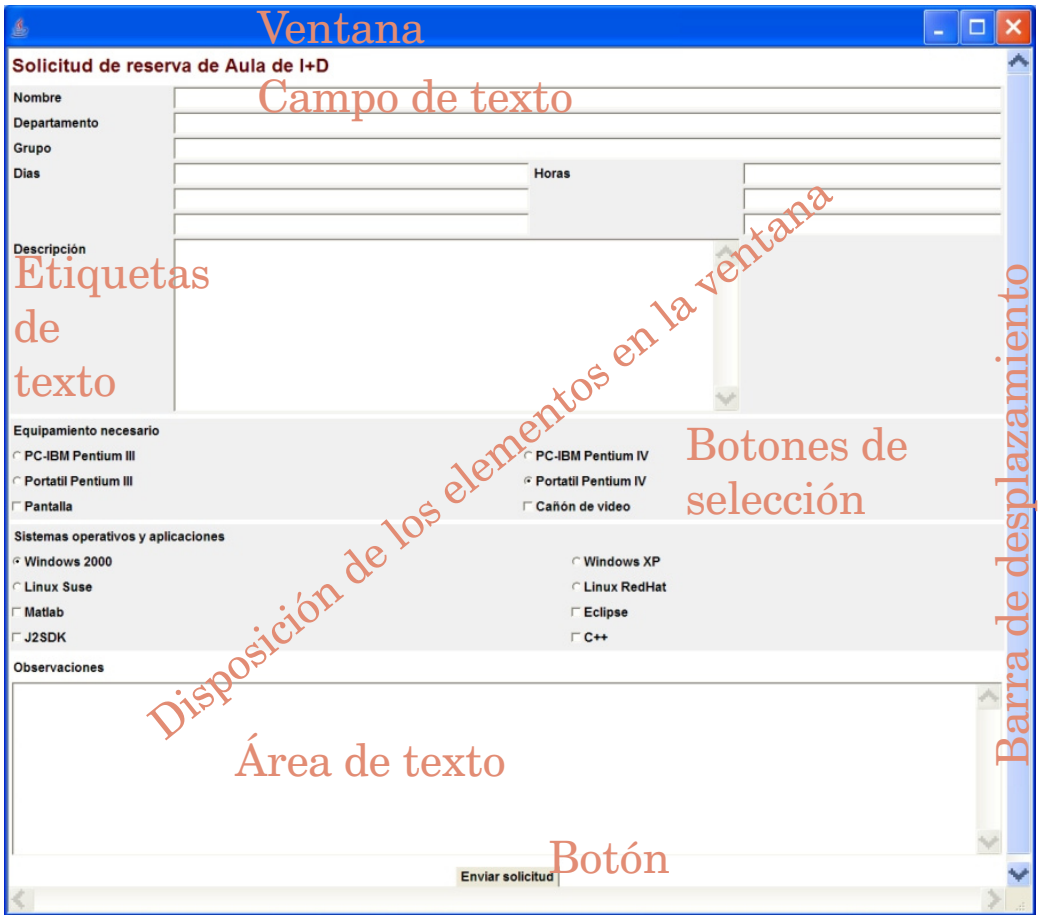
Tema 5

Interfaces Gráficas de Usuario

Departament d'Informàtica. Universitat de València

Índice



- Las interfaces de usuario se encargan de la comunicación con el usuario.
- Muchas veces el éxito de una aplicación depende de su interfaz (de poco sirve que una aplicación funcione bien si el usuario encuentra que es muy complicada la interfaz)
- Las interfaces con el usuario no tienen por qué ser únicamente gráficas. De hecho un concepto más amplio es el de Interacción Hombre-Máquina (hay gente trabajando en cosas como *Brain-Computer Interfaces*, *Perceptive User Interfaces* y se organizan conferencias sobre el tema).

Los factores que hay que tener en cuenta a la hora de diseñar una interfaz de usuario son:

- Sicología del usuario.

- *Se deberían tener en cuenta los conocimientos del usuario, sus capacidades y sus requisitos.*
- *Pensad en toda la tecnología que queda cuidadosamente oculta tras los botones y vínculos que hay tras un navegador de Internet. ¿Cuanta gente utilizaría Internet si tuviese que conocer los protocolos?*

- Comunicación visual

- *Los elementos visuales como iconos están bien y pueden facilitar la comunicación del usuario con la aplicación pero... ojo los símbolos pueden tener diferentes interpretaciones en función de cultura o región geográfica.*

¿Qué defecto tiene la interfaz gráfica de la actualidad? Que es una solución desproporcionada. Hacerlo todo visible es una buena idea cuando sólo hay veinte cosas que mostrar. Cuando hay veinte mil, sólo crea más confusión. Si se muestra todo a la vez el resultado es el caos[1].

- Colocación de los componentes.

- *En la interface de ejemplo los componentes están agrupados en categorías esto puede facilitar su utilización. Por ejemplo si hay que seleccionar un item de una lista para realizar una consulta a una base de datos y mostrar los datos en una tabla lo razonable sería poner esa lista al principio y no al final de la ventana.*

- Ingeniería de uso.

- *Métodos formales que se pueden aplicar a lo largo de todo el proceso: diseño, desarrollo y validación de las interfaces de usuario.*

En este tema vamos a ver algunos aspectos de JFC (Java Foundation Classes). JFC es un conjunto de componentes para trabajo con interfaces gráficas de usuario en Java.

Contiene:

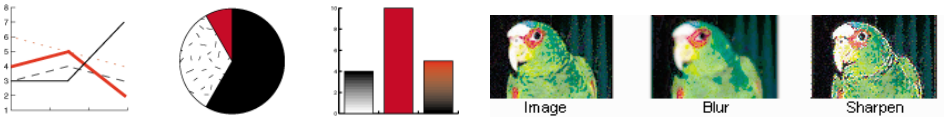
- Abstract Window Toolkit (AWT)

- *API para el diseño de interfaces gráficas de usuario que se integran en el sistema de ventanas nativo del sistema donde se ejecutan, incluyendo APIs para arrastrar y soltar.*

- Java 2D

[1] El ordenador invisible. Donald A. Norman. Paidós.

- *APIs para trabajar con gráficos 2D, trabajo con imágenes, texto e impresión.*



- Swing

- *APIs que extienden AWT para proporcionar una biblioteca de componentes para el diseño de interfaces gráficas de usuario enteramente realizadas en Java.*

- Accesibilidad

- *APIs para permitir que las aplicaciones sean accesibles a las personas con discapacidades.*

- Internacionalización

- *Todas estas APIs incluyen soporte para crear aplicaciones que puedan ser utilizadas independientemente de la localización del usuario.*

1. Programación dirigida por eventos

- En el contexto de las interfaces gráficas de usuario (GUI), un **evento** es un objeto que describe un cambio de estado en otro objeto (fuente del evento).

- Los eventos se producen debido a la interacción del usuario con los componentes de la GUI: al pulsar un botón, al pulsar una tecla sobre un campo de texto, al seleccionar un elemento de una lista, al minimizar una ventana, al maximizarla, etc.

Ejemplos:

- La forma en la que hay que actuar es: realiza un método que indique qué es lo que se debe realizar cuando se produzca un determinado evento asumiendo que cuando este evento se produzca el método que se ha realizado será llamado.

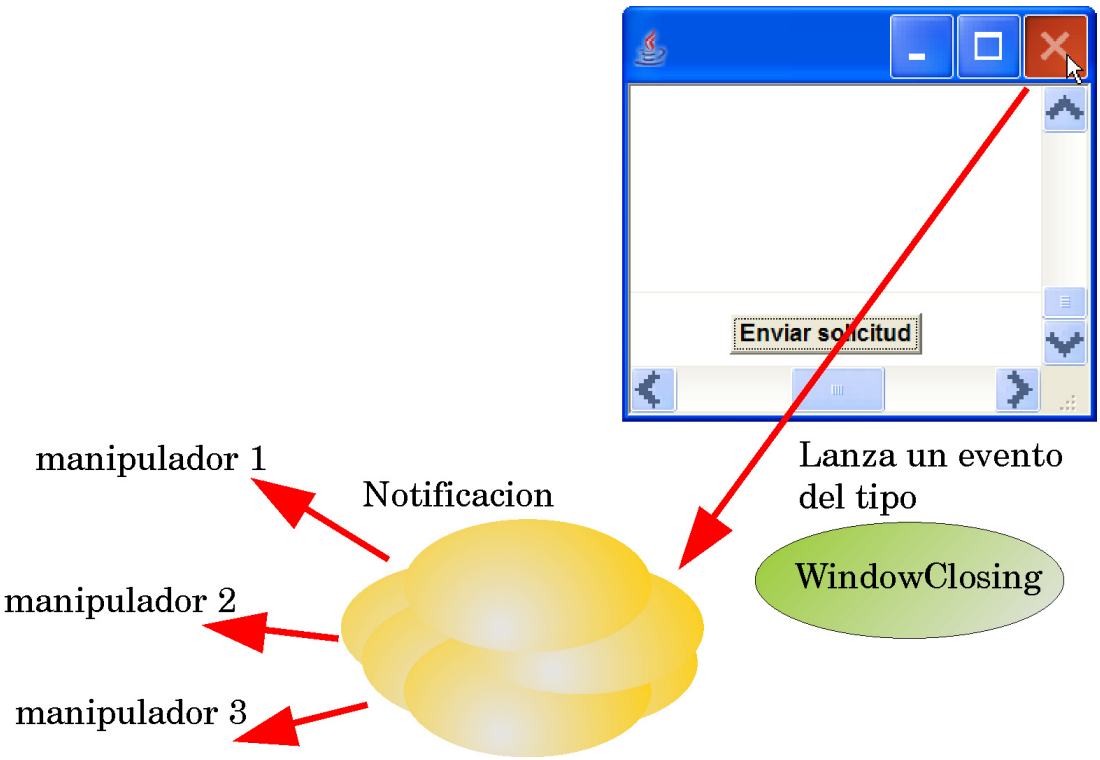
- La idea básica de la programación orientada a eventos es crear objetos con métodos que manejan los eventos apropiados, sin una atención explícita a cómo o cuando serán llamados estos métodos.

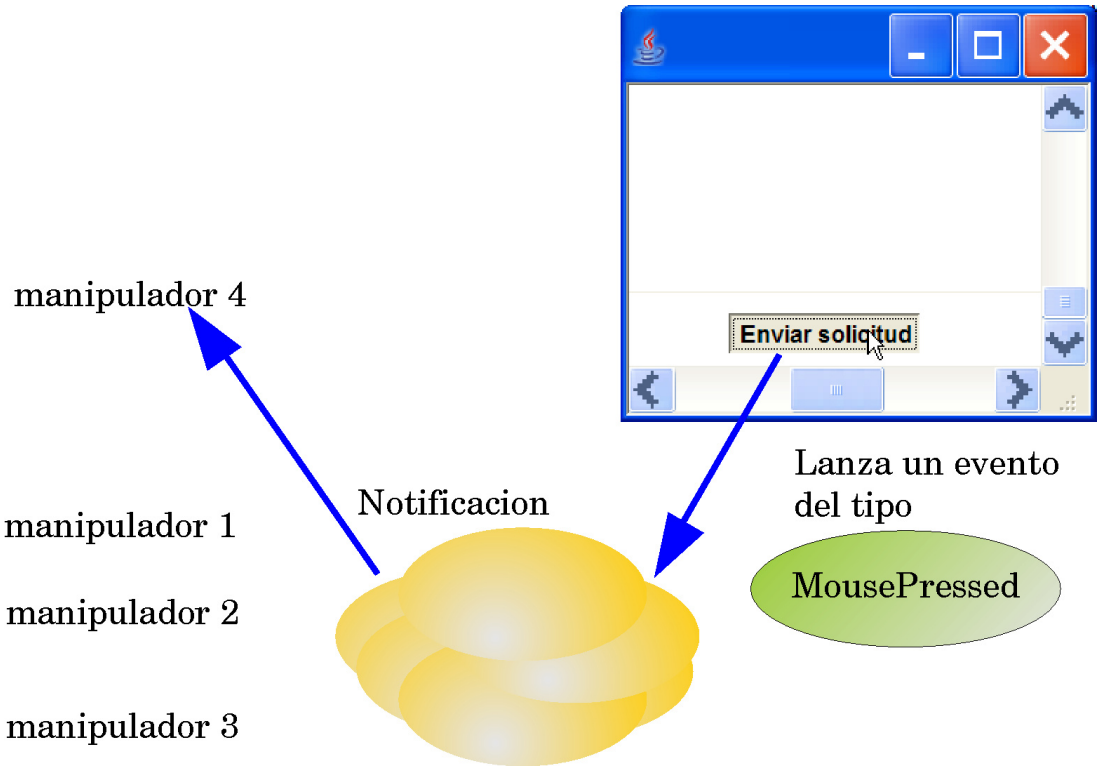
- Si $\mathcal{A}$ es un suceso o evento que se produce en algún instante, el objetivo de estos métodos es responder a preguntas del tipo:

  ¿Que se debe realizar cuando $\mathcal{A}$ suceda?

- Puesto que $\mathcal{A}$ es un evento (o suceso) y estos métodos especifican lo que hay que realizar cuando ocurra, estos métodos se conocen como **manipuladores** (*handler*) de eventos.

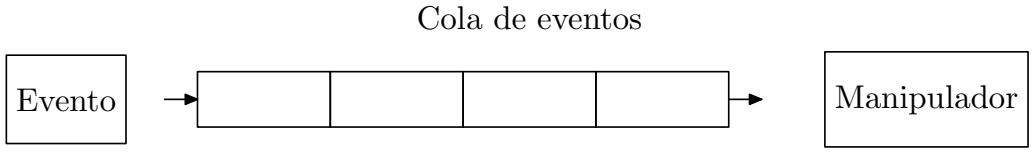
- Al escribir un método de respuesta ante un evento se asume que de algún modo el método será llamado cuando el evento se produzca.


Enviar solicitud
manipulador 1
manipulador 2
manipulador 3
Notificacion
Lanza un evento
del tipo
WindowClosing


Enviar solicitud
manipulador 4
manipulador 1
manipulador 2
manipulador 3
Notificacion
Lanza un evento
del tipo
MousePressed

- Como se puede observar en los dos ejemplos anteriores la producción y el tratamiento del evento se realizan en lugares diferentes.
- El productor del evento no necesita conocer qué acciones se van a realizar cuando genere un evento.
- Los manipuladores de eventos no necesitan conocer cómo serán llamados sus métodos.
- Los sistemas que utilizan esta separación entre productores de eventos y manipuladores de eventos generalmente contienen un componente adicional: una **cola de eventos** (*event queue*).
- Generalmente, la cola de eventos se proporcionada como una parte más dentro del sistema basado en eventos y Java no es una excepción.
- Es decir, no hay que definir (mediante una serie de clases) ni crear explícitamente una cola eventos para realizar una interfaz gráfica de usuario.

La cola de eventos sirve como intermediario entre el objeto que produce el evento y el objeto que lo debe manipular. Gráficamente:



La cola de eventos sirve de almacén de eventos de tal forma que cuando se produce una acción sobre un objeto que puede ser considerada como un evento es almacenada en la cola de eventos.

Además de servir como almacén, la cola de eventos tiene un *active instruction-follower* [2] que extrae un evento (típicamente el que más tiempo lleva en la cola) y notifica a los manipuladores interesados en el evento.

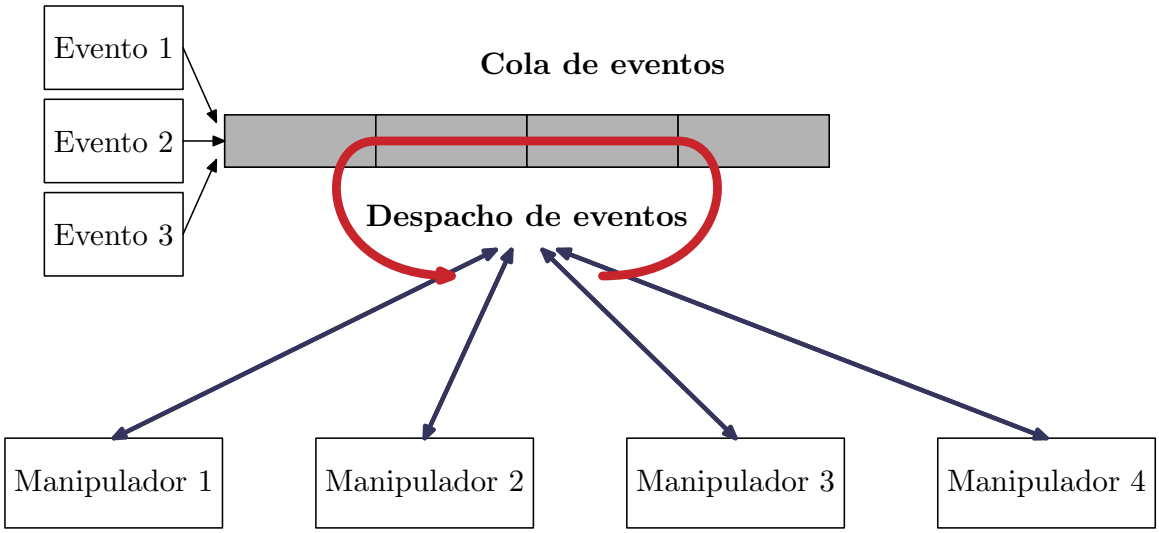
- La parte encargada de extraer el evento y notificar a los interesados se conoce como **despachador de eventos** (*event dispatcher*).
- Puesto que hasta que no se finalice con el tratamiento de un evento evento no se procesará el siguiente (debido a la ejecución secuencial no solapada), es importante que el manipulador finalice rápidamente.
- En Java, JFC proporciona la cola de eventos y el despachador de eventos. Estos objetos están ocultos y en principio no es necesario trabajar con ellos directamente.

[2] Ejecución secuencial no solapada. Significa se extrae un evento de la cola y hasta que no se procese no se pasará al siguiente. Es un funcionamiento similar una receta.

- ¿En qué momento se crean la cola y el despachador?
- Nada más crear y mostrar una ventana o un applet ya que a partir de ese momento el usuario debe poder actuar sobre la GUI.
- Por tanto al mostrar la GUI se crea una cola de eventos (`EventQueue`) que tiene como atributo un objeto del tipo (`EventDispatchThread`)[3].
- Este hilo inspecciona la cola de eventos y cuando se produce alguno determina qué se debe hacer. Los criterios que sigue son:

1. Sólo es posible despachar un evento cada vez
2. Si el evento $\mathcal{A}$ se encola antes que el $\mathcal{B}$ entonces $\mathcal{A}$ será despachado antes que $\mathcal{B}$.

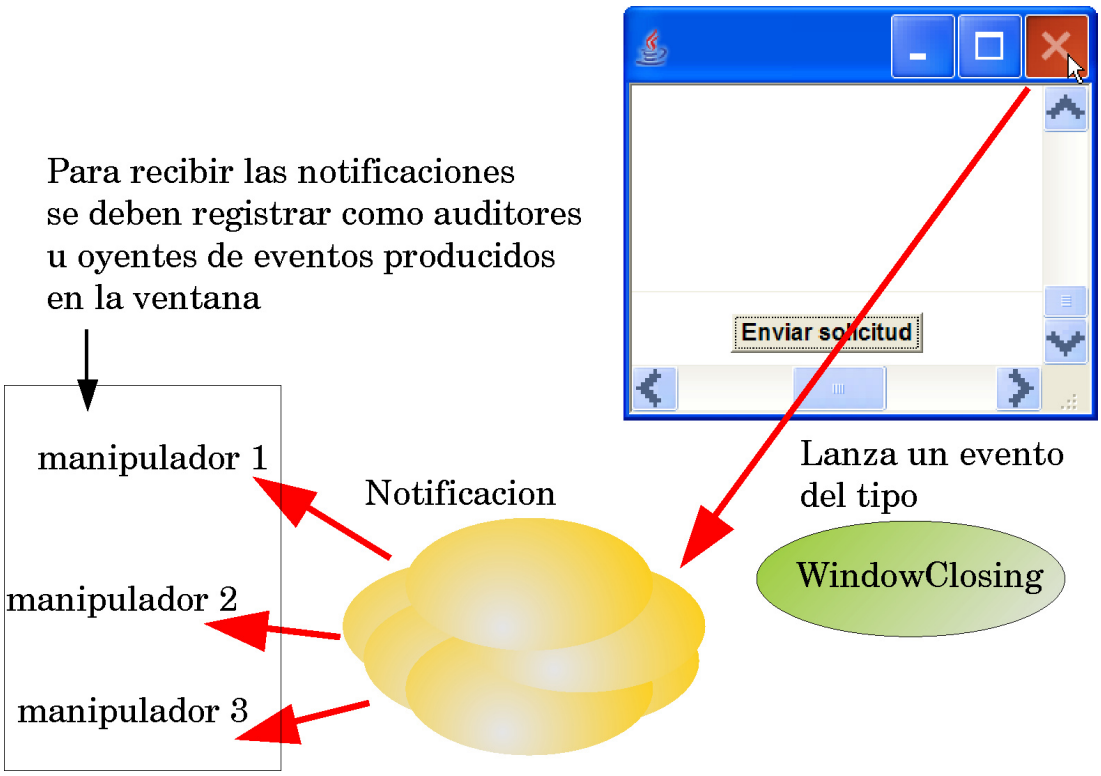
Gráficamente:



La cola de eventos es un lugar donde el productor (o fuente) del evento puede depositar la información de que a ocurrido un evento.

Un despachador de eventos comprueba la cola y determina a quién debe llamar.

- Si el productor y el manipulador son dos objetos diferentes, se necesita un mecanismo mediante el cual se asocie el evento (y los objetos en los que ocurren) con aquellos objetos interesados en tratar esos eventos.
- A estos objetos interesados se les conoce como **auditores** u **oyentes** (*listeners*).
- A un sistema en el cual un objeto escucha eventos que ocurren en otro objeto (un componente de la GUI) se le denomina **delegación de eventos**..
- Java resuelve el problema de ¿a quién notifico que se ha producido un evento? mediante un registro de oyentes.



En las dos secciones siguientes volveremos a tratar de forma específica los eventos para AWT y para Swing.

Se puede pensar en esto como si fuese una suscripción a un servicio de información vía correo electrónico (por ejemplo para recibir información sobre Java). Al realizar la suscripción se seleccionan una serie de temas en las que uno está interesado. Esto sería equivalente a registrar el interés en la cola de eventos. El servidor mantiene una lista de suscritos junto con sus temas de interés. Estos son los oyentes registrados. Cuando entra un nuevo artículo, es añadido a la lista de artículos a enviar (lo cual sería equivalente a poner el evento en la cola de eventos). El servidor enviaría a los suscritos (teniendo en cuenta sus temas de interés) los artículos por orden de llegada (lo cual es equivalente a despachar los eventos a los manipuladores).

Veremos que los componentes tienen métodos para registrar objetos interesados en tratar los eventos que se producen en aquellos.

En las dos secciones siguientes volveremos a tratar de forma específica los eventos para AWT y para Swing.

2. AWT

El paquete java.awt contiene cuatro grupos de clases que son útiles para la construcción de interfaces gráficas de usuario.

- La primera es la clase java.awt.Component y sus subclases. Definen elementos visibles como ventanas o botones.
- La segunda es la clase java.awt.Graphics que es necesaria para dibujar.
- El tercer grupo son las clases de eventos: java.awt.AWTEvent y sus subclases en el paquete java.awt.event.
- El cuarto grupo son las clases que sirven para la organización de los componentes en la ventana y dentro de otros componentes. Estas clases implementan a la interfaz java.awt.LayoutManager.

Componentes

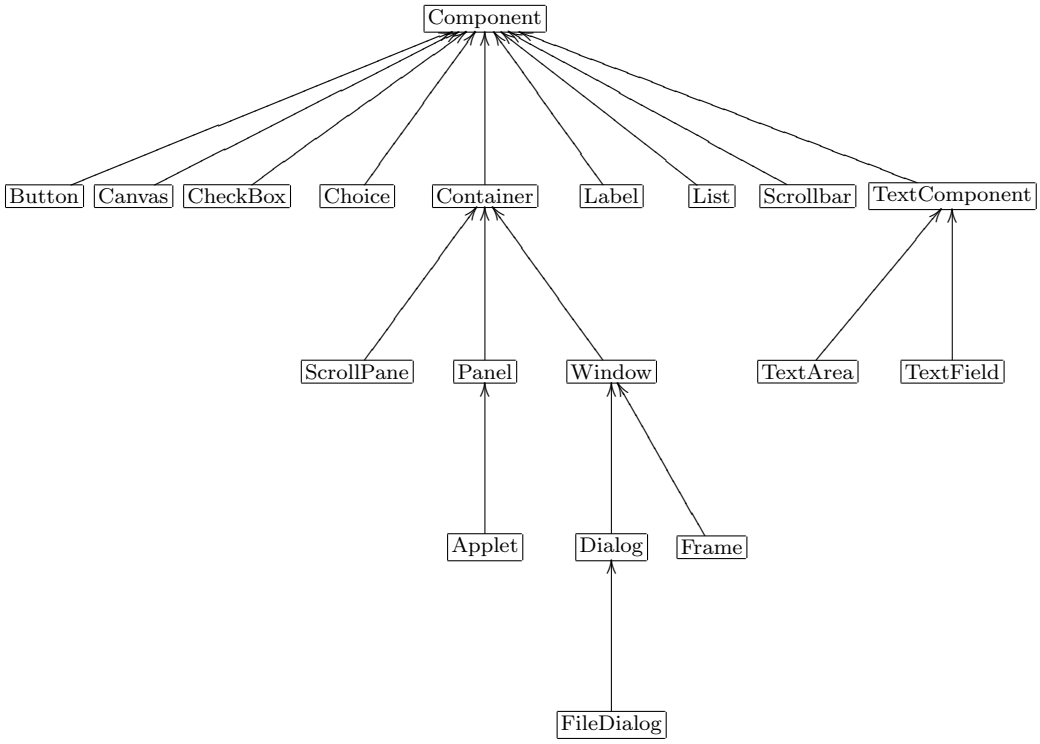
Un componente es un objeto que tiene una representación gráfica y que puede ser mostrado por pantalla y que puede utilizado por el usuario. Ejemplos de componentes son: Button, TextField, ScrollPane, TextArea, Canvas,...

Utilizan como base la clase java.awt.Component que está definida como abstracta. Todos los componentes (excepto los menús) extienden a esta clase.

Un conjunto de componentes está formado por widgets [4].

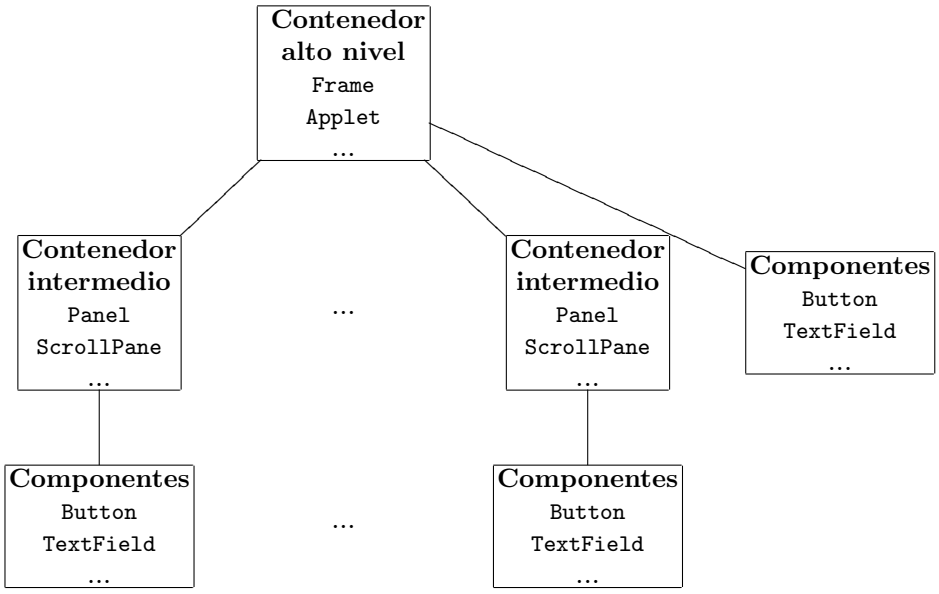
Otro conjunto está formado por contenedores. Estos componentes extienden a la clase java.awt.Container (que es una clase abstracta que extiende a Component). Los contenedores son componentes que pueden incluir otros componentes.

Las clases que se ofrecen en AWT que extienden a Component son:



[4] Contracción de Window y gadget. Una representación visible de un componente que puede ser manipulada por el usuario. Botones, campos de texto y barras de desplazamiento son ejemplos de *widgets*

La siguiente figura muestra la relación entre componentes y contenedores



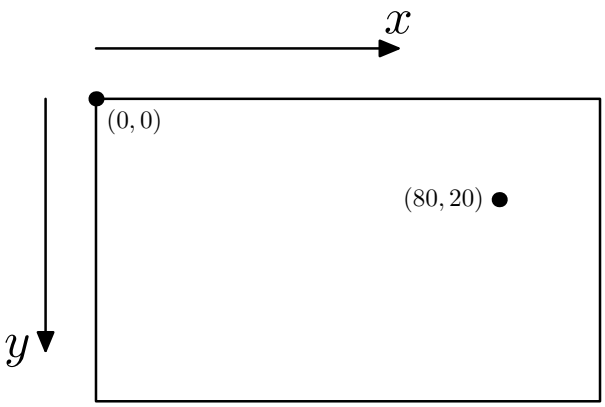
Ejemplo

Graphics

Un objeto del tipo java.awt.Graphics (llamado contexto gráfico) es el encargado pintar lo que se debe mostrar por pantalla.

El objeto Graphics utiliza un sistema de coordenadas para determinar las posiciones.

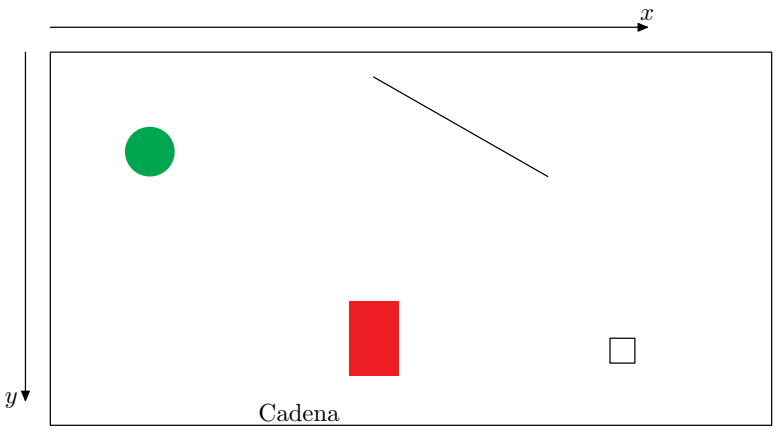
El origen de este sistema de coordenadas $(0, 0)$ es la esquina superior izquierda.



La clase Graphics tiene métodos del tipo drawLine, fillOval, drawArc, drawPoligon o setColor para realizar dibujos.

```
g.setColor(Color.green);
g.filloval(30,30,20,20);

g.setColor(Color.red);
g.fillRect(100,100,20,30);
```

```
g.setColor(Color.black);
g.drawLine(130,10,200,50);

g.drawRect(220,120,10,10);

g.drawString("Cadena",100,145);
```



Una de las cosas que debe realizar un componente que se hace visible en una GUI es pintarse.

Si un componente queda parcialmente cubierto por otra ventana y esta ventana es movida de forma que deja al descubierto al componente, éste debe mostrarse.

Java determina cuando deben mostrarse los componentes (introduciendo un evento de pintado en la cola de eventos) y pide al componente que se pinte.

Todos los componentes tienen un método paint(·) que funciona dirigido por eventos. Este método paint(·) no determina cuando se debe pintar el componente sino cómo se debe pintar cuando tenga que hacerlo.

El método paint(·) recibe un objeto del tipo Graphics. Este objeto contiene entre otras cosas, el sistema de coordenadas dentro del cual se encuentra el componente.

Cada componente especifica en este método cómo se muestra dentro de la GUI.

Si se desea cambiar la forma en la que aparecerá un componente en la pantalla lo que hay que hacer es sobreescribir este método y utilizar el objeto Graphics que recibe para dibujar.

Una de las implicaciones que tiene el hecho de que el método paint(·) sea llamado en un entorno dirigido por eventos es que no se debe llamar directamente a éste método.

Entonces... ¿qué hay que hacer si quiero que un determinado componente se redibuje?

Este puede ser el caso en el que tiene un programa que muestra objetos en movimiento en la pantalla, y se desea que estos objetos actualicen su movimiento cada cierto número de milisegundos.

Esto se puede conseguir realizando una llamada al método repaint() que está definido en java.awt.Component y que por tanto heredan todos los componentes.

La llamada a este método pone en la cola de eventos un evento de petición de repintado.

En resumen:

- *Cuando se desee modificar la forma en la que aparece un componente se sobreescribe el método paint(Graphics g) y se utiliza el objeto de tipo Graphics (que se le pasa como argumento) para dibujar*
- *Cuando deseemos que un componente se redibuje se envía un mensaje a repaint() que encola un evento de redibujado.*

Eventos

- Como vimos en la primera sección, el modelo de delegación de eventos se basa en que los componentes disparan eventos que pueden ser tratados por auditores u oyentes.
- Los auditores u oyentes se registran en un componente llamando a alguno de sus métodos del tipo addXXXListener(XXXListener). Donde XXXListener es una referencia a una interfaz en la que se especifican los métodos que debe tener el manipulador de los eventos. De este modo hay libertad para especificar la implementación.
- Una vez que el escuchador ha sido añadido al componente, cuando se produzca un evento, los métodos apropiados del manipulador (que han sido especificados en la interfaz) serán llamados.

La clase EventObject del paquete java.util es la clase padre de todos los eventos.

Su constructor recibe una referencia al objeto que genera el evento.

Esta clase tiene dos métodos: getSource() que devuelve el objeto que generó el evento y toString().

Los paquetes relacionados con los eventos en AWT son java.awt.event.

La clase abstracta AWTEvent definida en el paquete java.awt es una subclase de EventObject.

Es la superclase de todos los eventos basados en AWT utilizados por el modelo de delegación de eventos.

Hay dos clases de eventos:

- **Eventos de componente o de bajo nivel** ocurren cuando ocurre algo específico en un componente. Por ejemplo al moverse, entrar o salir el ratón sobre un componente, al ganar o perder la atención,...
- **Eventos semánticos** no son tan específicos como los anteriores y no son disparados necesariamente por una acción atómica tal y como una pulsación del ratón. Las acciones que disparan estos eventos depende del objeto: por ejemplo en una lista se disparan cuando sus elementos son pulsados dos veces, en un campo de texto se disparan cuando se pulsa la tecla *enter*.

Los eventos de componente o de bajo nivel son:

ComponentEvent, ContainerEvent, FocusEvent, InputEvent, KeyEvent, MouseEvent, MouseWheelEvent y WindowEvent

Los eventos semánticos son:

ActionEvent ,AdjustmentEvent , ItemEvent, TextEvent

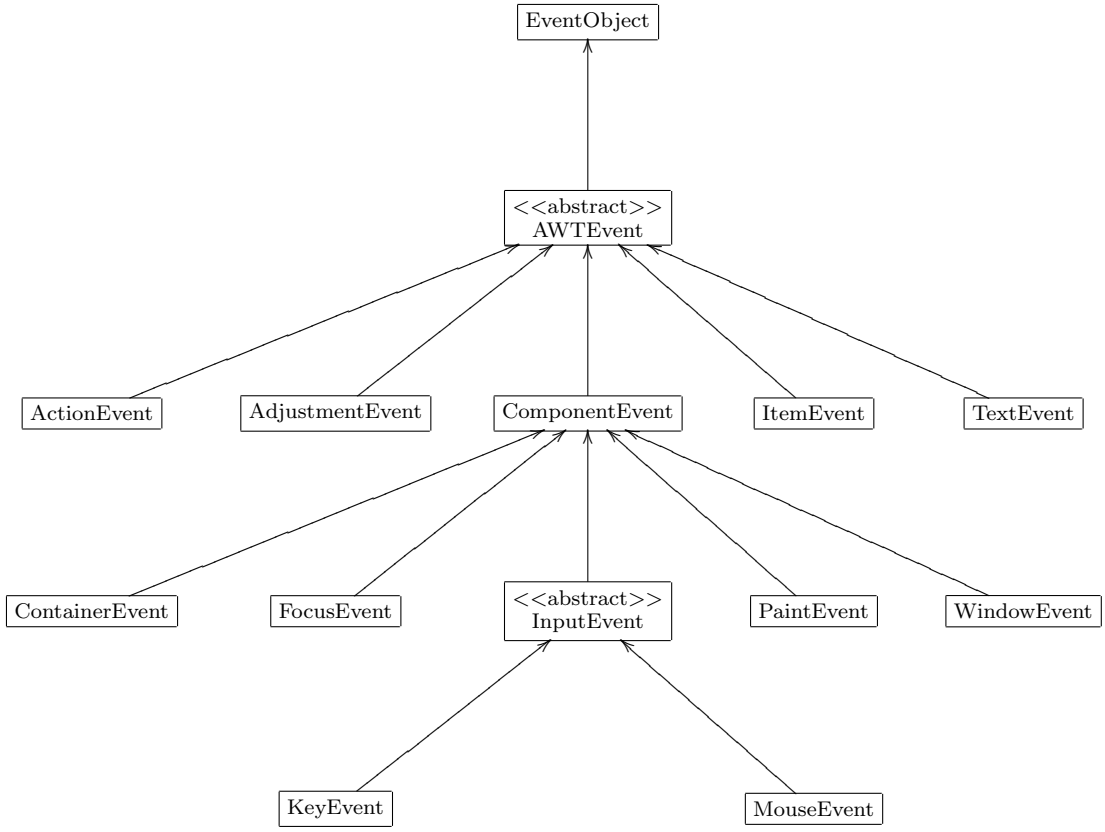
A continuación se muestran los eventos que se definen en AWT y cual es la acción que los produce

Eventos AWT	**Descripción**
`ActionEvent`	Se genera cuando el usuario pulsa un botón, pulsa *Return* en un campo de texto, selecciona un elemento de un menú o cuando un elemento de una lista se pulsado 2 veces.
`AdjustmentEvent`	Se genera cuando se manipula una barra de desplazamiento.
`ItemEvent`	Evento que indica que un elemento de una lista se ha seleccionado o ha dejado de estar seleccionado. Los siguientes componentes generan eventos de este tipo: `CheckBox, CheckBoxMenuItem, Choice, List`.
`TextEvent`	Se genera cuando se cambia el valor de un área de texto o de un campo de texto. Los objetos fuente de este evento son: `TextField` y `TextArea`.

Eventos AWT	**Descripción**
`ComponentEvent`	Un evento que indica que un componente ha sido movido, ha cambiado de tamaño o ha sido ocultado. AWT maneja este evento (es decir que aunque explícitamente tratemos este evento, AWT también lo hará).
`ContainerEvent`	Se genera cuando se añade o se elimina un componente de un contenedor. AWT trata este evento.
`FocusEvent`	Se genera cuando un componente gana o pierde la atención. Un componente tiene la atención al pulsar sobre él con el ratón o por que se ha llegado a él pulsando la tecla de tabulación. El componente que tiene la atención recibe los eventos de teclado.

Eventos AWT	**Descripción**
`KeyEvent`	Es una subclase de `InputEvent`. Se genera cuando se pulsa una tecla o libera una tecla.
`MouseEvent`	Es una subclase de `InputEvent`. Se genera cuando el ratón se mueve, se pulsa, se arrastra, o cuando entra o sale el ratón de un componente.
`MouseWheelEvent`	Un evento que indica que la rueda del ratón se ha movido en un componente.
`WindowEvent`	Se genera cuando una ventana se activa, se desactiva, se cierra, se minimiza se maximiza o se sale de ella.

La jerarquía de estas clases se muestra en el siguiente diagrama:



Para tratar eventos será necesario:

- Disponer de una clase que implemente a la interfaz apropiada según el tipo de evento que deseamos tratar. Estas interfaces definen métodos acordes con el tipo de evento (por ejemplo la interfaz relacionada con el ratón define métodos que son llamados cuando se pulsa un botón).

- Pasar una instancia de esta clase al componente fuente de los eventos a tratar. Los componentes definen métodos para registrar auditores según el tipo de evento en el que están interesados.

Vamos a ver ahora algunas de las interfaces que se ofrecen en `java.awt` con el fin de especificar los métodos que deben poseer los objetos oyentes o auditores para cada uno de los eventos.

```
public interface ComponentListener extends EventListener
```

Esta interfaz la deben implementar aquellas clases que estén interesadas en escuchar eventos del tipo ComponentEvent.

El objeto de este tipo debe registrarse en un componente utilizando su método addComponentListener.

Los métodos que define esta interfaz son:

```
//Metodo llamado cuando se oculta un componente
void componentHidden(ComponentEvent e)
```

```
//Metodo llamado cuando la posicion de un componente cambia
void componentMoved(ComponentEvent e)

//Metodo llamado cuando el tamaño de un componente cambia
void componentResized(ComponentEvent e)

//Metodo llamado cuando se muestra un componente
void componentShown(ComponentEvent e)
```

AWT maneja estos eventos de forma que la GUI funciona correctamente cuando la ventana se mueve o se modifica su tamaño.

```
public interface ContainerListener extends EventListener
```

Esta interfaz la deben implementar aquellas clases que estén interesadas en escuchar eventos del tipo ContainerEvent.

El objeto de este tipo debe registrarse en un componente utilizando su método addContainerListener. Los métodos que define esta interfaz son:

```
//Metodo llamado cuando un componente es añadido al contenedor
void componentAdded(ContainerEvent e)

//Metodo llamado cuando un componente es quitado del contenedor
void componentRemoved(ContainerEvent e)
```

AWT maneja estos eventos de forma que la GUI funciona correctamente cuando a un contenedor se añade o retira un componente.

```
public interface FocusListener extends EventListener
```

Esta interfaz la deben implementar aquellas clases que estén interesadas en escuchar eventos del tipo FocusEvent

El objeto de este tipo debe registrarse en un componente utilizando su método addFocusListener. Los métodos que define esta interfaz son:

```
//Metodo llamado cuando un componente gana la atención del teclado
void focusGained(FocusEvent e)

//Metodo llamado cuando un componente pierde la atención del teclado
void focusLost(FocusEvent e)
```

```
public interface KeyListener extends EventListener
```

Esta interfaz la deben implementar aquellas clases que estén interesadas en escuchar eventos del tipo KeyEvent

El objeto de esta clase debe registrarse en un componente utilizando su método addKeyListener.

Los métodos que define esta interfaz son:

```
//Metodo llamado cuando se pulsa una tecla
void keyPressed(KeyEvent e)

//Metodo llamado cuando se libera una tecla
```

```
    void keyReleased(KeyEvent e)

    //Metodo llamado cuando se pulsa y libera una tecla
    void keyTyped(KeyEvent e)
```

```
public interface MouseListener extends EventListener
```

Esta interfaz la deben implementar aquellas clases que estén interesadas en escuchar eventos del tipo MouseEvent

El objeto de este tipo debe registrarse en un componente utilizando su método addMouseListener.

Los métodos que define esta interfaz son:

```
    // Metodo llamado cuando se pulsa y libera un boton del raton
    // sobre un componente
    void mouseClicked(MouseEvent e)

    // Metodo llamado cuando el raton entra en un componente
    void mouseEntered(MouseEvent e)

    // Metodo llamado cuando el raton sale de un componente
    void mouseExited(MouseEvent e)

    // Metodo llamado al pulsar un boton del raton sobre un componente
    void mousePressed(MouseEvent e)

    // Metodo llamado al liberar un boton del raton sobre un componente
    void mouseReleased(MouseEvent e)
```

```
public interface WindowListener extends EventListener
```

Esta interfaz la deben implementar aquellas clases que estén interesadas en escuchar eventos del tipo WindowEvent

El objeto de este tipo debe registrarse en un componente utilizando su método addWindowListener.

Los métodos que define esta interfaz son:

```
    // Metodo llamado cuando se activa la ventana
    void windowActivated(WindowEvent e)

    // Metodo llamado cuando se ha cerrado la ventana
    void windowClosed(WindowEvent e)

    // Metodo llamado cuando el usuario cierra la ventana
    void windowClosing(WindowEvent e)

    // Metodo llamado cuando la ventana deja de estar activa
    void windowDeactivated(WindowEvent e)

    // Metodo llamado cuando la ventana pasa de icono a su estado normal
    void windowDeiconified(WindowEvent e)

    // Metodo llamado cuando se iconifica la ventana
    void windowIconified(WindowEvent e)

    // Metodo llamado la primera vez que se muestra la ventana
    void windowOpened(WindowEvent e)
```

```
public interface ActionListener extends EventListener
```

Esta interfaz la deben implementar aquellas clases que estén interesadas en escuchar eventos del tipo ActionEvent

El objeto de este tipo debe registrarse en un componente utilizando su método addActionListener.

Los métodos que define esta interfaz son:

```
// Metodo llamado cuando ocurre una accion
void actionPerformed(ActionEvent e)
```

```
public interface TextListener extends EventListener
```

Esta interfaz la deben implementar aquellas clases que estén interesadas en escuchar eventos del tipo TextEvent

El objeto de este tipo debe registrarse en un componente utilizando su método addTextListener.

Los métodos que define esta interfaz son:

```
// Metodo llamado cuando el texto ha sido modificado
void textValueChanged(TextEvent e)
```

```
public interface ItemListener extends EventListener
```

Esta interfaz la deben implementar aquellas clases que estén interesadas en escuchar eventos del tipo ItemEvent

El objeto de este tipo debe registrarse en un componente utilizando su método addItemListener.

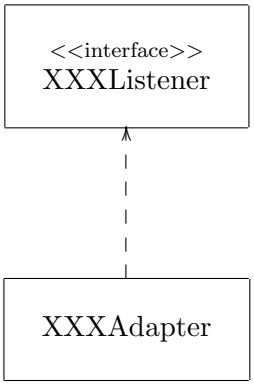
Los métodos que define esta interfaz son:

```
// Metodo llamado cuando el texto ha sido modificado
void itemStateChanged(ItemEvent e)
```

Hay ocasiones en las que puede resultar incómodo trabajar con estas interfaces si uno está interesado en implementar un único método, ya que hay que escribir el resto de los métodos dejándolos vacíos.

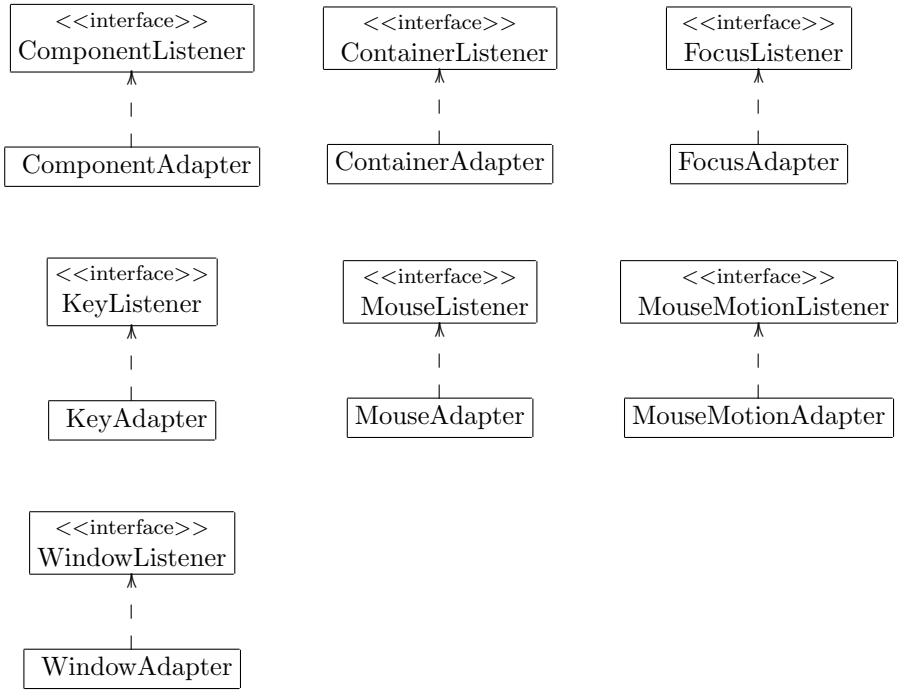
Para algunas de estas interfaces se ofrecen clases que las implementan pero dejando vacíos todos los métodos.

De esta forma, puede resultar más cómodo extender estas clases que implementar las interfaces.



donde XXX es el nombre del evento.

Las clases adaptadoras que se ofrecen son:



El código de la derecha y el de la izquierda son equivalentes:

```
Button b = new Button("Aceptar");

class Oyente implementes MouseListener{

    public void mouseClicked(MouseEvent e) {}

    public void mouseEntered(MouseEvent e) {}

    public void mouseExited(MouseEvent e) {}

    public void mousePressed(MouseEvent e) {
      System.out.println("Boton pulsado");
    }

    public void mouseReleased(MouseEvent e) {}
}

b.addMouseListener(new Oyente());
```

```
Button b = new Button("Aceptar");

class Oyente extends MouseAdapter{

    public void mousePressed(MouseEvent e) {
      System.out.println("Boton pulsado");
    }
}

b.addMouseListener(new Oyente());
```

La única diferencia a nivel de código es que en el caso de la derecha la clase implementa a la interfaz MouseListener y en el de la derecha la clase extiende a la clase MouseAdapter que a su vez implementa a la interfaz MouseListener.

Cuando una clase oyente de eventos se va a utilizar en un punto muy específico del código y no se va a reutilizar en otras partes (o en otras clases) existe la posibilidad de realizarla mediante una **clase anónima**.

Una clase anónima (como cabe esperar) es aquella a la que no se asigna un nombre.

La creación del objeto y la especificación de la clase se realiza en el mismo momento, el este caso en el argumento de un método.

Vamos a ver cómo se puede utilizar una clase anónima con el ejemplo anterior:

```
Button b = new Button("Aceptar");

b.addMouseListener(new MouseAdapter(){
    public void mousePressed(MouseEvent e){
      System.out.println("Boton pulsado");
    }
}
```

Organización

AWT soporta diversos esquemas de organización (*layout*) que pueden ser utilizados para organizar los componentes dentro de los contenedores.

Los gestores de organización se encargan de reorganizar los componentes en caso de que el usuario cambie el tamaño de la ventana.

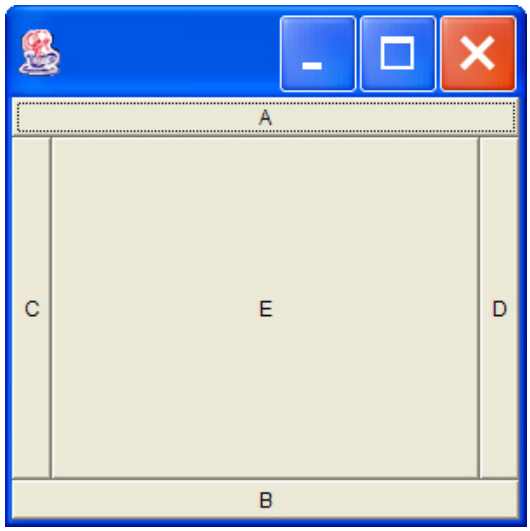
Los gestores de organización que ofrece Java son:

BorderLayout, FlowLayout, BoxLayout, CardLayout, GridLayout, GridBagLayout

El procedimiento es siempre el mismo, se crea un objeto de alguna de estas clases y se le indica al contenedor que organice los componentes utilizando el objeto (para ello los contenedores disponen del método setLayout(LayoutManager m).

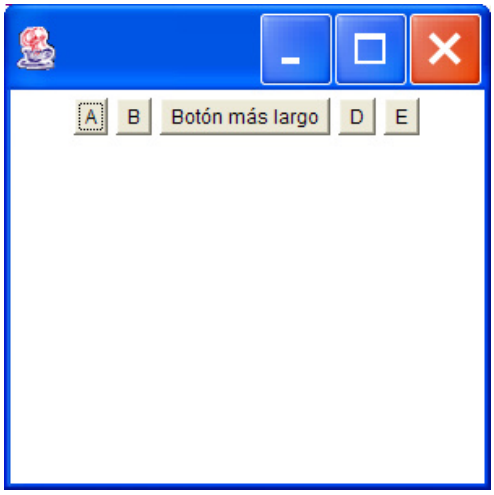
BorderLayout se puede utilizar para colocar en un contenedor cinco componentes como máximo ya que proporciona cinco posiciones donde colocar los componentes, estas son: NORTH (arriba), SOUTH (abajo), WEST (izquierda), EAST (derecha) y CENTER (en el centro).

```
import java.awt.*;

class Ventana extends Frame{
    public Ventana(){
      Button b1 = new Button("A");
      Button b2 = new Button("B");
      Button b3 = new Button("C");
      Button b4 = new Button("D");
      Button b5 = new Button("E");

      setLayout(new BorderLayout());
      add(b1,BorderLayout.NORTH);
      add(b2,BorderLayout.SOUTH);
      add(b3,BorderLayout.WEST);
      add(b4,BorderLayout.EAST);
      add(b5,BorderLayout.CENTER);
    }

    public static void main(String[] args){
      Ventana v = new Ventana();
      v.setSize(300,300);
      v.show();
    }
}
```



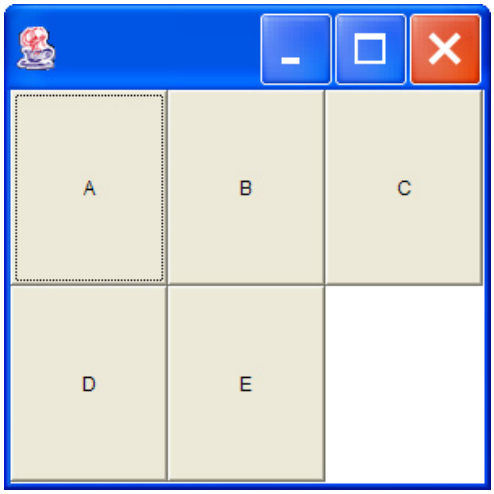
FlowLayout: coloca los componentes de izquierda a derecha conforme se van añadiendo a la ventana. El tamaño de los componentes se ajusta a su contenido.

```
import java.awt.*;

class Ventana extends Frame{
   public Ventana(){
     Button b1 = new Button("A");
     Button b2 = new Button("B");
     Button b3 = new Button("Botón más largo");
     Button b4 = new Button("D");
     Button b5 = new Button("E");

     setLayout(new FlowLayout());
     add(b1);
     add(b2);
     add(b3);
     add(b4);
     add(b5);
   }

   public static void main(String[] args){
     Ventana v = new Ventana();
     v.setSize(300,300);
     v.show();
   }
}
```



GridLayout: coloca los componentes del mismo tamaño y los coloca en filas y columnas en función de los valores pasados al constructor.

```
import java.awt.*;

class Ventana extends Frame{
   public Ventana(){
      Button b1 = new Button("A");
      Button b2 = new Button("B");
      Button b3 = new Button("C");
      Button b4 = new Button("D");
      Button b5 = new Button("E");

      setLayout(new GridLayout(2,3));

      add(b1);
      add(b2);
      add(b3);
      add(b4);
      add(b5);
   }

   public static void main(String[] args){
      Ventana v = new Ventana();
      v.setSize(300,300);
      v.show();
   }
}
```



CardLayout: se puede utilizar para mostrar de forma condicional unos elementos u otros, de forma que se puede controlar qué elementos serán visibles. Un ejemplo es una pila de cartas en las que sólo la superior es visible en un instante dado.

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

public class Ventana extends Frame{
   private CardLayout cardLayout;
   private Panel cardPanel;
   private Button b1;
   private Button b2;

   Ventana(){
      // El panel que va a contener tres paneles
      // organizados como CardLayout
```

```
        cardPanel = new Panel();

        cardLayout = new CardLayout();
        cardPanel.setLayout(cardLayout);

        // El primero de los paneles
        Panel botonesPanel = new Panel();
        botonesPanel.add(new Button("Boton 1"));
        botonesPanel.add(new Button("Boton 2"));
        botonesPanel.add(new Button("Boton 3"));

        // El segundo de los paneles
        Panel camposTextoPanel = new Panel();
        camposTextoPanel.add(new TextField(10));
        camposTextoPanel.add(new TextField(10));

        // El panel que se mostrara al inicio
        Panel inicioPanel = new Panel();
        inicioPanel.setBackground(new Color(.85f,.85f,.85f));
        inicioPanel.add(new Label("Ejemplo de CardLayout"));

        // Añadimos los tres paneles asignandoles un nombre
        cardPanel.add(inicioPanel,"inicial");
        cardPanel.add(botonesPanel,"botones");
        cardPanel.add(camposTextoPanel,"texto");

        // Panel de control desde donde se selecciona el subpanel a mostrar
        Panel controlPanel = new Panel();
        b1 = new Button("Botones");
        b2 = new Button("Campos de Texto");

        // Si se pulsa b1 mostramos el panel con nombre "botones"
        b1.addMouseListener(new MouseAdapter(){
            public void mousePressed(MouseEvent e){
                cardLayout.show(cardPanel,"botones");
            }
        });

        // Si se pulsa b2 mostramos el panel con nombre "texto"
        b2.addMouseListener(new MouseAdapter(){
            public void mousePressed(MouseEvent e){
                cardLayout.show(cardPanel,"texto");
            }
        });

        controlPanel.add(b1);
        controlPanel.add(b2);

        // Organizacion en la ventana
        setLayout(new BorderLayout());
        add(controlPanel,BorderLayout.NORTH);
        add(cardPanel,BorderLayout.CENTER);
    }
    public static void main(String[] args){
        Ventana v = new Ventana();
        v.setSize(300,300);
        v.show();
    }
}
```

GridBagLayout: Este es un organizador complejo que permite ajustar la posición de los componentes. Al colocar los componentes en los contenedores se especifican las restricciones que se deben cumplir (posición del componente, anchura y altura del componente, separación entre los componentes, posición dentro del espacio que ocupa, ...).

3. Swing

3.1. ¿Por qué Swing?

- AWT utiliza objetos que son dependientes de la plataforma, una aplicación ejecutada en Windows tendrá una apariencia similar al resto de las aplicaciones y si se ejecuta en Unix tendrá una apariencia similar a una aplicación XMotif.

- El problema es que el conjunto de componentes debe ser común a todas ellas (puesto que se basa en código nativo, no se puede proporcionar un componente que no esté en alguno de los sistemas ya que se rompería la portabilidad).

- Por ejemplo la clase Button de AWT debe tener un equivalente en el sistema de ventanas local (por ello se les llama componentes *heavyweight*). Puesto que los componentes equivalentes son específicos del sistema operativo el comportamiento y el aspecto visual de éstos no se podía modificar.

Estas consideraciones dieron lugar al diseño de las clases que forman el paquete `javax.swing`. Los objetivos fueron:

- Las clases Swing no utilizan código nativo para su implementación, están desarrolladas completamente en Java.

- Las clases Swing soportan Pluggable Look and Feel (PLAF), que permite a los desarrolladores de GUIs elegir la apariencia y el comportamiento de los componentes.

- Las clases Swing están implementadas utilizando una modificación del patrón clásico Model-View-Controller de diseño de GUIs. El View y el Controller se han combinado para formar clases UI.

- Las clases Swing vienen con más características y componentes que AWT, por lo que la elección a la hora de realizar una GUI suele ser Swing.

3.2. Características de los componentes Swing

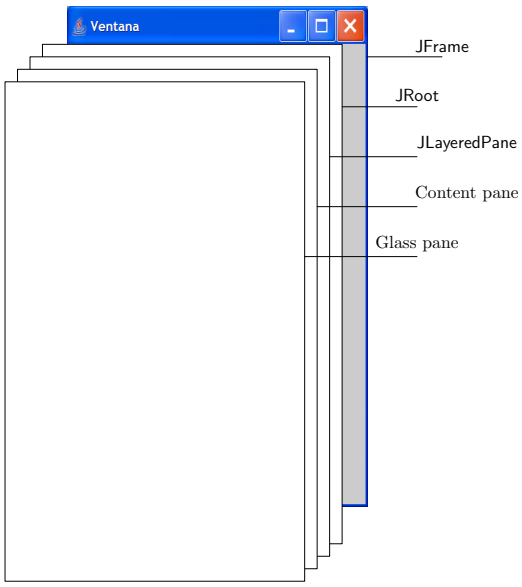
1. La jerarquía de Swing

Swing, al igual que AWT, propone una jerarquía de contenedores de componentes. Cualquier aplicación que utilice Swing debe tener un contenedor de alto nivel y uno o más contenedores intermedios y componentes atómicos.

Los contenedores de alto nivel más comúnmente utilizados son: JFrame, JApplet y JDialog.

Las ventanas son contenedores de otros componentes tales como barras de menú, campos de texto, botones, etc.

De hecho una ventana está constituida por una serie de capas:



Para añadir componentes a la ventana, primero se debe obtener el contenedor content pane y a continuación añadir los componentes a éste.

Los contenedores intermedios son aquellos que agrupan componentes atómicos, los comúnmente utilizados son: JPanel, JScrollPane, JSplitPane, JTabbedPane y JToolBar.

Los componentes atómicos son componentes que implementan una tarea GUI bien definida, Estos componentes son la parte utilizable de la GUI. Swing contiene una lista considerable de componentes atómicos para implementar GUIs. Por citar algunos JButton, JTextField, JList, JComboBox, JRadioButton, JCheckBox, JColorChooser, JFileChooser, ...

2. Gestores de organización

Los gestores de organización se utilizan para organizar los contenedores y los componentes atómicos dentro de los contenedores en los que están incluidos.

Swing utiliza los gestores de organización definidos en AWT que en general son apropiados y añade otros (como por ejemplo BoxLayout).

3. Manejo de eventos

Swing utiliza también el modelo de delegación de eventos comentado en la sección anterior (AWT).

El paquete javax.swing.event define eventos adicionales a los definidos por AWT.

4. Pintado

El mecanismo de pintado en Swing se maneja del mismo modo que en AWT: se comienza por los componentes de más alto nivel y se termina en los componentes atómicos.

El pintado es más eficiente ya que se emplea doble buffer de forma que el pintado de los componentes se realiza fuera de la pantalla y a continuación es enviado a la pantalla.

5. El patrón Model-View-Controller

Cada componente tiene tres características:

- El contenido o estado. En un botón el contenido almacenaría si el botón está pulsado o no, si está activo o no,... en un campo de texto el contenido almacenaría el texto.
- Su apariencia visual, que depende del estado (por ejemplo, un botón se muestra de forma diferente en función de si está pulsado o no).
- Su comportamiento, que está determinado por la forma en que responde ante eventos.

El diseño de los componentes Swing está basado en un patrón de diseño: el *Model-View-Controller*.

La motivación bajo este patrón es la siguiente: no hacer que un único objeto sea responsable de el contenido, la apariencia y comportamiento.

Este patrón de diseño recomienda realizar 3 clases:

- El *model* que se encarga del contenido o estado.
- El *view* que muestra el contenido.
- El *controller* que maneja la interacción con el usuario.

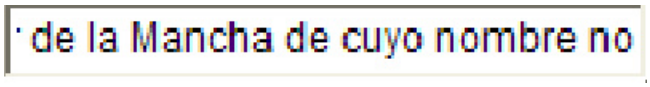
El patrón especifica de forma precisa cómo interaccionan estos objetos.

El *model* almacena el contenido y no tiene interfaz con el usuario.

El *view* muestra el contenido.

Pensemos en un campo de texto en el que hemos escrito la frase: *En un lugar de la Mancha, de cuyo nombre no.*

El campo de texto puede tener un tamaño (en cuanto al número de caracteres que puede mostrar) menor que la frase y en un momento dado puede mostrar lo siguiente:



El *model* debe ofrecer métodos para modificar y obtener el contenido.

Una ventaja del patrón *Model-View-Controller* es que es posible tener varios *view* de un mismo *model*.

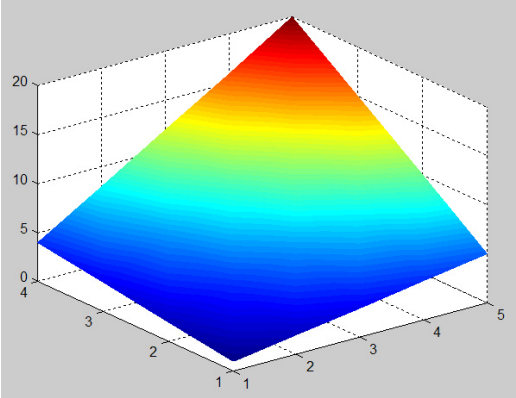
Por ejemplo, podríamos realizar un componente cuyo contenido sean 3 matrices: una almacena las coordenadas x, otra las coordenadas y y otra el valor de una función $z = f(x, y)$.

$$\text{Model:} \quad x = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix} \quad z = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \end{bmatrix} \quad y = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 & 10 \\ 3 & 6 & 9 & 12 & 15 \\ 4 & 8 & 12 & 16 & 20 \end{bmatrix}$$

Un *view* podría mostrar los datos como una tabla:

	x=1	x=2	x=3	x=4	x=5
y=1	1	2	3	4	5
y=2	2	4	6	8	10
y=3	3	6	9	12	15
y=4	4	8	12	16	20

y **simultáneamente** otro *view* mostraría los datos como una superficie:



El *controller* trata los eventos relacionados con el usuario tal como pulsaciones del ratón o pulsaciones de teclas.

Tiene que decidir si estos eventos se deben trasladar en cambios en el *model* o cambios en el *view*.

Por ejemplo si el usuario modifica un dato de la tabla, el *controller* notifica al *model* y éste notifica el *view* para que se actualice.

Si por ejemplo ofrecemos la posibilidad de de que pulsando sobre la superficie se realice un zoom y el usuario pulsa, el *controller* debe notificar al *view* pero esta operación no ha modificado los datos por lo que el *model* no interviene en ningún momento.

La ventaja de esta separación es que se consigue mayor independencia entre el contenido y la apariencia.

- Un mismo *model* puede ser utilizado por diferentes *view*.
- Si se realiza una aplicación que utilice la voz en lugar de ratón y teclado únicamente habría que modificar el *controller*.

De esta forma se consigue reutilizar código.

Los patrones de diseño han de ser vistos como una guía, como una posible solución a un problema y no como algo dogmático (que hay que aplicar de una forma inmutable).

Los diseñadores de Swing encontraron que el *model* se podía separar siembre y cada componente Swing tiene asociado un *model*.

Sin embargo las responsabilidades del *view* y del *controller* no se pueden separar con la misma claridad y se agrupan en lo que se conoce como clases delegadas o clases UI.

Por lo tanto, cada componente Swing tiene un modelo y un delegado asociado con él, el modelo es responsable del estado y el delegado de la apariencia y comportamiento (*look and feel*) así como de la comunicación entre el modelo y lo que se muestra.

Los **modelos de tipo datos** contienen información introducida por el usuario tal y como el valor de una celda en una tabla o los elementos mostrados en la lista. Para componentes Swing centrados en los datos (como por ejemplo JTable y JTree se recomienda que se trabaje con el modelo de datos).

Los **modelos de tipo GUI** no estarían relacionados con datos introducidos por el usuario. Un ejemplo de este tipo de *model* sería el de un botón que contiene datos sobre el estado.

La siguiente tabla muestra los diferentes componentes Swing y su correspondiente *model*.

Componente	**Modelo**	**Tipo de modelo**
JButton	ButtonModel	GUI
JToggleButton	ButtonModel	GUI/datos
JCheckBox	ButtonModel	GUI/datos
JRadioButton	ButtonModel	GUI/datos
JMenu	ButtonModel	GUI
JMenuItem	ButtonModel	GUI
JCheckBoxMenuItem	ButtonModel	GUI/datos
JRadioButtonMenuItem	ButtonModel	GUI/datos
JComboBox	ComboBoxModel	datos
JProgressBar	BoundedRangeModel	GUI/datos
JScrollBar	BoundedRangeModel	GUI/datos
JSlider	BoundedRangeModel	GUI/datos
JTabbedPane	SingleSelectionModel	GUI
JList	ListModel	datos
JList	ListSelectionModel	GUI
JTable	TableModel	datos
JTable	TableColumnModel	GUI
JTree	TreeModel	datos
JTree	TreeSelectionModel	GUI
JEditorPane	Document	datos
JTextPane	Document	datos
JTextArea	Document	datos
JTextField	Document	datos
JPasswordField	Document	datos

Todos los componentes Swing ofrecen métodos para obtener el modelo o para establecer uno diferente.

Por ejemplo la clase JButton dispone de los métodos:

```
public ButtonModel getModel()
public void setModel(ButtonModel model)
```

y la clase JTextArea dispone de los métodos:

```
public Document getDocument()
public void setModel(Document model)
```

Y lo mismo para el resto de componentes.

El siguiente ejemplo muestra una ventana con cuatro componentes del tipo JTextArea. Se obtiene el modelo del primero de ellos y se le asigna a los otros tres (un único *model* y cuatro *view*).

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

class EjemploModel extends JFrame{

   private JTextArea jta1;
   private JTextArea jta2;
   private JTextArea jta3;
   private JTextArea jta4;
   private Document doc;

  EjemploModel(){
      Container c = getContentPane();

      c.setLayout(new FlowLayout());

      jta1 = new JTextArea(10,20);
      jta1.setFont(new Font("Arial",Font.PLAIN,18));
      JScrollPane jsp1 = new JScrollPane(jta1);

      doc = jta1.getDocument(); // Obtencion del model

      jta2 = new JTextArea(10,20);
      jta2.setFont(new Font("Arial",Font.PLAIN,18));
      JScrollPane jsp2 = new JScrollPane(jta2);
      jta2.setDocument(doc); // Asignacion del model

      jta3 = new JTextArea(10,20);
      jta3.setFont(new Font("Arial",Font.PLAIN,18));
      JScrollPane jsp3 = new JScrollPane(jta3);
      jta3.setDocument(doc); // Asignacion del model

      jta4 = new JTextArea(10,20);
      jta4.setFont(new Font("Arial",Font.PLAIN,18));
      JScrollPane jsp4 = new JScrollPane(jta4);
      jta4.setDocument(doc); // Asignacion del model

      c.add(jsp1);
      c.add(jsp2);
      c.add(jsp3);
      c.add(jsp4);

      setSize(700,700);
      setVisible(true);

   }

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

La siguiente tabla muestra los eventos, interfaces que deben implementar los auditores y el modelo en que se producen.

Modelo	**Interface**	**Evento**
BoundedRangeModel	ChangeListener	ChangeEvent
ButtonModel	ChangeListener	ChangeEvent
SingleSelectionModel	ChangeListener	ChangeEvent
ListModel	ListDataListener	ListDataEvent
ListSelectionModel	ListSelectionListener	ListSelectionEvent
ComboBoxModel	ListDataListener	ListDataEvent
TreeModel	TreeModelListener	TreeModelEvent
TreeSelectionModel	TreeSelectionListener	TreeSelectionEvent
TableModel	TableModelListener	TableModelEvent
TableColumnModel	TableColumnModelListener	TableColumnModelEvent
Document	DocumentListener	DocumentEvent
Document	UndoableEditListener	UndoableEditEvent

El *Look-and-feel* de los componentes se puede cambiar en tiempo de ejecución.

El siguiente código muestra cómo se puede modificar la apariencia de un botón cada vez que es pulsado.

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

public class LnF extends JFrame{

   private int lnf=0;

   public LnF(){
      Container c = getContentPane();
      c.setLayout(new FlowLayout());

      final JButton b = new JButton("MetalLookAndFeel");
      final ButtonModel bm = b.getModel();

      System.out.println("Listado de L&F instalados: ");
      UIManager.LookAndFeelInfo[] info = UIManager.getInstalledLookAndFeels();

      for (int i=0; i<info.length ; i++)
         System.out.println(info[i]);

      // Auditor de eventos para el ButtonModel
      bm.addChangeListener( new ChangeListener(){
         public void stateChanged(ChangeEvent e){
            try{
               if (bm.isPressed()){
                  switch (lnf) {
                     case 0:
                        UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.
                            MotifLookAndFeel");
                        b.setText("MotifLookAndFeel");
                        lnf = 1;
                        break;
                     case 1:
                        UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.
                            WindowsLookAndFeel");
                        b.setText("WindowsLookAndFeel");
                        lnf = 2;
                        break;
                     case 2:
                        UIManager.setLookAndFeel("javax.swing.plaf.metal.
                            MetalLookAndFeel");
                        b.setText("MetalLookAndFeel");
                        lnf = 0;
                        break;
                  }
                  b.updateUI();
```

```
            }
         }catch(Exception ev){
            System.out.println("Error cambiando el L&F");
         }
      }
    });
    c.add(b);

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

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

3.3. Un catálogo de contenedores y componentes Swing

JFrame

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

public class Ventana extends JFrame{
  public Ventana(){
    Container c = getContentPane();
    c.setLayout(new FlowLayout());

    c.add(new JLabel("Ejemplo de JFrame"));
    setSize(200,200);
    setVisible(true);
  }
  public static void main(String[] args){
    new Ventana();
  }
}
```

JPanel

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

public class Ventana extends JFrame{
  public Ventana(){
    Container c = getContentPane();
    c.setLayout(new FlowLayout());

    JPanel p = new JPanel();
    p.add(new JLabel("Ejemplo de JPanel"));

    c.add(p);

    setSize(200,200);
    setVisible(true);
  }
  public static void main(String[] args){
    new Ventana();
  }
}
```

ImageIcon

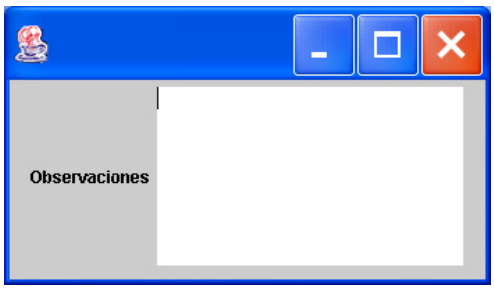
```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

public class Ventana extends JFrame{
   public Ventana(){
      Container c = getContentPane();
      c.setLayout(new FlowLayout());

      ImageIcon ii = new ImageIcon("telematica.jpg");

      c.add(new JLabel("",ii,JLabel.CENTER));

      setSize(350,300);
      setVisible(true);
   }
   public static void main(String[] args){
      new Ventana();
   }
}
```

JTextField

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

public class Ventana extends JFrame{

   public Ventana(){
      Container c = getContentPane();
      c.setLayout(new FlowLayout());

      JTextField campoTexto = new JTextField(20);

      c.add(new JLabel("Nombre"));
      c.add(campoTexto);

      setSize(350,200);
      setVisible(true);
   }
   public static void main(String[] args){
      new Ventana();
   }
}
```

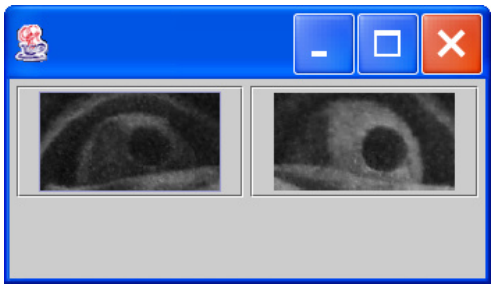
JTextArea

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

public class Ventana extends JFrame{

   public Ventana(){
      Container c = getContentPane();
      c.setLayout(new FlowLayout());

      JTextArea area = new JTextArea(8,20);

      c.add(new JLabel("Observaciones"));
      c.add(area);

      setSize(350,200);
      setVisible(true);
   }
   public static void main(String[] args){
      new Ventana();
   }
}
```



JButton

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

public class Ventana extends JFrame{

    public Ventana(){
       Container c = getContentPane();
       c.setLayout(new FlowLayout());

       ImageIcon i1 = new ImageIcon("ojo1.jpg");
       ImageIcon i2 = new ImageIcon("ojo2.jpg");

       JButton b1 = new JButton(i1);
       JButton b2 = new JButton(i2);

       c.add(b1);
       c.add(b2);

       setSize(350,200);
       setVisible(true);
    }
    public static void main(String[] args){
       new Ventana();
    }
}
```



JCheckBox

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

public class Ventana extends JFrame{

    public Ventana(){
       Container c = getContentPane();
       c.setLayout(new FlowLayout());

       JCheckBox cb = new JCheckBox("Pizarra");

       cb.setFont(new Font("Arial",Font.PLAIN,20));

       c.add(cb);

       setSize(200,200);
       setVisible(true);
    }
    public static void main(String[] args){
       new Ventana();
    }
}
```

JRadioButton

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

public class Ventana extends JFrame{

    public Ventana(){
        Container c = getContentPane();
        c.setLayout(new FlowLayout());

        JRadioButton rb = new JRadioButton("Pizarra");

        rb.setFont(new Font("Arial",Font.PLAIN,20));

        c.add(rb);

        setSize(200,200);
        setVisible(true);
    }
    public static void main(String[] args){
        new Ventana();
    }
}
```

JComboBox

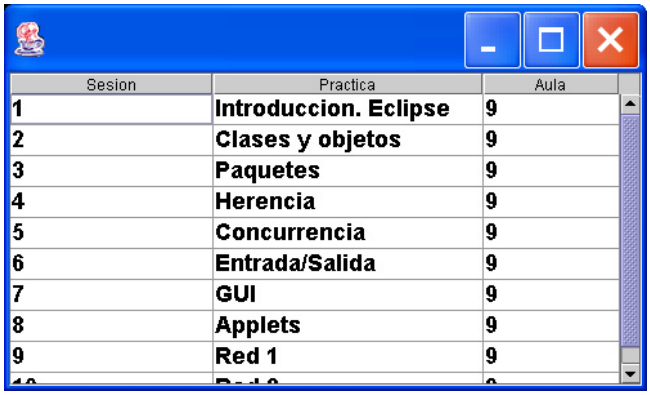
```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;

public class Ventana extends JFrame{

    public Ventana(){
        Container c = getContentPane();
        c.setLayout(new FlowLayout());

        JComboBox cb = new JComboBox();

        cb.setFont(new Font("Arial",Font.PLAIN,20));

        cb.addItem("Pizarra");
        cb.addItem("Pantalla");
        cb.addItem("Proyector");

        c.add(cb);

        setSize(200,200);
        setVisible(true);
    }
    public static void main(String[] args){
        new Ventana();
    }
}
```

JList

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

public class Ventana extends JFrame{

    public Ventana(){
       Container c = getContentPane();
       c.setLayout(new FlowLayout());

       String[] datos = {"Pizarra", "Pantalla", "Proyector"};
       JList lista = new JList(datos);

       lista.setFont(new Font("Arial",Font.PLAIN,20));

       c.add(lista);

       setSize(200,200);
       setVisible(true);
    }
    public static void main(String[] args){
       new Ventana();
    }
}
```

JTable

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

public class Ventana extends JFrame{
    public Ventana(){

       Container cp = getContentPane();
       cp.setLayout(new BorderLayout());

       // Nombres de las columnas
       final String[] nombreCol = {"Sesion","Practica","Aula"};

       // Datos
       Object[][] datos = {
        {"1","Introduccion. Eclipse","9"},
        {"2","Clases y objetos","9"},
        {"3","Paquetes","9"},
        {"4","Herencia","9"},
        {"5","Concurrencia","9"},
        {"6","Entrada/Salida","9"},
        {"7","GUI","9"},
        {"8","Applets","9"},
        {"9","Red 1","9"},
        {"10","Red 2","9"}};

       JTable tabla = new JTable(datos,nombreCol);

       tabla.setFont(new Font("Arial",Font.BOLD,18));
       tabla.setRowHeight(24);

       JScrollPane jsp = new JScrollPane(tabla); //,ver,hor);

       cp.add(jsp,BorderLayout.CENTER);

       setSize(500,300);
       setVisible(true);
    }

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

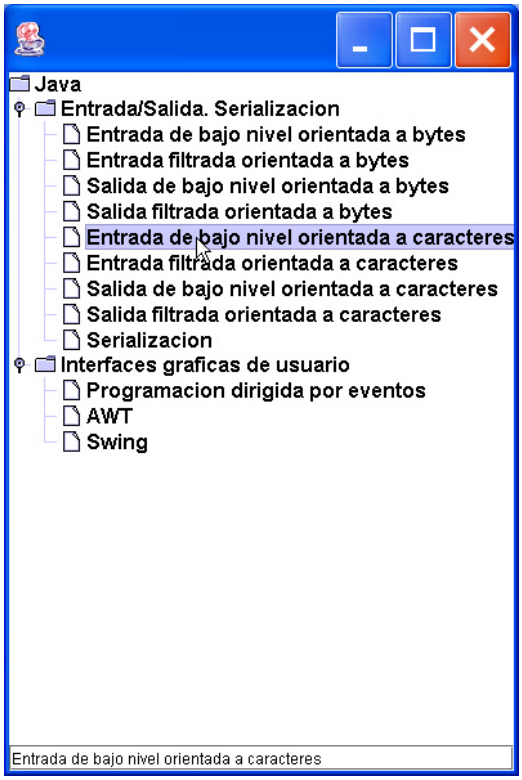


Sesion	Practica	Aula
1	Introduccion. Eclipse	9
2	Clases y objetos	9
3	Paquetes	9
4	Herencia	9
5	Concurrencia	9
6	Entrada/Salida	9
7	GUI	9
8	Applets	9
9	Red 1	9

JTree

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

public class Ventana extends JFrame{
    private JTree arbol;
    private JTextField campo = new JTextField();
    public Ventana(){


        Container c = getContentPane();
        c.setLayout(new BorderLayout());

        // Construccion del arbol

        DefaultMutableTreeNode asig = new DefaultMutableTreeNode("Java");

        DefaultMutableTreeNode tema = null;
        DefaultMutableTreeNode seccion = null;

        tema = new DefaultMutableTreeNode("Entrada/Salida. Serializacion");
        asig.add(tema);

        seccion = new DefaultMutableTreeNode("Entrada de bajo nivel orientada a bytes");
        tema.add(seccion);

        seccion = new DefaultMutableTreeNode("Entrada filtrada orientada a bytes");
        tema.add(seccion);

        seccion = new DefaultMutableTreeNode("Salida de bajo nivel orientada a bytes");
        tema.add(seccion);

        seccion = new DefaultMutableTreeNode("Salida filtrada orientada a bytes");
        tema.add(seccion);

        seccion = new DefaultMutableTreeNode("Entrada de bajo nivel orientada a caracteres");
        tema.add(seccion);

        seccion = new DefaultMutableTreeNode("Entrada filtrada orientada a caracteres");
        tema.add(seccion);

        seccion = new DefaultMutableTreeNode("Salida de bajo nivel orientada a caracteres");
        tema.add(seccion);

        seccion = new DefaultMutableTreeNode("Salida filtrada orientada a caracteres");
        tema.add(seccion);

        seccion = new DefaultMutableTreeNode("Serializacion");
        tema.add(seccion);

        tema = new DefaultMutableTreeNode("Interfaces graficas de usuario");
        asig.add(tema);

        seccion = new DefaultMutableTreeNode("Programacion dirigida por eventos");
        tema.add(seccion);

        seccion = new DefaultMutableTreeNode("AWT");
        tema.add(seccion);

        seccion = new DefaultMutableTreeNode("Swing");
        tema.add(seccion);

        arbol = new JTree(asig);

        arbol.setFont(new Font("Arial",Font.BOLD,16));
```

```
        // Oyente de eventos
        arbol.addTreeSelectionListener(new TreeSelectionListener(){
            public void valueChanged(TreeSelectionEvent e) {
                DefaultMutableTreeNode nodo = (DefaultMutableTreeNode)
                                arbol.getLastSelectedPathComponent();

                if (nodo == null) return;

                Object nodeInfo = nodo.getUserObject();
                if (nodo.isLeaf())
                  campo.setText(nodeInfo.toString());
            }
        });

        c.add(arbol,BorderLayout.CENTER);
        c.add(campo,BorderLayout.SOUTH);

        setSize(400,600);
        setVisible(true);
    }
    public static void main(String[] args){
        new Ventana();
    }
}
```



JMenu

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

class Ventana extends JFrame{
    private JMenuBar mb;
    private JTextField info = new JTextField(20);

    Ventana(){

        // Los objetos de esta clase se pueden registrar como oyentes de eventos
        // del tipo ActionEvent
        class OyenteMenu implements ActionListener{
            public void actionPerformed(ActionEvent e){
                JMenuItem seleccionado = (JMenuItem)e.getSource();
                String accion = seleccionado.getActionCommand();
```

```
        if (accion.compareTo("About")==0)
            JOptionPane.showMessageDialog (
                Ventana.this,
                "Este programa ilustra el uso de menús",
                "Información",
                JOptionPane.INFORMATION_MESSAGE);
        info.setText(seleccionado.getActionCommand());
    }
}

OyenteMenu oyMenu = new OyenteMenu();

mb = new JMenuBar();

// Añadimos menus
JMenu archivo = new JMenu("Archivo");

// Añadimos submenus

JMenuItem nuevo = new JMenuItem("Nuevo");

// Le asociamos un escuchador de eventos ActionEvent
nuevo.addActionListener(oyMenu);

// Lo añadimos
archivo.add(nuevo);

JMenuItem abrir = new JMenuItem("Abrir");

// Le asociamos un escuchador de eventos ActionEvent
abrir.addActionListener(oyMenu);

// Lo añadimos
archivo.add(abrir);

JMenuItem ver = new JMenuItem("Ver todos");

// Le asociamos un escuchador de eventos ActionEvent
ver.addActionListener(oyMenu);

// Lo añadimos
archivo.add(ver);

// Ahora añadimos archivo a la barra de menus
mb.add(archivo);

// Creamos el menu Editar
JMenu editar = new JMenu("Editar");

// Añadimos submenus

JMenuItem copiar = new JMenuItem("Copiar");

// Le asociamos un escuchador de eventos ActionEvent
copiar.addActionListener(oyMenu);

// Lo añadimos
editar.add(copiar);

JMenuItem pegar = new JMenuItem("Pegar");

// Le asociamos un escuchador de eventos de raton
pegar.addActionListener(oyMenu);

// Lo añadimos
editar.add(pegar);

JMenuItem cortar = new JMenuItem("Cortar");
// Le asociamos un escuchador de eventos ActionEvent
cortar.addActionListener(oyMenu);

// Lo añadimos
editar.add(cortar);

// Añadimos editar a la barra de menu
mb.add(editar);

// Creamos un menu
JMenu ayuda = new JMenu("Ayuda");

// Añadimos submenus

JMenuItem about = new JMenuItem("About");
// Le asociamos un escuchador de eventos ActionEvent
about.addActionListener(oyMenu);

// Lo añadimos
ayuda.add(about);

// Añadimos ayuda a la barra de menu
mb.add(ayuda);

setJMenuBar(mb);
```

```
        info.setFont(new Font("Arial",Font.BOLD,18));
        getContentPane().add(info,BorderLayout.SOUTH);

        setSize(500,500);
        setVisible(true);

    }

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