

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

## Tema 7 – Introducción a la programación gráfica

### Introducción a los GUI y a Swing

Una vez que tenemos un control general de las estructuras básicas del lenguaje Java así como de la POO, vamos a avanzar en el aspecto de presentación hacia el usuario. Dejaremos la aséptica consola y pasaremos a realizar aplicaciones en modo gráfico (GUI: Graphic User Interface). Un GUI se construye con los llamados componentes o controles gráficos conocidos también como widgets (Window Gadgets).

Tenemos que manejar entonces librerías de componentes gráficos. Estos componentes no son más que clases a partir de las cuales crearemos objetos que se convertirán en nuestros formularios, botones, menús, etc...

*Ejemplo: Un botón con el texto Aceptar en su interior no es más que un objeto de la clase JButton cuya propiedad text contiene el String "Aceptar". Se le puede dar valor a través del setter setText.*

Es decir, realmente no estudiaremos apenas nada nuevo de programación general (salvo la denominada orientación a eventos), si no que aprenderemos a manejar clases e interfaces que se encuentran en determinadas librerías gráficas.

En Java se dispone de varias posibilidades, las más conocidas son AWT (Abstract Window Toolkit) y Swing. La primera fue la forma inicial que surgió en Java de hacer aplicaciones gráficas multiplataforma. Constaba de una serie de clases que tenían los elementos más básicos y comunes de todos los entornos gráficos (XWindow, MacOS, Windows, etc...). Se establecía una forma estándar pero muy pobre ya que sólo se disponían de aquellas propiedades que eran comunes en todos los entornos gráficos: Textos, tamaños y funcionalidad básica. Surgieron entonces otras posibilidades que mejoraban esta funcionalidad. Una de ellas apareció de la mano de Netscape y viendo Sun la potencialidad de dichas librerías se asoció con Netscape para realizar Swing.

Swing es una librería de componentes basados en AWT (la base JComponent deriva del Container de AWT) que tienen más versatilidad que los primeros. Además el consumo de recursos es mucho menor ya que mientras AWT crea una ventana por cada componente, Swing tiene una organización en contenedores que hace que el consumo de recursos sea menor.

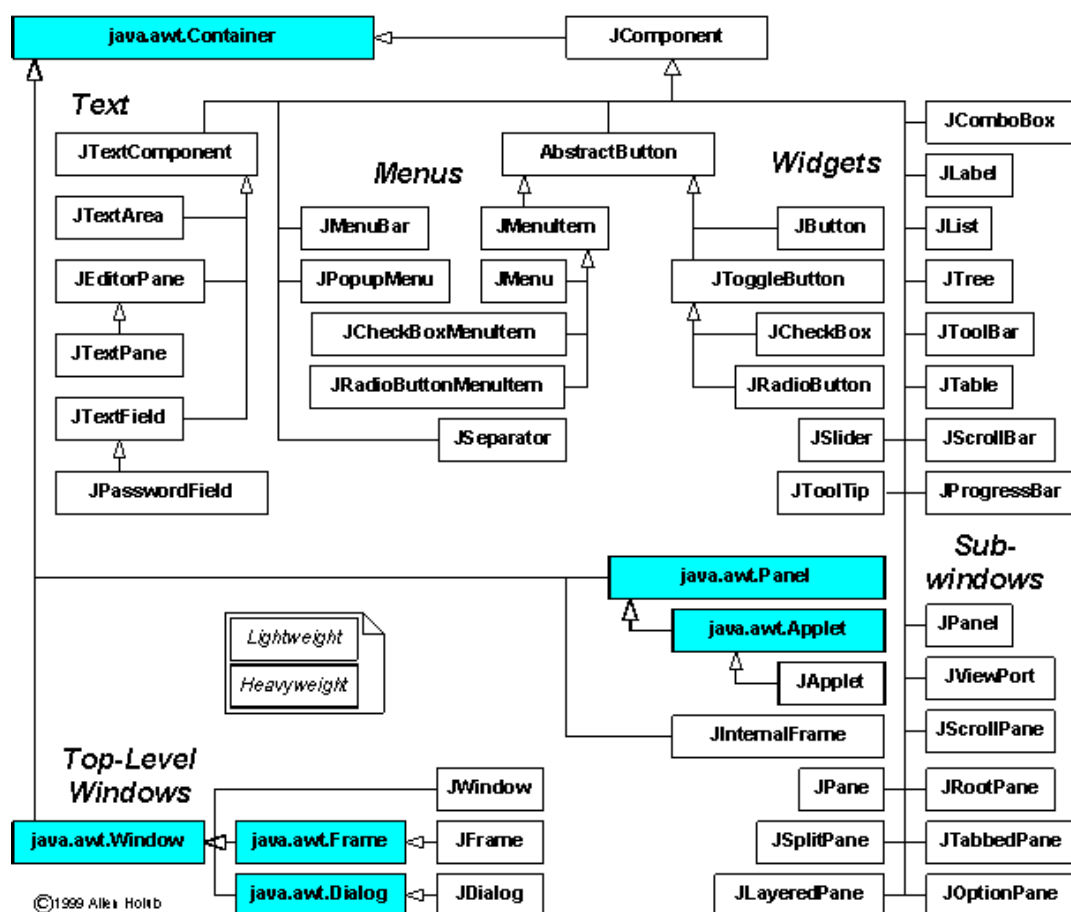
Todos los componentes gráficos se encuentran en la llamada JFC (Java Foundation Classes): tanto AWT como Swing, además de Java2D que permite realizar dibujos bidimensionales y clases para internacionalización, accesibilidad y Look and feel (apariencia) de las aplicaciones. Los componentes AWT se encuentran en concreto en el package *java.awt* mientras que Swing está en el package *javax.swing* el cual también importaremos pues será este grupo de componentes el que aprendamos a utilizar.

Swing usa la arquitectura MVC (Modelo-Vista-Controlador o Model-View-Controller). El modelo de un componente son los datos almacenados sobre dicho componente: estado de un botón, elementos en una lista, etc...La vista es la representación en pantalla y el controlador es el que gestiona el comportamiento como el click de un ratón o la pulsación de una tecla. Este tipo de arquitectura está muy extendido hoy en día.

Finalmente y mucho más novedosa que Swing y AWT está JavaFX. Según Oracle será (está empezando a ser) el sustituto de Swing aunque aún no está tan extendido como este último, motivo por el cual nos

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

quedaremos con las librerías Swing. Es más, según Oracle ya no va a actualizar más Swing en favor de JavaFX.



### GUI básica: JOptionPane

Como primer acercamiento al modo gráfico vamos a repetir un programa que vimos en el primer tema de programación:

```
import javax.swing.*;

public class HolaWin{
    public static void main(String[] args){
        JOptionPane.showMessageDialog(null,
            "Welcome to the Java World",
            "Usando Swing",
            JOptionPane.INFORMATION_MESSAGE);
    }
}
```

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

*Actividad: Analiza el programa desde tus conocimientos.*

*Nota: En VSCode puede haber algún problema con el IntelliSense para el autocompletado. Si es así el problema puede ser por la cantidad de resultados mostrada o por un problema que hay con algunas versiones. Las soluciones serían ambas en **settings**.*

*Por un lado busca Java → Completion:Max Results aumenta o pon a 0. Ten cuidado porque puede ralentizar dependiendo del ordenador.*

*Por otro busca java completion filteredTypes. Te lleva al Json. Ahí si tienes*

```
"java.completion.filteredTypes": [
    "java.awt.*",
    "com.sun.*"
]
```

*Elimina la líneas awt.*

*Si no lo tuvieras. Lo copias, grabas y luego eliminas las lineas awt.*

*Si lo consideras necesario reinicia el entorno.*

La clase *JOptionPane* contiene principalmente métodos estáticos para pedir datos al usuario, mostrar cierta información y pedir una respuesta a un diálogo.

Esto lo hace todo sacando un cuadro de diálogo denominado modal, lo que significa que bloquea el resto de la aplicación a nivel de interfaz de usuario hasta que se pulse uno de los botones del cuadro de diálogo. Prueba a meter esta línea de depuración antes de finalizar el main:

```
System.err.println("Esto no se ejecuta hasta que se cierra el diálogo modal");
```

Por cierto que el hecho de poder escribir cosas en consola de error nos va a resultar muy interesante para realizar líneas de depuración sin tener que realizar interfaz gráfico para las mismas.


Otro ejemplo más completo:

```
import javax.swing.*;

public class PruebaGUI {
    public static void main(String args[]) {

        String nombre=JOptionPane.showInputDialog("Dime tu nombre");
        JOptionPane.showMessageDialog(null,
            "Hola, "+nombre,
            "Saludo",
            JOptionPane.PLAIN_MESSAGE);

    }
}
```

	RAMA:	Informática	CICLO:	DAM			
	MÓDULO	Programación				CURSO:	1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:		
	AUTOR	Francisco Bellas Aláez (Curro)					

En este caso se usan dos métodos, uno para petición de datos (*showInputDialog*) y otro para mostrar datos (el ya conocido *showMessageDialog*). Se puede ver que para cualquiera de los dos hay varias sobrecargas con las que puedo indicar varias posibilidades: Título del cuadro, botones que aparecen, iconos que muestra, valor por defecto, etc...

Mira el [Apéndice I](#) para entender mejor el uso y los distintos miembros de la clase *JOptionPane*.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

## Introducción a las aplicaciones GUI

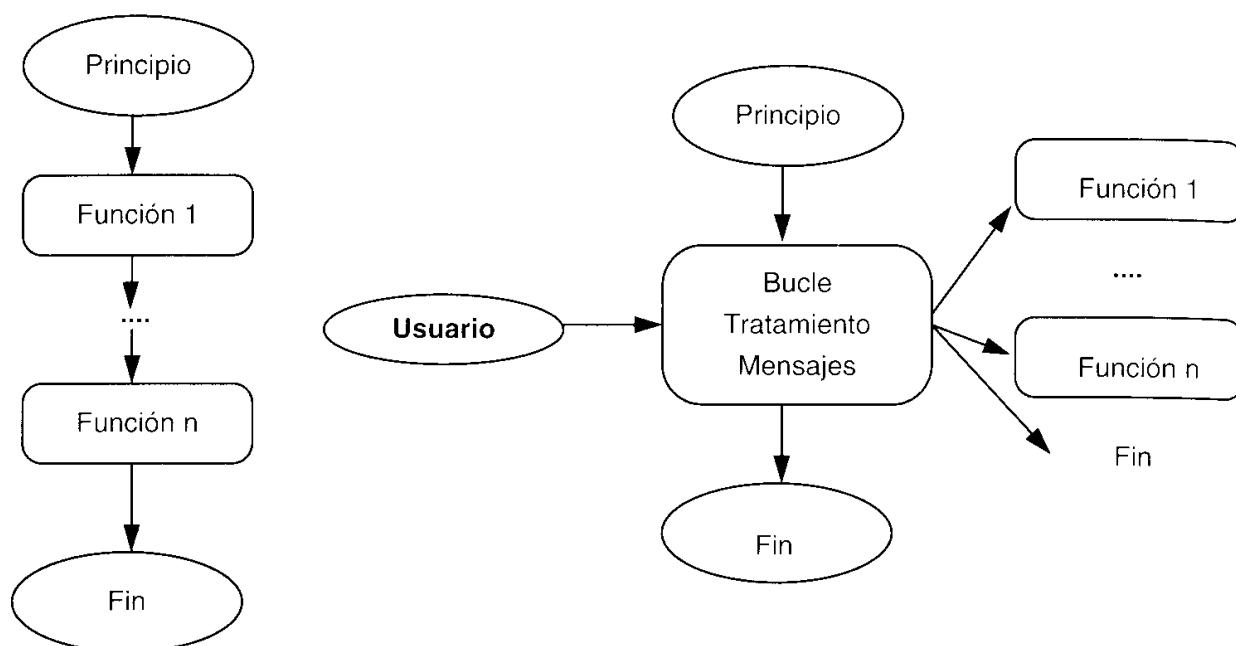
Lo que hemos visto en el punto anterior es básicamente cómo hacer una aplicación de consola pero con preguntas y respuestas en ventanas gráficas. Pero una aplicación gráfica completa es algo más complejo. En este punto sentaremos las bases para la realización de este tipo de aplicaciones.

### Programación orientada a eventos vs programación clásica

*Conéctate a la web <https://studio.code.org/flappy/3> y realiza alguno de los puzzles ¿Qué diferencia encuentras entre esta programación y la que hacías a principio de curso con Minecraft?  
Es una programación similar a la que estás haciendo en Javascript para web.*

En la programación que hemos visto hasta el momento el programador tenía el control completo de la aplicación. Se estructura el programa de forma que el programador va llevando al usuario a través de menús y opciones por donde le interesa.

En la programación orientada a eventos hay un cambio de paradigma. Es decir, cambia la interrelación entre el usuario y la aplicación. Ahora el programa no lleva al usuario por distintas zonas, si no que el programa espera a que el usuario o algún otro elemento del sistema realice una acción (denominada evento) y entonces el programa responde ante dicha acción.



Cuando por ejemplo el usuario genera un clic de ratón, la JVM recoge esa pulsación a través del sistema operativo y la convierte en un objeto tipo evento. Dicho objeto tiene información sobre el evento: que evento es, cuando se produjo, coordenadas dónde se produjo, si había alguna tecla pulsada, ... Esta

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

información se le pasa al programa que debe tener algún método de escucha de eventos que se encargue de recogerlos y pasarlos al método que se haya programado.

Es en la clase Component donde se genera este control de eventos. Puedes ver las fuentes en:

<http://developer.classpath.org/doc/java/awt/Component-source.html>

Busca funciones como *processMouseEvent* o *processKeyEvent* y verás ahí pequeños *switch* que son parte vario if que existen en *processEvent*.

### Creación de una ventana

La clase base para crear una ventana es *JFrame*. Para realizar una aplicación **heredaremos dicha clase** y la ampliaremos con nuestros componentes para crear ventanas más complejas, pero veamos como se haría una aplicación muy simple **sin herencia** (en el futuro NO lo haremos así).

```
import javax.swing.*;

public class PruebaGUI {
    public static void main(String[] args) {
        // Creamos la ventana con un título a través del constructor
        JFrame frame = new JFrame("The Sultans of Swing");

        // Creamos una "Etiqueta de texto" y especificamos su tooltip
        JLabel label = new JLabel("La aplicación perfecta");
        label.setToolTipText("Esto es un componente JLabel");

        // Incluimos la JLabel en la colección de componentes de la ventana
        frame.add(label);

        // Indica que si se le da al botón de cierre lo que se cierra es
        // la aplicación, no sólo la ventana
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Tamaño de la ventana
        frame.setSize(300, 100);

        // la hacemos visible y se queda a la espera de eventos
        frame.setVisible(true);
    }
}
```

De este código deducimos algunas cosas importantes a tener en cuenta en futuros programas:

- Un formulario cualquiera se crea a partir de una clase *JFrame*. En el constructor se puede indicar el título de dicha ventana.
- Cualquier componente que se vaya a colocar dentro del formulario se crea como un objeto.
- Para que un componente se convierta en parte de un formulario hay que añadirlo a la colección de componentes. Se puede hacer directamente con *add(elemento)*, o accediendo a la colección mediante *getContentPane().add(elemento)*. Más abajo hablamos del *ContentPane*.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

- Si no se establece lo que sucede al darle al botón de cierre, se cierra la ventana pero no la aplicación (no se cierra el bucle de gestión de eventos).
- Se pueden cambiar distintas propiedades tanto del formulario (*JFrame*) como de los componentes. Una de ellas es el tamaño que se cambia con *setSize()*.
- Cuando hacemos visible el frame, realmente se lanza un hilo en paralelo con el programa principal (main), de hecho prueba a poner la línea de depuración siguiente justo antes de finalizar el *main* y observa cuándo se ejecuta.

System.**err**.println("Ahora esta línea sí se ejecuta");

Veamos a continuación como haríamos lo mismo pero usando herencia. Por un lado establecemos una clase que herede de *JFrame* y será ahí donde creemos el interfaz de usuario:

```
//Heredamos de la clase JFrame para hacer un formulario a medida
public class FrmPrincipal extends JFrame {
    // Inicializamos la interfaz de usuario en el constructor
    public FrmPrincipal(){
        //Llamando a super podemos inicializar el título de la ventana
        super("The Sultans of Swing");

        // Creamos una "Etiqueta de texto" y especificamos su tooltip
        JLabel label = new JLabel("La aplicación perfecta");
        label.setToolTipText("Esto es un componente JLabel");

        // Incluimos la JLabel en la colección de componentes de la ventana
        this.add(label);
    }
}
```

Y en el main dejamos exclusivamente la instanciación e inicialización del objeto frame:

```
public static void main(String[] args) {
    // Creamos la ventana de tipo FrmPrincipal
    FrmPrincipal frame = new FrmPrincipal();

    // Indica que si se le da al botón de cierre lo que se cierra es
    // la aplicación, no sólo la ventana
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Tamaño de la ventana
    frame.setSize(300, 100);

    // la hacemos visible y se queda a la espera de eventos
    frame.setVisible(true);
}
```

Por supuesto podríamos meter todo (salvo la instanciación) en el constructor del formulario, pero normalmente las propiedades del formulario las inicializaremos en el main.

Cada *JFrame* tiene varias capas capas: el *Background* o *RootPane* es el fondo, luego está el *LayeredPane* que

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

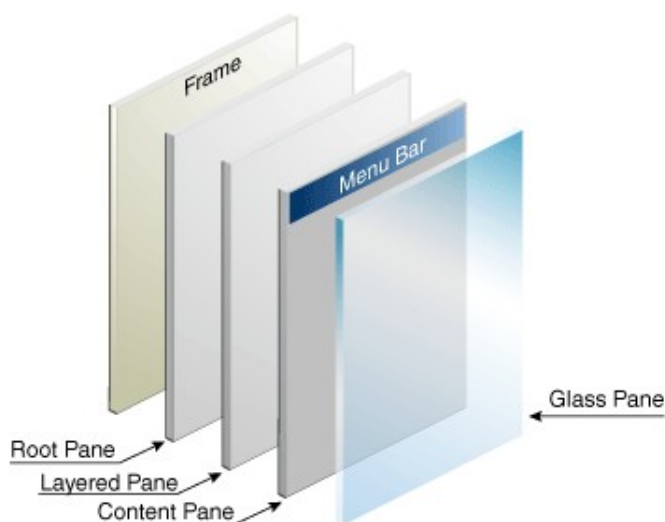
contiene la barra de menú y el *ContentPane* que es dónde se encuentran los componentes. Finalmente el *Glass Pane* es la parte frontal donde se colocan algunos elementos que tienen que estar por encima de los componentes como los *tooltip*.

Cuando hacemos *add* en el formulario, lo estamos añadiendo en la colección del *ContentPane*. Incluso puedes sustituir la línea que añade la *JLabel* al formulario por esta:

```
frame.getContentPane().add(label);
```

Aunque normalmente se usa la abreviada es adecuado saber el sitio real donde se guarda para ciertas situaciones que veremos más adelante.

Más información en: <https://docs.oracle.com/javase/tutorial/uiswing/components/rootpane.html>



## Layouts

Por defecto al trabajar con Java no podemos colocar los componentes en la posición que nos da la gana. Esto se debe a que Swing (realmente AWT) automatiza de diversas formas la colocación de dichos componentes precisamente por su capacidad multiplataforma. Esto es, si yo coloco los componentes donde me plazca puedo tener problemas dependiendo de la plataforma donde se vaya a usar.

Con los layout managers se consigue que los componentes se adapten automáticamente. La desventaja es que cuesta más el diseño y la programación de los mismos.

Debido a esto último, como este es un primer acercamiento a los GUIs no usaremos un layout concreto. Sin embargo veamos en funcionamiento esto que hemos comentado anteriormente ya que nos facilitará la labor en algunos programas y ejemplos.

En el siguiente ejemplo se crean dos etiquetas y dos botones y se ven como se colocan automáticamente en el formulario a pesar de que le damos una posición determinada.

Modifica *FrmPrincipal* de la siguiente forma



COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```

public FrmPrincipal(){
    super("Prueba de Layouts");
    JLabel etiqueta1= new JLabel("Etiqueta Uno");
    JLabel etiqueta2= new JLabel("Etiqueta Dos");
    JButton boton1=new JButton("Aceptar");
    JButton boton2=new JButton("Cancelar");

    etiqueta1.setLocation(0,0);
    etiqueta2.setLocation(100,0);
    boton1.setLocation(0,100);
    boton2.setLocation(100,100);
    this.add(etiqueta1);
    this.add(etiqueta2);
    this.add(boton1);
    this.add(boton2);
}

```

Si ejecutas este programa lo único que verás es un botón que llena todo el formulario. Cambiando el orden de añadir elementos cambia el componente que se ve. Es decir que no hace caso a las posiciones que le hemos mandado.

Esto es porque un formulario por defecto tiene un layout tipo *BorderLayout* que define 5 zonas en el contenedor (Norte, sur, este, oeste y centro) y si no indicamos zona, coge todo. Prueba a cambiar los *add* con esta sobrecarga:

```

this.add(etiqueta1, BorderLayout.NORTH);
this.add(etiqueta2, BorderLayout.SOUTH);
this.add(boton1, BorderLayout.EAST);
this.add(boton2, BorderLayout.WEST);

```

Si no lo has hecho debes importar *awt* ya que es dónde se encuentra *BorderLayout*.

Como ves, las posiciones cambian pero no son las que hemos programado. Es el layout manager quién controla la posición.

Otro layout muy típico es el *FlowLayout*, para cambiarlo introduce la siguiente línea en cualquier punto del constructor (habitualmente se hace justo después de la llamada a *super()*):

```

this.setLayout(new FlowLayout());

```

Puedes dejar los *add* como estaban antes. Nuevamente las posiciones cambian pero no son absolutas.

Finalmente vamos a eliminar los layout managers:

```

this.setLayout(null);

```

Si ahora lo ejecutas no ves nada. El problema está en que si no especificamos el layout, entonces tenemos que dar todos los datos, y no hemos dado datos de tamaño de componentes.

Para ello se puede sustituir los *setLocation* por *setBounds* que dan tanto posición como tamaño o usar *setSize*. Usaremos este último tomando como tamaño el denominado *preferredSize* que es el “ideal” para

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

ajustarse al tamaño del texto.

```
etiqueta1.setSize(etiqueta1.getPreferredSize());
etiqueta2.setSize(etiqueta2.getPreferredSize());
boton1.setSize(boton1.getPreferredSize());
boton2.setSize(boton2.getPreferredSize());
```

Es recomendable cambiar el el main el tamaño del formulario:

```
frame.setSize(300,300);
```

Para ver el uso de *setBounds* prueba a sustituir el *setLocation* y *setSize* de *boton2* por:

```
boton2.setBounds(100, 100, 100, 30);
```

Por supuesto se le puede dar un tamaño estándar:

```
etiqueta1.setSize(100,20);
```

En el caso de estar usando un Layout, este además de la posición también gestiona el tamaño de un componente ajustándolo según considere. Si se deseara fijar un tamaño de componente en un layout que no sea null, en lugar de usar *setSize* hay que usar ***setPreferredSize***. Prueba teniendo FlowLayout

```
etiqueta1.setPreferredSize(new Dimension(100,200));
```

Oracle, y en general dentro del mundo de la programación en Java **se recomienda el uso de layouts**. Sin embargo una de las desventajas de los layouts es que lleva tiempo aprender a manejarlos y usarlos con soltura. Además es recomendable no usar un único layout si no que meter unos layouts dentro de otros y usar distintos componentes contenedores como paneles y otros.

Por tanto en este primer acercamiento a los GUI usaremos solamente el *FlowLayout* o el posicionamiento absoluto (layout null) dependiendo de la aplicación y por tanto obviaremos el uso del resto de layouts por el momento salvo que el alumno desee ir aprendiendo a usar otros y los vaya añadiendo a sus aplicaciones.

Si quieres profundizar un poco más en los layouts te recomiendo este fantástico artículo de Javahispano:

<http://www.javahispano.org/portada/2011/8/1/tutorial-de-layouts.html>

También puedes leer más información sobre los layouts en la página de Oracle:

<http://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>

### Estructura final de un programa swing

Es posible que en el ejemplo que hemos estado haciendo el código te haya quedado algo desorganizado. A continuación se presenta como se debe establecer el código para que quede bien ordenado dentro del formulario que estamos creando:

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

1. Primero hacemos las declaraciones de componentes fuera del constructor.
2. Creamos el constructor con llamada a super y establecemos el layout
3. Instanciamos e inicializamos las propiedades de cada componente ordenadamente y por separado.

De esta forma tendríamos el siguiente código:

```
import javax.swing.*;

public class FormularioPrincipal extends JFrame {
    // Declaración de los componentes
    private JLabel etiqueta1;
    private JLabel etiqueta2;
    private JButton boton1;
    private JButton boton2;

    // Constructor
    public FormularioPrincipal() {
        super("Mi aplicación");
        this.setLayout(null);

        //Etiqueta 1
        etiqueta1 = new JLabel("Etiqueta Uno");
        etiqueta1.setSize(etiqueta1.getPreferredSize());
        etiqueta1.setLocation(0, 0);
        this.add(etiqueta1);

        //Etiqueta 2
        etiqueta2 = new JLabel("Etiqueta Dos");
        etiqueta2.setSize(etiqueta2.getPreferredSize());
        etiqueta2.setLocation(100, 0);
        this.add(etiqueta2);

        //Botón 1
        boton1 = new JButton("Aceptar");
        boton1.setLocation(0, 100);
        boton1.setSize(boton1.getPreferredSize());
        this.add(boton1);

        //Botón 2
```

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```

        boton2 = new JButton("Cancelar");
        boton2.setSize(boton2.getPreferredSize());
        boton2.setLocation(100, 100);
        this.add(boton2);
    }
}

```

Y en el programa principal (en otra clase) tendríamos simplemente:

```

import javax.swing.*;

public class PruebaGUI {
    public static void main(String[] args) {
        FormularioPrincipal aplicacion = new FormularioPrincipal();

        //Si quieres centrar la ventana escribe:
        aplicacion.setLocationRelativeTo(null);

        aplicacion.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        aplicacion.setSize(300, 300);
        aplicacion.setVisible(true);
    }
}

```

Si se han entendido los conceptos previos no debería tener mayor dificultad comprender la reordenación anterior.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

## Gestión de eventos (Event Handling)

Una vez que se crea una interfaz de usuario con distintos componentes, interesa que esos componentes respondan a distintas acciones que se hacen sobre ellos: realizar un click en un botón, mover el ratón por encima de una etiqueta, cambiar el texto en una caja de edición, etc...

Parte de estas acciones ya están controladas: el formulario cambia de tamaño, se cierra, al pasar por encima de un botón cambia de aspecto, etc... Pero nos interesa un control personalizado de eventos.

Para poder hacer esto es necesario que la clase que estoy construyendo tenga alguna forma de “escuchar y manejar” esos eventos. Esto se realiza mediante interfaces. El gestor de eventos (event handler) es una interfaz que tiene una serie de funciones predefinidas y que al implementarla hay que codificarlas. Esas funciones son ejecutadas cuando la JVM envía un mensaje de evento a la aplicación. Veámoslo con un ejemplo.

Escribe una nueva clase denominada *Eventos1* que herede de *JFrame* e implemente la interfaz *ActionListener* que se encargará de gestionar el evento click de un botón. Al escribir la línea:

```
public class Eventos1 extends JFrame implements ActionListener
```

Veremos que nos da un error de que no están implementados los métodos. El propio entorno nos da la opción de escribir automáticamente la cabecera de dichos métodos para luego “rellenarlos” nosotros.

Antes de eso creamos dicho formulario con un botón y una etiqueta de la siguiente forma:

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

public class Eventos1 extends JFrame implements ActionListener {
    // Declaración de los componentes
    private JLabel etiqueta1;
    private JButton boton1;

    // Constructor
    public Eventos1() {
        super("Controlando eventos I");
        this.setLayout(new FlowLayout());

        //Botón
        boton1 = new JButton("¡¡Púlsame!!");
        this.add(boton1);

        //Etiqueta
        etiqueta1 = new JLabel("Aún no has pulsado el botón");
        this.add(etiqueta1);
    }

    @Override
    public void actionPerformed(ActionEvent evento) {
    }
}
```

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

}

El programa principal es idéntico al del caso anterior. Si cabe se puede cambiar el tamaño del formulario a 300x100 y, por supuesto, el nombre de la clase que estamos creando a *Evento1*.

Esto compila perfectamente pero aún no hace nada. La acción que queremos que realice la tenemos que escribir dentro del método que ha aparecido: *actionPerformed*.

Este método está asociado a través de la interfaz al evento click de la máquina virtual (JVM). Por tanto el código que escribamos dentro se ejecutará cuando hagamos un click en el botón. Escribamos dentro del método la siguiente línea:

```
this.etiqueta1.setText("¡Botón pulsado!");
```

Si ejecutamos veremos que sigue sin funcionar. Esto es porque no hemos indicado que componente o componentes deben responder al evento. Esto se denomina registrar el componente y se trata simplemente de indicar al componente en cuestión quién se encarga de escuchar y manejar sus eventos.

En este caso es el propio objeto dónde está el botón: *this*. Con lo que añadimos en el constructor (en la inicialización del botón) la siguiente línea:

```
boton1.addActionListener(this); //Le indica al botón quién tiene el actionPerformed
```

Pruébalo y comprueba que funciona todo correctamente. Puedes añadir si quieres una línea de depuración en la función *actionPerformed* para ver que realmente pasa por ahí cada vez que se pulsa el botón.

Podríamos sacar el gestor de la clase y veremos como hacerlo más adelante, pero por ahora el manejador será la propia clase para no complicar demasiado la estructura de nuestros programas .

Resumo entonces los pasos necesarios para controlar un evento que actúa sobre un componente:

1. Definir una Interfaz relacionada con el evento a controlar. La interfaz incluso puede ser una clase aparte donde se haga todo el control de eventos del formulario (esto está más en relación con el modelo MVC comentado anteriormente) o puede ser una clase interna (inner class).

La lista de Listeners completa la tienes en

<http://docs.oracle.com/javase/tutorial/uiswing/events/eventsandcomponents.html>

aunque iremos viendo lo que más nos interesa a medida que avance el curso.

2. Crear y completar el código asociado a los métodos del interface que es necesarios sobrecribir para responder a los eventos.
3. Agregar el controlador (pueden ser varios) a la lista de gestores de eventos del componente.

### Propiedades "oyentes" y controlando eventos de varios componentes

Es evidente que cuando creamos una aplicación no usamos un único evento con un único componente. Para controlar qué componente es el que reacciona al evento tenemos el parámetro que se le pasa al método *actionPerformed*. En dicho parámetro hay información sobre el evento y sobre el componente sobre el que

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

recae el evento.

Lo veremos con el siguiente ejemplo. Crea una nueva clase denominada Eventos2 con el siguiente código fuente:

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

public class Eventos2 extends JFrame implements ActionListener {

    // Declaración de los componentes
    JLabel etiqueta1;
    JButton btnIncremento;
    JButton btnDecremento;
    int contador;

    // Constructor
    public Eventos2() {
        super("Controlando eventos II");
        this.setLayout(new FlowLayout());
        contador = 0;

        // Botón Incremento
        btnIncremento = new JButton("Incremento");
        btnIncremento.addActionListener(this);
        this.add(btnIncremento);

        // Botón Decremento
        btnDecremento = new JButton("Decremento");
        btnDecremento.addActionListener(this);
        this.add(btnDecremento);

        // Etiqueta
        etiqueta1 = new JLabel(String.valueOf(contador));
        this.add(etiqueta1);
    }

    @Override
    public void actionPerformed(ActionEvent evento) {

    }
}
```

Ahora vamos a programar la parte de los eventos de cada botón. En el método *actionPerformed* en laas propiedades del parámetro *evento* viene indicada información sobre el evento. Algunos interesantes:

- Mediante el método *getSource* obtenemos el componente sobre el que recae el evento

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

- Mediante *getActionCommand* se obtiene el String asociado al estado del componente. Esto depende del componente que lanza el evento. Por ejemplo en el caso de un JButton o JTextField es el texto que incluye en Text. En el caso del JButton también puede especificarse a medida mediante el método *setActionCommand*.
- Con *getModifiers* se puede saber las teclas modificadoras (CTRL, ALT, SHIFT) que estaban siendo pulsadas mientras se producía el evento.

Los vemos en funcionamiento en el código siguiente que hay que introducir en *actionPerformed*:

```
int salto=1;
// En el título del formulario ponemos el componente afectado
this.setTitle(evento.getActionCommand());

// Si se está pulsando SHIFT incrementa/decrementa en 10
if ((evento.getModifiers() & ActionEvent.SHIFT_MASK)
    ==ActionEvent.SHIFT_MASK) {
    salto=10;
}
// Si se ha pulsado el botón incremento...
if (evento.getSource() == btnIncremento) {
    contador+=salto;
}
// Si se ha pulsado el botón decremento...
if (evento.getSource() == btnDecremento) {
    contador-=salto;
}
// Actualizamos la etiqueta al nuevo valor
etiqueta1.setText(String.format("%10d", contador));
```

Mediante *getSource* detectamos el botón (o en general el componente) que ha sido pulsado.

Analiza el uso de *getModifiers*, con lo visto hasta el momento deberías entenderlo. Fíjate en el uso de un & simple en lugar de &&. Para entender mejor su funcionamiento añade en el *actionPerformed* la línea:

```
System.err.println(Integer.toString(Integer.parseInt(evento.getModifiers())));
```

cuando ejecutes observa el valor de *getModifiers* cuando pulsas distintas teclas modificadoras. Deberías entender el funcionamiento.

*Ejercicio:*

*Crea un formulario con 3 botones y de título Ejercicio. Uno con texto Amarillo, otro con texto Azul y un tercero con el texto Aleatorio. Si se pulsa el amarillo o el azul el color de fondo del formulario cambiará a dicho color (averigua cómo se hace). Si pulsa el botón Aleatorio cambiará la posición del formulario en la pantalla dentro de un área de 600x400 aproximadamente.*



COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

## Componentes y Eventos

En este apartado describiremos el uso de algunos componentes más típicos así como eventos y formas de uso más comunes.

Ten en cuenta que debido a la herencia, muchas de las cosas descritas en un componente las tienen también otros componentes (Texto, Colores, ToolTip, etc...), por lo que nos limitaremos a describir la propiedad o su setter en un componente y se entenderá explicada para el resto.

También nos centraremos en los setters de los componentes ya que son los que nos permiten cambiar sus propiedades, pero ten en cuenta que en la mayoría de los casos existe el getter correspondiente si se necesita obtener el valor.

### JLabels

Etiqueta informativa. Permite mostrar tanto texto como imágenes alineados de diferentes formas. No responde a eventos con *ActionListener*. Varios constructores sobrecargados para inicializar Texto y/o Icono.

Algunos métodos interesantes:

*setText(String)*: Cambia el texto que almacena la etiqueta. Acepta código HTML lo que puede ser útil para retornos de carro o formatos.

*setBackground/setForeground(Color)*: Establece el color de fondo o del texto. Para que funcione el color de fondo, en algunos sistemas hay que poner que la etiqueta sea opaca (*setOpaque(true)*).

*setFont(Font)*: Establece la fuente.

Ejemplo de definición de una JLabel:

```
etiqueta=new JLabel("Prueba");
etiqueta.setForeground(new Color(10,132,245));
etiqueta.setFont(new Font("Arial",Font.ITALIC+Font.BOLD, 16));
this.add(etiqueta);
```

Para los colores se puede usar la clase Color. Lo más cómodo es llamar al constructor mediante sus componentes RGB (puede ser en hexadecimal anteponiendo 0x). También tiene una serie de colores predefinidos (como enumerado) y otras funciones que nos permiten definir uno nuevo como *decode* que permite definirlo a partir de las componentes RGB en una cadena (tipo HTML).

Para establecer una fuente se usa alguna de las sobrecargas de Font. En el ejemplo se establece con Fuente Arial, en negrita y cursiva y de tamaño 16 puntos. No trabajaremos mucho con la clase Font, pero si por curiosidad quieres obtener la lista de fuentes de tu sistema puedes hacerlo con la siguiente rutina:

```
String[] nombreFuentes=GraphicsEnvironment
    .getLocalGraphicsEnvironment().getAvailableFontFamilyNames();
for(int i=0; i<nombreFuentes.length; i++){
    System.out.println(nombreFuentes[i]);
}
```

Más información en: <http://docs.oracle.com/javase/7/docs/api/javax/swing/JLabel.html>

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

## JButton

Botón estándar que puede tener un texto y/o una imagen que lo representa. Responde a eventos como ya hemos visto. Varios constructores sobrecargados para inicializar Texto y/o Icono.

Algunos métodos interesantes:

*setToolTipText(String)*: Establece el ToolTip del componente.

*doClick()*: Realiza la acción de pulsación del botón.

*setIcon(Icon)*: Establece un nuevo icono en el botón (usa new ImageIcon(Archivo imagen) para poner solo el nombre del archivo colócalo en el raíz del proyecto).

*setRolloverIcon(Icon)*: Establece un nuevo icono cuando se pasa por encima del botón sin necesidad de gestionar eventos.

Ejemplo: En el constructor del formulario:

```
salir=new JButton("Salir");
salir.setToolTipText("Sale de la aplicación");
salir.addActionListener(this);
this.add(salir);
```

En *actionPerformed*:

```
if (e.getSource()==salir) {
    System.exit(0);
}
```

Más información en: <http://docs.oracle.com/javase/7/docs/api/javax/swing/JButton.html>

## JTextField

Es la caja de edición estándar de una línea. Permite introducir datos al usuario siempre que sea una caja editable. Dispone de varios **constructores** con los siguientes parámetros:

*Parámetro String*: Aparece dicha cadena en la caja de texto. El tamaño será el necesario para que quepa el texto.

*Parámetro int*: Indica el número de columnas de tamaño. El ancho de columna es un promedio según el tipo de letra usado.

*Parámetro String e int*: Constructor con los dos casos anteriores.

### Métodos comunes:

*String getText()*: Para obtener el texto dentro de la caja de edición. Tiene una sobrecarga con dos parámetros (inicio y tamaño) que permite coger un fragmento de dicho texto. En este último caso hay que controlar la excepción de inicio/tamaño inválidos.

*setEditable(boolean)*: Dispone de un parámetro booleano. Si es *true* la caja de texto es editable, si es *false* el usuario no podrá modificar su contenido.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

*String getSelectedText():* Obtiene sólo el texto seleccionado.

*select(int, int):* Establece selección por código. Se le indica inicio y fin de la selección. También pueden usarse en su lugar *setSelectionStart* y *setSelectionEnd*.

*selectAll():* Selecciona todo.

Cuando se enlaza con el evento *ActionListener*, lo que hace es realizar el método *actionPerformed* cuando se pulsa la tecla *Enter*.

Ejemplo de uso (Muévete con el tabulador, no con el ratón, para ver la selección en txt3):

```
public class CajasTexto extends JFrame implements ActionListener {
    JLabel lbl1;
    JLabel lbl2;
    JTextField txt1;
    JTextField txt2;
    JTextField txt3;

    public CajasTexto() {
        super("Cajas de edición de texto");
        this.setLayout(new FlowLayout());

        lbl1 = new JLabel("Introduzca su DNI");
        add(lbl1);


        txt1 = new JTextField(9);
        txt1.addActionListener(this);
        add(txt1);

        txt2 = new JTextField("Escriba aquí su nombre");
        txt2.addActionListener(this);
        add(txt2);

        txt3 = new JTextField("Información no editable", 23);
        txt3.setEditable(false);
        txt3.addActionListener(this);
        add(txt3);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == txt1 || e.getSource() == txt3) {
            ((JTextField)e.getSource()).selectAll();
        }
        JOptionPane.showMessageDialog(null,
            ((JTextField) e.getSource()).getSelectedText(),
            "Cajas de Texto", JOptionPane.INFORMATION_MESSAGE);
    }
}
```

Más información en: <http://docs.oracle.com/javase/7/docs/api/javax/swing/JTextField.html>

	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

## JCheckBox

Componente que hereda de *JButton* y que por tanto se comporta como tal, pero una vez pulsado mantiene el estado (chequeado/no chequeado).

Contiene todos los métodos de *Button* por lo que pueden ser usados. Sin embargo el interface recomendado no es *ActionListener* si no *ItemListener*. La diferencia en cuanto al uso será simplemente que cambia el nombre del método ejecutado que será *itemStateChanged*. Sin embargo también el comportamiento es distinto: *ActionListener* se lanza ante un evento de usuario (click, barra espacio etc,...), *ItemListener* se lanza cuando cambia la propiedad *checked* ya sea por acción de usuario o por código.

**Es obligatorio** acostumbrarse a usar este listener con este componente.

*Nota:* Aquí una explicación sobre las diferencias: <http://stackoverflow.com/questions/9882845/jcheckbox-actionlistener-and-itemlistener>

Dispone de varias sobrecargas del constructor que le permiten iniciar propiedades como el texto, el icono o si está marcada o no al inicio.

Otros métodos:

*setEnabled(boolean):* Establece si un componente recibe o no eventos.

*isSelected:* devuelve true o false indicando si está o no seleccionado.

*setSelected(boolean):* Establece el estado del botón.

Veamos un ejemplo:

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

public class Botones extends JFrame implements ItemListener {
    JCheckBox chk1;
    JLabel lbl;

    public Botones() {
        super("Botones de estado");
        setLayout(new FlowLayout());

        lbl = new JLabel("Etiqueta de prueba");
        lbl.setFont(new Font("Arial", Font.PLAIN, 14));
        add(lbl);

        chk1 = new JCheckBox("Negrita");
        chk1.addItemListener(this);
        add(chk1);
    }
}
```

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

```

@Override
public void itemStateChanged(ItemEvent e) {
    if (chk1.isSelected()) {
        lbl.setFont(new Font("Arial", Font.BOLD, 14));
    } else {
        lbl.setFont(new Font("Arial", Font.PLAIN, 14));
    }
}
}

```

Aunque el ejemplo anterior funcione exactamente igual con un *ActionListener*, es recomendable usar el interfaz propio.

Más información en: <http://docs.oracle.com/javase/7/docs/api/javax/swing/JCheckBox.html>

## Timers Swing

Un timer es un temporizador que provoca que se ejecute un evento cada cierto tiempo si es repetitivo o una vez si no lo es. Tiene diversas aplicaciones como puede ser la de crear una animación o realizar una comprobación cada cierto tiempo.

En Java hay dos tipos de timers, uno en el paquete java.util y otro en swing. Veremos este último por ser más simple de usar.

El constructor permite inicializarlo ya con un tiempo de retardo en milisegundos y un *ActionListener*. **No hay que añadirlo a la lista de componentes** del formulario por no ser un elemento visual.

Métodos habituales:

*start*: inicia la temporización.

*stop*: para la temporización

*setRepeats*(boolean): establece si se repite o no se repite el timer.

*setInitialDelay*(int): Establece un retardo inicial distinto del resto en ms.

*setDelay*(int): retardo en ms.

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

Veamos un ejemplo:

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

public class TimerSwing extends JFrame implements ActionListener {
    JLabel tiempo;
    Timer temporizador;
    private int cont;

    public TimerSwing() {
        tiempo = new JLabel("0");
        add(tiempo);

        cont = 0;
        temporizador = new Timer(1000, this);
        temporizador.start();
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        cont++;
        tiempo.setText(cont + "");
        if (cont == 10) {
            temporizador.stop();
        }
    }
}
```

En este caso se usa un temporizador que se repite cada segundo cambiando el valor de una etiqueta. Prueba a cambiar el delay inicial, el normal o haz que se ejecute sólo una vez.

Debido a que el timer usa el evento *ActionListener*, si solo se tiene uno en la clase que estamos creando puede llegarse a hacer un poco confuso tener todo controlado en el mismo sitio. Aunque veremos otras formas de distribuir el control de eventos, vamos a ver una primera variante que me permite asociar lo que sucede en el timer (vale para cualquier componente) con un método para el sólo.

Se trata de pasarle un *ActionListener* que se crea en el propio parámetro. Escribe en el IDE la línea de esta forma:

```
temporizador = new Timer(1000, new Acti
```

y pulsa CTRL+espacio. Automáticamente lo completa dejándolo de la siguiente forma:

```
temporizador = new Timer(1000, new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // TODO Auto-generated method stub
    }
});
```

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

El método aquí creado será exclusivo del timer. Es más, si no hay otro componente que necesite un ActionListener, se puede eliminar el interface de la clase.

El programa quedaría así:

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

public class TimerSwing extends JFrame {
    JLabel tiempo;
    Timer temporizador;
    private int cont;

    public TimerSwing() {
        tiempo = new JLabel("0");
        add(tiempo);

        cont = 0;
        temporizador = new Timer(1000, new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                cont++;
                tiempo.setText(cont + "");
                if (cont == 10) {
                    temporizador.stop();
                }
            }
        });
        temporizador.start();
    }
}
```

Más información en: <http://docs.oracle.com/javase/7/docs/api/javax/swing/Timer.html>

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

## Referencias

### Libros:

Java for programmers. Second Edition. Deitel Developers Series. Prentice Hall 2012.

La Biblia del Java 2. Anaya Multimedia. Steven Holzner. Anaya Multimedia.

### Recursos Web:

Creating a GUI with Swing: <http://docs.oracle.com/javase/tutorial/uiswing/>

Swing, la solución para crear GUIs:

<http://www.dcc.uchile.cl/~lmateu/CC60H/Trabajos/edavis/swing.html>

Fuentes y Colores: [http://www.sc.ehu.es/sbweb/fisica/cursoJava/applets/grafico/color\\_font.htm](http://www.sc.ehu.es/sbweb/fisica/cursoJava/applets/grafico/color_font.htm)

Diseño de interfaces gráficas en Java:

<http://atobeto-eremita.blogspot.com.es/2009/08/disenio-de-interfaces-graficas-en-java.html>



COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

## Apéndice I: Uso ampliado del JOptionPane

Vimos que el JOptionPane es una clase que me permite sacar cuadros de diálogo estándar. Para entender un poco mejor el manejo de los mismos veamos más características.

Para indicar el tipo de mensaje se usan los siguientes valores constantes (son *int*):

`ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `WARNING_MESSAGE`, `QUESTION_MESSAGE` y `PLAIN_MESSAGE`.

Los tipos de cuadros que encontramos son:

- *showInputDialog*: Cualquiera de sus sobrecargas devuelve como *String* el dato que introduce el usuario. Esto es importante porque si se le piden números hay que realizar algún *parse*. Esto se debe tener en cuenta prácticamente para cualquier componente gráfico ya que todos trabajan habitualmente con cadenas.

Este cuadro de diálogo apenas lo vamos a usar una vez que nos metamos en programación más avanzada.

- *showMessageDialog*: Sirve para mostrar un mensaje con determinado icono que puede hacer resaltar una información.
- *showConfirmDialog*: para realizar una pregunta al usuario con respuestas tipo Si, No, Aceptar, Cancelar, etc... Probablemente este junto con el `MessageDialog` serán los que más usemos. Las posibilidades de botones las definen las constantes (son *int*):

`DEFAULT_OPTION`, `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, `OK_CANCEL_OPTION`

Vemos un ejemplo de uso de este último caso:

```
int respuesta = JOptionPane.showConfirmDialog(null, "Borrar Archivo",
    "Eliminar", JOptionPane.YES_NO_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE);
switch (respuesta) {
case JOptionPane.YES_OPTION:
    JOptionPane.showMessageDialog(null, "Sí");
    break;
case JOptionPane.NO_OPTION:
    JOptionPane.showMessageDialog(null, "No");
    break;
case JOptionPane.CANCEL_OPTION:
    JOptionPane.showMessageDialog(null, "Cancelar");
    break;
case JOptionPane.CLOSED_OPTION:
    JOptionPane.showMessageDialog(null, "No has elegido nada");
}
```

- *showOptionDialog*: Es el método más genérico, permite hacer cualquiera de las cosas anteriores y más. Se usa cuando se necesita crear un cuadro de diálogo que se sale de lo habitual. Úsalo cuando domines swing un poco más.

Más información en:

COLEXIO <b>VIVAS</b> S.L.	RAMA:	Informática	CICLO:	DAM		
	MÓDULO	Programación				CURSO: 1º
	PROTOCOLO:	Apuntes clases	AVAL:		DATA:	
	AUTOR	Francisco Bellas Aláez (Curro)				

<http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html>

## Apéndice II: Localización de los double

El uso que le damos habitualmente al *Double.parseDouble()* tiene el problema que para meter números con decimales es necesario usar el punto en lugar de la coma, a la que estamos más acostumbrados aquí. Si queréis tener esto en cuenta en vuestros programas, en lugar del *parseDouble* anteriormente mencionado debéis de utilizar un código con localización (podéis haceros una función, de hecho).

Suponiendo que tenéis un JTextField denominado txt1, en la gestión del evento podéis hacer algo como esto:

```
if (e.getSource() == txt1){
    NumberFormat format = NumberFormat.getInstance(new Locale ("ES","es"));
    Number number = null;
    try {
        number = format.parse(txt1.getText());
    } catch (ParseException ex) {
        ex.printStackTrace();
    }
    double d = number.doubleValue();
    System.err.println(d);
}
```

Os permite meter el número con coma y lo devuelve con . que es como trabaja Java.

Ojo porque el parse de NumberFormat al leer una cadena que empieza por numero y luego tiene letras se queda con la parte numérica y no salta excepción . Por ejmplo: 23abc leería 23.