Programación basada en eventos

Tipos de programas en Java

- Aplicaciones
  - Se pueden ejecutar directamente en un entorno Java
  - Tipos
    - Modo de consola
      - ***Interacción mediante teclado***
      - ***Interfaz basado en texto***
    - Aplicaciones con interfaz gráfico (GUI)
      - ***Ventanas graficas para entrada y salida de datos***
      - ***Iconos***
      - ***Dispositivos de entrada (e.g. ratón, teclado)***
      - ***Interacción directa***
- Applets
  - Pequeñas aplicaciones que se ejecutan dentro de un navegador (o en el visualizador de applets - Appletviewer)
    - Interfaz gráfico
    - Limitaciones por motivos de seguridad

Creación de una interfaz gráfico de usuario

- Composición de la interfaz gráfica de la aplicación (GUI)
  - Elección de un contenedor (ventana) en la que se incluyen el resto de los elementos gráficos de interacción
  - Diseño del interfaz gráfico añadiendo componentes gráficos de interacción (p.e. Botones, etiquetas, menús, ...)
- Establecer los gestores de eventos para responder a las interacciones de los usuarios con la interfaz gráfica
- Visualizar la interfaz gráfica
  - Java lanza un hilo que se encarga de gestionar la interacción del usuario con la interfaz gráfica,
    - por tanto el método principal puede acabar después de haberse creado y visualizado la interfaz

Programación basada en eventos

- Eventos o sucesos
  - Mensajes asíncronos producidos por interacción
    - Interacción del usuario con la interfaz gráfica de un programa
      - ***Ejemplo:***
        Interacción con el ratón (hacer click, arrastrar, mover, etc)
        Introducir texto en un determinado campo
  - Estos mensajes son objetos
- La programación basada en eventos consiste en proporcionar métodos que respondan a dichos mensajes

Modelo de evento Java 1.0

- Basado en herencia
- Un programa gestiona el tratamiento de eventos de un GUI
  - Especializando componentes del GUI y reescribiendo los métodos *action()* o *handleEvent()*
    - Devuelve *true* --> evento tratado
    - Devuelve *false* --> el evento se propaga hacia arriba en la jerarquía
- Estructuración del código
  - Cada componente se especializa para que gestione sus eventos
    - Muchas clases muy pequeñas
  - Todos los eventos de un grupo de componentes se tratan en un componente que los contenga
    - Los métodos action() o handleEvent() del componente contenedor debe tener una instrucción de selección para determinar a que elemento corresponde

Modelo de evento Java 1.0 - Problemas

- Muy complejo para interfaces con muchas funcionalidades
  - Poco escalable
- No permite una separación clara entre la aplicación y la interfaz (GUI)
- Todos los eventos se tratan mediante los mismos métodos
  - Muchas posibilidades de error
  - No hay filtrado de eventos
    - Problemas de eficiencia

Modelo de evento Java 1.1

- Objetivos de diseño
  - Simple y fácil de aprender a la vez que versátil
  - Permite una separación clara entre el código de la aplicación y el de la interfaz
  - Facilita la creación de código robusto para la gestión de eventos
  - Soporte en ejecución para las herramientas visuales
    - Eventos generados
    - Eventos tratados

Modelo de delegación

- Delegación
  - La responsabilidad de gestionar un evento, que ocurre en un objeto (fuente), la tiene otro objeto (oyente)
- Eventos
  - Eventos representados como objetos en una jerarquía de clases
    - Una subclase para cada evento o tipo de eventos relacionados
    - Estos objetos encapsulan toda la información del evento
      - ***El tipo de evento, por ejemplo, pulsar el ratón***
      - ***El objeto fuente sobre el que se produjo el evento (e.g. un botón)***
      - ***El instante de tiempo en el que se produjo el evento***
      - ***La posición (x, y) donde se produjo el evento***
      - ***La tecla que se pulsó (para eventos de teclado)***
      - ***El estado de las teclas modificadoras (CTRL, SHIFT, ...) presionadas o no por el usuario cuando sucedió el evento***
- Propagación de eventos de una *fuente* a un *oyente*
  - Invocación desde la fuente de un método del oyente pasándole el objeto que define el tipo de evento generado

Fuentes y Oyentes

- Fuente (Source)
  - Objeto que origina o lanza eventos
    - En el API de cada objeto se define el conjunto de eventos que origina
  - Proporciona un conjunto de métodos para registrar y eliminar oyentes específicos para estos eventos
    - set<TipoEvento>Listener - un único oyente
    - add<TipoEvento>Listener - múltiples oyentes
    - remove<TipoEvento>Listener - elimina un oyente
- Oyente (Listener)
  - Objeto que gestiona o responde a los eventos
  - Define uno o más métodos a ser invocados por la fuente de eventos en respuesta a cada evento específico
  - Objeto que implementa el interfaz *<TipoEvento>Listener*

Uso del modelo

- Los objetos que desean gestionar los eventos y recibir dichas notificaciones tienen que registrarse como oyentes e implementar los métodos de la interfaz correspondiente
- Cuando ocurre un evento la fuente informa a los oyentes registrados invocando a dichos métodos (*callback*)
- En un programa Swing, normalmente la fuente de eventos es un componente GUI y el oyente es un objeto "adaptador" que implementa el o los oyentes adecuados para que la aplicación gestione los eventos
- El oyente también puede ser otro componente Swing que implementa uno o más interfaces oyentes para agrupar objetos del GUI

Jerarquía de clases de eventos

- Cada clase representa un tipo de evento o un grupo de eventos relacionados
  - Una clase puede representar varios tipos de eventos
    - MouseEvent: mouse up, mouse down, mouse drag, mouse move
- Raíz de la jerarquía
  - *java.util.EventObject*
- Los programas pueden crear nuevos tipos de eventos especializando *java.util.EventObject* o los eventos AWT *java.awt.AWTEvent*
  - Los eventos del AWT se encuentran en el paquete *java.awt.event*

Gestion de eventos

- 1. Una clase que desee gestionar un evento debe implementar un interfaz (e.g. ActionListener)
  - class ClaseGestor implements ActionListener { ….}
- 2. Establecer la relación entre la fuente de eventos (componente) y el objetoClaseGestor que gestiona el evento
  - componente.addActionListener(objetoClaseGestor)
- 3. Implementación del método o métodos del interfaz en la clase que gestiona el evento
  - public void actionPerformed(ActionEvent ev) {
    // codigo que implementa la respuesta a la acción del usuario
    // sobre el componente
    }
  - Toda la información sobre el evento viene encapsulada en un objeto de una clase específica derivada de Event

Eventos: bajo nivel y semánticos

- Eventos de bajo nivel
  - Representan entradas o interacciones de bajo nivel con elementos del interfaz gráfico
    - ***Cambio de tamaño, cambio del foco, operación con el ratón o con el teclado***
- Eventos semánticos
  - Eventos de alto nivel que encapsulan la semántica del modelo de componentes del interfaz de usuario
    - ***Hacer una acción, un cambio de estado en un elemento, ...***
  - No están relacionados con una clase específica de componente sino que pueden aplicarse a todos los componentes que implementen un modelo semántico similar
    - Evento "action"
      - ***Lanzado por un botón cuando se pulsa una vez***
      - ***Lanzado por un elemento de una lista cuando se pulsa dos veces seguidas***
      - ***Lanzado por una opción de un menú cuando se selecciona***
      - ***Cuando se pulsa Enter en un campo de texto***

Eventos de bajo nivel	
Evento	**Significado**
ComponentEvent	Cambios en el tamaño, posición o visibilidad de un componente
FocusEvent	Cambio de foco (capacidad de un componente para recibir entradas desde el teclado)
KeyEvent	Operación con el teclado
MouseEvent	Operación con los botones del ratón o movimientos del ratón
WindowEvent	Cambio de estado en una ventana
AncestorEvent	Cambio en la composición, visibilidad o posición de un elemento superior (ancestro) de la jerarquía de composición
Eventos de alto nivel	
ActionEvent	Realización de la acción específica asociada al componente
ChangeEvent	Cambio en el estado del componente
ItemEvent	Elemento seleccionado o deseleccionado
CaretEvent	Cambio en la posición del cursor de inserción en un componente que gestiona texto
ListSelectionEvent	Cambio en la selección actual en una lista

Eventos AWT : bajo nivel y semánticos

bajo nivel

java.awt.AWTEvent
- java.awt.event.ComponentEvent (component resized, moved, etc.)
  - java.awt.event.FocusEvent (component got focus, lost focus)
  - java.awt.event.InputEvent
    - java.awt.event.KeyEvent (component got key-press, etc.)
    - java.awt.event.MouseEvent (component got mouse-down, etc.)
  - java.awt.event.ContainerEvent
  - java.awt.event.WindowEvent

semánticos

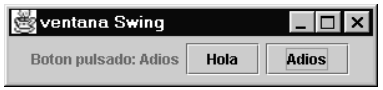
java.awt.AWTEvent
- java.awt.event.ActionEvent ("do a command")
- java.awt.event.AdjustmentEvent ("value was adjusted")
- java.awt.event.ItemEvent ("item state has changed")
- java.awt.event.TextEvent ("the value of the text object changed")

Ejemplo 1- tratamiento de eventos

```
import java.awt.event.*;
import javax.swing.*;
public class GUISimple extends JFrame {
        public GUISimple (){
                JPanel panel = new JPanel();
                JLabel etiqueta = new JLabel("Etiqueta 1");
                panel.add(etiqueta);
                JButton boton= new JButton("Correcto");
                panel.add(boton);
                GestorRaton oyenteBoton = new GestorRaton();
                boton.addActionListener(oyenteBoton);
                // se añade el panel al panel de contenido
                getContentPane().add(panel);
                setSize(250,80);
                setVisible(true);}
        public static void main(String args[]) {
                GUISimple ventana = new GUISimple();
                ventana.setTitle("ventana tipo JFrame"); }} // GUISimple
class GestorRaton implements ActionListener {
        public void actionPerformed(ActionEvent evento) {
                JButton boton = (JButton) evento.getSource();
                System.out.println("Se ha pulsado el boton: " + boton.getLabel());  }}
```

Ejemplo 9.1: Java:Iniciación y referencia

```
// importación de los componentes gráficos
import javax.swing.*;
// importación de las clases e interfaces para la gestión de eventos
import java.awt.event.*;
/* clase que implementa una ventana principal sencilla que
contiene una etiqueta y dos botones */
public class VentanaSimple extends JFrame {
        JLabel etiqueta;       // etiqueta
        JButton botonHola;  // botón de interacción
        JButton botonAdios; // botón de interacción
        // panel contenedor que agrupa a los otros componentes
        // no crea una ventana independiente
        JPanel panel;
```



Ejemplo 9.1: Java:Iniciación y referencia

```
public VentanaSimple (){
        // se crean los componentes de la ventana
        etiqueta = new JLabel("Etiqueta inicial");
        botonHola = new JButton("Hola");
        botonAdios = new JButton("Adios");
        panel = new JPanel();
        // se añaden los componentes al panel
        panel.add(etiqueta);
        panel.add(botonHola);
        panel.add(botonAdios);
        // añade el panel a la ventana principal de la aplicación
        getContentPane().add(panel);
        // se crea un objeto oyente de acción
        // que se registra en los dos botones
        OyenteAccion oyenteBoton = new OyenteAccion();
        botonHola.addActionListener(oyenteBoton);
        botonAdios.addActionListener(oyenteBoton);
  }
```


Ejemplo 9.1: Java:Iniciación y referencia

```
// metodo principal de la clase ventana simple
public static void main(String args[]) {
        // se crea un objeto de tipo ventana simple
        VentanaSimple ventana = new VentanaSimple();
        // se establecen distintas características de la ventana:
        // titulo, tamaño y que sea visible
        ventana.setTitle("ventana Swing");
        ventana.setSize(300, 70);
        ventana.setVisible(true);
  }

// oyente de eventos de acción (clase interna)
class OyenteAccion implements ActionListener {
        public void actionPerformed (ActionEvent evento){
           // se obtiene el botón fuente del evento
           JButton boton = (JButton) evento.getSource();
           // se modifica la etiqueta según el botón pulsado
           etiqueta.setText("Boton pulsado: " + boton.getText());
        }
  } // OyenteAccion
} // VentanaSimple
```


Ejemplo - separacion de GUI y aplicación-1

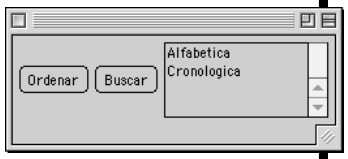
```
/* ejemplo de SUN sobre separacion entre GUI y aplicacion */
import java.awt.*;
import java.awt.event.*;

public class Aplicacion {
   public void buscar() {
      /* realiza la operacion de busqueda ...*/
      System.out.println("Buscando...");
   }
   public void ordenar() {
      /* realiza la operacion de ordenacion ...*/
      System.out.println("Ordenando....");
   }

   static public void main(String args[]) {
     Aplicacion aplicacion = new Aplicacion();
     GUI gui = new GUI(aplicacion);
   }
}// Aplicacion
```

Ejemplo- separacion de GUI y aplicación- 2

```
class GUI {
   public GUI(Aplicacion app) {
      Frame f = new Frame();
      f.setLayout(new FlowLayout());
      Orden ordenBuscar = new Orden(Orden.BUSCAR, app);
      Orden ordenOrdenar = new Orden(Orden.ORDENAR, app);
      Button b;
      f.add(b = new Button("Ordenar"));
      b.addActionListener(ordenOrdenar);
      f.add(b = new Button("Buscar"));
      b.addActionListener(ordenBuscar);
      List l;
      f.add(l = new List());
      l.add("Alfabetica");
      l.add("Cronologica");
      // el pulsado de dos veces sobre una opción de la lista produce una ordenación
      l.addActionListener(ordenOrdenar);
      f.pack(); // inicializa la ventana para presentación
      f.show();}} // GUI
```



Ejemplo- separacion de GUI y aplicación- 3

```
/** gestor de las ordenes del ratón */
class Orden implements ActionListener  {
   static final int BUSCAR = 0;
   static final int ORDENAR = 1;
   int id;
   Aplicacion aplicacion;
   public Orden(int id, Aplicacion app) {
      this.id = id;
      this.aplicacion = app;
   }
   public void actionPerformed(ActionEvent evento) {
      switch(id) {
        case BUSCAR:
          aplicacion.buscar();
          break;
        case ORDENAR:
          aplicacion.ordenar();
          break;
      }}} // Orden
```


Ejemplo con interfaz en Swing

```
import javax.swing.*;
class GUI {
   public GUI(Aplicacion app) {
      JFrame frame = new JFrame();
      Container panelContenido = frame.getContentPane();
      panelContenido.setLayout(new FlowLayout());
      Orden ordenBuscar = new Orden(Orden.BUSCAR, app);
      Orden ordenOrdenar = new Orden(Orden.ORDENAR, app);
      JButton boton;
      panelContenido.add(boton = new JButton("Ordenar"));
      boton.addActionListener(ordenOrdenar);
      panelContenido.add(boton = new JButton("Buscar"));
      boton.addActionListener(ordenBuscar);

      JCheckBox casilla = new JCheckBox("Alfabetica");
      panelContenido.add(casilla);
      casilla.addActionListener(ordenOrdenar);
      frame.setSize(100,200);
      frame.setVisible(true); }}
```

Clases adaptadoras de eventos

- Como hay interfaces oyentes que pueden “escuchar” distintos subtipos de eventos la clase gestora del evento debe implementar todos los métodos
  - MouseListener
    - ***Mouse-down, mouse-up, mouse-enter, etc***
- Para cada interfaz oyente que contenga más de un método en *java.awt.event* se define una clase abstracta adaptadora
  - Proporciona métodos vacios para los métodos de la interfaz
  - Nombrado: <Interfaz>Adapter
    - MouseAdapter, WindowAdapter, ContainerAdapter
- Simplifican el desarrollo ya que se pueden especializar los adaptadores y reescribir sólo los métodos relacionados con los eventos en los que está interesado

Clases adaptadoras de eventos

java.awt.event.ComponentAdapter
java.awt.event.ContainerAdapter
java.awt.event.FocusAdapter
java.awt.event.KeyAdapter
java.awt.event.MouseAdapter
java.awt.event.MouseMotionAdapter
java.awt.event.WindowAdapter

Clase adaptadora para el ratón

java.awt.event.MouseAdapter

Métodos

```
mousePressed(MouseEvent)
mouseReleased(MouseEvent)
mouseEntered(MouseEvent)
mouseExited(MouseEvent)
mouseClicked(MouseEvent) (sin equivalente en 1.0)
```


Ejemplo - Clase adaptadora para el ratón

```
// se implementan sólo aquellos métodos que nos interesan para
// responder a un evento concreto e.j. el pulsado del ratón

class ClaseGestorRaton extends MouseAdapter {
  public void mousePressed (MouseEvent evento) {
    System.out.println ("Boton ratón pulsado");
    if ((evento.getModifiers() & InputEvent.BUTTON3_MASK) != 0) {
              System.out.println ("El botón pulsado es el de la derecha");
             // mismo comportamiento con InputEvent.META_MASK
    }
  }
} // ClaseGestorRaton
```


Componentes y eventos

Componente Swing	Eventos que puede generar					
	Action	Item	Change	Caret	ListSelection	Otros eventos
JButton	X	X	X			
JCheckBox	X	X	X			
JComboBox	X	X				
JEditorPane				X		Document UndoableEdit Hyperlink
JFileChooser	X					
JList					X	ListData
JMenu						Menu
JMenuItem	X		X			Document UndoableEdit MenuDragMouse
JPasswordField	X			X		Document UndoableEdit
JPopupMenu						PopupMenu
JRadioButton	X	X	X			
JTabbedPane			X			
JTextArea				X		Document UndoableEdit
JTextField	X			X		Document UndoableEdit
JTextPane						UndoableEdit Hyperlink
JToggleButton	X	X	X			

Demasiadas clases

- Una interfaz de usuario normal puede tener muchos botones, componentes, menús, etc.
- Se necesita una clase diferente para cada oyente
  - Demasiadas clases a tener en cuenta
  - Confusión en el nombrado
- Solución en Java 1.1
  - Clases internas
  - Clases internas anónimas
    - Clases internas (anidadas) a otras clases que no tienen nombre
    - No necesitan tener un nombre ya que sólo se crea un ejemplar o instancia de cada una
- ¡ Cuidado con la legibilidad !

Ejemplo - Gestor de ventana anónimo

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

public class AdaptadorClaseInterna extends JFrame {
public AdaptadorClaseInterna(){
        setTitle("Ventana que se puede cerrar");
        setSize(300, 100);
        addWindowListener(
                new WindowAdapter() {
                        public void windowClosing(WindowEvent e){
                                System.exit(0); // salida de la aplicación
                        }
                });
    setVisible(true);
  }
   public static void main(String args[]) {
        AdaptadorClaseInterna ventana = new AdaptadorClaseInterna();
  }}
```


Tratamiento de eventos

- Para evitar problemas en el tratamiento de eventos hay que realizar las siguientes comprobaciones:
  - a) el oyente registrado es el más apropiado para el evento que se desea detectar;
  - b) este oyente se ha registrado en el componente adecuado; y
  - c) se ha implementado el método o métodos correspondientes en el oyente