ACTIVIDAD GUIADA 5

Eventos

Programación de eventos

La gestión de eventos es de vital importancia en los programas que contienen una interfaz gráfica de usuario, pues es quien nos va a permitir responder a las acciones del usuario. Los tres principales pilares en los que se basa la gestión de eventos son:

- **Evento** (hacer clic en un botón, pulsar una tecla, etc.)
- **Interfaz** `Listener` (`ActionListener`, `WindowListener`, etc.), "escucha" el evento
- **Objeto** (botón, marco, campo de texto, etc.) sobre el que se desencadena el evento.

Cada vez que el usuario realiza una acción como escribir un carácter o pulsar en un botón, la máquina virtual de Java (JVM) genera un **evento**.

Cada evento puede ser capturado por uno o más **oyentes** de dicho evento.

Un **oyente de eventos** es una interfaz de Java que contiene una colección de declaraciones de métodos. A priori, las clases que implementan la interfaz deben definir todos estos métodos.

Cada uno de dichos métodos representa los eventos que pueden producirse en relación con dicho oyente. *Por ejemplo, un oyente de teclado permite distinguir dos eventos (entre otros): Uno que se produce al pulsar una tecla y otro que se produce al soltarla.*

No todos los componentes soportan todos los oyentes. *Por ejemplo, no tiene sentido un evento de cierre de ventana sobre un botón.*

Por otra parte, los métodos que representan eventos reciben un **objeto como parámetro** que se encarga de representar información relacionada con el mismo. *Por ejemplo, un evento de pulsación de teclado recibirá un objeto que representa la tecla que hemos pulsado, mientras que un evento de movimiento de ratón contendrá las coordenadas de la pantalla a las cuales se ha movido el ratón.*

Por ejemplo, cuando el usuario hace clic en un botón, el **`ActionListener`** asociado al botón será notificado y, a continuación, desencadenará el método **`actionPerformed`** a ejecutar. A ficho método se le pasa un objeto de tipo **`ActionEvent.`**

Tipos de eventos y oyentes

Vamos a ver por encima las interfaces asociadas a cada oyente, así como sus métodos (los eventos que soporta), sin ánimo de ser un listado exhaustivo. Para más información puede consultarse en el tutorial oficial de Java (en inglés):

- Oyentes que todos los componentes Swing soportan.
- Otros oyentes que sólo algunos componentes Swing soportan.

ActionListener		
Son los manejadores de eventos más comunes y más fáciles de usar. Se implementan para indicar qué hay que hacer cuando un usuario realiza ciertas acciones. Ejemplos: • Cuando un usuario pulsa un botón • Cuando un usuario escoge un elemento de un menú. • Cuando un usuario pulsa *Enter* en un campo de texto. Es decir, a diferencia del resto de oyentes, que se utilizan para acciones específicas, ActionListener se utiliza como un oyente de eventos general que sirve para varios tipos de acciones.		
Métodos	`actionPerformed(ActionEvent e)`	Se llama cuando se ha realizado alguna de las acciones descritas.

KeyListener		
Escucha eventos de teclado. Para que un evento escuche un evento de teclado, dicho elemento debe tener el foco (ser el elemento activo).		
Métodos	`keyPressed(KeyEvent e)`	Se llama tras pulsar un carácter Unicode (imprimible) en el componente que se escucha
	`keyReleased(KeyEvent e)`	Se llama tras **pulsar** una tecla mientras el componente que se escucha tiene el foco
	`keyTyped(KeyEvent e)`	Se llama tras **soltar** una tecla mientras el componente que se escucha tiene el foco
KeyEvent Proporciona datos sobre la tecla pulsada. Principales métodos: • **`getKeyChar()`**: Devuelve el carácter Unicode pulsado • **`getKeyCode()`**: Obtiene el código asociado con el carácter pulsado, lo que nos permite identificar caracteres tanto imprimibles como no imprimibles.		

FocusListener		
Escucha eventos relacionados con el foco. Éstos se producen cuando un objeto gana o pierde el foco, ya sea a través del ratón o con el teclado (habitualmente con la tecla TAB).		
Métodos	`focusGained(FocusEvent e)`	Se llama cuando el componente que se escucha gana el foco
	`focusLost(KeyEvent e)`	Se llama cuando el componente que se escucha pierde el foco

MouseListener		
Escucha eventos producidos cuando se utiliza el ratón (o un dispositivo similar) para interactuar con un componente, a excepción de los relacionados con la rueda central que se tratan en un oyente distinto.		
Métodos	`mouseClicked(MouseEvent e)`	El usuario pulsa en el componente que escucho.
	`mouseEntered(MouseEvent e)`	El cursor entra los límites del componente que escucha.
	`mouseExited(MouseEvent e)`	El cursor sale de los límites del componente que escucha.
	`mousePressed(MouseEvent e)`	El usuario pulsa un botón mientras el cursor está sobre componente que escucha.
	`mouseReleased(MouseEvent e)`	El usuario suelta un botón mientras el cursor está sobre componente que escucha.

`MouseEvent`

Proporciona datos sobre el evento de ratón. Principales métodos:

- **`int getX(), int getY(), Point getPoint()`**
  - Devuelve la posición (x,y) en la cual el evento ocurre. Las coordenadas son relativas al componente que desemboca el evento
- **`int getXOnScreen(), int getYOnScreen(), int getLocationOnScreen()`**:
  - Devuelve la posición absoluta (x,y) del evento.
- **`int getClickCount().`** Devuelve el número de pulsaciones consecutivas que se han realizado (por ejemplo, 2 en un doble click).

WindowListener		
Permiten detectar cambios de estado en las ventanas.		
Métodos	`windowOpened(WindowEvent e)`	Cuando la ventana que se escucha se ha abierto por primera vez.
	`windowClosing(WindowEvent e)`	Se llama justo antes de que la ventana comience a cerrarse.
	`windowClosed(WindowEvent e)`	Se llama una vez que la ventana se ha cerrado.
	`windowIconified(WindowEvent e)` `WindowDeiconified(WindowEvent e)`	Se llama una vez que la ventana que se escucha se ha minimizado o maximizado respectivamente.
	`windowActivated(WindowEvent e)` `windowDeactivated(WindowEvent e)`	Se llama una vez que la ventana que se escucha se ha activado (ganado el foco) o desactivado respectivamente.

A pesar de lo complicado que puede parecer, hay que tener en cuenta que siempre se programa de la misma forma. Solo hay que cambiar el componente que se quiere programar y asignar un nombre a la función donde se programará el evento.

Ejemplo de programación de eventos de acción

Un **evento de acción** hace referencia a la activación de un objeto (un botón, un cuadro de texto, un combo, etc...). Solo existe un tipo de evento de acción, llamado *actionPerformed*.

El evento *actionPerformed* pertenece a un objeto oyente llamado ***ActionListener***.

La forma de programar el evento *actionPerformed* de un componente xxx es la siguiente:

```
xxx.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
            xxxActionPerformed(evt);
      }
});
```

Para entender la sintaxis de la programación de un evento de acción, supongamos el siguiente ejemplo:

Se quiere programar el evento de un botón llamado btnSumar desde código. He aquí el código para crear el evento. Una explicación del código:

El componente cuyo evento se quiere programar. En este ejemplo, el botón btnSumar

El método *addActionListener* permite asignar el oyente al botón.

El evento *actionPerformed* pertenece a un objeto oyente llamado *ActionListener*.

```
btnSumar.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
            btnSumarActionPerformed(evt);
      }
});
```

Dentro del procedimiento *actionPerformed* se hace una llamada a otro procedimiento. El nombre de este otro procedimiento puede ser cualquiera, pero usaremos el nombre del componente seguido de "ActionPerformed"

El evento *actionPerformed*, es en realidad un procedimiento, que tiene como parámetro un objeto del tipo *ActionEvent* al que llamaremos *evt* por convención.

El código anterior permite crear y asignar el evento *actionPerformed* al botón *btnSumar*, pero no programa el evento. Para programar el evento es necesario crear el procedimiento cuya llamada se incluye dentro del evento *actionPerformed*:

```
public void btnSumarActionPerformed(ActionEvent evt) {
      … aquí se programa el evento actionPerformed del botón btnSumar …
}
```

Hay que tener en cuenta que en el código expuesto antes participan nuevas clases como son ActionEvent y ActionListener, y se tendrán que agregar los import correspondientes.

Registrar un gestor de eventos

Para escuchar un evento, un objeto debe seguir estos tres pasos:

1. Registrarse para ser el oyente de dicho evento. Esto se hace implementando la interfaz oyente correspondiente.

```
public class MiClase implements ActionListener
```

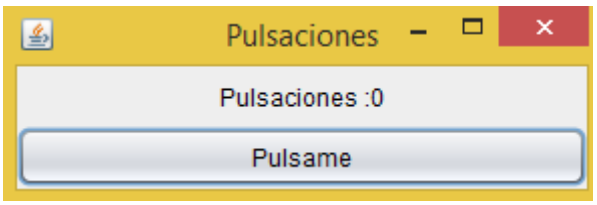
2. Dentro de la clase anterior, definir los métodos de la interfaz de escucha:

```
public void actionPerformed(ActionEvent e){
...//código que reacciona ante la acción…
}
```

3. Agregar el oyente al objeto cuyos eventos queremos escuchar

```
JButton btnEjemplo=new JButton("Ejemplo");
btnEjemplo.addActionListener(new MiClase());
```

Veamos un ejemplo donde aplicar lo visto. Vamos a implementar una interfaz simple compuesta por botón y una etiqueta que lleva cuenta de las veces que hemos pulsado el botón:



Como propiedades de la clase, además del botón y la etiqueta, tendremos un entero que representa el número de pulsaciones y que usaremos para ir actualizando la etiqueta.

Siguiendo los pasos que acabamos de describir:

1. Registramos la clase para ser el oyente del evento. Para registrar eventos de pulsaciones debemos implementar la interfaz **`ActionListener`**. El encargado de implementar la interfaz será el propio frame:

```
public class CuentaPulsaciones extends JFrame implements ActionListener
```

2. Definimos los métodos de interfaz de escucha. El interfaz **`ActionListener`** tiene un único método de escucha llamado **`actionPerformed`**.

```
@Override
public void actionPerformed(ActionEvent e) {
  //Aquí iría el código asociado a la acción
}
```

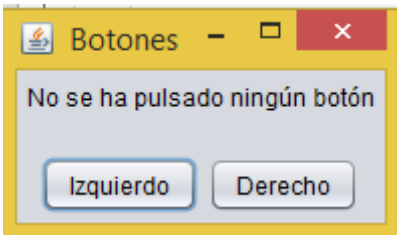
3. Agregamos el oyente al botón. Dado que el oyente es el propio frame, al añadir `this` estamos agregando el objeto donde nos encontramos.

```
btnPulsame.addActionListener(this);
```

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

public class CuentaPulsaciones extends JFrame implements ActionListener{
    JLabel lblPulsaciones;
    JButton btnPulsame;
    int numeroPulsaciones;

    public CuentaPulsaciones(){
        super("Pulsaciones");
        inicializaPulsaciones();

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        lblPulsaciones=new JLabel("Pulsaciones :"+getPulsaciones(),JLabel.CENTER);
        btnPulsame=new JButton("Pulsame");
        btnPulsame.addActionListener(this);

        GridLayout admin=new GridLayout(2,1);
        setLayout(admin);
        add(lblPulsaciones);
        add(btnPulsame);

        setSize(300,100);
        setVisible(true);
    }

private void inicializaPulsaciones(){
        numeroPulsaciones=0;
    }

    private int getPulsaciones(){
        return numeroPulsaciones;
    }

    private void muestraPulsaciones(){
        lblPulsaciones.setText("Pulsaciones: "+getPulsaciones());
    }

     private void actualizaPulsaciones(){
        numeroPulsaciones++;
    }
}
      @Override
    public void actionPerformed(ActionEvent e) {
       actualizaPulsaciones();
       muestraPulsaciones();
    }

    public static void main(String[] args) {
       CuentaPulsaciones frPulsaciones=new CuentaPulsaciones();
    }
}
```

Escuchando objetos múltiples

Podemos tener varios objetos escuchando el mismo evento de varios objetos mediante un mismo oyente. El problema puede plantearse para distinguir, dentro del oyente, cuál de los objetos ha desencadenado el evento. Supongamos que tenemos una aplicación como la siguiente:



En función del botón que pulsemos queremos que se actualice la etiqueta superior indicando si se ha pulsado el botón izquierdo (**btnIzquierdo**) o el derecho (**btnDerecho**). Además, queremos utilizar el mismo oyente para escuchar el evento de pulsar ambos botones.

El método **getSource()** de **ActionEvent** devuelve el objeto que ha desencadenado el evento, siendo fácil distinguir qué botón se ha pulsado. El método de escucha podría ser el siguiente:

```
@Override
public void actionPerformed(ActionEvent e) {
  if(e.getSource()==btnDerecho)
      actualizaEtiqueta("derecho");
  else
      actualizaEtiqueta("izquierdo");
}
```

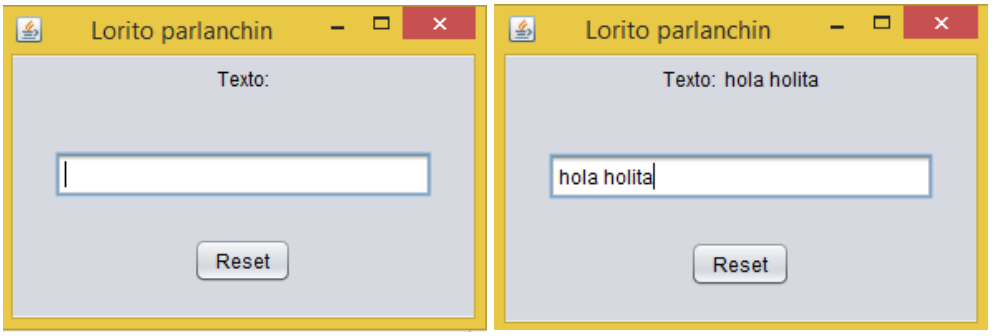
El método **actualizaEtiqueta()** cambia el texto de la etiqueta superior en función de la cadena que le pasemos.

A continuación se muestra el código completo del programa:

```
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class BotonAlternativo extends JFrame implements ActionListener{

    JLabel lblBotonPulsado;
    JButton btnIzquierdo;
    JButton btnDerecho;

    public BotonAlternativo(){
        super("Botones");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        lblBotonPulsado=new JLabel("No se ha pulsado ningún botón",JLabel.CENTER);

        btnIzquierdo=new JButton("Izquierdo");
        btnDerecho=new JButton("Derecho");

        btnIzquierdo.addActionListener(this);
        btnDerecho.addActionListener(this);

        GridLayout admin=new GridLayout(2,1);
        setLayout(admin);

        JPanel pEtiqueta=new JPanel();
        pEtiqueta.add(lblBotonPulsado);
        add(pEtiqueta);

        JPanel pBotones=new JPanel();
        FlowLayout adminPanel=new FlowLayout();
        pBotones.setLayout(adminPanel);
        pBotones.add(btnIzquierdo);
        pBotones.add(btnDerecho);
        add(pBotones);

        pack();
        setVisible(true);
    }

   private void actualizaEtiqueta(String boton){
       lblBotonPulsado.setText("Has pulsado el botón "+boton);
   }

 public static void main(String[] args) {
       BotonAlternativo frEventos=new BotonAlternativo();
    }

    @Override
    public void actionPerformed(ActionEvent e) {
      if(e.getSource()==btnDerecho)
          actualizaEtiqueta("derecho");
      else
          actualizaEtiqueta("izquierdo");

    }
```

Escuchando eventos múltiples

Vamos a suponer que queremos escuchar eventos que provienen de varias fuentes. Por ejemplo, queremos implementar la siguiente ventana, compuesta por una etiqueta, caja de texto y un botón. A medida que vayamos escribiendo texto en la caja de texto, éste se irá incorporando a la etiqueta superior. Por el contrario, si pulsamos el botón *Reset*, tanto la caja de texto como la etiqueta restablecerán su valor inicial.



Necesitamos tener un oyente para cada uno de los eventos que queremos escuchar. El botón tendrá que escuchar mediante un **`ActionListener`** como en el caso anterior, y la caja de texto tendrá que escuchar mediante un **`TextListener`** para estar atenta a las pulsaciones de teclado.

Primero, hacemos que el frame implemente las interfaces correspondientes:

```
public class Loro extends JFrame implements ActionListener,TextListener{
```

La interfaz **`TextListener`** tiene los métodos **`keyPressed()`**, **`keyTiped()`** y **`keyRelased()`** que debemos implementar. A los anteriores se suma el **`actionPerformed()`** de **`ActionListener`**:

```
    @Override
    public void actionPerformed(ActionEvent ae) {}
    @Override
    public void keyPressed(KeyEvent ke) { }
    @Override
    public void keyTyped(KeyEvent ke) {}
    @Override
    public void keyReleased(KeyEvent ke) {   }
```

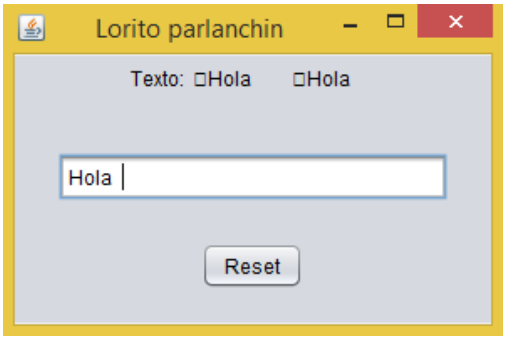
Únicamente nos interesa implementar los dos primeros métodos. El **`actionPerformed()`** borraría el resto de componentes, mientras que el **`keyTyped()`** reproduciría la tecla que he pulsado en la etiqueta. A la tecla podemos acceder mediante el método **`getKeyChar()`** de **`KeyEvent`**, que nos devuelve el carácter pulsado (usando codificación UTF-8).

```
@Override
 public void actionPerformed(ActionEvent ae) {
    lblLoro.setText("");
    txtLoro.setText("");
 }
 @Override
 public void keyPressed(KeyEvent ke) {
       lblLoro.setText(lblLoro.getText()+ke.getKeyChar());
 }
```

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

public class Loro extends JFrame implements ActionListener, KeyListener{
    JLabel lblLoro;
    JTextField txtLoro;
    JButton btnReset;

    public Loro(){
        super("Lorito parlanchin");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel lblTexto=new JLabel("Texto:");
        lblLoro=new JLabel("",JLabel.CENTER);
        txtLoro=new JTextField(20);
        btnReset=new JButton("Reset");

        txtLoro.addKeyListener(this);
        btnReset.addActionListener(this);

        GridLayout admin=new GridLayout(3,1);
        setLayout(admin);

        JPanel panel1=new JPanel();
        panel1.add(lblTexto);
        panel1.add(lblLoro);

        JPanel panel2=new JPanel();
        panel2.add(txtLoro);

        JPanel panel3=new JPanel();
        panel3.add(btnReset);

        add(panel1);
        add(panel2);
        add(panel3);

        setSize(300,200);
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent ae) {
       lblLoro.setText("");
       txtLoro.setText("");
    }
    @Override
    public void keyPressed(KeyEvent ke) {
                lblLoro.setText(lblLoro.getText()+ke.getKeyChar());
    }
    @Override
    public void keyTyped(KeyEvent ke) { }

    @Override
    public void keyReleased(KeyEvent ke) { }

    public static void main(String[] args) { Loro frLoro=new Loro();    }
}
```

Si probamos el programa con el código anterior veremos que hay un par de problemas asociados al mismo:

- Por una parte, si intentamos escribir en mayúsculas veremos que no funciona, pues se imprimen caracteres "raros" (dado que SHIFT o BLOQ MAYUS son teclas no imprimibles).



- Por otra parte, si pulsamos la tecla de borrado veremos que se borra el carácter correspondiente en la caja de texto pero no en la etiqueta. Esto es porque no estamos procesando el carácter de borrado correctamente.

El siguiente fragmento de código modifica el código de `keyPressed()` para tener en cuenta los problemas anteriores. Haremos varias comprobaciones:

- Por un lado comprobamos si hemos pulsado la tecla de borrar. Esto lo hacemos usando el método **`getKeyCode()`** de **`KeyEvent`**, que nos devuelve un código que representa el carácter borrado (y que comparamos con la variable de clase **`VK_BACK_SPACE`** definida en la clase **`KeyEvent`** que representa la tecla de borrado). El uso de este tipo de constantes permite la comprobación de teclas que no tienen un equivalente imprimible como la propia tecla de borrado.
    - En caso de haber pulsado la tecla, y si tenemos algo de texto en la caja de texto, reescribimos el texto de la misma omitiendo el último carácter (utilizamos los métodos **`substring()`** y **`length()`** de **`String`**).

- En segundo lugar, antes de imprimir el carácter en la etiqueta, comprobamos si el código del mismo no se corresponde con la tecla SHIFT ni el Bloq Mayus usando el mismo método que se acaba de describir. De ser así, añadimos el carácter pulsado a la etiqueta, en este caso usando el método **`getKeyChar()`** de **`KeyEvent`**.

```
@Override
public void keyPressed(KeyEvent ke) {
    if(ke.getKeyCode()==KeyEvent.VK_BACK_SPACE){
      if(lblLoro.getText().length()>0)
         lblLoro.setText(lblLoro.getText().substring(0, lblLoro.getText().length()-1));
    }
    else
        if((ke.getKeyCode()!=KeyEvent.VK_SHIFT)&&(ke.getKeyCode()!=KeyEvent.VK_CAPS_LOCK))
            lblLoro.setText(lblLoro.getText()+ke.getKeyChar());
}
```

Clases adaptadoras

En el ejemplo anterior, hemos implementado el interfaz **`KeyListener`**. Esto implica que hemos tenido que implementar todos los métodos de dicho interfaz, aunque sólo nos interesaba dar comportamiento a uno de ellos [**`keyPressed()`**], mientras que los otros [**`keyTyped()`** y **`keyReleased()`**] están vacíos.

En el caso de algunas interfaces de escucha, como es el caso de **`WindowListener`** (que tiene 7 métodos) este proceso puede ser algo tedioso. Para simplificar esta tarea, Java ha desarrollado una clase adaptadora para cada interfaz de escucha que tenga más de un método. Una clase de este tipo implementa todos los métodos en la interfaz pero no hace nada con ellos. A continuación se indica cómo utilizar un adaptador. En el ejemplo anterior, se reemplaza la llamada a **`txtLoro.addKeyListener(this)`** por la siguiente sentencia:

```
txtLoro.addKeyListener(new KeyAdapter() {
     public void keyPressed(KeyEvent ke){
         lblLoro.setText(lblLoro.getText()+ke.getKeyChar());
     }
 });
```

NOTA: Por motivos de simplicidad, se está usando la primera versión del método que no tiene en cuenta pulsación de teclas no imprimibles.

En este caso la clase adaptadora es una clase anónima, que se define en el momento para ser usada como escuchador. Además, al ser una clase interna (definida dentro de otra clase), tiene acceso automático a los miembros de la clase contenedora (en nuestro caso el frame), luego se puede acceder a los campos y métodos de la clase contenedora como si fueran de su propiedad.