**Conocimiento: Manejo de Eventos en Java**

Cuando tenemos una aplicación que contiene el método main, es dicho método quién decide el orden de ejecución de las operaciones.

Al utilizar aplicaciones con interfaz gráfica de usuario, el orden en que se ejecutan las operaciones dependerá de la interacción del usuario con los diferentes componentes de dicha interfaz.

Se entiende como evento a la operación que tiene como resultado una acción tras la intervención del usuario. También se define como la encapsulación de una información que puede ser enviada de manera asíncrona a la aplicación y pueden corresponder a a acciones físicas (Mouse o teclado) y acciones lógicas.

En java, los eventos forman parte del paquete **java.awt.event** y están organizados en una jerarquía de clases dentro de dicho paquete:

* La clase **java.util.EventObject**: es la clase base de todos los eventos en java
* La subclase **java.awt.AWTEvent**: es la clase base de todos los eventos que se utilizan en la construcción de interfaces gráficas de usuarios.
* Cada tipo de evento XxxEvent tiene asociada una interfaz XxxListener que es la que permite definir manejadores de eventos.



Este esquema permite visualizar un ejemplo del modelo de evento que maneja java

El modelo de eventos de java hace uso de fuentes de eventos (Source) y receptores de eventos (Listener). Una fuente de eventos es un objeto que tiene la capacidad de detectar eventos y notificar a los receptores de eventos que se han producido esos eventos. Aunque el programador puede establecer el entorno en que se producen esas notificaciones, siempre hay un escenario por defecto.

Un objeto receptor de eventos es una clase (o una subclase de una clase) que implementa un interfaz receptor específico. Hay definidos un determinado número de interfaces receptores, donde cada interfaz declara los métodos adecuados al tratamiento de los eventos de su clase. Luego, hay un emparejamiento natural entre clases de eventos y definiciones de interfaces. Por ejemplo, hay una clase de eventos de ratón que incluye muchos de los eventos asociados con las acciones del ratón, y hay un interfaz que se utiliza para definir los receptores de esos eventos.

**Clases de Eventos**

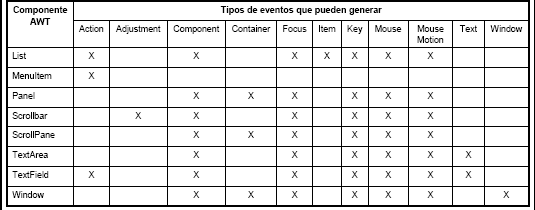
Las clases que representan eventos son el núcleo del mecanismo de gestión de eventos de java. Estas clases proporcionan una forma fácil y potente de encapsular eventos y todas heredan de **EvenObject**, que se encuentra en java.util.

La clase **AWTEvent** es subclase de EvenObject y se encuentra en el paquete java.awt, de ella heredan una serie de clases encargadas del manejo de eventos físicos (clickear el Mouse) y semánticos (ajustar, incorporara, borrar).

La siguiente tabla resume las principales clases de eventos en java.awt.event.

|  |  |
| --- | --- |
| **Nombre Clase** | **Descripción** |
| ActionEvent | Esta clase se genera cuando se pulsa un botón, se hace doble clic en un elemento de una lista o cuando se selecciona un elemento de un menú |
| AdjustmentEvent | Esta clase se genera cuando se manipula una barra de desplazamiento |
| ComponentEvent | Esta clase se genera cuando un componente se oculta, se mueve, cambia de tamaño o se hace visible |
| ContainerEvent | Esta clase se genera cuando un componente se añade o se elimina de un contenedor |
| FocusEvent | Esta clase se genera cuando un componente recibe o pierde el foco del teclado |
| InputEvent | Esta superclase es abstracta. Está definida para cualquier clase de evento de entrada de componente |
| ItemEvent | Esta clase se genera cuando se hace clic en una casilla de verificación o en un elemento de una lista, también cuando se selecciona una opción o cuando se selecciona o deselecciona un elemento de un menú de opciones |
| KeyEvent | Esta clase se genera cuando se recibe una entrada desde el teclado |
| MouseEvent | Esta clase se genera cuando se arrastra, se mueve, se hace clic, se pulsa o se libera el Mouse, también cuando el Mouse entra o sale de un componente. |
| MouseMotionEvent | Esta clase se genera cuando se arrastra o se mueve el Mouse |
| MouseWheelEvent | Esta clase se genera cuando se mueve la rueda del Mouse |
| TextEvent | Esta clase se genera cuando se cambia el valor de un campo o de un área de texto |
| WindowEvent | Esta clase se genera cuando una ventana se activa, se cierra, se desactiva, se minimiza , se abre o se sale de ella. |

Cada componente de la interfaz gráfica puede gatillar o activar un evento, esto se muestra en la siguiente tabla:



Para interpretar esta tabla, por ejemplo un objeto de tipo Button (botón) puede generar eventos de tipo ActionEvent, ComponentEvent, FocusEvent, KeyEvent, MouseEvent y MouseMotionEvent, pero nunca generar un evento de tipo ItemEvent.

**Interfaces de oyentes de eventos**

El modelo de delegación de eventos tiene dos componentes fundamentales: fuentes y oyentes.. Los oyentes se crean implementando una o más interfaces que se definen en el paquete java.awt.event. Cuando un evento se da, la fuente del evento llama al método adecuado que define el oyente y proporciona un objeto de evento como su argumento.

La siguiente tabla muestra un listado de interfaces de oyentes de uso más común y de los métodos que define:

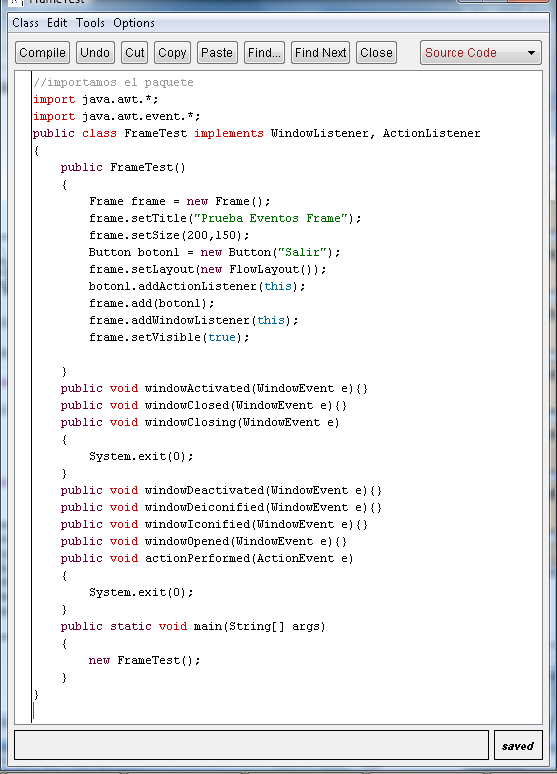
|  |  |
| --- | --- |
| **Nombre Interfaz** | **Descripción** |
| ActionListener | Eventos de acción, método:  void actionPerformed(ActionEvent e) |
| AdjustementListener | Evento de ajuste, método:  void adjustmentValueChanged(AdjustmentEvent e) |
| ComponentListener | Evento de cambio de un componente, métodos:  void componentResized(ComponentEvet e)  void componentMoved(ComponentEvet e)  void componentShown(ComponentEvet e)  void componentHidden(ComponentEvet e) |
| ContainerListener | Evento de contenedor, métodos:  void componentAdded(ContainerEvent e)  void componentRemoved(ContainerEvent e) |
| FocusListener | Evento de foco del teclado, métodos:  void focusGained(FocusEvent e)  void focusLost(FocusEvent e) |
| ItemListener | Evento de cambio de ítem, método:  void itemStateChanged(ItemEvent e) |
| KeyListener | Evento de teclado, métodos:  void keyPressed(KeyEvent e)  void keyRelease(KeyEvent e)  void keyTyped(KeyEvent e) |
| MouseListener | Evento de acción de Mouse, métodos:  void mouseCliked(MouseEvent e)  void mouseEntered(MouseEvent e)  void mouseExited(MouseEvent e)  void mousePressed(MouseEvent e)  void mouseReleased(MouseEvent e) |
| MouseMotionListener | Evento de arrastre de Mouse, métodos:  void mouseDragged(MouseEvent e)  void mouseMoved(MouseEvent e) |
| MouseWheelListener | Evento de la rueda del Mouse, método:  void mouseWheelMoved(MouseWheelEvent e) |
| TextListener | Evento de cambio de texto, método:  void textChanged(TextEvent e) |
| WindowFocusListener | Evento de foco de ventana, métodos:  void windowGainedFocus(WindowEvent e)  void windowLostFocus(WindowEvent e) |
| WindowListener | Evento de manipulación de ventana, métodos:  void windowActivated(WindowEvent e)  void windowClosed(WindowEvent e)  void windowClosing(WindowEvent e)  void windowDeactivated(WindowEvent e)  void windowDeiconified(WindowEvent e)  void windowIconified(WindowEvent e)  void windowOpened(WindowEvent e) |

La implementación de los eventos debe considerar lo siguiente:

* Una fuente puede generar varios tipos de eventos
* Cada evento debe ser registrado por separado
* Se debe implementar todas las interfaces necesarias para recibir todos los eventos posibles.

En este punto ya se conoce los eventos, los oyentes y las interfaces que contienen los métodos de manejo de eventos y que deben ser sobrescritos por la aplicación. A continuación veremos un ejemplo de creación de una ventana y un botón, y se programará el cierre de la ventana y el evento de cierre del botón.

El código sería:



Analicemos este ejemplo:

1. En la cáscara se agregan las interfaces necesarias:

public class FrameTest implements WindowListener, ActionListener

La interfaz WindowListener para el evento de cierre de la ventana

La interfaz ActionListener para el evento del botón.

1. Se agrega la implementación del oyente a cada componente a través del método de la clase que se encarga de eso:

boton1.addActionListener(this);

frame.addWindowListener(this);

1. Se implementan los métodos de cada interfaz, dándole el código necesario

Para la interfaz WindowListener, se implementa:

public void windowActivated(WindowEvent e){}

public void windowClosed(WindowEvent e){}

public void windowClosing(WindowEvent e)

{

System.exit(0);

}

public void windowDeactivated(WindowEvent e){}

public void windowDeiconified(WindowEvent e){}

public void windowIconified(WindowEvent e){}

public void windowOpened(WindowEvent e){}

Para la interfaz ActionListener, se implementa:

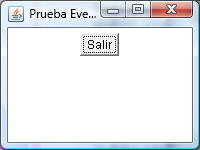
public void actionPerformed(ActionEvent e)

{

System.exit(0);

}

Al ejecutar este código aparece la siguiente ventana:



Al presionar el botón la ventana se cierra, igual al presionar la ventana en la parte superior derecha, sobre el símbolos también se cierra.

**Clase Adaptadora**

Java proporciona una característica especial, llamada clase Adaptadora, que en algunas circunstancias puede simplificar la creación de los gestores de eventos.

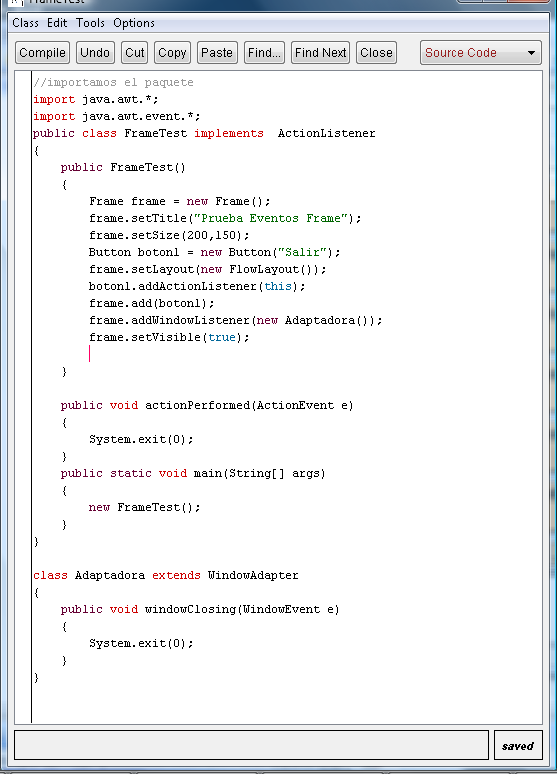
Una clase adaptadora proporciona una implementación vacía de todos los métodos de una interfaz. Estas clases son útiles cuando se quieren recibir y procesar solo algunos eventos que gestiona la interfaz oyente de eventos, por lo que se puede heredar de esa clase adaptadora y sobrescribir el o los métodos necesarios para la implementación.

Cada interfaz oyente que tenga más de un método definido tiene una clase adaptadora.

La siguiente tabla representa las principales clases adaptadoras:

|  |  |
| --- | --- |
| **Clase Adaptadora** | **Interfaz Oyente** |
| ComponenteAdapter | ComponentListener |
| ContainerAdapter | ContainerListener |
| FocusAdapter | FocusListener |
| KeyAdapter | KeyListener |
| MouseAdapter | MouseListener |
| MouseMotionAdapter | MouseMotionListener |
| WindowAdapter | WindowListener |

Por ejemplo, se modifica el código anterior reemplazando la implementación de la interfaz WindowListener por la extensión de la clase WindowAdapter, de la siguiente manera:



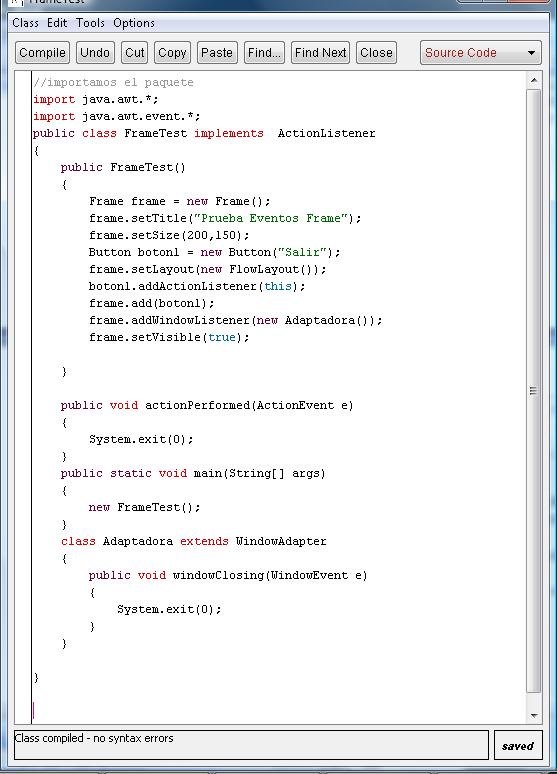
Este código implementa el método que permite cerrar la ventana partir de la extensión de la clase adaptadora WidowAdapter, sobrescribiendo el método windowClosing() solamente, el resto no es necesario que esté, por tanto se ahorra código.

**Clases Internas**

Las clases internas son clases de java que se definen dentro de un bloque de código, puede ser dentro de una clase, dentro de un método o dentro de un ciclo.

Estas clases permiten simplificar el código cuando se usan clases adaptadoras.

Realicemos el mismo ejemplo anterior pero la clase Adaptadora va a ser interna, entonces el código quedaría:

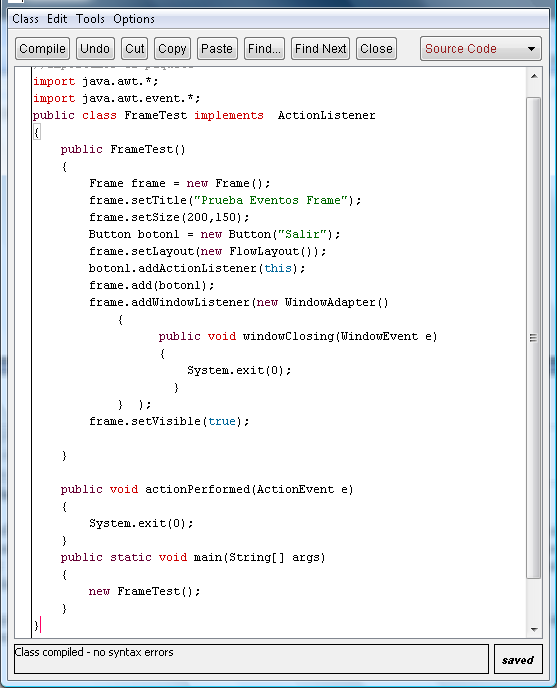


En este ejemplo, la clase Adaptadora está dentro de la clase FrameTest.

**Clase interna Anónima**

La utilidad de una clase interna se resalta al utilizarla anónima. Una clase anónima es aquellas que no tiene nombre alguno, y que permite facilitar la escritura ya que cada vez que necesitemos implementar un método de oyente se puede dar el código adecuada a cada caso.

Por ejemplo, veamos el código anterior al que se le implementa una clase anónima que sobrescriba el método windowClosing().



En este caso al agregar el oyente a la venta se instancia la clase WindowAdapter y se sobrescribe el método necesario. Esta es la forma más corta para implementar oyentes.

**Modelo Vista Controlador**

Modelo Vista Controlador (MVC) es un patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos. El patrón MVC se ve frecuentemente en aplicaciones web, donde la vista es la página HTML y el código que provee de datos dinámicos a la página, el modelo es el Sistema de Gestión de Base de Datos y la Lógica de negocio y el controlador es el responsable de recibir los eventos de entrada desde la vista.

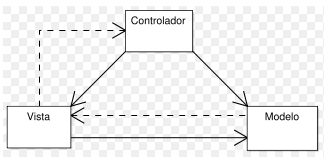
Descripción del patrón

**Modelo**: Esta es la representación específica de la información con la cual el sistema opera. La lógica de datos asegura la integridad de estos y permite derivar nuevos datos; por ejemplo, no permitiendo comprar un número de unidades negativo, calculando si hoy es el cumpleaños del usuario o los totales, impuestos o importes en un carrito de la compra.

**Vista**: Este presenta el modelo en un formato adecuado para interactuar, usualmente la interfaz de usuario.

**Controlador**: Este responde a eventos, usualmente acciones del usuario e invoca cambios en el modelo y probablemente en la vista.

El siguiente esquema representa la relación entre estos tres componentes:



Este modelo de arquitectura presenta varias ventajas:

* Hay una clara separación entre los componentes de un programa; lo cual nos permite implementarlos por separado
* Hay un API muy bien definido; cualquiera que use el API, podrá reemplazar el Modelo, la Vista o el Controlador, sin aparente dificultad.
* La conexión entre el Modelo y sus Vistas es dinámica; se produce en tiempo de ejecución, no en tiempo de compilación.

Al incorporar el modelo de arquitectura MVC a un diseño, las piezas de un programa se pueden construir por separado y luego unirlas en tiempo de ejecución. Si uno de los Componentes, posteriormente, se observa que funciona mal, puede reemplazarse sin que las otras piezas se vean afectadas.

Aunque se pueden encontrar diferentes implementaciones de MVC, el flujo que sigue el control generalmente es el siguiente:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace)

2. El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos.

3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario). Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.

4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista. Sin embargo, el patrón de observador puede ser utilizado para proveer cierta indirección entre el modelo y la vista, permitiendo al modelo notificar a los interesados de cualquier cambio. Un objeto vista puede registrarse con el modelo y esperar a los cambios, pero aun así el modelo en sí mismo sigue sin saber nada de la vista. El controlador no pasa objetos de dominio (el modelo) a la vista aunque puede dar la orden a la vista para que se actualice. Nota: En algunas implementaciones la vista no tiene acceso directo al modelo, dejando que el controlador envíe los datos del modelo a la vista.

5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

**Conocimiento: Implementación de la Interfaz**

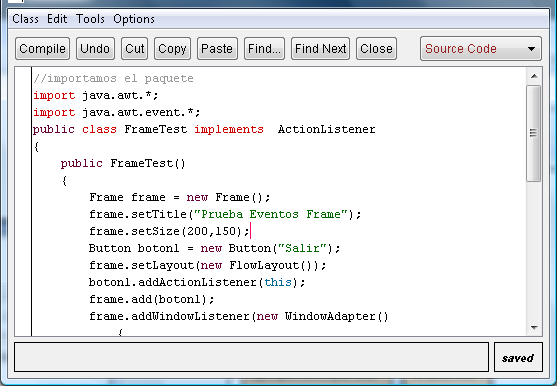
La implementación de una interfaz gráfica de usuario debe contener un contenedor, componentes, un diseño y eventos adecuados a la lógica del negocio.

Para poder implementar contenedores se puede utilizar una de las siguientes técnicas:

1. Por Colaboración
2. Por Herencia

Implementación por Colaboración:

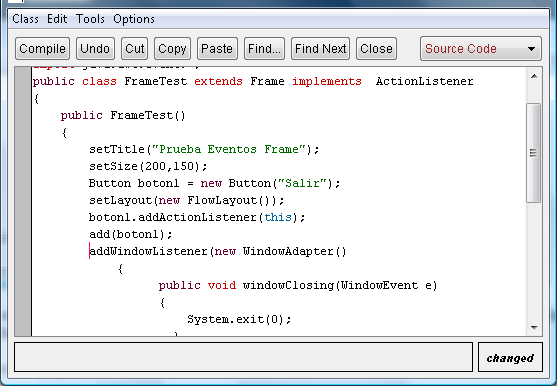
En esta técnica se crea una nueva clase donde el contenedor sea una variable de la clase, por ejemplo:



En este ejemplo, se crea un objeto de la clase Frame, y para acceder a cada método de la clase Frame se debe hacer a través del operador punto.

Implementación por Herencia

En esta técnica se extiende de un contenedor, haciéndolo como parte superior de la clase en que se trabaja, por ejemplo:



En este ejemplo en la cáscara se manifiesta la herencia de la clase Frame, y para acceder a cada método de dicha clase basta con invocarlo directamente.