TEMA 12: MODELO DE EVENTOS

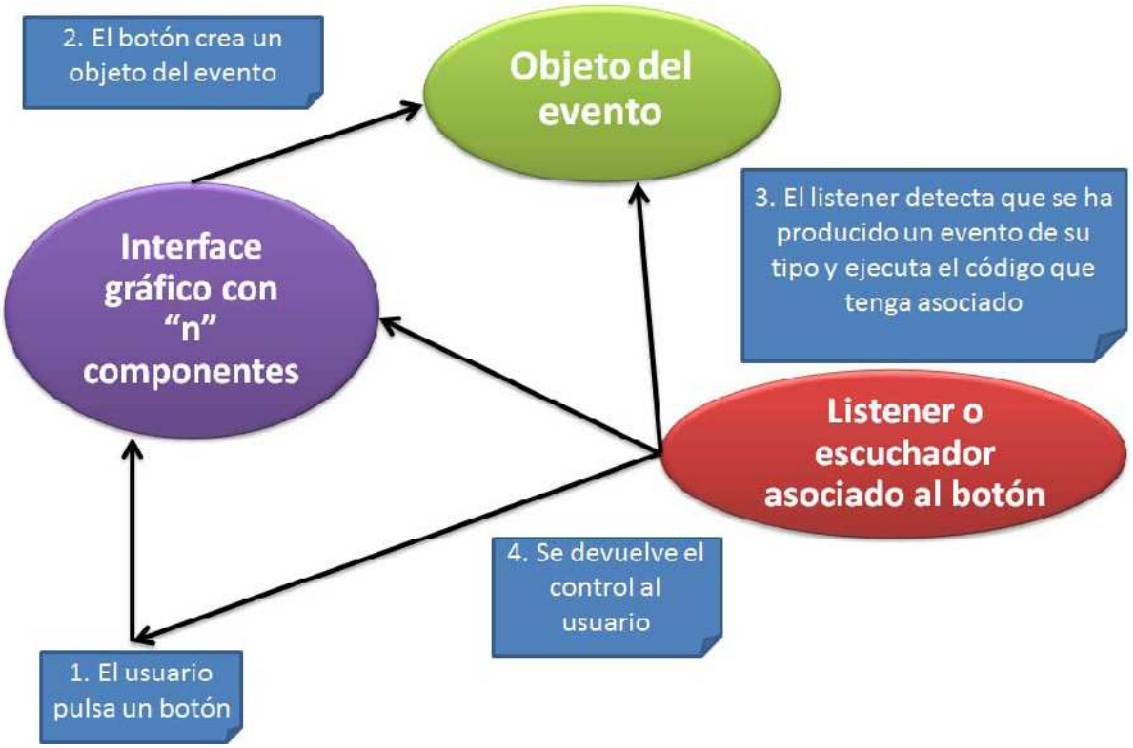
1. INTRODUCCIÓN

Una vez que ya hemos definido como crear proyectos en Java para desarrollar interfaces gráficas de usuario vamos a aprender a interactuar con el usuario, es decir, ante una acción realizada como puede ser que pulsar un botón que se produzca una respuesta, a esto es a lo que se le conoce como el **Modelo de Eventos**.

El modelo de eventos de ***Java*** está basado en que los objetos sobre los que se producen los eventos (***event sources***) "registran" los objetos que habrán de gestionarlos (***event listeners***), para lo cual los ***event listeners*** habrán de disponer de los ***métodos*** adecuados. Estos métodos se llamarán automáticamente cuando se produzca el evento. La forma de garantizar que los ***event listeners*** disponen de los métodos apropiados para gestionar los eventos es obligarles a implementar una determinada interface ***Listener***. Las interfaces ***Listener*** se corresponden con los tipos de ***eventos*** que se pueden producir. En los apartados siguientes se verán con más detalle los ***componentes*** que pueden recibir ***eventos***, los distintos tipos de ***eventos*** y los ***métodos*** de las interfaces ***Listener*** que hay que definir para gestionarlos. En este punto es muy importante ser capaz de buscar la información correspondiente en la documentación de ***Java***.

Dicho en otras palabras, cada vez que un usuario ejecuta una acción sobre un programa desarrollado en Java se produce un evento que el sistema operativo recoge y transmite al sistema Java. Cuando la JVM recibe un evento genera una clase de ese tipo y es transmitido a un método para que lo procese. Es decir se crea un objeto al producirse un evento (***Source***) pero lo gestiona otro objeto (***Listener***).

Un esquema general del comportamiento de una interface gráfica es:



No es tarea del desarrollador identificar que evento se ha producido, lo que debe hacer es capturar ese evento e implementar la funcionalidad deseada.

La que identifica el tipo de acción que ha ejecutado un usuario es la JVM a través de una llamada del sistema operativo. Un esquema de este proceso es:

2. PROCESO A SEGUIR PARA CREAR UNA APLICACIÓN INTERACTIVA (ORIENTADA A EVENTOS).

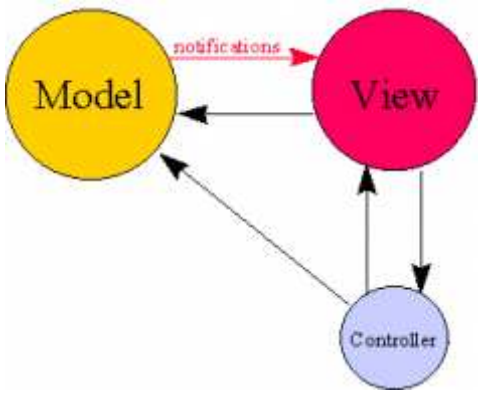
Para avanzar un paso más, se resumen a continuación los pasos que se pueden seguir para construir una aplicación orientada a eventos sencilla, con interface gráfica de usuario:

1. Determinar los ***componentes*** que van a constituir la interface de usuario (botones, cajas de texto, menús, etc.).
2. Crear una ***clase*** para la aplicación que contenga la función ***main()***.
3. Crear una clase ***Ventana***, sub-clase de ***JFrame***, que responda al evento ***WindowClosing()***.
4. La función ***main()*** deberá crear un objeto de la clase ***Ventana*** (en el que se van a introducir las componentes seleccionadas) y mostrarla por pantalla con el tamaño y posición adecuados.
5. Añadir al objeto ***Ventana*** todos los ***componentes*** y ***menús*** que deba contener.
6. Definir los objetos ***Listener*** (objetos que se ocuparán de responder a los eventos, cuyas clases implementan las distintas interfaces ***Listener***) para cada uno de los eventos que deban estar soportados. En aplicaciones pequeñas, el propio objeto ***Ventana*** se puede ocupar de responder a los eventos de sus componentes. En programas más grandes se puede crear uno o más objetos de clases especiales para ocuparse de los eventos.
7. Finalmente, se deben implementar los métodos de las interfaces ***Listener*** que se vayan a hacer cargo de la gestión de los eventos.

Para nuestros primeros ejemplos, nos valdrá el planteamiento anterior, pero cuando nuestras prácticas comiencen a crecer, será más fácil seguir el Modelo Vista Controlador.

3.- MODELO VISTA CONTROLADOR (MVC)

Esta arquitectura da lugar a todo un enfoque de desarrollo muy arraigado en los entornos gráficos de usuario realizados con técnicas orientadas a objetos.



El MVC, en palabras simples, es la forma (Patrón de Diseño) que utilizamos los programadores para implementar nuestras aplicaciones, además permite separar nuestra aplicación en un modelo, una vista y con controlador. Este patrón fue introducido por primera vez en el lenguaje "Smalltalk".

Nos encontramos tres tipos de entidades, cada una especializada en su tarea.

- ***Modelo****:* Es el encargo de administrar la lógica de la aplicación. Tiene como finalidad servir de abstracción de algún proceso en el mundo real, además tiene acceso a nuestra Base de Datos, en caso de que esta exista, teniendo funciones que controlan la integridad del sistema.
- ***Vista:*** Sencillamente es la representación visual del modelo. Es la encargada de representar los componentes visuales en la pantalla. Esta asociada a un Modelo, esto le permite que al momento de cambiar el Modelo, la vista redibujara la parte afectada para reflejar los cambios.
- ***Controlador****:* Es el oyente de los eventos que genere el usuario, es decir es el que permite que interactúen el usuario con el sistema. Interpreta los eventos (las entradas) a través del teclado y/o ratón.

Podemos disponer de:
- Un modelo.
- Varias vistas.
- Varios controladores.

Vistas y Controladores están muy relacionados. A veces puede disponerse de varias vistas para el mismo controlador. Por su parte, el modelo debe ser lo más independiente posible de la vista y del controlador.

¿Por qué usar MVC?

Porque fue diseñada para reducir el esfuerzo a la hora de desarrollar un programa. Además porque permite una clara separación entre los componentes de un programa; lo cual nos permite implementarlos por separado. Permite el trabajo en equipo.

¿Java implementa MVC?

Con decir, que la mayoría de los componentes SWING, han tomado como patrón de Diseño a MVC, esto es una gran ventaja para los programadores, porque nos permite implementar nuestro PROPIO modelo de datos para cada componente swing.

Sin embargo es bueno recordar que el modelo original MVC prescribía un alto acoplamiento entre controladores y vistas.

Ejemplo de MVC – La clase Cuenta

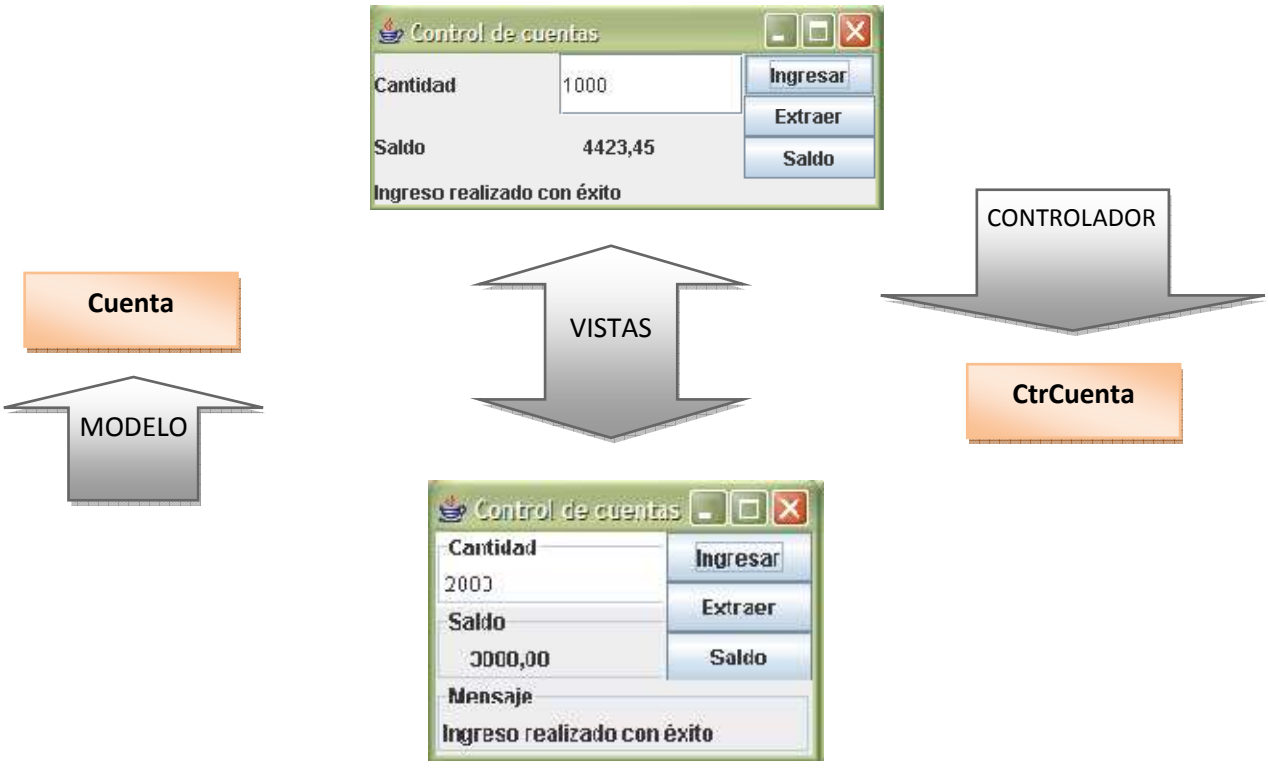
Utilizaremos un objeto de esta clase como modelo. La clase Cuenta permite manipular una cuenta bancaria. Dispondrá de métodos para ingresar, extraer y consultar el saldo.

```
public class Cuenta {
	private double saldo;

	public Cuenta(double si) { saldo = Math.max(0, si); }

	public void ingresa(double ing) {saldo += ing; }

	public double extrae(double extrae) {
		double realExtrae = extrae;
		if (saldo < extrae) {
			realExtrae = saldo;
			saldo = 0;
		} else {
			saldo -= realExtrae;
		}
		return realExtrae;
	}

	public double saldo() { return saldo;}
}
```

Una posible aplicación para esta clase Cuenta en modo consola sería:

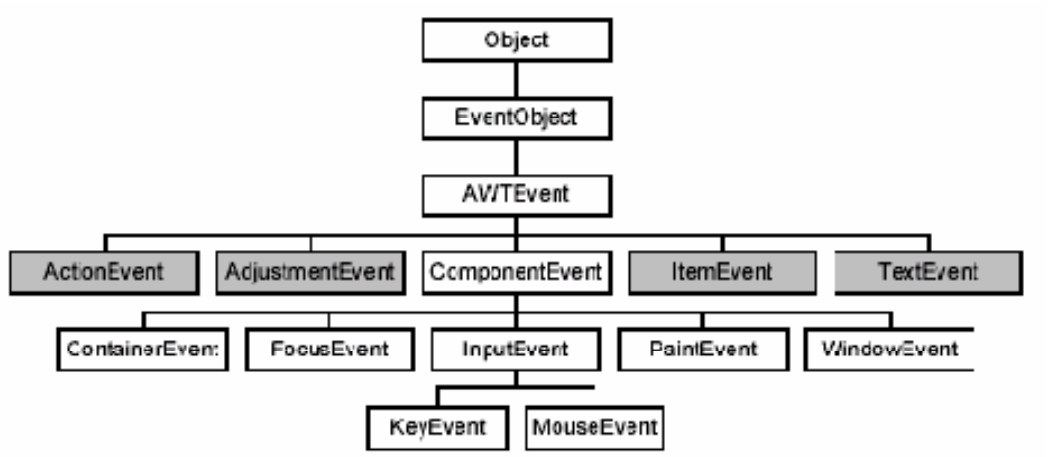
```
public class ApCuenta {
	public static void main(String args[]) {
		Cuenta cuenta = new Cuenta(Double.parseDouble(args[0]));
		cuenta.ingresa(3000);
		double realExt = cuenta.extrae(6000);
		System.out.println("Saldo = " + cuenta.saldo());
		System.out.println("Extraído = " + realExt);
	}
}
```

A continuación se muestra un esquema usando en modo gráfico el MVC.



4. TIPOS DE EVENTOS

Todos los eventos de *AWT* son objetos de clases que pertenecen a una jerarquía como la de la figura. Las clases de la jerarquía se encuentran se encuentran definidas en el *package* **java.awt.event.**



En ***AWT*** existen dos tipos de eventos, los eventos de alto nivel y los eventos de bajo nivel. Los eventos de alto nivel suelen implicar muchos de bajo nivel.

Los de alto nivel son:

- **ActionEvent**, clicar sobre un botón o elegir un elemento del menú, y algunas otras que ya veremos.
- **AdjustmentEvent**, mover las barras de desplazamiento.
- **ItemEvent**, elegir valores.
- **TextEvent**, cambiar texto.

Los de bajo nivel son:

- **ComponentEvent**, eventos elementales relacionados con componentes.
- **ContainerEvent**, eventos elementales relacionados con contenedores.
- **KeyEvent,** eventos relacionados con las pulsaciones sobre el teclado.
- **MouseEvent**, eventos relacionados con las pulsaciones del ratón.
- **FocusEvent**, eventos relacionados con el focus.
- **WindowEvent**, eventos elementales relacionados con ventanas.

El modelo de eventos que utiliza ***Swing*** es el mismo que *AWT*, el de *Java* 1.1, añadiendo algunos nuevos eventos para los nuevos componentes. Utilizando igualmente las interfaces *Listener*, las clases *Adapter* o las clases anónimas para registrar los objetos que se encargaran de gestionar los eventos. Algunos de los nuevos eventos son:

Eventos de bajo nivel

- **MenuKeyEvent**
- **MenuDragMouseEvent**

Eventos de alto nivel

- **AncestorEvent:** Antecesor añadido desplazado o eliminado.
- **CaretEvent:** El signo de intercalación del texto ha cambiado.

- **ChangeEvent:** Un componente ha sufrido un cambio de estado.
- **DocumentEvent:** Un documento ha sufrido un cambio de estado.
- **HyperlinkEvent:** Algo relacionado con un vínculo hipermedia ha cambiado.
- **InternalFrameEvent:** Un AWTEvent que añade soporte para objetos JInternalFrame.
- **ListDataEvent:** El contenido de una lista ha cambiado o se ha añadido o eliminado un intervalo.
- **ListSelectionEvent**: La selección de una lista ha cambiado.
- **MenuEvent:** Un elemento de menú ha sido seleccionado o mostrado o bien no seleccionado o cancelado.
- **PopupMenuEvent:** Algo ha cambiado en JPopupMenu.
- **TableColumnModelEvent:** El modelo para una columna de tabla ha cambiando.
- **TableModelEvent:** El modelo de una tabla ha cambiado.
- **TreeExpansionEvent:** El nodo de un árbol se ha extendido o se ha colapsado.
- **TreeModelEvent:** El modelo de un árbol ha cambiado.
- **TreeSelectionEvent:** La selección de un árbol ha cambiado de estado.
- **UndoableEditEvent:** Ha ocurrido una operación que no se puede realizar.

A continuación se resumen los eventos que pueden ser generados por componentes Swing (tanto los suyos propios, como los heredados de AWT), enfocándose en los eventos que manejan típicamente los programas. Desde este punto de vista los eventos generados por componentes Swing se dividen en tres categorías:

- Eventos que todos los componentes Swing pueden generar
- Otros eventos comunes.
- Eventos no manejados comúnmente .

Eventos que todos los componentes Swing pueden generar

Como todos los componentes Swing descienden de la clase **Component** del AWT, todo ellos soportan los siguientes eventos definidos enel AWT:

- **Component**: Notifica a los oyentes cambios en el tamaño, posición o visibilidad del componente.
- **Focus**: Notifica a los oyentes que el componente a ganado o perdido la posibilidad de recibir entrada desde el teclado.
- **Key**: Notifica a los oyentes las pulsaciones de teclas; sólo generado por el componente que tiene el foco del teclado.
- **Mouse:** Notifica a los oyentes las pulsaciones del ratón y los movimientos de entrada y salida del usuario en el área de dibujo del componente.
- **Mouse Motion**: Notifica a los oyentes cambios en la posición del cursor sobre el componente.

Aunque todos los componentes Swing descienden de la clase **Container** del AWT, muchos de ellos no son usados como contenedores. Por eso, técnicamente hablando cualquier componente Swing puede generar eventos **Container**, que notifican a los oyentes que se ha añadido o eliminado un componente del contenedor. Sin embargo, hablando en forma real sólo los contenedores como los paneles, marcos, etc., generan eventos container.

Otros Eventos comunes

La siguiente tabla lista los eventos más comúnmente manejados que varios componentes Swing pueden generar.

Nota: un asterisco '*' en una cabecera de columna indica un evento definido en el AWT. Todos los otros eventos de la tabla está definidos en **javax.swing.event**.

Componente Swing	Tipos de Eventos Generados								
	action *	caret	change	document	internal frame	item *	list selection	undoable edit	window *
ColorSelectionModel (JColorChooser'smodelo de seleccion por defecto.) **Nota:** Esta *no* es una subclase de JComponent!			X						
Document (JTextComponent's modelo de datos.) **Nota:** Esta *no* es una sublcase de JComponent!				X				X	
JButton	X		X			X			
JCheckBox	X		X			X			
JComboBox	X					X			
JDialog									X
JEditorPane		X							
JFileChooser	X								
JFrame									X
JInternalFrame					X				
JList							X		
JMenuItem	X		X			X			
JOptionpane									X
JPasswordField	X	X							
JProgressBar			X						
JRadioButton	X		X			X	▪		
JSlider			X						
JTabbedPane			X						
JTextArea		X							
JTextComponent		X							
JTextField	X	X							
JTextPane		X							
JToggleButton	X		X			X			
JViewport			X						
ListSelectionModel (JList's modelo de selección por defecto.) **Nota:** Esta *no* es una subclase JComponent!							X		
Timer **Nota:** Esta *not* es una subclase JComponent!	X								

Eventos no manejados comúmente

Como recordatorio, esta sección lista otros eventos que los componentes Swing pueden generar pero que los programas típicos no necesitan manejar.

Todos los componentes que descienden de la clase **JComponent** pueden generar los eventos descritos en la siguiente lista:

- **Ancestor** : Un componente genera un evento Ancestor cuando uno de sus contenedores acenstros es añadido o eliminado de un contenedor, es ocultado, visualizado o movido. Este tipo de evento es una implementación detallada y generalmente puede ser ignorado.
- **Property Change** : Definido en **java.beans** los componentes Swing generan este tipo de eventos porque son compatibles con JavaBeans. Los Beans utilizan los eventos Change para implementar propiedades compartidas.
- **Vetoable Change** : Definido en **java.beans** los componentes Swing generan este tipo de eventos porque son compatibles con JavaBeans. Los Beans utilizan estos eventos para implementar propiedades restringidas.

La siguiente tabla lista todos los otos eventos definidos en **javax.swing.event** que no se han mencionado anteriormente.

Eventos de Editor de Celdas	Eventos de Teclas de Menú	Eventos de Expansión de Árboles
Hyperlink	Menu	Tree Model
List Data	Popup Menu	Tree Selection
Menu Drag Mouse	Table Model	Tree Will Expand

5.- INTERFACES LISTENER.

Una vez vistos los distintos eventos que se pueden producir, conviene ver cómo se deben gestionar estos eventos. A continuación se detalla cómo se gestionan los eventos según el modelo de ***Java***:

1. Cada objeto que puede recibir un evento (***event source***), "registra" uno o más objetos para que los gestionen (***event listener***). Esto se hace con un método que tiene la forma,

```
eventSourceObject.addEventListener(eventListenerObject);
```

donde ***eventSourceObject*** es el objeto en el que se produce el evento, y ***eventListenerObject*** es el objeto que deberá gestionar los eventos. La relación entre ambos se establece a través de una interface ***Listener*** que la clase del ***eventListenerObject*** debe implementar. Esta interface proporciona la declaración de los métodos que serán llamados cuando se produzca el evento. La interface a implementar depende del tipo de evento. La siguiente tabla relaciona los distintos tipos de eventos, con la interface que se debe implementar para gestionarlos. Se indican también los métodos declarados en cada interface.

En la tabla que sigue, cada fila describe un grupo de eventos particular correspondiente a un interface oyente. La primer columna ofrece el nombre el interface, la segunda columna nombra la correspondiente clase adaptador (se verán más adelante), si existe, la tercera columna indica el paquete en que se definen el interface, la clase event y la case adaptador y la cuarta columna lista los métodos que contiene el interface.

Interface	Clase Adaptador	Paquete	Métodos
ActionListener	ninguna	`java.awt.event`	`actionPerformed`
CaretListener	ninguna	`javax.swing.event`	`caretUpdate`
ChangeListener	ninguna	`javax.swing.event`	`stateChanged`

ComponentListener	ComponentAdapter	`java.awt.event`	`componentHidden`
			`componentMoved`
			`componentResized`
			`componentShown`

ContainerListener	ContainerAdapter	`java.awt.event`	`componentAdded` `componentRemoved`

DocumentListener	ninguna	`javax.swing.event`	`changedUpdate`
			`insertUpdate`
			`removeUpdate`

FocusListener	FocusAdapter	`java.awt.event`	`focusGained` `focusLost`

InternalFrameListener	InternalFrameAdapter	`javax.swing.event`	`internalFrameActivated`
			`internalFrameClosed`
			`internalFrameClosing`
			`internalFrameDeactivated`
			`internalFrameDeiconified`
			`internalFrameIconified`
			`internalFrameOpened`

ItemListener	ninguna	`java.awt.event`	`itemStateChanged`

KeyListener	KeyAdapter	`java.awt.event`	`keyPressed`
			`keyReleased`
			`keyTyped`

ListSelectionListener	ninguna	`javax.swing.event`	`valueChanged`

MouseListener	MouseAdapter
	MouseInputAdapter *
		`java.awt.event`
		`javax.swing.event`
			`mouseClicked`
			`mouseEntered`
			`mouseExited`
			`mousePressed`
			`mouseReleased`

MouseMotionListener	MouseMotionAdapter MouseInputAdapter *	`java.awt.event` `javax.swing.event`	`mouseDragged` `mouseMoved`
UndoableEditListener	none	`javax.swing.event`	`undoableEditHappened`

WindowListener	WindowAdapter	`java.awt.event`	`windowActivated`
			`windowClosed`
			`windowClosing`
			`windowDeactivated`
			`windowDeiconified`
			`windowIconified`
			`windowOpened`

***** Swing proporciona la clase **MouseInputAdapter** por conveniencia. Implementa los interfaces **MouseListener** y **MouseMotionListener** haciendo más fácil para nosotros el manejo de ambos tipos de eventos.

Es importante observar la correspondencia entre ***eventos*** e ***interfaces Listener***. Cada evento tiene su interface, excepto el ratón que tiene dos interfaces ***MouseListener*** y ***MouseMotionListener***. La razón de esta duplicidad de interfaces se encuentra en la peculiaridad de los eventos que se producen cuando el ratón se mueve. Estos eventos, que se producen con muchísima más frecuencia que los simples clicks, por razones de eficiencia son gestionados por una interface especial: ***MouseMotionListener***.

Recuérdese que el ***nombre de la interface*** coincide con el ***nombre del evento***, sustituyendo la palabra ***Event*** por ***Listener***. A modo de ejemplo, si el evento que se genera es de tipo ***action***, su clase será ActionEvent, y la interfaz que lo maneja ActionListener, o si el evento fuese de tipo ***item***, su clase sería ItemEvent, y su interfaz ItemListener.

2. Una vez registrado el objeto que gestionará el evento, perteneciente a una clase que implemente la correspondiente interface ***Listener***, se deben definir los métodos de dicha interface. Siempre hay que definir ***todos los métodos*** de la interface, aunque algunos de dichos métodos puedan estar "vacíos".

No obstante se puede evitar tener que definir todos los métodos utilizando para ello adaptadores, como ser estudiará más adelante.

Veamos a continuación un primer ejemplo, en este caso el oyente será la propia ventana o clase, es decir esta clase implementa la interfaz listener necesaria.

```
public class GUI01A extends JFrame implements ActionListener{

   // Variable de instancia
   private JLabel etiqueta;
   private JButton bSi, bNo;

   // Constructor
   public GUI01A()
   {
          // Poner título a la ventana
          super("Con actionListener");

          // Creo un objeto de tipo container con el método getContentPane()
          Container contentPane = this.getContentPane();

          // Establecer el gestor de esquemas
          contentPane.setLayout(new FlowLayout());

          // Crear los componentes
          bSi = new JButton("Si");
          bNo = new JButton("No");
          etiqueta = new JLabel("Pulsaciones");

          // Añadir los componentes al JFrame
          contentPane.add(etiqueta);
          contentPane.add(bSi);
          contentPane.add(bNo);

          // Pack, setVisible, control de cierre, etc
          this.pack();
          this.setVisible(true);
          this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

          // Añadir control a los botones (enlazar el componente event con
          // la interfaz que lo que programa)
          bSi.addActionListener(this);
          bNo.addActionListener(this);
   }

   // Programa principal
   public static void main(String[] args) {
          new GUI01A();
   }


   // Método de la interfaz ActionListener
   public void actionPerformed(ActionEvent e) {

          if (e.getSource()==bSi)
                 etiqueta.setText("Si pulsado");
          else
                 etiqueta.setText("No pulsado");

   }
}
```

Continuaremos ahora viendo otra vez, el mismo ejemplo, pero en este caso, el oyente no será la propia ventana, sino un con objeto con visibilidad, es decir, tendremos una clase distinta que será la que implemente la interfaz necesaria.

```
public class GUI01B extends JFrame{

	// Variable de instancia
	private JLabel etiqueta;
	private JButton bSi, bNo;

	// Constructor
	public GUI01B()
	{
		// Poner título a la ventana
		super("Con actionListener");

		// Creo un objeto de tipo container con el método getContentPane()
		Container contentPane = this.getContentPane();

		// Establecer el gestor de esquemas
		contentPane.setLayout(new FlowLayout());

		// Crear los componentes
		bSi = new JButton("Si");
		bNo = new JButton("No");
		etiqueta = new JLabel("Pulsaciones");

		// Añadir los componentes al JFrame
		contentPane.add(etiqueta);
		contentPane.add(bSi);
		contentPane.add(bNo);

		// Pack, setVisible, control de cierre, etc
		this.pack();
		this.setVisible(true);
		this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		// Añadir control a los botones
		ControlBoton cb = new ControlBoton();
		bSi.addActionListener(cb);
		bNo.addActionListener(cb);
	}

	// Programa principal
	public static void main(String[] args) {
		new GUI01B();
	}


	class ControlBoton implements ActionListener
	{
		// Método de la interfaz ActionListener
		public void actionPerformed(ActionEvent e) {
			if (e.getSource()==bSi)
				etiqueta.setText("Si pulsado");
			else
				etiqueta.setText("No pulsado");
		}
	}
}
```

En estos dos ejemplos hemos definido, todos los componentes de nuestra interfaz gráfica, como variables de instancia, para poder acceder a ellas desde fuera del constructor, pero en realidad, sólo sería necesario que estuvieseen como variables de instancia aquellas variables a las que necesitemos posteriormente acceder desde fuera del, valga la redundancia, el constructor. En este caso en realidad, no estamos accediendo ni al Jlabel, ni a al botón NO, por tanto podrían haber sido variables locales al constructor.

6. CLASES EVENT

6.1. Clases EventObject y AWTEvent.

Todos los métodos de las interfaces ***Listener*** relacionados con el AWT tienen como argumento único un objeto de alguna clase que desciende de la clase ***java.awt.AWTEvent***.

La clase ***AWTEvent*** desciende de ***java.util.EventObject***. La clase ***AWTEvent*** no define ningún método, pero hereda de ***EventObject*** el método `getSource()`:

```
Object getSource();
```

que devuelve una referencia al objeto que generó el evento. Las clases de eventos que descienden de ***AWTEvent*** definen métodos similares a ***getSource()*** con unos valores de retorno menos genéricos. Por ejemplo, la clase ***ComponentEvent*** define el método ***getComponent()***, cuyo valor de retorno es un objeto de la clase ***Component***.

6.2. Clase ComponentEvent.

Los eventos ***ComponentEvent*** se generan cuando un ***Component*** de cualquier tipo se muestra, se oculta, o cambia de posición o de tamaño. Los eventos de *mostrar* u *ocultar* ocurren cuando se llama al método ***setVisible(boolean)*** del ***Component***, pero no cuando se *minimiza* la ventana.

Otro método útil de la clase ***ComponentEvent*** es `Component getComponent()` que devuelve el componente que generó el evento. Se puede utilizar en lugar de ***getSource()***.

6.3. Clase ContainerEvent.

Los ***ContainerEvents*** se generan cada vez que un ***Component*** se añade o se retira de un ***Container***. Estos eventos sólo tienen un *papel de aviso* y no es necesario gestionarlos para que se realize la operación.

Los métodos de esta clase son `Component getChild()`, que devuelve el ***Component*** añadido o eliminado, y `Container getContainer()`, que devuelve el ***Container*** que generó el evento, se puede usar en lugar de ***getSource()***.

6.4. Clase ActionEvent.

Los eventos ***ActionEvent*** son probablemente los más sencillos y comunes de manejar, se producen al clicar con el ratón en un botón (***Button***), al elegir un comando de un menú (***MenuItem***), y al pulsar ***Intro*** para introducir un texto en una caja de texto (***TextField***).

La clase **ActionEvent** define dos métodos muy útiles:

- El método `String getActionCommand()` devuelve el texto asociado con la acción que provocó el evento. Este texto se puede fijar con el método `setActionCommand(String str)` de las clases ***Button*** y ***MenuItem***. Si el texto no se ha fijado con este método, el método ***getActionCommand()*** devuelve el texto mostrado por el componente (su etiqueta). Para objetos con varios items el valor devuelto es el nombre del item seleccionado.

- El método `int getModifiers()` devuelve un entero representando una constante definida en ***ActionEvent*** (SHIFT_MASK, CTRL_MASK, META_MASK y ALT_MASK). Estas constantes sirven para determinar si se pulsó una de estas teclas modificadores mientras se clicaba. Por ejemplo, si se estaba pulsando la tecla CTRL la siguiente expresión es distinta de cero:

```
actionEvent.getModifiers() & Action.Event.CTRL_MASK
```

Veamos otra vez, en una tercera versión, el ejemplo que vimos en el apartado anterior, pero en esta ocasión utilizando los métodos *actionCommand ("alias")*. En este caso, el oyente vuelve a ser un objeto con visibilidad, es decir, está en otra clase distitnta.

```
public class GUI01C extends JFrame{

	// Variable de instancia
	private JLabel etiqueta;
	private JButton bSi, bNo;

	// Constructor
	public GUI01C()
	{
		// Poner título a la ventana
		super("Con actionListener");

		// Creo un objeto de tipo container con el método
		// getContentPane()
		Container contentPane = this.getContentPane();

		// Establecer el gestor de esquemas
		contentPane.setLayout(new FlowLayout());

		// Crear los componentes
		bSi = new JButton("Si");
		bNo = new JButton("No");
		etiqueta = new JLabel("Pulsaciones");

		// Añadir "alias" con el metodo setActionCommnad a los botones
		bSi.setActionCommand("botonSi");
		bNo.setActionCommand("botonNo");


		// Añadir los componentes al JFrame
		contentPane.add(etiqueta);
		contentPane.add(bSi);
		contentPane.add(bNo);

		// Pack, setVisible, control de cierre, etc
		this.pack();
		this.setVisible(true);
		this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

		// Añadir control a los botones
		ControlBoton cb = new ControlBoton();
		bSi.addActionListener(cb);
		bNo.addActionListener(cb);
	}

	// Programa principal
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		new GUI01C();
	}


	class ControlBoton implements ActionListener
	{
		// Método de la interfaz ActionListener
		public void actionPerformed(ActionEvent e) {
			if (e.getActionCommand().equals("botonSi"))
				etiqueta.setText("Si pulsado");
			else
				etiqueta.setText("No pulsado");
		}
	}
}
```

6.5. Clase FocusEvent.

El ***Focus*** está relacionado con la posibilidad de sustituir al ratón por el teclado en ciertas operaciones. De los componentes que aparecen en pantalla, en un momento dado hay sólo uno que puede recibir las acciones del teclado y se dice que ese componente tiene el ***Focus***. El componente que tiene el ***Focus*** aparece diferente de los demás (resaltado de alguna forma). Se cambia el elemento que tiene el ***Focus*** con la tecla ***Tab*** o con el ratón. Se produce un ***FocusEvent*** cada vez que un componente gana o pierde el ***Focus***.

El método `requestFocus()` de la clase ***Component*** permite hacer desde el programa que un componente obtenga el ***Focus***.

El método `boolean isTemporary()`, de la clase ***FocusEvent***, indica si la pérdida del ***Focus*** es o no temporal (puede ser temporal por haberse ocultado o dejar de estar activa la ventana, y recuperarse al cesar esta circunstancia).

El método ***Component getComponent()*** es heredado de ***ComponentEvent***, y permite conocer el componente que ha ganado o perdido el ***Focus***. Las constantes de esta clase ***FOCUS_GAINED*** y ***FOCUS_LOST*** permiten saber el tipo de evento ***FocusEvent*** que se ha producido.

6.6. Clase ItemEvent.

Los eventos item son generados por componentes que implementan el interface **ItemSelectable**. Estos son componentes que mantienen el estado – generalmente on/off -- de uno o más ítems. Los componentes Swing que pueden generar estos eventos son **JCheckbox, JCheckboxMenuItem, y JComboBox**. En AWT los **List**, también generaban este tipo de evento, pero en Swing, los eventos de la clase **Jlist**, tienen sus propios eventos del tipo *ListSelectionEvent*.

La clase **ItemEvent** define los siguientes métodos:

- `Object getItem():` Devuelve el objeto component específico asociado con el ítem cuyo estado ha cambiado. Normalmente es un **String** que contiene el texto del ítem seleccionado. Para un evento item generado por un **JComboBox**, es un **Integer** que especifica el índice del ítem seleccionado.
- `ItemSelectable getItemSelectable():` Devuelve el componente que genero el evento item. Podemos usarlo en lugar del método **getSource**.
- `int getStateChange():` Devuelve el nuevo estado del ítem. La clase ***ItemEvent*** define las constantes enteras SELECTED y DESELECTED, que se pueden utilizar para comparar con el valor devuelto por el método ***getStateChange()***.

6.7. Clases InputEvent, MouseEvent y MouseMotionEvent.

De la clase ***InputEvent*** descienden los eventos del ratón y el teclado. Esta clase dispone de métodos para detectar si los botones del ratón o las teclas especiales han sido pulsadas. Estos botones y estas teclas se utilizan para cambiar o modificar el significado de las acciones del usuario. La clase ***InputEvent*** define unas constantes que permiten saber qué teclas especiales o botones del ratón estaban pulsados al producirse el evento, como son: SHIFT_MASK, ALT_MASK, CTRL_MASK, BUTTON1_MASK, BUTTON2_MASK y BUTTON3_MASK, cuyo significado es evidente.

Por ejemplo, la siguiente expresión es verdadera si se pulsó el botón derecho:

```
(mouseEvent.getModifiers() & InputEvent.BUTTON3_MASK) == InputEvent.BUTTON3_MASK
```

La siguiente tabla muestra algunos métodos de esta clase:

Métodos heredados de la clase InputEvent	***Función que realizan***
`isShiftDown(), isAltDown(), isControlDown()`	Devuelven un boolean con información sobre si esa tecla estaba pulsada o no
`int getModifiers()`	Obtiene información con una máscara de bits sobre las teclas y botones pulsados
`long getWhen()`	Devuelve la hora en que se produjo el evento

Se produce un ***MouseEvent*** cada vez que el cursor movido por el ratón (o dispositivo de entrada similar) entra o sale de un componente visible en la pantalla, al clicar, o cuando se pulsa o se suelta un botón del ratón. Los métodos de la interface ***MouseListener*** se relacionan con estas acciones, y son los siguientes:

- ***mouseClicked( ).*** Llamado justo después de que el usuario pulse sobre el componente escuchado.
- ***mouseEntered( ).***Llamado justo después de que el cursor entre en los límites del componente escuchado.
- ***mouseExited( ).***Llamado justo después de que el cursor salga de los límites del componente escuchado.
- ***mousePressed( ).*** Llamado justo después de que el usuario pulse un botón del ratón mientras el cursor está sobre el componente escuchado
- ***mouseReleased( ).*** Llamado justo después de que el usuario libere un botón del ratón después de una pulsación sobre el componente escuchado.

Todos son *void* y reciben como argumento un objeto ***MouseEvent***. La siguiente tabla muestra algunos métodos de la clase ***MouseEvent***.

Los Métodos de la clase MouseEvent	***Función que realizan***
`int getClickCount()`	Devuelve el número de clicks en ese evento
`Point getPoint(), int getX(), int getY()`	Devuelven la posición del ratón al producirse el evento
`boolean isPopupTrigger()`	Devuelve true si el evento mouse debería hacer que apareciera un menú popup. Como los disparadores de menús popup son dependientes de la plataforma, si nuestro programa los usa, deberíamos llamar a **isPopupTrigger** en todos los eventos mouse-pressed y mouse-released generados por componentes sobre los que el popup pueda aparecer.

La clase ***MouseEvent*** define una serie de constantes ***int*** que permiten identificar los tipos de eventos que se han producido: MOUSE_CLICKED, MOUSE_PRESSED, MOUSE_RELEASED, MOUSE_MOVED, MOUSE_ENTERED, MOUSE_EXITED, MOUSE_DRAGGED.

Además, el método ***Component getComponent()***, heredado de ***ComponentEvent***, devuelve el componente sobre el que se ha producido el evento.

Una complicación afecta a los eventos mouse-entered, mouse-exited, y mouse-released. Cuando el usuario arrastra (pulsa y mantiene el botón del ratón y luego mueve el ratón), entonces el componente sobre el que estaba el cursor cuando empezó el arrastre es el que recibe todos los subsecuentes eventos de mouse y mouse-motion incluyendo la liberación del botón. Esto significa que ningún otro componente recibera un solo evento del ratón -- ni siquiera un evento mouse-released -- mientras está ocurriendo el arrastre.

La clase **MouseEvent** hereda otros muchos métodos útiles de **InputEvent**:

Los Métodos de la clase MouseEvent	***Función que realizan***
`void consume()`	Hace que el evento no sea procesado por el padre del componente. Se podría usar este método para descartar letras tecleadas en un campo de texto que sólo acepta números.
`int getWhen()`	Devuelve el momento en que ocurrió el evento.

`boolean isAltDown()`
`boolean isControlDown()`
`boolean isMetaDown()`
`boolean isShiftDown()`
	Devuelven el estado individual de las teclas modificadores en el momento en que se generó el evento.

`int getModifiers()`	Devuelve el estado de todas las teclas modificadoras y botones del ratón, cuando se generó el evento. Podemos usar este método para determinar qué botón fue pulsado (o liberado) cuando el evento del ratón fue generado.

La clase **SwingUtilities** contiene métodos de conveniencia para determinar si se ha pulsado un botón particular del ratón:

- `static boolean isLeftMouseButton(MouseEvent)`
- `static boolean isMiddleMouseButton(MouseEvent)`
- `static boolean isLEFTMouseButton(MouseEvent)`

Los eventos ***MouseEvent*** disponen de una segunda interface para su gestión, la interface ***MouseMotionListener***, cuyos métodos reciben también como argumento un evento de la clase ***MouseEvent***. Estos eventos están relacionados con el ***movimiento del ratón***. Se llama a un método de la interface ***MouseMotionListener*** cuando el usuario utiliza el ratón (o un dispositivo similar) para mover el cursor o arrastrarlo sobre la pantalla. Los métodos de la interface ***MouseMotionListener*** son:

- ***mouseMoved() .*** Llamado en respuesta a un movimiento del ratón por parte del usuario mientras mantiene pulsado uno de los botones del ratón. Este evento es disparado por el componente que disparó el evento mouse-pressed más reciente, incluso si el cursor ya no está sobre ese componente.

- ***mouseDragged()***.Llamado en respuesta a un movimiento del ratón por parte del usuario sin ningún botón pulsado. El evento es disparado por el evento que se encuentra actualmente debajo del cursor.

6.8. Clase KeyEvent.

Se produce un ***KeyEvent*** al pulsar sobre el teclado. Como el teclado no es un componente del AWT, ni de Swing, es el ***objeto que tiene el focus*** en ese momento quien genera los eventos ***KeyEvent*** (que es el *event source*).

Hay dos tipos de ***KeyEvents***:

1. ***key-typed***, que representa la introducción de un carácter Unicode.
2. ***key-pressed*** y ***key-released***, que representan pulsar o soltar una tecla. Son importantes para teclas que no representan caracteres, como por ejemplo F1.

En general, sólo deberíamos manejar los eventos key-typed a menos que necesitemos saber cuando el usuario ha pulsado teclas que no corresponden con caracteres. Por ejemplo, si queremos saber cuando el usuario teclea algún carácter Unicode -- siempre como resultado de una pulsación de tecla como 'a' de la pulsación de una secuencia de teclas -- deberíamos manejar eventos key-typed. Por otro lado, si queremos saber cuando el usuario ha pulsado la tecla F1, necesitaremos manejar eventos key-pressed.

Nota: Para generar eventos de teclado, un componente **debe** tener el foco del teclado.

Para hacer que un componente obtenga el foco del teclado debemos seguir estos pasos:

1.- Asegurarnos de que el componente puede obtener el foco del teclado. Por ejemplo, en algunos sistemas las etiquetas no pueden obtenerlo.

2.- Asegurarnos de que el componente pide el foco en el momento apropiado. Para componentes personalizados, probablemente necesitaremos implementar un **MouseListener** que llame al método **requestFocus** cuando se pulsa el ratón.

3.- Si estamos escribiendo un componente personalizado, implementaremos el método **isFocusTraversable** del componente, para que devuelva true cuando el componente está activado. Esto permite al usuario usar Tab para ir a nuestro componente.

Para estudiar estos eventos son muy importantes las ***Virtual Key Codes*** (VKC). Los VKC son unas constantes que se corresponden con las ***teclas físicas*** del teclado, sin considerar minúsculas (que no tienen VKC). Se indican con el prefijo VK_, como VK_SHIFT o VK_A. La clase ***KeyEvent*** (en el package ***java.awt.event***) define constantes VKC para todas las teclas del teclado.

Por ejemplo, para escribir la letra "A" mayúscula se generan 5 eventos: *key-pressed VK_SHIFT*, *key-pressed VK_A*, *key-typed "A"*, *key-released VK_A*, y *key-released VK_SHIFT*. La siguiente tabla muestra algunos métodos de la clase ***KeyEvent*** y otros heredados de ***InputEvent***.

Métodos de la clase KeyEvent	***Función que realizan***
`int getKeyChar(),` `void setKeyChar(char)`	Obtiene o establece el carácter Unicode asociado con el evento
`int getKeyCode(),void setKeyCode(int)`	Obtiene o establece el VKC de la tecla pulsada o soltada.
`boolean isActionKey()`	Indica si la tecla del evento es una ActionKey (HOME, END, …)
`String getKeyText(int keyCode)`	Devuelve un String que describe el VKC, tal como "HOME", "F1" o "A". Estos Strings se definen en el fichero awt.properties

`void setModifiers(int)`	Selecciona el estado de las teclas modificadoras para este evento. Podemos obtener el estado de las teclas modificadoras usando el método **getModifiers** de **InputEvent**.
`String getKeyText()` `String getKeyModifiersText(int modifiers)`	Devuelve un String que describe las teclas modificadoras, tales como "Shift" o "Ctrl+Shift" (fichero awt.properties) y de la tecla modificadora, respectivamente.
Métodos heredados de InputEvent	***Función que realizan***

`boolean isShiftDown(),`
`boolean isControlDown(),`
`boolean IsMetaDown(),`
`boolean isAltDown(), int getModifiers()`
	Permiten identificar las teclas modificadoras

Las constantes de ***InputEvent*** permiten identificar las teclas modificadoras y los botones del ratón. Estas constantes son SHIFT_MASK, CTRL_MASK, META_MASK y ALT_MASK. A estas constantes se pueden aplicar las operaciones lógicas de bits para detectar combinaciones de teclas o pulsaciones múltiples.

6.9. Clase WindowEvent

Se produce un ***WindowEvent*** cada vez que se *abre*, *cierra*, *iconiza*, *restaura*, *activa* o *desactiva* una ventana. La interface ***WindowListener*** contiene los siete métodos siguientes, con los que se puede responder a este evento:

- **windowOpened(WindowEvent).** Llamado justos después de que la ventana escuchadas sea mostrada por primera vez.
- **windowClosing(WindowEvent).** Llamada en respuesta a una petición de usuario de que la ventana escuchada sea cerrada. Para cerrar realmente la ventana, el oyente debería invocar a los métodos **dispose** o **setVisible(false)** de window.
- **windowClosed(WindowEvent).** Llamado justo después de que la ventana escuchada sea cerrada.
- **windowIconified(WindowEvent) y windowDeiconified(WindowEvent).** Llamado justo después de que la ventana escuchada sea iconificada o desiconificada, respectivamente.
- **windowActivated(WindowEvent) y windowDeactivated(WindowEvent).** Llamado justo después de que la ventana escuchada sea activada o desactivada, respectivamente.

El uso más frecuente de ***WindowEvent*** es para cerrar ventanas (por defecto, los objetos de la clase ***Frame*** no se pueden cerrar más que con ***Ctrl+Alt+Supr***. También se utiliza para detener ***threads*** y liberar recursos al iconizar una ventana (que contiene por ejemplo animaciones) y comenzar de nuevo al restaurarla.

La clase WindowEvent define la siguiente serie de constantes que permiten identificar el tipo de evento:

```
WINDOW_OPENED, WINDOW_CLOSING, WINDOW_CLOSED, WINDOW_ICONIFIED, WINDOW_DEICONIFIED,
WINDOW_ACTIVATED, WINDOW_DEACTIVATED.
```

En la clase WindowEvent el método `Window getWindow()` devuelve la Window que generó el evento. Se utiliza en lugar de getSource().

6.10. Clase AdjustmentEvent.

Se produce un evento ***AdjustementEvent*** cada vez que se cambia el valor (entero) de una ***Scrollbar***. Hay cinco tipos de ***AdjustementEvent***:

- ***track***: se arrastra el cursor de la ***Scrollbar***.
- ***unit increment***, ***unit decrement***: se clica en las flechas de la ***Scrollbar***.
- ***block increment***, ***block decrement***: se clica encima o debajo del cursor.

Métodos de la clase AdjustementEvent	*Función que realizan*
`Adjustable getAdjustable()`	Devuelve el Component que generó el evento (implementa la interface Adjustable)
`int getAdjustementType()`	Devuelve el tipo de adjustement
`int getValue()`	Devuelve el valor de la Scrollbar después del cambio

La tabla anterior muestra algunos métodos de esta clase. Las constantes UNIT_INCREMENT, UNIT_DECREMENT, BLOCK_INCREMENT, BLOCK_DECREMENT, TRACK permiten saber el tipo de acción producida, comparando con el valor de retorno de ***getAdjustementType()***.

6.11. Clase TextEvent.

Se produce un ***TextEvent*** cada vez que cambia algo en un ***TextComponent*** (***TextArea*** y ***TextField***). Se puede desear evitar ciertos caracteres y para eso hay que gestionar los eventos correspondientes.

La interface ***TextListener*** tiene un único método:

```
void textValueChanged(TextEvent te).
```

No hay métodos propios de la clase ***TextEvent***. Se puede utilizar el método ***Object getSource()***, que es heredado de ***EventObject***.

6.12. Clase CaretEvent

Los eventos de Caret ocurren cuando se mueve el cursor (caret = punto de inserción) en un componente de texto o cuando cambia la selección en un componente de texto. Se puede añadir un oyente de caret a un ejemplar de cualquiera de la subclase de **JTextComponent** con el método **addCaretListener**. Si nuestro programa tiene un cursor personalizado, podríamos encontrar más conveniente añadir un oyente al objeto caret en vez de la componente de texto al que pertenece. Un cursor genera eventos change en vez de eventos caret, por eso necesitaremos escribir un oyente de change en vez de un puente de caret.

El interface **CaretListener** sólo tiene un método y por lo tanto no tiene clase adaptadora:

- **caretUpdate(CaretEvent).** Se le llama cuando se mueve el cursor de un componente de texto o cuando se modifica la selección en un componente de texto. Para obtener el componente de texto que generó el evento, se usa el método `getSource()` que **CaretEvent** hereda de **EventObject**.

La clase **CaretEvent** define dos métodos muy útiles:

- `int getDot().` Devuelve la posición actual del cursor. Si hay texto seleccionado, el cursor marca uno de los finales de la selección.
- `int getMark()`. Devuelve el otro final de la selección. Si no hay nada seleccionado, el valor devuelto por este método es igual al devuelto por **getDot.**

6.13. Clase ChangeEvent

Los eventos Change ocurren cuando un componente que tiene estado cambia éste. Por ejemplo, una barra deslizadora genera un evento change cuando usuario mueve su cursor.

El interface **ChangeListener** tiene sólo un método:

- **stateChanged(ChangeEvent).** Se le llama cuando el componente escuchado cambia de estado. Para obtener el componente que generó el evento se usa el método `getSource()` que **ChangeEvent** hereda de **EventObject**. La clase **ChangeEvent** no define métodos adicionales.

6.14. Clase DocumentEvent

Un componente de texto Swing usa un **Document** para contener y editar un texto. Los eventos Document ocurren cuando el contenido de un documento cambia de alguna forma. Se le añade el oyente de Document al documento del componente, en vez de al propio componente.

El interface **DocumentListener** contiene estos tres métodos:

- **changedUpdate(DocumentEvent).** Se le llama cuando se modifica el estilo o algo del texto. Este tipo de eventos sólo se generan desde un **StyledDocument**—un **PlainDocument** no genera este tipo de eventos.
- **insertUpdate(DocumentEvent).** Se le llama cuando se inserta texto en el documento escuchado.
- **removeUpdate(DocumentEvent).** Se le llama cuando se elimina texto del documento escuchado.

Nunca debemos modificar el contenido de un documento desde dentro de un oyente de document. El programa se podría quedar bloqueado. Para evitarlo, podemos usar un documento personalizado para el componente de texto.

Cada método de evento document tiene un sólo parámetros, un ejemplar de una clase que implemente el interface **DocumentEvent**. Típicamente, el objeto pasado a este método será un ejemplar de **DefaultDocumentEvent** que está definido en **AbstractDocument**.

Para obtener el documento que generó el evento, podemos usar el método `getDocument()` de **DocumentEvent**. Observa que **DocumentEvent** no desciende de **EventObject** como las otras clases de eventos. Por lo tanto, no hereda el método **getSource**. Además de **getDocument**, la clase **DocumentEvent** proporciona otros métodos:

- `int getLength().` Devuelve la longitud del cambio.
- `int getOffset().` Devuelve la posición dentro del documento del primer carácter modificado.
- `ElementChange getChange(Element).` Devuelve detalles sobre qué elementos del documento han cambiado y cómo. **ElementChange** es un interface definido dentro del interface **DocumentEvent**.
- `EventType getType().` Devuelve el tipo de cambio que ha ocurrido. **EventType** es una clase definida dentro del interface **DocumentEvent** que enumera los posibles cambios que pueden ocurrir en un document: insertar y eliminar texto y cambiar el estilo.

6.15. Clase InternalFrameEvent

Los eventos Internal frame son a los **JInternalFrame** lo que los eventos window son a los **JFrame**. Al igual que los eventos window, los eventos internal frame notifican a sus oyentes que la "window" ha sido mostrada por primera vez, ha sido eliminada, iconificada, maximizada, activada o desactivada. Antes de usar este tipo de eventos deberíamos familiarizarnos con los Oyentes de Window.

El interface **InternalFrameListener** contienen estos métodos:

- **internalFrameOpened(InternalFrameEvent).** Se le llama después de el internal frame escuchado se muestre por primera vez.
- **internalFrameClosing(InternalFrameEvent).** Se le llama en respuesta a una petición del usuario de que el internal frame escuchado sea cerrado. Por defecto, **JInternalFrame** oculta la ventana cuando el usuario lo cierra. Podemos usar el método **setDefaultCloseOperation** de **JInternalFrame** para especificar otra opción, que puede ser **DISPOSE_ON_CLOSE** o **DO_NOTHING_ON_CLOSE** (ambas definidas en **WindowConstants**, un interface que implementa **JInternalFrame**). O implementando un método **internalFrameClosing** el oyente del internal frame, podemos añadir un comportamiento personalizado (como mostrar un dialogo o salvar datos) para cerrar un internal frame.
- **internalFrameClosed(InternalFrameEvent).** Se le llama después de que haya desaparecido el internal frame escuchado.
- **internalFrameIconified(InternalFrameEvent) y internalFrameDeiconified(InternalFrameEvent).** Se les llama después de que el internal frame escuchado sea iconificado o maximizado.
- **internalFrameActivated(InternalFrameEvent) y internalFrameDeactivated(InternalFrameEvent).** Se les llama después de que el internal frame escuchado sea activado o desactivado.

6.16. Clase ListSelectionEvent

Los eventos ListSelection ocurren cuando una selección en una list o una table cambia o acaba de cambiar. Los eventos ListSelection son disparados por un objeto que implementa el interface **ListSelectionModel**.

Para detectar un evento ListSelection debemos registrar un oyente con el objeto SelectionModel apropiado. La clase **JList** también ofrece la opción de registrar un oyente sobre la propia lista, mejor que directamente al SelectionModel de la lista.

El interface **ListSelectionListener** sólo tiene un método:

- **void valueChanged(ListSelectionEvent).** Se le llama cuando cambia la selección del componente escuchado, también se la llama después de que la selección haya cambiado.

Cada método de evento ListSelection tiene un sólo parámetro: un objeto **ListSelectionEvent**. Este objeto le dice al oyente que la selección ha cambiado. Un evento ListSelection puede indicar un cambio en la selección de múltiples ítems, discontiguos de la lista. Para obtener la fuente de un **ListSelectionEvent**, se usa el método **getSource**, que **ListSelectionEvent** hereda de **EventObject**. Si registramos un oyente de ListSelection directamente sobre una lista, la fuente de cada evento será la propia lista. De otra forma, sería el SelectionModel.

La clase **ListSelectionEvent** define los siguientes métodos:

- `int getFirstIndex()`. Devuelve el índice del primer ítem cuyo valor de selección ha cambiado. Observa que para selecciones de intervalo múltiple, el primer y último ítem es seguro que han cambiado, pero los ítems que hay entre medias podrían no haberlo hecho.

- `int getLastIndex().` Devuelve el índice del último ítem cuyo valor de selección ha cambiado. Observa que para selecciones de intervalo múltiple, el primer y último ítem es seguro que han cambiado, pero los ítems que hay entre medias podrían no haberlo hecho.
- `int getValueIsAdjusting().` Devuelve **true** si se está modificando todavía la selección. Muchos oyentes de ListSelection sólo están interesados en el estado final de la selección y pueden ignorar eventos cuando este método devuelve **true**.

6.17. Clase UnodableEditEvent

Los eventos UndoableEdit ocurren cuando una operación que puede ser reversible ocurre sobre un componente. Actualmente, sólo los componentes de texto pueden generar eventos de este tipo, y sólo indirectamente. El documento del componente genera el evento. Para los componentes de texto, las operaciones undoables incluyen insertar caracteres, borrarlos, y modificar el estilo del texto.

El interface **UndoableEditListener** tiene un sólo método:

- **void undoableEditHappened(UndoableEditEvent).** Llamado cuando ocurre un evento undoable sobre el componente escuchado.

Los programas normalmente escuchan los eventos UndoableEdit para asistir en la implementación de los comandos "deshacer/repetir".

El método **undoableEditHappened** tiene un sólo parámetros: un objeto **UndoableEditEvent**. Para obtener el documento que generó el evento se usa el método **getSource** que **UndoableEditEvent** hereda de **EventObject**.

La clase **UndoableEditEvent** define un método que devuelve un objeto que contiene información detalladas sobre la edición que ha ocurrido.

- `UndoableEdit getEdit().` Devuelve un objeto **UndoableEdit** que representa la edición ocurrida y contiene información sobre los comandos para deshacer o repetir la edición.

7. ADAPTADORES.

Es tedioso tener que implementar todos los métodos de un interfaz. Por ejemplo, de un WindowListener puede interesar sólo el método de cierre, es decir el método *windowClosing*. El problema radica en que si se implementa el interfaz hay que implementar todas las funciones, aunque estén vacías. Una solución son los adaptadores.

Java proporciona ayudas para definir los métodos declarados en las interfaces ***Listener***. Una de estas ayudas son las clases ***Adapter***, que existen para cada una de las interfaces ***Listener*** que tienen más de un método. Su nombre se construye a partir del nombre de la interface, sustituyendo la palabra "*Listener*" por "*Adapter*". Algunos ejemplos de clases adapter son: *ComponentAdapter*, *ContainerAdapter*, *FocusAdapter*, *KeyAdapter*, *MouseAdapter*, *MouseMotionAdapter*, *InternalFrameAdapter* y *WindowAdapter*, …

Las clases ***Adapter*** derivan de ***Object***, y son clases predefinidas que contienen ***definiciones vacías*** para todos los métodos de la interface. Para crear un objeto que responda al evento, en vez de crear una clase que implemente la interface ***Listener***, basta crear una clase que derive de la clase ***Adapter*** correspondiente, y redefina sólo los métodos de interés. Por ejemplo, la clase ***VentanaCerrable*** se puede definir de la siguiente forma:

```
1. // Fichero VentanaCerrable.java
2. import java.awt.*;
3. import java.awt.event.*;
4. class VentanaCerrable extends Frame {
5.      // constructores
6.      public VentanaCerrable() { super(); }
7.      public VentanaCerrable(String title) {
8.            super(title);
9.            setSize(500,500);
10.           CerrarVentana cv = new CerrarVentana();
11.           this.addWindowListener(cv);
12.     }
13. } // fin de la clase VentanaCerrable
14. // definición de la clase CerrarVentana
15. class CerrarVentana extends WindowAdapter {
16.     public void windowClosing(WindowEvent we) { System.exit(0); }
17. } // fin de la clase CerrarVentana
```

Las sentencias 15-17 definen una clase auxiliar (***helper class***) que deriva de la clase ***WindowAdapter***. Dicha clase hereda definiciones vacías de todos los métodos de la interface ***WindowListener***. Lo único que tiene que hacer es redefinir el único método que se necesita para cerrar las ventanas. El constructor de la clase ***VentanaCerrable*** crea un objeto de la clase ***CerrarVentana*** en la sentencia 10 y lo registra como ***event listener*** en la sentencia 11. En la sentencia 11 la palabra ***this*** es opcional: si no se incluye, se supone que el ***event source*** es el objeto de la clase en la que se produce el evento, en este caso la propia ventana.

Todavía hay otra forma de responder al evento que se produce cuando el usuario desea cerrar la ventana. Las ***clases anónimas*** de ***Java*** son especialmente útiles en este caso. En realidad, para gestionar eventos sólo hace falta un objeto que sea registrado como ***event listener*** y contenga los métodos adecuados de la interface ***Listener***. Las ***clases anónimas*** son útiles cuando sólo se necesita un objeto de la clase, como es el caso. La nueva definición de la clase ***VentanaCerrable*** podría ser como sigue:

```
1. // Fichero VentanaCerrable3.java
2. import java.awt.*;
3. import java.awt.event.*;
4. class VentanaCerrable3 extends Frame {
5.      // constructores
6.      public VentanaCerrable3() { super(); }
7.      public VentanaCerrable3(String title) {
8.            super(title);
9.            setSize(500,500);
10.           this.addWindowListener(new WindowAdapter() {
11.                                    public void windowClosing() {System.exit(0);}
12.                                  });
13.     }
14. } // fin de la clase VentanaCerrable
```

Obsérvese que el objeto ***event listener*** se crea justamente en el momento de pasárselo como argumento al método ***addWindowListener()***. Se sabe que se está creando un nuevo objeto porque aparece la palabra ***new***. Debe tenerse en cuenta que no se está creando un nuevo objeto de ***WindowAdapter*** (entre otras cosas porque dicha clase es ***abstract***), sino extendiendo la clase ***WindowAdapter***, aunque la palabra ***extends*** no aparezca. Esto se sabe por las ***llaves*** que se abren al final de la línea 10. Los ***paréntesis vacíos*** de la línea 10 podrían contener los argumentos para el constructor de ***WindowAdapter***, en el caso de que dicho constructor necesitara argumentos. En la sentencia 11 se redefine el método ***windowClosing()***. En la línea 12 se cierran las llaves de la ***clase anónima***, se cierra el paréntesis del método ***addWindowListener()*** y se pone el ***punto y coma*** de terminación de la sentencia que empezó en la línea 10.

ANEXO I: GRÁFICOS, TEXTO, IMÁGENES Y ANIMACIONES

En esta parte final del tema se van a describir, muy sucintamente, algunas clases y métodos para realizar dibujos y añadir texto e imágenes a la interface gráfica de usuario. No obstante se puede ampliar la información con el archivo Tema_11_12_Anexo_Swing, apartado "Trabajar con gráficos".

1.- INTRODUCCIÓN AL DIBUJO

Podrías no necesitar la información de esta sección, en absoluto. Sin embargo, si tus componentes parece que no se dibujan correctamente, entender los conceptos de esta sección podría ayudarte a ver qué hay erróneo. De igual modo, necesitarás entender esta sección si creas código de dibujo personalizado para un componente.

Cómo funciona el dibujo

Cuando un GUI Swing necesita dibujarse a sí mismo -- la primera vez, o en respuesta a la vuelta de un ocultamiento, o porque necesita reflejar un cambio en el estado del programa -- empieza con el componente más alto que necesita ser redibujado y va bajando por el árbol de contenidos. Esto está orquestado por el sistema de dibujo del AWT, y se ha hecho más eficiente mediante el manejador de dibujo de Swing y el código de doble buffer.

Los componentes Swing generalmente se redibujan a sí mismos siempre que es necesario. Por ejemplo, cuando llamamos al método **setText** de un componente, el componente debería redibujarse automáticamente a sí mismo, y si es necesario, redimensionarse. Si no lo hace así es un bug (*error o defecto en el software o hardware que hace que un programa funcione incorrectamente*). El atajo es llamar al método **repaint** sobre el componente para pedir que el componente se ponga en la cola para redibujado. Si se necesita cambiar el tamaño o la posición del componente pero no automáticamente, deberíamos llamar al método **revalidate** sobre el componente antes de llamar a **repaint**.

Al igual que el código de manejo de eventos, el código de dibujo se ejecuta en el thread del despacho de eventos. Mientras se esté manejando un evento no ocurrirá ningún dibujo. De forma similar, si la operación de dibujado tarda mucho tiempo, no se manejará ningún evento durante ese tiempo.

Los programas sólo deberían dibujarse cuando el sistema de dibujo se lo diga. La razón es que cada ocurrencia de dibujo de un propio componente debe ser ejecutado sin interrupción. De otro modo, podrían ocurrir resultados imprecedibles, como que un botón fuera dibujado medio pulsado o medio liberado. Para acelerar, el dibujo Swing usa **doble-buffer** por defecto – realizado en un buffer fuera de pantalla y luego lanzado a la pantalla una vez finalizado. Podría ayudar al rendimiento si hacemos un componente Swing opaco, para que el sistema de dibujo de Swing pueda conocer lo que no tiene que pintar detrás del componente. Para hacer opaco un componente Swing, se llama al método **setOpaque(true)** sobre el componente.

Los componentes no-opacos de Swing puede parecer que tienen cualquier forma, aunque su área de dibujo disponible es siempre rectangular. Por ejemplo, un botón podría dibujarse a sí mismo dibujando un octógono relleno. El componente detrás del botón, (su contenedor, comúnmente) sería visible, a través de las esquinas de los lados del botón. El botón podría necesitar incluir código especial de detección para evitar que un evento action cuando el usuario pulsa en las esquinas del botón.

2. GRÁFICOS, TEXTO E IMÁGENES.

2.1. Capacidades gráficas del AWT.

La clase ***Component*** tiene tres métodos muy importantes relacionados con gráficos: ***paint()***, ***repaint()*** y ***update()***. Cuando el usuario llama al método ***repaint()*** de un componente, el AWT llama al método ***update()*** de ese componente, que por defecto llama al método ***paint()***.

2.1.1. *Método paint(Graphics g).*

El método ***paint()*** está definido en la clase ***Component***, pero ese método no hace nada y hay que redefinirlo en una de sus clases derivadas. El programador no tiene que preocuparse de llamar a este método: el sistema operativo lo llama al dibujar por primera vez una ventana, y luego lo vuelve a llamar cada vez que entiende que la ventana o una parte de la ventana debe ser re-dibujada (por
ejemplo, por haber estado tapada por otra ventana y quedar de nuevo a la vista).

2.1.2. *Método update(Graphics g).*

El método ***update()*** hace dos cosas: primero re-dibuja la ventana con el color de fondo y luego llama al método ***paint()***. Este método también es llamado por el AWT, y también puede ser llamado por el programador, quizás porque ha realizado algún cambio en la ventana y necesita que se dibuje de nuevo.

La propia estructura de este método -el comenzar pintando de nuevo con el color de fondo-hace que se produzca ***parpadeo*** (***flicker***) en las animaciones. Una de las formas de evitar este efecto es redefinir este método de una forma diferente, cambiando de una imagen a otra sólo lo que haya que cambiar, en vez de re-dibujar todo otra vez desde el principio. Este método no siempre propor-ciona los resultados buscados y hay que recurrrir al método del ***doble buffer***.

2.1.3. *Método repaint().*

Este es el método que con más frecuencia es llamado por el programador. El método ***repaint()*** llama "lo antes posible" al método ***update()*** del componente. Se puede también especificar un número de milisegundos para que el método ***update()*** se llame transcurrido ese tiempo. El método ***repaint()***tiene las cuatro formas siguientes:

```
repaint()
repaint(long time)
repaint(int x, int y, int w, int h)
repaint(long time, int x, int y, int w, int h)
```

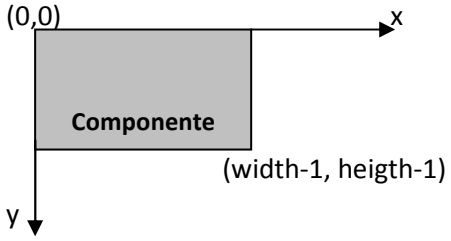
Las formas tercera y cuarta permiten definir una ***zona rectangular*** de la ventana a la que aplicar el método.

2.2. Clase Graphics.

El único argumento de los métodos ***update()*** y ***paint()*** es un objeto de esta clase. La clase ***Graphics*** dispone de métodos para soportar dos tipos de gráficos:

1. Dibujo de ***primitivas gráficas*** (*texto, líneas, círculos, rectángulos, .. …*).
2. Presentación de ***imágenes*** en formatos ****.gif*** y ****.jpeg***.

Además, la clase ***Graphics*** mantiene un ***contexto gráfico***: un área de dibujo actual, un color de dibujo del background y otro del foreground, un font con todas sus propiedades, etc. La siguiente figura muestra el sistema de coordenadas utilizado en Java. Como es habitual en Informática, tiene el origen en el vértice superior izquierdo (0,0). Las coordenadas se miden siempre en ***pixels***.



2.3. Primitivas gráficas.

Java dispone de métodos para realizar dibujos sencillos, llamados a veces "primitivas" gráficas. Las coordenadas se miden en pixels, empezando a contar desde cero. La clase ***Graphics*** dispone de los métodos para primitivas gráficas reseñados en la siguiente tabla.

Método gráfico	*Función que realizan*
`DrawLine(int x1, int y1, int x2, int y2)`	Dibuja una línea entre dos puntos
`DrawRect(int x1, int y1, int w, int h)`	Dibuja un rectángulo (w-1, h-1)
`FillRect(int x1, int y1, int w, int h)`	Dibuja un rectángulo y lo rellena con el color actual
`ClearRect(int x1, int y1, int w, int h)`	Borra dibujando con el background color
`draw3DRect(int x1, int y1, int w, int h, boolean raised)`	Dibuja un rectángulo resaltado (w+1, h+1)
`fill3DRect(int x1, int y1, int w, int h, boolean raised)`	Rellena un rectángulo resaltado (w+1, h+1)
`DrawRoundRect(int x1, int y1, int w, int h, int arcw, int arch)`	Dibuja un rectángulo redondeado
`FillRoundRect(int x1, int y1, int w, int h, int arcw, int arch)`	Rellena un rectángulo redondeado
`DrawOval(int x1, int y1, int w, int h)`	Dibuja una elipse
`FillOval(int x1, int y1, int w, int h)`	Dibuja una elipse y la rellena de un color
`DrawArc(int x1, int y1, int w, int h, int startAngle, int arcAngle)`	Dibuja un arco de elipse (ángulos en grados)
`FillArc(int x1, int y1, int w, int h, int startAngle, int arcAngle)`	Rellena un arco de elipse
`DrawPolygon(int x[], int y[], int nPoints)`	Dibuja y cierra el polígono de modo automático
`drawPolyline(int x[], int y[], int nPoints)`	Dibuja un polígono pero no lo cierra
`fillPolygon(int x[], int y[], int nPoints)`	Rellena un polígono

Excepto los polígonos y las líneas, todas las formas geométricas se determinan por el rectángulo que las comprende, cuyas dimensiones son ***w*** *(width)* y ***h*** *(height)*. Los polígonos admiten un argumento de la clase ***java.awt.Polygon***.

Los métodos ***draw3DRect()***, ***fill3DRect()***, ***drawOval()***, ***fillOval()***, **drawArc()** y ***fillArc()*** dibujan objetos cuyo tamaño total es (w+1, h+1) pixels.

2.4. Clases Graphics y Font.

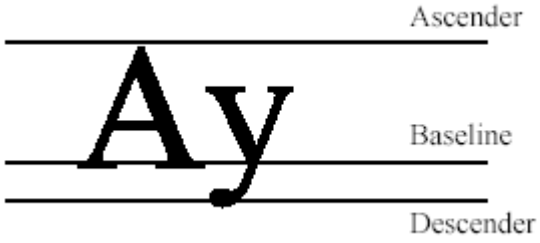
La clase ***Graphics*** permite "dibujar" texto, como alternativa al texto mostrado en los componentes ***Label***, ***TextField*** y ***TextArea***. Los métodos de esta clase para dibujar texto son los siguientes:

```
drawBytes(byte data[], int offset, int length, int x, int y);
drawChars(char data[], int offset, int length, int x, int y);
drawString(String str, int x, int y);
```

En estos métodos, los argumentos ***x*** e ***y*** representan las coordenadas de la ***línea base***. La figura siguiente muestra las líneas importantes en un tipo de letra. Cada tipo de letra está representado por un objeto de la clase ***Font***. Las clases ***Component*** y ***Graphics*** disponen de métodos ***setFont()*** y ***getFont()***. El constructor de ***Font*** tiene la forma:

```
Font(String name, int style, int size)
```

donde el ***style*** se puede definir con las constantes Font.PLAIN, Font.BOLD y Font.ITALIC. Estas constantes se pueden combinar en la forma: Font.BOLD | Font.ITALIC.



La clase ***Font*** tiene tres variables *protected*, llamadas ***name***, ***style*** y ***size***. Además tiene tres constantes enteras: PLAIN, BOLD e ITALIC. Esta clase dispone de los métodos ***String getName()***, ***int getStyle()***, ***int getSize()***, ***boolean isPlain()***,***boolean isBold()*** y ***boolean isItalic()***, cuyo significado es inmediato.

Para mayor portabilidad se recomienda utilizar nombres lógicos de fonts, tales como ***Serif*** (*Times New Roman*), ***SansSerif*** (*Arial*) y ***Monospaced*** (*Courier*).

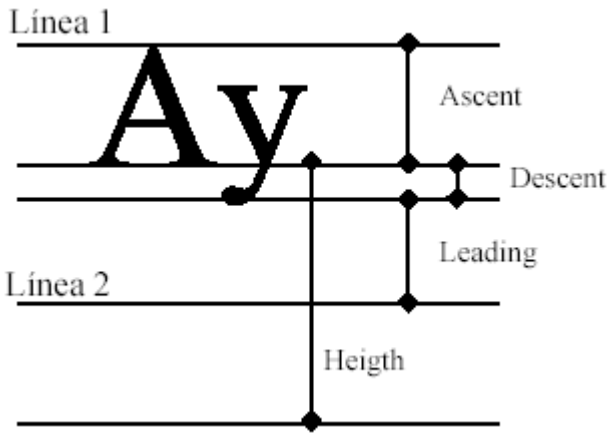
2.5. Clase FontMetrics

La clase ***FontMetrics*** permite obtener información sobre una ***font*** y sobre el espacio que ocupa un ***char*** o un ***String*** utilizando esa ***font***. Esto es muy útil cuando se pretende rotular algo de modo que quede siempre centrado y bien dimensionado.

La clase ***FontMetrics*** es una clase ***abstract***. Esto quiere decir que no se pueden crear directamente objetos de esta clase ni llamar a su constuctor. La forma habitual de soslayar esta dificultad es creando una subclase. En la práctica ***Java*** resuelve esta dificultad para el usuario, ya que la clase ***FontMetrics*** tiene como variable miembro un objeto de la clase ***Font***. Por ello, un objeto de la clase ***FontMetrics*** contiene información sobre la ***font*** que se le ha pasado como argumento al constructor. De todas formas, el camino más habitual para obtener esa información es a partir de un objeto de la clase ***Graphics*** que ya tiene un ***font*** definido. A partir de un objeto ***g*** de la clase ***Graphics*** se puede obtener una referencia ***FontMetrics*** en la forma:

```
FontMetrics miFontMet = g.getFontMetrics();
```

donde está claro que se está utilizando una referencia de la clase *abstract* ***FontMetrics*** para refererirse a un objeto de una clase derivada creada dentro del API de ***Java***. Con una referencia de tipo ***FontMetrics*** se pueden utilizar todos los métodos propios de dicha clase.



La Tabla siguiente muestra algunos métodos de la clase ***FontMetrics***, para los que se debe tener en cuenta la terminología introducida en la figura anterior.

Métodos de la clase FontMetrics	**Función que realizan**
FontMetrics(Font font)	Constructor
int getAscent(), int getMaxAscent()	Permiten obtener el "Ascent" actual y máximo para esa font
int getDescent(), int getMaxDescent()	Permiten obtener el "Descent" actual y máximo para esa font
int getHeight(), int getLeading()	Permiten ontener la distancia entre líneas y la distancia entre el descender de una línea y el ascender de la siguiente
int getMaxAdvance()	Da la máxima anchura de un carácter de esa font, incluyendo el espacio hasta el siguiente carácter
int charWidth(int ch), int charWidth(char ch), int stringWidth(String str)	Dan la anchura de un carácter (incluyendo el espacio hasta el siguiente carácter) o de toda una cadena de caracteres
int charsWidth(char data[], int start, int len), int bytesWidth(byte data[], int start, int len)	Dan la anchura de un array de caracteres o de bytes. Permiten definir la posición del comienzo y el número de caracteres

2.6. Clase Color.

La clase ***java.awt.Color*** encapsula colores utilizando el formato RGB (Red, Green, Blue). Las componentes de cada color primario en el color resultante se expresan con números enteros entre 0 y 255, siendo 0 la intensidad mínima de ese color, y 255 la máxima.

En la clase ***Color*** existen constantes para colores predeterminados de uso frecuente: ***black***, ***white***, ***green***, ***blue***, ***red***, ***yellow***, ***magenta***, ***cyan***, ***orange***, ***pink***, ***gray***, ***darkGray***, ***lightGray***. La siguiente tabla muestra algunos métodos de la clase ***Color***.

Métodos de la clase Color	***Función que realizan***
`Color(int), Color(int,int,int), Color(float,float,float)`	Constructores de Color, con enteros entre 0 y 255 y float entre 0.0 y 1.0
`Color brighter(), Color darker()`	Obtienen una versión más o menos brillante de un color
`Color getColor(), int getRGB()`	Obtiene un color en los tres primeros bytes de un int
`int getGreen(), int getRed(), int getBlue()`	Obtienen las componentes de un color
`Color getHSBColor()`	Obtiene un color a partir de los valores de "hue", "saturation" y "brightness" (entre 0.0 y 1.0)
`float[] RGBtoHSB(int,int,int,float[]), int HSBtoRGB(float,float,float)`	Métodos static para convertir colores de un sistema de definición de colores a otro

2.7. Imágenes.

Java permite incorporar imágenes de tipo GIF y JPEG definidas en ficheros. Se dispone para ello de la clase ***java.awt.Image***. Para cargar una imagen hay que indicar la localización del fichero (URL) y cargarlo mediante los métodos ***Image getImage(String)*** o ***Image getImage(URL, String)***. Estos métodos existen en las clases ***java.awt.Toolkit*** y ***java.applet.Applet***.

Cuando estas imágenes se cargan en ***applets***, para obtener el URL pueden ser útiles las funciones ***getDocumentBase()*** y ***getCodeBase()***, que devuelven el URL del fichero HTML que llama al ***applet***, y el directorio que contiene el ***applet*** (en forma de ***String***).

Para cargar una imagen hay que comenzar creando un objeto ***Image***, y llamar al método ***getImage()***, pasándole como argumento el URL. Por ejemplo:

```
Image miImagen = getImage(getCodeBase(), "imagen.gif")
```

Una vez cargada la imagen, hay que representarla, para lo cual se redefine el método ***paint()*** y se utiliza el método ***drawImage()*** de la clase ***Graphics***. Dicho método admite varias formas, aunque casi siempre hay que incluir el nombre del objeto imagen creado, las dimensiones de dicha imagen y un objeto ***ImageObserver***. Para más información sobre dicho método dirigirse a la referencia de la API.

ImageObserver es una interface que declara métodos para observar el estado de la carga y visualización de la imagen. Si se está programando un ***applet***, basta con poner como ***ImageObserver*** la referencia ***this***, ya que en la mayoría de los casos, la implementación de esta interface en la clase ***Applet*** proporciona el comportamiento deseado.

La clase ***Image*** define ciertas constantes para controlar los algoritmos de cambio de escala:
SCALE_DEFAULT, SCALE_FAST, SCALE_SMOOTH,SCALE_REPLICATE, SCALE_AVERAGE.

La siguiente tabla muestra algunos métodos de la clase ***Image***.

Métodos de la clase Image	***Función que realizan***
`Image()`	Constructor
`int getWidth(ImageObserver)` `int getHeight(ImageObserver)`	Determinan la anchura y la altura de la imagen. Si no se conocen todavía, este método devuelve -1 y el objeto ImageObserver especificado será notificado más tarde
`Graphics getGraphics()`	Crea un contexto gráfico para poder dibujar en una imagen no visible en pantalla. Este método sólo se puede llamar para métodos no visibles en pantalla
`Object getProperty(String, ImageObserver)`	Obtiene una propiedad de una imagen a partir del nombre de la propiedad
`Image getScaledInstance(int w, int h, int hints)`	Crea una versión de la imagen a otra escala. Si w o h son negativas se utiliza la otra dimensión manteniendo la proporción. El último argumento es información para el algoritmo de cambio de escala

3. ANIMACIONES.

Las animaciones tienen un gran interés desde diversos puntos de vista. Una imagen vale más que mil palabras y una imagen en movimiento es todavía mucho más útil: para presentar o describir ciertos conceptos el movimiento animado es fundamental. Además, las animaciones o mejor dicho, la forma de hacer animaciones en ***Java*** ilustran mucho la forma en que dicho lenguaje realiza los gráficos.

Se pueden hacer animaciones de una forma muy sencilla: se define el método ***paint()*** de forma que cada vez que sea llamado dibuje algo diferente de lo que ha dibujado la vez anterior. De todas formas, recuérdese que el programador no llama directamente a este método. El programador llama al método ***repaint()***, quizás dentro de un bucle ***while*** que incluya una llamada al método ***sleep()*** de la clase ***Thread***, para esperar un cierto número de milisegundos entre dibujo y dibujo (entre *frame* y *frame*, utilizando la terminología de las animaciones). Recuérdese que ***repaint()*** llama a ***update()*** lo antes posible, y que ***update()*** borra todo redibujando con el color de fondo y llama a ***paint()***.

La forma de proceder descrita da buenos resultados para animaciones muy sencillas, pero produce ***parpadeo*** o ***flicker*** cuando los gráficos son un poco más complicados. La razón está en el propio proceso descrito anteriormente, combinado con la velocidad de refresco del monitor. La velocidad de refresco vertical de un monitor suele estar entre 60 y 75 herzios. Eso quiere decir que la imagen se actualiza unas 60 ó 75 veces por segundo. Cuando el refresco se realiza después de haber borrado la imagen anterior pintando con el color de fondo y antes de que se termine de dibujar de nuevo toda la imagen, se obtiene una imagen incompleta, que sólo aparecerá terminada en uno de los siguientes pasos de refresco del monitor. Ésta es la causa del ***flicker***. A continuación se verán dos formas de reducirlo o eliminarlo.

3.1. Eliminación del parpadeo o flicker redefiniendo el método update().

El problema del ***flicker*** se localiza en la llamada al método ***update()***, que borra todo pintando con el color de fondo y después llama a ***paint()***. Una forma de resolver esta dificultad es ***re-definir*** el método ***update()***, de forma que se adapte mejor al problema que se trata de resolver.

Una posibilidad es no re-pintar todo con el color de fondo, no llamar a ***paint()*** e introducir en ***update()*** el código encargado de realizar los dibujos, cambiando sólo aquello que haya que cambiar. A pesar de esto, es necesario re-definir ***paint()*** pues es el método que se llama de forma automática cuando la ventana de ***Java*** es tapada por otra que luego se retira. Una posible solución es hacer que ***paint()*** llame a ***update()***, terminando por establecer un orden de llamadas opuesto al de defecto. Hay que tener en cuenta que, al no borrar todo pintando con el color de fondo, el programador tiene que preocuparse de borrar de forma selectiva entre *frame* y *frame* lo que sea necesario. Los métodos ***setClip()*** y ***clipRect()*** de la clase ***Graphics*** permiten hacer que las operaciones gráficas no surtan efecto fuera de un área rectangular previamente determinada. Al ser dependiente del tipo de gráficos concretos de que se trate, este método no siempre proporciona soluciones adecuadas.

3.2. Técnica del doble buffer.

La técnica del ***doble buffer*** proporciona la mejor solución para el problema de las animaciones, aunque requiere una programación algo más complicada. La idea básica del ***doble buffer*** es realizar los dibujos en una imagen invisible, distinta de la que se está viendo en la pantalla, y hacerla visible cuando se ha terminado de dibujar, de forma que aparezca instantáneamente.

Para crear el segundo buffer o imagen invisible hay que crear un objeto de la clase ***Image*** del mismo tamaño que la imagen que se está viendo y crear un contexto gráfico u objeto de la clase ***Graphics*** que permita dibujar sobre la imagen invisible. Esto se hace con las sentencias,

```
Image imgInv;
Graphics graphInv;
Dimension dimInv;
Dimension d = size(); // se obtiene la dimensión del panel
```

en la clase que controle el dibujo (por ejemplo en una clase que derive de ***Panel***). En el método ***update()*** se modifica el código de modo que primero se dibuje en la imagen invisible y luego ésta se haga visible:

```
   public void update (Graphics.g) {
     // se comprueba si existe el objeto invisible y si sus dimensiones son correctas
     if ((graphInv==null) || (d.width!=dimInv.width) || (d.height!=dimInv.height)) {
            dimInv = d;
           // se llama al método createImage de la clase Component
           imgInv = createImage(d.width, d.height);
           // se llama al método getGraphics de la clase Image
           graphInv = imgInv.getGraphics();
     }
     // se establecen las propiedades del contexto gráfico invisible,
     // y se dibuja sobre él
     graphInv.setColor(getBackground());
     ...
     // finalmente se hace visible la imagen invisible a partir del punto (0, 0)
     // utilizando el propio panel como ImageObserver
     g.drawImage(imgInv, 0, 0, this);
} // fin del método update()
```

Los gráficos y las animaciones son particularmente útiles en las applets.