DESARROLLO DE INTERFACES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Escuchadores de eventos en componentes visuales

Los eventos permiten que el usuario pueda establecer una «**comunicación**» con cualquier aplicación para que esta se ajuste a las decisiones del usuario en lo que respecta a su recorrido por un sitio web, aplicación o herramienta.

Funcionalidades de lenguajes dinámicos

Reflexión: **recuperar y modificar** de forma dinámica diferentes **datos** relativos a la **estructura de un objeto**: los **métodos** y **propiedades** de la clase, los **constructores**, las **interfaces** o el **nombre** original **de la clase**, entre otros.

Introspección: permiten a entornos visuales de diseño **tomar de forma dinámica** todos los **métodos**, **propiedades** o **eventos** asociados a un **componente**, que se colocan sobre el lienzo de diseño simplemente arrastrando y soltando.

La **introspección** de los componentes visuales **requiere de la reflexión**.

Ambas propiedades son dos **características clave** en el diseño de **JavaBean**.

Persistencia del componente

Permite que el **estado de una determinada clase no varíe** y es posible a través de la **serialización**. Cuando se crea un **nuevo componente**, será necesario **implementar** alguna de las **interfaces** siguientes en función del **tipo de serialización** escogida:

- Serialización **automática**. Utiliza la interfaz: **java.io.Serializable.**
- Serialización **programada**. Utiliza la interfaz: **java.io.Externalizable**.

Clases de eventos

La programación basada en eventos es la clave de la **iteración** entre el usuario y una interfaz. Este tipo de programación podría **dividirse** en **dos** grandes **bloques**:

- la **detección** de los eventos.
- las **acciones** asociadas a su tratamiento.

En función del **origen del evento** (dónde se haya producido), diferenciamos entre:

• Eventos **internos**. Este tipo de eventos está producido por el **propio sistema**.

• Eventos **externos**. Los eventos externos son aquellos producidos por el **usuario**, habitualmente, a través del teclado o del puntero del ratón.

Los **objetos** que definen todos los **eventos** que se verán en este tema se basan en las siguientes clases:

Clase	Descripción
EventObject	Derivan **TODOS** los eventos.
MouseEvent	Acción del **ratón** sobre el componente.
Component-Event	**Cambio de un componente (**tamaño, posición...).
ContainerEvent	**Añadir o eliminar componente** sobre un objeto de tipo **Container.**
WindowsEvent	**Variación en una ventana (**apertura o cierre, cambio de tamaño...).
ActionEvent	**Acción sobre un componente**. De los más **comunes**, modela acciones como la pulsación sobre un botón o el check en un menú de selección.

Componentes

Los **componentes** presentan habitualmente un tipo de **evento asociado**.

No es lo mismo el **tipo de detección** asociado a un **botón** o a la pulsación de una **tecla**, que la forma de detección de la apertura o cierre de una **ventana**.

Asociación del evento al componente

Paso 1: crear un componente:

```
JButton btnInicializar = new JButton("Inicializar");
```

Paso 2: añadir al JPanel:

```
contentPane.add(btnInicializar);
```

Paso 3: añadir el escuchador:

```
btnInicializar.addActionListener(new ActionListener() {...}
```

Paso 4: programar la acción:

```
@Override
public void actionPerformed(ActionEvent e) {...}
```

En la siguiente tabla, se muestran los componentes más habituales y el tipo de evento asociado a estos.

Nombre componente	Nombre evento	Descripción del evento
JTextField	**ActionEvent**	Detecta la **pulsación** de la tecla **Enter** tras completar un **campo de texto**.
JButton	**ActionEvent**	Detecta la **pulsación** sobre un componente de tipo **botón**.
JComboBox	**ActionEventItemEvent**	Se detecta la **selección** de uno de los valores del **menú**.
JCheckBox	**ActionEventItemEvent**	Se detecta el **marcado** de una de las **celdas** de selección.
JTextComponent	**TextEvent**	Se produce un **cambio** en el **texto**.
JScrollBar	**AdjustmentEvent**	Detecta el **movimiento** de la barra de desplazamiento (**scroll**).

Listeners

Los listeners o escuchadores quedan a la espera («escuchando») si ese componente produce un evento. Si este se produce, se ejecutan las acciones asociadas a tal ocurrencia. Todo **evento requiere de un listener** que controle su **activación**.

Tipos de **listeners asociados** al tipo de **evento** al que corresponden:

Un mismo **tipo de escuchador** (**listener**) puede estar presente en **varios eventos** y componentes diferentes, aunque, normalmente, presentan un **comportamiento muy similar**.

KeyListener

Al pulsar cualquier **tecla**. Se contemplan varios **tipos de pulsaciones**, cada uno de los cuales presentará un método de control propio. Se implementan los **eventos ActionEvent**.

- **KeyPressed**: Se produce al **pulsar** la tecla.
- **KeyTyped**: Se produce al **pulsar y soltar** la tecla.
- **KeyReleased**: Se produce al **soltar** una tecla.

ActionListener

Pulsación (ratón o tecla) sobre un componente. Está presente **en varios tipos de elementos** y es uno de los escuchadores más **comunes**.

La detección tiene lugar ante **dos tipos de acciones**:

- ➔ Pulsación sobre el componente con la tecla **Enter**, siempre que el **foco** esté sobre el elemento;
- ➔ Pulsación sobre el componente con el puntero del **ratón**.

Estos **componentes** implementan los **eventos** de tipo **ActionEven**:

Componente asociado a ActionListener

- **JButton**: Al hacer **clic** sobre el botón o pulsar la tecla **Enter** con el **foco** situado sobre el componente.
- **JtextField**: Al pulsar la tecla **Enter** con el **foco** situado sobre la **caja de texto**.
- **JmenuItem**: Al **seleccionar** alguna **opción** del componente menú.
- **JList**: Al hacer **doble clic** sobre uno de los **elementos** del componente **lista**.

MouseListener

Este evento se produce al hacer **clic** con el ratón sobre algún **componente**. Es posible **diferenciar** entre distintos tipos de **pulsaciones** y asociar a cada una de ellas una **acción diferente**.

Estos **componentes** implementan los **eventos** de tipo **MouseEvent**:

Componente asociado a **MouseListener**

- **mouseClicked**: Se produce al **pulsar y soltar** con el puntero del ratón sobre el componente.
- **mouseEntered**: Se produce al **acceder** a un componente utilizando el puntero del ratón.
- **mouseExited**: Se produce al **salir** de un componente utilizando el puntero del ratón.
- **mousePressed**: Se produce al **presionar** sobre el componente con el puntero.
- **mouseReleased**: Se produce al **soltar** el puntero del ratón.

MouseMotionListener

Este evento se produce ante la detección del **movimiento** del ratón.

Componente asociado a MouseMotionListener

- **mouseMoved**: Se produce al **mover** sobre un componente el **puntero** del ratón.
- **mouseDragged**: Se produce al **arrastrar un elemento** haciendo clic previamente sobre él.

FocusListener

Este evento se produce cuando un elemento está **seleccionado** o deja de estarlo, es decir, al tener el **foco** sobre el componente o dejar de tenerlo.

Se implementan objetos de la clase de **eventos FocusEvent**.

Diferencia entre eventos y escuchadores:

- ***Evento*** *es hacer que* ***algo suceda****, que los programas cobren vida.*
- ***Escuchador*** *o listener es el encargado de* ***escuchar los eventos*** *que suceden. Gracias a ello se logra que suceda lo que queramos cuando se detecta que ha ocurrido el evento deseado. Los listeners se encargarán de* ***controlar los eventos****,* ***esperando*** *a que el evento se produzca para realizar una serie de acciones.* ***Según el evento****, necesitaremos* ***un listener u otro*** *que lo controle.*

Métodos

Cada uno de los **eventos** utilizará un **método** para el **tratamiento** del mismo.

Tras enlazar al escuchador con la ocurrencia de un evento, será necesario ejecutar un método u otro **en función del tipo de evento** asociado.

Relación entre el Listener y el método propio de cada evento

ActionListener:

public void **actionPerformed**(ActionEvent e)

KeyListener:

public void **keyPressed**(KeyEvent e)

public void **keyTyped**(KeyEvent e)

public void **keyReleased**(KeyEvent e)

MouseEntered

public void **focusGained**(FocusEvent e)

public void **lostGained**(FocusEvent e)

MouseListener

public void **mouseClicked**(MouseEvent e)

public void **mouseExited**(MouseEvent e)

public void **mousePressed**(MouseEvent e)

public void **mouseReleased**(MouseEvent e)

public void **mouseEntered**(MouseEvent e)

MouseMotionListener

public void **mouseMoved**(MouseEvent e)

public void **mouseDragged**(MouseEvent e)

```
// Ejemplo de uso con evento sobre Button

JButton btn = new JButton("botón");
btn.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {...}

...}
```

Asociación de acciones a eventos

Creación del listener

Se crea una nueva **instancia del evento** y este se **vincula con el componente** sobre el que va a actuar.

Para que se produzca esta detección, será necesario **enlazar el evento con los escuchadores**, elementos que se encuentren a la espera de que se produzca algún evento (Listeners). Estos elementos son interfaces que implementan un conjunto de métodos.

La implementación puede realizarse:

- **Añadiendo el evento directamente al componente** y, posteriormente, codificando las **acciones** que se van a llevar a cabo (**Sintaxis 1**).
- Modelando primero todo el tratamiento del **evento** y, a continuación, **asociándolo con el componente** sobre el que actúa (**Sintaxis 2**).

Sintaxis 1:

```
nombreComponente.addtipoEventoListener(new tipoEventoListener() {...}
```

Sintaxis 2:

```
tipoEventoListener nombreEvento = new tipoEventoListener (){...}

nombreComponente.addtipoEventoListener(nombreEvento);
```

El valor de **tipoEventoListener** se obtiene escogiéndolo **en función del evento que se vaya a tratar**.

Asociación de la acción al evento

Cuando se activa y vincula un escuchador o listener a un componente y, por tanto, a la ocurrencia de un evento, los componentes **no realizan un filtrado previo de los eventos** para determinar si los manejan o no, sino que los reciben todos. A través de la **asociación de la acción al evento** se determinará **si se maneja el evento** o no.

A continuación, se implementa el **método** bajo el cual se **desarrolla la acción**, que se ejecutará tras la **ocurrencia** del evento.

En el lenguaje de programación Java, cada **evento está asociado a un objeto** de la clase **EventObject** y, por lo tanto, a un método concreto.

Estructura general para la **definición de este método**:

```
public void métodoDeEvento(TipoEvento e){...}
```

Los métodos relativos a cada evento, prestando especial atención a las diferentes casuísticas que presentan algunos eventos, son los estudiados en el apartado 4.

Ejemplo de detección de la pulsación sobre un botón

Se mostrará el mensaje *Hola Mundo* en la misma ventana en la que se encuentra el botón.

En **primer lugar**, se crea una nueva **clase JFrame** y se inserta un **panel JPanel** para ubicar encima el resto de elementos.

Se coloca una **etiqueta** y un **botón**, en la distribución que se desee, desde la vista de diseño. En la parte del código, de forma automática, se habrá generado:

```
// Se coloca una etiqueta en Panel
JLabel lblNewLabel = new JLabel("...");
// Se añade al panel
contentPane.add(lblNewLabel);
// Se crea un nuevo botón y añade al panel
JButton btnNewButton = new JButton("Pulsa aquí");
contentPane.add(btnNewButton);
```

Crea el **código** relativo a la **producción y detección de eventos** para cada componente:

Desde la vista de **diseño**, hacemos doble clic sobre el botón, lo que nos lleva al código, donde ahora aparecen algunas líneas nuevas.

```
btnNewButton.addActionListener(new ActionListener({
```

Se **implementa** tanto **el** `escuchador` vinculado al botón (ActionListener) como **el** `método` dentro del cual se desarrollan las **acciones desencadenadas** por el evento (actionPerformed), que recibe por **parámetro** un **`ObjectEvent`** de **tipo** `ActionEvent`.

Finalmente, solo quedará colocar el **código** que envía a la **etiqueta de texto** creada en el inicio el **mensaje** «Hola Mundo» y la muestra por pantalla.

```
        public void actionPerformed(ActionEvent e) {
                lblNewLabel.setText("Hola mundo");
        }
});
```

Pruebas unitarias

El desarrollo de **pruebas** de software es importante en la implementación de **nuevos componentes**, puesto que, de esta forma, se reducen considerablemente los tiempos de desarrollo.

Si se ha **probado y verificado el comportamiento** de un componente, esta acción ya no será necesaria, aunque sí habría que **verificar** cómo sería su **uso en el marco de otro proyecto** que lo utilice.

Las pruebas se diseñan en base al **comportamiento esperado** de un componente, extendiéndose a todas las **casuísticas** posibles.

Las pruebas unitarias son pruebas que se realizan sobre una **funcionalidad concreta**, es decir, sobre una parte del programa, con el objetivo de comprobar si funciona de forma correcta. Para el desarrollo de pruebas unitarias, encontramos el framework de **Java**, **JUnit**. Para **intalarlo**, basta con incorporar algunas **librerías JAR** al **entorno** de desarrollo.

Como resultado de las pruebas, se distinguen dos tipos de escenarios: la **respuesta deseada** y la respuesta **real**. Cuando estas **no coinciden**, será necesario hacer una **revisión completa del código** que se está probando.

Además, las pruebas unitarias deben cumplir las características del conocido como **principio FIRST**.

- **Fast**: **Rápida** ejecución.
- **Isolated**: **Independencia** respecto a otros test.
- **Repeatable:** Se pueda **repetir** en el tiempo.
- **Self-Validating:** **Cada test** debe poder **validar si es correcto** o no a sí mismo.
- **Timely:** ¿Cuándo se deben desarrollar los test? ¿Antes o después de que esté todo implementado? Sabemos que cuesta **hacer primero los test y después la implementación** (TDD: Test-driven development), pero es lo suyo para **centrarnos en lo que realmente se desea** implementar.

JUnit en Eclipse

Para trabajar con JUnit desde Eclipse, lo primero que necesitamos es tener nuestro proyecto creado y añadir la **librería JUnit desde Build Path**. La manera más sencilla es haciendo clic con el botón derecho sobre el proyecto y seleccionando:

- ➔ *Build Path*
  - ➔ *Add Libraries*
    - ➔ *Junit*

Después, nos aparecerá una pantalla con las versiones de la librería disponibles, y seleccionamos JUnit 5.

Ejemplo clase Prueba unitaria con JUnit5

Crearemos nuestra clase de prueba, que se llamará **como la clase que queremos probar**, pero con la palabra '**test' delante**. Si la clase que queremos probar se llama calculadora.java, la clase de prueba se llamara TestCalculadora.java. Además, esta nueva clase debe **extender de la clase TestCase** e **importar junit.framework.TestCase**.

```
import junit.framework.TestCase;
public class TestCalculadora extends TestCase {...}
```

Métodos de prueba

Los **métodos** de esta clase serán **similares** a los de la **clase que se quiere probar**, pero con la palabra '**test' delante**, es decir, si el método original es sumar, el método de prueba será testSumar().

Importante: este método **no devolverá nada**, por lo que se declarará como public **void** TestSumar(). Dentro de cada método, llamaremos al constructor de nuestra clase de prueba y, para **comprobar** si el **resultado obtenido** coincide con el **esperado**, utilizaremos los **métodos assert**. **Los más utilizados** según el tipo de dato que queramos comprobar son:

- **assertTrue**.
- **assertFalse**.
- **assertEquals.**
- **assertNull**.

Comprobación de errores

Por último, comprobaremos los resultados compilando la clase prueba como: **Run As, JUnit Test**. En ese momento, aparecerá un **panel llamado JUnit**, en forma de **árbol**, que mostrará los **resultados correctos** en color verde y los **errores** en rojo.