeight}()$.
- **image.setRGB(x,y,rgb)** — cambia el color del pixel de la imagen de coordenadas (x,y) al color codificado en el entero rgb.

Estos métodos usan enteros para codificar el color. Si *c* es una variable de tipo **Color**, el entero que lo codifica se obtiene ejecutando **c.getRGB()** y la operación contraria la puedes realizar con el constructor **new Color(rgb)**. Teniendo esto en cuenta, las siguientes sentencias obtienen y fijan el color de un pixel de dos imágenes:

```
Color c = new Color( image1.getRGB(x,y) );  
image2.setRGB( x, y, c.getRGB() );
```

Al codificar los componentes de un color RGB en un entero, el valor blue está en el byte menos significativo, el green en el siguiente, y el red en el siguiente (los 8 bits más significativos del entero almacenan el componente alfa, la transparencia). Usando los operadores binarios de desplazamiento (\ll y \gg) y lógicos ($\&$ y $|$) es sencillo manipular cada componente por separado evitando definir un objeto **Color**. Ej:

```
int red = (rgb >> 16) & 0xFF;
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
int green = (rgb >> 8) & 0xFF;
int blue = rgb & 0xFF;
int rgb = (red << 16) | (green << 8) | blue; / Tb. puedes usar +
```

Por ejemplo, para emborronar o difuminar una imagen por donde pases el ratón, en ese momento se copia un rectángulo de por ejemplo 7x7 píxels fuera de pantalla a 3 arrays llamados sRed, sGreen y sBlue en el método **mousePressed()**. El array es de tipo double[][] para que al hacer cálculos se mantenga cierta precisión.

```
int w = OSC.getWidth();
int h = OSC.getHeight();
int x = evt.getX();
int y = evt.getY();
for (int i = 0; i < 7; i++)
    for (int j = 0; j < 7; j++) {
        int r = y + j - 3;
        int c = x + i - 3;
        if (r < 0 || r >= h || c < 0 || c >= w) {
            // Un -1 en el array indica que el pixel
            // está fuera de la imagen
            sRed[i][j] = -1;
        }
        else {
            int color = OSC.getRGB(c,r);
            sRed[i][j] = (color >> 16) & 0xFF;
            sGreen[i][j] = (color >> 8) & 0xFF;
            sBlue[i][j] = color & 0xFF;
        }
    }
}
```



Figura 37: Imagen emborronada.



UNIDAD 8. Aplicaciones Controladas por Eventos.

Los colores de los pixels se reemplazan por una media ponderada entre los colores actuales de la imagen y los colores del array. Esto tiene el efecto de mover algunos colores desde la posición anterior del ratón a la nueva posición. Al mismo tiempo, los colores del array se cambian por una media de los anteriores colores del array con los colores de la imagen. Así se mueven algunos colores desde la imagen hasta el array:

```
int curCol = OSC.getRGB(c,r);
int curRed = (curCol >> 16) & 0xFF;
int curGreen = (curCol >> 8) & 0xFF;
int curBlue = curCol & 0xFF;
int newRed = (int)(curRed * 0.7 + sRed[i][j] * 0.3);
int newGreen = (int)(curGreen * 0.7 + sGreen[i][j] * 0.3);
int newBlue = (int)(curBlue * 0.7 + sBlue[i][j] * 0.3);
int newCol = newRed << 16 | newGreen << 8 | newBlue;
OSC.setRGB(c, r, newCol);
sRed[i][j] = curRed * 0.3 + sRed[i][j] * 0.7;
sGreen[i][j] = curGreen * 0.3 + sGreen[i][j] * 0.7;
sBlue[i][j] = curBlue * 0.3 + sBlue[i][j] * 0.7;
```

RECURSOS

Con recursos nos referimos a datos que sean sonidos, vídeos, ficheros de imágenes o de otros datos, etc. Cuando uno de estos ficheros es una parte de un programa (es necesario para que el programa funcione) se le llama recurso del programa.

Los recursos suelen almacenarse en ficheros que se encuentran en el mismo lugar que las clases que lo usan, así que el usuario no necesita preocuparse de buscarlo. Los ficheros .class los cargan un cargador de clases (objeto de tipo **ClassLoader**). Tiene una lista de lugares donde buscar los ficheros. Esta lista es la ruta de las clases (**class path**) y generalmente incluye el directorio actual.

Si el programa está almacenado en un fichero .jar, el fichero está



UNIDAD 8. Aplicaciones Controladas por Eventos.

incluido en la class path. Además de ficheros de clases, los `ClassLoader` son capaces de cargar ficheros de recursos:

```
ClassLoader cl = getClass().getClassLoader();
URL imagenURL = cl.getResource("cartas.png");
Image cartas = Toolkit.getDefaultToolkit().createImage(imagenURL);
```

Otra forma de obtener un cargador de clases es usar el nombre de una clase, por ejemplo **`MiClase.getClassLoader()`**. Si el recurso indicado en la segunda línea no puede encontrarse, se devuelve `null`. Por último, puedes usar la URL para crear el objeto que contenga el recurso.

Los sonidos son otro tipo de recurso, el programa puede usar el método estático de la clase `java.awt.Applet` para crear un objeto que representa un sonido:

```
public static AudioClip newAudioClip(URL soundURL)
```

EJEMPLO 32: reproducir un audio.

```
private void playAudioResource(String recursoAudioNombre) {
    ClassLoader cl = getClass().getClassLoader();
    URL url = cl.getResource(recursoAudioNombre);
    if (url != null) {
        AudioClip sonido = Applet.newAudioClip(url);
        sonido.play();
    }
}
```

La clase **`AudioClip`** soporta ficheros de audio como WAV, AIFF y formatos AU.

CURSORES E ICONOS

La posición del puntero del ratón se indica mediante una pequeña imagen llamada cursor. En Java, un cursor es un objeto de la clase **`java.awt.Cursor`**. Un `Cursor` tiene asociada una imagen y un spot, el



UNIDAD 8. Aplicaciones Controladas por Eventos.

pixel de la imagen que indica su posición exacta. Por ejemplo en una flecha, el spot suele coincidir con la punta de la flecha, y en una cruz con el centro de la misma.

La clase **Cursor** define varios cursores estándar identificados por constantes como **Cursor.CROSSHAIR_CURSOR** y **Cursor.DEFAULT_CURSOR**. Puedes obtener el cursor estándar usando el método estático **Cursor.getPredefinedCursor(code)**, donde code es una de las constantes de tipo cursor. También puedes crear cursores personalizados a partir de una imagen que podrías tener como un recurso del programa. Los cursores suelen ser pequeños, de 16x16 o 24x24 pixels. Un cursor personalizado se crea con el método estático **createCustomCursor()** de la clase **Toolkit**. Ej:

```
Cursor c= Toolkit.getDefaultToolkit().createCustomCursor(img,spot,nomb);
```

spot es de tipo **Point** y nomb es un **String**.

Puedes asociar un cursor a un componente llamando al método **setCursor(cursor)** del componente. Ej:

```
panel.setCursor( Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR) );
```

Para volver a asociarle el cursor por defecto:

```
panel.setCursor( Cursor.getDefaultCursor() );
```

EJEMPLO 33: poner de cursor una imagen de un fichero de recursos.

```
ClassLoader cl = getClass().getClassLoader();
URL url = cl.getResource(recurso);
if (resourceURL != null) {
    Toolkit toolkit = Toolkit.getDefaultToolkit();
    Image img = toolkit.createImage(url);
    Point hs = new Point(7,7);
    Cursor cursor = toolkit.createCustomCursor(img, hs, "cursor");
    panel.setCursor(cursor);
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
}
```

Un icono es una pequeña imagen representada por un objeto de tipo **Icon**, que es una interfaz, no una clase. La clase **ImageIcon**, implementa esa interfaz y sirve para crear iconos a partir de imágenes **new ImageIcon(image)**.

E/S DE FICHEROS IMAGEN

La clase **javax.imageio.ImageIO** simplifica la carga y almacenamiento de imágenes desde un programa. Encapsula los formatos de imagen más comunes: PNG, JPEG y GIF.

El método estático **ImageIO.write()** devuelve false si el formato no se soporta. Ej:

```
boolean tienesFormato = ImageIO.write(OSC, formato, fichero);
if ( ! tienesFormato )
    throw new Exception("Formato " + formato + " no disponible.");
```

EJEMPLO 34: Almacenar una imagen en un formato estándar.

```
/**
 * Intenta almacenar una imagen.
 * @param img la imagen BufferedImage a guardar
 * @param formato como "PNG" o "JPEG"
 */
private void doSaveFile(BufferedImage img, String formato) {
    if (fd == null)
        fd = new JFileChooser();
    fd.setSelectedFile(new File("img." + formato.toLowerCase()));
    fd.setDialogTitle("Seleccione Fichero para guardar imagen");
    int opcion = fd.showSaveDialog(this);
    if (opcion != JFileChooser.APPROVE_OPTION)
        return; // Usuario cancela la operación
    File sf = fd.getSelectedFile();
    if (sf.exists()) {
        int res = JOptionPane.showConfirmDialog(null,
            "Fichero \"" + sf.getName() +
            "\" existe. ¿Lo reemplaza?",
            "Confirma guardar",
            JOptionPane.YES_NO_OPTION,
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
        JOptionPane.WARNING_MESSAGE);
    if (res != JOptionPane.YES_OPTION)
        return; // No quiere reemplazar
    }
    try {
        boolean tienesFormato = ImageIO.write(img, formato, sf);
        if ( ! tienesFormato )
            throw new Exception(formato + " no disponible.");
    }
    catch (Exception e) {
        JOptionPane.showMessageDialog(this,
            "Lo siento, ha ocurrido un error al guardar.");
        e.printStackTrace();
    }
}
```

También tiene el método estático **ImageIO.read(inputFile)** que acepta un **File** y devuelve un **BufferedImage**. Devuelve **null** si el formato no es soportado o el fichero no es una imagen.

TRANSPARENCIA

Si un color se codifica en un int, y cada uno de los 3 componentes RGB tiene 8 bits, nos quedan 8 bits libres que pueden utilizarse para el componente alfa (puede tener varios usos y uno de ellos es la transparencia). Cuando dibujas con un color transparente, no oculta totalmente lo que haya debajo, si no que se mezcla como si dibujases una capa de plástico.

El máximo valor del componente alfa (255) indica que el color es completamente opaco y el mínimo valor (0) indica que es completamente transparente. Los valores intermedios ocultarán más lo que hay debajo a medida que sea más alto.

Los colores con transparencia aparecen con **Graphics2D**, pero pueden utilizarse en objetos **Graphic**. Para indicar un color con componente alfa puedes usar uno de los siguientes constructores de **Color**:



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
public Color(int red, int green, int blue, int alpha);  
public Color(float red, float green, float blue, float alpha);
```

En el primero, todos los parámetros son enteros de 0 a 255, y en el segundo son float de 0.0 a 1.0. Ej:

```
Color rojoTransparente = new Color( 255, 0, 0, 200 );  
Color cyanTransparente = new Color( 0.0F, 1.0F, 1.0F, 0.5F);
```

Una vez que tienes un color transparente *c*, puedes usarlo como otro cualquiera para dibujar en un contexto gráfico *g*, por ejemplo con ***g.setColor(c)*** y una operación de dibujo.

En una imagen de tipo **BufferedImage** si es de tipo **BufferedImage.TYPE_INT_ARGB** se pueden usar colores transparentes. El color de cada pixel puede tener su propio componente alfa.

Si guardas la imagen en un fichero, según el formato que elijas, la información sobre la transparencia puede perderse, por ejemplo PNG soporta transparencia pero JPEG no.

Una imagen ARGB BufferedImage es completamente transparente al crearse, pero si quieres establecer un fondo transparente puedes usar métodos de la clase **Graphics2D** como **setBackground()** y **clearRect()**. Ej:

```
BufferedImage i = new BufferedImage(w, h,  
                                     BufferedImage.TYPE_INT_ARGB);  
Graphics2D g2 = i.createGraphics();  
g2.setBackground( new Color(0,0,0,0) ); // (R, G, B, A)  
g2.clearRect(0, 0, w, h);
```

Poner un color transparente en un pixel no hace que el pixel lo sea, solamente cuando dibujas se observa. Además, la transparencia de un pixel no cambia cuando se realiza una operación de dibujo.



UNIDAD 8. Aplicaciones Controladas por Eventos.

EJEMPLO:

```
private void usaCursorCuadrado() {
    BufferedImage i =
        new BufferedImage(24,24,BufferedImage.TYPE_INT_ ARGB);
    Graphics2D g2 = i.createGraphics();
    g2.setBackground( new Color(0,0,0,0) );
    g2.clearRect(0, 0, 24, 24);
    g2.setColor(Color.RED);
    g2.drawRect(0,0,23,23);
    g2.drawRect(1,1,21,21);
    g2.drawRect(2,2,19,19);
    g2.dispose();
    Point hotSpot = new Point(12,12);
    Toolkit tk = Toolkit.getDefaultToolkit();
    Cursor cursor = tk.createCustomCursor(i, hotSpot, "cuadrado");
    setCursor(cursor);
}
```

TRAZOS (Strokes) Y PINTURAS EN LAS LÍNEAS

Al dibujar con la clase **Graphics**, las líneas serán sólidas de un pixel de anchura. Con la clase **Graphics2D** puedes cambiar el trazo modificando su ancho, y el tipo de trazo (punteadas, discontinuas, etc.). Un objeto de la clase **Stroke** contiene la información de como dibujar (ancho y patrón). Cada objeto **Graphics2D** tiene asociado un objeto **Stroke**.

Stroke es una interfaz y la clase **BasicStroke** la implementa. Ej:

```
BasicStroke linea3 = new BasicStroke(3); // linea 3 pixels ancho
```

Si g2 es de tipo **Graphics2D**, puedes cambiar el trazo con **setStroke()**:

```
g2.setStroke(linea3)
```

El ancho de la línea es un **float** no un entero (si tiene decimales activa el antialiasing):

```
g2.setStroke( new BasicStroke(2.5F) ); // Se activa antialiasing
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

También tienes opciones para indicar como deben acabar los extremos de las línea. La figura muestra varias posibilidades. Igual ocurre al dibujar las líneas que se unen de una figura. Un constructor con más opciones es:

```
public BasicStroke( float width, int capType, int joinType,  
                   float miterlimit, float[] dashPattern,  
                   float dashPhase );
```

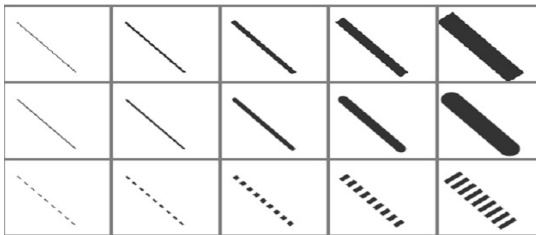


Figura 38: Ejemplos de trazos.

Una explicación básica de los parámetros:

- **width:** anchura del trazo.
- **CapType:** como acaba una línea. Los posibles valores son `BasicStroke.CAP_SQUARE` (por defecto se usa esta), `BasicStroke.CAP_ROUND` y `BasicStroke.CAP_BUTT`. En la figura la primera, segunda y tercera filas usan estos valores respectivamente.
- **JoinType:** como se unen 2 líneas para formar una esquina. Los valores posibles son `BasicStroke.JOIN_MITER` (por defecto), `BasicStroke.JOIN_ROUND` y `BasicStroke.JOIN_BEVEL`.
- **MiterLimit:** cuando el `joinType` es `JOIN_MITER` se usa el valor 10.0F.
- **DashPattern:** se usa para indicar trazos punteados o rayas. Los valores en el array indican longitudes de puntos y rayas. Un primer valor indica la longitud de de un trozo sólido, seguido de



UNIDAD 8. Aplicaciones Controladas por Eventos.

la longitud de un trozo transparente, seguido de un trozo sólido y así.

- **DashPhase**: indica donde comenzar en el patrón.

En la figura, la tercera fila tiene un `dashPattern new float[] {5,5}`. Una línea punteada de ancho 1 debería ser `new float[] {1,1}`. Un patrón de líneas cortas y largas podría ser `new float[] {10,4,4,4}`.

Para que las líneas tengan un color distinto al negro, debemos usar el objeto **Paint**, que es una interfaz que implementa la clase **Color**. También están las clases **TexturePaint** y varios tipos de gradientes. En una textura, el color de cada pixel dibujado proviene de una imagen que se repite para cubrir la zona dibujada. En un gradiente, el color aplicado a cada pixel va cambiando gradualmente de un color inicial a otro final a medida que se cambia de pixel. Java tiene 3 tipos de gradientes: **GradientPaint**, **LinearGradientPaint** y **RadialGradientPaint**.

8.6. DISEÑO DE INTERFACES DESDE UN IDE.

WindowBuilder es un potente y sencillo diseñador de interfaces GUI que ahorra mucho tiempo de pruebas y líneas de código simplemente para mostrar un formulario en una ventana. Tu defines la interfaz con sus componentes de forma interactiva y la herramienta genera el código Java necesario para definir la interfaz. También puedes programar los manejadores de eventos y escribir el código Java de respuesta ante su aparición. Tienes disponibles una gran variedad de controles que puedes personalizar en tiempo de diseño usando el **editor de propiedades**.

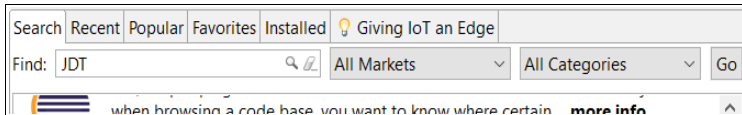
Es un plug-in del IDE eclipse y otros que se basan en él (RAD, RSA, MyEclipse, JBuilder, etc.).



UNIDAD 8. Aplicaciones Controladas por Eventos.

INSTALACIÓN Y PRIMEROS PASOS

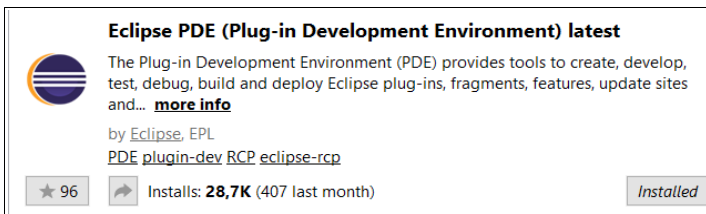
Necesita algunos elementos adicionales. En primer lugar vamos a **Help / MarketPlaces**, escribimos JDT en la caja Find y pulsa Go.



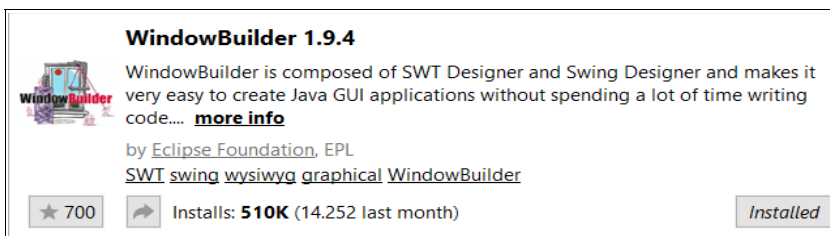
Buscamos el elemento de la imagen y pulsamos en install.



Ahora acemos lo mismo con PDE:



Y por último hacemos lo mismo con "window builder"

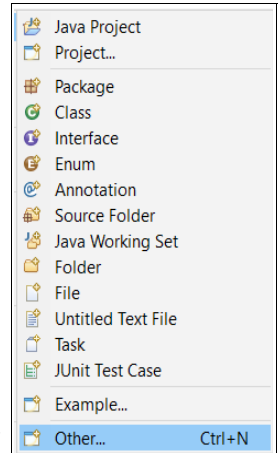
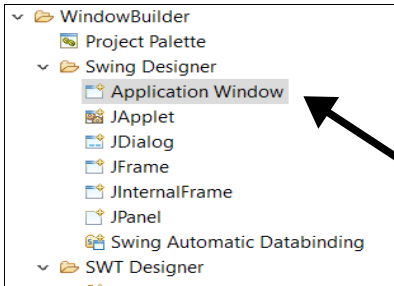




UNIDAD 8. Aplicaciones Controladas por Eventos.

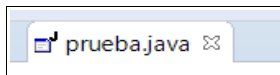
Tendrás que reiniciar eclipse y aceptar licencias de uso. Una vez instalado el plug-in tendremos varias utilidades nuevas: swing designer, SWT designer...

Ahora para crear un nuevo proyecto, debemos elegir Other, no Java Project. Y de las opciones que nos aparezcan, Window Builder -> Swing Designer -> Application Window.

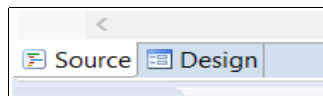


En el

fichero .java podremos ver un icono en la pestaña que nos indica que es window builder.



Y tenemos dos lugares (vistas) donde trabajar: source para meditar el código fuente y Design para usar el diseñador interactivo de la interfaz. Ambas aparecen en el panel donde antes solo teníamos la consola, errores, etc.

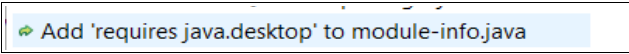


Otra cosa importante es que en el fichero fuente nos aparecerán un montón de errores porque necesitamos modificar el fichero module-



UNIDAD 8. Aplicaciones Controladas por Eventos.

info del proyecto para indicar que necesita java.desktop (interfaz gráfica). Puedes pulsar sobre el primer error y seleccionar la solución:



O bien modificarlo a mano:

```
module Cuestionario4 {  
    requires java.desktop;  
}
```

Es fácil de usar y ahorra mucho trabajo, veremos una demo en clase y con el uso lo dominarás rápidamente. A continuación te indico los elementos que te ofrece el diseñador.

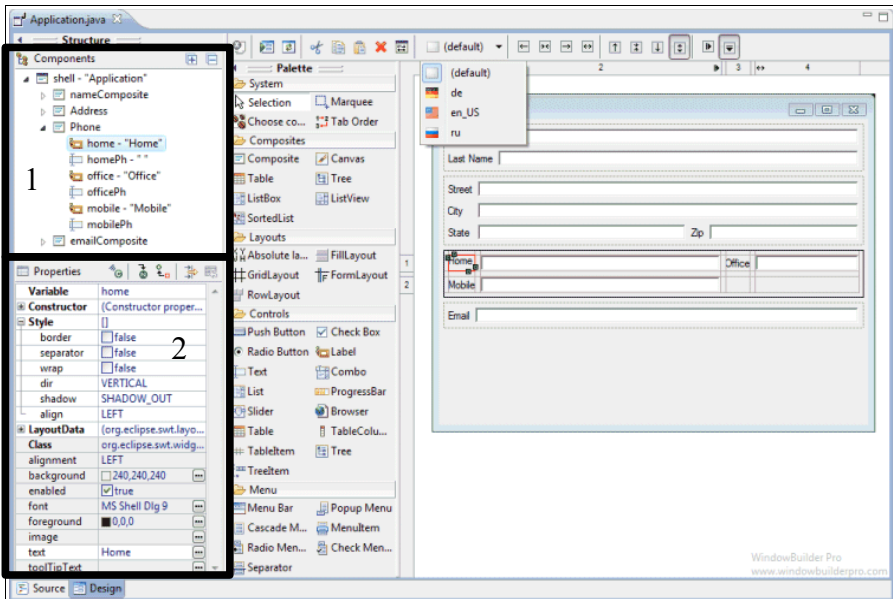
INTERFAZ DE USUARIO

El editor de GUI tiene los siguientes elementos principales:

- **Vista de Diseño:** - el área de dibujo interactivo.
- **Vista de Fuente:** - código Java.
- **Vista de Estructura:** - formada por **Component Tree** y el **Property Pane**.
 - **Component Tree**- muestra relación jerárquica entre componentes para saber dentro de qué está uno. (1)
 - **Property Pane**- muestra las propiedades y eventos del componente(s) seleccionado(s). (2)
- **Paleta**- acceso rápido a los componentes disponibles.
- **Toolbar**- acceso a acciones más comunes.
- **Menú**- acceso a acciones.



UNIDAD 8. Aplicaciones Controladas por Eventos.

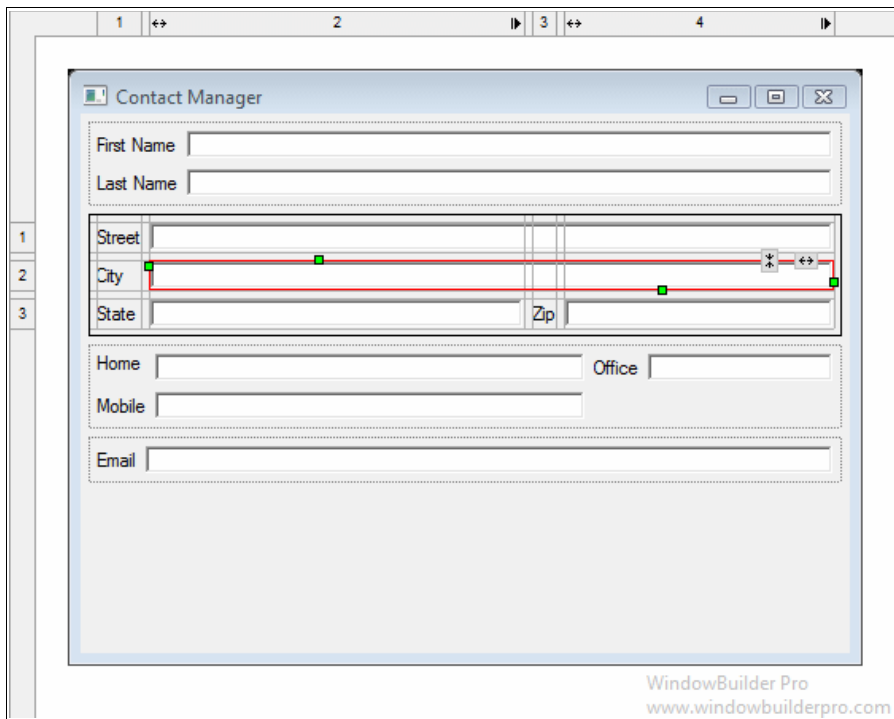


VISTA DE DISEÑO

Añades, cambias tamaño, mueves y eliminas componentes de la GUI que estás diseñando.




UNIDAD 8. Aplicaciones Controladas por Eventos.



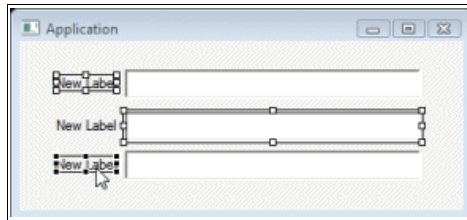
Cuando seleccionas un componente, aparece su panel de propiedades que puedes cambiar. También se dibuja un recuadro con puntos sensibles que te permiten cambiar propiedades de aspecto según el layout manager que lo controle. Si pulsas el botón derecho, accedes al menú contextual que también hay en el **Component Tree**.


Los componentes se pueden añadir seleccionándolos de la paleta y se pueden borrar usando la teclas de borrado.

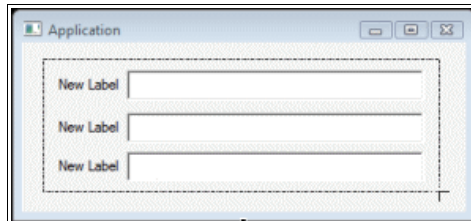
Para seleccionar componentes, marcas la herramienta de selección en la paleta (). Con las teclas **Mayúscula** o **Ctrl** puedes seleccionar más de uno.



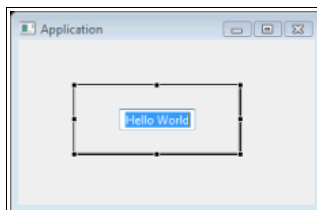
UNIDAD 8. Aplicaciones Controladas por Eventos.



Para seleccionar componentes dentro de un área usas la herramienta marcar () de la paleta. Si pulsas la tecla **Alt** y arrastras el ratón, se activa directamente.



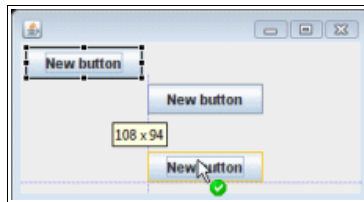
Editar directamente el texto de componentes como botones, labels, campos de texto, grupos, check box, etc. puedes pulsar el espacio tras seleccionarlos y se activa un editor.



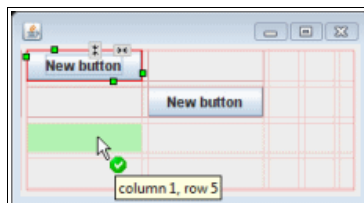
Para mover componentes desde la vista de diseño, el movimiento lo puede realizar el layout manager. Si usas un layout absoluto orientado a posiciones x,y puedes arrastrar el componente seleccionado a la nueva posición. La herramienta te muestra puntos de alineación con otros componentes cercanos.



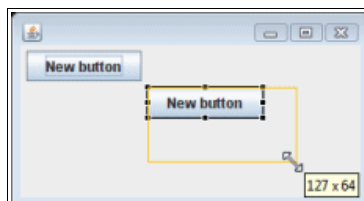
UNIDAD 8. Aplicaciones Controladas por Eventos.



En layouts de tipo rejilla (grid) te va mostrando la celda que va ocupando. Si la celda está ocupada, aparece un borde rojo, si la celda está vacía el borde es verde y si el cursor está sobre un borde de celda/columna se ilumina de amarillo.



El cambio de tamaño, depende del layout usado. En uno orientado a coordenadas absolutas, un tooltip muestra el tamaño actual, aparecen los puntos de alineamiento y puede afectar a otros componentes.



VISTA DE FUENTE

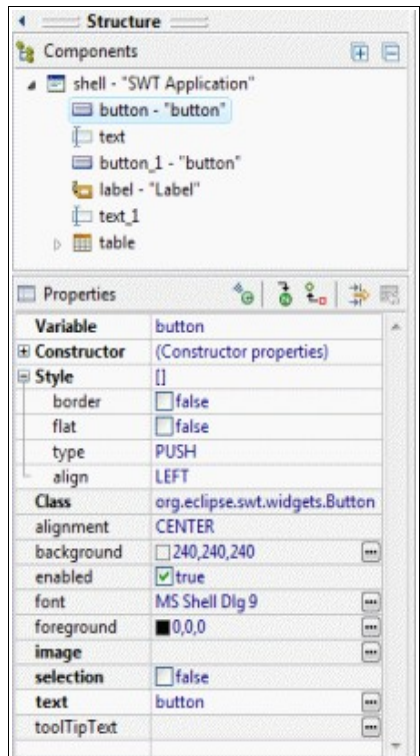
Es donde escribes el código Java y donde puedes modificar el código generado por la herramienta. Es simplemente un editor de código Java.



UNIDAD 8. Aplicaciones Controladas por Eventos.

VISTA DE ESTRUCTURA

Está formada por el **Component Tree** y por el **Property Pane**. El primero muestra las relaciones jerárquicas del componente actual en la vista de diseño. El segundo (panel de propiedades) muestra las propiedades y eventos de cada componente y ofrece editores de texto, listas y otros controles para permitirte cambiar las propiedades de cada control. La posición de ambos elementos puede cambiarse.



PALETA

Permite elegir rápidamente juegos de componentes estándar y creados por el usuario. Organiza los componentes en categorías que pueden ser expandidas, colapsadas y ocultas. Para añadir un componente a la vista de diseño puedes:

- Seleccionarlo de la paleta y arrastarlo a la vista de diseño o al árbol de componentes.
- Usar la acción **Choose Component** desde un diálogo.

Si pulsas la tecla **Ctrl** puedes añadir varios del mismo tipo. Algunas de las categorías de la paleta son:

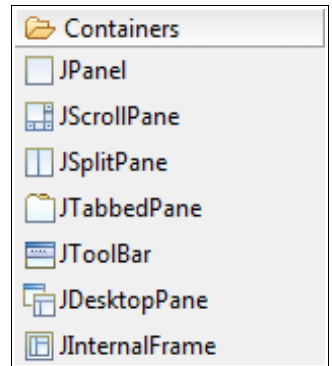


UNIDAD 8. Aplicaciones Controladas por Eventos.

CONTENEDORES

Ofrece contenedores para crear aplicaciones Swing. Los componentes son:

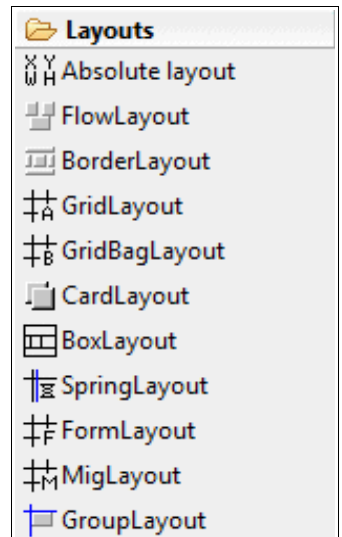
- **JPanel**- un contenedor genérico básico.
- **JScrollPane**- Una vista desplazable del contenedor básico. Gestiona un viewport, barras de desplazamiento opcionales horizontales y verticales y cabeceras de fila y columna de viewports opcionales.
- **JSplitPane**- divide dos componentes que pueden ser redimensionados por el usuario.
- **JTabbedPane**- permite al usuario intercambiar componentes.
- **JToolBar**- muestra acciones o controles.
- **JDesktopPane**- aplicación MDI (multiple-document interface).
- **JInternalFrame**- un objeto ligero similar a un frame.



LAYOUTS

Tiene diferentes layout managers.

- **Absolute Layout**- límites basados en coordenadas.
- **FlowLayout**- organiza componentes como un flujo de derecha a izquierda o al contrario.
- **BorderLayout**- organiza componentes en zonas central, norte, sur, este y oeste.



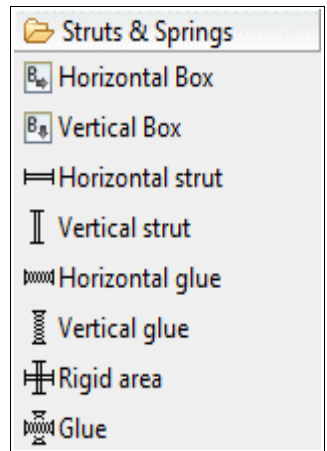


UNIDAD 8. Aplicaciones Controladas por Eventos.

- **GridLayout**- organiza en las celdas de una rejilla rectangular de igual tamaño.
- **GridBagLayout**- como el grid layout pero las celdas pueden ser de diferente tamaño.
- **CardLayout**- Solo una de las cartas del layout es visible en cierto momento.
- **BoxLayout**- permite que los componentes se desalíneen horizontal o verticalmente. Así no pierden el orden vertical/horizontal si hay cambios de tamaño.
- **FormLayout**- JGoodies FormLayout es un layout potente, cada componente puede ocupar una o más celdas de un grid.
- **SpringLayout**- organiza según ciertas restricciones de cada componente.
- **GroupLayout**- mezcla de grid layout y free form layout.

STRUTS Y SPRINGS

- **Horizontal Box**- un panel con un [BoxLayout](#) horizontal.
- **Vertical Box**- un panel que tiene un [BoxLayout](#) vertical.
- **Horizontal strut**- componente invisible de anchura fija.
- **Vertical strut**- componente invisible de altura fija.
- **Horizontal glue**- componente pegamento horizontal.
- **Vertical glue**- componente pegamento vertical.
- **Rigid area**- componente invisible del mismo tamaño.
- **Glue**- componente invisible de tipo pegamento.



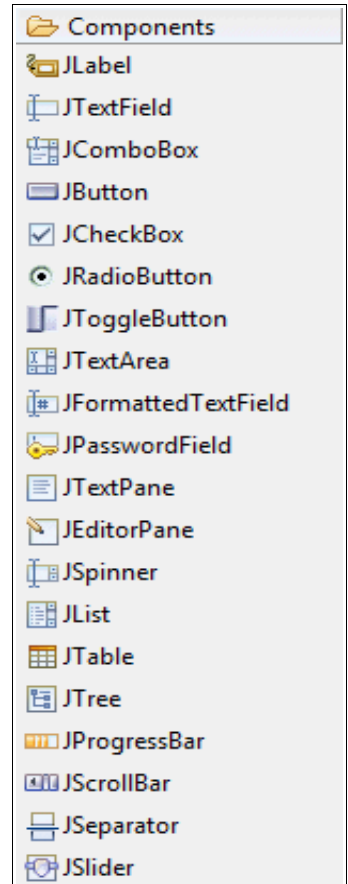


UNIDAD 8. Aplicaciones Controladas por Eventos.

COMPONENTES

Son elementos visuales de la interfaz con los que el usuario va a interactuar.

- **JLabel**- área para mostrar un texto, una imagen o ambas cosas.
- **JTextField**- permite modificar un texto de una sola línea.
- **JComboBox**- combina un campo modificable (si quieres) y una lista desplegable de la que puede elegir un elemento.
- **JButton**- un botón pulsable.
- **JCheckBox**- una casilla para marcar. Pueden marcarse varias en un mismo grupo.
- **JRadioButton**- un círculo marcapunto, en un grupo solo puede haber uno marcado.
- **JToggleButton**- botón de dos estados.
- **JTextArea**- permite editar textos de varias líneas.
- **JFormattedTextField**- permite editar textos que coincidan con una máscara predefinida.
- **JPasswordField**- permite modificar textos que se ocultan a la vista del usuario.
- **JTextPane**- permite representar gráficamente texto y gráficos.
- **JEditorPane**- edita varios tipos de texto.





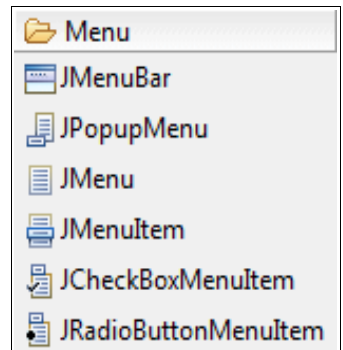
UNIDAD 8. Aplicaciones Controladas por Eventos.

- **JSpinner**- permite escoger un valor de un conjunto ordenado de ellos.
- **JList**- muestra una lista de la que escoger uno o varios elementos.
- **JTable**- muestra y edita texto en forma tabular, en celdas.
- **JTable sobre JScrollPane**- pues eso.
- **JTree**- datos jerárquicos.
- **JProgressBar**- muestra un valor dentro de un intervalo.
- **JScrollBar**- permite desplazar l+un indicador.
- **JSeparator**- una línea separadora vertical u horizontal.
- **JSlider**- selecciona un valor usando una especie de barra de desplazamiento.

MENÚ

Contiene los elementos para definir menús de opciones.

- **JMenuBar**- Una barra de menú.
- **JPopupMenu**- un menú desplegable.
- **JMenu**- otro menú desplegable a partir de otro o una barra de menú.
- **JMenuItem**- un único elemento de menú.
- **JCheckBoxMenuItem**- un elemento de menú de tipo checkbox.
- **JRadioButtonMenuItem**- un elemento de menú de tipo radio.



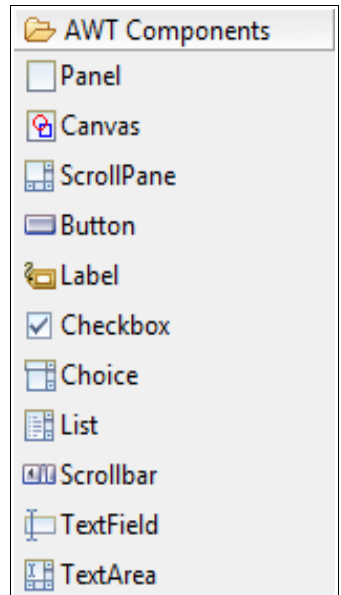
COMPONENTES AWT

- **Panel**- el contenedor más simple.
- **Canvas**- un área rectangular para dibujar y atrapar eventos de usuario.



UNIDAD 8. Aplicaciones Controladas por Eventos.

- **ScrollPane**- contenedor con desplazamiento horizontal y vertical automático.
- **Button**- botón etiquetado.
- **Label**- muestra un texto.
- **Checkbox**- una caja para marcar.
- **Choice**- menú de opciones, la opción elegida se muestra.
- **List**- Lista de líneas de texto desplazables de donde elegir una o varias.
- **Scrollbar**- permite escoger valores desplazando el indicador.
- **TextField**- un campo de texto de una sola línea.
- **TextArea**- un campo de texto para modificar textos de varias líneas.



Por lo demás, todo lo visto con anterioridad en el tema sigue siendo válido. Cada componente tendrá una propiedad **name**, que será la variable que debes usar en el código Java para interactuar con el objeto desde el programa.

La ayuda de la herramienta tiene tutoriales. Otros IDEs, como netbeans ya tienen su propio diseñador de interfaces integrado, es similar. Una vez que usas uno, los conoces todos.



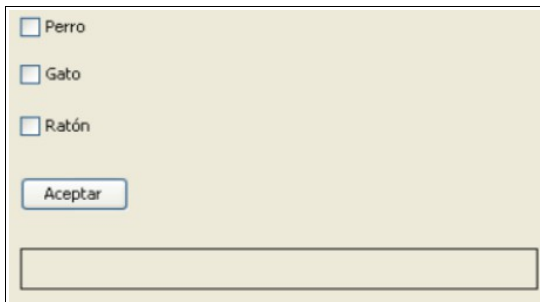
UNIDAD 8. Aplicaciones Controladas por Eventos.

8.7. EJERCICIOS.

EJERCICIO 1. Crea el proyecto GUI1 y en la ventana principal debes añadir lo siguiente:

- Un botón "Aceptar" llamado bAceptar.
- Una etiqueta con borde llamada lResultado.
- Añade también 3 cajas de verificación (JCheckBox) y cambia su texto de forma que aparezca "Perro", "Gato" y "Ratón". Cambia también su nombre. Se llamarán: cbPerro, cbGato, cbRaton.

La ventana tendrá el siguiente aspecto cuando termines:



El programa debe funcionar de la siguiente forma: cuando el usuario pulse el botón Aceptar, en la etiqueta con recuadro aparecerá un mensaje indicando qué animales han sido "seleccionados". Para ello hay que programar el evento actionPerformed del botón Aceptar. En su manejador de eventos añadirás el código necesario.

EJERCICIO 2. Crea el proyecto GUI2 y diseña la ventana principal donde debes añadir lo siguiente:

- Un botón "Aceptar" llamado bAceptar.
- Una etiqueta con borde llamada lbResultado.
- Un panel (JPanel) al que le pones un borde de tipo TitledBorder



UNIDAD 8. Aplicaciones Controladas por Eventos.

(borde con título) y pon el título "colores".

- Dentro del panel, añade 3 botones de opción (botones de radio) Son objetos del tipo `JRadioButton`. Cambia su texto para que aparezca "Rojo", "Verde" y "Azul". Cambia su nombre por `rbRojo`, `rbVerde`, `rbAzul` respectivamente.

La ventana tendrá el siguiente aspecto cuando termines:



Si ejecutas el programa, observarás que pueden seleccionarse varios colores a la vez. Esto no es interesante, ya que los botones de radio se usan para activar solo una opción de entre varias. Añade un objeto **ButtonGroup** al formulario (este objeto es invisible, no se ve) aunque si se puede ver en el árbol de componentes.

- Da un nombre al `ButtonGroup` (`bgColores` por ejemplo).
- Debes conseguir que los 3 botones radio pertenezcan al mismo grupo. Selecciona el botón de opción `rbRojo` y cambia su propiedad `buttonGroup` seleccionando el grupo que acabas de crear.

Pruebe el programa de nuevo a ver si ahora solamente uno de ellos puede estar marcado.



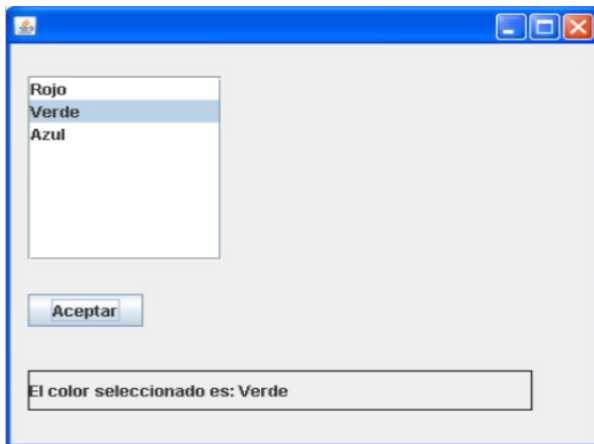
UNIDAD 8. Aplicaciones Controladas por Eventos.

Ahora interesa que la opción "Rojo" salga activada por defecto. Una forma de hacer esto es usar **setSelected(true)**; desde el "Constructor" de la ventana. Prueba el programa cuando lo hayas hecho.

Por último, cuando el usuario pulse el botón Aceptar, en la etiqueta aparece el nombre del color elegido y el texto de la etiqueta saldrá en ese color. Para ello, crea el código en el actionPerformed del botón.

EJERCICIO 3. Crea el proyecto GUI03 y en la ventana principal añade:

- Un botón "Aceptar" llamado bAceptar.
- Una etiqueta con borde llamada lbResultado.
- Añade un JScrollPane y dentro una lista (JList). Cambia el nombre al JList por lstColores. Los items del JList pueden cambiarse a través de su propiedad **Model**. Busca esta propiedad y haz clic en el botón de los tres puntos. Aparecerá un cuadro de diálogo donde puedes cambiarlos en tiempo de diseño. Crea los siguientes valores: Rojo, Verde y Azul.





UNIDAD 8. Aplicaciones Controladas por Eventos.

Ahora programa el `actionPerformed` del botón Aceptar para conseguir que en la etiqueta aparezca el color que hay seleccionado si es que hay alguno seleccionado, o "Color seleccionado: ninguno" si no se ha elegido ninguno. Ejecuta el programa y observa su funcionamiento.

EJERCICIO 4. Modifica el ejercicio 3 y copia el código del `actionPerformed` del botón aceptar en el evento `mouseClicked` de la lista. Elimina el botón y prueba el programa.

EJERCICIO 5. Realiza un nuevo proyecto *GUI05*, abre el diseñador y en la ventana principal debes añadir:

- Un combo (`JComboBox`) llamado `cboNumeros`.
- Un botón "Pares" llamado `bPares` y un botón "Impares" llamado `bImpares`.
- Una etiqueta con borde llamada `lbResultado`.

Usa la propiedad "model" del combo para cambiar sus elementos. Si tiene los borras. Después de haber hecho todo esto, tu ventana debe quedar más o menos así:



En el evento `actionPerformed` del botón Pares, programa lo siguiente:

```
int i;  
DefaultComboBoxModel modelo = new DefaultComboBoxModel();  
for (i=0; i <10; i+= 2) {  
    modelo.addElement("Nº " + i);  
}
```



UNIDAD 8. Aplicaciones Controladas por Eventos.

```
}  
cboNumeros.setModel(modelo);
```

Ejecuta el programa y observa el funcionamiento del botón Pares. Haz tu el de impares. Programa el actionPerformed del combo para que al seleccionar un elemento aparezca en la etiqueta.

```
lbResultado.setText(cboNumeros.getSelectedItem().toString());
```

Recuerda el uso de `getSelectedItem()` para recoger el elemento seleccionado y el uso de `toString()` para convertirlo a texto. Prueba el programa. Prueba los botones Pares e Impares y prueba el combo.

Añade un botón "Vaciar" llamado `bVaciar` que vacía el contenido del combo. Esto se haría simplemente creando un modelo vacío y asignarlo al combo o busca si el modelo/combo tiene algún método para vaciarlo.

EJERCICIO 6. Realiza un nuevo proyecto GUI06 y en la ventana principal debes añadir lo siguiente:

- Una etiqueta con borde llamada `lbResultado`.
- Un cuadro de lista (`JList`).
- Borra todo el contenido de la lista (propiedad `model`) y cámbia su nombre a `lstNombres`. Recuerda que las listas deben aparecer dentro de un objeto del tipo `JScrollPane`.
- Añade 2 botones al formulario. Uno de ellos tendrá el texto "Curso 1" y se llamará `bCurso1` y el otro tendrá el texto "Curso 2" y se llamará `bCurso2`.



UNIDAD 8. Aplicaciones Controladas por Eventos.



En el evento `actionPerformed` del botón "Curso 1" programa lo siguiente:

```
DefaultListModel modelo = new DefaultListModel();
modelo.addElement("Juan");
modelo.addElement("María");
modelo.addElement("Luis");
lstNombres.setModel(modelo);
```

En el evento `actionPerformed` del botón "Curso 2" programa lo siguiente:

```
DefaultListModel modelo = new DefaultListModel();
modelo.addElement("Ana");
modelo.addElement("Marta");
modelo.addElement("Jose");
lstNombres.setModel(modelo);
```

Cuando en la lista se seleccione un elemento, que aparezca en el label.

EJERCICIO 7. Crea el proyecto GUI07, en la ventana deja el layout por defecto del panel que tiene el `contentPane` y en el centro añade un componente (label) al que le das de color de fondo verde. En la zona sur del layout de la ventana crea otro label con un recuadro. Queremos que al entrar el ratón dentro de la superficie del componente del centro aparezca un mensaje en la otra etiqueta indicándolo. También queremos que aparezca un mensaje al salir el ratón y al pulsar un botón



UNIDAD 8. Aplicaciones Controladas por Eventos.

del ratón sobre la etiqueta que diga el botón y las coordenadas dentro del componente.

Para hacer esto, puedes crear un `MouseAdapter` que contenga los siguientes eventos: `mouseEntered`, `mouseExited` y `mousePressed`. Observa que no es necesario que contenga el `mouseReleased` ni el `mouseClicked`. Luego, el `MouseAdapter` se registrará en el panel.

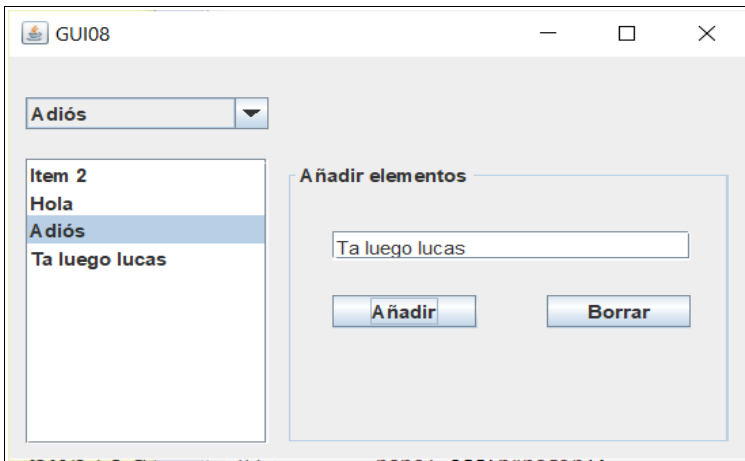


EJERCICIO 8. Crea la interfaz de la figura en la que puedes observar componentes `JComboBox`, `JList`. Probad a modificar estas propiedades y ver qué efectos tiene:

- En el `JComboBox`
 - **`maximunRowCount`**
- En el `JList`
 - **`selectionMode`**
 - **`LayoutOrientation`**



UNIDAD 8. Aplicaciones Controladas por Eventos.



EJERCICIO 9. Siguiendo con el proyecto del ejercicio 8, utiliza los métodos **getSelectedIndex()**, **removeItemAt(int index)** y **addItem(Object)** en los eventos **actionPerformed** de cada uno de los botones para que lo que escribas en la caja de edición puedas añadirlo a la lista y al ComboBox (Lo ideal es que compartas el modelo y uses **getSelectedIndex()**, **addElement()** y **removeElementAt(indice)** del propio modelo) aunque también puedes usar métodos de cada componente. Para sincronizar lo que hay seleccionado en el comobobox, Jlist y textfield debes ver los eventos que se producen en los componentes JList (**valueChanged**) y JComboBox (**itemStateChanged**) y el uso de los métodos **getSelectedIndex()** y **getSelectedValue()**.

EJERCICIO 10. Utiliza Window Builder para crear una ventana con un botón con el texto "Borrar" y un panel con fondo blanco, borde de tipo Titled con el título "Haz clic" y una etiqueta con borde BorderLine.

- Cuando hagas clic en el panel debe dibujarse un círculo relleno de color azul de radio 20 con centro en el punto clicado, e



UNIDAD 8. Aplicaciones Controladas por Eventos.

imprimir las coordenadas donde se ha realizado el clic en el label, en varias líneas y el punto clicado en cursiva (los componentes de swing entienden HTML puedes pasar "<html> </html>" en el texto).

- Cuando pulses el botón, borra el dibujo del panel y lo rellenas de blanco.
- Consigue que no se borre el dibujo cuando maximizas/minimizas, mueves o solapas la ventana.

