

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

Tema 9 - Introducción a la programación gráfica, 2ª Parte.

Otros eventos y adaptadores

Hasta el momento hemos controlado los eventos principales asociados a algunos componentes. Si embargo nos interesará controlar otro tipo de eventos como los movimientos del ratón, pulsaciones de teclas o cerrado de la aplicación. Estos son los que aquí trataremos.

Eventos de ratón: *MouseListener* y *MouseMotionListener*

Explicaremos aquí los interfaces "listener" asociados a la gestión de eventos de ratón. Son el *MouseListener* y el *MouseMotionListener*. Existe un interface que implementa los dos anteriores para juntarlos en uno solo: el *MouseListener* de forma que si se van a usar varios eventos llega con este.

El primero de los interface, *MouseListener*, obliga a sobrecribir los siguientes métodos:

```
public void mousePressed(MouseEvent e): Cuando un botón del ratón es presionado.
```

```
public void mouseClicked(MouseEvent e): Cuando un botón del ratón es presionado y luego liberado (un click completo).
```

```
public void mouseReleased(MouseEvent e): Cuando un botón del ratón es liberado.
```

```
public void mouseEntered(MouseEvent e): Cuando el cursor del ratón entra en los límites de un componente.
```

```
public void mouseExited(MouseEvent e): Cuando el cursor del ratón sale de los límites de un componente.
```

Y el segundo, el *MouseMotionListener*, dispone de los siguientes:

```
public void mouseDragged(MouseEvent e): Cuando se mueve el ratón encima de un componente a la vez que se está presionando un botón.
```

```
public void mouseMoved(MouseEvent event): Cuando el ratón se mueve encima del componente.
```

Además en todos estos eventos el parámetro *e* de tipo *MouseEvent* dispone de diversos métodos con información sobre el evento. Algunos de ellos son:

```
e.getX(), e.getY(): Posición del ratón en el componente. La posición 0,0 es arriba a la izquierda.
```

```
e.getButton(): Devuelve un número entero que indica que botón del ratón cambia de estado (no el hecho de estar pulsado). Hay constantes como MouseEvent.BUTTON1 que representa al botón izquierdo o MouseEvent.BUTTON3 que representa al derecho.
```

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

e.isControlDown(): Booleano que indica si se está pulsando la tecla CTRL a la vez que sucede el evento.

e.isAltDown(): Booleano que indica si se está pulsando la tecla ALT a la vez que sucede el evento. Puede no funcionar en algunos sistemas si no es en combinación con otra tecla.

e.isShiftDown(): Booleano que indica si se está pulsando la tecla SHIFT (Mayúsculas) a la vez que sucede el evento.

e.getClickCount(): Número de clicks con los que sucede el evento.

e.getSource() / **getComponent()**: Componente sobre el que se produjo el evento.

Vemos el manejo de estos eventos con el siguiente ejemplo en el que además como novedad se usa una clase dentro de otra (inner class), más adelante entenderemos por qué lo hacemos así:

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

public class EventosRaton extends JFrame {
    JLabel lbl, lbl2;

    public EventosRaton() {
        super("Pueba eventos");
        setLayout(new FlowLayout());

        lbl = new JLabel("<html><h1>Probando eventos de ratón</h1></html>");
        lbl.addMouseListener(new MouseHandler());
        lbl.addMouseMotionListener(new MouseHandler());
        add(lbl);

        lbl2 = new JLabel("Datos");
        add(lbl2);
    }

    private class MouseHandler implements MouseListener, MouseMotionListener {

        @Override
        public void mouseDragged(java.awt.event.MouseEvent e) {

            lbl2.setText(String.format(
                "Posición X:%d Y:%d Botón pulsado: %s",
                e.getX(), e.getY(),
                SwingUtilities.isLeftMouseButton(e) ?
                "principal" : "otro"));
        }
    }
}
```

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

/* No es posible usar *getButton* en *mouseDragged* debido a que este método devuelve un cambio de estado en el botón, no que esté pulsado, y el botón no cambia durante la operación Drag por eso se usa las funciones de *SwingUtilities* */

```
@Override
public void mouseMoved(java.awt.event.MouseEvent e) {
    lbl2.setText(String.format("Posición X:%d Y:%d ", e.getX(),
        e.getY()));
}

@Override
public void mouseClicked(java.awt.event.MouseEvent e) {
    if(e.getClickCount() > 1) {
        lbl2.setText("Pulsación múltiple");
    } else {
        lbl2.setText(String.format(
            "<html>Boton Pulsado: %d<br>Teclas: CTRL:%s ALT:%s SHIFT:%s ",
            e.getButton(),
            e.isControlDown() ? "Sí" : "No",
            e.isAltDown() ? "Sí" : "No",
            e.isShiftDown() ? "Sí" : "No"));
    }
}
```

/* La tecla ALT solo se detecta en combinación con otras, si no la captura el SO para la gestión de menús. Este comportamiento puede cambiar según el sistema de ventanas */

```
@Override
public void mouseEntered(java.awt.event.MouseEvent e) {
    lbl1.setForeground(Color.BLUE);
}
```

```
@Override
public void mouseExited(java.awt.event.MouseEvent e) {
    lbl1.setForeground(Color.BLACK);
    lbl2.setText("Fuera de la etiqueta");
}
```

```
@Override
public void mousePressed(java.awt.event.MouseEvent e) {
    lbl1.setForeground(Color.YELLOW);
}
```

```
@Override
public void mouseReleased(java.awt.event.MouseEvent e) {
    lbl1.setForeground(Color.BLUE);
}
```

```
}
}
```

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

Por supuesto, podríamos hacer la implementación de los Listeners directamente en la clase EventosRaton y ahorrarnos la clase MouseHandler, pero como ya se comentó, en el apéndice de adaptadores entenderemos la utilidad de esta.

Importante: Cuando se trabaja con eventos de ratón **directamente** en el **formulario**, hay que tener en cuenta que si añadimos el listener directamente al formulario (*this.addMouseListener(...)*) las coordenadas que nos da será desde el límite superior izquierda del formulario. Si lo que nos interesa es obtener coordenadas de la zona donde están los componentes, se debe añadir el listener al **Content Pane** (*this.getContentPane().addMouseListener(...)*).

En general los eventos suelen recaer mejor en el Content Pane por lo que otros como el *mouseExited* también se debe hacer sobre esta parte del formulario.

Algunas notas sobre detección teclas pulsadas junto con eventos de ratón:

1. Suponiendo que disponemos del parámetro *MouseEvent* *e*, podemos usar el método

```
e.isMetaDown()
```

En principio esta función devuelve si se ha presionado la tecla meta (tecla Windows en teclados PC y tecla Command en teclados Mac), pero también detecta la presión del botón derecho del ratón (uso habitual en ratones de un único botón como en Mac).

2. Dependiendo del sistema operativo en el que estéis puede ser que la tecla ALT no funcione si se usa sola, si no que funcione sólo si se pulsa junto a CTRL. Esto es porque el uso de ALT está reservado para acceder a los menús.

3. Uso del *e.getModifiers()* y *getModifiersEx()* que también devuelven información sobre teclas pulsadas o botones pulsados. Más libre de conflictos. Por ejemplo:

```
(e.getModifiers() & InputEvent.BUTTON1_MASK) == InputEvent.BUTTON1_MASK
```

es true si se pulsa el botón principal del ratón y

```
(e.getModifiers() & InputEvent.BUTTON3_MASK) == InputEvent.BUTTON3_MASK
```

es true si se pulsa el secundario.

getModifiers devuelve cambio de estado y *getModifiersEx* devuelve estado.

4. Existen una serie de funciones estáticas que encontramos en *SwingUtilities*. Por ejemplo:

```
public void mouseMoved(MouseEvent e) {
    System.err.println(SwingUtilities.isRightMouseButton(e));
}
```

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

Más información:

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

<https://stackoverflow.com/questions/16410228/what-is-the-difference-between-button1-mask-and-button1-down-mask>

<https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingUtilities.html>

Adaptadores

Cuando necesitamos gestionar un evento de ratón, se hace tedioso que en nuestro código existan un montón de líneas de cabeceras de funciones que no hacen nada. Por ejemplo, si solo necesitamos la entrada y la salida del ratón en un componente, tendríamos que poner todos los métodos del *MouseListener*.

Para evitar esto swing nos ofrece los adaptadores. Un adaptador es una clase que implementa el interface y completa los métodos pero vacíos. Así nosotros simplemente heredamos (no implementamos) el adaptador y sólo sobrescribimos aquellos elementos que nos interesen.

Hay varios adaptadores pero nos quedaremos con **MouseAdapter** para ratón y **KeyAdapter** para teclado. Veamos un ejemplo basado en *EventosRaton*:

```
private class MouseHandler extends MouseAdapter {

    @Override
    public void mouseEntered(java.awt.event.MouseEvent e) {
        lbl.setForeground(Color.BLUE);
    }

    @Override
    public void mouseExited(java.awt.event.MouseEvent e) {
        lbl.setForeground(Color.BLACK);
    }

}
```

Para probarlo sustituye la clase *MouseHandler* del ejemplo anterior por esta. Como se puede observar sólo se escriben los métodos que se necesitan cambiar.

Aquí se ve una de las utilidades de usar una clase interna (o embebida o inner class). No se puede usar el adaptador directamente sobre *EventosRaton* porque el adaptador **se hereda** no se implementa, y *EventosRaton* ya **hereda de JFrame** por lo que no podemos meter más herencia.

Se podría hacer en una clase externa normal, pero tendríamos que buscar la manera de pasar como parámetro el formulario o los componentes implicados en el evento, lo cual complicaría algo la estructura.

Si en un momento dado es necesario acceder al elemento *this* de la clase externa, no se puede hacer directamente ya que eso implicaría acceder al objeto de la clase interna. Para ello se usa el nombre de la

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

clase seguido de `.this`. Por ejemplo en el ejemplo anterior dentro de un método de la clase interna podría poner **`EventosRaton.this.lbl`**.

En cualquier caso no se deben usar alegremente las clases internas porque tienen otras implicaciones que no trataremos en este curso.

Eventos de teclado: `KeyListener`

De forma similar al apartado anterior veremos en este caso los eventos relacionados con la pulsación de las distintas teclas.

En este caso el interface a implementar es el `KeyListener` y sólo disponemos de tres métodos:

```
public void keyPressed(KeyEvent e) : Cuando es pulsada una tecla
cualquiera.

public void keyReleased(KeyEvent e) : Cuando es liberada una tecla
cualquiera.

public void keyTyped(KeyEvent e) : Cuando es pulsada una tecla que devuelve
un código Unicode. Es decir, no se ejecuta con teclas de acción como los
cursores, teclas de función, etc...
```

En cualquier pulsación de una tecla saltan los tres eventos. Los dos primeros son para controlar el código de teclado: si se pulsa la A, es la tecla A sin preocuparse de que sea mayúscula o minúscula. Esto se obtiene mediante `getKeyCode`. Para saber la tecla se usa las constantes en `KeyEvent` que empiezan por `VK`.

En `KeyTyped`, `getKeyCode` siempre devuelve 0.

Además puede saberse si simultáneamente se pulsa una tecla como CTRL con el método del evento **`isControlDown`** de forma similar a como se hacía con los eventos de ratón.

También se puede saber el Unicode mediante `getKeyChar`. Este es de hecho el único elemento que se puede obtener con el tercer evento y es el que se debe usar para dicho propósito.

Igual que para el caso del ratón tenemos como adaptador el `KeyAdapter` si no queremos escribir las tres funciones.

Veamos su funcionamiento con un ejemplo:

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

```

public class EventosTeclado extends JFrame implements KeyListener {
    JLabel lbl, lbl2;

    public EventosTeclado(){
        super("Eventos de teclado");
        setLayout(null);
        setFocusable(true);

        lbl=new JLabel("Tecla Pulsada: Ninguna");
        lbl.setSize(lbl.getPreferredSize().width,
                    lbl.getPreferredSize().height);
        lbl.setLocation(10,10);
        add(lbl);

        lbl2=new JLabel("Letra Seleccionada: Ninguna");
        lbl2.setSize(lbl2.getPreferredSize().width,
                    lbl2.getPreferredSize().height);
        lbl2.setLocation(10,40);
        add(lbl2);

        addKeyListener(this);
    }

    @Override
    public void keyPressed(KeyEvent e) {
        lbl.setText("Tecla Pulsada: "+e.getKeyCode());

        if (e.getKeyCode()==KeyEvent.VK_LEFT){
            lbl2.setLocation(lbl2.getX()-5, lbl2.getY());
        }

        if (e.getKeyCode()==KeyEvent.VK_RIGHT){
            lbl2.setLocation(lbl2.getX()+5, lbl2.getY());
        }

        if (e.isControlDown() && e.isShiftDown()){
            System.err.println("Además estás pulsando CTRL y SHIFT");
        }
    }

    @Override
    public void keyReleased(KeyEvent e) {
        lbl.setText("Tecla Liberada: "+e.getKeyCode());
    }

    @Override
    public void keyTyped(KeyEvent e) {
        lbl2.setText("Letra Seleccionada: "+e.getKeyChar());
    }
}

```

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

En este programa se añade los eventos al propio formulario, no a los componentes que hay en él. Esto es porque sólo hay *JLabel* que son componentes que no toman nunca el foco (no se vuelven activos). Si hubiera otro componente como el *TextField* que sí coge el foco, habría que añadir a dicho componente el mismo evento con la misma acción. Y así para todos los que tomen el foco.

Existen formas más óptimas de hacer esto (como los *KeyBindings*) pero no las veremos.

Si el control de teclas se desea hacer sobre un componente que habitualmente no puede coger el foco (como una *JLabel*), para que funcione hay que cambiar este comportamiento ejecutando el método del componente ***setFocusable(true)***. El parámetro indica que sí tomará el foco. En este caso lo usamos sobre el propio formulario por dicho motivo.

Más información:

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

<https://stackoverflow.com/questions/14301775/get-key-combinations/14301882>

Confirmar el cierre de una aplicación: *WindowListener*

En ocasiones interesa confirmar al usuario si está seguro de que desea cerrar una aplicación por distinto motivos. Por ejemplo es muy típico cuando se tiene un documento modificado y no se ha guardado.

Esta situación es muy fácil cuando hay un botón de Salir, una combinación de teclas o una opción del menú. Pero ¿Qué pasa si el usuario cierra la aplicación con el icono X de salida?

Existen diversas formas de controlar esta salida, pero quizá una muy sencilla es la de provocar nosotros el cierre de forma manual en lugar de forma automática tal y como lo estamos haciendo actualmente. Lo primero sería entonces lanzar la aplicación con el *setDefaultCloseOperation* de forma que no haga nada:

```
app.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

Lo siguiente sería preguntar al usuario si se desea cerrar o no la aplicación cuando pulsa el icono X de salida. Controlar esto es sencillo ya que existe un evento denominado *windowClosing* que se invoca al tratar de cerrar un formulario. En el método asociado a dicho evento debemos realizar la pregunta y si al final deseamos cerrar la aplicación hacerlo manualmente mediante el método *dispose()*.

Dicho evento se encuentra dentro del interface *WindowListener*, pero como dicho interface dispone de varios eventos, heredaremos el adaptador *WindowAdapter* para usar solo el *windowClosing* anteriormente mencionado.

En el constructor del formulario principal gestionamos el adaptador y el método que controla el evento:

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

```
addWindowListener(new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e) {
        int res = JOptionPane
            .showConfirmDialog(null, "Deseas cerrar el programa?",
                "Eventos Teclado", JOptionPane.YES_NO_OPTION,
                JOptionPane.QUESTION_MESSAGE);
        if (res == JOptionPane.OK_OPTION)
            e.getWindow().dispose();
    }
});
```

Fíjate en el uso de `getWindow()` en lugar de `getSource()` del evento para evitar realizar castings. Además `getWindow` devuelve un objeto tipo `Window`, no solo `JFrame`, lo que incluye además del formulario el gestor de eventos lo que hace que se cierre la aplicación. También se puede sustituir por un

`System.exit(0);`

Por supuesto también se podría gestionar el evento en una clase aparte (o interna) que herede de `WindowAdapter`. En este caso se define dicha clase por ejemplo de la siguiente forma:

```
class CierreVentana extends WindowAdapter {
    @Override
    public void windowClosing(WindowEvent e) {
        int res = JOptionPane.showConfirmDialog(null,
            "Deseas cerrar el programa?", "Eventos Teclado",
            JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE);
        if (res == JOptionPane.OK_OPTION)
            e.getWindow().dispose();
    }
}
```

Y en el constructor se añade el listener de forma más simple:

```
addWindowListener(new CierreVentana());
```

Si tienes que hacer esta tarea desde varios botones, menús, etc... puedes crear un método estático y llamarlo desde dónde haga falta:

```
static void exit(Window f){
    int res = JOptionPane.showConfirmDialog(f,
        "Deseas cerrar el programa?", "Eventos Teclado",
        JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE);
    if (res == JOptionPane.OK_OPTION)
        f.dispose();
}
```

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

NOTA: Ojo porque si hay timers u otros threads corriendo, el dispose de la ventana no los libera y la aplicación no acaba. En esos casos mejor realizar `System.exit(0)`; o parar previamente los timers.

Se pueden controlar otros eventos con el WindowListener como apertura de ventana; minimización, maximización y restauración de ventana,...

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

Si lo que buscas en controlar el tamaño de un componente se usa ComponentListener:
<https://docs.oracle.com/javase/tutorial/uiswing/events/componentlistener.html>

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

Más componentes

JTextArea

Componente similar al *TextField* pero que permite el uso de varias líneas de texto. No tiene evento asociado a *ActionListener* ya que al pulsar Enter se pasa a la línea siguiente. Al constructor más usual se le pasa el número de filas y de columnas que se desea que se muestren.

Otros métodos de interés:

setTabSize(): Establece el número de espacios que mide el tabulador.

setLineWrap(boolean): Establece si cuando se llega al final de línea se sigue en esa línea o se pasa a la línea siguiente.

setWrapStyleWord(boolean): Indica si al partir una línea lo hace en un espacio o en el lugar donde coincida.

append(String): Añade al final del texto la cadena que se pasa como parámetro.

insert(String, int) : Inserta la cadena en la posición indicada.

replaceRange(String, int, int): Sustituye el rango indicado por la cadena especificada en el parámetro.

Veamos un ejemplo con *FlowLayout*:

```
texto=new JTextArea(10,10);
add(texto);
```

Un problema que aparece en algunos layouts es que cuando el texto sobrepasa uno de los lados, el *TextArea* crece según sea necesario. Para evitar esto se puede usar (sin entrar en el funcionamiento del mismo) un objeto que permite gestionar barras de scroll, el *JScrollPane*. Realmente es una zona delimitada por el componente que se meta dentro de forma que si dicho componente crece el panel hace que aparezcan barras de scroll.

```
texto=new JTextArea(10,10);
scroll=new JScrollPane(texto);
add(scroll);
```

A continuación un ejemplo más completo en el que no se usa *Layout*:

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

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

```

public class AreaTexto extends JFrame implements ActionListener {
    JButton btnAnadir;
    JTextField txtAnadir;
    JTextArea texto;
    JScrollPane scroll;

    public AreaTexto() {
        super("Textos");
        setLayout(null);

        txtAnadir=new JTextField("Escribe aquí texto a añadir");
        txtAnadir.setLocation(10,10);
        txtAnadir.setSize(txtAnadir.getPreferredSize());
        txtAnadir.addActionListener(this);
        add(txtAnadir);

        btnAnadir=new JButton("Añadir");
        btnAnadir.setLocation(txtAnadir.getWidth()+20,10);
        btnAnadir.setSize(btnAnadir.getPreferredSize());
        btnAnadir.addActionListener(this);
        add(btnAnadir);

        texto=new JTextArea(100,100);
        texto.setTabSize(3);
        texto.setLineWrap(true);
        texto.setWrapStyleWord(true);
        scroll=new JScrollPane(texto);
        scroll.setLocation(10,txtAnadir.getHeight()+20);
        scroll.setSize(txtAnadir.getWidth()+20+btnAnadir.getWidth(), 100);
        add(scroll);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        texto.append(txtAnadir.getText());
    }
}

```

Si en el ejemplo anterior no se usara *JScrollPane*, habría que darle tanto tamaño (size) como posición(location) al objeto *JTextArea* y su constructor podría ser sin parámetros. Además en este caso dicho componente no crecería al escribir texto.

Para que funcionen bien los métodos de selección es necesario ejecutar el método *requestFocus()* o *requestFocusInWindow()* antes si no la primera vez no funciona. Tenlo en cuenta para la realización de los ejercicios.

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

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

JRadioButton (y ButtonGroup)

Elemento similar al *JCheckBox* pero que permite selección uno entre varios en vez de selección entre elementos independientes. Usa también el evento *ItemListener*. Para ello se tienen que agrupar los radio buttons deseados en un *ButtonGroup* para que funcionen como entidad lógica.

Veamos el funcionamiento con un ejemplo. Comienza escribiendo en el constructor ese código:

```
super("Botones de estado");
setLayout(new FlowLayout());

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

negro=new JRadioButton("Negro");
negro.setForeground(Color.BLACK);
negro.addItemListener(this);
add(negro);

rojo=new JRadioButton("Rojo");
rojo.setForeground(Color.RED);
rojo.addItemListener(this);
add(rojo);

arial=new JRadioButton("Arial");
arial.setFont(new Font("Arial", Font.PLAIN, 14));
arial.addItemListener(this);
add(arial);

serif=new JRadioButton("Serif");
serif.setFont(new Font("Serif", Font.PLAIN, 14));
serif.addItemListener(this);
add(serif);
```

Al ejecutarlo verás que cada *RadioButton* funciona de forma totalmente independiente al resto, de forma similar a como lo hace un *JCheckBox*. Agrupémoslos para que tengan un funcionamiento más correcto añadiendo las siguientes líneas al constructor:

```
grupoColor=new ButtonGroup();
grupoColor.add(negro);
grupoColor.add(rojo);

grupoFont=new ButtonGroup();
grupoFont.add(arial);
grupoFont.add(serif);
```


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

Algunos métodos:

getSelectedItem/getSelectedIndex: devuelve el ítem/índice del elemento seleccionado.

setSelectedIndex: fuerza la selección a determinado índice (los índices empiezan en 0).

addItem(item): Añade un ítem al final de la lista.

insertItem(item, posicion): inserta un ítem en la posición indicada.

removeAllItems: Elimina todos los ítems.

removeItemAt(posicion): Elimina ítem de la posición indicada.

removeItem(item): Elimina ítem indicado.

getItemCount: Devuelve la cantidad de ítems en la colección.

setEditable(boolean): indica si se puede o no editar la posición principal. En caso de que se pueda editar, se puede obtener el valor seleccionado mediante *getSelectedItem* cuando el componente pierde el foco. Además el índice en ese caso será -1 ya que el elemento no pertenece a la lista.

Veamos un ejemplo:

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

public class ComboBox extends JFrame implements ItemListener, ActionListener {
    JComboBox<String> cbMarcas;
    JButton btnAnadir;
    JTextField txtMarca;
    JLabel lblSeleccion;

    String[] marcasInfo = { "AMD", "Apple", "Canonical", "IBM",
                            "Microsoft", "Novell", "Oracle" };

    public ComboBox() {
        super("Lista desplegable");
        setLayout(new FlowLayout());

        cbMarcas = new JComboBox<String>(marcasInfo);
        cbMarcas.setMaximumRowCount(8);
        cbMarcas.setSelectedIndex(3);
        cbMarcas.addItemListener(this);
        add(cbMarcas);

        txtMarca = new JTextField(20);
        txtMarca.addActionListener(this);
        add(txtMarca);
    }
}
```

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

```

btnAnadir = new JButton("Añadir");
btnAnadir.addActionListener(this);
add(btnAnadir);

lblSeleccion = new JLabel(String.format(
    "Elemento seleccionado: %s, Índice: %d",
    cbMarcas.getSelectedItem(),
    cbMarcas.getSelectedIndex()));
add(lblSeleccion);

```

```

}

```

```

@Override

```

```

public void itemStateChanged(ItemEvent arg0) {
    lblSeleccion.setText(String.format(
        "Elemento seleccionado: %s, Índice: %d",
        cbMarcas.getSelectedItem(),
        cbMarcas.getSelectedIndex()));
}

```

```

}

```

```

@Override

```

```

public void actionPerformed(ActionEvent e) {
    if (!txtMarca.getText().trim().equals("")) {
        cbMarcas.addItem(txtMarca.getText());
    }
}

```

```

}

```

```

}

```

Entendiendo la explicación anterior este programa no debería tener pegas.

Lo que sí se debe tener en cuenta es que cada vez que se selecciona un elemento del combobox, se lanza dos veces el evento y por tanto la función *itemStateChanged*. Esto es porque un elemento se deselecta y otro se selecciona. Para diferenciar esto se pueden comprobar mediante *ItemEvent.SELECTED* y *ItemEvent.DESELECTED* lo que está sucediendo. Prueba a añadir las siguientes líneas al *itemStateChanged*:

```

if (arg0.getStateChange() == ItemEvent.SELECTED) {
    System.err.println("Selecciono "+arg0.getItem());
}

if (arg0.getStateChange() == ItemEvent.DESELECTED) {
    System.err.println("Deselecciono "+arg0.getItem());
}

```

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

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

JPanel

Clase que permite reorganizar los componentes por grupos dentro de un formulario. Es un contenedor y por tanto tiene su propio layout. Esto permite tener un layout en el *JFrame* y otro distinto en el panel para conseguir distintas posibilidades de colocación y organización de componentes.

El *JPanel* por defecto tiene un *FlowLayout* si no se especifica nada.

Veamos un ejemplo sencillo. Sustituye el constructor hecho en la clase *ComboBox* por el siguiente:

```
super("Lista desplegable");
setLayout(null); //Prueba aquí también un FlowLayout

panel1=new JPanel();
panel1.setSize(300,100);
panel1.setLocation(10,10);
add(panel1);

panel2=new JPanel();
panel2.setSize(300,100);
panel2.setLocation(10,140);
add(panel2);

cbMarcas = new JComboBox<String>(marcasInfo);
cbMarcas.setMaximumRowCount(8);
cbMarcas.setSelectedIndex(3);
cbMarcas.addItemListener(this);
panel1.add(cbMarcas);

btnAnadir = new JButton("Añadir");
btnAnadir.addActionListener(this);
panel2.add(btnAnadir);

txtMarca = new JTextField(20);
txtMarca.addActionListener(this);
panel2.add(txtMarca);

lblSeleccion = new JLabel(String.format(
    "Elemento seleccionado: %s, Índice: %d",
    cbMarcas.getSelectedItem(),
    cbMarcas.getSelectedIndex()));
panel1.add(lblSeleccion);
```

Analiza el código pues es fácil de comprender.

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

Creación dinámica de componentes

Cuando hay que crear una gran cantidad de componentes similares ir haciendo uno a uno es bastante tedioso y además se está repitiendo un montón de código de forma innecesaria. Por este motivo se recomienda crear dichos componentes mediante uno o varios bucles de forma que se simplifica mucho la labor.

A continuación vemos un ejemplo sencillo en el que se crean varios botones de forma dinámica y mediante la aplicación de polimorfismo en el *actionPerformed* se el resultado es distinto según el botón pulsado.

```
public class Botones extends JFrame implements ActionListener{

    private String[] programadores = { "Ada Lovelace", "Denis Ritchie",
        "Grace Hopper", "Tim Berners Lee",
        "Margaret Hamilton", "Sheldon Cooper" };

    public Botones() {
        super();
        setLayout(null);

        int x = 10, y = 10;
        for (int i = 0; i < programadores.length; i++) {
            JButton b = new JButton(programadores[i]);
            b.setSize(200, 30);
            b.setLocation(x, y);
            b.addActionListener(this);
            this.add(b);
            if ((i + 1) % 3 == 0) {
                x = 10;
                y += 40;
            } else {
                x += 210;
            }
        }
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(this,
            "Ha seleccionado "+((JButton)e.getSource()).getText());
    }
}
```

Los puntos a tener en cuenta de lo anterior:

- La colocación según las coordenadas. En este caso se hacen filas de 3 columnas. Por supuesto esto se puede hacer de diversas formas, por ejemplo con un doble bucle donde uno cuente filas y otro columnas. Pero esta aquí presentada es una forma bastante sencilla y cómoda.

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

- El valor interno de cada componente: Si dichos valores son muy distintos como en este caso (nombres de personas) estos deben venir de un array, colección o incluso de un archivo.

Hay ciertos casos donde el valor es similar y se puede establecer automáticamente (por ejemplo si hay que poner números en los componentes).

El dato no se limita al texto, pueden ser también colores, tamaño u otras propiedades. Cualquiera de estas se puede meter en un array/colección o archivo.

- En el `actionPerformed` para detectar el componente simplemente lo vemos mediante `getSource()`, y en este caso como sabemos que son todos botones pues no tenemos que hacer comprobación para realizar el casting.

En el apéndice II aparecen otros ejemplos de creación dinámica. Es particularmente interesante el primero que ayuda de forma muy cómoda a crear menús muy grandes usando arrays.

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

Uso de varios formularios en una aplicación

Muchas aplicaciones tienen un único formulario para realizar todas las acciones, sin embargo, en ocasiones es necesario que una aplicación saque ventanas secundarias con distintos objetivos: pedir una serie de datos, mostrar información, crear una barra de herramientas, etc...

Para realizar esto llega con poner en práctica distintos conceptos vistos en este curso ya que cualquier formulario es una clase, y que desde un formulario principal se muestra un secundario no es más que crear el objeto a partir de la clase del secundario y luego lanzarlo de forma similar a como lo hacemos con el principal.

Sin embargo, por seguir el estándar clásico de Swing no usaremos la clase *JFrame* como clase base para formularios secundario. Usaremos *JDialog* que nos da más versatilidad (además de no aparecer como aplicación nueva en una barra de tareas).

Y si lo que se necesita es acceder desde el secundario al principal se necesita un constructor en el que se le pase el formulario principal para tener acceso al mismo.

Lo que cambia es el *setDefaultCloseOperation* ya que ahora no queremos que se salga de la aplicación si se cierra el secundario. Las opciones disponibles:

DO_NOTHING_ON_CLOSE: Si no se hace nada al pulsar el icono de cierre tendremos que gestionar por programación (evento *windowClosing*) lo que deseamos que suceda.

HIDE_ON_CLOSE: Se oculta (no visible) al pulsar el cierre. Es la opción por defecto. Si se deja esto para volver a mostrarlo no es necesario volver a crearlo, llega con realizar *setVisible(true)*.

DISPOSE_ON_CLOSE: Libera todos sus recursos al pulsar cierre. Si se configura así el principal, tiene el mismo efecto que *EXIT_ON_CLOSE*.

EXIT_ON_CLOSE : Sale de la aplicación usando *System.exit(0)*. Recomendable sólo para ventana principal de aplicaciones.

Teniendo esto en cuenta veamos un Ejemplo en el que se crean dos clases, Principal (*JFrame*) es la ventana de la aplicación y Secundario (*JDialog*) un formulario que se lanza desde el principal. En el secundario se muestra información del sistema que se obtiene en el principal y mediante el secundario podemos cambiar el título del formulario principal.

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

Formulario Principal:

```

public class Principal extends JFrame implements ActionListener {
    JButton btn;

    public Principal() {
        super("Ventana principal");
        setLayout(new FlowLayout());

        btn = new JButton("Información");
        btn.addActionListener(this);
        add(btn);
    }

    @Override
    public void actionPerformed(ActionEvent arg0) {
        Secundario f = new Secundario(this);
        f.lblInfo.setText("<html><b>Sistema: "
            + System.getProperty("os.name")
            + "<br>Directorio de almacenamiento temporal:"
            + System.getProperty("java.io.tmpdir") + "<br>Usuario:"
            + System.getProperty("user.name") + "</b></html>");
        f.pack(); // Tamaño necesario para ver todos los componentes
        f.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        f.setVisible(true);
    }
}

```

Formulario secundario:

```

public class Secundario extends JDialog implements ActionListener {
    JLabel lblInfo, lblTitulo;
    JTextField txtTitulo;

    public Secundario(Principal f) {
        super(f);
        setLayout(new FlowLayout());
        setTitle("Ventana secundaria");

        lblInfo = new JLabel();
        add(lblInfo);

        lblTitulo = new JLabel(
            "Introduce nuevo título para el formulario principal");
        add(lblTitulo);
    }
}

```

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

```

txtTitulo = new JTextField(10);
txtTitulo.addActionListener(this);
add(txtTitulo);
}

@Override
public void actionPerformed(ActionEvent e) {
    Principal f=(Principal)this.getOwner();
    f.setTitle(txtTitulo.getText());
}
}

```

Algunos comentarios:

- El secundario podría tener varias sobrecargas según nos interese pasarle otros parámetros como el título o no pasarle ninguno si no tenemos que acceder al principal.
- Conviene inicializar los elementos que queramos del secundario antes de mostrarlo pero, lógicamente, siempre después de crearlo (new).
- Se realiza un *DISPOSE_ON_CLOSE* para liberar los recursos del secundario pero sin cerrar la aplicación. Otra opción muy habitual es *HIDE_ON_CLOSE*.
- El método *pack()* da el tamaño adecuado al formulario para que quepan todos los componentes (similar a un *preferredSize*).
- Hay que tener en cuenta que si el principal necesita acceder a componentes del secundario (o viceversa) estos no pueden ser privados.
- Mediante *getOwner* se recoge el formulario que se pasa como parámetro al constructor (mediante el *super(f)* del constructor). En este caso es el principal.
- Al pasar mediante *super* el formulario principal a la clase padre se hace propietario y se “enlaza”. Por ejemplo si se mueve el principal se mueve el secundario.
- *System.getProperty()* Es una llamada a una función estática que mediante una clave hash informa de distintos aspectos (variables de entorno y otros) de la máquina donde se está ejecutando el programa y que me puede facilitar la labor.

Puedes ver una lista de propiedades estándar en:

[http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#getProperties\(\)](http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#getProperties())

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

Formularios secundario Modales

Si has probado el caso anterior verás que tiene un pequeño problema y es que cada vez que pulso el botón del formulario principal lanza un secundario. Esto puede controlarse fácilmente desactivando el botón o mostrando y ocultando el formulario secundario según sea de nuestro interés.

Pero hay ocasiones en que lo que interesa ya no es sólo evitar que salgan varios formularios secundarios si no que mientras el formulario secundario esté activo que no se tenga acceso al principal.

Este funcionamiento ya lo conocemos de componentes como el *JOptionPane* o el *JFileChooser*, pero veamos cómo extrapolarlo a un formulario secundario cualquiera diseñado por nosotros.

Este comportamiento se consigue simplemente pasándole un booleano a *true* como parámetro al constructor del *JDialog* o a alguna de sus sobrecargas. Realiza estos cambios en el programa anterior para entenderlo:

El constructor del **secundario** empezará así:

```
public Secundario(Principal f) {
    super(f, true); // true: modal; false: no modal
    ...
}
```

Finalmente conviene cambiar la forma de cierre:

```
f.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
```

Esto se debe a que en ocasiones queremos recoger información del secundario y corremos el riesgo de que los recursos ya estén liberados si hacemos un *DISPOSE_ON_CLOSE*.

Si ahora lo pruebas verás que el funcionamiento es tal cual lo describimos más arriba.

Para rematar el aspecto visual opcionalmente se puede poner en el *actionPerformed* del **secundario**:

```
setVisible(false);
```

con el objetivo de que el formulario secundario desaparezca al meter el dato.

Si lo que deseas es además gestionar una respuesta a un diálogo, se puede programar tal y como lo indica en este enlace:

<https://stackoverflow.com/a/4089370/2586768>

Más información sobre *JDialog*: <http://docs.oracle.com/javase/6/docs/api/javax/swing/JDialog.html>

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

Notación usada para los componentes

Como ya se recomendó en alguna ocasión, los nombres de variables (objetos) que representan a los componentes deben, por claridad, tener en el nombre indicativo tanto del tipo de componente como de su utilidad. Por ejemplo un botón de Cancelar lo denominaríamos btnCancelar.

Esto es lo que se llama notación húngara y puede ser extrapolado a cualquier tipo de variable, pero puede llegar a provocar un código complejo de leer. Sin embargo es muy útil en el caso de componentes Swing ya que viendo el nombre sabemos qué es y su utilidad.

De esta forma la regla que usaremos es la de usar 3 caracteres para el tipo y luego una o más palabras que indiquen la utilidad. Por regla general los 3 caracteres del tipo serán quitando la J inicial, las vocales y consonantes repetidas.

De esta forma los nombres de los objetos swing empezarán por:

*JLabel: **lbl***

*JButton: **btn***

*JComboBox: **cmb***

*JList: **lst***

*JPanel: **pnl***

*JCheckBox: **chk***

*JRadioButon: **rdb***

*JPasswordField: **psw***

*JTimer: **tmr***


Otros que por claridad nos saltaremos el estándar en los siguientes casos:

*TextField: **txf***

*TextArea: **txa***

*ButtonGroup: **grp***

*Todos los items de menú: **mnu***

	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.

Recursos Web:

<http://docs.oracle.com>

<http://stackoverflow.com>

<http://jungla.dit.upm.es/~santiago/docencia/apuntes/Swing/componentes.htm>

Aprende a programar un juego en Java:

<https://www.youtube.com/playlist?list=PLN9W6BC54TJJr3erMptodGOQFX7gWfKTM>

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

Apéndice I: gestión de JRadioButton/ButtonGroup

La gestión de estos componentes puede ser muy variada dependiendo del uso que se les quiera dar, debido a esto y por no alargar el tema se deja en este apéndice otras formas de trabajo con JRadioButton. Se parte del ejemplo usado.

En el constructor:

```
super("Botones de estado");
setLayout(new FlowLayout());

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

negro=new JRadioButton("Negro");
negro.setForeground(Color.BLACK);
negro.addItemListener(this);
add(negro);

rojo=new JRadioButton("Rojo");
rojo.setForeground(Color.RED);
rojo.addItemListener(this);
add(rojo);

arial=new JRadioButton("Arial");
arial.setFont(new Font("Arial", Font.PLAIN, 14));
arial.addItemListener(this);
add(arial);

serif=new JRadioButton("Serif");
serif.setFont(new Font("Serif", Font.PLAIN, 14));
serif.addItemListener(this);
add(serif);

grupoColor=new ButtonGroup();
grupoColor.add(negro);
grupoColor.add(rojo);

grupoFont=new ButtonGroup();
grupoFont.add(arial);
grupoFont.add(serif);

negro.setSelected(true);
arial.setSelected(true);
```

Vimos que mediante el parámetro en itemStateChanged y su propiedad getSource se puede comprobar que ítem se ha seleccionado. Veamos que más podemos hacer.

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

Obtener el grupo mediante getModel()

Otra posibilidad es obtener el elemento seleccionado del grupo. Esto se complica ligeramente porque hay que acceder a través del interface ButtonModel que permite realizar ciertas tareas a todos los que lo implementen. Para obtener el ButtonModel de un botón se usa getModel, una vez que lo tengo, puedo acceder al grupo en el que se encuentra el JRadioButton. Usando este método podemos hacer lo mismo que en el caso visto en el tema:

```
JRadioButton rb=(JRadioButton)e.getSource();
ButtonGroup grupo=((DefaultButtonModel)rb.getModel()).getGroup();
if (grupo==grupoColor)
    lbl.setForeground(rb.getForeground());
else
    lbl.setFont(rb.getFont());
```

Esto resultará cómodo si tenemos muchos elementos dentro de cada grupo para que las comparaciones no queden tan largas.

Usar distintos Listener por cada grupo

También se puede disponer de un Listener por cada grupo. Para ello se crean dos clases que implementen el listener de la siguiente forma.

```
class AccionFuentes implements ItemListener{
    @Override
    public void itemStateChanged(ItemEvent e) {
        System.err.println("Fuentes");
    }
}

class AccionColores implements ItemListener{
    @Override
    public void itemStateChanged(ItemEvent e) {
        System.err.println("Colores");
    }
}
```

En los componentes, en lugar de añadir this, se añade un objeto de estas nuevas clases. Por ejemplo:

```
arial.addItemListener(new AccionFuentes());
rojo.addItemListener(new AccionColores());
```

Uso de ItemEvet.SELECTED

Si pruebas el código anterior, verás que cada vez que pulsas en un radiobutton el mensaje aparece dos veces. Esto es porque el evento se lanza dos veces: una al deseleccionar uno de las JRadioButton y otro al

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

seleccionar el nuevo. Para que una acción se realice solo una vez se puede comprobar que es un evento de selección (SELECTED) o de deselección (DESELECTED).

```
public void itemStateChanged(ItemEvent e) {
    if (e.getStateChange() == ItemEvent.SELECTED) {
        System.err.println("Colores");
    }
}
```

Guardar el radiobutton seleccionado

En muchos casos es necesario quedarse con el radiobutton marcado. Para eso lo ideal es utilizar una variable auxiliar en la que se guarda de alguna forma el radiobutton seleccionado cuando se produce el evento.

```
String colorSeleccionado;

@Override
public void itemStateChanged(ItemEvent e) {
    if (e.getStateChange() == ItemEvent.SELECTED) {
        colorSeleccionado = ((JRadioButton) e.getSource()).getText();
        System.err.println(colorSeleccionado);
    }
}
```

Mostrar el estado de los radiobuttons

Como curiosidad y experimento final se indica a continuación como mostrar el estado de todos los radiobuttons. Una posibilidad es meterlos en un vector y recorrer dicho vector. Pero si son muchos radiobuttons hacer este vector puede ser incómodo. En ese caso lo ideal es recorrer la colección de componentes y detectar los que son radiobutton.

Para ello tenemos que recorrer la colección de componentes que está en la propiedad *ContentPane* (Ojo, no directamente en *this*).

Veámos esta posibilidad con un ejemplo. Añade un *JButton* al constructor e implementa el interface *ActionListener*. La cabecera de la clase y definición del botón deben quedar así:

```
public class Botones extends JFrame implements ItemListener, ActionListener {
    JButton btn;
```

...

El constructor añades las líneas de inicialización del botón:


```
btn=new JButton("Mostrar");
btn.addActionListener(this);
add(btn);
```

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

Finalmente codificas el método *actionPerformed*:

```
@Override
public void actionPerformed(ActionEvent arg0) {
    String seleccionados="";
    for (Component c : this.getContentPane().getComponents())
        if (c.getClass()==JRadioButton.class)
            if (((JRadioButton)c).isSelected())
                seleccionados+=((JRadioButton)c).getText()+"\n";
    JOptionPane.showMessageDialog(this, seleccionados);
}
```

Este código no es complicado de entender y lo que hace es mostrar en un cuadro de diálogo los elementos seleccionados.

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

Apéndice II: Otros componentes

JPasswordField

Componente que hereda de *JTextField* con el objetivo de que lo que está escribiendo el usuario no se vea ya que está tapado por un carácter de echo predefinido; es decir, en lugar de mostrar lo que escribe el usuario presenta un único carácter repetido en pantalla. Añade los siguientes métodos interesantes:

char[] getPassword: Obtiene un array de caracteres con la password. Se puede usar *getText* pero se considera obsoleto (*deprecated*).

setEchoChar(char): establece un nuevo carácter echo en lugar del que viene por defecto.

En el ejemplo anterior añade al final del constructor:

```
lbl2 = new JLabel("Introduzca su contraseña");
add(lbl2);

psw=new JPasswordField(10);
psw.setEchoChar('$');
add(psw);
```

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

JList

Este elemento muestra una lista de ítems de los cuales se pueden elegir uno o varios dependiendo de la situación. En principio es muy similar al combobox visto, pero usa métodos distintos y es más versátil aunque menos compacto visualmente.

El *JList* tiene un constructor que permite inicializar el componente con una lista de ítems al igual que el combobox.

También está parametrizado desde la versión Java 7, por lo que lo inicializaremos con `<String>` para indicar que metemos cadenas (se puede usar cualquier objeto como en el *JComboBox*).

El interface de escucha de eventos es en este caso el *ListSelectionListener* que dispone del método *valueChanged* que se provoca cuando cambia algo en la selección de elementos.

Además de la lista de ítems que se inicializa en el constructor, es habitual al inicializar el componente indicar el modo de selección mediante *setSelectionMode*. Dispone de tres modos:

ListSelectionModel.SINGLE_SELECTION: Sólo permite seleccionar un elemento.

ListSelectionModel.SINGLE_INTERVAL_SELECTION: Varios elementos a la vez pero contiguos.

ListSelectionModel.MULTIPLE_INTERVAL_SELECTION: Varios elementos. Es el modo por defecto

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

Otros métodos de interés:

int getSelectedIndex(): Devuelve el índice del primer elemento seleccionado.

int[] getSelectedIndices(): Devuelve un vector con los índices de los elementos seleccionados.

Object getSelectedValue(): Devuelve el primer elemento seleccionado.

Object[] getSelectedValuesList(): Devuelve un vector con los elementos seleccionados.

boolean isEmpty(): Indica si no hay nada seleccionado (true).

Veamos un ejemplo de uso de un *JList*:

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

public class Lista extends JFrame implements ListSelectionListener {
    JList<String> lst;
    JLabel lblSeleccion;

    String[] nombresDeColores = { "Rojo", "Verde", "Azul", "Amarillo", "Blanco",
    "Negro", "Gris" };
    Color[] colores = { Color.RED, Color.GREEN, Color.BLUE, Color.YELLOW, Color.WHITE,
    Color.BLACK, Color.GRAY };

    public Lista() {
        super("listas de selección");
        setLayout(new FlowLayout());

        lst = new JList<String>(nombresDeColores);
        lst.setVisibleRowCount(5);
        lst.addListSelectionListener(this);
        add(new JScrollPane(lst));

        lblSeleccion = new JLabel("Elementos seleccionados: Ninguno");
        add(lblSeleccion);
    }

    @Override
    public void valueChanged(ListSelectionEvent arg0) {
        String s = "Elementos seleccionados: ";
        if (lst.isEmpty())
            s += "Ninguno";
        else
            for (Object item : lst.getSelectedValuesList()) {
                s += (String) item + " ";
            }
    }
}
```

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

```

lblSeleccion.setText(s);

if (lst.getSelectedIndex() >= 0 && lst.getSelectedIndex() < colores.length)
    this.getContentPane().setBackground(colores[lst.getSelectedIndex()]);
}
}

```

En el caso de este evento, que también se lanza dos veces cuando se selecciona un ítem, pues es una cadena de eventos: *mouseDown* , Dragmientras arrastro y al final *mouseUp*. Para evitar esto se usa *getValueIsAdjusting* que indica estando a *true* cuando no es el evento final de una serie de ellos. Por tanto el uso sería así:

```

if (!arg0.getValueIsAdjusting())
    System.err.println("Evento valueChanged");

```

Esto se explica con más detalle en: <https://stackoverflow.com/a/12461895/2586768>

Este componente no permite añadir nuevos ítems directamente. La solución pasa por utilizar una clase auxiliar denominada *DefaultListModel*. Esta clase mantiene una colección interna y se enlaza con el *JList* en el constructor.

Dispone de funciones para añadir y eliminar elementos de su lista interna de forma que automáticamente son presentados y eliminados de la *JList* con la que está enlazada lo que la hace muy versátil. Veamos un ejemplo de inicialización con esta clase:

```

DefaultListModel<String> modelo=new DefaultListModel<String>();

lst=new JList(modelo);
lst.setVisibleRowCount(5);
lst.addListSelectionListener(this);
add( new JScrollPane(lst));

for (int i=0; i<10; i++)
    modelo.addElement("item "+i);

```

El código anterior se introduce dentro del constructor teniendo en cuenta que *modelo* es una propiedad de la clase.

Por supuesto, si se desea que el modelo sea accesible desde distintos métodos, debe ser declarado fuera del constructor (como el resto de los componentes).

Mediante el método *addElement* se añaden elementos uno a uno. Para eliminarlos se puede usar *removeElementAt(indice)* o *removeElement(elemento)*. Dispone también de otros métodos que se pueden ver en:

<http://docs.oracle.com/javase/7/docs/api/javax/swing/DefaultListModel.html>

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

Existen otros métodos de apariencia y manejo de las listas en la propia clase.

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

Menús

Para crear una estructura de menús en una aplicación se usan varias clases:

JMenuBar: es la clase que gestiona la barra principal de menús. A su constructor no le pasaremos parámetros. Para añadir el menú al *JFrame* llega con añadir esta barra, pero se hará a través de la función *setJMenuBar* del *JFrame* para que aparezca como menú principal (en el Layered Pane) y no como un componente más.

JMenu: cada una de las opciones que hay en la barra principal. A su constructor le pasaremos el texto que queremos que aparezca.

JMenuItem: Cada uno de los elementos de un menú. Se comporta de forma prácticamente idéntica a un *JButton*, de hecho hereda también de *AbstractButton*. Existen también, *JRadioButtonMenuItem* y *JCheckBoxMenuItem* para disponer de ese tipo de botones en los menús. A su constructor le pasaremos el texto que queremos que aparezca.

Como cada elemento del menú *JMenuItem* es un botón, lo visto para *JButton* es válido aquí: como colocar imágenes, controlar eventos, etc...

A los métodos ya conocidos de *JButton* cabe mencionar alguno más:

add: Este método ya conocido sirve en un *JMenuItem* para añadir un submenú. En un *JMenu* para añadir un *JMenuItem* y en un *JMenuBar* para añadir un *JMenu*.

addSeparator(): En objetos de la clase *JMenu* sirve para añadir una línea separadora.

setMnemonic(char): En objetos que hereden de *AbstractButton* como *JMenuItem* sirve para indicar la tecla de acceso mediante pulsación de ALT. Dicha letra aparece subrayada. También es válido por herencia para *JButton*, *JRadioButton* y *JCheckBox*.

Se puede, y es muy típico, ir creando opciones de menú por separado y añadir en cada *addActionListener* el método que gestiona el evento tal cual se ve en el siguiente ejemplo:

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

public class Menu extends JFrame implements ActionListener {
    JMenuBar mnuPrincipal;
    JMenu mnuArchivo, mnuEdicion, mnuEstilo;
    JMenuItem mnuAbrir, mnuGuardar, mnuSalir, mnuCopiar, mnuNegrita;
    JLabel lbl;

    public Menu() {
        super("Menú simple");
        setLayout(new FlowLayout());
    }
}
```

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

//Menú Archivo

```
mnuAbrir = new JMenuItem("Abrir");
mnuAbrir.setMnemonic('A');
mnuAbrir.addActionListener(this);
```

```
mnuGuardar = new JMenuItem("Guardar");
mnuGuardar.setMnemonic('G');
mnuGuardar.addActionListener(this);
```

```
mnuSalir = new JMenuItem("Salir");
mnuSalir.setMnemonic('S');
mnuSalir.addActionListener(this);
```

```
mnuArchivo = new JMenu("Archivo");
mnuArchivo.setMnemonic('A');
mnuArchivo.add(mnuAbrir);
mnuArchivo.add(mnuGuardar);
mnuArchivo.addSeparator();
mnuArchivo.add(mnuSalir);
```

//Menú Edición

```
mnuNegrita = new JMenuItem("<html><b>Negrita</b></html>");
mnuNegrita.addActionListener(this);
mnuNegrita.addActionListener(this);
```


```
mnuEstilo = new JMenu("Estilo");
mnuEstilo.setMnemonic('E');
mnuEstilo.add(mnuNegrita);
mnuEstilo.addActionListener(this);
```

```
mnuCopiar = new JMenuItem("Copiar");
mnuCopiar.setMnemonic('C');
mnuCopiar.addActionListener(this);
```

```
mnuEdicion = new JMenu("Edicion");
mnuEdicion.setMnemonic('E');
mnuEdicion.add(mnuEstilo);
mnuEdicion.addSeparator();
mnuEdicion.add(mnuCopiar);
```

//Menú Principal

```
mnuPrincipal = new JMenuBar();
mnuPrincipal.add(mnuArchivo);
mnuPrincipal.add(mnuEdicion);
this.setJMenuBar(mnuPrincipal);
```

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

```

        lbl = new JLabel();
        add(lbl);
    }

```

```

@Override
public void actionPerformed(ActionEvent e) {
    lbl.setText(e.getActionCommand());
}

```

Prueba a cambiar el método *setJMenuBar* por *add* y comprueba lo que sucede: colocas el menú en el Content Pane en lugar de hacerlo en el Layered Pane que es su sitio.

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

Apéndice III: Otros ejemplos de creación dinámica

Cuando el menú es muy grande, es obvio que el código anterior crece notablemente. Una posibilidad es hacer funciones de inicialización de los distintos *JMenuItem*. Otra es meter las opciones de menús y los mnemónicos (y otros elementos como imágenes, etc...) en arrays (¡o archivos para internacionalización!) y crear el menú mediante bucles. Un ejemplo podría ser este:

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

public class Menu extends JFrame implements ActionListener {
    JMenuBar mnuPrincipal;
    JLabel lbl;

    String[] menus = { "Archivo", "Editar", "Ayuda" };
    char[] mnMenus = { 'A', 'E', 'Y' };
    String[][] submenus = {
        { "Nuevo", "Abrir", "Guardar", "Guardar como...", "-", "Imprimir",
          "Vista previa...", "-", "Salir" },
        { "Copiar", "Cortar", "Pegar", "-", "Buscar", "Reemplazar" },
        { "Indice", "Buscar", "On line" } };

    char[][] mnSubmenus = { { 'N', 'A', 'G', 'u', '-', 'I', 'V', '-', 'S' },
                           { 'C', 'o', 'P', '-', 'B', 'R' },
                           { 'I', 'B', 'O' } };

    private JMenuItem item(String tit, char mn) {
        JMenuItem mnu = new JMenuItem(tit);
        mnu.setMnemonic(mn);
        mnu.addActionListener(this);
        return mnu;
    }

    public Menu() {
        super("Menú Grande");
        setLayout(new FlowLayout());

        mnuPrincipal = new JMenuBar();
        this.setJMenuBar(mnuPrincipal);

        for (int i = 0; i < submenus.length; i++) {
            JMenu m = new JMenu(menus[i]);
            m.setMnemonic(mnMenus[i]);
            for (int j = 0; j < submenus[i].length; j++) {
                if (submenus[i][j].equals("-")) {
                    m.addSeparator();
                }
            }
        }
    }
}
```

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

```

        } else {
            JMenuItem mi=item(submenus[i][j],mnSubmenus[i][j]);
            m.add(mi);
        }
    }
    mnuPrincipal.add(m);
}

lbl = new JLabel();
add(lbl);
}

@Override
public void actionPerformed(ActionEvent e) {
    lbl.setText(e.getActionCommand());
}
}

```

Fíjate que para saltarse algunos elementos como separadores llega con meter cierto carácter en los arrays para luego comprobarlo y meter el separador en el lugar correspondiente.

Cuando un ítem de menú comparta acción con otro botón, se debe usar o el mismo *ActionListener* o ejecutar un *doClick* del botón en el *ActionListener* particular.

Evidentemente aún así la construcción de menús es tediosa salvo que se disponga de un editor gráfico, herramienta que facilita en gran medida la creación de estos elementos imprescindibles en cualquier aplicación.

Lo realizado en el ejemplo anterior no es exclusivo para menús. Cuando se dispone de una serie de componentes similares y en una cantidad considerable conviene plantearse si es mejor crearlos mediante un bucle en lugar de uno a uno.

En el ejemplo que ves a continuación se establece jugando con herencia, clases internas y creación dinámica de componentes la elección de elementos entre dos grupos de radiobuttons. Deberías entender el código sin ningún problema:

```

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

// Ampliamos la clase JRadioButton a nuestro antojo
class JRadioButtonEntero extends JRadioButton {
    private int numero;

    public int getNumero() {
        return numero;
    }
}

```

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

```

public void setNumero(int numero) {
    this.numero = numero;
    this.setText(numero + "");
}
}


public class RadioButtons extends JFrame implements ActionListener {
    private JRadioButton[] rbProgramadores;
    private JRadioButtonEntero[] rbNivel;
    private ButtonGroup grpProgramadores, grpNiveles;
    private String[] programadores = { "Ada Lovelace", "Denis Ritchie", "Grace Hopper",
                                        "Tim Berners Lee", "Margaret Hamilton", "Sheldon
Cooper" };
    private JButton aceptar;
    private String programadorSeleccionado = "";
    private int nivelSeleccionado = 0;

    public RadioButtons() {
        super("Ejemplo Radiobuttons dinámicos");
        setLayout(null);

        // Creación de radiobuttons de programadores
        rbProgramadores = new JRadioButton[programadores.length];
        grpProgramadores = new ButtonGroup();
        int x = 10, y = 10;
        for (int i = 0; i < rbProgramadores.length; i++) {
            rbProgramadores[i] = new JRadioButton(programadores[i]);
            rbProgramadores[i].setSize(rbProgramadores[i].getPreferredSize());
            rbProgramadores[i].setLocation(x, y);
            x += 180;
            if ((i + 1) % 3 == 0) {
                x = 10;
                y += 30;
            }
            rbProgramadores[i].addItemListener(new ItemListener() {
                @Override
                public void itemStateChanged(ItemEvent ie) {
                    programadorSeleccionado = ((JRadioButton) ie.getSource()).getText();
                }
            });
            add(rbProgramadores[i]);
            grpProgramadores.add(rbProgramadores[i]);
        }

        // Creación de radiobuttons de nivel

```

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

```

    rbNivel = new JRadioButtonEntero[11];
    grpNiveles = new ButtonGroup();
    x = 10;
    y = 100;
    for (int i = 0; i < rbNivel.length; i++) {
        rbNivel[i] = new JRadioButtonEntero();
        rbNivel[i].setNumero(i);
        rbNivel[i].setSize(rbNivel[i].getPreferredSize());
        rbNivel[i].setLocation(x, y);
        x += 30;
        rbNivel[i].addItemListener(new GestionaNivel());
        add(rbNivel[i]);
        grpNiveles.add(rbNivel[i]);
    }

    // Botón Aceptar para mostrar información seleccionada
    aceptar = new JButton("Aceptar");
    aceptar.setBounds(10, 200, 80, 30);
    aceptar.addActionListener(this);
    add(aceptar);

    rbNivel[0].setSelected(true);
    rbProgramadores[0].setSelected(true);
}

// Clase interna para gestión de Nivel
class GestionaNivel implements ItemListener {
    @Override
    public void itemStateChanged(ItemEvent ie) {
        RadioButtons.this.nivelSeleccionado = ((JRadioButtonEntero)
ie.getSource()).getNumero();
    }
}

@Override
public void actionPerformed(ActionEvent ae) {
    JOptionPane.showMessageDialog(null, String.format("%s %d", programadorSeleccionado,
nivelSeleccionado),
        "Programadores", JOptionPane.INFORMATION_MESSAGE);
}
}

```

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

OTRAS POSIBILIDADES

Apéndice I: Ideas para hacer un videojuego en Swing

Apendice II: Java2D

Apéndice III: JavaFX