11.B. Eventos.

Sitio: Aula Virtual CIERD (CIDEAD)
Curso: Programación_DAM
Libro: 11.B. Eventos.
Imprimido por: LUIS PUJOL
Día: miércoles, 20 de mayo de 2020, 09:20

Tabla de contenidos

1 Eventos.

Llegamos ahora a uno de los temas más importantes en toda clase de lenguajes de programación modernos: los eventos. Estos, además son de una especial importancia cuando estamos haciendo programación gráfica. La gran mayoría de los sistemas gráficos disponen de algún modelo de eventos que permite notificar al sistema la interacción del usuario. Existen eventos de naturaleza muy variada, como son la pulsación de teclado, el movimiento del ratón y clicado, la pulsación de un botón representado en pantalla o la elección de un item a través de un selector gráfico, etc.

Debemos implementar en las clases mecanismos que permitan registrar eventos.

1.1 Introducción.

¿Qué es un **evento**?

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

- pulsar un botón con el ratón;
- hacer doble clic;
- pulsar y arrastrar;
- pulsar una combinación de teclas en el teclado;
- pasar el ratón por encima de un componente;
- salir el puntero de ratón de un componente;
- abrir una ventana;
- etc.

¿Qué es la **programación guiada por eventos**?
Imagina la ventana de cualquier aplicación, por ejemplo la de un procesador de textos. En esa ventana aparecen multitud de elementos gráficos interactivos, de forma que no es posible que el programador haya previsto todas las posibles entradas que se pueden producir por parte del usuario en cada momento.

Con el control de flujo de programa de la **programación imperativa**, el programador tendría que estar continuamente leyendo las entradas (de teclado, o ratón, etc) y comprobar para cada entrada o interacción producida por el usuario, de cual se trata de entre todas las posibles, usando estructuras de flujo condicional (`if-then-else`, `switch`) para ejecutar el código conveniente en cada caso. Si piensas que para cada opción del menú, para cada botón o etiqueta, para cada lista desplegable, y por tanto para cada componente de la ventana, incluyendo la propia ventana, habría que comprobar todos y cada uno de los eventos posibles, nos damos cuenta de que las posibilidades son casi infinitas, y desde luego impredecibles. Por tanto, de ese modo es imposible solucionar el problema.

Para abordar el problema de tratar correctamente las interacciones del usuario con la interfaz gráfica de la aplicación hay que cambiar de estrategia, y la **programación guiada por eventos es una buena solución**, veamos cómo funciona el modelo de gestión de eventos.

1.2 Modelo de gestión de eventos.

¿Qué sistema operativo utilizas? ¿Posee un entorno gráfico? Hoy en día, la mayoría de sistemas operativos utilizan interfaces gráficas de usuario. Este tipo de sistemas operativos están **continuamente monitorizando el entorno para capturar y tratar los eventos** que se producen.

El sistema operativo informa de estos eventos a los programas que se están ejecutando y entonces cada programa decide, según lo que se haya programado, qué hace para dar respuesta a esos eventos.

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

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

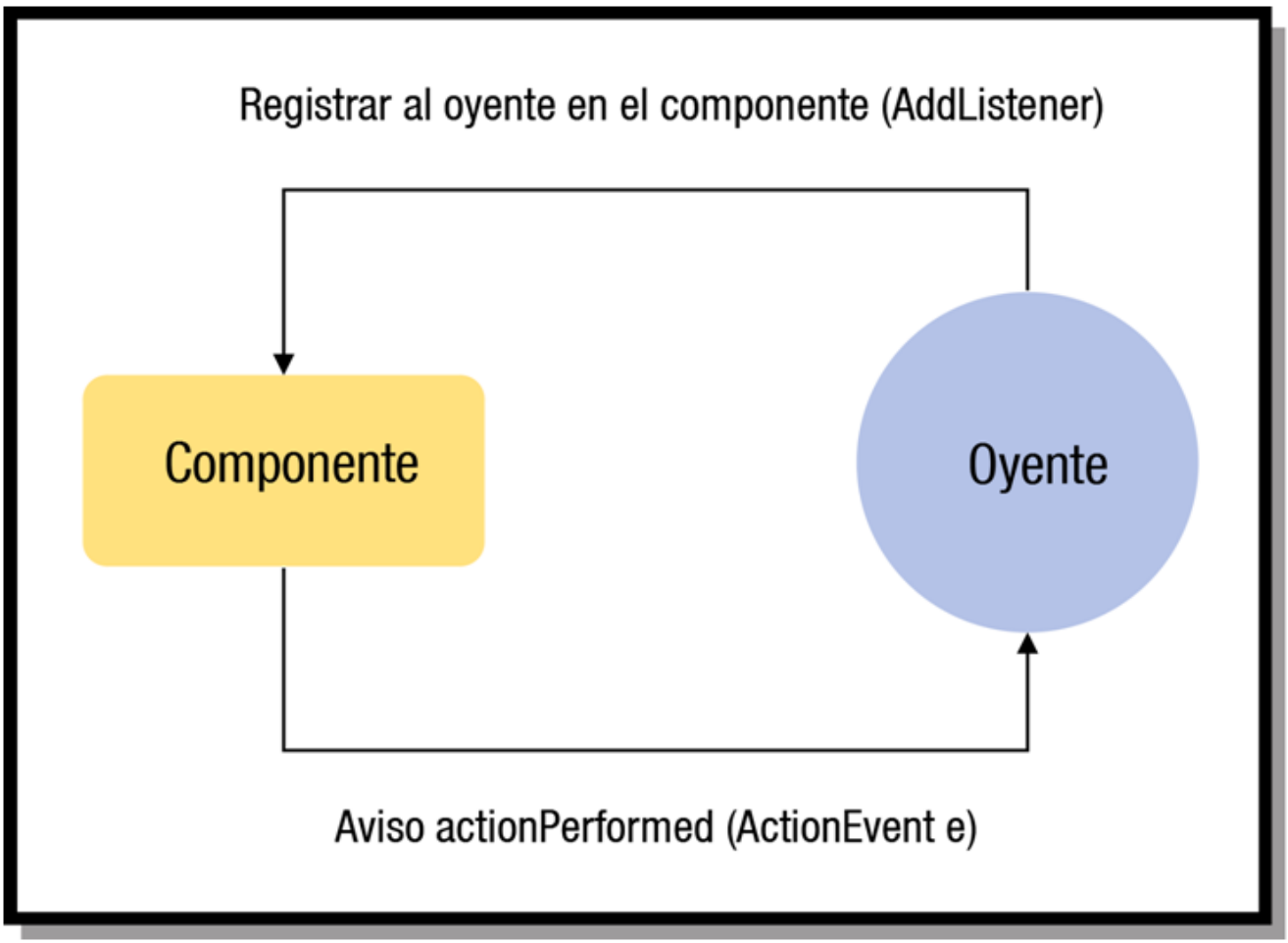


Imagen extraída de curso Programación del MECD.

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

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

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

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

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

1. Crear la clase oyente que implemente la interfaz.
   - Ej. `ActionListener`: pulsar un botón.

2. Implementar en la clase oyente los métodos de la interfaz.
   - Ej. `void actionPerformed(ActionEvent)`.

3. Crear un objeto de la clase oyente y registrarlo como oyente en uno o más componentes gráficos que proporcionen interacción con el usuario.

Autoevaluación

Con la programación guiada por eventos, el programador se concentra en estar continuamente leyendo las entradas de teclado, de ratón, etc., para comprobar cada entrada o interacción producida por el usuario.

- ○ Verdadero.
- ○ Falso.

1.3 Tipos de eventos.

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

- **Físicos** o de **bajo nivel**: que corresponden a un evento hardware claramente identificable. Por ejemplo, se pulsó una tecla (`KeyStrokeEvent`). Destacar los siguientes:
  - En componentes: `ComponentEvent` . Indica que un componente se ha movido, cambiado de tamaño o de visibilidad
  - En contenedores: `ContainerEvent` . Indica que el contenido de un contenedor ha cambiado porque se añadió o eliminó un componente.
  - En ventanas: `WindowEvent` . Indica que una ventana ha cambiado su estado.
  - `FocusEvent` , indica que un componente ha obtenido o perdido la entrada del foco.

- **Semánticos** o de mayor nivel de abstracción: se componen de un conjunto de eventos físicos, que se suceden en un determinado orden y tienen un significado más abstracto. Por ejemplo: el usuario elige un elemento de una lista desplegable (`ItemEvent`).
  - `ActionEvent` , `ItemEvent` , `TextEvent` , `AdjustmentEvent` .

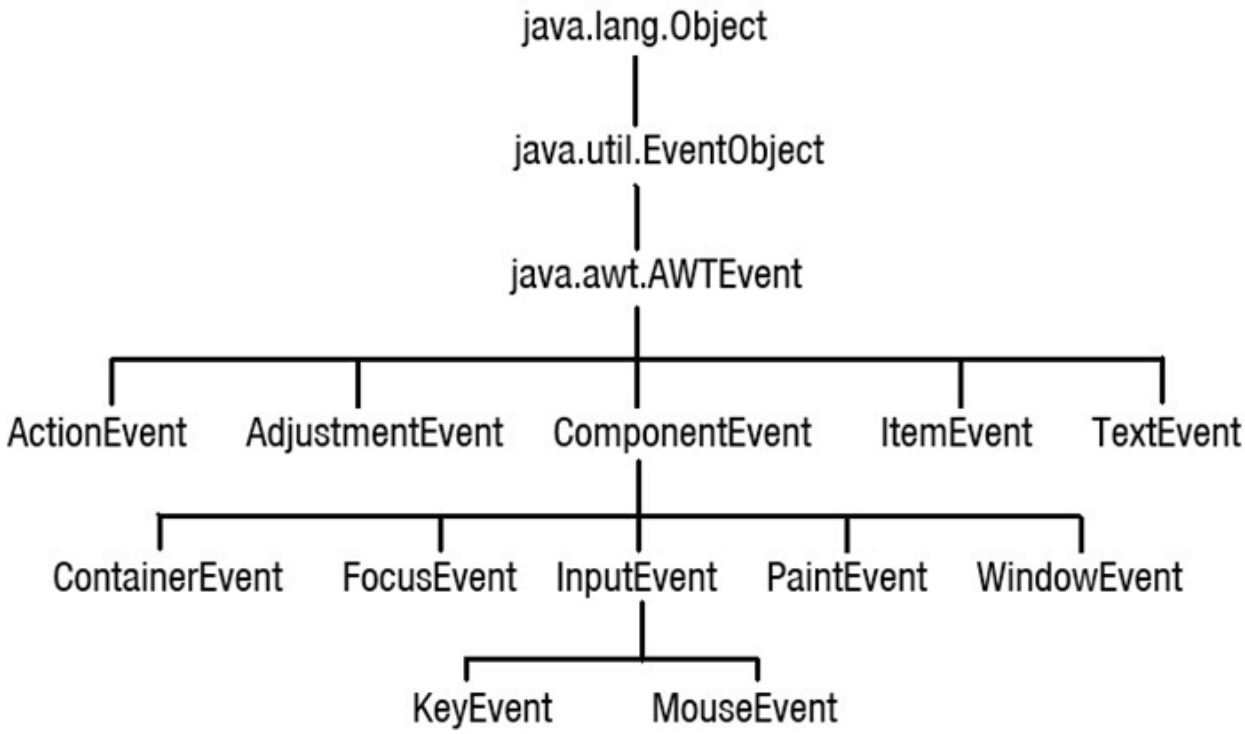


Imagen extraída de curso Programación del MECD.

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

- La clase `java.util.EventObject` es la clase base de todos los eventos en Java.

- La clase `java.awt.AWTEvent` es la clase base de todos los eventos que se utilizan en la construcción de GUI.

- Cada tipo de evento `LoqueseaEvent` tiene asociada una interfaz `LoqueseaListener` que nos permite definir manejadores de eventos.

- Con la idea de simplificar la implementación de algunos manejadores de eventos, el paquete `java.awt.event` incluye clases `loqueseaAdapter` que implementan las interfaces `loqueseaListener`.

Autoevaluación

El evento que se dispara cuando le llega el foco a un botón es un evento de tipo físico.

- ○ Verdadero.
- ○ Falso.

1.4 Eventos de teclado.

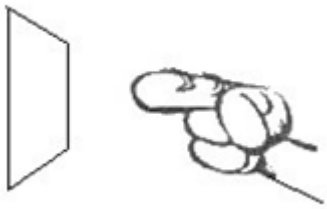
Los eventos de teclado se generan como respuesta a que el usuario pulsa o libera una tecla mientras un componente tiene el foco de entrada.

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

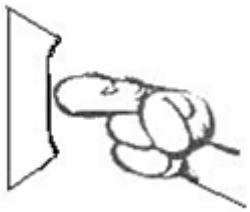
`KeyEvent` (evento de teclas)	
Métodos más usuales	**Explicación**
`char getKeyChar()`	Devuelve el carácter asociado con la tecla pulsada.
`int getKeyCode()`	Devuelve el valor entero que representa la tecla pulsada.
`String getKeyText()`	Devuelve un texto que representa el código de la tecla.
`Object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

La clase `KeyEvent`, define muchas constantes así:

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



Botón en estado normal.



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

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

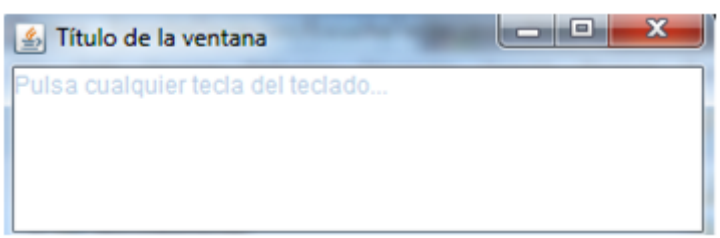
En la siguiente código se puede ver un ejemplo del uso eventos. En concreto vemos cómo se están capturando los eventos que se producen al pulsar una tecla y liberarla. El programa escribe en un área de texto las teclas que se oprimen.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package Escuchando;  //Quitar esta línea si utilizasel paquete por defecto

import javax.swing.* ;
import java.awt.event.* ;
/**
 *
 * @author JJBH
 */
// Definimos la clase que hereda de JFrame
public class EscuchaTeclas extends JFrame {
    // Variables para escribir
    private String linea1 = "", linea2 = "", linea3 = "";
    private JTextArea areaTexto;

    // Constructor de la clase
    public EscuchaTeclas () {
        // Crear objeto JTextArea
        areaTexto = new JTextArea( 10, 15 );
        areaTexto.setText( "Pulsa cualquier tecla del teclado..." );
```

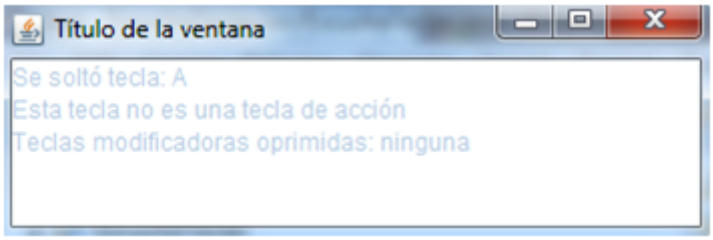
```
    areaTexto.setEnabled( false );

    // Añadir al JFrame el objeto areaTexto
    this.getContentPane().add( areaTexto );

    // Crear el objeto oyente de teclas
    OyenteTeclas oyenteTec = new OyenteTeclas() ;

    // Registrar el oyente en el JFrame
    this.addKeyListener(oyenteTec);

}

// Implementar la clase oyente que implemente el interface KeyListener
class OyenteTeclas implements KeyListener{
    // Gestionar evento de pulsación de cualquier tecla
    public void keyPressed( KeyEvent evento )
    {
       linea1 = "Se oprimió tecla: " + evento.getKeyText( evento.getKeyCode() );
       establecerTexto( evento );
    }

    // Gestionar evento de liberación de cualquier tecla
    public void keyReleased( KeyEvent evento )
    {
      linea1 = "Se soltó tecla: " + evento.getKeyText( evento.getKeyCode() );
       establecerTexto( evento );
    }

    // manejar evento de pulsación de una tecla de acción
    public void keyTyped( KeyEvent evento )
     {
      linea1 = "Se escribió tecla: " + evento.getKeyChar();
      establecerTexto( evento );
     }
}
```

```
    // Establecer texto en el componente areaTexto
    private void establecerTexto( KeyEvent evento )
    {
       // getKeyModifiersText devuelve una cadena que indica
       // el modificador de la tecla, por ejemplo Shift
       String temp = evento.getKeyModifiersText( evento.getModifiers() );

       linea2 = "Esta tecla " + ( evento.isActionKey() ? "" : "no " ) +
        "es una tecla de acción";
       linea3 = "Teclas modificadoras oprimidas: " + ( temp.equals( "" ) ? "ninguna" : temp );

       // Establecer texto en el componente areaTexto
       areaTexto.setText( linea1 + "\n" + linea2 + "\n" + linea3 + "\n" );
    }

    public static void main( String args[] )
    {
      // Crear objeto y establecer propidades
      EscuchaTeclas ventana = new EscuchaTeclas();
      ventana.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      ventana.setTitle("Título de la ventana");
      ventana.setSize( 360, 120 );
      ventana.setVisible(true);

    }
}
```

Inicialmente la salida será:



Si pulsamos la tecla "A", aparecerá el texto "se pulsó la tecla A" y al soltar la tecla ya quedará finalmente "se soltó la tecla A":


Título de la ventana
Se soltó tecla: A
Esta tecla no es una tecla de acción
Teclas modificadoras oprimidas: ninguna

1.5 Eventos de ratón.

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

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

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

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

En el siguiente códigoo podemos ver una demostración de un formulario con dos botones. Implementamos un oyente `MouseListener` y registramos los dos botones para detectar tres de los cinco eventos del interface.

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

public class MiMarcoRaton extends javax.swing.JFrame {

  /** Constructor: crea nuevo marco miMarcoRaton */
  public MiMarcoRaton() {
    initComponents();
```

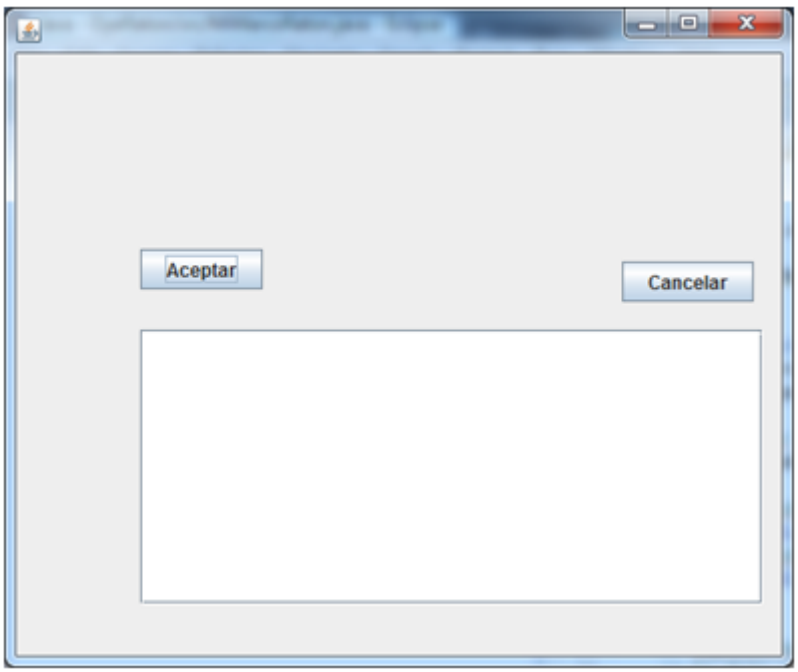
```
    // Crear el objeto oyente de ratón
    OyenteRaton oyenteRat = new OyenteRaton() ;

    // Registrar el oyente en el botón de Aceptar
    jButton1.addMouseListener(oyenteRat);
    // Registrar el oyente en el botón de Cancelar
    jButton2.addMouseListener(oyenteRat);
}

// Implementar la clase oyente que implemente el interface MouseListener
// Se deja en blanco el cuerpo de mouseEntered y de mouseExited, ya que
// no nos interesan en este ejemplo. Cuando se desea escuchar algún
// tipo de evento, de deben implementar todos los métodos del interface
// para que la clase no tenga que ser definida como abstracta
class OyenteRaton implements MouseListener{
    // Gestionar evento de pulsación de cualquier tecla
    public void mousePressed(MouseEvent e) {
        escribir("Botón de ratón pulsado", e) ;
    }
    public void mouseReleased(MouseEvent e) {
        escribir("Botón de ratón liberado", e);
    }
    public void mouseEntered(MouseEvent e) {
    }
    public void mouseExited(MouseEvent e) {
    }
    public void mouseClicked(MouseEvent e) {
        escribir("Click en el botón del ratón", e);
    }

    void escribir(String eventDescription, MouseEvent e) {
        // Escribir en el área de texto la descripción  que
        // se recibe como parámetro
        jTextArea1.append(eventDescription + ".\n");
```

```
        // Comprobamos cuál de los dos botones es y lo escribimos
        if (e.getComponent().getName().equals(jButton1.getName()) )
          jTextArea1.append("Es el botón Aceptar.\n");
        else
          jTextArea1.append("Es el botón Cancelar.\n");
    }

}

/** This method is called from within the constructor to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
@SuppressWarnings("unchecked")
// <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-BEGIN:initComponents
private void initComponents() {

    jButton1 = new javax.swing.JButton();
    jButton2 = new javax.swing.JButton();
    jScrollPane1 = new javax.swing.JScrollPane();
    jTextArea1 = new javax.swing.JTextArea();

    setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

    jButton1.setText("Aceptar");
    jButton1.setName("Aceptar"); // NOI18N

    jButton2.setText("Cancelar");
    jButton2.setName("Cancelar"); // NOI18N

    jTextArea1.setColumns(20);
    jTextArea1.setRows(5);
    jScrollPane1.setViewportView(jTextArea1);

    javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
```

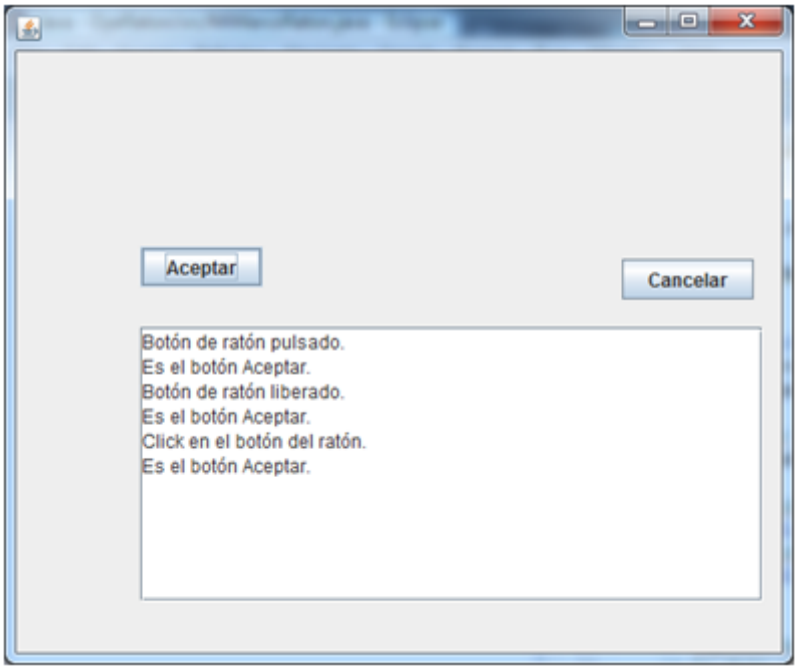
```
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addGap(80, 80, 80)
                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING, false)
                    .addGroup(layout.createSequentialGroup()
                        .addComponent(jScrollPane1, javax.swing.GroupLayout.PREFERRED_SIZE, 402, javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addContainerGap())
                    .addGroup(layout.createSequentialGroup()
                        .addComponent(jButton1)
                        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                        .addComponent(jButton2)
                        .addGap(18, 18, 18))))
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addGap(126, 126, 126)
                .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                    .addGroup(layout.createSequentialGroup()
                        .addComponent(jButton1)
                        .addGap(26, 26, 26))
                    .addGroup(layout.createSequentialGroup()
                        .addComponent(jButton2)
                        .addGap(18, 18, 18)))
                .addComponent(jScrollPane1, javax.swing.GroupLayout.PREFERRED_SIZE, 177, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addContainerGap(33, Short.MAX_VALUE))
        );

        pack();
    }// </editor-fold>//GEN-END:initComponents
```

```
    /**
    * @param args the command line arguments
    */
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new MiMarcoRaton().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JButton jButton1;
    private javax.swing.JButton jButton2;
    private javax.swing.JScrollPane jScrollPane1;
    private javax.swing.JTextArea jTextArea1;
    // End of variables declaration//GEN-END:variables

}
```

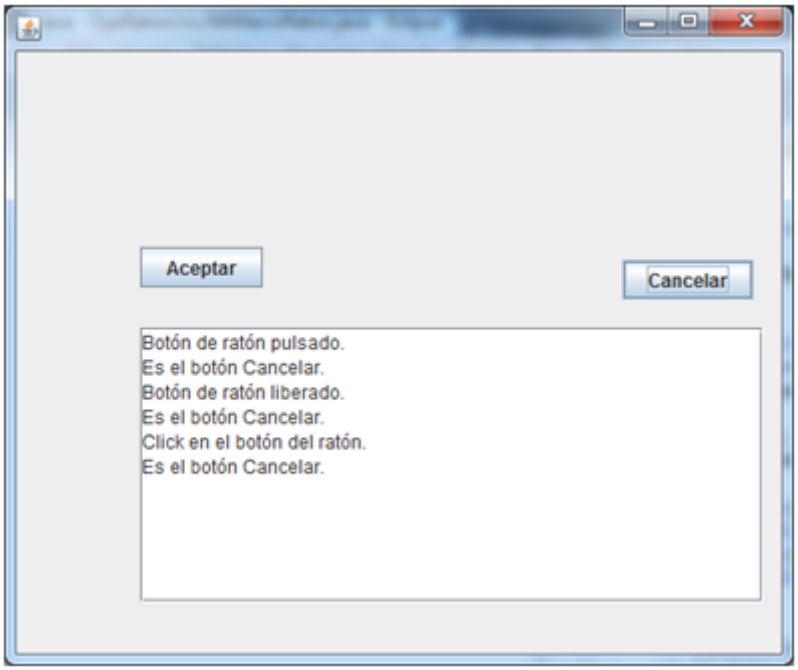
La salida es:



Al clicar sobre el botón Aceptar:



Al clicar sobre el botón Cancelar:



Como se ve en el código, se deja en blanco el cuerpo de `mouseEntered` y de `mouseExited`, ya que no nos interesan en este ejemplo. Cuando se desea escuchar algún tipo de evento, de deben implementar todos los métodos del interface para que la clase no tenga que ser definida como abstracta. Para evitar tener que hacer esto, podemos utilizar adaptadores.

Para saber más

En el enlace que ves a continuación, hay también un ejemplo interesante de la programación de eventos del ratón.

La programación del ratón

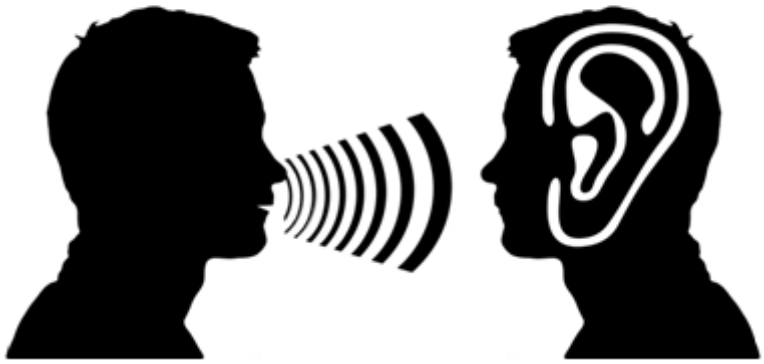
Autoevaluación

Cuando el usuario deja de pulsar una tecla se invoca a keyReleased(KeyEvent e)**.**

- Verdadero.
- Falso.

1.6 Creación de controladores de eventos.

A partir del JDK1.4 se introdujo en Java la clase `EventHandler` para soportar oyentes de evento muy sencillos.



La utilidad de estos controladores o manejadores de evento es:

- Crear oyentes de evento sin tener que incluirlos en una clase propia.

- Esto aumente el rendimiento, ya que no "añade" otra clase.

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

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

```
miBoton.addActionListener(
```

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

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

```
// Código equivalente empleando una clase interna en lugar de EventHandler.
```

```
miBoton.addActionListener(new ActionListener() {
```

```
        public void actionPerformed(ActionEvent e) {
```

```
                frame.dibujar();
```

```
        }
```

```
});
```

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

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

Esto correspondería a la implementación de la siguiente clase interna:

```
// Código equivalente utilizando una clase interna en vez de EventHandler.
```

```
new ActionListener {
```

```
        public void actionPerformed(ActionEvent e) {
```

```
                miBoton.setLabel(((JTextField)e.getSource()).getText());
```

```
        }
```

```
}
```

Autoevaluación

El uso de EventHandler **tiene como inconveniente, que los errores no se detectan en tiempo de ejecución.**

- Verdadero.
- Falso.