CAPÍTULO 22

Gestión de eventos

Este capítulo examina un aspecto importante de Java: los eventos. La gestión de eventos es fundamental para la programación en Java porque se integra a la creación de applets y otros tipos de programas basados en interfaces gráficas. Como se explicó en el Capítulo 21, los applets son programas basados en eventos, que utilizan la interfaz gráfica de usuario, para interactuar con el usuario. Además, cualquier programa que utilice una interfaz gráfica de usuario, tal como una aplicación escrita en Java para Windows, se basa en eventos. Así que no se pueden escribir este tipo de programas sin un sólido dominio de manejo de eventos. Los eventos están soportados por un conjunto de paquetes, incluyendo **java.util**, **java.awt** y **java.awt.event**.

La mayoría de los eventos a los que un programa responderá son generados cuando el usuario interactúa con un programa basado en GUI. Este tipo de eventos son los que se examinan en este capítulo. Dichos eventos se pasan al programa de muchas formas diferentes, con el método específico dependiente del evento actual. Hay muchos tipos de eventos, incluidos aquellos que son generados por el ratón, el teclado, y otros controles GUI como los botones, scrollbar o checkbox.

Este capítulo comienza con una visión general del mecanismo de gestión de eventos de Java. Luego se examinan las interfaces y las clases de eventos principales utilizadas por el AWT y se desarrollan varios ejemplos que demuestran los fundamentos del procesamiento de eventos. En este capítulo también se explica cómo utilizar clases adaptadoras, clases internas y clases internas anónimas para estilizar el código al gestionar eventos. Los ejemplos que se muestran en el resto del libro hacen frecuentemente uso de estas técnicas.

NOTA *Este capítulo se enfoca en los eventos relacionados con los programas basados en GUI. Sin embargo, los eventos son también ocasionalmente utilizados para propósitos no directamente relacionados con programas basados en GUI. En todos los casos, las mismas técnicas básicas de gestión de eventos pueden ser aplicadas.*

Dos mecanismos para gestionar eventos

Antes de comenzar con la gestión de eventos, hay que dejar claro que la forma de gestionar los eventos ha cambiado significativamente entre la versión original de Java (1.0) y versiones posteriores de Java, comenzando por la versión 1.1. El modelo de gestión de eventos de la versión 1.0 todavía funciona, pero no se recomienda utilizarlo en programas nuevos. Además, muchos de los métodos que soportaba el antiguo modelo de eventos de la versión 1.0 se han quedado obsoletos. El nuevo modelo de gestión de eventos se debe utilizar en todos los programas nuevos, por ello será empleado por los programas en este libro.

El modelo de delegación de eventos

La propuesta actual de gestión de eventos se basa en lo que se llama *modelo de delegación de eventos*, el cual define mecanismos coherentes y estándares para generar y procesar eventos. Su concepto es muy sencillo: una *fuente* genera un *evento* y lo envía a uno o más *listeners*. En este esquema, los listeners simplemente esperan hasta que reciben un evento. Una vez recibido lo procesa y lo devuelve. La ventaja de este diseño es que la lógica de aplicación que procesa los eventos está claramente separada de la lógica de la interfaz de usuario que genera esos eventos. Un elemento de interfaz de usuario es capaz de"delegar"el procesamiento de un evento a una parte separada de código.

En el modelo de delegación de eventos, los listeners atienden a una fuente de la cual reciben la notificación de un evento. Esto da una ventaja importante: las notificaciones se envían sólo a los listeners que quieren recibirlas. Ésta es una forma más eficiente de gestionar eventos que el utilizado por el antiguo Java 1.0. Anteriormente, se propagaba un evento jerárquicamente hasta llegar a un componente que lo gestionaba. Esto requería componentes para recibir eventos que no iban a procesarse, lo cual significaba una pérdida de tiempo valioso. Con el modelo de delegación de eventos se evitan estos costos.

NOTA *Java también permite procesar eventos sin tener que utilizar el modelo de delegación de eventos. Se puede hacer extendiendo un componente AWT. Esta técnica se verá al final del Capítulo 24. Sin embargo, por las razones citadas anteriormente, el diseño preferido es el modelo de delegación de eventos.*

Las siguientes secciones definen los eventos y describen los roles de las fuentes y listeners.

Eventos

En el modelo de delegación, un *evento* es un objeto que describe un cambio de estado en una fuente. Se puede generar como consecuencia de que una persona interactúe con los elementos en una interfaz gráfica de usuario. Algunas de las actividades que causan la generación de eventos son presionar un botón, meter un carácter mediante el teclado, seleccionar un ítem de una lista, y hacer clic con el ratón, entre otros muchos.

Puede ocurrir que no se provoque un evento directamente por la interacción con una interfaz de usuario. Por ejemplo, se puede generar un evento cuando se termina un cronómetro, cuando un contador pasa de un cierto valor, cuando hay un fallo de software o hardware, o cuando se completa una operación. El programador es libre de definir los eventos que uno considere mejores para su aplicación.

Fuentes de eventos

Una *fuente* es un objeto que genera un evento. Esto ocurre cuando cambia de alguna manera el estado interno de ese objeto. Las fuentes pueden generar más de un tipo de eventos.

Una fuente tiene que registrar los listeners para que estos reciban notificaciones sobre un tipo específico de evento. Cada tipo de evento tiene su propio método de registro. La forma general es:

public void add*Type*Listener(*Type*Listener *el*)

Donde *Type* es el nombre del evento y *el* es una referencia al listener. Por ejemplo, el método que registra un evento de teclado a un listener es **addKeyListener( )**. El método que registra a un

listener de movimiento de ratón es **addMouseMotionListener( )**. Cuando ocurre un evento, se notifica a todos lo listeners registrados, y reciben una copia del objeto evento. Esto es lo que se conoce como *multidifusión* del evento. En todos los casos, las notificaciones se envían sólo a los listeners que quieren recibirlos.

Algunas fuentes sólo permiten registrar a un listener. La forma general del método de registro es:

```
public void add Type Listener(Type Listener el)
  throws java.util.TooManyListenersException
```

Donde *Type* es el nombre del evento y *el* es una referencia al listener. Cuando se produce tal evento, se notifica al listener que está registrado. Esto es conocido como *difusión única* del evento.

Una fuente también debe proporcionar un método que permita a un listener eliminar un registro en un tipo específico de evento. La forma general es:

```
public void remove Type Listener(Type Listener el)
```

Aquí, *Type* es el nombre del evento y *el* es una referencia al listener. Por ejemplo, para borrar un listener de teclado, se llamaría a **removeKeyListener( ).**

La fuente que genera eventos es la que proporciona los métodos para añadir o quitar listeners. Por ejemplo, la clase **Component** proporciona métodos para añadir o quitar listeners de eventos de teclado o ratón.

Auditores de eventos

Nota del traductor. Los auditores de eventos son mejor conocidos por su nombre en inglés **listener**. En el libro se mantendrá el uso de este término en inglés.

Un **listener** es un objeto que es notificado cuando ocurre un evento. Tiene dos requisitos principales. Primero, tiene que ser registrado con una o más fuentes para recibir notificaciones sobre eventos de tipos específicos. Segundo, tiene que implementar métodos para recibir y procesar esas notificaciones.

Los métodos que reciben y procesan eventos se definen en un conjunto de interfaces que están definidas en el paquete **java.awt.event.** Por ejemplo, la interfaz **MouseMotionListener** define dos métodos para recibir notificaciones cuando se arrastra o mueve el ratón. Cualquier objeto puede recibir y procesar uno de estos eventos, o ambos, si implementa esa interfaz. En este y otros capítulos se verán muchas más interfaces para **listeners**.

Clases de eventos

Las clases que representan eventos son el núcleo del mecanismo de gestión de eventos de Java. Así que una discusión sobre la gestión de eventos debe comenzar con la clases de eventos. Es importante comprender, sin embargo, que Java define varios tipos de eventos y que no todas las clases de eventos pueden ser discutidas en este capítulo. Los eventos más ampliamente utilizados son los definidos por ATW y los definidos por Swing. Este capítulo se enfoca en los eventos de AWT. La mayoría de estos eventos también aplican para Swing. Algunos eventos específicos de Swing se describen en el Capítulo 29.

Como raíz en la jerarquía de clases de eventos Java está la clase **EventObject**, definida en el paquete **java.util**. Ésta es la superclase para todos los eventos. Su constructor es el siguiente:

```
EventObject(Object src)
```

Donde *src* es el objeto que genera ese evento.

EventObject contiene dos métodos: **getSource( )** y **toString( )**. El método **getSource( )** devuelve la fuente del evento. Su forma general es la siguiente:

```
Object getSource( )
```

Como se esperaba, **toString( )** devuelve la cadena equivalente al evento.

La clase **AWTEvent**, definida dentro del paquete **java.awt**, es una subclase de **EventObject**. Esta es la superclase (directa o indirectamente) de todos los eventos basados en AWT utilizados por el modelo de delegación de eventos. Se puede utilizar el método **getID( )** para determinar el tipo del evento. La representación de este método se muestra aquí:

```
int getID( )
```

Al final del Capítulo 24 se verán más detalles sobre **AWTEvent**. Ahora sólo es importante conocer que las demás clases que se ven en este apartado son subclases de **AWTEvent**.

Como resumen:

- **EventObject** es la superclase de todos los eventos.
- **AWTEvent** es la superclase de todos los eventos AWT que se gestionan por medio del modelo de delegación de eventos.

El paquete **java.awt.event** define muchos tipos de eventos que se generan mediante elementos de interfaz de usuario. La Tabla 22-1 enumera las clases de eventos más importantes y describe brevemente cuándo se generan. Los constructores y métodos que se utilizan normalmente en cada clase se describen en apartados posteriores.

La clase ActionEvent

Un evento **ActionEvent** se genera cuando se presiona un botón, se hace doble clic en un elemento de una lista, o se selecciona un elemento de un menú. La clase **ActionEvent** define cuatro constantes enteras que se pueden utilizar para identificar cualquier modificador asociado con este tipo de evento: **ALT_MASK**, **CTRL_MASK**, **META_MASK** y **SHIFT_MASK**. Además existe una constante entera, **ACTION_PERFORMED**, que se puede utilizar para identificar eventos de acción.

Clase	Descripción
ActionEvent	Se genera cuando se presiona un botón, se hace doble clic en un elemento de una lista, o se selecciona un elemento de menú
AdjustmentEvent	Se genera cuando se manipula un scrollbar.
ComponentEvent	Se genera cuando un componente se oculta, se mueve, se cambia de tamaño, o se hace visible.
ContainerEvent	Se genera cuando se añade o se elimina un componente de un contenedor.
FocusEvent	Se genera cuando un componente gana o pierde el foco.
InputEvent	Superclase abstracta para cualquier clase de evento de entrada de componente.
ItemEvent	Se genera cuando se hace clic en un checkbox o en un elemento de una lista; también ocurre cuando se hace una selección en elemento de opción o cuando se selecciona o se deselecciona un elemento de un menú de opciones.

KeyEvent	Se genera cuando se recibe una entrada desde el teclado.
MouseEvent	Se genera cuando el ratón se arrastra, se mueve, se hace clic, se presiona, o se libera; también se genera cuando el ratón entra o sale de un componente.
MouseWheelEvent	Se genera cuando se mueve la rueda del ratón.
TextEvent	Se genera cuando se cambia el valor de un área de texto o un campo de texto.
WindowEvent	Se genera cuando una ventana se activa, se cierra, se desactiva, se minimiza, se maximiza, se abre, o se sale de ella.

TABLA 22-1 Principales clases de eventos en **java.awt.event**

La clase **ActionEvent** tiene estos tres constructores:

ActionEvent(Object *src*, int *tipo*, String *cmd*)
ActionEvent(Object *src*, int *tipo*, String *cmd*, int *modificadores*)
ActionEvent(Object *src*, int *tipo*, String *cmd*, long cuando, int *modificadores*)

Donde *src* es una referencia al objeto que ha generado este evento. El *tipo* del evento se especifica con tipo, y la cadena correspondiente a su comando es *cmd*. El argumento modificadores indica qué teclas modificadoras (ALT, CTRL, META y/o SHIFT) se han presionado cuando se ha generado el evento.Y el parámetro *cuando* especifica cuando ocurrió el evento.

Se puede obtener el nombre del comando del objeto **ActionEvent** invocado utilizando el método **getActionCommand( )**, como se muestra a continuación:

String getActionCommand( )

Por ejemplo, cuando se presiona un botón, se genera un evento de acción que tiene un nombre de comando igual a la etiqueta de ese botón.

El método **getModifiers( )** devuelve un valor que indica qué tecla modificadora (ALT, CTRL, META, y/o SHIFT) se ha presionado cuando el evento fue generado. Su forma es la siguiente:

int getModifiers( )

El método **getWhen( )** devuelve el tiempo en el cuál el evento tuvo lugar. Este dato es conocido como *la estampa en el tiempo* del evento. El método **getWhen( )** se muestra aquí:

long getWhen( )

La clase AdjustmentEvent

Se genera un objeto evento de la clase **AdjustmentEvent** como resultado de un cambio sobre una scrollbar. Existen cinco tipos de eventos del tipo **AdjustmentEvent**. La clase **AdjustmentEvent** define constantes enteras que se pueden utilizar para identificar dichos tipos. Las constantes y sus significados son los siguientes:

BLOCK_DECREMENT	El usuario hace clic dentro de la scrollbar para decrementar su valor.
BLOCK_INCREMENT	El usuario hace clic dentro de la scrollbar para incrementar su valor.
TRACK	Se arrastra el botón móvil de la scrollbar.
UNIT_DECREMENT	Se ha hecho clic en el botón que está al final de la scrollbar para decrementar su valor.

UNIT_INCREMENT	Se ha hecho clic en el botón que está al final de la scrollbar para incrementar su valor.

Además, hay una constante entera, **ADJUSTMENT_VALUE_CHANGED**, que indica que ha ocurrido un cambio.

AdjustmentEvent tiene el siguiente constructor:

AdjustmentEvent(Adjustable *src*, int *id*, int *tipo*, int *data*)

Aquí, *src* es una referencia al objeto que ha generado ese evento. El *id* específica el evento. El tipo del ajuste es especificado por *tipo*, y sus datos asociados están en *data*.

El método **getAdjustable( )** devuelve el objeto que ha generado el evento. Su forma es la siguiente:

Adjustable getAdjustable( )

El tipo de evento de ajuste se puede obtener mediante el método **getAdjustmentType( )**, que devuelve una de las constantes definidas por **AdjustmentEvent**. La forma general es la siguiente:

int getAdjustmentType( )

La cantidad del ajuste se puede obtener del método **getValue( )**, que se muestra a continuación:

int getValue( )

Por ejemplo, cuando se manipula una scrollbar, este método devuelve el valor representado por ese cambio.

La clase ComponentEvent

Se genera un objeto evento de la clase **ComponentEvent** cuando cambia el tamaño, posición o visibilidad de un componente. Hay cuatro tipos de eventos generados por un componente. La clase **ComponentEvent** define constantes enteras que se pueden utilizar para identificarlos. Las constantes y sus significados son las siguientes:

COMPONENT_HIDDEN	Se ha ocultado el componente.
COMPONENT_MOVED	Se ha movido el componente.
COMPONENT_RESIZED	Se ha cambiado el tamaño del componente.
COMPONENT_SHOWN	El componente se ha hecho visible.

ComponentEvent tiene este constructor:

ComponentEvent(Component *src*, int *tipo*)

Donde *src* es una referencia al objeto que ha generado el evento. El tipo del evento es especificado por *tipo*.

ComponentEvent es la superclase directa o indirectamente, de **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent** y **WindowEvent**.

El método **getComponent( )** devuelve el componente que ha generado el evento, como se ve a continuación:

Component getComponent( )

La clase ContainerEvent

Se genera un objeto evento de la clase **ContainerEvent** cuando se añade o se borra un componente desde un contenedor. Hay dos tipos de eventos generados por un contenedor. La clase **ContainerEvent** define constantes enteras que se pueden utilizar para identificarlos: **COMPONENT_ADDED** y **COMPONENT_REMOVED**. Éstas indican que se ha añadido o se ha borrado, respectivamente, un componente desde el contenedor.

ContainerEvent es una subclase de **ComponentEvent** y tiene el siguiente constructor:

ContainerEvent(Component *src*, int *tipo*, Component *comp*)

Donde *src* es una referencia al contenedor que genera ese evento. El tipo del evento es especificado por *tipo*, y el componente que se ha añadido o se ha borrado desde el contenedor es referenciado por *comp*.

Utilizando el método **getContainer( )**, se puede obtener una referencia al contenedor que ha generado el evento:

Container getContainer( )

El método **getChild( )** devuelve una referencia al componente que se ha añadido o se ha borrado desde el contenedor. Su forma general es la siguiente:

Component getChild( )

La clase FocusEvent

Se genera un objeto evento de la clase **FocusEvent** cuando un componente está o deja de estar activo. Estos eventos se identifican mediante las constantes enteras **FOCUS_GAINED** y **FOCUS_LOST**.

FocusEvent es una subclase de **ComponentEvent** y tiene estos constructores:

FocusEvent(Component *src*, int *tipo*)
FocusEvent(Component *src*, int *tipo*, boolean *temporalBandera*)
FocusEvent(Component *src*, int *type*, boolean *temporalBandera*, Component *otro*)

Donde *src* es una referencia al componente que ha generado este evento. El tipo del evento es especificado por *tipo*. El argumento *temporalBandera* es puesto en **verdadero** si el evento foco es temporal. En cualquier otro caso, se pone a **falso**. Un evento foco temporal se produce como resultado de otra operación de interfaz de usuario. Por ejemplo, supongamos que lo que está activo —tiene el foco— es un campo de texto. Si el usuario mueve el ratón para modificar una scrollbar, se pierde temporalmente el foco.

El otro componente envuelto en los cambios de foco, llamado el *componente opuesto*, se pasa en el argumento *otro*. Por lo tanto, si un evento **FOCUS_GAINED** ocurre, *otro* hará referencia al componente que perdió el foco. Por el contrario, si un evento **FOCUS_LOST** ocurre, *otro* hará una referencia al componente que obtuvo el foco.

Se puede determinar el componente otro llamando al método **getOppositeComponent( )**, que se muestra aquí:

Component getOppositeComponent( )

Devuelve el componente opuesto.

El método **isTemporary( )** indica si este cambio de foco es temporal. Su forma es la siguiente:

boolean isTemporary( )

El método devuelve **verdadero** si el cambio es temporal. Si no, devuelve **falso**.

La clase InputEvent

La clase abstracta **InputEvent** es una subclase de **ComponentEvent** y es la superclase para los eventos de entrada de componentes. Sus subclases son **KeyEvent** y **MouseEvent**.

La clase **InputEvent** define varias constantes enteras que representan a sus modificadores, por ejemplo presionar la tecla CTRL. Originalmente la clase **InputEvent** definía los siguientes ocho valores para definir a sus modificadores:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTONI_MASK	CTRL_MASK	

Sin embargo, a causa de posibles conflictos entre los modificadores utilizados por eventos del teclado y ratón con otros eventos, los siguientes valores de modificadores extendidos fueron agregados:

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CRTL_DOWN_MASK	

Cuando se escribe un código nuevo, se recomienda que se utilicen los nuevos modificadores extendidos en lugar de los modificadores originales.

Para verificar si un modificador fue presionado en el momento en que un evento se genera, se utilizan los métodos **isAltDown( )**, **isAltGraphDown( )**, **isControlDown( )**, **isMetaDown( )** y **isShiftDown( )**. Las formas de estos métodos son las siguientes:

```
boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
```

El método **getModifiers( )** devuelve un valor que contiene todas las etiquetas de los modificadores para ese evento:

```
int getModifiers( )
```

Se pueden obtener los modificadores extendidos llamando **getModifiersEx( )**, el cual se muestra a continuación:

```
int getModifiersEx( )
```

La clase ItemEvent

Se genera un objeto evento de la clase **ItemEvent** cuando se hace clic en un checkbox o en un elemento de una lista o cuando se selecciona o se deselecciona un elemento de un menú de opciones. Los checkboxes y listboxes se describen más adelante. Hay dos tipos de eventos de elemento, que se identifican por las siguientes constantes enteras:

DESELECTED	El usuario deselecciona un elemento.
SELECTED	El usuario selecciona un elemento.

Además, **ItemEvent** define una constante entera, **ITEM _STATE_CHANGED**, que significa un cambio de estado.

ItemEvent tiene el siguiente constructor:

ItemEvent(ItemSelectable *src*, int *tipo*, Object *entrada*, int *estado*)

Donde *src* es una referencia al componente que ha generado ese evento. Un ejemplo podría ser una lista o un elemento de elección. El tipo de elemento es especificado por *tipo*. Lo que específicamente genera el evento de elemento se pasa con *entrada*. El estado actual del elemento es *estado*.

El método **getItem( )** se puede utilizar para obtener una referencia al elemento que ha generado un evento. Su forma es la siguiente:

Object getItem( )

El método **getItemSelectable( )** se puede utilizar para obtener una referencia al objeto **ItemSelectable** que ha generado el evento. Su forma general es la siguiente:

ItemSelectable getItemSelectable( )

Las listas y las listas desplegables son ejemplos de elementos de interfaz de usuario que implementan la interfaz **ItemSelectable**.

El método **getStateChange( )** devuelve el cambio de estado (por ejemplo, **SELECTED** o **DESELECTED**) para el evento. Su forma general es la siguiente:

int getStateChange( )

La clase KeyEvent

Se genera un objeto evento de la clase **KeyEvent** cuando se pulsa una tecla. Hay tres clases de eventos de teclado, que están definidos por las siguientes constantes enteras: **KEY_PRESSED**, **KEY_RELEASED**, y **KEY_TYPED**. Los primeros dos eventos se generan cuando se presiona o se libera cualquier tecla. El último evento sólo se da cuando se genera un carácter. Recordemos que no siempre se generan caracteres al presionar teclas. Por ejemplo, si se presiona la tecla SHIFT no se genera carácter alguno.

KeyEvent define otras muchas constantes enteras. Por ejemplo, **VK_0** a **VK_9** y **VK_A** a **VK_Z** definen los equivalentes ASCII de números y letras. Algunas otras constantes son:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_ PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

Las constantes **VK** especifican *códigos de teclas virtuales* y son independientes de cualquier modificador, como control, shift o alt.

KeyEvent es una subclase de **InputEvent** y éste es uno de sus constructores:

KeyEvent(Component *src*, int *tipo*, long *cuando*, int *modificadores*, int *codigo*, char *ch*)

Donde *src* es una referencia al componente que genera ese evento. El tipo del evento es especificado por *tipo*. El tiempo en el que se ha presionado la tecla se pasa con *cuando*. El argumento *modificadores* indica qué modificador se ha presionado cuando ha ocurrido ese evento de teclado. Los códigos de tecla virtual, como **VK_UP**, **VK_A**, y demás, se pasan en el argumento llamado *codigo*. El carácter equivalente (si existe alguno) se pasa en *ch*. Si no existe

ningún carácter válido, entonces *ch* contiene **CHAR_UNDEFINED**. Para los eventos **KEY_TYPED**, el argumento *codigo* contendrá a **VK_UNDEFINED**.

La clase **KeyEvent** define varios métodos, pero los que más se usan son **getKeyChar( )**, que devuelve el carácter que se ha tecleado y **getKeyCode( )**, que devuelve el código de la tecla. Sus formas generales son las siguientes:

```
char getKeyChar( )
int getKeyCode( )
```

Si ningún carácter válido está disponible, entonces **getKeyChar( )** devuelve **CHAR_UNDEFINED**. Cuando se produce un evento **KEY_TYPED**, **getKeyCode( )** devuelve **VK_UNDEFINED**.

La clase MouseEvent

Existen ocho tipos de eventos de ratón. La clase **MouseEvent** define las siguientes constantes enteras, que se pueden utilizar para identificarlos:

MOUSE_CLICKED	El usuario hace clic con el ratón.
MOUSE_DRAGGED	El usuario arrastra el ratón.
MOUSE_ENTERED	El ratón entra a un componente.
MOUSE_EXITED	El ratón sale de un componente.
MOUSE_MOVED	Se mueve el ratón.
MOUSE_PRESSED	Se presiona el ratón.
MOUSE_RELEASED	Se libera el ratón.
MOUSE_WHEEL	La rueda del ratón fue movida

MouseEvent es una subclase de **InputEvent** y tiene este constructor:

```
MouseEvent(Component src, int tipo, long cuando, int modificadores,
int x, int y, int clics, boolean t)
```

Donde *src* es una referencia al componente que ha generado el evento. El tipo del evento es especificado por *tipo*. El tiempo al momento en el que ha ocurrido el evento de ratón se pasa en el argumento llamado *cuando*. El argumento *modificadores* indica qué modificador se ha presionado cuando ha ocurrido el evento. Las coordenadas del ratón se pasan con *x y*. El número de clics se pasa en *clics*. La etiqueta *t* indica si ese evento hace que aparezca un menú en esa plataforma.

Dos métodos comúnmente utilizados de esta clase son **getX( )** y **getY( )**. Estos métodos devuelven las coordenadas X,Y del ratón dentro de un componente cuando ha ocurrido el evento. Sus formas son las siguientes:

```
int getX( )
int getY( )
```

También se puede utilizar el método **getPoint( )** para obtener las coordenadas del ratón, como se muestra a continuación:

```
Point getPoint( )
```

Este método devuelve un objeto **Point** que contiene las coordenadas X,Y en sus miembros enteros: **x**, **y**. El método **translatePoint( )** traduce la posición del evento. Su forma es la siguiente:

void translatePoint(int *x*, int *y*)

Aquí, los argumentos *x*, *y* se añaden a las coordenadas del evento.

El método **getClickCount( )** proporciona el número de clics que se han hecho con el ratón para este evento. Su forma es la siguiente:

int getClickCount( )

El método **isPopupTrigger( )** prueba si el evento ha hecho aparecer un menú en la plataforma. Su forma es la siguiente:

boolean isPopupTrigger( )

También está disponible el método **getButton( )**, que se muestra a continuación

int getButton( )

Dicho método devuelve el valor que representa el botón que causa el evento. El valor devuelto será una de las constantes definidas por **MouseEvent**

NOBUTTON	BUTTON1	BUTTON2	BUTTON3

El valor de **NOBUTTON** indica que no hay botón presente o liberado.

Java SE 6 agrega tres métodos a **MouseEvent** que obtienen las coordenadas del ratón relativas a la pantalla en lugar de al componente. Estos métodos se muestran aquí:

Point getLocationOnScreen( )

int getXOnScreen( )

int getYOnScreen( )

El método **getLocationOnScreen( )** devuelve un objeto **Point** que contiene las dos coordenadas X y Y. Los otros dos métodos devuelven la coordenada correspondiente.

La clase MouseWheelEvent

La clase **MouseWheelEvent** encapsula un evento de la rueda del ratón. Es una subclase de **MouseEvent**. No todos los ratones tienen rueda. Si el ratón tiene rueda, ésta se encuentra entre los botones derecho e izquierdo. La rueda se utiliza para desplazarse. **MouseWheelEvent** define estas dos constantes enteras:

WHEEL_BLOCK_SCROLL	Un evento de página arriba o página abajo ocurrió
WHEEL_UNIT_SCROLL	Un evento de línea arriba o línea abajo ocurrió

A continuación uno de los constructores definidos por **MouseWheelEvent**:

MouseWheelEvent (Component *src*, int *tipo*, long *cuando*, int *modificadores*,
int *x*, int *y*, int *clics*, boolean *t*,
int *sc*, int *monto*, int *cuenta*)

Donde, *src* es una referencia al objeto que genera el evento. El tipo de evento se especifica en *tipo*. La hora en que el evento del ratón ocurrió se pasa en *cuando*. El argumento modificadores indica cuales modificadores fueron activados cuando el evento ocurrió. Las coordenadas del ratón se pasan a través de *x*, *y*. El número de clics que la rueda ha rotado se pasa mediante *clics*.

El argumento *t* es una bandera que indica si este evento causa que un menú popup aparezca en esta plataforma. El valor *sc* debe ser **WHEEL_UNIT_SCROLL** o **WHEEL_BLOCK_SCROLL**. El número de unidades de scroll es pasado en *monto*. El parámetro *cuenta* indica el número de rotaciones en que la rueda se movió.

MouseWheelEvent define métodos que permiten acceder a los eventos de la rueda. Para obtener el número de unidades rotacionales, se llama a **getWheelRotation( )**, como se muestra aquí:

```
int getWheelRotation( )
```

Este método devuelve el número de unidades rotacionales. Si el valor es positivo, la rueda se movió en contra de las manecillas del reloj. Si el valor es negativo, la rueda se movió en el sentido de las manecillas del reloj.

Para obtener el tipo de scroll, se llama a **getScrollType( )**, como se muestra aquí:

```
int getScrollType( )
```

que devuelve **WHEEL_UNIT_SCROLL** o **WHEEL_BLOCK_SCROLL**.

Si el tipo de scroll es **WHEEL_UNIT_SCROLL**, se puede obtener el número de unidades de scroll llamando al método **getScrollAmount( )**. Como se muestra aquí;

```
int getScrollAmount( )
```

La clase TextEvent

Con esta clase se describen eventos de texto. Se generan mediante campos de texto y áreas de texto cuando el usuario o el programa introducen caracteres. **TextEvent** define la constante entera **TEXT_VALUE_CHANGED**.

El único constructor para esta clase es el siguiente:

```
TextEvent(Object src, int tipo)
```

Donde *src* es una referencia al objeto que ha generado el evento. El tipo de evento es especificado por *tipo*.

El objeto **TextEvent** no incluye los caracteres que ya están en el componente de texto que ha generado el evento, sino que es el programa el que debe de utilizar otros métodos asociados con el componente de texto para recuperar la información. Esta operación es diferente a la de otros objetos de evento que se ven en esta sección. Por esta razón, no se estudia aquí ningún método para la clase **TextEvent**. Hay que pensar en una notificación de evento de texto como una señal a un listener para que este recupere información desde un componente de texto específico.

La clase WindowEvent

Hay diez tipos de eventos de ventana. La clase **WindowEvent** define constantes enteras que se pueden utilizar para identificarlos. Las constantes y sus significados son los siguientes:

WINDOW _ACTIVATED	Se ha activado la ventana.
WINDOW _CLOSED	Se ha cerrado la ventana.
WINDOW _CLOSING	El usuario ha pedido que se cierre la ventana.
WINDOW _DEACTIVATED	La ventana ha dejado de estar activa.
WINDOW_DEICONIFIED	Se ha mostrado la ventana tras pulsar su incono.
WINDOW_GAINED_FOCUS	La ventana ahora está en foco de entrada

WINDOW_ICONIFIED	Se ha minimizado la ventana a un icono.
WINDOW_LOST_FOCUS	La ventana ha perdido el foco de entrada.
WINDOW _OPENED	La ventana ha sido abierta.
WINDOW_STATE_CHANGED	El estado de la ventana ha cambiado.

WindowEvent es una subclase de **ComponentEvent** y define varios constructores. El primero se muestra aquí:

WindowEvent(Window *src*, int *tipo*)

Donde *src* es una referencia al objeto que ha generado el evento. El tipo del evento es *tipo*.

Los tres constructores siguientes ofrecen un control más detallado:

WindowsEvent(Windows *src*, int *tipo*, Window *otro*)
WindowsEvent(Windows *src*, int *tipo*, int *deEstado*, in *aEstado*)
WindowsEvent(Windows *src*, int *tipo*, Window *otro*, int *deEstado*, int *aEstado*)

Donde, *otro* especifica la ventana opuesta cuando un evento de foco o activación ocurre. El parámetro *deEstado* especifica el estado anterior de la ventana, y *aEstado* especifica el nuevo estado que la ventana tendrá cuando un cambio de estado de ventana ocurre.

El método más comúnmente utilizado de esta clase es **getWindow( )**, que devuelve el objeto **Window** que ha generado el evento. Su forma general es la siguiente:

Window getWindow( )

WindowEvent también define métodos que devuelven la ventana contraria (cuando un evento de foco o activación ha ocurrido), el estado anterior de la ventana, y el estado actual de la ventana. Estos métodos se muestran aquí:

Window getOppositeWindow( )
int getOldState( )
int getNewState( )

Fuentes de eventos

La Tabla 22-2 lista algunos de los componentes de interfaz de usuario que pueden generar los eventos descritos en el apartado anterior. Además de esos elementos de interfaz gráfica de usuario, cualquier clase derivada de **Component**, como **Applet**, puede generar eventos.

Fuente del evento	Descripción
Button	Genera eventos de tipo ActionEvent cuando se presiona el botón.
Checkbox	Genera eventos de tipo ItemEvent cuando se selecciona o se deselecciona un checkbox.
Choice	Genera eventos de tipo ItemEvent cuando se cambia una opción.
List	Genera eventos de tipo ActionEvent cuando se hace doble clic sobre un elemento; genera eventos de tipo ItemEvent cuando se selecciona o se deselecciona un elemento.

TABLA 22-2 Ejemplos de componentes que pueden generar eventos

Fuente del evento	Descripción
MenuItem	Genera eventos de tipo ActionEvent cuando se selecciona un elemento de menú; genera eventos de tipo ItemEvent cuando se selecciona o se deselecciona un elemento de un menú de opciones.
Scrollbar	Genera eventos de tipo AdjustmentEvent cuando se manipula el scrollbar.
Componentes de tipo Text	Genera eventos de tipo TextEvent cuando el usuario introduce un carácter.
Window	Genera eventos de tipo WindowEvent cuando una ventana se activa, se cierra, se desactiva, se minimiza, se maximiza, se abre o se sale de ella.

TABLA 22-2 Ejemplos de componentes que pueden generar eventos (*Continuación*)

Por ejemplo, se pueden recibir eventos del ratón y del teclado desde un applet. También el programador puede construir sus propios componentes que generen eventos. En este capítulo sólo se van a gestionar eventos de ratón y de teclado, pero en los dos siguientes capítulos se verá cómo gestionar los eventos que aparecen en la Tabla 22-2.

Las interfaces de auditores de eventos

Como se explicó anteriormente, el modelo de delegación de eventos tiene dos partes: fuentes y listeners. Los listeners se crean implementando una o más interfaces definidas en el paquete **java.awt.event**. Cuando se produce un evento, la fuente del evento invoca al método apropiado definido por el listener y proporciona un objeto evento como su argumento. En la Tabla 22-3 aparecen las interfaces de listener que más se utilizan, y también aporta una breve descripción de los métodos que definen esas interfaces. En las siguientes secciones se examinan los métodos específicos que hay en cada interfaz.

La interfaz ActionListener

Esta interfaz define el método **actionPerformed( )** que se invoca cuando un evento de acción ocurre. Su forma general se muestra a continuación:

```
void actionPerformed(ActionEvent ae)
```

Interfaz	Descripción
ActionListener	Define un método para recibir eventos de acción.
AdjustmentListener	Define un método para recibir eventos de ajuste.
ComponentListener	Define cuatro métodos para reconocer cuándo se oculta, se mueve, se cambia de tamaño o se muestra un componente.
ContainerListener	Define dos métodos para reconocer cuándo se añade o se elimina un componente de un contenedor.
FocusListener	Define dos métodos para reconocer cuándo un componente gana o pierde el foco del teclado.
ItemListener	Define un método para reconocer cuándo cambia el estado de un elemento.

KeyListener	Define tres métodos para reconocer cuándo se presiona, se libera o se golpea una tecla.
MouseListener	Define cinco métodos para reconocer cuándo se presiona o se libera un botón del ratón, se hace clic con él, o el ratón entra en un componente o sale de él.
MouseMotionListener	Define dos métodos para reconocer cuándo se arrastra o se mueve el ratón.
MouseWheelListener	Define un método para reconocer cuando la rueda del ratón se mueve.
TextListener	Define un método para reconocer cuándo cambia un valor de texto.
WindowFocusListener	Define dos métodos para reconocer cuando una ventana gana o pierde foco de entrada.
WindowListener	Define siete métodos para reconocer cuándo una ventana se activa, se cierra, se desactiva, se minimiza, se maximiza, se abre o se sale de ella.

TABLA 22-3 Interfaces de listener que más se utilizan

La interfaz AdjustmentListener

Esta interfaz define el método **adjustmentValueChanged( )** que se invoca cuando se produce un evento de ajuste. Su forma general es la siguiente:

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

La interfaz ComponentListener

Esta interfaz define cuatro métodos que se invocan cuando a un componente se le cambia el tamaño, se mueve, se muestra o se oculta. Sus formas generales son las siguientes:

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

La interfaz ContainerListener

Esta interfaz tiene dos métodos. Cuando se añade un componente a un contenedor, se invoca a **componentAdded( )**. Cuando se borra un componente de un contenedor, se invoca a **componentRemoved( )**. Sus formas generales son las siguientes:

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

La interfaz FocusListener

Esta interfaz define dos métodos. Cuando un componente obtiene el foco del teclado, se invoca **focusGained( )**. Cuando un componente pierde el foco del teclado, se llama a **focusLost( )**. Sus formas generales son las siguientes:

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

La interfaz ItemListener

Esta interfaz define el método **itemStateChanged( )** que se invoca cuando cambia el estado de un elemento. Su forma general es la siguiente:

void itemStateChanged(ItemEvent *ie*)

La interfaz KeyListener

Esta interfaz define tres métodos. Los métodos **keyPressed( )** y **keyReleased( )** se invocan cuando se presiona y se libera una tecla, respectivamente. El método **keyTyped( )** se invoca cuando se ha introducido un carácter.

Por ejemplo, si un usuario presiona y libera la tecla A, se generan tres eventos en secuencia: tecla presionada, carácter introducido, tecla liberada. Si un usuario presiona y libera la tecla INICIO, se generan dos eventos en este orden: tecla presionada, tecla liberada.

Las formas generales de estos eventos son las siguientes:

void keyPressed(KeyEvent *ke*)
void keyReleased(KeyEvent *ke*)
void keyTyped(KeyEvent *ke*)

La interfaz MouseListener

Esta interfaz define cinco métodos. Si se presiona y se libera el ratón en el mismo punto, se invoca al método **mouseClicked( )**. Cuando el ratón entra en un componente, se llama al método **mouseEntered( )**. Cuando el ratón sale del componente, se llama a **mouseExited( )**. Los métodos **mousePressed( )** y **mouseReleased( )** se invocan cuando se presiona y se libera el ratón, respectivamente. Las formas generales de estos métodos son las siguientes:

void mouseClicked(MouseEvent *me*)
void mouseEntered(MouseEvent *me*)
void mouseExited(MouseEvent *me*)
void mousePressed(MouseEvent *me*)
void mouseReleased(MouseEvent *me*)

La interfaz MouseMotionListener

Esta interfaz define dos métodos. Al método **mouseDragged( )** se le llama tantas veces como se arrastre al ratón. Al método **mouseMoved( )** se le llama tantas veces como se mueva el ratón. Sus formas generales son las siguientes:

void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)

La interfaz MouseWheelListener

Esta interfaz define el método **mouseWheelMoved( )** que se invoca cuando la rueda del ratón se mueve. Su forma general es:

void mouseWheelMoved(MouseWheelEvent mwe)

La interfaz TextListener

Esta interfaz define el método **textChanged( )** que se invoca cuando hay un cambio en una área de texto o en un campo de texto. Su forma general es la siguiente:

void textChanged(TextEvent *te*)

La interfaz WindowFocusListener

Esta interfaz define dos métodos: **windowGainedFocus( )** y **windowLostFocus( )**. Estos métodos son llamados cuando una ventana gana o pierde el foco de entrada. Sus formas generales se muestran a continuación:

void windowGainedFocus(WindowEvent *we*)
void windowLostFocus(WindowEvent *we*)

La interfaz WindowListener

Esta interfaz define siete métodos. Los métodos **windowActivated( )** y **windowDeactivated( )** se invocan cuando se activa o desactiva una ventana, respectivamente. Si una ventana se minimiza a un icono, se llama al método **windowIconified( )**. Cuando una ventana es mostrada pulsando en su icono, se llama al método **windowDeiconified( )**. Cuando se abre o se cierra una ventana, se llama a los métodos **windowOpened( )** o **windowClosed( )**, respectivamente. Al método **windowClosing( )** se le llama cuando se está cerrando una ventana. Las formas generales de estos métodos son:

void windowActivated(WindowEvent *we*)
void windowClosed(WindowEvent w*e*)
void windowClosing(WindowEvent *we*)
void windowDeactivated(WindowEvent *we*)
void windowDeiconified(WindowEvent *we*)
void windowIconified(WindowEvent *we*)
void windowOpened(WindowEvent *we*)

Uso del modelo de delegación de eventos

Una vez que se ha visto la teoría sobre la que trabaja el método de delegación de eventos y se tiene una idea general de sus componentes, se puede comenzar a trabajar con él. Utilizar el método de delegación de eventos es bastante fácil. Sólo hay que seguir estos dos pasos:

1. Implementar la interfaz apropiada en el listener, de tal manera que reciba el tipo de evento deseado.
2. Implementar el código para registrar y eliminar (si fuese necesario) el registro del listener como destinatario de las notificaciones de eventos.

Hay que recordar que una fuente puede generar muchos tipos de eventos. Cada evento se tiene que declarar de forma separada. Asimismo, se puede registrar un objeto para que reciba varios tipos de eventos, pero se deben implementar todas las interfaces que hagan falta para recibir esos eventos.

Para ver cómo trabaja el método de delegación de eventos, vamos a ver algunos ejemplos donde se gestionan los dos generadores de eventos que más se utilizan: el ratón y el teclado.

La gestión de eventos de ratón

Para gestionar eventos de ratón, se deben implementar las interfaces **MouseListener** y **MouseMotionListener** (posiblemente se desee también implementar **MouseWheelListener**, pero no se hará aquí). El siguiente applet muestra el proceso. Se van a visualizar las coordenadas

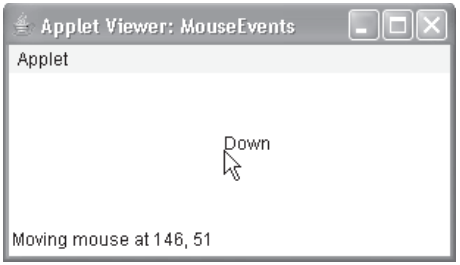
actuales del ratón en la ventana de estado del applet. Cada vez que se presione un botón, se verá la palabra "Abajo" en la posición que apunta el ratón. Y cuando se libere el botón, aparecerá la palabra "Arriba". Si se hace clic en un botón, se verá el mensaje "Clic del ratón" en la esquina superior izquierda del panel del applet.

Cuando el ratón entre o salga de la ventana del applet, se verá un mensaje en la esquina superior izquierda del panel del applet. Cuando se arrastre el ratón, se mostrará un *, que se irá dejando pista de por dónde se ha ido arrastrando el ratón. Hay que tener en cuenta que las dos variables, **mouseX** y **mouseY**, guardan la posición del ratón cuando se da un evento presionar, liberar o arrastrar el ratón. Estas coordenadas las utiliza **paint( )** para pintar la salida en el punto en que se han dado esos eventos.

```
// Ejemplo de la gestión de eventos de ratón.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code = "MouseEvent" width=300 height=100>
</applet>
*/

public c1ass MouseEvents extends Applet
  implements MouseListener, MouseMotionListener {

  String msg ="";
  int mouseX = 0, mouseY = 0; // coordenadas del ratón

  public void init() {
    addMouseListener(this);
    addMouseMotionListener(this);
  }

  // Gestiona los clics del ratón.
  public void mouseClicked(MouseEvent me) {
    // Guarda coordenadas
    mouseX = 0;
    mouseY = 10;
    msg = "Clic del ratón";
    repaint();
  }

  // Gestiona entradas del ratón
  public void mouseEntered(MouseEvent me) {
    // Guarda coordenadas
    mouseX = 0;
    mouseY = 10;
    msg = "El ratón ha entrado.";
    repaint();
  }

  // Gestiona salidas del ratón
  public void mouseExited(MouseEvent me) {
    // Guarda las coordenadas
    mouseX = 0;
    mouseY = 10;
    msg = "El ratón ha salido.";
    repaint() ;
  }
```

```
  // Gestiona presionar los botones.
  public void mousePressed(MouseEvent me) {
    // Guarda las coordenadas
    mouseX = me.getX();
    mouseY = me.getY() ;
    msg = "Abajo";
    repaint();
  }

  // Gestiona la liberación de botones.
  public void mouseReleased(MouseEvent me) {
    // Guarda las coordenadas
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Arriba";
    repaint();
  }

  // Gestiona el arrastre del ratón
  public void mouseDragged(MouseEvent me) {
    // Guarda las coordenadas
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "*";
    showStatus("Arrastrando el ratón en " + mouseX + ", " + mouseY);
    repaint();
  }

  // Gestiona los movimientos del ratón.
  public void mouseMoved(MouseEvent me) {
    // Muestra el estatus
    showStatus("Moviendo el ratón en " + me.getX () + ", " + me.getY());
  }

  // Muestra msg en la ventana del applet en la actual posición
  public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
  }
}
```

Una muestra de la salida del programa es la siguiente:



Vamos a ver más detalladamente este ejemplo. La clase **MouseEvents** extiende a **Applet** e implementa a las interfaces **MouseListener** y **MouseMotionListener**. Estas dos interfaces tienen métodos que reciben y procesan los distintos tipos de eventos de ratón. Hay que tener en cuenta que el applet es tanto la fuente como el listener para estos eventos. Esto funciona porque **Component**, que suministra los métodos **addMouseListener( )** y **addMouseMotionListener( )**, es una

superclase de **Applet**. Ser tanto la fuente como el listener para los eventos es algo normal en los applets.

Dentro del método **init( )**, el applet se declara a sí mismo como listener para eventos de ratón. Esto se hace utilizando **addMouseListener( )** y **addMouseMotionListener( )**, que, como ya se ha dicho, son miembros de **Component**. Se muestran a continuación:

void addMouseListener(MouseListener *ml*)
void addMouseMotionListener(MouseMotionListener *mml*)

Donde *ml* es una referencia al objeto que recibe los eventos de ratón, y *mml* es una referencia al objeto que recibe los eventos de movimiento del ratón. En este programa se utiliza el mismo objeto para las dos cosas.

De esta forma, el applet implementa todos los métodos definidos mediante las interfaces **MouseListener** y **MouseMotionListener**. Éstas son las que gestionan los diferentes eventos de ratón. Cada método gestiona su evento y luego devuelve el control.

La gestión de eventos de teclado

Para gestionar eventos de teclado se utiliza la misma arquitectura que se acaba de ver para los eventos de ratón en la sección anterior. La diferencia es, obviamente, que se implementará la interfaz **KeyListener**.

Antes de ver un ejemplo, repasemos cómo se generan los eventos de teclado. Cuando se presiona una tecla, se genera un evento **KEY_PRESSED**, y se llama al método manejador de eventos **keyPressed( )**. Cuando se libera una tecla, se genera un evento **KEY_RELEASED** y se ejecuta el manejador **keyReleased( )**. Si se genera un carácter, entonces se envía un evento **KEY_TYPED** y se invoca al manejador **keyTyped( )**. Es decir, cada vez que el usuario presiona una tecla, se generan al menos dos eventos, y muchas veces tres. Si al programador sólo le interesan los caracteres reales, es posible olvidar la información que se pasa a través de los eventos de presionar y liberar una tecla. Pero si el programa necesita gestionar teclas especiales, como son las teclas de dirección (flechas) o las teclas de función, entonces hay que echar un vistazo a esa información con el manejador **keyPressed( ).**

En el siguiente programa se puede ver una entrada por teclado. El ejemplo envía las teclas pulsadas a la ventana del applet y muestra el estado "presionada/liberada" de cada tecla en la ventana de estado.

```
// Ejemplo de los manejadores de eventos de teclado.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="SimpleKey" width=300 height=l00>
  </applet>
*/

public class SimpleKey extends Applet
  implements KeyListener {

  String msg = "";
  int X = 10, y = 20; // Salida de las coordenadas

  public void init() {
    addKeyListener(this);
  }
```

```
  public void keyPressed(KeyEvent ke) {
    showStatus("Tecla Abajo");
  }

  public void keyReleased(KeyEvent ke) {
    showStatus("Tecla Arriba");
  }

  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }

  / / Muestra la pulsación de las teclas.
  public void paint(Graphics g) {
    g.drawString(msg, X, Y);
  }
}
```

Un ejemplo de la salida del programa es la siguiente:

Si se quieren gestionar las teclas especiales, como las teclas de dirección (flechas) o las teclas de función, se debe utilizar el manejador **keyPressed( )** para responder a ellas, ya que no están disponibles a través de **keyTyped( )**. Para identificar las teclas, hay que utilizar sus códigos de tecla virtuales. Por ejemplo, con el siguiente applet se muestra el nombre de algunas teclas especiales:

```
// Ejemplo de algunos códigos de tecla virtuales.
import java.awt.*;
import java.awt.event .*;
import java.applet.*;
/*
<applet code="KeyEvents" width=300 height=l00>
</applet>
*/

public class KeyEvents extends Applet
  implements KeyListener {

  String msg = "";
  int X = 10, Y = 20; // coordenadas de la salida

  public void init() {
    addKeyListener(this);
  }

  public void keyPressed(KeyEvent ke) {
  showStatus("Tecla Abajo");
```

```
  int key = ke.getKeyCode();
  switch(key) {
    case KeyEvent.VK_Fl:
      msg += "<Fl>";
      break;
    case KeyEvent.VK_F2:
      msg += "<F2>";
      break;
    case KeyEvent.VK_F3:
      msg += "<F3>";
      break;
    case KeyEvent.VK_PAGE_DOWN:
      msg += "<PgDn>";
      break;
    case KeyEvent.VK_PAGE_UP:
      msg += "<PgUp>" ;
      break;
    case KeyEvent.VK_LEFT:
      msg += "<Flecha Izquierda>" ;
      break;
    case KeyEvent.VK_RIGHT:
      msg += "<Flecha Derecha>" ;
      break;
    }
    repaint();
  }
  public void keyReleased(KeyEvent ke) {
    showStatus("Tecla Arriba");
  }
  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }
  // Muestra la pulsación de teclas.
  public void paint(Graphics g) {
    g.drawString(msg, X, Y};
  }
}
```

Un ejemplo de la salida se muestra a continuación:

Los procedimientos que se han mostrado en los ejemplos anteriores de eventos de ratón y teclado se pueden generalizar a cualquier tipo de gestión de eventos, incluidos aquellos eventos

generados por controles. En capítulos posteriores, se verán muchos ejemplos que gestionan otro tipo de eventos, pero todos ellos seguirán la misma estructura básica que los programas que se acaban de mostrar.

TABLA 22-4
Interfaces de listener, comúnmente utilizadas, implementadas por las clases Adapter

Clase adaptadora	Interfaz para listener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
Window Adapter	WindowListener

Clases adaptadoras

Java proporciona una característica especial, llamada *clase adaptadora*, que en algunas circunstancias puede simplificar la creación de manejadores de eventos. Una clase adaptadora proporciona una implementación vacía de todos los métodos en una interfaz listener de evento. Estas clases adaptadoras son útiles cuando se quiere recibir y procesar sólo alguno de los eventos que está gestionando una interfaz listener de eventos en particular. Se puede definir una nueva clase para actuar como un listener de eventos extendiendo una de las clases adaptadoras e implementando sólo aquellos eventos que interesen.

Por ejemplo, la clase **MouseMotionAdapter** tiene dos métodos, **mouseDragged( )** y **mouseMoved( )** los cuales son métodos definidos por la interfaz **MouseMotionListener**. Si sólo estamos interesados en eventos de arrastre de ratón, se puede simplemente extender **MouseMotionAdapter** y sobrescribir **mouseDragged( )**. La implementación vacía de **mouseMoved( )** gestionaría los eventos de movimiento de ratón por nosotros.

En la Tabla 22-4 se enlistan las diferentes clases adaptadoras comúnmente utilizadas en **java.awt.event** y la interfaz que implementa cada una.

El siguiente ejemplo es una muestra de una clase adaptadora. El ejemplo visualiza un mensaje en la barra de estado de un visor de applets o de un navegador cuando se arrastra o se hace clic con el ratón. Sin embargo, todos los demás eventos del ratón son silenciosamente ignorados. El programa tiene tres clases. **AdapterDemo** extiende a **Applet**. Su método **init( )** crea una instancia de **MyMouseAdapter** y declara al objeto para que pueda recibir notificaciones de eventos de ratón. También crea una instancia de **MyMouseMotionAdapter** y declara al objeto para que pueda recibir notificaciones de eventos de movimiento de ratón. Los dos constructores reciben una referencia al applet como argumento.

MyMouseAdapter extiende de **MouseAdapter** y sobrescribe el método **mouseClicked( )**. Los otros eventos de ratón se ignoran mediante código heredado de la clase **MouseAdapter**. **MyMouseMotionAdapter** extiende de **MouseMotionAdapter** y sobrescribe el método **mouseDragged( )**. Los otros eventos de movimiento de ratón se ignoran mediante código heredado de la clase **MouseMotionAdapter**.

Es importante notar que las dos clases de listener de eventos guardan una referencia al applet. Esta información se proporciona a sus constructores como un argumento y se utiliza posteriormente para invocar al método **showStatus( )**.

```
// Ejemplo de un adaptador.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="AdapterDemo" width=300 height=l00>
  </applet>
*/
public class AdapterDemo extends Applet {
  public void init() {
    addMouseListener(new MyMouseAdapter(this));
    addMouseMotionListener(new MyMouseMotionAdapter(this));
  }
}
class MyMouseAdapter extends MouseAdapter {
  AdapterDemo adapterDemo;
  public MyMouseAdapter(AdapterDemo adapterDemo) {
    this.adapterDemo = adapterDemo;
  }
  // Gestiona los clics del ratón.
  public void mouseClicked(MouseEvent me) {
    adapterDemo.showStatus("El ratón tuvo un clic");
  }
}
class MyMouseMotionAdapter extends MouseMotionAdapter {
  AdapterDemo adapterDemo;
  public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
    this.adapterDemo = adapterDemo;
  }
  // Gestiona el arrastre del ratón.
  public void mouseDragged(MouseEvent me) {
    adapterDemo.showStatus("Ratón arrastrado");
  }
}
```

Como se puede ver en el programa, no se tienen que implementar todos los métodos definidos por las interfaces **MouseMotionListener** y **MouseListener**, lo que ahorra un considerable esfuerzo y previene contra posibles confusiones con los métodos vacíos. Como ejercicio, se podría intentar reescribir uno de los ejemplos de entrada por teclado mostrados anteriormente para hacer que utilice **KeyAdapter**.

Clases internas

En el Capítulo 7 se explicaron los fundamentos básicos de las clases internas. Ahora se verá por qué son importantes. Recuerde que una *clase interna* es una clase definida dentro de otra clase, o incluso dentro de una expresión. Este apartado muestra cómo se pueden utilizar las clases internas para simplificar el código cuando se utilizan clases adaptadoras de eventos.

Para comprender las ventajas de las clases internas, consideremos el applet mostrado a continuación. Este *no* utiliza una clase interna. Su objetivo es visualizar la cadena "Se ha

presionado el ratón" en la barra de estado del visor de applets o del navegador cuando se presione un botón del ratón.

En este programa hay dos clases de alto nivel. **MousePressedDemo** que extiende a **Applet**, y **MyMouseAdapter** que extiende a **MouseAdapter**. El método **init( )** de **MousePressedDemo** instancia a **MyMouseAdapter** y pasa ese objeto como un argumento al método **addMouseListener( )**.

Es importante observar que se pasa una referencia al applet como argumento al constructor **MyMouseAdapter**. Esta referencia se guarda en una variable de instancia para que la utilice más tarde el método **mousePressed( )**. Cuando se presiona el botón del ratón, se invoca al método **showStatus( )** del applet a través de la referencia al applet que se tenía guardada. En otras palabras, se invoca a **showStatus( )** en función de la referencia guardada por **MyMouseAdapter**.


```
// Este applet NO utiliza una clase interna.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="MousePressedDemo" width=200 height=100>
  </applet>
*/

public class MousePressedDemo extends Applet {
  public void init() {
    addMouseListener(new MyMouseAdapter(this));
  }
}

class MyMouseAdapter extends MouseAdapter {
  MousePressedDemo mousepressedDemo;
  public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
    this.mousePressedDemo = mousepressedDemo;
  }
  public void mousePressed(MouseEvent me) {
    mousePressedDemo.showStatus("Se ha presionado el ratón.");
  }
}
```

A continuación se muestra cómo se puede mejorar el programa anterior utilizando una clase interna. Aquí, **InnerClassDemo** es una clase de nivel superior que extiende a **Applet**. **MyMouseAdapter** es una clase interna que extiende a **MouseAdapter**. Puesto que se define a **MyMouseAdapter** dentro del ámbito de **InnerClassDemo**, tiene acceso a todos los métodos y variables dentro del ámbito de esa clase. Por lo tanto, el método **mousePressed( )** puede llamar directamente al método **showStatus( )**. Al hacer esto, ya no se necesita guardar referencia alguna al applet. Es decir, ya no es necesario que **MyMouseAdapter( )** pase una referencia al objeto invocado.

```
// Ejemplo de una clase interna.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="InnerClassDemo" width=200 height=100>
  </applet>
*/
```

```
public class InnerClassDemo extends Applet {
  public void init() {
    addMouseListener(new MyMouseAdapter());
  }
  class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent me) {
      showStatus("Se ha presionado el ratón");
    }
  }
}
```

Clases internas anónimas

Una clase interna *anónima* es aquella a la que no se le asigna un nombre. En este apartado se muestra cómo es más fácil la escritura de manejadores de eventos utilizando clases internas anónimas. Considere al applet que se muestra a continuación. Su objetivo es, al igual que antes, visualizar la cadena "Se ha presionado el ratón" en la barra de estado del visor de applets o del navegador cuando se presione el ratón.

```
// Ejemplo de una clase interna anónima.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="AnonymousInnerClassDemo" width=200 height=l00>
  </applet>
*/
public class AnonymousInnerClassDemo extends Applet {
  public void init() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        showStatus("Se ha presionado el ratón");
      }
    });
  }
}
```

En este programa hay una clase de alto nivel: **AnonymousInnerClassDemo**. El método **init( )** llama al método **addMouseListener( )**. Su argumento es una expresión que define y cita una clase interna anónima. Es interesante analizar con detalle esta expresión.

La sintaxis **new MouseAdapter( ) { ... }** indica al compilador que el código que está entre llaves define una clase interna anónima. Además, esa clase extiende a **MouseAdapter**. Esta nueva clase no tiene nombre, pero es automáticamente solicitada cuando se ejecuta esta expresión.

Debido a que se define esta clase interna anónima dentro del ámbito de **AnonymousInnerClassDemo**, ésta tiene acceso a todos los métodos y variables dentro del ámbito de esa clase. Por lo tanto, puede llamar directamente al método **showStatus( )**.

Como se acaba de mostrar, tanto las clases internas anónimas como las no anónimas resuelven algunos molestos problemas de una manera efectiva y sencilla. También permiten crear un código más eficiente.