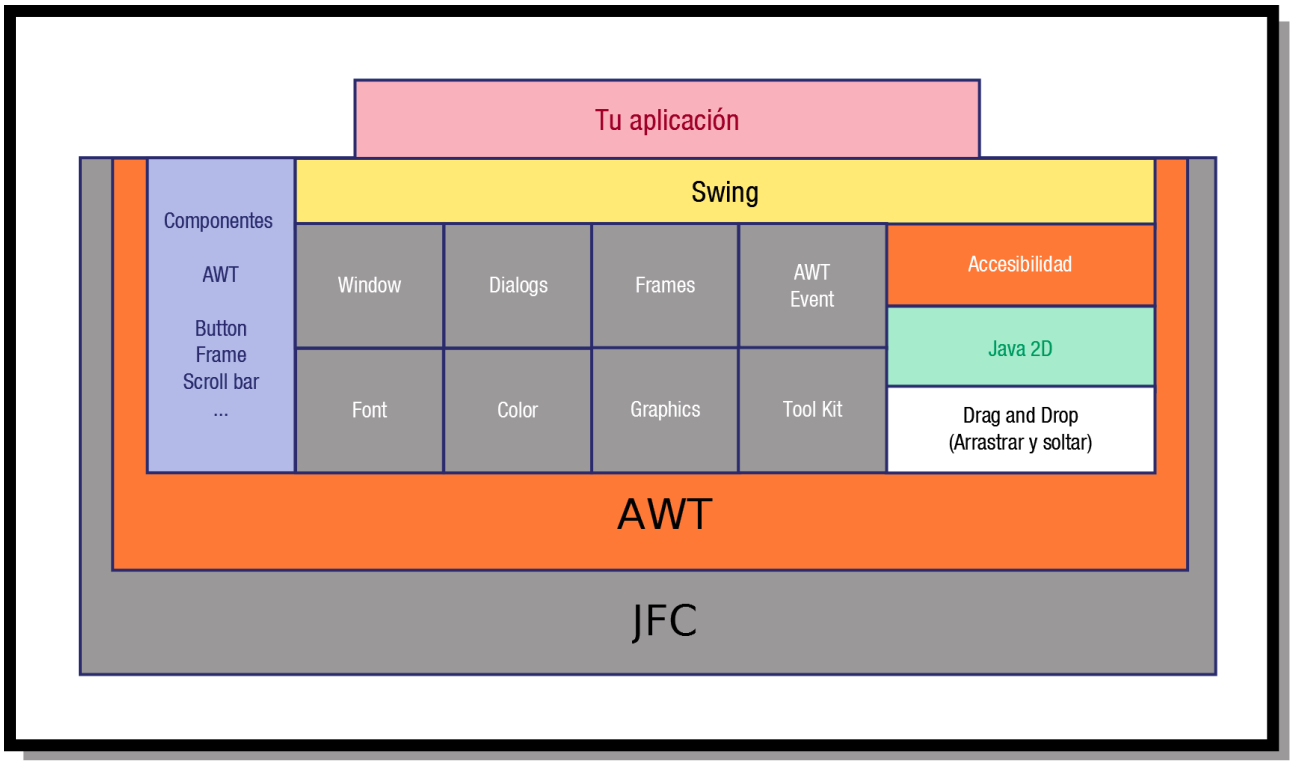
¡¡NOTA ACLARATORIA!!

Estos ficheros son un resumen literal del contenido de las unidades de trabajo expuestas en la plataforma FEDAN de educación semipresencial de la Junta de Andalucía. El motivo de sintetizarlos y subirlos a mi perfil de GitHub es conservarlos para que en un futuro pueda disponer de este contenido y consultarlo con fines didácticos.

TEMA 9

INTERFACES GRÁFICAS DE USUARIO (GUI)

En ocasiones verás otras definiciones de interfaz, como la que define una interfaz como un dispositivo que permite comunicar dos sistemas que no hablan el mismo lenguaje. También se emplea el término interfaz para definir el juego de conexiones y dispositivos que hacen posible la comunicación entre dos sistemas.



LIBRERÍAS JAVA PARA EL DESARROLLO DE INTERFACES GRÁFICAS DE USUARIO

El API de Java proporciona librerías de clases para el desarrollo de interfaces gráficas de usuario. Esas librerías se engloban bajo los nombres de **AWT** y **Swing**, que a su vez forman parte de las **Java Foundation Classes o JFC.**

Los elementos que componen las JFC son:

- **Componentes Swing**: encontramos componentes tales como botones, cuadros de texto, ventanas o elementos de menú.
- **Soporte de diferentes aspectos y comportamientos** (**Look and Feel**): permite la elección de diferentes apariencias de entorno. Por ejemplo, el mismo programa puede adquirir un aspecto **Metal** Java (multiplataforma, igual en cualquier entorno), **Motif** (el aspecto estándar para entornos Unix) o **Windows** (para entornos Windows). De esta forma, el aspecto de la aplicación puede ser independiente de la plataforma, lo cual está muy bien para un lenguaje

que lleva la seña de identidad de multiplataforma, y se puede elegir el que se desee en cada momento.

- **Interfaz de programación Java** : permite incorporar gráficos en dos dimensiones, texto e imágenes de alta calidad.
- **Soporte de arrastrar y soltar** (Drag and Drop) entre aplicaciones Java y aplicaciones nativas. Es decir, se implementa un portapapeles. Llamamos aplicaciones nativas a las que están desarrolladas en un entorno y una plataforma concretos (por ejemplo Windows o Unix), y sólo funcionarán en esa plataforma.
- **Soporte de impresión.**
- **Soporte sonido:** captura, reproducción y procesamiento de datos de audio y MIDI
- **Soporte de dispositivos de entrada distintos del teclado**, para japonés, chino, etc.
- **Soporte de funciones de Accesibilidad, para crear interfaces para discapacitados:** permite el uso de tecnologías como los lectores de pantallas o las pantallas Braille adaptadas a las personas discapacitadas.

AWT(Abstract Window Toolkit)

Las clases AWT se desarrollaron usando código nativo (o sea, código asociado a una plataforma concreta), así que para poder conservar la portabilidad era necesario restringir la funcionalidad a los mínimos comunes a todas las plataformas donde se pretendía usar AWT. Como consecuencia, AWT es una librería con una funcionalidad muy pobre, **adecuada para interfaces gráficas sencillas**, pero no para proyectos más complejos.

La estructura básica de la librería gira en torno a **componentes** y **contenedores**. Los contenedores contienen componentes y son componentes a su vez, de forma que los eventos pueden tratarse tanto en contenedores como en componentes.

Más tarde, con **Java 2** surgió una librería más robusta, versátil y flexible: **Swing**. Pero AWT sigue siendo imprescindible, ya que todos **los componentes Swing se construyen haciendo uso de clases de AWT**. De hecho, todos los componentes Swing, como por ejemplo `JButton`, derivan de la clase `JComponent`, que a su vez deriva de la clase AWT `Container`.

Las clases asociadas a cada uno de los **componentes** AWT se encuentran en el paquete `java.awt`. Las clases relacionadas con el manejo de **eventos** en AWT están en el paquete `java.awt.event`.

AWT fue la primera forma de construir las ventanas en Java, pero:

- Limitaba la portabilidad.
- Restringía la funcionalidad.
- Requería demasiados recursos.

SWING

Swing es una librería de Java para la generación del GUI en aplicaciones.

Swing se apoya sobre AWT y añade `JComponents`. La arquitectura de los componentes de Swing facilita la personalización de apariencia y comportamiento, si lo comparamos con los componentes AWT.

Por cada componente AWT (excepto `Canvas`) existe un componente Swing equivalente, **cuyo nombre empieza por J**, que permite más funcionalidad siendo menos pesado. Así, por ejemplo, para el componente AWT `Button` existe el equivalente Swing `JButton`, que permite como funcionalidad adicional la de crear botones con distintas formas (rectangulares, circulares, etc), incluir imágenes en el botón, tener distintas representaciones para un mismo botón según esté seleccionado, o bajo el cursor, etc.

La razón por la que no existe `JCanvas` es que los paneles de la clase `JPanel` ya soportan todo lo que el componente `Canvas` de AWT soportaba. No se consideró necesario añadir un componente Swing `JCanvas` por separado.

Algunas otras características más de Swing son:

- Es **independiente de la arquitectura** (metodología no nativa propia de Java).
- Proporciona **todo lo necesario para la creación de entornos gráficos**, tales como diseño de menús, botones, cuadros de texto, manipulación de eventos, etc**.**
- **Los componentes Swing no necesitan una ventana propia del sistema operativo cada uno**, sino que son visualizados dentro de la ventana que los contiene mediante métodos gráficos, por lo que requieren bastantes menos recursos.
- **Las clases Swing están completamente escritas en Java**, **con lo que la portabilidad es total**, a la vez que no hay obligación de restringir la funcionalidad a los mínimos comunes de todas las plataformas.
- Las clases Swing aportan una considerable gama de funciones que haciendo uso de la funcionalidad básica propia de AWT **aumentan las posibilidades de diseño** de interfaces gráficas.
- Debido a sus características, los componentes **AWT** se llaman componentes **“de peso pesado”** por la gran cantidad de recursos del sistema que usan, y los componentes **Swing** se llaman componentes **“de peso ligero”** por no necesitar su propia ventana del sistema operativo y por tanto consumir muchos menos recursos.
- Aunque todos los componentes Swing derivan de componentes AWT y de hecho **se pueden mezclar en una misma aplicación componentes de ambos tipos**, se desaconseja hacerlo. **Es preferible desarrollar aplicaciones enteramente Swing**, que requieren menos recursos y son más portables.

Los pasos para crear y ejecutar aplicaciones, se pueden resumir en:

- Crear un proyecto.
- Construir la interfaz con el usuario.
- Añadir funcionalidad, en base a la lógica de negocio que se requiera.
- Ejecutar el programa.

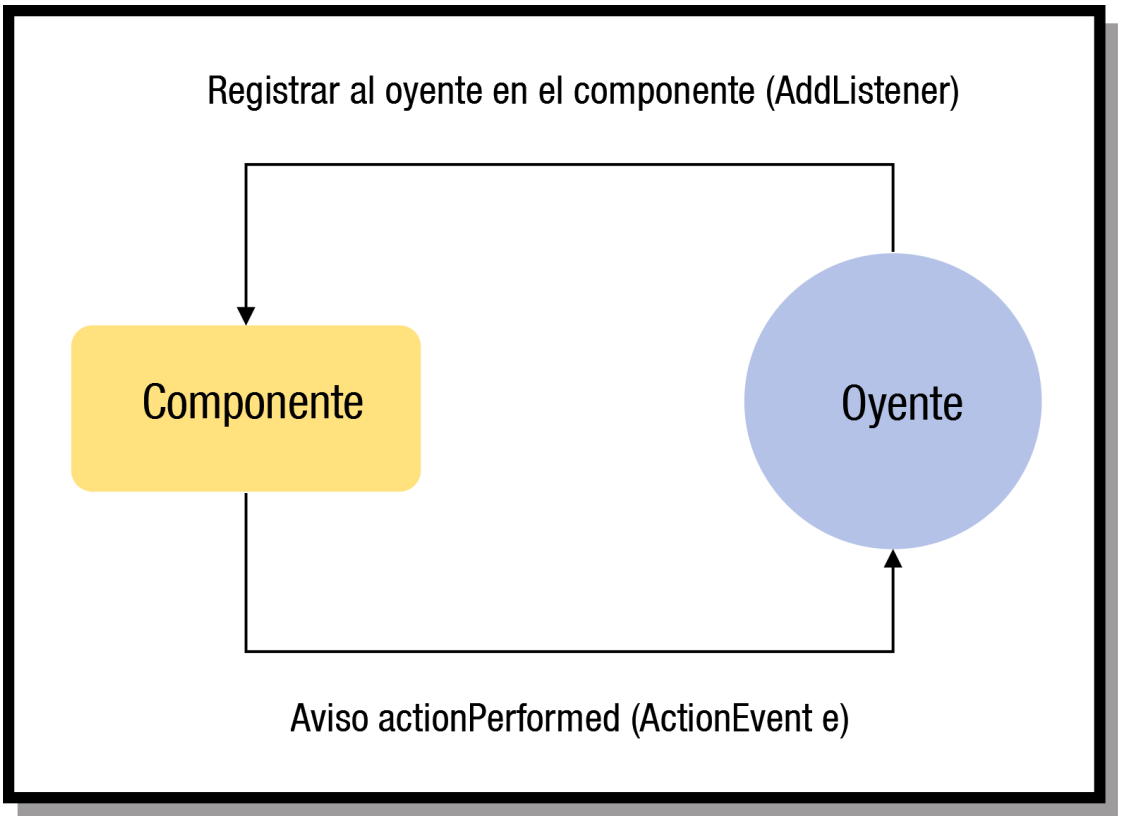
EVENTOS

Es todo hecho que ocurre mientras se ejecuta la aplicación. Normalmente, llamamos evento a cualquier interacción que realiza el usuario con la aplicación, como puede ser:

- pulsar un botón con el ratón;
- hacer doble clic;
- pulsar y arrastrar;
- etc...

Para abordar el problema de tratar correctamente las interacciones del usuario con la interfaz gráfica de la aplicación recurrimos a la **programación guiada por eventos.**

MODELO DE GESTIÓN DE EVENTOS



Cada vez que el usuario realiza una determinada acción sobre una aplicación que estamos programando en Java (un clic sobre el ratón, presionar una tecla, etc.), se produce un evento que el sistema operativo transmite a Java.

Java crea un objeto de una determinada clase de evento, y este evento se transmite a un determinado método para que lo gestione.

El **modelo de eventos de Java está basado en delegación**, es decir, la responsabilidad de gestionar un evento que ocurre en un objeto fuente la tiene otro objeto **oyente**.

Las **fuentes de eventos** (event sources) son objetos que detectan eventos y notifican a los receptores que se han producido dichos eventos. Ejemplos de fuentes:

- Botón sobre el que se pulsa o pincha con el ratón.
- Campo de texto que pierde el foco.
- Campo de texto sobre el que se presiona una tecla.
- Ventana que se cierra.
- Etc.

En el apartado anterior de creación de interfaces con ayuda de los asistentes del IDE, vimos lo fácil que es realizar este tipo de programación, ya que el IDE hace muchas cosas, genera código automáticamente por nosotros.

Pero también podríamos hacerlo nosotros todo, si no tuviéramos un IDE como NetBeans, o porque simplemente nos apeteciera hacerlo todo desde código, sin usar asistentes ni diseñadores gráficos. En este caso, **los pasos a seguir** se pueden resumir en:

1. Crear la clase oyente que implemente la interfaz.
    - Ejemplo: `ActionListener`: pulsar un botón.
2. Implementar en la clase oyente los métodos de la interfaz.
    - Ejemplo: `void actionPerformed(ActionEvent)`.
3. Crear un objeto de la clase oyente y registrarlo como oyente en uno o más componentes gráficos que proporcionen interacción con el usuario.

Antes de ver el siguiente ejemplo he de hacer un inciso. Con el código que se muestra en el flash player del tema 9 apartado 4.2 Modelo de Gestión de Eventos el programa no funciona. No sé por qué, pero se arregla declarando el atributo "`etiqueta`" de tipo **JPanel** como `static` en la clase **Ventanilla** y añadiendo en la clase **OyenteAccion** la sentencia:

Ventanilla.`etiqueta`.`setText`("Botón pulsado: " + `boton`.`getText`());

Veamos el ejemplo que a mi no me funciona:

Creando el programa(I)

`Importamos las clases necesarias para implementar algunos elementos de las interfaces gráficas`

```
1  import javax.swing.JButton;
2  import javax.swing.JFrame;
3  import javax.swing.JLabel;
4  import javax.swing.JPanel;
5
6  public class Ventanilla extends JFrame {
7
8      //Declaramos los atributos
9      JLabel etiqueta;
10     JButton botonUno, botonDos;
11     JPanel panel;
12
```

Definimos los elementos que llevará nuestro diseño

```
13      /*
14       * Creamos los componentes:
15       * una etiqueta, dos botones
16       * y un panel
17       */
18      public Ventanilla() {
19
20          etiqueta = new JLabel("Mi etiqueta");
21          botonUno = new JButton("Botón 1");
22          botonDos = new JButton("Botón 2");
23          panel = new JPanel();
24
25
26          //Añadimos los componentes al panel
27          panel.add(etiqueta);
28          panel.add(botonUno);
29          panel.add(botonDos);
30
31          //Añadimos el panel al Frame
32          getContentPane().add(panel);
33
34          /*
35           * Creamos el objeto de la clase oyente
36           * para cuando se pulse el botón
37           */
38          OyenteAccion oyenteBoton = new OyenteAccion();
39
40          /*
41           * Registrar el objeto como oyente
42           * en los dos botones
43           */
44          botonUno.addActionListener(oyenteBoton);
45          botonDos.addActionListener(oyenteBoton);
46      }
47
48  }
```

Ahora creamos la clase oyente

```
1  import java.awt.event.ActionEvent;
7
8  //Implementación de la clase oyente
9  class OyenteAccion implements ActionListener{
10
11     //Cuando se pinche el botón
12     public void actionPerformed(ActionEvent evento) {
13
14         //Obtener el botón que disparó el evento
15         JButton boton = (JButton) evento.getSource();
16
17         //Escribir en la etiqueta el botón pulsado
18         etiqueta.setText("Botón pulsado " + boton.getText());
19
20     }
```

Y a continuación el main

```
22     //Programa principal
23     public static void main(String args[]) {
24
25         //Creamos la ventana
26         Ventanilla ventana = new Ventanilla();
27
28         //Establecer título, tamaño y visibilidad
29         ventana.setTitle("Nueva Ventana");
30         ventana.setVisible(true);
31         ventana.setSize(500, 100);
32
33
34
35     }
36
37 }
```

De esta manera se muestra el código en el temario pero, como digo, en mi eclipse Oxygen no me funciona. Ciertamente, se está haciendo una llamada a un atributo de otra clase sin más que el nombre de dicho atributo, ni siquiera se declara y, como se puede comprobar, no existen métodos get con los que acceder a los atributos de la otra clase.

El código modificado, apenas sufre modificaciones salvo las ya reseñadas y quedaría como sigue:

Clase Ventanilla (I)

```
1  import javax.swing.JButton;
2  import javax.swing.JFrame;
3  import javax.swing.JLabel;
4  import javax.swing.JPanel;
5
6  public class Ventanilla extends JFrame {
7
8      //Declaramos los atributos
9      static JLabel etiqueta;
10     JButton botonUno, botonDos;
11     JPanel panel;
12
13     /*
14      * Creamos los componentes:
15      * una etiqueta, dos botones
16      * y un panel
17      */
18     public Ventanilla() {
19
20         etiqueta = new JLabel("Mi etiqueta");
21         botonUno = new JButton("Botón 1");
22         botonDos = new JButton("Botón 2");
23         panel = new JPanel();
24
25
```

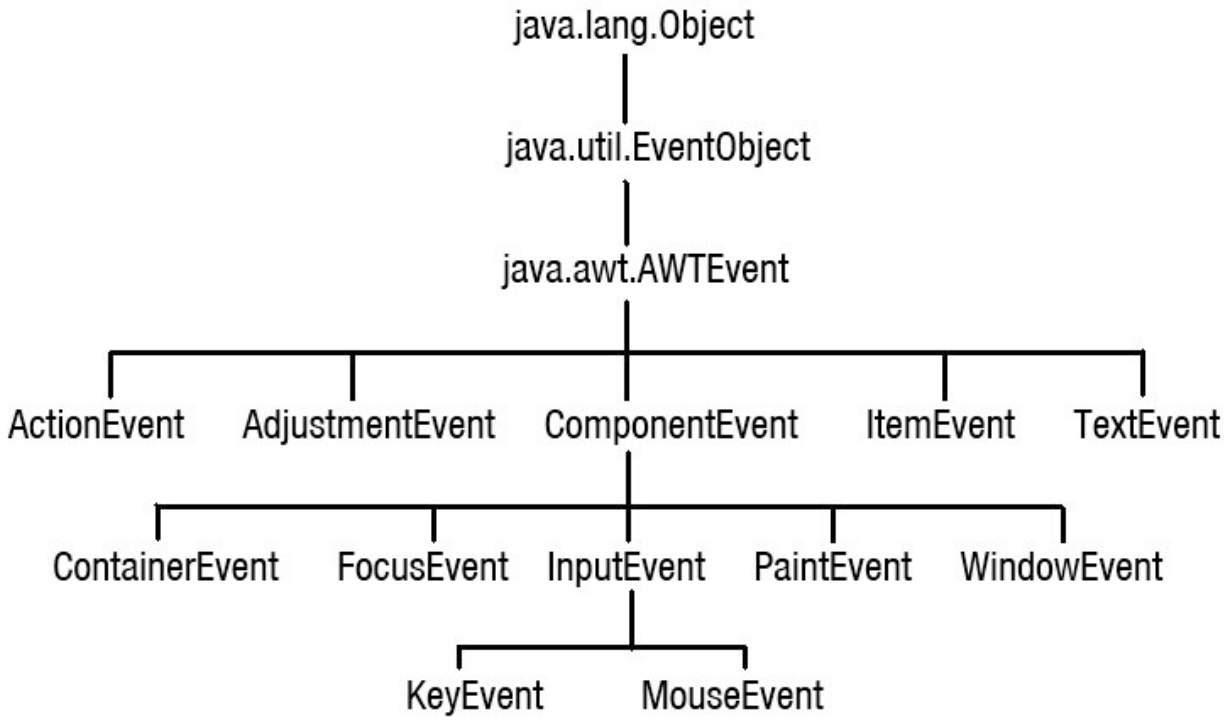
Clase Ventanilla (II)

```
26         //Añadimos los componentes al panel
27         panel.add(etiqueta);
28         panel.add(botonUno);
29         panel.add(botonDos);
30
31         //Añadimos el panel al Frame
32         getContentPane().add(panel);
33
34         /*
35          * Creamos el objeto de la clase oyente
36          * para cuando se pulse el botón
37          */
38         OyenteAccion oyenteBoton = new OyenteAccion();
39
40         /*
41          * Registrar el objeto como oyente
42          * en los dos botones
43          */
44         botonUno.addActionListener(oyenteBoton);
45         botonDos.addActionListener(oyenteBoton);
46     }
47
48 }
```

Clase OyenteAccion y main

```
1  import java.awt.event.ActionEvent;
2  import java.awt.event.ActionListener;
3
4  import javax.swing.AbstractButton;
5  import javax.swing.JButton;
6  import javax.swing.JLabel;
7
8  //Implementación de la clase oyente
9  class OyenteAccion implements ActionListener{
10
11     //Cuando se pinche el botón
12     public void actionPerformed(ActionEvent evento) {
13
14         //Obtener el botón que disparó el evento
15         JButton boton = (JButton) evento.getSource();
16
17         //Escribir en la etiqueta el botón pulsado
18         Ventanilla.etiqueta.setText("Botón pulsado: " + boton.getText());
19
20     }
21
22     //Programa principal
23     public static void main(String args[]) {
24
25         //Creamos la ventana
26         Ventanilla ventana = new Ventanilla();
27
28         //Establecer título, tamaño y visibilidad
29         ventana.setTitle("Nueva Ventana");
30         ventana.setVisible(true);
31         ventana.setSize(500, 100);
32
33
34
35     }
36
37 }
38
```

Tipos de eventos.



En la mayor parte de la literatura escrita sobre Java, encontrarás dos tipos básicos de eventos:

- **Físicos** o de **bajo nivel**: que corresponden a un evento hardware claramente identificable. Por ejemplo, se pulsó una tecla (`KeyStrokeEvent`). Destacar los siguientes:
  - En componentes: `ComponentEvent`. Indica que un componente se ha movido, cambiado de tamaño o de visibilidad
  - En contenedores: `ContainerEvent`. Indica que el contenido de un contenedor ha cambiado porque se añadió o eliminó un componente.
  - En ventanas: `WindowEvent`. Indica que una ventana ha cambiado su estado.
  - `FocusEvent`, indica que un componente ha obtenido o perdido la entrada del foco.
- **Semánticos** o de mayor nivel de abstracción: se componen de un conjunto de eventos físicos, que se suceden en un determinado orden y tienen un significado más abstracto. Por ejemplo: el usuario elige un elemento de una lista desplegable (`ItemEvent`).
  - `ActionEvent, ItemEvent, TextEvent, AdjustmentEvent`.

Los eventos en Java se organizan en una jerarquía de clases:

- La clase **`java.util.EventObject`** es la clase base de todos los eventos en Java.
- La clase **`java.awt.AWTEvent`** es la clase base de todos los eventos que se utilizan en la construcción de GUI.
- Cada tipo de evento *`loqueseaEvent`* tiene asociada una interfaz *`loqueseaListener`* que nos permite definir manejadores de eventos.
- Con la idea de simplificar la implementación de algunos manejadores de eventos, el paquete **`java.awt.event`** incluye clases *`loqueseaAdapter`* que implementan las interfaces *`loqueseaListener`*.

EVENTOS DE TECLADO

`KeyListener` (oyente de teclas).

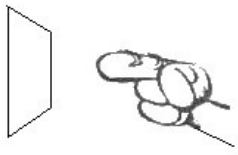
Método	Causa de la invocación
`keyPressed (KeyEvent e)`	Se ha pulsado una tecla.
`keyReleased (KeyEvent e)`	Se ha liberado una tecla.
`keyTyped (KeyEvent e)`	Se ha pulsado (y a veces soltado) una tecla.

`KeyEvent` (evento de teclas)

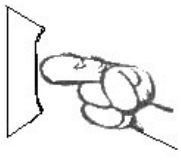
Métodos más usuales	Explicación
`char getKeyChar()`	Devuelve el carácter asociado con la tecla pulsada.
`int getKeyCode()`	Devuelve el valor entero que representa la tecla pulsada.
`String getKeyText()`	Devuelve un texto que representa el código de la tecla.
`Object getSource()`	Método perteneciente a la clase **`EventObject`**. Indica el objeto que produjo el evento.

La clase `KeyEvent`, define muchas constantes así:

- `KeyEventVK_A` especifica la tecla A.
- `KeyEventVK_ESCAPE` especifica la tecla ESCAPE.



Botón en estado normal.



Al pulsar la tecla se disparará el evento **`KeyPressed`**.

Al liberar la tecla se genera el evento **`KeyReleased`**.

EVENTOS DE RATÓN

De manera similar a los eventos de teclado, los eventos del ratón se generan como respuesta a que el usuario pulsa o libera un botón del ratón, o lo mueve sobre un componente.

Método	Causa de la invocación
`mousePressed (MouseEvent e)`	Se ha pulsado un botón del ratón en un componente.
`mouseReleased (MouseEvent e)`	Se ha liberado un botón del ratón en un componente.
`mouseClicked (MouseEvent e)`	Se ha pulsado y liberado un botón del ratón sobre un componente.
`mouseEntered (MouseEvent e)`	Se ha entrado (con el puntero del ratón) en un componente.
`mouseExited (MouseEvent e)`	Se ha salido (con el puntero del ratón) de un componente.

`MouseMotionListener` (oyente de ratón)

Método	Causa de la invocación
`mouseDragged (MouseEvent e)`	Se presiona un botón y se arrastra el ratón.
`mouseMoved (MouseEvent e)`	Se mueve el puntero del ratón sobre un componente.

`MouseWheelListener` (oyente de ratón)

Método	Causa de la invocación
`MouseWheelMoved (MouseWheelEvent e)`	Se mueve la rueda del ratón.

CREACIÓN DE CONTROLADORES DE EVENTOS

La utilidad de estos controladores o manejadores de evento es:

- Crear oyentes de evento sin tener que incluirlos en una clase propia.
- Esto aumente el rendimiento, ya que no "añade" otra clase.

Como inconveniente, destaca la dificultad de construcción: **los errores no se detectan en tiempo de compilación**, sino en tiempo de ejecución.
Por esta razón, es mejor crear controladores de evento con la ayuda de un asistente y documentarlos todo lo posible.

El uso más sencillo de `EventHandler` consiste en instalar un oyente que llama a un método, en el objeto objetivo sin argumentos. En el siguiente ejemplo creamos un `ActionListener` que invoca al método *dibujar* en una instancia de **`javax.Swing.JFrame`**.

```
miBoton.addActionListener(
        (ActionListener)EventHandler.create(ActionListener.class, frame, "dibujar"));
```

Cuando se pulse miBoton, se ejecutará la sentencia **`frame.dibujar()`**. Se obtendría el mismo efecto, con mayor seguridad en tiempo de compilación, definiendo una nueva implementación al interface `ActionListener` y añadiendo una instancia de ello al botón:

```
// Código equivalente empleando una clase interna en lugar de EventHandler.
miBoton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
                frame.dibujar();
        }
});
```

Probablemente el uso más típico de `EventHandler` es extraer el valor de una propiedad de la fuente del objeto evento y establecer este valor como el valor de una propiedad del objeto destino. En el siguiente ejemplo se crea un `ActionListener` que establece la propiedad "`label`" del objeto destino al valor de la propiedad "`text`" de la fuente (el valor de la propiedad "`source`") del evento.

```
EventHandler.create(ActionListener.class, miBoton, "label", "source.text")
```

Esto correspondería a la implementación de la siguiente clase interna:

```
// Código equivalente utilizando una clase interna en vez de EventHandler.
new ActionListener {
        public void actionPerformed(ActionEvent e) {
                miBoton.setLabel(((JTextField)e.getSource()).getText());
        }
}
```

CONTENEDORES

En Swing esa función la desempeñan un grupo de componentes llamados **contenedores** Swing.

Existen dos tipos de elementos contenedores:

- **Contenedores de alto nivel** o "peso pesado".
  - Marcos:**`JFrame`** y **`JDialog`** para aplicaciones
  - **`JApplet`**, para applets.
- **Contenedores de bajo nivel** o "peso ligero". Son los paneles: `JRootPane` y `JPanel`.

Cualquier aplicación, con interfaz gráfico de usuario típica, comienza con la apertura de una ventana principal, que suele contener la barra de título, los botones de minimizar, maximizar/restaurar y cerrar, y unos bordes que delimitan su tamaño.

Esa ventana constituye un marco dentro del cual se van colocando el resto de componentes que necesita el programador: menú, barras de herramientas, barra de estado, botones, casillas de verificación, cuadros de texto, etc.

Esa ventana principal o marco sería el contenedor de alto nivel de la aplicación.

Toda aplicación de interfaz gráfica de usuario Java tiene, al menos, un contenedor de alto nivel.

Los contenedores de alto nivel extienden directamente a una clase similar de AWT, es decir, `JFrame` extiende de `Frame`. Es decir, realmente necesitan crear una ventana del sistema operativo independiente para cada uno de ellos.

Los demás componentes de la aplicación no tienen su propia ventana del sistema operativo, sino que se dibujan en su objeto contenedor.

En los ejemplos anteriores del tema, hemos visto que podemos añadir un `JFrame` desde el diseñador de NetBeans, o bien escribiéndolo directamente por código. De igual forma para los componentes que añadamos sobre el.

CERRAR LA APLICACIÓN

Cuando quieres terminar la ejecución de un programa, ¿qué sueles hacer? Pues normalmente pinchar en el icono de cierre de la ventana de la aplicación.

En Swing, una cosa es cerrar una ventana, y otra es que esa ventana deje de existir completamente, o cerrar la aplicación completamente.

- **Se puede hacer que una ventana no esté visible**, y sin embargo que ésta siga existiendo y ocupando memoria para todos sus componentes, usando **el método `setVisible(false)`**. En este caso bastaría ejecutar para el **`JFrame`** el método **`setVisible(true)`** para volver a ver la ventana con todos sus elementos.
- **Si queremos cerrar la aplicación**, es decir, que no sólo se destruya la ventana en la que se mostraba, sino que se destruyan y liberen todos los recursos (memoria y CPU) que esa aplicación tenía reservados, tenemos que invocar al método `System.exit(0)`.
- **También se puede invocar para la ventana `JFrame` al método** `dispose()`, heredado de la clase **`Window`**, que no requiere ningún argumento, **y que borra todos los recursos de pantalla usados por esta ventana y por sus componentes, así como cualquier otra ventana que se haya abierto como hija de esta** (dependiente de esta). Cualquier memoria que ocupara esta ventana y sus componentes se libera y se devuelve al sistema operativo, y tanto la ventana como sus componentes se marcan como "no representables". Y sin embargo, el objeto ventana sigue existiendo, y podría ser reconstruido invocando al método **`pack()`** o la método **`show()`**, aunque deberían construir de nuevo toda la ventana.

Las ventanas `JFrame` de Swing permiten establecer una operación de cierre por defecto con el método `setDefaultCloseOperation()`, definido en la clase `JFrame`. ¿Cómo se le indica al método el modo de cerrar la ventana?

Los valores que se le pueden pasar como parámetros a este método son una serie de constantes de clase:

- **`DO_NOTHING_ON_CLOSE`**: **no hace nada**, necesita que el programa maneje la operación en el método `windowClosing()` de un objeto **`WindowListener`** registrado para la ventana.
- **`HIDE_ON_CLOSE`**: **Oculta** de ser mostrado en la pantalla pero no destruye el marco o ventana después de invocar cualquier objeto **`WindowListener`** registrado.
- **`DISPOSE_ON_CLOSE`**: **Oculta y termina (destruye) automáticamente el marco o ventana** después de invocar cualquier objeto **`WindowListener`** registrado.
- **`EXIT_ON_CLOSE:`** Sale de la aplicación usando el método `System.exit(0)`. Al estar definida en **`JFrame`**, se puede usar con aplicaciones, pero no con applets.

ORGANIZADOR DE CONTENEDORES: LAYOUT MANAGERS

Los layout managers son fundamentales en la creación de interfaces de usuario, ya que determinan las posiciones de los controles en un contenedor.

En lenguajes de programación para una única plataforma, el problema sería menor porque el aspecto sería fácilmente controlable. Sin embargo, dado que Java está orientado a la portabilidad del código, éste es uno de los aspectos más complejos de la creación de interfaces, ya que las medidas y posiciones dependen de la máquina en concreto.

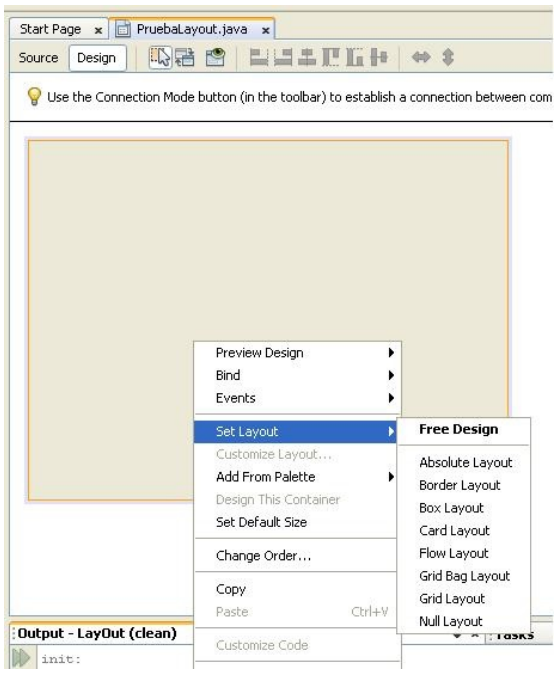
En algunos entornos los componentes se colocan con coordenadas absolutas. En Java se desaconseja esa práctica porque en la mayoría de casos es imposible prever el tamaño de un componente.

Por tanto, en su lugar, se usan organizadores o también llamados **administradores de diseño** o **layout managers** o **gestores de distribución** que permiten colocar y maquetar de forma independiente de las coordenadas.

Debemos hacer un buen diseño de la interfaz gráfica, y así tenemos que elegir el mejor gestor de distribución para cada uno de los contenedores o paneles de nuestra ventana.

Esto podemos conseguirlo con el método `setLayout()`, al que se le pasa como argumento un objeto del tipo de Layout que se quiere establecer.

En NetBeans, una vez insertado un `JFrame`, si nos situamos sobre él y pulsamos botón derecho, se puede ver, como muestra la imagen, que aparece un menú, el cual nos permite elegir el layout que queramos.



CONTENEDOR LIGERO: JPanel

Es la clase utilizada como contenedor genérico para agrupar componentes. Normalmente cuando una ventana de una aplicación con interfaz gráfica cualquiera presenta varios componentes, para hacerla más atractiva y ordenada al usuario se suele hacer lo siguiente:

- Crear un marco, un **`JFrame`**.
- Para organizar mejor el espacio en la ventana, añadiremos varios paneles, de tipo **`JPanel`**. (Uno para introducir los datos de entrada, otro para mostrar los resultados y un tercero como zona de notificación de errores.)
- Cada uno de esos paneles estará delimitado por un borde que incluirá un título. Para ello se usan las clases disponibles en **`BorderFactory`** (`BevelBorder`, `CompoundBorder`, `EmptyBorder`, `EtchedBorder`, `LineBorder`, `LoweredBevelBorder`, `MatteBorder` y `TitledBorder`) que nos da un surtido más o menos amplio de tipos de bordes a elegir.
- En cada uno de esos paneles se incluyen las etiquetas y cuadros de texto que se necesitan para mostrar la información adecuadamente.

Con NetBeans es tan fácil como arrastrar tantos controles `JPanel` de la paleta hasta el `JFrame`. Por código, también es muy sencillo, por ejemplo podríamos crear un panel rojo, darle sus características y añadirlo al JFrame del siguiente modo:

```
…
JPanel panelRojo = new JPanel();
panelRojo.setBackground(Color.RED);
panelRojo.setSize(300,300);
// Se crea una ventana
JFrame ventana=new JFrame("Prueba en rojo");
ventana.setLocation(100,100);
ventana.setVisible(true);
// Se coloca el JPanel en el content pane
Container contentPane=ventana.getContentPane();
contentPane.add(panelRojo);
…
```

Cuando en Sun desarrollaron Java, los diseñadores de Swing, por alguna circunstancia, determinaron que para algunas funciones, como añadir un `JComponent`, los programadores no pudiéramos usar `JFrame.add`, sino que en lugar de ello, primeramente, tuviéramos que obtener el objeto `Container` asociado con `JFrame.getContentPane()`, y añadirlo.

Sun se dio cuenta del error y ahora permite utilizar `JFrame.add`, desde Java 1.5 en adelante. Sin embargo, podemos tener el problema de que un código desarrollado y ejecutado correctamente en 1.5 falle en máquinas que tengan instalado 1.4 o anteriores.

ETIQUETAS Y CAMPOS DE TEXTO

Los **cuadros de texto** Swing vienen implementados en Java por la clase **`JTextField`.**

Para insertar un campo de texto, el procedimiento es tan fácil como: **seleccionar el botón correspondiente a `JTextField` en la paleta de componentes**, en el diseñador, y **pinchar sobre el área de diseño** encima del panel en el que queremos situar ese campo de texto. El tamaño y el lugar en el que se sitúe, dependerá del Layout elegido para ese panel.

El componente Swing etiqueta **`JLabel`**, se utiliza para crear etiquetas de modo que podamos insertarlas en un marco o un panel para visualizar un texto estático, que no puede editar el usuario. Los constructores son:

- `JLabel()`. Crea un objeto `JLabel` sin nombre y sin ninguna imagen asociada.
- `JLabel(Icon imagen)`. Crea un objeto sin nombre con una imagen asociada.
- `JLabel(Icon imagen, int alineacionHorizontal)`. Crea una etiqueta con la imagen especificada y la centra en horizontal.
- `JLabel(String texto)`. Crea una etiqueta con el texto especificado.
- `JLabel(String texto, Icon icono, int alineacionHorizontal)`. Crea una etiqueta con el texto y la imagen especificada y alineada horizontalmente.
- `JLabel(String texto, int alineacionHorizontal)`. Crea una etiqueta con el texto especificado y alineado horizontalmente.

Propiedades de los textos `JtextField`:

- ✔ **background**
  - ✔ Métodos asociados a la propiedad:
    - ✔ `setBackground()`
    - ✔ `getBackGround()`
  - ✔ Descripción:
    - ✔ Establece el color del fondo del cuadro de texto
  - ✔ Ejemplo de uso:
    - ✔ `jTextField1.setBackground`(java.awt.Color.GREEN);
    - ✔ `jTextField1.setBackground`(newjava.awt.Color(204, 255, 204));

- ✔ Color c = jTextField1.getBackground();

- ✔ **border**
  - ✔ Métodos asociados a la propiedad:
    - ✔ `setBorder()`
  - ✔ Descripción:
    - ✔ Permite seleccionar las caracteristicas del borde que tiene el cuadro de texto para delimitarlo.
  - ✔ Ejemplo de uso:
    - ✔ `jTextField1.setBorder`(new javax.swing.border.LineBorder (new java.awt.Color (0, 0, 255), 1, **`true`**));

- ✔ **editable**
  - ✔ Métodos asociados a la propiedad:
    - ✔ `setEditable()`
  - ✔ Descripción:
    - ✔ Establece si se va a poder modificar el texto del campo de texto por el usuario de la aplicación. Aunque se establezca su valor a **`false`**, el programa si que podrá modificar ese texto.
  - ✔ Ejemplo de uso:
    - ✔ `jTextField1.setEditable`(**`false`**);

- ✔ **font**
  - ✔ Métodos asociados a la propiedad:
    - ✔ `setFont()`
    - ✔ `getFont()`
  - ✔ Descripción:
    - ✔ Establece las propiedades de la fuente (tipo de letra, estilo y tamaño del texto)
  - ✔ Ejemplo de uso:
    - ✔ `jTextField1.setFont`(new java.awt.Font(“Dialog”, 1, 12));
    - ✔ Font f = `jTextField1.getFont()`;

- **foreground**
  - Métodos asociados a la propiedad:
    - `setForeground()`
  - Descripción:
    - Establece el color del texto
  - Ejemplo de uso:
    - `jTextField1.setForeground`(new java.awt.Color(0, 0, 255));
- **horizontalAlignment**
  - Métodos asociados a la propiedad:
    - `setHorizontalAlignment()`
  - Descripción:
    - Establece la justificación del texto dentro del cuadro de texto (Justificado a la izquierda, centrado o a la derecha)
  - Ejemplo de uso:
    - `jTextField1.setHorizontalAlignment`(javax.swing.jTextField.CENTER);
- **text**
  - Métodos asociados a la propiedad:
    - `setText()`
    - `getText()`
  - Descripción:
    - Establece el texto que contiene el cuadro de texto. También es posible mezclar texto de distintos colores y fuentes, usando etiquetas HTML como parte del texto
  - Ejemplo de uso:
    - `jTextField1.setText`(“Campo de texto”);
    - `String` texto = `jTextField1.getText()`;

- **toolTipText**
  - Métodos asociados a la propiedad:
    - `setToolTipText()`
    - `getToolTipText()`
  - Descripción:
    - Añade un comentario de ayuda flotante de forma que se despliega al detener el cursor encima del componente.
  - Ejemplo de uso:
    - `jTextField1.setToolTipText`(“Cuadro de texto”);
- **enabled**
  - Métodos asociados a la propiedad:
    - `setEnabled()`
  - Descripción:
    - Establece si el componente va a estar activo o no.Un componente que no está activo no puede responder a las entradas de usuario.Algunos componentes, como JtextField, modifican su representación visual para dejar claro que no están activos, mostrando un color gris atenuado.
  - Ejemplo de uso:
    - `jTextField1.setEnabled(false);`
- **maximumSize**
  - Métodos asociados a la propiedad:
    - `setMaximumSize()`
    - `getMaximumSize()`
  - Descripción:
    - Establece el tamaño máximo para el componente.
  - Ejemplo de uso:
    - `jTextField1.setMaximumSize`(new java.awt.Dimension (200, 50));

- **minimumSize**
  - Métodos asociados a la propiedad:
    - `setMinimumSize()`
    - `getMinimumSize()`
  - Descripción:
    - Establece el tamaño mínimo para el componente.
  - Ejemplo de uso:
    - `jTextField1.setMinimumSize`(new java.awt.Dimension(50, 10));
- **preferredSize**
  - Métodos asociados a la propiedad:
    - `setPreferredSize()`
    - `setPreferredSize()`
  - Descripción:
    - Establce el tamaño preferido para el componente. El valor preferido se tendrá en cuenta a la hora de redimensionar la ventana, según el Layout que se haya usado, para procurar que el aspecto sea siempre el mejor posible, pero no hay garantías de que se respete siempre.
  - Ejemplo de uso:
    - `jTextField1.setPreferredSize`(new java.awt.Dimension(175, 35));
- **opaque**
  - Métodos asociados a la propiedad:
    - `setOpaque()`
  - Descripción:
    - Establece si el componente va a ser opaco, o si por el contrario va a permitir que se vean otros componentes que puedan estar debajo de él, como si fuera un cristal transparente.
  - Ejemplo de uso:
    - jTextField1.setOpaque(`true`);

BOTONES

Ya has podido comprobar que prácticamente todas las aplicaciones incluyen botones que al ser pulsados efectúan alguna acción: hacer un cálculo, dar de alta un libro, aceptar modificaciones, etc.

Estos botones se denominan **botones de acción**, precisamente porque realizan una acción cuando se pulsan. En Swing, la clase que los implementa en Java es la `JButton`.

Los principales métodos son:

- `void setText(String).` Asigna el texto al botón.
- `String getText().` Recoge el texto.

Se pueden asociar imágenes con los botones, mediante `ImageIcon`, que permite especificar fácilmente un archivo de imagen (jpeg o GIF, incluyendo GIF animados). La forma más fácil de asociar una imagen con un `JButton` es pasar el `ImageIcon` al constructor. Un JButton realmente permite siete imágenes asociadas:

1. La imagen principal (se usa `setIcon` para especificarla, si no se proporciona en el constructor).
2. La imagen a usar cuando el botón se presiona (`setPressedIcon`).
3. La imagen a usar cuando el ratón está sobre el (`setRolloverIcon`, aunque tienes que llamar primero a `setRolloverEnabled(true)`).
4. La imagen a usar cuando el botón está deshabilitado (`setDisabledIcon`),
5. La imagen a usar cuando el botón está seleccionado y habilitado (`setSelectedIcon`).
6. La imagen a usar cuando está seleccionado pero deshabilitado (`setDisabledSelectedIcon`).
7. La imagen a utilizar cuando el ratón está sobre ello mientras se selecciona (`setRolloverSelectedIcon`).

Hay un tipo especial de botones, que se comportan como **interruptores de dos posiciones** o estados (pulsados-on, no pulsados-off). Esos botones especiales se denominan botones on/off o `JtoggleButton`.

EJEMPLO DE GESTIÓN DE EVENTOS PARA UN BOTÓN: ActionListener

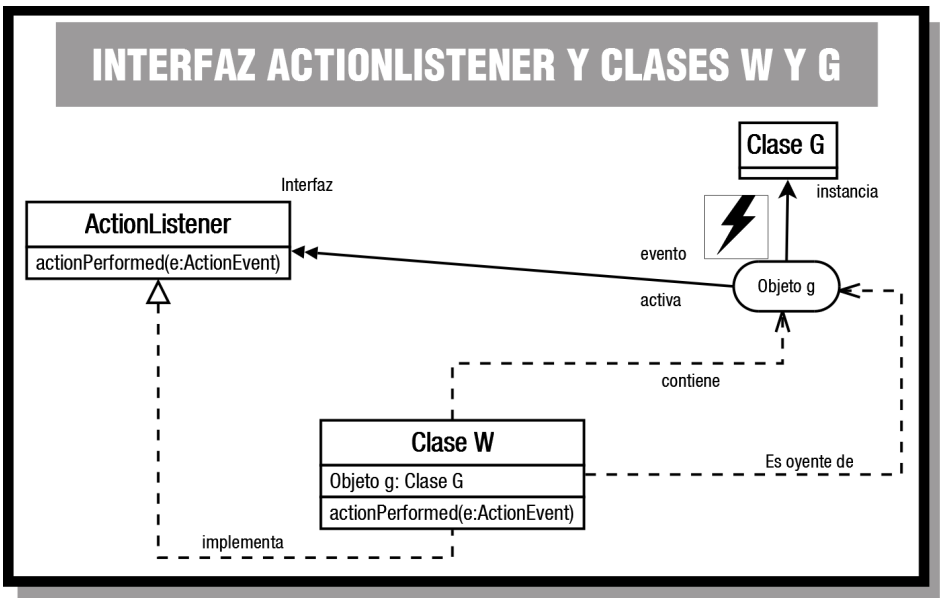
Vamos a ver un ejemplo de una **interfaz** proporcionada por la API de Java que puede ser implementada por alguna clase creada por ti dentro de una pequeña aplicación. Hemos escogido la interfaz `ActionListener`.

La **interfaz** `ActionListener` ya la has utilizado en la unidad dedicada a las **interfaces gráficas**. Las clases que quieran realizar una determinada acción cada vez que se produzca cierto **evento** en el sistema deben implementar esta **interfaz**. Este tipo de **interfaces** se encuentran dentro de los Event **Listeners** u "**oyentes de eventos**" y son útiles para detectar que se ha producido un determinado **evento** asíncrono durante la ejecución de tu aplicación (pulsación de una tecla, clic de ratón, etc.). Son intensivamente utilizadas en el desarrollo de las **interfaces gráficas de usuario**.

Vamos a ver un ejemplo lo más sencillo posible: se trata de desarrollar una pequeña aplicación de escritorio (utilizando la API de **Swing**) con una **ventana** que contenga un par de **botones** y que al pulsar alguno de esos **botones** (**evento**), la aplicación sea capaz de detectar que se ha producido ese **evento** y por tanto realizar una determinada **acción** (por ejemplo generar un sonido o mostrar un determinado texto).

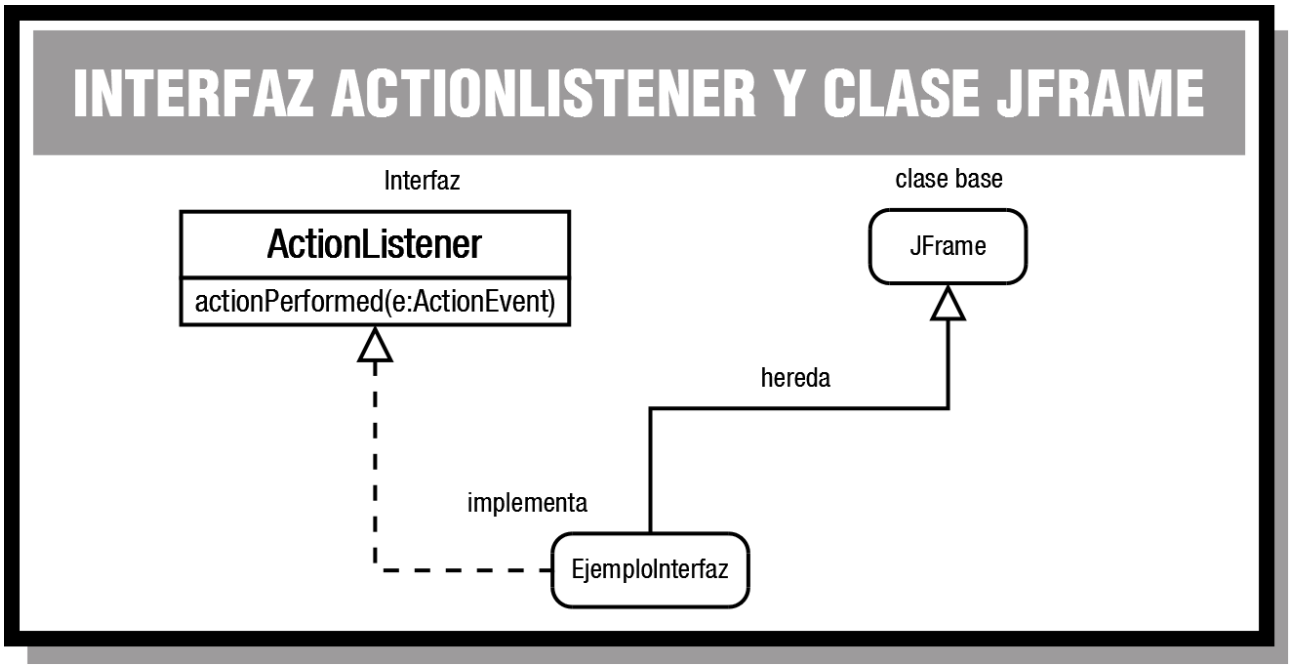
Para ello es necesario que la clase que vaya a gestionar el evento sea un "**oyente**" de ese evento, es decir, que sea capaz de enterarse de que se ha producido ese evento. Esa capacidad o habilidad la proporciona la API de Java a través de las interfaces de tipo "**oyente**". En concreto vamos a implementar la **interfaz ActionListener**.

La `interface ActionListener` contiene un único método: `actionPerformed`. Toda clase **W** que implemente esta interfaz tendrá que definir ese método. Si incrustamos un determinado objeto gráfico **G** en una ventana **W** y asociamos a ese objeto gráfico la clase **W**, indicando que es su **oyente y gestor de eventos**, cuando se produzca algún evento sobre **G** se ejecutará el método `actionPerformed` implementado por **W**.



En nuestro ejemplo podríamos hacer lo siguiente:

- Definir una **ventana principal**, por ejemplo una clase basada en el tipo **marco** o **frame** (**subclase** de `JFrame`) que implemente la **interfaz** `ActionListener`.
- Definir uno o varios **botones** (objetos de la clase `JButton`) dentro de la ventana.
- Asociar como **oyente** a los objetos de tipo **botón** la propia **ventana principal** (**frame**), pues va a ser capaz de gestionar **eventos** (implementa el método `actionPerformed`).
- Implementar el método `actionPerformed` como un **método de la ventana principal** (**frame**) para **reaccionar** de algún modo cuando se produzca algún **evento desencadenado por un botón** (pues se ha establecido a la ventana principal como **oyente** y **gestora de los eventos** del botón).



CASILLAS DE VERIFICACIÓN Y BOTONES DE RADIO

Las casillas de verificación en Swing están implementadas para Java por la clase **`JCheckBox`**, y los botones de radio o de opción por la clase **`JRadioButton`**. Los grupos de botones, por la clase **`ButtonGroup`**.

La funcionalidad de ambos componentes es en realidad la misma.

- Ambos tienen **dos "estados**": seleccionados o no seleccionados (marcados o no marcados).
- Ambos **se marcan o desmarcan** usando el método **`setSelected(boolean estado)`**, que establece el valor para su propiedad **`selected`**. (El estado toma el valor **`true`** para seleccionado y **`false`** para no seleccionado).
- A ambos le **podemos preguntar si están seleccionados o no**, mediante el método **`isSelected()`**, que devuelve **`true`** si el componente está seleccionado y **`false`** si no lo está.
- Para ambos **podemos asociar un icono distinto** para el estado de **seleccionado** y el de **no seleccionado**.

`JCheckBox` pueden usarse en menús mediante la clase `JCheckBoxMenuItem`.

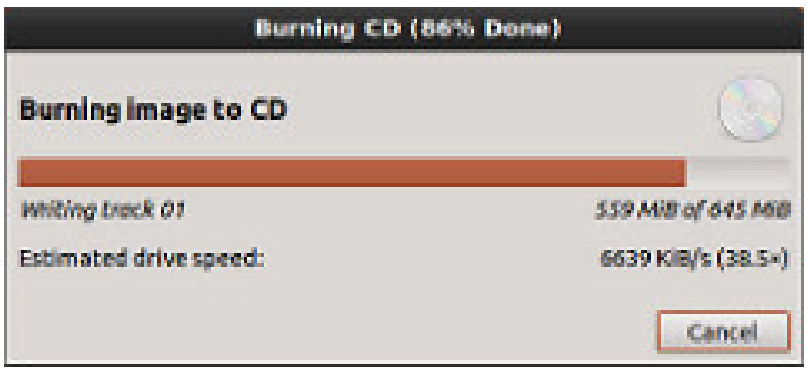
`JButtonGroup` permite agrupar una serie de casillas de verificación (`JRadioButton`), de entre las que sólo puede seleccionarse una. Marcar una de las casillas implica que el resto sean desmarcadas automáticamente. La forma de hacerlo consiste en añadir un `JButtonGroup` y luego, agregarle los botones.

Cuando en **un contenedor** aparezcan **agrupados** varios **botones de radio** (o de opción), **entenderemos** que no son opciones independientes, sino que sólo uno de ellos podrá estar activo en cada momento, y necesariamente uno debe estar activo. Por tanto en ese contexto entendemos que son opciones excluyentes entre sí.

BARRAS DE PROGRESO

¿Alguna vez has creído que un programa se te había quedado "colgado" y no respondía, y realmente sí estaba pasando algo, y finalmente terminaba el proceso? Seguro que más de una vez, y la solución para que no ocurra eso es poner una barra de progreso que muestre al usuario que el proceso está ahí, que algo está pasando aunque sea lentamente.

En la web encontrarás un montón de ejemplos de implementación, alguno con hilos, a continuación, en el siguiente enlace te mostramos un enlace a una implementación muy sencilla y explicada.



LISTAS

En casi todos los programas, nos encontramos con que se pide al usuario que introduzca un dato, y además un dato de entre una lista de valores posibles, no un dato cualquiera.

La clase `JList` constituye un componente lista sobre el que se puede ver y seleccionar uno o varios elementos a la vez. En caso de haber más elementos de los que se pueden visualizar, es posible utilizar un componente `JScrollPanel` para que aparezcan barras de desplazamiento.

En los componentes `JList`, un modelo code>ListModel representa **los contenidos de la lista**. Esto significa que los datos de la lista se guardan en una estructura de datos independiente, denominada modelo de la lista. Es fácil mostrar en una lista los elementos de un vector o array de objetos, usando un constructor de `JList` que cree una instancia de `ListModel` automáticamente a partir de ese vector.

Vemos a continuación un ejemplo parar crear una lista `JList` que muestra las cadenas contenidas en el vector info[]:

```
String[] info = {"Pato", "Loro", "Perro", "Cuervo"};
JList listaDatos = new JList(info);
/* El valor de la propiedad model de JList es un objeto que proporciona una visión de sólo lectura del vector info[].
El método getModel() permite recoger ese modelo en forma de Vector de objetos, y utilizar con los métodos de la clase
Vector, como getElementAt(i), que proporciona el elemento de la posición i del Vector.  */
for (int i = 0; i < listaDatos.getModel().getSize(); i++) {
System.out.println(listaDatos.getModel().getElementAt(i));
}
```

LISTAS (II)

Cuando trabajamos con un componente `JList`, podemos seleccionar un único elemento, o varios elementos a la vez, que a su vez pueden estar contiguos o no contiguos. La posibilidad de hacerlo de una u otra manera la establecemos con la propiedad `selectionMode`.

Los valores posibles para la propiedad `selectionMode` para cada una de esas opciones son las siguientes constantes de clase del interface **`ListSelectionModel`**:

- **`MULTIPLE_INTERVAL_SELECTION`**: Es el valor por defecto. Permite seleccionar múltiples intervalos, manteniendo pulsada la tecla CTRL mientras se seleccionan con el ratón uno a uno, o la tecla de mayúsculas, mientras se pulsa el primer elemento y el último de un intervalo.

- **`SINGLE_INTERVAL_SELECTION`**: Permite seleccionar un único intervalo, manteniendo pulsada la tecla mayúsculas mientras se selecciona el primer y último elemento del intervalo.
- **`SINGLE_SELECTION`**: Permite seleccionar cada vez un único elemento de la lista.

Podemos establecer el valor de la propiedad `selectedIndex` con el método `setSelectedIndex()` es decir, el índice del elemento seleccionado, para seleccionar el elemento del índice que se le pasa como argumento.

Como hemos comentado, los datos se almacenan en un modelo que al fin y al cabo es un vector, por lo que tiene sentido hablar de índice seleccionado.

También se dispone de un método `getSelectedIndex()` con el que podemos averiguar el índice del elemento seleccionado.

El método `getSelectecValue()` devuelve el objeto seleccionado, de tipo `Object`, sobre el que tendremos que aplicar un "casting" explícito para obtener el elemento que realmente contiene la lista (por ejemplo un `String`). Observa que la potencia de usar como modelo un vector de `Object`, es que en el `JList` podemos mostrar realmente cualquier cosa, como por ejemplo, una imagen.

El método `setSelectedValue()` permite establecer cuál es el elemento que va a estar seleccionado.

Si se permite hacer selecciones múltiples, contamos con los métodos:

- `setSelectedIndices()`, al que se le pasa como argumento un vector de enteros que representa los índices a seleccionar.
- `getSelectedIndices()`, que devuelve un vector de enteros que representa los índices de los elementos o ítems que en ese momento están seleccionados en el `JList`.
- `getSelectedValues()`, que devuelve un vector de `Object` con los elementos seleccionados en ese momento en el `JList`.

LISTAS DESPLEGABLES

Una **lista desplegable** se representa en Java con el componente Swing `JComboBox`. Consiste en una lista en la que sólo se puede elegir una opción. Se pueden crear `JComboBox` tanto editables como no editables.

Una lista desplegable es una mezcla entre un campo de texto editable y una lista. Si la propiedad **`editable`** de la lista desplegable la fijamos a verdadero, o sea a **`true`**, el usuario podrá seleccionar uno de los valores de la lista que se despliega al pulsar el botón de la flecha hacia abajo y dispondrá de la posibilidad de teclear directamente un valor en el campo de texto.

Establecemos la propiedad **`editable`** del **`JComboBox`** el método **`setEditable()`** y se comprueba con el método **`isEditable()`**. La clase **`JComboBox`** ofrece una serie de métodos que tienen nombres y funcionalidades similares a los de la clase **`JList`**.

MENÚS

En las aplicaciones informáticas siempre se intenta que el usuario disponga de un menú para facilitar la localización una operación. La filosofía, al crear un menú, es que contenga todas las acciones que el usuario pueda realizar en la aplicación. Lo más normal y útil es hacerlo clasificando o agrupando las operaciones por categorías en submenús.

En Java usamos los componentes `JMenu` y `JMenuItem` para crear un menú e insertarlo en una barra de menús. La barra de menús es un componente `JMenuBar`.

Podemos crear un menú con la paleta de componentes, la manera más fácil, o bien por código.

También podríamos construir un menú directamente por código. Los constructores son los siguientes:

- `JMenu()`. Construye un menú sin título.
- `JMenu(String s)`. Construye un menú con título indicado por s.
- `JMenuItem()`. Construye una opción sin icono y sin texto.
- `JMenuItem(Icon icono)`. Construye una opción con icono y con texto.

Podemos construir por código un menú sencillo como el de la imagen con las siguientes sentencias:

```
// Crear la barra de menú
JMenuBar barra = new JMenuBar();
// Crear el menú Archivo
JMenu menu = new JMenu("Archivo");
// Crear las opciones del menú
JMenuItem opcionAbrir = new JMenuItem("Abrir");
menu.add(opcionAbrir);
JMenuItem opcionguardar = new JMenuItem("Guardar");
menu.add(opcionguardar);
JMenuItem opcionSalir = new JMenuItem("Salir");
menu.add(opcionSalir);
// Añadir las opciones a la barra
barra.add(menu);
// Establecer la barra
setJMenuBar(barra);
```

Frecuentemente, dispondremos de alguna opción dentro de un menú, que al elegirla, nos dará paso a un conjunto más amplio de opciones posibles.

Cuando en un menú un elemento del mismo es a su vez un menú, se indica con una flechita al final de esa opción, de forma que se sepa que, al seleccionarla, nos abrirá un nuevo menú.

Para incluir un menú como submenú de otro basta con incluir como ítem del menú, un objeto que también sea un menú, es decir, incluir dentro del `JMenu` un elemento de tipo `Jmenu`.

SEPARADORES

A veces, en un menú pueden aparecer distintas opciones. Por ello, nos puede interesar destacar un determinado grupo de opciones o elementos del menú como relacionados entre sí por referirse a un mismo tema, o simplemente para separarlos de otros que no tienen ninguna relación con ellos.

El **componente Swing que tenemos en Java para esta funcionalidad es:** `JSeparator`, que dibuja una línea horizontal en el menú, y así separa visualmente en dos partes a los componentes de ese menú. En la imagen podemos ver cómo se han separado las opciones de Abrir y Guardar de la opción de Salir, mediante este componente.

Si lo hacemos de manera gráfica, simplemente arrastramos un `JSeparator` a donde nos interese. Si lo hacemos por código, al código anterior, tan sólo habría que añadirle una línea, la que resaltamos en negrita:

```
...
menu.add(opcionguardar);
menu.add(new JSeparator());
JMenuItem opcionSalir = new JMenuItem("Salir");
...
```

ACELERADORES DE TECLADO Y MNEMÓNICOS

A veces, tenemos que usar una aplicación con interfaz gráfica y no disponemos de ratón, porque se nos ha roto, o cualquier causa.

Además, cuando diseñamos una aplicación, debemos preocuparnos de las características de accesibilidad.

Algunas empresas de desarrollo de software obligan a todos sus empleados a trabajar sin ratón al menos un día al año, para obligarles a tomar conciencia de la importancia de que todas sus aplicaciones deben ser accesibles, usables sin disponer de ratón.

Ya no sólo en menús, sino en cualquier componente interactivo de una aplicación: campo de texto, botón de acción, lista desplegable, etc., es muy recomendable que pueda seleccionarse sin el ratón, haciendo uso exclusivamente del teclado.

Para conseguir que nuestros menús sean accesibles mediante el teclado, la idea es usar aceleradores de teclado o **atajos de teclado** y de **mnemónicos**.

Un acelerador o **atajo de teclado** es una combinación de teclas que se asocia a una opción del menú, de forma que pulsándola se consigue el mismo efecto que abriendo el menú y seleccionando esa opción.

Esa combinación de teclas **aparece escrita a la derecha de la opción del menú**, de forma que el usuario pueda tener conocimiento de su existencia.

Para añadir un atajo de teclado, lo que se hace es emplear la propiedad **accelerator** en el diseñador

Los atajos de teclado **no suelen** asignarse a todas las opciones del menú, sino **sólo a** las que más se usan.

Un **mnemónico** consiste en resaltar una tecla dentro de esa opción, mediante un subrayado, de forma que pulsando + <el mnemónico> se abra el menú correspondiente. Por ejemplo en la imagen con Alt + A abriríamos ese menú que se ve, y ahora con Ctrl+G se guardaría el documento.

Para añadir mediante código un mnemónico a una opción del menú, se hace mediante la **propiedad mnemonic**. Los mnemónicos sí que deberían ser incluidos para todas las opciones del menú, de forma que todas puedan ser elegidas haciendo uso sólo del teclado, mejorando así la accesibilidad de nuestra aplicación.

BARRAS DE HERRAMIENTAS

Frecuentemente, en la mayoría de aplicaciones informáticas se introducen Barras de Botones con imágenes de las operaciones principales (o más habituales) que realiza dicha aplicación. Especialmente para abrir y cerrar ficheros, guardar cambios, o las típicas cortar, copiar y pegar, etc.

¿Crees que será útil disponer de la misma funcionalidad repetida en varias partes de la aplicación?

Por **ejemplo**, si ya tenemos la opción Salir en un menú para terminar la ejecución de la aplicación, ¿resultará útil disponer de esa funcionalidad en algún otro sitio, como por ejemplo un botón?

Desde luego, resulta tan útil, que es lo que suelen hacer la mayoría de las aplicaciones, repetir la funcionalidad de sus menús en botones.

Normalmente será rentable repetir funcionalidades para aquellos aspectos de nuestra aplicación que se ejecuten con mucha frecuencia, garantizando un acceso más rápido que por medio de los menús. En tales casos, a estas opciones se les suele asociar un icono, de forma que el botón que duplica la funcionalidad lo único que contiene es ese icono, usualmente sin texto alguno.

Por tanto es el icono común el que identifica que se trata de una función común.

Pero imagina que tenemos muchas funciones en nuestra aplicación que por su importancia queremos tener repetidas en botones. Podría resultar incómodo ir añadiendo montones de botones sueltos por toda la ventana de nuestra aplicación. Por tanto, resulta mucho más cómodo agruparlos todos en una barra de herramientas. Seguramente te resulta familiar el uso de estas barras de herramientas, ya que la mayoría de las aplicaciones disponen de alguna de ellas, pero ¿cómo se añade una barra de herramientas a nuestra aplicación Java?

- **En Java las barras de herramientas se añaden como objetos de la clase `JToolBar`**,y haciendo uso del diseñador**.**
- **Basta con añadir un objeto `JToolBar` con el ratón al área de diseño, y luego añadir los botones a esa barra de tareas, en vez de colocarlos en un panel, por ejemplo.**
- Si queremos que los botones sólo contengan una imagen, basta con eliminarles el texto en la **propiedad `text`** y asociarles un icono con la **propiedad `icon`** desde el diseñador.

Las principales propiedades de una barra de herramientas **`JToolBar`** son:

- `floatable`: Indica que va a ser (o no) una barra flotante, que va a poder desplazarse en forma de ventana flotante al lugar de la ventana de la aplicación al que la desplacemos arrastrándola con el ratón, o estar fija en una posición.
- `orientation`: Si toma el valor 0, la barra estará orientada en horizontal. Si toma el valor 1, estará orientada en vertical.
- `rollover`: Indica si los botones de la barra van a tener (o no) bordes alrededor

Por último añadir que es conveniente que todos los botones de una barra de herramientas tengan las mismas dimensiones, para que el aspecto sea visualmente más agradable.