# Programación Avanzada
# UD8

Autor:
P. Pablo Garrido Abenza pgarrido@umh.es

Universidad Miguel Hernández

04 de Noviembre de 2013



# ÍNDICE GENERAL

CAPÍTULO 8

# APLICACIONES GRÁFICAS



En este capítulo aprenderemos a desarrollar aplicaciones gráficas, es decir, programas con interface gráfica o *Graphical User Interface* (GUI). Existen básicamente dos librerías de clases de Java que pueden utilizarse para ello: el *Abstract Window Toolkit* (AWT) y Swing, las cuales están incluidas en la distribución estándar de Java. Existe alguna más, como la librería *Standard Widget Toolkit* (SWT) incluida en el entorno de desarrollo (IDE) Eclipse, o una extensión a Swing como la SwingX. Por último, simplemente mencionar que las aplicaciones Android utilizan el lenguaje XML para el desarrollo de las interfaces gráficas, pero todo ello sale del ámbito de estos apuntes.

En este capítulo nos vamos a centrar únicamente en las dos primeras mencionadas, AWT y Swing, pues son las más portables, ya que, como hemos comentado anteriormente, ambas vienen de serie en la distribución de Java. Ambas son muy similares de utilizar, puesto que Swing fue una extensión de la librería AWT, la cual es muy básica. Por ello, comenzaremos por explicar en primer lugar AWT, y luego comentaremos las pocas diferencias a tener en cuenta para utilizar Swing, y su mucho mayor repertorio de componentes gráficos que AWT. Ambas librerías son las que utiliza el entorno de desarrollo NetBeans para generar código cuando se utiliza su diseñador gráfico.

## 8.1. Objetivos

Los objetivos de este capítulo es el de dominar las clases y los conceptos necesarios para el desarrollo de aplicaciones gráficas utilizando las librerías AWT y Swing, en concreto:

- Conocer los contenedores y componentes, así como la distribución de componentes en los contenedores.
- Comprender el concepto de evento, y captura de eventos.
- Crear menús desplegables y contextuales.
- Saber utilizar los cuadros de diálogo predefinidos, así como crear nuestros propios diálogos.

- Insertar componentes multimedia como iconos, dibujos y sonidos.
- Conocer las diferencias entre las librerías AWT y Swing.

Como se ha dicho previamente, comenzaremos por la librería AWT, explicaremos las diferencias para poder migrar nuestra aplicaciones AWT a Swing, del cual veremos más o menos contenido en función del tiempo disponible.

## 8.2. Abstract Window Toolkit (AWT) - Introducción

Para escribir cualquier aplicación gráfica con AWT es necesario importar ciertas clases del paquete `java.awt`, y si se necesita capturar eventos, también del paquete `java.awt.event`.

```
1 import java.awt.*;
2 import java.awt.event.*;
```

El siguiente ejemplo básico simplemente abre una ventana en cuya barra de título aparece el texto `HolaMundo`; el programa no hace nada más. La idea básica consiste en crear un objeto de tipo ventana o `Frame` y hacerlo visible.

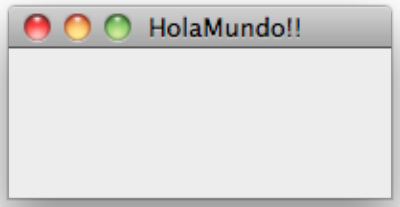
```
1  import java.awt.Frame;
2
3  public class HolaMundoAWT_v0 {
4
5     public static void main (String args[]) {
6        // Creamos objeto ventana y establecemos el titulo
7        Frame f = new Frame ("HolaMundo!!");
8
9        // Establecemos tamaño de la ventana
10       f.setSize (200,100);
11
12       // Mostramos la ventana
13       f.setVisible (true);  // show()
14    }
15 }
```

La forma de ejecutar un programa gráfico es similar a un programa en modo texto, aunque hay otra posibilidad, como se explicará en la siguiente sección 8.3.

Al ejecutar el ejemplo anterior obtendremos la ventana que se muestra a continuación. Comentar que aunque no es obligatorio establecer el tamaño de la ventana, en la práctica sí lo es, pues de no hacerlo nos aparecería una ventana sin zona de trabajo, únicamente con la barra de título, tal y como se muestra en la figura de la derecha. Por otro lado, en vez de hacer visible la ventana mediante el método `setVisible(true)`, podemos invocar a la función `show()` que, aunque es válida, su uso no se aconseja por ser una función

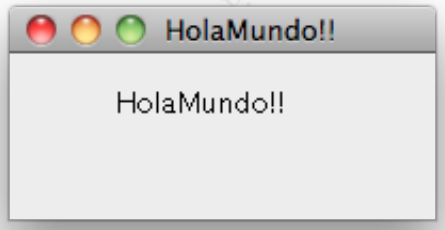
obsoleta (*deprecated*); se comenta únicamente por existir muchos ejemplos en la bibliografía que todavía podrían utilizar dicha función.





Por simplicidad, en el programa anterior no se capturan eventos, por lo que no hemos importado el paquete `java.awt.event`. Sin embargo, esto tiene un problema: no podemos cerrar la aplicación desde el botón de cerrar la ventana (botón `X` en Windows o Unix, botón rojo en Mac). Más adelante, en la sección 8.6 se explicarán los eventos y las modificaciones que tendríamos que efectuar para poder cerrar la ventana. En esta situación, la única posibilidad de cerrar la ventana sería cancelar la tarea.

A continuación modificamos el ejemplo anterior para que se muestre el mensaje `HolaMundo!!` en un lugar concreto dentro de la ventana.




Si utilizásemos la la función `print()` o `println()` que hemos utilizado hasta ahora, el texto saldría en la consola del sistema desde donde se hubiese invocado al programa, o en la zona de mensajes de nuestro entorno de desarrollo. Para que el texto se muestre en una zona concreta de la ventana es necesario sobreescribir la función `paint()` de la clase `Frame`, y utilizar las funciones de dibujo; todo ello se explicará con más detalle en la sección 8.9. Sin embargo, para poder sobreescribir dicha función `paint()` tendremos que escribir una nueva clase que herede de `Frame`. Puesto que los constructores no se heredan (tal y como se explicó en un capítulo anterior), es necesario sobreescribir al menos un constructor.

Por tanto, el programa `HolaMundo!!` quedaría como sigue; nótese que ahora, en la función `main()` ya no se crea un objeto `Frame` directamente, sino que se crea un objeto de nuestra nueva clase que hereda de `Frame`.

```
1  import java.awt.Frame;
2  import java.awt.Graphics;
3
4  public class HolaMundoAWT_v1 {
5     public static void main (String args[]) {
6        // Creamos objeto ventana y establecemos el titulo
7        MyFrame f = new MyFrame ("HolaMundo!!");
8        // Establecemos tamaño de la ventana
9        f.setSize (200,100);
10       // Mostramos la ventana
11       f.setVisible (true);
12    }
13 }
14
15 class MyFrame extends Frame {
16    // Constructor
17    public MyFrame (String titulo) {
18       super (titulo);
19    }
20
21    // Sobreescribimos funcion paint() heredada
22    public void paint (Graphics g) {
23       g.drawString ("HolaMundo!!", 50, 50);
24    }
25 }
```

En la práctica, para crear nuestras aplicaciones gráficas siempre tendremos que hacer lo que se muestra en este último ejemplo, es decir, una clase que hereda de la clase `Frame` con un constructor cuya primera línea sea siempre `super()`, y a continuación se crearían los distintos componentes gráficos (botones, listas, ...); de esto último se hablará en la sección 8.4.

## 8.3. Ejecución de un programa gráfico

Antes de continuar explicando la creación de aplicaciones gráficas vamos a explicar brevemente cómo se ejecutan las aplicaciones gráficas desde la consola del sistema. Básicamente es igual que para ejecutar aplicaciones en modo texto (con la orden `java`), aunque disponemos de otra posibilidad para ejecutarlas desde línea de órdenes (con la orden `javaw`). Ambas utilidades están incluidas en el JRE:

```
C:\ java HolaMundoAWT_v0
```

```
C:\ javaw HolaMundoAWT_v0
```

Aunque ambas utilidades reciben los mismos parámetros y realizan la misma función, invocar a la máquina virtual Java para ejecutar el programa especificado, existe una diferencia entre ellas: la orden `java` abre la ventana de la aplicación gráfica, y hasta que no se cierre, la consola del sistema quedará bloqueada; por otra parte, la orden `javaw` abrirá la ventana pero se nos devolverá el control inmediatamente, pudiendo ejecutar otras órdenes.

## 8.4. Componentes y Contenedores

Un *componente* es un objeto gráfico que puede interactuar con el usuario, como por ejemplo un botón que se pulsa, una lista de elementos que se desplaza, una casilla que se marca, o una zona de texto en la que se escribe, entre otros. Cada uno de estos componentes que se han citado son objetos de una clase concreta, pero todas ellas heredarán, de forma directa o indirecta, de la clase `Component` del paquete `java.awt`.

Por otro lado, un *contenedor* es un objeto gráfico que contiene a otros componentes, como puede ser una ventana o un cuadro de diálogo. Al igual que en el caso de los componentes, cada contenedor será un objeto de una clase concreta, y todas estas clases heredarán de la clase `Container` del mismo paquete. Si nos fijamos en la jerarquía de clases que se muestra a la derecha veremos que la clase `Container` hereda de `Component`; esto indica que los contenedores también son componentes; por ejemplo, un panel se utiliza para agrupar varios componentes y éste, a su vez, puede insertarse en otro contenedor como si fuera un componente más.

```
java.lang.Object
|
+--java.awt.Component
```

```
java.lang.Object
|
+--java.awt.Component
   |
   +--java.awt.Container
```

Como es de imaginar, el tener esta jerarquía de componentes significa que todos los métodos y constantes definidos en la clase `Component` se heredarán en las distintas subclases que se irán presentando, es decir, estarán disponibles para todos los componentes que utilicemos, más todos aquellos elementos específicos que se definan en la clase de cada uno de ellos; lo mismo ocurre con los contenedores, que heredan de la clase `Container` y `Component`.

### Contenedores

Vamos a comenzar a ver los contenedores existentes en la librería AWT. Como hemos dicho anteriormente, la clase `Container` es la clase raíz del resto de clases de contenedores, por lo que todas ellas heredan los métodos definidos en ella, que se resumen en la tabla 8.1. Las clases que realmente utilizaremos son las marcadas en color azul: `Frame`, `Dialog`, `FileDialog`, `Panel`, y `Applet`. Esta última y el concepto de *applet* se explica en un capítulo posterior.

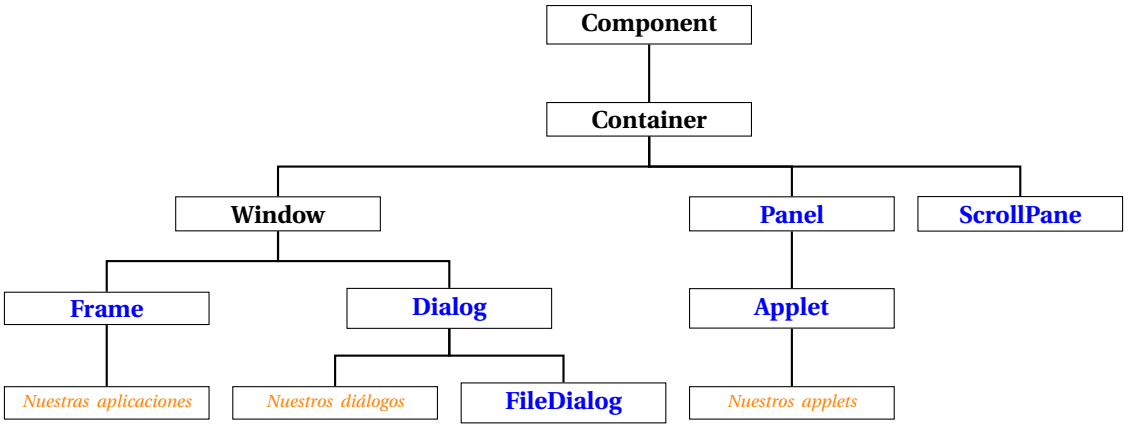



Figura 8.1: Contenedores: jerarquía de **clases**

La tabla 8.1 es un resumen de los métodos más utilizados de la clase `Container`; existen más métodos o diferentes versiones de estos, los cuales se pueden consultar en la documentación on-line de Java. De entre estos, los imprescindibles son los que se muestran aparecen en color azul. Notar que, puesto que la clase `Container` hereda de la clase `Component`, además de estos métodos también se pueden utilizar los métodos que hereda de la clase `Component`, que explicaremos en el punto 8.4.

| Método | Descripción |
|---|---|
| `add(Component c)` | Añadir el componente especificado al contenedor. |
| `getComponent(int i)` | Obtener el componente especificado del contenedor. |
| `getComponentCount()` | Obtener el número de componentes insertados en el contenedor. |
| `getLayout()` | Obtener el gestor de esquemas establecido en el componente, el cual puede establecerse con `setLayout()`. |
| `getMaximumSize()` | Obtener el tamaño máximo del contenedor, el cual puede establecerse con `setMaximumSize()` de la clase `Component`. |
| `getMinimumSize()` | Obtener el tamaño mínimo del contenedor, el cual puede establecerse con `setMinimumSize()` de la clase `Component`. |
| `remove(int i)` | Eliminar el componente especificado del contenedor. |
| `removeAll()` | Eliminar todos los componentes del contenedor. |
| `setLayout(LayoutManager m)` | Establecer el gestor de esquemas a utilizar para distribuir los componentes de forma automática al añadirlos con `add()`; ver el apartado 8.4 para más detalles. |

Cuadro 8.1: Clase `Container` - métodos

| Método | Descripción |
|---|---|
| `addWindowListener(WindowListener l)` | Registra el manejador de eventos de la ventana (minimizar, maximizar, cerrar, ...). |
| `dispose()` | Ocultar la ventana y liberar todos los recursos utilizados por ella y sus componentes. |
| `isActive()` | Averiguar si la ventana es la ventana que está activa en el sistema. |
| `isAlwaysOnTop()` | Averiguar si la ventana está marcada como siempre visible. |
| `pack()` | Redimensionar la ventana para ajustarse a los componentes que tiene según su tamaño y el gestor de esquemas establecido. |
| `setAlwaysOnTop(boolean b)` | Establecer si la ventana estará siempre visible o no. |
| `setCursor(Cursor cursor)` | Establecer el cursor, útil para mostrar un reloj de esperas, por ejemplo. |
| `setIconImage(Image image)` | Establecer el icono que se mostrará en la ventana (normalmente un `Frame`). |
| `setLocationByPlatform(boolean locationByPlatform)` | Establece si la ventana principal (`Frame`) se mostrará la próxima vez en la posición por defecto de la pantalla (`false`), o en la posición actual (`true`). |
| `setLocationRelativeTo(Component c)` | Establece la posición de la ventana centrada con relación a la ventana propietaria (útil para las ventanas secundarias o cuadros de diálogo). |
| `setVisible(boolean b)` | Mostrar o esconder la ventana. |
| `toBack()` | Si la ventana es la ventana activa, llevarla a atrás. |
| `toFront()` | Traer la ventana al frente si está visible (aunque oculta por otras ventanas), y convertirla en la ventana activa. |

Cuadro 8.2: Clase `Window` - métodos

La utilidad de la Programación Orientada a Objetos se ve cláramente al desarrollar aplicaciones gráficas, puesto que cada "cosa" que se ve en pantalla es un objeto de una clase concreta; dependiendo de la clase utilizada, los objetos tendrán un aspecto diferente, y podemos crear tantos objetos de esa clase como queramos. Los contenedores que vamos a utilizar en este capítulo son:

- `Frame`: representa a la ventana principal de una aplicación gráfica. En los ejemplos que se han mostrado al principio de este capítulo se puede comprobar su aspecto.
- `Dialog`: representa a un cuadro de diálogo o ventana secundaria abierta desde la ventana principal o desde otro `Dialog`, con objeto de mostrar algún mensaje o pedir algún valor al usuario. Un tipo específico de cuadro de diálogo es el que se muestra para pedir al usuario el nombre de un archivo a leer o grabar; puesto que esto es muy común, existe un cuadro de diálogo ya predefinido para ello: el `FileDialog`. De todo ello se hablará en el apartado 8.8.
- `Panel`: es una agrupación de componentes sin tamaño definido que, a su vez, puede insertarse en otro contenedor, facilitando la creación de las aplicaciones gráficas.
- `ScrollPane`: es un panel que permite desplazarse horizontal y verticalmente mediante unas barras de desplazamiento.



En todos ellos podemos utilizar los métodos de las clases `Component`, `Container`, y `Window` (propagación de herencia). La clase `Component` se explica en el siguiente punto. En cuanto a la clase `Window`, superclase de `Frame` y `Dialog`, comentar que no puede utilizarse por sí sola, es decir, sólo podremos crear `Frame` o `Dialog` (o subclases), pero lo que sí podremos hacer es utilizar sus métodos, que son los mostrados en la tabla 8.2.

En cuanto a la clase `Frame`, que son las ventanas principales de cualquier aplicación gráfica, comentar que ofrece algunos métodos adicionales, que podremos utilizar además de los comentados en las clases `Window`, `Container` y `Component` (propagación de herencia). En la tabla 8.3 se muestran los constructores y los métodos adicionales más habituales. En la sección 8.8 se explicará la clase `Dialog` y subclases.

| Método | Descripción |
|---|---|
| `Frame()` | Constructor por defecto. |
| `Frame(String title)` | Constructor especificando el título de la ventana. |
| `getTitle()` | Obtener el título de la ventana. |
| `isResizable()` | Averiguar si el usuario puede cambiar el tamaño de la ventana o no. |
| `setIconImage(Image image)` | Establecer el icono de la ventana (ver sección 8.9). |
| `setMenuBar(MenuBar mb)` | Establecer la barra de menús de la ventana (ver sección 8.7). |
| `setResizable(boolean resizable)` | Establecer si el usuario puede cambiar el tamaño de la ventana (`true`) o no (`false`). |
| `setTitle(String title)` | Establecer el título de la ventana. |

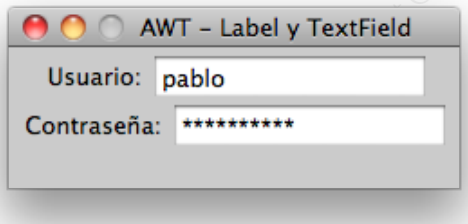
Cuadro 8.3: Clase `Frame` - constructores y métodos

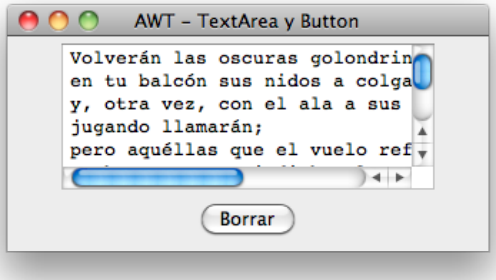
### Componentes

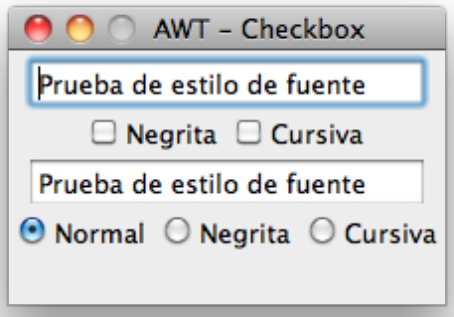
Los componentes AWT que podemos insertar en los contenedores que hemos presentado en el punto anterior son los siguientes:

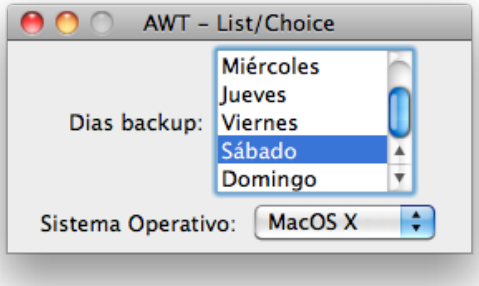
- `Label`: etiqueta de texto fijo de una sóla línea. El texto puede modificarse por el programa durante la ejecución, pero el usuario no puede escribir en el componente.
- `TextField`: zona de texto de una sóla línea editable, en la que el usuario puede escribir.
- `TextArea`: zona de texto de varias líneas, en la que el usuario puede escribir.
- `Button`: botón con una etiqueta que al ser pulsado realiza alguna acción.
- `Checkbox`: casilla que se puede marcar o desmarcar.
- `CheckboxGroup`: agrupación de objetos `Checkbox`, pudiendo el usuario marcar uno solo de los `Checkbox` que forman el grupo.
- `List`: lista desplazable de elementos de texto, pudiendo el usuario seleccionar uno o varios elementos.
- `Choice`: lista desplegable de elementos. Inicialmente, la lista sólo muestra el elemento seleccionado (único en este caso), y el usuario tiene que desplegarla para poder para poder seleccionar un elemento diferente.

El **aspecto gráfico** que presentan los objetos de las clases mencionadas se muestra en las ventanas que se muestran a continuación, cuyo código se irá mostrando a lo largo de este capítulo:



Estas clases forman parte de la **jerarquía de clases** de Java, heredando de la clase `Component`:

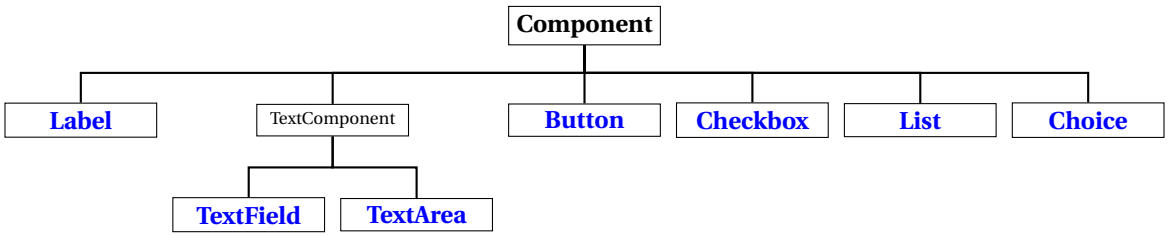



Figura 8.2: Componentes: jerarquía de **clases**

Puesto que todas estas clases heredan de la clase `Component`, pueden utilizarse todos los métodos que heredan de ella, los cuales se resumen en la tabla 8.4.

| Método | Descripción |
|---|---|
| `getHeight()` | Obtener el alto actual del componente. |
| `getWidth()` | Obtener el ancho actual del componente. |
| `paint(Graphics g)` | Método que redibuja el componente o contenedor; se debe sobreescribir por el usuario en el caso de querer modificar el comportamiento por defecto, por ejemplo, si se requiere dibujar sobre el componente. Este método es el que se invoca automáticamente por el sistema cuando se detecta que el componente necesita ser redibujado, o cuando se fuerza su refresco mediante el método `repaint()`. |
| `repaint()` | Redibujar el componente, utilizado principalmente por los contenedores. |
| `setBackground(Color c)` | Establecer el color del fondo del texto (papel). |
| `setBounds(int x, int y, int width, int height)` | Establecer la posición y tamaño del componente. |
| `setEnabled(boolean b)` | Habilitar o deshabilitar el componente, es decir, establecer o no que el usuario pueda interactuar con él. |
| `setFont(Font f)` | Establecer la fuente del texto. |
| `setForeground(Color c)` | Establecer el color del texto. |
| `setLocation(int x, int y)` | Establecer la posición del componente dentro de su contenedor. El origen es la esquina superior izquierda. |
| `setMaximumSize(Dimension d)` | Establecer el tamaño máximo del componente; puede consultarse después con `getMaximumSize()`. |
| `setMinimumSize(Dimension d)` | Establecer el tamaño mínimo del componente; puede consultarse después con `getMinimumSize()`. |
| `setSize(int width, int height)` | Establecer el tamaño actual del componente. |
| `setVisible(boolean b)` | Mostrar o esconder el componente. |

Cuadro 8.4: Clase `Component` - métodos

Además de los anteriores, existe un grupo de métodos relacionados con la captura de eventos (ver tabla 8.5). Aunque la captura de eventos se explicará en el apartado 8.6, los mostramos aquí a modo de referencia.

A continuación se explican los diferentes componentes de la librería AWT. La forma de añadirlos a un contenedor se explicará en el apartado 8.5, aunque de momento adelantamos que se realiza mediante el método `add()`.

| Método | Descripción |
|---|---|
| `addComponentListener (ComponentListener l)` | Registrar el objeto encargado de capturar ciertos eventos relacionados con el componente, como por ejemplo, al cambiar de tamaño. |
| `addFocusListener (FocusListener l)` | Registrar el objeto encargado de capturar los eventos que se generan cuando un componente pierde o gana el foco del teclado (componente en el que se escribe). |
| `addKeyListener (KeyListener l)` | Registrar el manejador de eventos de teclado (pulsar o soltar una tecla). |
| `addMouseListener (MouseListener l)` | Registrar el objeto encargado de gestionar los eventos que se generan al pulsar o soltar alguno de los botones del ratón. |
| `addMouseMotionListener (MouseMotionListener l)` | Registrar el objeto encargado de gestionar los eventos que se generan mientras se mueve o arrastra el ratón. |
| `addMouseWheelListener (MouseWheelListener l)` | Registrar el manejador de eventos cuando se mueve la rueda de desplazamiento del ratón. |

Cuadro 8.5: Clase `Component` - métodos para registrar eventos

**Componente `Label` (etiquetas)**

Un objeto `Label` consiste en una etiqueta de texto fijo de una sóla línea. El usuario no puede escribir en ella, aunque su texto puede modificarse por el programa durante la ejecución. Como con todas las clases, los distintos constructores nos permiten crear objetos `Label` de diferentes formas (ver tabla 8.6), y los métodos realizar alguna acción sobre ellos (ver tabla 8.7).

| Constructor | Descripción |
|---|---|
| Label() | Construye una etiqueta vacía; será necesario establecerle posteriormente el texto. |
| Label(String text) | Construye una etiqueta con el texto especificado, que será el que normalmente mantendrá durante toda la ejecución, y justificado a la izquierda. |
| Label(String text, int alignment) | Idem al anterior, pero especificando a qué lado se ajusta el texto con una de las constantes: `CENTER`, `LEFT`, `RIGHT`. |

Cuadro 8.6: Clase `Label` - constructores

| Método | Descripción |
|---|---|
| getAlignment() | Obtener a qué lado está justificado el texto. |
| setAlignment(int alignment) | Establecer a qué lado está justificado el texto (`CENTER`, `LEFT`, `RIGHT`). |
| getText() | Obtener el texto de la etiqueta. |
| setText(String text) | Establecer el texto de la etiqueta. |

Cuadro 8.7: Clase `Label` - métodos

Además de los constructores y los métodos mencionados, la clase `Label` dispone de las 3 constantes estáticas `CENTER`, `LEFT`, y `RIGHT`. Puesto que son estáticas, para utilizarlas es necesario escribir el nombre de la clase, por ejemplo, `Label.CENTER`.

**Componente `TextField` (cuadros de texto)**

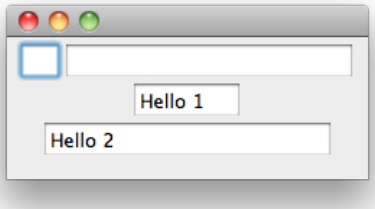
Un objeto `TextField` es una zona de texto de una sóla línea en la que el usuario puede escribir. Se puede especificar un texto inicial durante su creación, así como un ancho o número máximo de caracteres a escribir. La tabla 8.8 resume los constructores disponibles.

| Constructor | Descripción |
|---|---|
| TextField() | Construye un objeto sin texto y sin un tamaño concreto. |
| TextField(int columns) | Construye un objeto con el número especificado de columnas, es decir, el número máximo de caracteres que permite escribir. |
| TextField(String text) | Construye un objeto con el texto especificado. |
| TextField(String text, int columns) | Construye un objeto con un texto inicial, pero con capacidad para almacenar el número de caracteres (columnas) especificado. |

Cuadro 8.8: Clase `TextField` - constructores

A continuación se muestra un fragmento de código que crea varios objetos `TextField` de diferentes formas. La forma de añadir y distribuir estos componentes en la ventana se verá más adelante (sección 8.5).

```
// tf1: TextField en blanco
TextField tf1 = new TextField();
// tf2: TextField de 20 columnas
TextField tf2 = new TextField("", 20);
// tf3: TextField con un texto concreto
TextField tf3 = new TextField("Hello 1");
// tf4: TextField con un texto y 30 columnas
TextField tf4 = new TextField("Hello 2", 20);
```




| Método | Descripción |
|---|---|
| getColumns() | Obtener el número máximo de columnas o caracteres que pueden escribirse. Normalmente se estable durante la creación del componente, aunque también puede modificarse posteriormente mediante `setColumns()`. |
| getEchoChar() | Obtener el carácter que se muestra mientras se escribe; si es `'\0'` (por defecto) significa que lo que se escriba es lo que se muestra, es decir, no se ha establecido un carácter de eco con `setEchoChar()`. |
| getText() | Obtener el texto actual del componente (heredado de `TextComponent`). |
| setColumns(int n) | Establecer el número máximo de columnas o caracteres que pueden escribirse; normalmente no se utiliza ya que se establece cuando se crea el componente con el constructor adecuado. |
| setEchoChar(char c) | Establecer el carácter que se muestra mientras se escribe; si no se establece ninguno se mostrarán los caracteres tal cual. Su uso es útil cuando no se quiere mostrar lo que se escribe, como en el caso de teclear una contraseña, que se suele establer un asterisco `'*'`. Estableciendo el carácter `'\0'` se vuelve al comportamiento normal de mostrar los caracteres tal cual. |
| setText(String s) | Establecer el texto actual del componente. |

Cuadro 8.9: Clase `TextField` - métodos

**Ejemplo** A continuación se muestra un ejemplo completo que utiliza objetos `Label` y `TextField` para pedir el nombre de usuario y contraseña, ocultándola mediante un asterisco `'*'` como carácter de eco. Algunas partes del ejemplo aun no se han explicado, como la colocación de componentes en la ventana o captura de ciertos eventos para poder cerrar la ventana, por lo que, de momento, nos fijaremos únicamente en las líneas 31 a 38.

```
1  /* ============================================================
2     Fichero    : AWT_LabelTextField.java
3     Descripción: Ejemplos de componentes AWT (Label + TextField)
4     ============================================================ */
5
6  import java.awt.*;
7  import java.awt.event.*;
8
9  public class AWT_LabelTextField {
10    // Función principal
11    public static void main(String[] args) {
12       FrameAWT_LabelTextField f;
13       f = new FrameAWT_LabelTextField("AWT - Label y TextField");
14       frame.setVisible(true);
15    }
16 }
17
18 class FrameAWT_LabelTextField extends Frame {
19    Label     lblUsuario, lblPassword;
20    TextField txtUsuario, txtPassword;
21
22    // Constructor
23    FrameAWT_LabelTextField(String sTitulo) {
24       // Establecemos título (llamando al padre)
25       super(sTitulo);
26       // Establecemos un tamaño fijo para el formulario
27       setSize(250,100);
28       setResizable(false);
29       setBackground(Color.lightGray);
30
31       // Creamos los objetos gráficos
32       lblUsuario  = new Label("Usuario:"   , Label.RIGHT);
33       txtUsuario  = new TextField(15);
34       lblPassword = new Label("Contraseña:", Label.RIGHT);
35       txtPassword = new TextField(15);
36       // Establecemos como carácter eco el "*", para evitar
37       // que se pueda ver la contraseña
38       txtPassword.setEchoChar('*');
39
```

```
40          // Establecemos el gestor de esquemas por defecto
41          this.setLayout(new FlowLayout());
42          // Añadimos los objetos creados al frame
43          this.add(lblUsuario);
44          this.add(txtUsuario);
45          this.add(lblPassword);
46          this.add(txtPassword);
47
48          // Establecemos gestor de eventos para la ventana
49          addWindowListener (new WindowAdapter() {
50             public void windowClosing(WindowEvent e) {
51                System.exit(0);
52             }
53          });
54       }
55    }
56
57    // ================================================================
```

El programa anterior genera la siguiente salida:

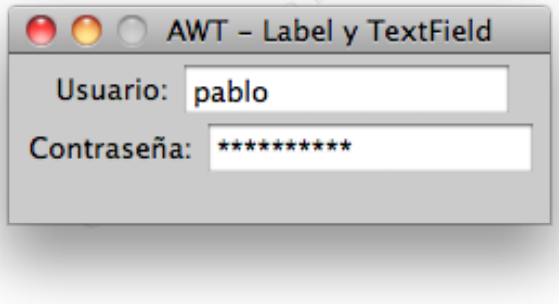



Figura 8.3: Ejemplo de componentes AWT: `Label` y `TextField`

**Componente `TextArea` (zonas de texto)**

Un `TextArea` es una zona de texto de varias líneas en la que el usuario puede escribir y moverse libremente mediante las teclas del cursor o las barras de desplazamiento opcionales (horizontal y vertical). La tabla 8.10 resume los constructores disponibles, pudiendo establecer el número de filas y columnas, su texto inicial y si queremos barras de desplazamiento o no.

| Constructor | Descripción |
|---|---|
| TextArea() | Construye un objeto sin texto y sin un tamaño concreto, incluyendo barras de desplazamiento vertical y horizontal. |
| TextArea(int rows, int cols) | Construye un objeto sin texto pero con el número especificado de filas y columnas, incluyendo barras de desplazamiento vertical y horizontal. |
| TextArea(String text) | Construye un objeto con el texto especificado, pero sin un tamaño concreto, incluyendo barras de desplazamiento vertical y horizontal. |
| TextArea(String text, int rows, int cols) | Construye un objeto con un texto inicial, y con el tamaño especificado, incluyendo barras de desplazamiento. |
| TextArea(String text, int rows, int cols, int scrollbars) | Idem al anterior, pudiendo especificar además si se quiere barra de desplazamiento horizontal (`SCROLLBARS_HORIZONTAL_ONLY`), vertical (`SCROLLBARS_VERTICAL_ONLY`), ambas (`SCROLLBARS_BOTH`), o ninguna (`SCROLLBARS_NONE`). |

Cuadro 8.10: Clase `TextArea` - constructores

Además de los constructores y los métodos mencionados, la clase `TextArea` dispone de las constantes estáticas `SCROLLBARS_HORIZONTAL_ONLY`, `SCROLLBARS_VERTICAL_ONLY`, `SCROLLBARS_BOTH` (por defecto), y `SCROLLBARS_NONE`, que permiten incluir o eliminar barras de desplazamiento en el objeto. Puesto que son estáticas, para utilizarlas es necesario escribir el nombre de la clase, por ejemplo, `TextArea.SCROLLBARS_BOTH`.

| Método | Descripción |
|---|---|
| append(String text) | Añadir el texto especificado al final del texto que actualmente tiene el componente. La forma de establecer un nuevo texto por completo es mediante `setText()` (heredado de `TextComponent`). |
| getColumns() | Obtener el número de columnas establecido. |
| getRows() | Obtener el número de filas establecido. |
| insert(String text, int pos) | Insertar el texto especificado en la posición especificada. |
| replaceRange(String text, int start, int end) | Sustituir el texto indicado entre las posiciones `start` y `end` con el texto especificado. |
| setColumns(int cols) | Establecer el número de columnas establecido. |
| setRows(int rows) | Establecer el número de filas establecido. |

Cuadro 8.11: Clase `TextArea` - métodos

Además de los métodos anteriores, podemos utilizar los métodos heredados de la clase `TextComponent`, entre ellos los métodos `getText()` y `setText(String text)`, para obtener o establecer el texto del componente, respectivamente.

| Método | Descripción |
|---|---|
| getBackground() | Obtener el color del fondo (por defecto blanco). |
| getCaretPosition() | Obtener la posición actual del cursor. |
| getSelectedText() | Obtener el texto seleccionado (manualmente por el usuario o mediante programación). |
| getSelectionEnd() | Obtener la posición final del texto seleccionado. |
| getSelectionStart() | Obtener la posición inicial del texto seleccionado. |
| getText() | Obtener el texto actual del componente. |
| isEditable() | Averiguar si el usuario puede escribir texto o ha sido bloqueado mediante `setEditable()`. |
| select(int start, int end) | Seleccionar el texto comprendido entre las posiciones `start` y `end`. |
| setBackground(Color c) | Establecer el color del fondo (por defecto blanco); ver la clase `Color` para una lista de colores. |
| setCaretPosition(int pos) | Establecer la posición actual del cursor. |
| setEditable(boolean b) | Establecer si el usuario puede escribir (`true`) o no (`false`). |
| setSelectionEnd(int pos) | Establecer la posición final del texto seleccionado. |
| setSelectionStart(int pos) | Establecer la posición inicial del texto seleccionado. |
| setText(String text) | Establecer el texto especificado en el componente, borrando cualquier texto anterior. |

Cuadro 8.12: Clase `TextComponent` - métodos

**Componente `Button` (botones)**

Un `Button` es un botón con una etiqueta que al ser pulsado realiza alguna acción. Cuando se pulsa se genera un evento denominado `ActionEvent` y se invoca automáticamente a un método denominado `actionPerformed()`, el cual recibe como argumento el evento generado. De la gestión de eventos se hablará más adelante en el apartado 8.6; en este apartado mostramos los constructores (tabla 8.13) y métodos (tabla 8.14) de la clase `Button`, obviando aquellos relacionados con la gestión de eventos.

```
1 public void actionPerformed (ActionEvent e) {
2     ...
3 }
```

| Constructor | Descripción |
|---|---|
| Button() | Construye un botón sin etiqueta, que posiblemente haya que asignar posteriormente mediante el método `setLabel()`. |
| Button(String label) | Construye un botón con la etiqueta especificada. |

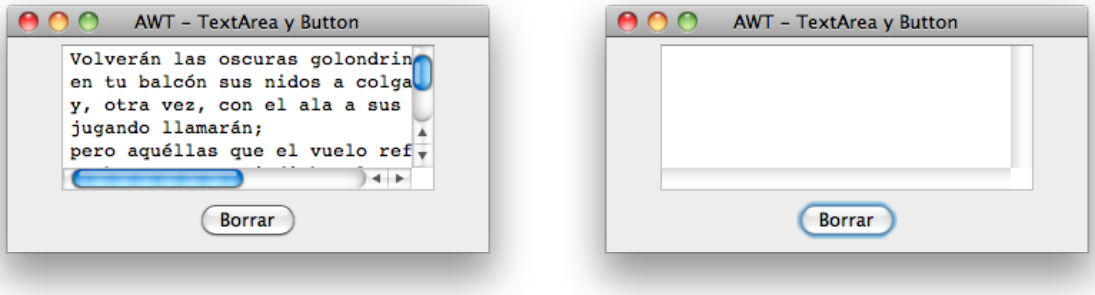
Cuadro 8.13: Clase `Button` - constructores

| Método | Descripción |
|---|---|
| getLabel() | Obtener la etiqueta del botón. |
| setLabel(String label) | Establecer una nueva etiqueta para el botón. |

Cuadro 8.14: Clase `Button` - métodos

**Ejemplo** A continuación mostramos un ejemplo de los componentes `TextArea` y `Button`; el usuario puede escribir sobre el `TextArea`, y pulsando un botón se borra el texto para volver a empezar un texto nuevo.




El código necesario para el ejemplo anterior es el siguiente.

```
1  /* ============================================================
2     Fichero    : AWT_TextAreaButton.java
3     Descripción: Ejemplos de componentes (Interfaz gráfica AWT)
4     ============================================================ */
5
6  import java.awt.*;
7  import java.awt.event.*;
8
9  public class AWT_TextAreaButton {
10    // Función principal
11    public static void main(String[] args) {
12       FrameAWT_TextAreaButton f;
13       f = new FrameAWT_TextAreaButton ("AWT - TextArea y Button");
14       f.setVisible(true);
15    }
16 }
17
18 class FrameAWT_TextAreaButton extends Frame implements ActionListener {
19    TextArea txtArea;
20    Button   pbLimpiar;
21
22    public FrameAWT_TextAreaButton(String titulo) {
23       // Establecemos título (llamando al padre)
24       super(titulo);
25       // Establecemos un tamaño fijo para la ventana
26       setSize(275,170);
27
28       // Creamos los objetos gráficos
29       txtArea = new TextArea("Texto inicial\r\n",5,25,
30                              TextArea.SCROLLBARS_BOTH);
31       pbLimpiar = new Button("Borrar");
```

```
32          // Asociamos gestor de eventos al boton
33          pbLimpiar.addActionListener(this);
34
35          // Establecemos el gestor de esquemas por defecto
36          this.setLayout(new FlowLayout());
37          // Añadimos los objetos creados al frame
38          this.add(txtArea);
39          this.add(pbLimpiar);
40
41          // Establecemos gestor de eventos para la ventana
42          addWindowListener (new WindowAdapter() {
43             public void windowClosing(WindowEvent e) {
44                System.exit(0);
45             }
46          });
47       }
48
49       // Captura de eventos
50       public void actionPerformed(ActionEvent event) {
51          if (event.getSource() == pbLimpiar) {
52             txtArea.setText("");
53          }
54       }
55    }
56
57    // ============================================================
```

**Componente `Checkbox` (casillas)**

Un `Checkbox` es una casilla que se puede marcar o desmarcar (casillas marcables). Es posible agrupar varios `Checkbox` en un objeto `CheckboxGroup`, pudiendo el usuario marcar uno solo de los `Checkbox` que forman el grupo (en este caso se dice que son casillas seleccionables). Ambos tipos de casillas son objetos `Checkbox`, salvo que en el caso de las casillas seleccionables se especifica que pertenecen a un grupo (objeto `CheckboxGroup` previamente creado).

| Constructor | Descripción |
|---|---|
| Checkbox() | Construye una casilla sin etiqueta, que posiblemente haya que asignar posteriormente mediante el método `setLabel()`. |
| Checkbox(String label) | Construye una casilla con la etiqueta especificada y desmarcada. |
| Checkbox(String label, boolean state) | Construye una casilla con la etiqueta especificada y marcada o no según el parámetro `state`. |
| Checkbox(String label, boolean state, CheckboxGroup group) | Idem al anterior, pero indicando que la casilla pertenece a un grupo, siendo por tanto, no una casilla marcable sino una casilla seleccionable. |

Cuadro 8.15: Clase `Checkbox` - constructores

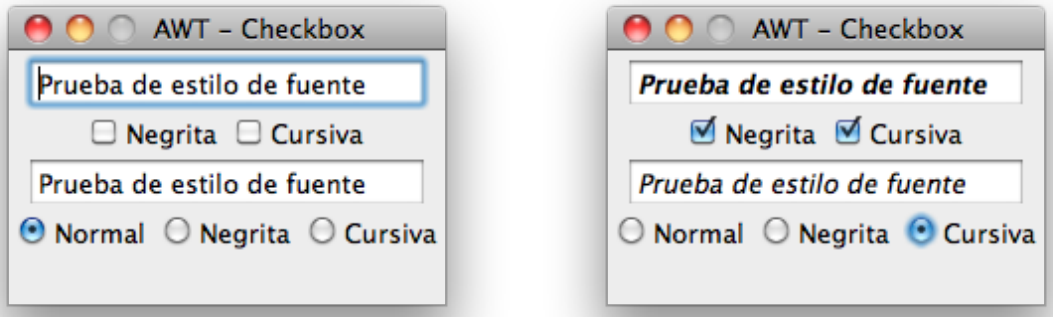
| Método | Descripción |
|---|---|
| getLabel() | Obtener la etiqueta de la casilla. |
| getState() | Averiguar si la casilla está marcada o seleccionada (`true`) o no (`false`). |
| setLabel(String label) | Establecer una nueva etiqueta para la casilla. |
| setState(boolean state) | Marcar/seleccionar (`true`) o no (`false`) la casilla. |

Cuadro 8.16: Clase `Checkbox` - métodos

En el caso de las casillas seleccionables, es decir, cuando pertenecen a un mismo grupo, sólo una de ellas puede seleccionarse, quedando el resto como no seleccionadas. Podemos averiguar cuál de ellas es la seleccionada comprobándolo mediante el método `getState()` una por una, o bien, podemos utilizar el método `getSelectedCheckbox()` de la clase `CheckboxGroup`. Del mismo modo, para seleccionar una casilla concreta, podemos utilizar el método `setState()` sobre la casilla a seleccionar, o bien, utilizar el método `setSelectedCheckbox(Checkbox box)` de la clase `CheckboxGroup`.

**Ejemplo** A continuación mostramos un ejemplo de uso de los componentes `Checkbox` y `CheckboxGroup`. Se trata de una aplicación con dos campos de texto en los que el usuario puede escribir cualquier texto. Los objetos Checkbox se utilizan para establecer un estilo de fuente de su campo de texto correspondiente. En el primero de ellos, puesto que se trata de casillas marcables (de forma independiente) podremos marcar una, dos o ninguna de las casillas, es decir, el texto podrá estar en modo negrita, en modo cursiva, ambos o texto normal. Por el contrario, puesto que en el segundo caso se trata de casillas seleccionables (agrupadas con un `CheckboxGroup`), podremos seleccionar una y sólo una opción, es decir, el texto podrá estar en modo normal, en negrita, o en cursiva, pero no en negrita y cursiva al mismo tiempo. La forma de capturar eventos y cómo establecer la fuente de letra en un campo de texto `TextField` se explicará más adelante. De momento tan sólo nos fijamos en la forma de crear los distintos objetos `Checkbox` (líneas 31 a 41).




```java
/* ============================================================
   Fichero    : AWT_Checkbox.java
   Descripción: Ejemplos de componentes AWT (Checkbox)
   ============================================================ */

import java.awt.*;
import java.awt.event.*;

public class AWT_Checkbox {
   // Función principal
   public static void main(String[] args) {
      FrameAWT_Checkbox f;
      f = new FrameAWT_Checkbox("AWT - Checkbox");
      f.setVisible(true);
   }
}

class FrameAWT_Checkbox extends Frame implements ItemListener {
   TextField texto1, texto2;
   Checkbox  checkBold, checkItalic;
   CheckboxGroup grupo;
   Checkbox      radioNormal, radioBold, radioItalic;

   public FrameAWT_Checkbox(String titulo) {
      // Establecemos título (llamando al padre)
      super(titulo);
      // Establecemos un tamaño fijo para el formulario
      setSize(220,150);
      setResizable(false);

      // Creamos los objetos gráficos
      texto1      = new TextField("Prueba de estilo de fuente", 20);
      checkBold   = new Checkbox("Negrita"); // En un solo paso
```

```
34          checkItalic = new Checkbox();           // En dos pasos: 1 ...
35          checkItalic.setLabel("Cursiva");        // ... y 2
36
37          texto2      = new TextField("Prueba de estilo de fuente", 20);
38          grupo       = new CheckboxGroup();
39          radioNormal = new Checkbox("Normal" , grupo, true);
40          radioBold   = new Checkbox("Negrita", grupo, false);
41          radioItalic = new Checkbox("Cursiva", grupo, false);
42
43          // Establecemos el gestor de esquemas por defecto
44          this.setLayout(new FlowLayout());
45          // Añadimos los objetos creados al frame
46          this.add(texto1);
47          this.add(checkBold);
48          this.add(checkItalic);
49          this.add(texto2);
50          this.add(radioNormal);
51          this.add(radioBold);
52          this.add(radioItalic);
53
54          // Asociamos gestor de eventos para las opciones
55          checkBold.addItemListener(this);
56          checkItalic.addItemListener(this);
57          radioNormal.addItemListener(this);
58          radioBold.addItemListener(this);
59          radioItalic.addItemListener(this);
60
61          // Establecemos gestor de eventos para la ventana
62          addWindowListener (new WindowAdapter() {
63             public void windowClosing(WindowEvent e) {
64                System.exit(0);
65             }
66          });
67       }
68
69       public void itemStateChanged(ItemEvent event) {
70          Font font = null;
71          int style = Font.PLAIN;
72          TextField texto = null;
73
74          if ((event.getSource() == checkBold) ||
75              (event.getSource() == checkItalic)) {
76             // Opción "Texto negrita" / "Texto cursiva" pulsada
77             style = (checkBold.getState() ? Font.BOLD : Font.PLAIN);
78             style += (checkItalic.getState() ? Font.ITALIC : Font.PLAIN);
79             texto = texto1;
```

```
80          } else if (event.getSource() == radioNormal) {
81              // Opción "Texto normal" pulsada
82              style = Font.PLAIN;
83              texto = texto2;
84          } else if (event.getSource() == radioBold) {
85              // Opción "Texto negrita" pulsada
86              style = Font.BOLD;
87              texto = texto2;
88          } else if (event.getSource() == radioItalic) {
89              // Opción "Texto cursiva" pulsada
90              style = Font.ITALIC;
91              texto = texto2;
92          }
93          // Creamos una nueva fuente idéntica a la actual,
94          // cambiando solamente el estilo
95          font = texto.getFont();
96          font = new Font(font.getFontName(), style, font.getSize());
97          texto.setFont(font);
98      }
99  }
100
101 // ============================================================
```

**Componente `List` (lista desplazable)**

Un objeto `List` es una lista desplazable de elementos de texto, pudiendo el usuario seleccionar uno o varios elementos. Si la lista tiene más elementos de los que pueden estar visibles según su tamaño aparecerán unas barras de desplazamiento horizontal y/o vertical para poder desplazarse por todos ellos.

Inicialmente la lista se crea sin elementos, éstos se añaden posteriormente de uno en uno mediante el método `add()`. Después, para averiguar el elemento que ha seleccionado el usuario se utiliza el método `getSelectedIndex()` o `getSelectedItem()`, que nos retorna el índice o el texto del elemento seleccionado, respectivamente. Hay algunos métodos más (ver tabla 8.18), pero estos que hemos comentado son los más habituales.

| Constructor | Descripción |
|---|---|
| List() | Construye una lista desplazable en blanco. |
| List(int rows) | Construye una lista desplazable asegurando un número de elementos (filas) visibles. |
| List(int rows, boolean multiple) | Idem al anterior, pudiendo activar el modo múltiple o no (por defecto desactivado). El modo múltiple permite seleccionar varios elementos de la lista (`true`). |

Cuadro 8.17: Clase `List` - constructores

| Método | Descripción |
|---|---|
| add(String item) | Añadir un nuevo elemento al final de la lista. |
| add(String item, int index) | Inserta un nuevo elemento en la posición especificada. |
| deselect(int index) | Deselecciona el elemento de la posición especificada. |
| getItem(int index) | Obtener el elemento de la posición especificada. |
| getItemCount() | Obtener el número de elementos de la lista. |
| getItems() | Obtener un array con todos los elementos. |
| getRows() | Obtener el número de elementos visibles como máximo. |
| getSelectedIndex() | Obtener el índice del elemento seleccionado. |
| getSelectedIndexes() | Obtener un array con los índices de los elementos seleccionados, en caso de estar activado el modo de selección múltiple. |
| getSelectedItem() | Obtener el texto del elemento seleccionado. |
| getSelectedItems() | Obtener un array con los textos de los elementos seleccionados, en caso de estar activado el modo de selección múltiple. |
| isIndexSelected(int index) | Averiguar si el elemento especificado está seleccionado. |
| isMultipleMode() | Averiguar si está activado el modo de selección múltiple. |
| makeVisible(int index) | Hacer que el elemento de la posición especificada esté visible. |
| remove(int index) | Eliminar el elemento de la posición especificada (índice). |
| remove(String item) | Eliminar el primer elemento que coincida con el texto especificado. |
| removeAll() | Eliminar todos los elementos de la lista. |
| replaceItem(String item, int index) | Sustituir el texto del elemento de la posición especificada de la lista. |
| select(int index) | Selecciona el elemento de la posición especificada. |
| setMultipleMode(boolean multiple) | Establecer el modo de selección múltiple (`true`) o desactivarlo (`false`); normalmente se establecerá al crear la lista y no se cambiará después. |

Cuadro 8.18: Clase `List` - métodos

**Componente `Choice` (lista desplegable)**

Una lista `Choice` es una lista desplegable de elementos muy similar a una lista desplazable de tipo `List`, salvo en su aspecto gráfico, y que sólo se puede seleccionar un único elemento, es decir, no existe modo múltiple. El siguiente fragmento crea una lista `Choice` y añade algunos elementos. Por defecto aparecerá seleccionado el primer elemento añadido; el usuario tiene que desplegar la lista para poder seleccionar un elemento diferente y, una vez seleccionado el elemento, la lista se vuelve a cerrar.

```
1 Choice lstColores = new Choice();
2 lstColores.add("Rojo");
3 lstColores.add("Verde");
4 lstColores.add("Azul");
```

| Constructor | Descripción |
|---|---|
| Choice() | Construye una lista desplegable. |

Cuadro 8.19: Clase `Choice` - constructores

| Método | Descripción |
|---|---|
| add(String item) | Añadir un nuevo elemento al final de la lista. |
| getItem(int index) | Obtener el elemento de la posición especificada. |
| getItemCount() | Obtener el número de elementos de la lista. |
| getSelectedIndex() | Obtener el índice del elemento seleccionado. |
| getSelectedItem() | Obtener el texto del elemento seleccionado. |
| insert(String item, int index) | Insertar un nuevo elemento en la posición especificada de la lista. |
| remove(int index) | Eliminar el elemento de la posición especificada (índice). |
| remove(String item) | Eliminar el primer elemento que coincida con el texto especificado. |
| removeAll() | Eliminar todos los elementos de la lista. |
| select(int index) | Selecciona el elemento de la posición especificada. |
| select(String item) | Selecciona el elemento de con el texto especifica. |

Cuadro 8.20: Clase `Choice` - métodos

**Ejemplo** Lo siguiente es un ejemplo de uso de los componentes `List` y `Choice`; el usuario puede seleccionar elementos de cada una de las dos listas. De momento nos fijaremos sólo en las líneas 30 a 46.

El código necesario para el ejemplo anterior es el siguiente.

```
/* ============================================================
   Fichero    : AWT_ListChoice.java
   Descripción: Ejemplos de componentes (Interfaz gráfica AWT)
   ============================================================ */

import java.awt.*;
import java.awt.event.*;

public class AWT_ListChoice {
   // Función principal
   public static void main(String[] args) {
      FrameAWT_ListChoice f;
      f = new FrameAWT_ListChoice ("AWT - List/Choice");
      f.setVisible(true);
   }
}

class FrameAWT_ListChoice extends Frame {
   Label  lblDias, lblOS;
   List   listaDias;
   Choice listaOS;

   public FrameAWT_ListChoice(String titulo) {
      // Establecemos título (llamando al padre)
      super(titulo);
      // Establecemos un tamaño fijo para el formulario
      setSize(280,150);
      setResizable(false);

      // Creamos los objetos gráficos
      // Lista desplazable con 5 elementos visibles y con selección múltiple
      lblDias   = new Label("Dias backup:");
      listaDias = new List(5,true);
      listaDias.add("Lunes");
      listaDias.add("Martes");
      listaDias.add("Miércoles");
      listaDias.add("Jueves");
      listaDias.add("Viernes");
      listaDias.add("Sábado");
      listaDias.add("Domingo");
      // Lista desplegable
      lblOS   = new Label("Sistema Operativo:");
      listaOS = new Choice();
      listaOS.add("Windows");
```

```
45          listaOS.add("Linux");
46          listaOS.add("MacOS X");
47
48          // Establecemos el gestor de esquemas por defecto
49          this.setLayout(new FlowLayout());
50          // Añadimos los objetos creados al frame
51          this.add(lblDias);
52          this.add(listaDias);
53          this.add(lblOS);
54          this.add(listaOS);
55
56          // Establecemos gestor de eventos para la ventana
57          addWindowListener (new WindowAdapter() {
58             public void windowClosing(WindowEvent e) {
59                System.exit(0);
60             }
61          });
62       }
63    }
64
65    // ============================================================
```

## 8.5. Colocación de componentes

En este apartado nos vamos a centrar en la forma de añadir los componentes a una ventana, aunque éstos aun no respondan a las acciones del usuario; la captura de eventos se explicará en la siguiente sección 8.6. A la hora de distribuir los distintos componentes de una ventana podemos hacerlo de forma manual o de forma automática mediante el uso de los gestores de esquemas (*layout managers*) proporcionados por el lenguaje. No obstante, se recomienda el uso de gestores de esquemas, puesto que los componentes se ajustan de forma automática, sobre todo cuando la ventana cambia de tamaño de forma dinámica.

El procedimiento general para colocar un componente en ambos casos se muestra en la figura 8.4, y se explica de forma detallada en los siguientes apartados.

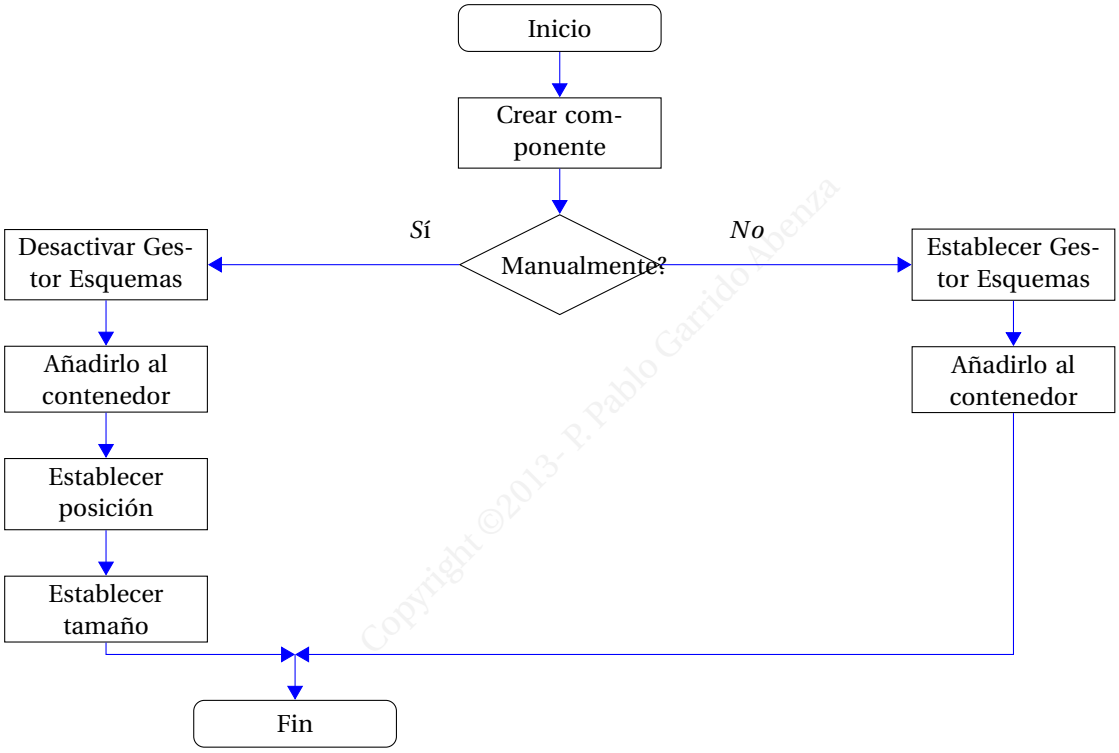



Figura 8.4: Colocación de componentes

### Distribución manual

Podemos distribuir los componentes de forma manual, utilizando posicionamiento absoluto, es decir, especificando exactamente la posición (x,y) y tamaño (en píxeles) para cada componente que añadamos al contenedor. Si nos fijamos en el ordinograma mostrado en la figura 8.4, los pasos para colocar un componente de forma manual son los siguientes:

1. Desactivar el gestor de esquemas: `setLayout(null)`.
2. Establecer la posición del componente: `setLocation(x,y)`.
3. Establecer el tamaño del componente: `setSize(width,height)`.

El primer paso es necesario porque, por defecto, los contenedores (`Frame`, `Panel`, etc.) tienen activado un gestor de esquemas concreto. Puesto que en este caso no se utilizan los gestores de esquemas (*layout managers*), es necesario desactivarlo. Este paso tan sólo es necesario realizarlo una vez, antes de insertar cualquier componente en el contenedor. Por otro lado, los dos últimos pasos podrían también realizarse en un solo paso: `setBounds(x,y,width,height)` (ver tabla 8.4). En cualquier caso, el origen de coordenadas (0,0) se considera en la esquina superior izquierda.

Como se ha comentado anteriormente, no se aconseja el uso de este método, sobre todo si el usuario puede modificar el tamaño de la ventana, puesto que es posible que se oculten algunos componentes, o se quede un espacio vacío en alguna zona. No obstante, mediante la captura de eventos `ComponentListener` podría modificarse dinámicamente la posición y tamaño de los componentes (ver apartado 8.6), pero si es necesario hacer eso, el uso de gestores de esquemas es una alternativa mucho más sencilla.

### Distribución mediante gestores de esquemas

Mediante el uso de los gestores de esquemas (*layout managers*) proporcionados por el lenguaje Java los componentes se distribuyen automaticamente, y también se ajustan su tamaño, evitando tener que trabajar con coordenadas.

Existen varios gestores de esquemas disponibles en AWT:

- `FlowLayout`: distribuye los componentes por filas, ajustando su tamaño de forma individual para que puedan mostrar toda su etiqueta; en caso de llenar una fila, se comienza una nueva si se siguen insertando componentes.
- `GridLayout`: divide el contenedor en casillas según el número de filas y columnas especificadas, todas del mismo tamaño. Los componentes se insertan en las casillas de derecha a izquierda y de arriba a abajo conforme se añaden al contenedor, siendo todos del mismo tamaño.
- `BorderLayout`: permite añadir componentes en 5 zonas: arriba, abajo, izquierda, derecha, y centro.



Existe alguno más, sobre todo si se utiliza Swing, pero con los 3 anteriores es suficiente para poder crear cualquier tipo de ventana. Dependiendo del gestor de esquema establecido en un contenedor, así se distribuirán los componentes al añadirlos con `add()`, sin necesidad de especificar ni su posición ni su tamaño. Además, otra ventaja es que se ajustarán automáticamente si el usuario amplía o comprime la ventana. Comentar que también es posible crear nuevos gestores de esquemas personalizados, aunque esto sale del ámbito de estos apuntes.

Para establecer cualquier gestor de esquemas se utiliza el método `setLayout()` (una sola vez), y luego los componentes se añaden simplemente mediante `add()` (para cada componente a añadir). El método `setLayout()` recibe on objeto de las clases anteriores (`xxxLayout`), el cual ya debe haberse creado previamente; puesto que sólo se invoca una vez a `setLayout()`, es posible hacer ambas cosas en la misma línea de código. Por ello, las dos siguientes formas de establecer un gestor de esquemas es equivalente, aunque normalmente se utiliza la de la derecha, en donde no es necesario declarar previamente el objeto.

```
1 FlowLayout layout = new FlowLayout();
2 setLayout(layout);
```

```
1 setLayout(new FlowLayout());
```

A continuación vamos a describir los tres gestores de esquema mencionados, poniendo un ejemplo en cada caso.

**FlowLayout**

Este gestor de esquemas es el más sencillo, y es el que está establecido por defecto en los objetos `Panel` (`JPanel` en Swing). Los componentes se van añadiendo al contenedor por filas, de izquierda a derecha (normalmente), con su tamaño por defecto (según su etiqueta, etc.), como cuando se escribe en un papel; si al añadir un nuevo componente éste ya no cave en la fila actual (se ha completado), entonces se añade en la siguiente fila.

Dentro de cada fila, los componentes tienen una **alineación**, por defecto de forma centrada aunque pueden ajustarse de otras formas. Normalmente esto se especifica durante su creación (ver constructores en la tabla 8.21), aunque también puede modificarse posteriormente mediante el método `setAlignment(int align)`. La tabla 8.22 resume los métodos principales, pero normalmente no se necesita utilizar ninguno de ellos, ya que se establecen los parámetros adecuados al crear el gestor de esquemas y ya no se cambia ninguno de ellos posteriormente. En cualquier caso, para establecer la alineación se utiliza una de las constantes que se enumeran a continuación (normalmente alguna de las 3 primeras):

- `FlowLayout.CENTER`: al centro (por defecto).
- `FlowLayout.LEFT`: a la izquierda.
- `FlowLayout.RIGHT`: a la derecha.
- `FlowLayout.LEADING`: a la izquierda cuando la orientación es de izquierda a derecha (por defecto); a la derecha en caso de que la orientación sea de derecha a izquierda (ver más adelante).
- `FlowLayout.TRAILING`: a la derecha cuando la orientación es de izquierda a derecha (por defecto); a la izquierda en caso de que la orientación sea de derecha a izquierda (ver más adelante).



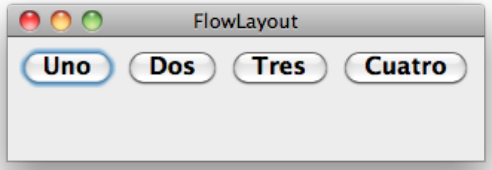
| Constructor | Descripción |
|---|---|
| FlowLayout() | Construye un gestor de esquemas alineando los componentes de cada fila al centro, y con un salto horizontal (`hgap`) y vertical (`vgap`) de 5 unidades. |
| FlowLayout(int align) | Idem al anterior pero modificando el alineamiento de los componentes en cada fila, mediante una de las constantes `FlowLayout.CENTER`, `FlowLayout.LEFT`, `FlowLayout.RIGHT`, ... |
| FlowLayout(int align, int hgap, int vgap) | Idem al anterior modificando el espacio que se deja entre cada componente dentro de una fila (`hgap`), y el salto vertical entre líneas (`vgap`), por defecto 5 unidades en ambos valores. |

Cuadro 8.21: Clase `FlowLayout` - constructores

Un ejemplo de uso del gestor de esquemas `FlowLayout` puede verse con el siguiente fragmento de código. Puede verse que no hemos tenido que trabajar con coordenadas, y que conforme se amplía la ventana los botones se reajustan de forma automática según el gestor de esquemas establecido. Puesto que hemos utilizado el constructor por defecto, la alineación es centrada en cada fila.

| Método | Descripción |
|---|---|
| getAlignment() | Obtener la alineación de los componentes dentro de cada fila. |
| getHgap() | Obtener el espacio que se deja entre cada componente dentro de una fila. |
| getVgap() | Obtener el espacio vertical que se deja entre cada fila. |
| setAlignment(int align) | Establecer la alineación de los componentes dentro de cada fila. |
| setHgap(int hgap) | Establecer el espacio que se deja entre cada componente dentro de una fila. |
| setVgap(int vgap) | Establecer el espacio vertical que se deja entre cada fila. |

Cuadro 8.22: Clase `FlowLayout` - métodos




```
1  // Establecemos gestor de esquemas
2  FlowLayout layout = new FlowLayout();
3  setLayout(layout);
4
5  // Creamos componentes
6  Button b1 = new Button("Uno");
7  Button b2 = new Button("Dos");
8  Button b3 = new Button("Tres");
9  Button b4 = new Button("Cuatro");
10
11 // Añadimos componentes al contenedor
12 add(b1);
13 add(b2);
14 add(b3);
15 add(b4);
```

Aunque no suele modificarse la **orientación**, los componentes también pueden añadirse en sentido contrario, es decir, de derecha a izquierda, modificando la propiedad `componentOrientation` del contenedor mediante el método `setComponentOrientation()` de la clase `Component`, especificando uno de los siguientes valores. Esto también es aplicable a otros gestores de esquemas (`GridLayout`, `BorderLayout`, ...), pero puesto que no se utiliza mucho ya no lo comentaremos de aquí en adelante.

- `ComponentOrientation.LEFT_TO_RIGHT` (por defecto)
- `ComponentOrientation.RIGHT_TO_LEFT`

### GridLayout

El gestor de esquemas `GridLayout` consiste en una rejilla con celdas del mismo tamaño distribuidas en varias filas y varias columnas. Es necesario, por tanto, especificar el número de filas y columnas al crear el gestor de esquemas (ver tabla 8.23). Conforme se añaden componentes a la ventana se irán completando celdas, de izquierda a derecha y de arriba a abajo, pudiendo insertar un total de *filasxcolumnas* componentes. En caso de ampliar o comprimir la ventana, todos los componentes se ajustarán al nuevo tamaño, pero siempre serán todos de igual tamaño. Al igual que para el `FlowLayout`, normalmente no se utilizan los métodos (ver tabla 8.24), ya que se especifican al crear el gestor de esquemas.

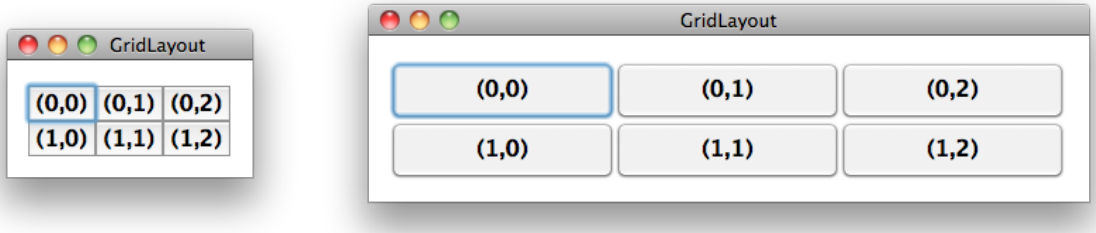
| Constructor | Descripción |
|---|---|
| GridLayout() | Construye un gestor de esquemas con una única fila y una única columna, es decir, para un único componente. |
| GridLayout(int rows, int cols) | Construye un gestor de esquemas con el número de filas y columnas especificado (este es el constructor más utilizado). |
| GridLayout(int rows, int cols, int hgap, int vgap) | Idem al anterior modificando el espacio que se deja entre cada componente dentro de una fila (`hgap`), y el salto vertical entre líneas (`vgap`), por defecto 5 unidades en ambos valores. |

Cuadro 8.23: Clase `GridLayout` - constructores

| Método | Descripción |
|---|---|
| getColumns() | Obtener el número de columnas de la rejilla. |
| getHgap() | Obtener el espacio que se deja entre cada componente dentro de una fila. |
| getRows() | Obtener el número de filas de la rejilla. |
| getVgap() | Obtener el espacio vertical que se deja entre cada fila. |
| setColumns(int cols) | Establecer el número de columnas de la rejilla. |
| setHgap(int hgap) | Establecer el espacio que se deja entre cada componente dentro de una fila. |
| setRows(int rows) | Establecer el número de filas de la rejilla. |
| setVgap(int vgap) | Establecer el espacio vertical que se deja entre cada fila. |

Cuadro 8.24: Clase `GridLayout` - métodos

Lo siguiente es un ejemplo de `GridLayout` de 2 filas y 3 columnas, con un botón en cada casilla. Por tanto, tenemos en total 6 objetos `Button`, a los que se les ha puesto como etiqueta su número de fila y columna. Puede observarse que al ampliar la ventana, todos los botones aumentan también de tamaño, distribuyéndose siempre el espacio disponible a partes iguales.



A continuación mostramos el código del ejemplo:

```
// Establecemos gestor de esquemas (2 filas, 3 columnas)
FlowLayout layout = new GridLayout(2,3);
setLayout(layout);

// Creamos componentes
Button b1 = new Button("(0,0)");
Button b2 = new Button("(0,1)");
Button b3 = new Button("(0,2)");
Button b4 = new Button("(1,0)");
Button b5 = new Button("(1,1)");
Button b6 = new Button("(1,2)");

// Añadimos componentes al contenedor
add(b1);
add(b2);
add(b3);
add(b4);
add(b5);
add(b6);
```

**BorderLayout**

Es el gestor de esquemas establecido por defecto en todos los objetos ventana (`Window`). Podemos colocar los componentes cerca de los bordes del contenedor, en 5 zonas denominadas norte, sur, este, oeste y centro. Cuando añadamos componentes con el método `add()`, debemos especificar la zona en la que queremos colocarlo mediante las cadenas ‘‘`North`’’, ‘‘`South`’’, ‘‘`East`’’, ‘‘`West`’’ y ‘‘`Center`’’, o con sus constantes equivalentes `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST` y `BorderLayout.CENTER`, respectivamente.

Los constructores (tabla 8.25) y métodos (tabla 8.26) son muy similares a los de los gestores anteriores.

| Constructor | Descripción |
|---|---|
| BorderLayout() | Construye un gestor de esquemas sin ninguna separación entre los 5 componentes. |
| BorderLayout(int hgap, int vgap) | Idem al anterior modificando el espacio que se deja entre los componentes. |

Cuadro 8.25: Clase `BorderdLayout` - constructores

| Método | Descripción |
|---|---|
| getHgap() | Obtener el espacio que se deja entre cada componente dentro de una fila. |
| getVgap() | Obtener el espacio vertical que se deja entre cada fila. |
| setHgap(int hgap) | Establecer el espacio que se deja entre cada componente dentro de una fila. |
| setVgap(int vgap) | Establecer el espacio vertical que se deja entre cada fila. |

Cuadro 8.26: Clase `BorderLayout` - métodos

A continuación vemos un ejemplo tipico que añade 5 botones en cada una de las zonas, y vemos el efecto de redimensionar la ventana. Cuando la ventana se hace más ancha, los componentes que se hacen más anchos son los del norte y el sur; cuando la ventana se hace más alta, los componentes que se amplían son los del este y el oeste. La zona central siempre se amplía o reduce convenientemente para utilizar el resto del espaqcio disponible. Comentar que no estamos obligados a asignar un elemento a todas las zonas, siendo la zona central la que se apropiará de cualquiera de las otras zonas no utilizadas.

A continuación mostramos el código del ejemplo:

```
// Establecemos gestor de esquemas
FlowLayout layout = new BorderLayout();
setLayout(layout);

// Creamos componentes
Button b1 = new Button("Norte");
Button b2 = new Button("Sur");
Button b3 = new Button("Este");
Button b4 = new Button("Oeste");
Button b5 = new Button("Centro");

// Añadimos componentes al contenedor
add(b1, BorderLayout.NORTH);
add(b2, BorderLayout.SOUTH);
add(b3, BorderLayout.EAST);
add(b4, BorderLayout.WEST);
add(b5, BorderLayout.CENTER);
```

### Paneles

Tal cual hemos explicado los gestores de esquemas hasta ahora podría parecer que tienen un uso muy limitado. Por ejemplo, el `BorderLayout` tan sólo permite insertar 5 componentes (norte, sur, este, oeste y centro), pero resulta que los contenedores también son componentes; recuerda que la clase `Container` hereda de la clase `Component`, es decir, un `Container` es-un `Component`. Existe además la clase `Panel`, que es el contenedor más simple que existe, el cual permite albergar varios componentes (lease también contenedores), y éste, a su vez, ser insertado como un único componente en otro contenedor (y así recursivamente). La jerarquía de clases que acabamos de mencionar es la siguiente:

```
java.lang.Object
|
+--java.awt.Component
   |
   +--java.awt.Container
      |
      +--java.awt.Panel
```

Como se comentó anteriormente, un `Panel` tiene como gestor de eventos por defecto el `FlowLayout`, aunque puede cambiarse mediante el método `setLayout()`. La utilidad del uso de paneles deriva en que podemos agrupar diferentes componentes distribuidos mediante un gestor de esquemas diferente en cada uno. Es decir, la ventana principal o `Frame` puede tener, por ejemplo, como gestor de esquemas un `BorderLayout`, y en cada una de las 5 zonas podemos insertar un componente aislado, o bien, un panel que incluya varios componentes distribuidos según un gestor de eventos diferente.

Como **ejemplo** de aplicación del uso de paneles vamos a desarrollar una calculadora como la que se muestra en la figura 8.5. La mayor parte de la ventana es una rejilla, por lo que podría pensarse en utilizar un `GridLayout`, pero tenemos un display en la parte superior. La solución es utilizar el gestor de esquemas `BorderLayout` para la ventana (`Frame`), colocando un componente `TextField` en el norte, y luego un panel en el centro conteniendo todos los botones; este último panel ahora sí que puede tener un `GridLayout(4,4)` como gestor de esquemas para distribuir las 4 filas por 4 columnas de botones.

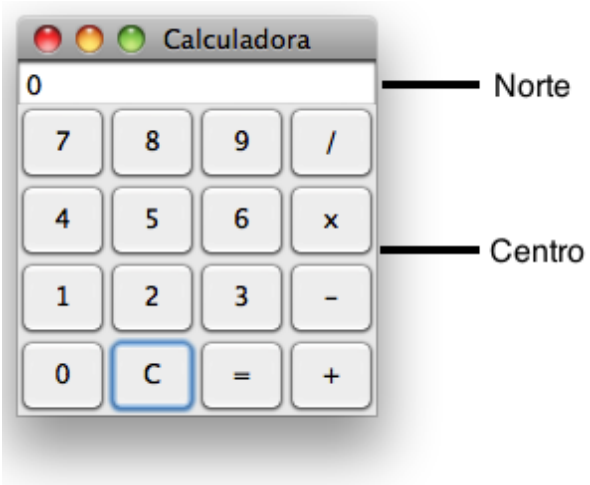



Figura 8.5: Uso de paneles: calculadora

El código necesario es el siguiente. Puesto que aun no se ha explicado la gestión de eventos, de momento tan sólo mostramos el código correspondiente a la creación del interfaz gráfico, es decir, la pulsación de cualquier botón no tendría efecto.

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class CalculadoraAWT_v1 {
5     public static void main(String Args[]) {
6        FrameCalc_v1 f = new FrameCalc_v1 ();
7        f.setSize (180,200);
8        f.setVisible(true);
9     }
10 }
11
12 // ActionListener: para los eventos de los botones
13 class FrameCalc_v1 extends Frame
14 {
15    TextField display;
16    Button    boton[][];
17
18    // Etiquetas de los botones
19    String etiquetaBoton[][] = {
20       { "7", "8", "9", "/" },
21       { "4", "5", "6", "x" },
22       { "1", "2", "3", "-" },
23       { "0", "C", "=", "+" }
24    };
25
26    public FrameCalc_v1 () {
27       super("Calculadora");
28       setLayout(new BorderLayout());
29
30       // Display
31       display = new TextField("0", 15);
32       display.setEditable(false);
33       add (display, BorderLayout.NORTH);
34
35       // Botones
36       Panel panelBotones = new Panel();
37       panelBotones.setLayout(new GridLayout(4,4));
38       add (panelBotones, BorderLayout.CENTER);
39
40       // Debemos crear cada uno de los objetos button
41       boton = new Button[4][4];
42       // Añadimos todos los botones al frame mediante un bucle.
```

```
43          // Les asociamos a todos el mismo manejador de eventos.
44          for (int i=0; i<4; i++) {
45             for (int j=0; j<4; j++) {
46                boton[i][j] = new Button(etiquetaBoton[i][j]);
47                panelBotones.add (boton[i][j]);
48             }
49          }
50
51          // Gestor de eventos de la ventana
52          addWindowListener( new WindowAdapter() {
53             public void windowClosing(WindowEvent e) {
54                System.exit(0);
55             }
56          });
57       }
58    }
```

Aspectos a destacar de este ejemplo:

- En la zona norte de la ventana, puesto que sólo hay un componente (el `TextField` correspondiente al display) no es necesario el uso de un panel (líneas 30 a 33).
- Para evitar que el usuario pueda escribir en el display lo bloqueamos haciendo que no sea editable (línea 32).
- Para insertar varios componentes en una de las zonas sí que es necesario el uso paneles, como en el caso del panel central para los botones (líneas 35 a 48).
- No se han capturado los eventos necesarios para realizar acciones al pulsar los distintos botones, lo cual se explica en el siguiente punto.



### Ajustes a los gestores de esquemas

Al utilizar gestores de esquemas, los componentes se distribuyen de forma automática, como ya se ha explicado anteriormente. Vimos también que podíamos ajustar la separación que se deja entre los componentes, ya sea horizontal o vertical. Esto se podía hacer durante la creación del gestor de esquemas (utilizando el constructor adecuado), o posteriormente mediante los métodos `setHgap()` y `setVgap()` (por defecto 5 unidades). El siguiente fragmento de código muestra cómo ajustar la separación entre los componentes:

```
1  class MiFrame extends Frame {
2     public MiFrame () {
3        ...
4        // Flowlayout, GridLayout, BorderLayout
5        FlowLayout layout = new FlowLayout ();
6        // Separacion horizonal (pixels)
7        layout.setHgap (10);
```

```
8         // Separacion vertical (pixels)
9         layout.setVgap (10);
10        // Establecemos gestor de esquemas
11        setLayout (layout);
12    }
13 }
```

Otro ajuste que puede realizarse es el margen o separación de los componentes con respecto el borde del contenedor (ventana), mediante los llamados **intercalados** o *Insets*. El siguiente fragmento ajusta el espacio de los 4 lados a 5 píxeles (arriba, izquierda, abajo, derecha). Se trata de redefinir el método `insets()` del contenedor, en el cual retorna un objeto de la clase `Insets` que se crea previamente.

```
1  lass MiFrame extends Frame {
2     ...
3     public Insets insets() {
4        // top, left, bottom, right (pixels)
5        Insets i = new Insets (5, 5, 5, 5);
6        return (i);
7     }
8  }
```

La figura 8.6 muestra uno de los ejemplos explicados anteriormente, el que utilizaba el `GridLayout` como gestor de esquemas, con las 4 posibles combinaciones de los ajustes comentados: por defecto, ampliando la separación entre componentes, aumentando el margen, y ambas cosas a la vez.


Figura 8.6: Posibles ajustes en los gestores de esquemas en un `Frame`

En los casos anteriores se mostraba como realizar estos ajustes en un `Frame`, que prácticamente siempre se hace en una subclase de `Frame`. Cuando se trabaja con paneles, que normalmente se crean objetos `Panel` directamente, se hace del mismo modo, pero si se quieren modificar los intercalados (`insets`) es necesario escribir una subclase de `Panel`, y escribir ahí el código necesario (del mismo modo que en un `Frame`). El siguiente fragmento de código muestra un ejemplo, pudiéndose modificar para otros gestores de esquemas, puesto que esto es aplicable a todos los gestores de esquemas vistos: `FlowLayout`, `GridLayout`, y `BorderLayout`.

```
1  class PanelConfigurable extends Panel {
2
3     public PanelConfigurable() {
4        super();
5        GridLayout layout = new GridLayout(4,4);
6        layout.setHgap(5);  // Separacion horizontal
7        layout.setVgap(5);  // Separacion vertical
8        setLayout (layout);
9     }
10
11    public Insets insets() {
12       // top, left, bottom, right (pixels)
13       return (new Insets(5,5,5,5));
14    }
15 }
```

La clase `PanelConfigurable` anterior podría utilizarse tal cual en el ejemplo de la calculadora visto anteriormente, puesto que los botones se colocaban en un panel distribuidos mediante un `GridLayout(4,4)`. Modificando los ajustes podríamos obtener los resultados mostrados en la figura 8.7. El código que habría que modificar en aquel ejemplo es el que se muestra a la derecha, el cual hace uso de la clase `PanelConfigurable`.

```
// Botones (sin ajustes)
Panel panelBotones = new Panel();
panelBotones.setLayout(
   new GridLayout(4,4));
add (panelBotones,
     BorderLayout.CENTER);
```

```
// Botones (con ajustes)
PanelConfigurable panelBotones =
   new PanelConfigurable();
add (panelBotones,
     BorderLayout.CENTER);
```

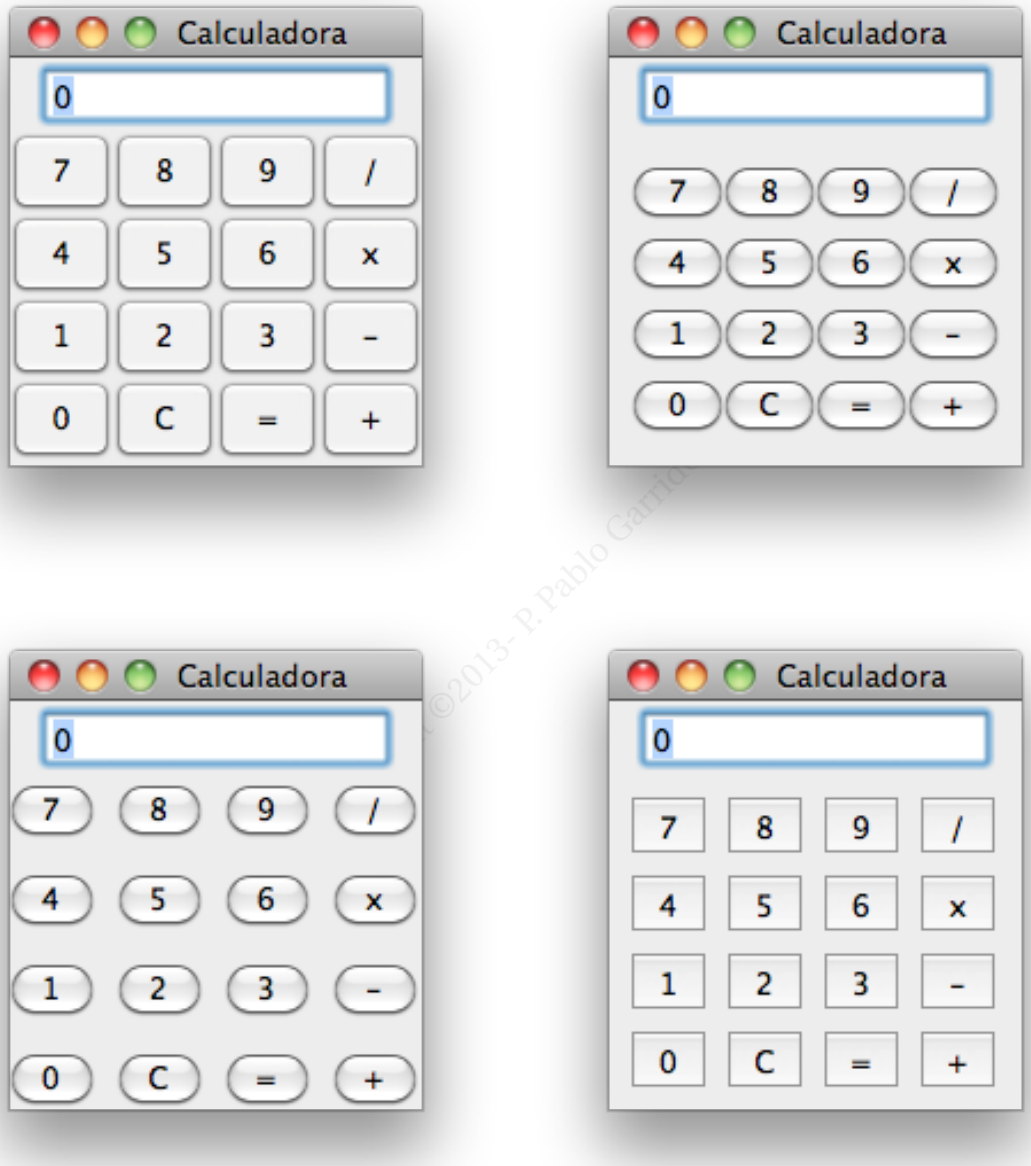



Figura 8.7: Posibles ajustes en los gestores de esquemas en un `Panel`

## 8.6. Captura de eventos

Un **evento** es el resultado de una acción del usuario sobre cualquier componente, como puede ser pulsar (hacer click) sobre un botón, pulsar una tecla, mover el ratón o pulsar uno de sus botones, cambiar de tamaño una ventana, etc. En cada uno esos casos se genera un evento, ejecutándose un método concreto según el evento generado. Por tanto, si el programador quiere que el programa responda a uno de esos eventos para realizar una acción deberá capturar el evento y escribir el código deseado en su método correspondiente.

**IMPORTANTE**: desde la versión 1.1 de Java se utiliza un modelo de **gestión de eventos delegado**, que consiste en capturar únicamente aquellos eventos que sean necesarios, para un mejor rendimiento. Sin embargo, por compatibilidad aun se mantiene el modelo de gestión de eventos anterior, el cual es diferente. El modelo antiguo no se utiliza para nuevos desarrollos, pero se comenta porque es posible encontrar ejemplos de aplicaciones gráficas en ciertos libros publicados hace unos años o referencias de Internet obsoletas que aun utilicen el modelo antiguo.

Tal como se comentó al principio de este capítulo, para desarrollar una aplicación gráfica con AWT es necesario importar ciertas clases del paquete `java.awt`, y si se necesita capturar eventos, también del paquete `java.awt.event`.

```
1 import java.awt.*;
2 import java.awt.event.*;
```

Los tipos de eventos que vamos a explicar son los siguientes, aunque hay alguno más. Para cada uno de ellos existe una interface (que hereda de la interface `EventListener`), que el programador debe implementar si desea capturar sus eventos.

- `WindowListener`: acciones sobre ventanas (abrir, cerrar, minimizar, maximizar, activar, desactivar, ...)
- `ActionListener`: acciones sobre algún componente: pulsar un botón, ...
- `ItemListener`: acciones sobre casillas o listas: marcar, desmarcar, seleccionar, deseleccionar.
- `FocusListener`: acciones cuando un componente gana o pierde el foco del teclado.
- `KeyListener`: acciones al pulsar o soltar una tecla.
- `MouseListener`: acciones cuando se pulsa o suelta alguno de los botones del ratón.
- `MouseMotionListener`: acciones cuando se mueve el ratón.
- `ComponentListener`: acciones cuando se mueve una ventana, se cambia de tamaño, ...

Cada interface declara una serie de métodos, que se corresponden con cada uno de los posibles eventos que se pueden generar, ejecutándose cada vez que se produzca el evento correspondiente. Puesto que se

trata de interfaces, el programador debe implementar todos los métodos que declare, pudiendo implementar tantos interfaces como sea necesario. La tabla 8.27 resume los métodos de los interfaces mencionados, los cuales se explicarán con más detalle en los siguientes apartados. La tercera columna indica la clase del parámetro (único) que reciben cada uno de los métodos, el cual contendrá toda la información del evento generado.

| Listener (interface) | Métodos (eventos) | Parámetro |
|---|---|---|
| `WindowListener` | windowActivated, windowClosed, windowClosing, windowDeactivated, windowDeiconified, windowIconified, windowOpened | `WindowEvent` |
| `ActionListener` | actionPerformed | `ActionEvent` |
| `ItemListener` | itemStateChanged | `ItemEvent` |
| `FocusListener` | focusGained, focusLost | `FocusEvent` |
| `KeyListener` | keyPressed, keyReleased, keyTyped | `KeyEvent` |
| `MouseListener` | mouseClicked, mouseEntered, mouseExited, mousePressed, mouseReleased | `MouseEvent` |
| `MouseMotionListener` | mouseDragged, mouseMoved | `MouseEvent` |
| `ComponentListener` | componentResized, componentMoved, componentHidden, componentShown | `ComponentEvent` |

Cuadro 8.27: Eventos: métodos de cada interface `Listener`

Además de implementar el o los interfaces necesarios, es decir, escribir código para todos los métodos de las interfaces que se necesiten, es necesario registrar el objeto manejador de eventos, es decir, decirle de algún modo al componente que va a generar el evento dónde se han escrito dichos métodos. Normalmente será en la propia clase actual, aunque también puede hacer en otra clase (clase delegada), lo cual se explicará en la sección 8.6. La forma de registrar el manejador de eventos es utilizar un método `addXXXListener()` de la clase `Component` (nombre de la interface precedido de `add`) que recibe como parámetro el objeto de la clase que implementa los métodos de la interfaz (ver tabla 8.28); si la clase que implementa los métodos de la interface es la clase actual, este objeto es el objeto actual: `this`, como aparece en la tabla 8.28, aunque podría ser cualquier otro objeto. En cualquier caso, el objeto que pasemos a estos métodos debe ser instancia de una clase que implemente dicha interface.

| Listener (interface) | Método para registrar manejador |
|---|---|
| `WindowListener` | `addWindowListener(this)` |
| `ActionListener` | `addActionListener(this)` |
| `ItemListener` | `addItemListener(this)` |
| `FocusListener` | `addFocusListener(this)` |
| `KeyListener` | `addKeyListener(this)` |
| `MouseListener` | `addMouseListener(this)` |
| `MouseMotionListener` | `addMouseMotionListener(this)` |
| `ComponentListener` | `addComponentListener(this)` |

Cuadro 8.28: Eventos: métodos para registrar manejador de eventos

En los siguientes apartados se explican estos interfaces y se muestran ejemplos completos en cada caso.

### WindowListener

La interface `WindowListener` se utiliza para gestionar las acciones sobre ventanas. Los métodos que define se describen en la tabla 8.29. Todos ellos reciben un único parámetro de la clase `WindowEvent`.

| Método | Descripción |
|---|---|
| `windowActivated()` | Invocado cuando la ventana pasa a ser la ventana activa del sistema. |
| `windowClosed()` | Invocado cuando la ventana se ha cerrado. |
| `windowClosing()` | Invocado cuando se quiere cerrar la ventana desde el botón cerrar (botón X en Windows, botón rojo en Mac OS X, ...). |
| `windowDeactivated()` | Invocado cuando la ventana deja de ser la ventana activa del sistema. |
| `windowDeiconified()` | Invocado cuando la ventana deja de estar minimizada y pasa a su estado normal. |
| `windowIconified()` | Invocado cuando se minimiza la ventana. |
| `windowOpened()` | Invocado cuando la ventana se abre y es visible por primera vez. |

Cuadro 8.29: Eventos: métodos de la interface `WindowListener`

Comentar, que al igual que con el resto de interfaces, se debe escribir el código para todos estos métodos en la clase que implemente la interface; en caso de que no interese hacer nada especial en algún evento escribiremos el método aunque sea con código vacío ({}).

**Ejemplo** En el apartado 8.2 se mostró un primer ejemplo de aplicación gráfica (HolaMundo), el cual tenía el problema de que no podía cerrarse con el botón de cerrar (no respondía a la acción del usuario). A continuación modificamos dicho ejemplo capturando los eventos de la ventana para poder cerrarla (evento `windowClosing()`; el resto de eventos no se utilizan).

```
1  import java.awt.* ;
2  import java.awt.event.*;
3
4  public class HolaMundoAWT_v2 {
5     public static void main (String args[]) {
6        // Creamos objeto ventana y establecemos el titulo
7        MyFrame_v2 f = new MyFrame_v2 ("HolaMundo!!");
8        // Establecemos tamaño de la ventana
9        f.setSize (200,100);
10       // Mostramos la ventana
11       f.setVisible (true);
12    }
13 }
14
15 class MyFrame_v2 extends Frame implements WindowListener {
16    // Constructor
17    public MyFrame_v2 (String titulo) {
18       super (titulo);
19       addWindowListener (this);
20    }
```

```
21
22      // Sobreescribimos funcion paint() heredada
23      public void paint (Graphics g) {
24         g.drawString ("HolaMundo!!", 50, 50);
25      }
26
27      public void windowClosing     (WindowEvent e) {
28         System.exit(0);  // dispose();
29      }
30      public void windowOpened      (WindowEvent e) { }
31      public void windowClosed      (WindowEvent e) { }
32      public void windowActivated   (WindowEvent e) { }
33      public void windowDeactivated (WindowEvent e) { }
34      public void windowIconified   (WindowEvent e) { }
35      public void windowDeiconified (WindowEvent e) { }
36
37  }
```

Aspectos a destacar de este ejemplo:

- La clase que hereda de `Frame` indica que implementa la interface `WindowListener` (línea 15).
- En el constructor se registra el manejador de eventos, esto es, el objeto cuya clase implementa la interface, que en este caso es `this` puesto que lo hace la clase actual (línea 19).
- Se escriben los 7 métodos de la interface `WindowListener`; aunque sólo nos interese el `windowClosing()`, es necesario escribir los otros 6, aunque sea con código vacío (líneas 27 a 35).
- Una aplicación gráfica puede terminarse como una aplicación en modo texto con `System.exit(0)` o descargando la ventana mediante `dispose()` (línea 28).

En otros ejemplos posteriores se mostraba otra forma de cerrar la ventana más compacta, aunque no tan fácil de entender; aquella forma se explicará más adelante, cuando se expliquen clases adaptadoras y clases anónimas.

### ActionListener

Los eventos de tipo `ActionListener` se utilizan para capturar acciones sobre algún componente; el ejemplo típico es cuando el usuario pulsa un botón (objeto `Button`), aunque pueden aplicarse a otros componentes. Esta interface tan sólo define un método (ver tabla 8.30), el cual recibe un único parámetro de la clase `ActionEvent`.

| Método | Descripción |
|---|---|
| `actionPerformed()` | Invocado cuando ocurre una acción sobre el componente. |

Cuadro 8.30: Eventos: métodos de la interface `ActionListener`

**Ejemplo** En el apartado 8.4 se mostró un ejemplo que hacía uso de un botón (objeto `Button`) para borrar el texto escrito en una zona de texto (objeto `TextArea`). El ejemplo estaba completo, pero no se explicó la parte de la gestión de eventos. Los aspectos a destacar del citado ejemplo son:

- La clase que hereda de `Frame` indica que implementa la interface `ActionListener` (línea 18).
- En el constructor se le registra el manejador de eventos al objeto `Button`, pasándole el objeto cuya clase implementa la interface, que en este caso es `this` puesto que lo hace la clase actual (línea 33).
- Se escribe el método de la interface `ActionListener`, que en este caso el único que es necesario es el método `actionPerformed()` (líneas 50 a 54).
- En el caso de que hubiesen varios botones, la pulsación de cualquiera de ellos generaría un evento y se invocaría el método `actionPerformed()`; para averiguar cuál de los componentes ha sido el que ha generado el evento, podemos utilizar el método `getSource()` sobre el parámetro, de tipo `ActionEvent` en este caso (línea 51).

Lo siguiente es un fragmento del código de aquel ejemplo en relación con lo que se comenta; para ver el código completo ir al apartado 8.4:

```
class FrameAWT_TextAreaButton extends Frame implements ActionListener {
   public FrameAWT_TextAreaButton(String titulo) {
      pbLimpiar = new Button("Borrar");
      pbLimpiar.addActionListener(this);
      ...
   }
   // Captura de eventos
   public void actionPerformed(ActionEvent event) {
      if (event.getSource() == pbLimpiar) {
         txtArea.setText("");
      }
   }
}
```

**Ejemplo 2 - calculadora** En la sección 8.5 se mostró como ejemplo de distribución de componentes el de una calculadora. En aquel ejemplo tan sólo nos centramos en la construcción de los componentes gráficos, pero sin escribir el código necesario para la captura de eventos de los distintos botones. A continuación ampliamos ese mismo ejemplo para que la calculadora sea completamente funcional.

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class CalculadoraAWT_v1 {
5     public static void main(String Args[]) {
6        FrameCalc_v1 f = new FrameCalc_v1 ();
7        f.setSize (180,200);
8        f.setVisible(true);
9     }
10 }
11
12 // ActionListener: para los eventos de los botones
13 class FrameCalc_v1 extends Frame implements ActionListener   // (*)
14 {
15    TextField display;
16    Button    boton[][];
17
18    // Etiquetas de los botones
19    String etiquetaBoton[][] = {
20       { "7", "8", "9", "/" },
21       { "4", "5", "6", "x" },
22       { "1", "2", "3", "-" },
23       { "0", "C", "=", "+" }
24    };
25
26    public FrameCalc_v1 () {
27       super("Calculadora");
28       setLayout(new BorderLayout());
29
30       // Display
31       display = new TextField("0", 15);
32       display.setEditable(false);
33       add (display, BorderLayout.NORTH);
34
35       // Botones
36       Panel panelBotones = new Panel();
37       panelBotones.setLayout(new GridLayout(4,4));
38       add (panelBotones, BorderLayout.CENTER);
39
40       // Debemos crear cada uno de los objetos button
41       boton = new Button[4][4];
```

```
42        // Añadimos todos los botones al frame mediante un bucle.
43        // Les asociamos a todos el mismo manejador de eventos.
44        for (int i=0; i<4; i++) {
45           for (int j=0; j<4; j++) {
46              boton[i][j] = new Button(etiquetaBoton[i][j]);
47              panelBotones.add (boton[i][j]);
48              boton[i][j].addActionListener(this);           // (*)
49           }
50        }
51
52        // Gestor de eventos de la ventana
53        addWindowListener( new WindowAdapter() {
54           public void windowClosing(WindowEvent e) {
55              System.exit(0);
56           }
57        });
58     }
59
60     public void actionPerformed (ActionEvent event) {       // (*)
61        // Todos los eventos seran producidos por un boton
62        Button boton = (Button) event.getSource();
63        String etiqueta = boton.getLabel();
64
65        if ( etiqueta.equals("0") || etiqueta.equals("1") ||
66             etiqueta.equals("2") || etiqueta.equals("3") ||
67             etiqueta.equals("4") || etiqueta.equals("5") ||
68             etiqueta.equals("6") || etiqueta.equals("7") ||
69             etiqueta.equals("8") || etiqueta.equals("9") ) {
70           // Hemos pulsado un digito: lo Añadimos al display
71           añadirDigito (etiqueta);              // [0] .. [9]
72        } else if ( etiqueta.equals("+") || etiqueta.equals("-") ||
73                    etiqueta.equals("x") || etiqueta.equals("/") ) {
74           // Hemos pulsado una operacion: memorizamos el primer valor y la
                operacion
75           pulsarOperacion (etiqueta);           // [+], [-], [x], [/]
76        } else if ( etiqueta.equals("=") ) {
77           // Hemos pulsado el "=": realizamos la operacion selecciona
                anteriormente
78           if ( op!='\0' ) realizarOperacion(); // [=]
79        } else if ( etiqueta.equals("C") ) {
80           // Inicializamos la calculadora
81           inicializar();                        // [C]
82        }
83     }
84
85
```

```
86      /* Inicializamos la calculadora */
87      private void inicializar() {
88         v1 = v2 = 0;
89         op = '\0';
90         display.setText ( "0" );
91      }
92
93      /* Añadimos un digito al final del valor del display */
94      private void añadirDigito(String digito) {
95         String str = display.getText();
96
97         // Si en el display tenemos "0" o "E" lo reemplazamos
98         if ( str.equals("0") || str.equals("E") ) str = "";
99         display.setText ( str + digito );
100     }
101
102     /* Acciones a realizar cuando se pulsa un Botón de operacion: +, -, *, / */
103     private void pulsarOperacion (String opPulsada) {
104        try {
105           v1 = Integer.parseInt(display.getText());
106           op = opPulsada.charAt(0);
107           display.setText ( "0" );
108        } catch (NumberFormatException e) {
109           v1 = 0;
110           op = '\0';
111           display.setText ( "E" );
112        }
113     }
114
115     /* Realizamos la operacion al pulsar '='
116      * (solo si antes hemos pulsado algun boton de operacion) */
117     private void realizarOperacion() {
118        try {
119           v2 = Integer.parseInt(display.getText());
120           if ( op == '+' ) {
121              res = v1 + v2;
122           } else if ( op == '-' ) {
123              res = v1 - v2;
124           } else if ( op == 'x' ) {
125              res = v1 * v2;
126           } else if ( op == '/' ) {
127              // OJO: Se puede producir una division por cero si v2==0.
128              // Se debe controlar con if o con try..catch.
129              if ( v2 == 0 ) {
130                 display.setText ( "E" );
131                 return;
```

```
132                 } else {
133                     res = v1 / v2;
134                 }
135             }
136             display.setText ( ""+res );
137         } catch (NumberFormatException e) {
138             display.setText ( "E" );
139         }
140         v1 = v2 = 0;
141         op = '\0';
142     }
143 }
```

En el código se ha marcado mediante un comentario con `(*)` los cambios necesarios, que son:

- Indicar que la clase implementará la interface `ActionListener` (línea 13).
- Implementar sus métodos: `actionPerformed()` (líneas 60 a 83).
- Registrar el gestor de eventos de cada botón: `addActionListener()` (línea 48).

Respecto al método `actionPerformed()`, comentar que se utiliza el método `getSource()` para obtener una referencia al botón pulsado, y con esta referencia podríamos ir comparando con los distintos objetos `Button` creados (array). Sería algo así:

```
public void actionPerformed (ActionEvent event) {
    Button b = (Button) event.getSource();

    if (b == boton[0][0]) {          // [7]
        añadirDigito ("7");
    } else if (b == boton[0][1]) {   // [8]
        añadirDigito ("8");
    } else ...
}
```

Sin embargo, en el ejemplo se ha utilizado la etiqueta de los botones para averiguar el botón que se ha pulsado (líneas 60 a 83), ya que parece más sencillo de esa forma.

### ItemListener

Los eventos `ItemListener` se generan al marcar o desmarcar casillas (`Checkbox`), y al seleccionar o deseleccionar elementos de una lista (`List`). En cualquier caso se genera un único evento que se muestra en la tabla 8.31, y que recibe un parámetro de la clase `ItemEvent`.

| Método | Descripción |
|---|---|
| `itemStateChanged()` | Invocado cuando se marca (o desmarca) una casilla, o cuando se selecciona (o deselecciona) un elemento de la lista. |

Cuadro 8.31: Eventos: métodos de la interface `ItemListener`

**Ejemplo** Como ejemplo se puede revisar el código del ejemplo mostrado en la sección 8.4, el cual era totalmente funcional. Se trataba de poder establecer la fuente de letra de un texto marcando o desmarcando casillas (`Checkbox`), y seleccionando casillas (`Checkbox` agrupados en un `CheckboxGroup`).

### FocusListener

Los eventos `FocusListener` se generan cuando un componente gana o pierde el foco del teclado, es decir, el componente en el que se escribe. Este interface declara los 2 métodos que se declaran en la tabla 8.32, los cuales reciben un argumento de la clase `FocusEvent`.

| Método | Descripción |
|---|---|
| `focusGained()` | Invocado cuando el componente obtiene el foco del teclado. |
| `focusLost()` | Invocado cuando el componente pierde el foco del teclado. |

Cuadro 8.32: Eventos: métodos de la interface `FocusListener`

**Ejemplo** No se muestra ningún ejemplo en este apartado. Si se domina cualquiera de las interfaces anteriores para la gestión de eventos, el alumno no tendrá problema en saber utilizar el mismo procedimiento para estos eventos y cualquiera de los que quedan por ver.

### KeyListener

Los eventos `KeyListener` son la respuesta cuando se pulsa o suelta una tecla del teclado. Este interface declara los 3 métodos que se declaran en la tabla 8.33, los cuales reciben un argumento de la clase `KeyEvent`.

| Método | Descripción |
|---|---|
| `keyPressed()` | Invocado cuando se pulsa una tecla. |
| `keyReleased()` | Invocado cuando se suelta una tecla. |
| `keyTyped ()` | Invocado cuando se pulsa y suelta una tecla. |

Cuadro 8.33: Eventos: métodos de la interface `KeyListener`

### MouseListener

Los eventos `MouseListener` se generan cuando se pulsa o suelta alguno de los botones del ratón. Este interface declara los 3 métodos que se declaran en la tabla 8.34, los cuales reciben un argumento de la clase `MouseEvent`.

| Método | Descripción |
|---|---|
| `mouseClicked()` | Invocado cuando se pulsa y suelta un botón de ratón. |
| `mouseEntered()` | Invocado cuando el ratón entra en un componente. |
| `mouseExited ()` | Invocado cuando el ratón sale de un componente. |
| `mousePressed()` | Invocado cuando se pulsa un botón de ratón. |
| `mouseReleased()` | Invocado cuando se suelta un botón de ratón. |

Cuadro 8.34: Eventos: métodos de la interface `MouseListener`

### MouseMotionListener

Los eventos `MouseMotionListener` se generan siempre que se esté moviendo el ratón, con o sin algún botón pulsado. Este interface declara los 2 métodos que se declaran en la tabla 8.35, los cuales reciben un argumento de la clase `MouseEvent` (igual que el interface `MouseListener`).

| Método | Descripción |
|---|---|
| `mouseDragged()` | Invocado cuando se mueve el ratón con uno de los botones pulsados (arrastrar). |
| `mouseMoved()` | Invocado cuando se mueve el ratón sin mantener pulsado ningún botón. |

Cuadro 8.35: Eventos: métodos de la interface `MouseMotionListener`



### ComponentListener

Los eventos `ComponentListener` son útiles cuando se quiere realizar alguna acción cuando se mueve una ventana, o cambia de tamaño, ... Aunque se pueden aplicar a cualquier componente, normalmente se utilizan con `Frame` o `Dialog`, que son contenedores (recuerda que la clase `Container` hereda de `Component`). Este interface declara los 4 métodos que se declaran en la tabla 8.36, los cuales reciben un argumento de la clase `ComponentEvent`.

| Método | Descripción |
|---|---|
| `componentHidden()` | Invocado cada vez que el componente deja de estar visible. |
| `componentMoved()` | Invocado cuando el componente (lease también contenedor) se mueve con el ratón. |
| `componentResized()` | Invocado cuando el componente cambia de tamaño (redimensionando con el ratón). |
| `componentShown()` | Invocado cada vez que el componente se hace visible. |

Cuadro 8.36: Eventos: métodos de la interface `ComponentListener`

**Ejemplo** El siguiente ejemplo consiste en una ventana gráfica (`Frame`) tal que al moverla o cambiar su tamaño se escribe en la ventana su posición (x,y) y su tamaño (anchoxalto), tal como se muestra en la imagen.


```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class AWT_ComponentListener {
5     public static void main (String args[]) {
6        // Creamos objeto ventana y establecemos el titulo
7        AWT_ComponentListener_Frame f =
8              new AWT_ComponentListener_Frame ("Ejemplo ComponentListener");
9        // Establecemos tamaño de la ventana
10       f.setSize (200,100) ;
11       // Mostramos la ventana
12       f.setVisible (true);
13    }
14 }
15
16 class AWT_ComponentListener_Frame extends Frame
17                       implements WindowListener, ComponentListener {
18    // Constructor
19    public AWT_ComponentListener_Frame (String titulo) {
20       super (titulo);
21       addWindowListener (this);
22       addComponentListener (this);
23    }
24
25    // --- Interface WindowListener -----------------
26    public void windowClosing (WindowEvent e) {
27       System.exit (0);    // dispose();
28    }
29    public void windowOpened      (WindowEvent e) {}
30    public void windowClosed      (WindowEvent e) {}
31    public void windowActivated   (WindowEvent e) {}
32    public void windowDeactivated (WindowEvent e) {}
33    public void windowIconified   (WindowEvent e) {}
34    public void windowDeiconified (WindowEvent e) {}
35    // ----------------------------------------------
36
```

```
37      // --- Interface ComponentListener --------------
38      // Invocado cada vez que el componente deja de estar visible
39      public void componentHidden (ComponentEvent e) {}
40
41      // Invocado cuando el componente se mueve con el ratón
42      public void componentMoved (ComponentEvent e) {
43         //System.out.println("Posición: "+getX()+","+getY());
44         repaint();
45      }
46
47      // Invocado cuando el componente cambia de tamaño
48      public void componentResized (ComponentEvent e) {
49         //System.out.println("Tamaño : "+getWidth()+"x"+getHeight());
50         repaint();
51      }
52
53      // Invocado cada vez que el componente se hace visible
54      public void componentShown (ComponentEvent e) {}
55      // ------------------------------------------------
56
57      // Sobreescribimos funcion paint() heredada
58      public void paint (Graphics g) {
59         g.drawString ("Posición: "+getX()+","+getY(), 30, 50);
60         g.drawString ("Tamaño : "+getWidth()+"x"+getHeight(), 30, 70);
61      }
62  }
```

De este ejemplo es necesario explicar lo siguiente:

- Los eventos capturados de la interface `ComponentListener` prácticamente no hacen nada; simplemente, cuando se genera el evento invoca a `repaint()` (líneas 44 y 50), el cual fuerza a que se ejecute el método `paint()`, y éste, finalmente ya dibuja en la ventana los textos de posición y tamaño. En el código también se han escrito sentencias `println()`, pero están comentadas (líneas 43 y 49); si se descomentan, la salida de dichas sentencias será la ventana de la consola (o la zona de mensajes del entorno de desarrollo).

- La función `paint()` se invoca de forma automática cuando el sistema detecta que la ventana debe redibujarse, o bien, cuando se lo pedimos expresamente mediante `repaint()`. Esto se explicará en el apartado 8.9

- Dentro de la función `paint()` se escriben sentencias de dibujado, como `drawString()`, que se encarga de dibujar texto por pantalla, similar al primer ejemplo HolaMundo que se vio en este capítulo (líneas 58 a 61).

### Clases delegadas

Hasta ahora, en todos los ejemplos que se han presentado, era la clase que heredaba de `Frame` la que implementaba todos los métodos de todas las interfaces necesarias. Esto puede dar lugar a que dicha clase quede demasiado abarrotada.

Un solución a esto es el uso de clases delegadas, en las que cada una se encarga de implementar los eventos de uno o más interfaces de eventos, con lo que dividimos la tarea en varios fragmentos. Una vez escritas las clases delegadas ya podemos utilizarlas, aunque la forma de registrar el objeto manejador de eventos es diferente: si antes pasábamos la referencia al objeto actual (`this`) a los métodos `addXXXListener(this)`, ahora tenemos que crear un objeto de la clase delegada, y éste será el que pasaremos a dichos métodos. Salvo esto, por lo demás no hay ninguna diferencia en la gestión de los eventos.

El uso de clases delegadas es una buena conducta de programación, ya que favorece la reutilización de código, por lo que se recomienda su uso.

**Ejemplo** Como ejemplo de uso de clases delegadas vamos a modificar el ejemplo presentado en el apartado anterior 8.6, en el que la clase que heredaba de `Frame` se encargaba de implementar las interfaces `WindowListener` y `ComponentListener`. Vamos transformarlo para que ambas interfaces sean implementadas por su correspondiente clase delegada, reduciéndose el código de la clase principal.

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class AWT_ComponentListener2 {
5     public static void main (String args[]) {
6        // Creamos objeto ventana y establecemos el titulo
7        AWT_ComponentListener2_Frame f =
8              new AWT_ComponentListener2_Frame ("Ejemplo ComponentListener");
9        // Establecemos tamaño de la ventana
10       f.setSize (200,100) ;
11       // Mostramos la ventana
12       f.setVisible (true);
13    }
14 }
15
16 class AWT_ComponentListener2_Frame extends Frame {
17    // Constructor
18    public AWT_ComponentListener2_Frame (String titulo) {
19       super (titulo);
20       WindowListenerDelegada o1 = new WindowListenerDelegada();
21       addWindowListener (o1);
22       ComponentListenerDelegada o2 = new ComponentListenerDelegada(this);
23       addComponentListener (o2);
24    }
25
26    // Sobreescribimos funcion paint() heredada
```

```
27      public void paint (Graphics g) {
28          g.drawString ("Posición: "+getX()+","+getY(), 30, 50);
29          g.drawString ("Tamaño : "+getWidth()+"x"+getHeight(), 30, 70);
30      }
31  }
32
33  /* Clase delegada para la interface WindowListener */
34  class WindowListenerDelegada implements WindowListener {
35      public void windowClosing (WindowEvent e) {
36          System.exit (0);    // dispose();
37      }
38      public void windowOpened      (WindowEvent e) {}
39      public void windowClosed      (WindowEvent e) {}
40      public void windowActivated   (WindowEvent e) {}
41      public void windowDeactivated (WindowEvent e) {}
42      public void windowIconified   (WindowEvent e) {}
43      public void windowDeiconified (WindowEvent e) {}
44  }
45
46  /* Clase delegada para la interface ComponentListener */
47  class ComponentListenerDelegada implements ComponentListener {
48      private Frame f;
49
50      public ComponentListenerDelegada (Frame f) {
51          this.f = f;
52      }
53
54      // Invocado cada vez que el componente deja de estar visible
55      public void componentHidden (ComponentEvent e) {}
56
57      // Invocado cuando el componente se mueve con el ratón
58      public void componentMoved (ComponentEvent e) {
59          //System.out.println("Posición: "+getX()+","+getY());
60          f.repaint();
61      }
62
63      // Invocado cuando el componente cambia de tamaño
64      public void componentResized (ComponentEvent e) {
65          //System.out.println("Tamaño : "+getWidth()+"x"+getHeight());
66          f.repaint();
67      }
68
69      // Invocado cada vez que el componente se hace visible
70      public void componentShown (ComponentEvent e) {}
71  }
```

Aspectos interesantes de este ejemplo:

- La forma de utilizar una clase delegada es creando un objeto de dicha clase y ese es el objeto que hay que pasar al método `addXXXListener()` en lugar de la referencia al objeto `Frame` actual `this` (líneas 20 a 23).
- Las dos clases delegadas escritas son reutilizables en otras aplicaciones, sin tener que escribir nada más.
- En el caso de la clase `ComponentListenerDelegada` es necesario escribir un constructor para poder pasarle la referencia al objeto `Frame`, puesto que necesitamos invocar a `repaint()` (líneas 60 y 66)

### Clases adaptadoras

Como hemos visto en los apartados anteriores, la forma de capturar eventos es implementado alguna interface `Listener`, es decir, implementando todos y cada uno de sus métodos. Algunas interfaces declaraban sólo 1 o 2 métodos, pero otras tenían hasta 7, que el programador tenía que implementar aunque no se fueran a utilizar (había que escribir código vacío). Para esos casos, Java ofrece las llamadas **clases adaptadoras**, que son clases que precisamente hacen lo que se acaba de comentar, implementar todos los métodos de una interface concreta con código vacío. La tabla 8.37 enumera las clases adaptadoras correspondientes a las interfaces explicadas en los apartados anteriores.

| Listener (interface) | Clase adaptadora |
|---|---|
| `WindowListener` | `WindowAdapter` |
| `ActionListener` | - |
| `ItemListener` | - |
| `FocusListener` | `FocusAdapter` |
| `KeyListener` | `KeyAdapter` |
| `MouseListener` | `MouseAdapter` |
| `MouseMotionListener` | `MouseAdapter` |
| `ComponentListener` | `ComponentAdapter` |

Cuadro 8.37: Eventos: métodos de cada interface `Listener`

Por ejemplo, la clase `WindowAdapter` es una clase que ya incorpora Java y que implementa los 7 métodos de la interface `WindowListener`, por lo que más o menos tendrá el siguiente código:

```
1 class WindowAdapter implements WindowListener {
2     public void windowOpened      (WindowEvent e) {}
3     public void windowClosing     (WindowEvent e) {}
4     public void windowClosed      (WindowEvent e) {}
5     public void windowActivated   (WindowEvent e) {}
6     public void windowDeactivated (WindowEvent e) {}
7     public void windowIconified   (WindowEvent e) {}
8     public void windowDeiconified (WindowEvent e) {}
9 }
```

Aparentemente estas clases no aportan nada, puesto que no tienen código, pero su intención es evitar que el programador tenga que escribir todos esos métodos si tan sólo va a utilizar uno o dos. La forma de utilizar estas clases adaptadoras es heredando de ellas. Es decir, **en vez de implementar una interface, lo que se hace es heredar de su correspondiente clase adaptadora**. De esta forma, puesto que nuestra clase hereda todos los métodos de dicha clase adaptadora, lo único que tenemos que escribir son los métodos o eventos que nos interese realizar alguna acción; el resto no es necesario escribirlos puesto que los heredamos ya implementados con código vacío.

Como se puede observar en la tabla 8.37, no existe clase adaptadora para las interfaces `ActionListener` e ItemListener, ya que al incluir un único método (ver tabla 8.27) no tendría sentido la existencia de una clase adaptadora.

**Ejemplo** Como ejemplo de uso de clases adaptadoras vamos a modificar el ejemplo presentado en el apartado anterior 8.6 en el que se utilizaban clases delegadas.

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class AWT_ComponentListener3 {
5     public static void main (String args[]) {
6        // Creamos objeto ventana y establecemos el titulo
7        AWT_ComponentListener3_Frame f =
8              new AWT_ComponentListener3_Frame ("Ejemplo ComponentListener");
9        // Establecemos tamaño de la ventana
10       f.setSize (200,100) ;
11       // Mostramos la ventana
12       f.setVisible (true);
13    }
14 }
15
16 class AWT_ComponentListener3_Frame extends Frame {
17    // Constructor
18    public AWT_ComponentListener3_Frame (String titulo) {
19       super (titulo);
20       WindowListenerDelegadaAdaptadora o1 =
21          new WindowListenerDelegadaAdaptadora();
22       addWindowListener (o1);
23       ComponentListenerDelegadaAdaptadora o2 =
24          new ComponentListenerDelegadaAdaptadora(this);
25       addComponentListener (o2);
26    }
27
28    // Sobreescribimos funcion paint() heredada
29    public void paint (Graphics g) {
30       g.drawString ("Posición: "+getX()+","+getY(), 30, 50);
31       g.drawString ("Tamaño : "+getWidth()+"x"+getHeight(), 30, 70);
```

```
32      }
33  }
34
35  /* Clase delegada que utiliza la clase adaptadora WindowAdapter */
36  class WindowListenerDelegadaAdaptadora extends WindowAdapter {
37      public void windowClosing (WindowEvent e) {
38          System.exit (0);    // dispose();
39      }
40  }
41
42  /* Clase delegada que utiliza la clase adaptadora ComponentAdapter */
43  class ComponentListenerDelegadaAdaptadora extends ComponentAdapter {
44      private Frame f;
45
46      public ComponentListenerDelegadaAdaptadora (Frame f) {
47          this.f = f;
48      }
49
50      // Invocado cuando el componente se mueve con el ratón
51      public void componentMoved (ComponentEvent e) {
52          //System.out.println("Posición: "+getX()+","+getY());
53          f.repaint();
54      }
55
56      // Invocado cuando el componente cambia de tamaño
57      public void componentResized (ComponentEvent e) {
58          //System.out.println("Tamaño : "+getWidth()+"x"+getHeight());
59          f.repaint();
60      }
61  }
```

Lo destacable de este ejemplo es lo siguiente:

- Las clases delegadas ahora no implementan las interfaces sino que heredan de sus correspondientes clases adaptadoras (líneas 36 y 43), sobreescribiendo únicamente los métodos de los eventos que interesa realizar alguna acción. Como resultado, su código se ha reducido.

**Cuestión** En el ejemplo anterior, lo que eran clases delegadas se han modificado para ser clases delegadas y adaptadoras. ¿Sería posible utilizar clases adaptadoras sin utilizar clases delegadas? Es decir, ¿sería posible que la clase que hereda de `Frame` en vez de implementar las interfaces, como se mostró en los primeros ejemplos, pueda utilizar las clases adaptadoras? La respuesta es **no**, ya que en Java no existe la herencia múltiple, y puesto que la clase ya hereda de la clase `Frame` (cosa necesaria), ya no podemos especificar que también hereda de otra clase adicional, como sería el caso de utilizar una clase adaptadora. Por tanto, **las clases adaptadoras sólo pueden utilizarse en el caso de clases delegadas**, y también en el caso de clases anónimas, cosa que se explica en el siguiente punto.

### Clases anónimas

A continuación vamos a explicar dos nuevas formas de capturar eventos. Son las que permiten capturar eventos con el mínimo código posible, aunque las más difíciles de entender.

Para no perdernos, la tabla 8.38 muestra las posibles formas de capturar eventos que estamos explicando en este capítulo. En la tabla se ve el caso comentado en el apartado anterior de que la clase que hereda de la clase `Frame` no puede utilizar clases adaptadoras. Nos falta por ver los dos últimos casos, que en realidad no difieren mucho de los otros, aunque hay que explicar el concepto de clase interna y clase anónima.

| | Implementar la interface | Heredar de clase adaptadora |
|---|---|---|
| Clase hereda de `Frame` | ✓ | ✗ |
| Clase delegada | ✓ | ✓ |
| Clase anónima | ✓ | ✓ |

Cuadro 8.38: Eventos: distintas formas de capturar eventos

**Clase interna** El lenguaje Java permite definir una clase dentro de otra. Las clases que están dentro de otras se denominan clases internas o anidadas.

**Clase anónima** Es posible definir una clase interna sin necesidad de asignar un nombre, si sólo va a utilizarse en un punto concreto. En este caso recibe el nombre de clase anónima. A la hora de instanciar un objeto de dicha clase anónima, en vez de su nombre se escribe todo su contenido.

**Ejemplo de clase anónima + interface Listener** Como ejemplo de cada caso vamos a seguir con el ejemplo que hemos estado modificando en los apartados anteriores aunque más simplificado; tan sólo será un `Frame` que captura los eventos para cerrar la ventana (`WindowListener`), sin escribir nada en la ventana (`ComponentListener`). En el primer caso se utiliza una clase anónima que implementa la interface `WindowListener`, por lo que dicha clase anónima debe implementar todos los métodos que declara la interface (líneas 21 a 31).

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class AWT_ComponentListener4 {
5     public static void main (String args[]) {
6        // Creamos objeto ventana y establecemos el titulo
7        AWT_ComponentListener4_Frame f =
8              new AWT_ComponentListener4_Frame ("Ejemplo ComponentListener");
9        // Establecemos tamaño de la ventana
10       f.setSize (200,100) ;
11       // Mostramos la ventana
12       f.setVisible (true);
13    }
14 }
15
```

```
16 class AWT_ComponentListener4_Frame extends Frame {
17    // Constructor
18    public AWT_ComponentListener4_Frame (String titulo) {
19       super (titulo);
20       // Atender eventos con clase anónima + interface
21       WindowListener o = new WindowListener() {
22          public void windowClosing (WindowEvent e) {
23             System.exit (0);    // dispose();
24          }
25          public void windowOpened      (WindowEvent e) {}
26          public void windowClosed      (WindowEvent e) {}
27          public void windowActivated   (WindowEvent e) {}
28          public void windowDeactivated (WindowEvent e) {}
29          public void windowIconified   (WindowEvent e) {}
30          public void windowDeiconified (WindowEvent e) {}
31       };
32       addWindowListener (o);
33    }
34 }
```

**Ejemplo de clase anónima + clase adaptadora** Ahora modificamos el ejemplo, para que la clase anónima herede de la clase adaptadora `WindowAdapter`, por lo que sólo es necesario sobreescribir los métodos que nos interese, que en este caso es sólo el `WindowClosing` (líneas 21 a 25). Vemos que con este código hemos escrito lo mínimo posible para capturar este evento, aunque es algo más complicado de entender que los otros casos.

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class AWT_ComponentListener5 {
5     public static void main (String args[]) {
6        // Creamos objeto ventana y establecemos el titulo
7        AWT_ComponentListener5_Frame f =
8              new AWT_ComponentListener5_Frame ("Ejemplo ComponentListener");
9        // Establecemos tamaño de la ventana
10       f.setSize (200,100) ;
11       // Mostramos la ventana
12       f.setVisible (true);
13    }
14 }
15
16 class AWT_ComponentListener5_Frame extends Frame {
17    // Constructor
18    public AWT_ComponentListener5_Frame (String titulo) {
19       super (titulo);
```

```
20          // Atender eventos con clase anónima + clase adaptadora
21          WindowListener o = new WindowAdapter() {
22             public void windowClosing (WindowEvent e) {
23                System.exit (0);     // dispose();
24             }
25          };
26          addWindowListener (o);
27       }
28    }
```

## 8.7. Menús

Las aplicaciones gráficas suelen tener una serie de menús desplegables desde la parte superior que permiten seleccionar las distintas opciones. A diferencia de los componentes que se han explicado hasta ahora, los menús no se colocan como ellos, sino que se crea una barra de menús asociada a una ventana (`Frame`), y a ella se le añaden los distintos menús.

La figura 8.8 muestra los componentes más habituales en relación con los menús de una aplicación gráfica: una barra de menús (`MenuBar`) en la parte superior asociada a una ventana (`Frame`), de la que cuelgan uno o varios menús (`Menu`). Cada uno de estos menús, a su vez, una serie de opciones seleccionables (`MenuItem`), opciones marcables (`CheckboxMenuItem`), separadores y otros submenús (`Menu`), formando una estructura jerárquica.

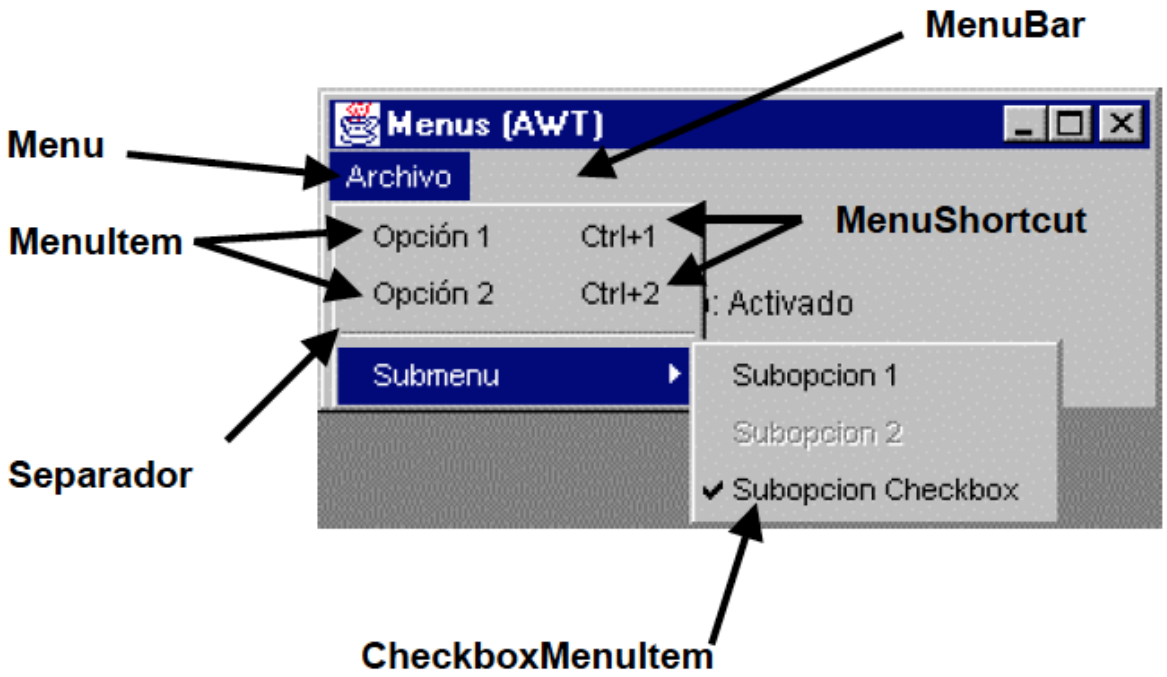



Figura 8.8: Menús: ejemplo de aplicación gráfica con menús

Las clases que se utilizan son las siguientes:

- `MenuBar`: representa una barra de menús desplegables asociada a un `Frame`.
- `Menu`: cada uno de los menús desplegables (*pull-down*) que cuelgan de una barra de menús (`MenuBar`), y que se abren al pulsar sobre su etiqueta (p.e. "Archivo"), mostrando todas sus opciones, las cuales tienen que ser `MenuItem` o subclases (ver figura 8.9).
- `MenuItem`: cada una de las opciones de un menú (`Menu`). Consisten en un texto que al pulsarlo se ejecutará alguna acción (`actionPerformed()`), similar a un botón (`Button`).

- `MenuShortcut`: consiste en una tecla rápida que puede asociarse de forma opcional a una opción de menú (`MenuItem`), con objeto de ejecutarla simplemente pulsando una combinación de teclas desde el teclado.
- `CheckboxMenuItem`: se trata de una opción de menú especial que representa una casilla marcable (`Checkbox`), de tal manera que cada vez que se seleccione se marcará o desmarcará, generándose un evento diferente al de los `MenuItem` estándar (`itemStateChanged()`).
- `PopupMenu`: se trata de un menú contextual o emergente que se muestra al hacer click con el ratón en la posición que se especifique, normalmente las coordenadas actuales del ratón (ver sección 8.7).

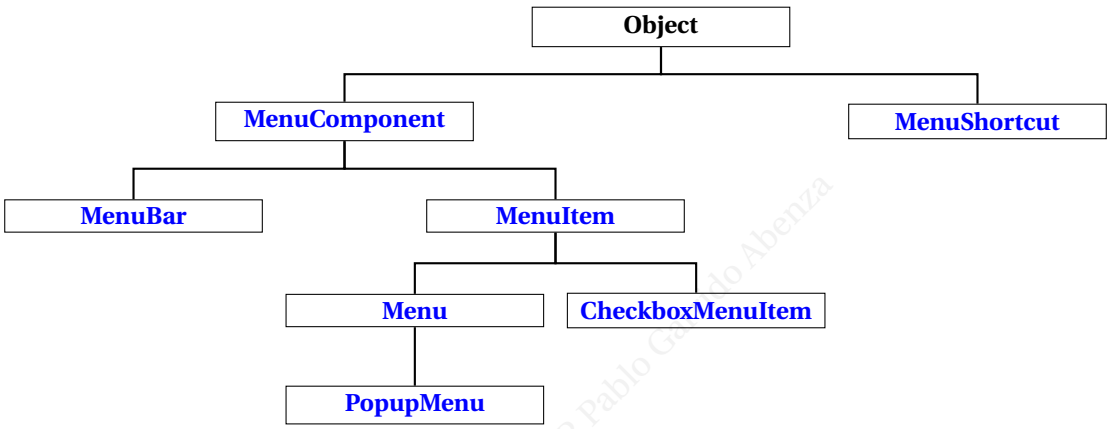



Figura 8.9: Menús: jerarquía de **clases**

A continuación mostramos en una misma tabla los constructores y métodos más interesantes de las clases mencionadas, resumidas a modo de referencia, aunque en la práctica sólo se usan unas pocas de ellas (las mostradas en color azul); después se mostrará un ejemplo de uso.

| Constructor/Método | Descripción |
|---|---|
| `MenuBar()` | Unico constructor. |
| `add(Menu m)` | Añadir el menú especificado a la barra. |
| `getHelpMenu()` | Obtener la referencia al menú de ayuda de la barra, el cual se habrá establecido con `setHelpMenu()`. |
| `getMenu(int pos)` | Obtener el menú de la posición especificada de la barra. |
| `getMenuCount()` | Obtener el número de menús de la barra. |
| `remove(int pos)` | Eliminar el menú de la posición especificada de la barra. |
| `setHelpMenu(Menu m)` | Establecer el menú de ayuda de la barra, el cual se sitúa en la parte derecha de la ventana. |

Cuadro 8.39: Menús: constructores y métodos de la clase `MenuBar`

La forma de establecer una barra de menús a una ventana `Frame` es mediante el método `setMenuBar()` de la clase `Frame` (ver tabla 8.3).

| Constructor/Método | Descripción |
|---|---|
| `Menu(String label)` | Constructor de un menú con la etiqueta especificada, que será la que se muestre en la barra de menús si se añade a la barra de menús, o como si fuera una opción de menú si se se añade a otro menú (submenú). |
| `add(MenuItem i)` | Añadir la opción de menú ya creada al menú (al final). |
| `addSeparator()` | Añadir una línea separadora al menú (al final). |
| `getItem(int pos)` | Obtener la referencia a la opción de menú de la posición especificada. |
| `remove(int pos)` | Eliminar la opción de menú de la posición especificada. |
| `removeAll()` | Eliminar todas las opciones de menú añadidas. |

Cuadro 8.40: Menús: constructores y métodos de la clase `Menu`

| Constructor/Método | Descripción |
|---|---|
| `MenuItem(String label)` | Constructor de una opción de menú con la etiqueta especificada, que será la que se muestre en el menú. |
| `MenuItem(String label, MenuShortcut s)` | Idem al anterior pero especificando una tecla rápida (`MenuShortcut`), que habrá que crear previamente. |
| `addActionListener(ActionListener l)` | Añadir el objeto que se encargará de la gestión de los eventos (método `actionPerformed()`). |
| `getLabel()` | Obtener la etiqueta establecida. |
| `isEnabled()` | Averiguar si la opción está habilitada (`true`) o no (`false`). |
| `setEnabled(boolean b)` | Establecer si la opción está habilitada (`true`) o no (`false`); cuando una opción está deshabilitada aparecerá en color gris y no se podrá seleccionar. |
| `setLabel(String label)` | Establecer la etiqueta de la opción. Normalmente no es necesaria, salvo que se quiera modificar dinámicamente en tiempo de ejecución. |
| `setShortcut(MenuShortcut s)` | Establecer la tecla rápida asociada a la opción de menú. Normalmente no es necesaria si se utiliza el constructor que recibe el objeto `MenuShortcut`. |

Cuadro 8.41: Menús: constructores y métodos de la clase `MenuItem`

| Constructor/Método | Descripción |
|---|---|
| `MenuShortcut(int key)` | Constructor de una tecla rápida con la tecla especificada, que será la que se utilice en una opción de menú (`MenuItem`). El código de tecla se especifica utilizando una de las constantes estáticas definidas en la clase `KeyEvent` (por ejemplo, `VK_0` ... `VK_1`, `VK_A` ... `VK_Z`, etc.). La combinación final de teclas a pulsar depende de la plataforma, y puede obtenerse con el método `Toolkit.getMenuShortcutKeyMask()`; por ejemplo, en Windows es necesario utilizar la tecla [Ctrl] junto con la tecla que se haya especificado, mientras que en Mac OS X es la tecla [Command]. |
| `MenuShortcut(int key, boolean shift)` | Idem al anterior pero especificando si hay que pulsar la tecla de mayúsculas [Shift] (`true`) o no (`false`). |
| `getKey()` | Obtener el código de la tecla (ver clase `KeyEvent`). |
| `usesShiftModifier()` | Averiguar si se ha establecido que hay que pulsar la tecla de mayúsculas [Shift] (`true`) o no (`false`) junto con el código de la tecla. |

Cuadro 8.42: Menús: constructores y métodos de la clase `MenuShortcut`

| Constructor/Método | Descripción |
|---|---|
| `CheckboxMenuItem(String label)` | Constructor de una opción de menú marcable con la etiqueta especificada, y no marcada por defecto. |
| `CheckboxMenuItem(String label, boolean state)` | Idem al anterior, pero especificando si la opción aparecerá marcada (`true`) o no (`false`) por defecto. |
| `addItemListener(ItemListener l)` | Añadir el objeto que se encargará de la gestión de los eventos (método `itemStateChanged()`). Cuando el usuario selecciona una opción de menú de este tipo, si estaba ya marcada se desmarca, y si no lo estaba se queda marcada, sin tener que hacerlo nosotros manualmente. |
| `getState()` | Averiguar si la opción está marcada (`true`) o no (`false`). |
| `setState(boolean state)` | Establecer manualmente si la opción está marcada (`true`) o no (`false`). |

Cuadro 8.43: Menús: constructores y métodos de la clase `CheckboxMenuItem`

**Ejemplo** A continuación mostramos el código necesario para crear la ventana mostrada anteriormente, capturando los eventos necesarios para responder a las acciones del usuario sobre los elementos del menú. Lo que se hace es simplemente mostrar un texto sobre un objeto `Label` indicando la opción que se ha ejecutado.

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class AWTMenus {
5     public static void main (String args[]) {
6        // Creamos objeto ventana
7        AWT_Menus_Frame f = new AWT_Menus_Frame("Menus (AWT)");
8        // Establecemos tamaño de la ventana
9        f.setSize (300,100);
10       // Mostramos la ventana
11       f.setVisible (true);
12    }
13 }
14
15 class AWT_Menus_Frame extends Frame
16                   implements WindowListener, ActionListener, ItemListener {
17    MenuBar      menubar;
18    Menu         menu, submenu;
19    MenuItem     menuitem1, menuitem2, subitem1, subitem2;
20    MenuShortcut shortcut;
21    CheckboxMenuItem cbsubitem;
22
23    Label label;
24
25    public AWT_Menus_Frame(String titulo) {
```

```
26          super(titulo);
27
28          setLayout(new GridLayout(1,1));
29          label = new Label("Selecciona una opción ...");
30          add(label);
31
32          menubar = new MenuBar();
33          setMenuBar(menubar);
34
35          menu = new Menu("Archivo");
36          menubar.add(menu);
37
38          shortcut = new MenuShortcut(KeyEvent.VK_1);
39          menuitem1 = new MenuItem("Opción 1", shortcut);
40          menu.add(menuitem1);
41          menuitem1.addActionListener(this);
42
43          shortcut = new MenuShortcut(KeyEvent.VK_2);
44          menuitem2 = new MenuItem("Opción 2", shortcut);
45          menu.add(menuitem2);
46          menuitem2.addActionListener(this);
47
48          menu.addSeparator();
49
50          submenu = new Menu("Submenú");
51          menu.add(submenu);
52
53          subitem1 = new MenuItem("Subopción 1");
54          submenu.add(subitem1);
55          subitem1.addActionListener(this);
56          subitem2 = new MenuItem("Subopción 2");
57          submenu.add(subitem2);
58          subitem2.addActionListener(this);
59          subitem2.setEnabled(false);       // Deshabilitamos
60
61          cbsubitem = new CheckboxMenuItem("Subopción Checkbox");
62          submenu.add(cbsubitem);
63          cbsubitem.addItemListener(this);
64
65          // Registramos gestor de eventos de ventana
66          addWindowListener(this);
67      }
68
69      public void windowClosing     (WindowEvent e) {
70          System.exit(0);  // dispose();
71      }
```

```
72      public void windowOpened      (WindowEvent e) { }
73      public void windowClosed      (WindowEvent e) { }
74      public void windowActivated   (WindowEvent e) { }
75      public void windowDeactivated (WindowEvent e) { }
76      public void windowIconified   (WindowEvent e) { }
77      public void windowDeiconified (WindowEvent e) { }
78
79      // Opciones de menu
80      public void actionPerformed (ActionEvent event) {
81         if (event.getSource() == menuitem1) {
82            label.setText("Opcion 1");
83         } else if (event.getSource() == menuitem2) {
84            label.setText("Opcion 2");
85         } else if (event.getSource() == subitem1) {
86            label.setText("Subopcion 1");
87         } else if (event.getSource() == subitem2) {
88            label.setText("Subopcion 2");
89         }
90      }
91
92      // Opciones de menu de tipo CheckBox
93      public void itemStateChanged(ItemEvent event) {
94         if (event.getSource() == cbsubitem) {
95            if (cbsubitem.getState()) {
96               label.setText("Subopcion 3 (Checkbox): Activado");
97            } else {
98               label.setText("Subopcion 3 (Checkbox): Desactivado");
99            }
100        }
101     }
102  }
```

Aspectos a destacar de este ejemplo:

- Los objetos `MenuItem` estándar generan los eventos de la interface `ActionListener`, por lo que debemos implementar el método `actionPerformed()`.
- Los eventos que generan los objetos de tipo `CheckboxMenuItem` son los de la interface `ItemListener`; es decir, debemos implementar el método `itemStateChanged()`.

### Menús emergentes o contextuales

Además de las barras de menús, es posible mostrar un menú en cualquier lugar dentro de un componente, normalmente al hacer click-dcho. con el ratón, mostrándose lo que se llama un menú emergente justo en la posición del cursor. Se denominan también menús contextuales porque dependiendo de la zona o componente en la que hayamos pulsado el ratón (contexto) podremos mostrar un menú u otro, con opciones distintas.

Para ello se utiliza la clase `PopupMenu` y el método `show()` para mostrarlo en el lugar que queramos. Como se puede observar en la figura 8.9, la clase `PopupMenu` hereda directamente de la clase `Menu`, por lo que un objeto `PopupMenu` puede utilizarse de la misma forma que un objeto `Menu`, es decir, podríamos colgarlo en una barra de menús. Sin embargo, si hacemos esto ya no podremos mostrarlo en otro lugar mediante el método `show()`.

Puesto que utilizando herencia no se heredan los constructores, es necesario que la clase `PopupMenu` disponga de algún constructor. Además, esta clase extiende la clase `Menu` con el mencionado método `show()`, el cual recibe como parámetros las coordenadas donde queremos mostrar el menú (tabla 8.44). Normalmente, estas coordenadas son las coordenadas actuales del cursor del ratón, por lo que, como veremos en el siguiente ejemplo, es necesario capturar los eventos del ratón relacionados con la pulsación de sus botones: `MouseListener`.

| Constructor/Método | Descripción |
|---|---|
| `PopupMenu()` | Constructor de un menú emergente sin etiqueta (lo habitual). |
| `PopupMenu(String label)` | Constructor de un menú emergente con la etiqueta especificada, sólo utilizable en el caso de querer colgar el menú en una barra de menús, para lo cual sería preferible utilizar la clase `Menu`. |
| `show(Component origin, int x, int y)` | Mostrar el menú contextual en las coordenadas (x,y) relativas al origen de coordenadas del componente especificado, siendo el origen (0,0) en la esquina superior izquierda. |

Cuadro 8.44: Menús: constructores y métodos de la clase `PopupMenu`

**Ejemplo** Como ejemplo de uso de un menú contextual, vamos escribir un ejemplo parecido al anterior sustituyendo la barra de menús por un menú contextual que aparecerá en la posición del cursor al hacer click-dcho. con dos opciones únicamente. Para responder a la pulsación de los botones del ratón se requiere implementar la interface `MouseListener`; en este ejemplo utilizaremos una clase delegada. Para cerrar la ventana, en este caso utilizamos también una clase delegada, la cual hereda de la clase adaptadora `WindowAdapter`, en vez de implementar la interface `WindowListener`.

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class AWT_MenusEmergentes {
5    public static void main (String args []) {
6       // Creamos objeto ventana
7       FramePopupMenusAWT f = new FramePopupMenusAWT ("Menus emergentes (AWT)");
```

```
8         // Establecemos tamaño de la ventana
9         f.setSize (300,100) ;
10        // Mostramos la ventana
11        f.setVisible (true);
12     }
13  }
14
15  // ActionListener: para las opciones de menus
16  class FramePopupMenusAWT extends Frame implements ActionListener {
17     Panel     panel;                 // Utilizado como contenerdor
18     PopupMenu m;                     // Menu contextual
19     MenuItem  menuitem1, menuitem2;  // Elementos del menu
20     Label     label;                 // Una etiqueta
21
22     public FramePopupMenusAWT(String titulo) {
23        // Establecemos el titulo
24        super(titulo);
25        setLayout(new BorderLayout());
26
27        // Creamos el panel contenedor
28        // (necesario para capturar eventos del raton)
29        panel = new Panel();
30        add(panel, BorderLayout.CENTER);
31        // Añadimos componentes al contenedor
32        label = new Label("Selecciona una opción del menu contextual ...");
33        add(label, BorderLayout.SOUTH);
34
35        // Creamos el menu contextual y le añadimos las opciones
36        m = new PopupMenu("Menu emergente");
37        //
38        menuitem1 = new MenuItem("Opcion 1");
39        m.add(menuitem1);
40        menuitem1.addActionListener(this);
41        //
42        menuitem2 = new MenuItem("Opcion 2");
43        m.add(menuitem2);
44        menuitem2.addActionListener(this);
45        // Añadimos el popup menu a la aplicacion
46        this.add(m);
47        // Captura eventos de raton
48        // ATENCION: El manejador de eventos se asocia al panel,
49        // Si se asocia al "frame" no funciona.
50        panel.addMouseListener(new ControladorRaton(m));
51        // Captura eventos de la ventana
52        this.addWindowListener(new ControladorVentana());
53     }
```

```
54
55      // Seleccion de opciones
56      public void actionPerformed (ActionEvent event) {
57          if (event.getSource() == this.menuitem1) {
58              this.label.setText("Opcion 1");
59          } else if (event.getSource() == this.menuitem2) {
60              this.label.setText("Opcion 2");
61          }
62      }
63  }
64
65  // Eventos de la ventana (clase delegada + adaptadora)
66  class ControladorVentana extends WindowAdapter {
67      public void windowClosing(WindowEvent e) {
68          System.exit(0);
69      }
70  }
71
72  // Eventos del raton (clase delegada)
73  class ControladorRaton implements MouseListener { // extends MouseAdapter {
74      private PopupMenu m;
75
76      public ControladorRaton (PopupMenu m) {
77          this.m = m;
78      }
79
80      // Pulsacion de teclas del raton
81      public void mousePressed(MouseEvent event) {
82          // Averiguamos si hemos pulsado el boton derecho (Button3)
83          //if ((event.getModifiers() & InputEvent.BUTTON3_MASK) == InputEvent.
                BUTTON3_MASK) {
84          if (event.getButton() == MouseEvent.BUTTON3) {
85              // Obtenemos coordenadas actuales del cursor del ratón
86              Component c = event.getComponent();
87              int x = event.getX();
88              int y = event.getY();
89              // Mostramos el menú emergente en esas coordenadas
90              m.show(c, x, y);
91          }
92      }
93      public void mouseClicked (MouseEvent event) {};
94      public void mouseEntered (MouseEvent event) {};
95      public void mouseExited  (MouseEvent event) {};
96      public void mouseReleased(MouseEvent event) {};
97  }
```

Aspectos a destacar de este ejemplo:

- El uso de **clases delegadas** para la gestión de eventos evita abarrotar la clase `Frame`. Sin embargo, puede ser necesario pasar algún argumento, como es el caso del menú contextual a la clase delegada de los eventos del ratón (líneas 50 y 77), para poder mostrarlo (línea 90).
- El uso de **clases adaptadoras** reduce el código, evitando tener que escribir todos los métodos de la interface aunque sea con código vacío (líneas 66 a 70).
- Con el **parámetro** `MouseEvent` podemos saber cuál de los botones se ha pulsado (incluso con alguna tecla adicional), obtener el componente que ha generado el evento, y las coordenadas (x,y) del cursor del ratón en el momento de producirse el evento (líneas 84 a 88).

## 8.8. Cuadros de diálogo

Un *cuadro de diálogo*, o simplemente *diálogo*, es una ventana secundaria que se utiliza para pedir algún valor al usuario o confirmación para hacer algo. Se requiere que alguna otra ventana (`Frame` u otro `Dialog`) la cree y la muestre, y la llamaremos ventana padre o propietaria (*owner*). Normalmente, la ventana padre es un `Frame`, es decir, la ventana principal, aunque un diálogo `Dialog` puede también abrir otro diálogo. Por ello, en la documentación de la clase `Dialog` existen varios constructores duplicados, en los que podemos especificar como ventana padre (*owner*) un objeto `Frame`, un objeto `Dialog`, o un objeto `Window`.

Puesto que tanto las clases `Frame` como `Dialog` heredan de la clase `Window` (ver jerarquía de la figura 8.1), en la tabla 8.45 se muestran sólo los constructores que reciben un objeto `Window` como ventana padre (*owner*), entendiéndose que se puede especificar un objeto cualquiera de esas dos clases.

Además, puesto que la clase `Window` heredaba, a su vez, de `Container`, y ésta de `Component`, a la hora de trabajar con un objeto `Dialog` podremos utilizar todos los métodos heredados de todas sus superclases (propagación de herencia), junto con alguno más que se muestra en la tabla 8.45, que son los mismos que tenía la clase `Frame` (ver tabla 8.3), más los relacionados con establecer o consultar si el diálogo es modal o no.

Un diálogo se puede abrir en modo *modal* o en modo *no-modal*. Un diálogo *modal* es uno que bloquea las acciones sobre cualquier otra ventana mientras esté abierto, es decir, hasta que el usuario cierre la ventana; por otra parte, un diálogo *no-modal* sí permite volver a activar cualquier otra ventana, incluida la ventana principal `Frame` que abrió dicho diálogo, e interactuar con ella mientras el diálogo permanece abierto.

| Constructor/Método | Descripción |
|---|---|
| `Dialog(Window owner)` | Constructor de un diálogo sin título, no modal. |
| `Dialog(Window owner, String title)` | Constructor de un diálogo con el título especificado, no modal. |
| `Dialog(Window owner, boolean modal)` | Constructor de un diálogo sin título, y en modo modal o no según se haya especificado. |
| `Dialog(Window owner, String title, boolean modal)` | Constructor de un diálogo con el título especificado, y en modo modal o no según se haya especificado. |
| `getTitle()` | Obtener el título de la ventana. |
| `isModal()` | Averiguar el diálogo es modal o no modal. |
| `isResizable()` | Averiguar si el usuario puede cambiar el tamaño de la ventana o no. |
| `setModal(boolean modal)` | Especificar si el diálogo será modal (`true`) o no (`false`). |
| `setResizable(boolean resizable)` | Establecer si el usuario puede cambiar el tamaño de la ventana (`true`) o no (`false`). |
| `setTitle(String title)` | Establecer el título de la ventana. |

Cuadro 8.45: Clase `Dialog` - constructores y métodos

Los `Dialog` se diferencian de los Frame básicamente en los siguientes aspectos:

1. Un `Dialog` siempre deberá tener una ventana padre, que podrá ser un `Frame` o también otro `Dialog`.
2. Un `Dialog` no puede tener una barra de menús, aunque sí menús contextuales.
3. En Windows, los `Dialog` sólo tienen el botón cerrar [X], pero no el de maximizar ni minimizar. En Mac OS X no tienen el botón de minimizar, pero sí el de cerrar y maximizar.

Para **crear nuestros propios diálogos** procedemos de la misma forma que para los `Frame`, es decir, tenemos que escribir una clase que herede de `Dialog`, en este caso, y definir al menos un constructor en el que invoquemos a cualquiera de los constructores de dicha superclase `Dialog`.

En cuanto a la forma de **cerrar una ventana** de tipo `Dialog`, tenemos diversas opciones:

- `dispose()`: cierra el diálogo y lo descarga de la memoria (libera recursos).
- `setVisible(false)`: oculta el diálogo, pero continúa en la memoria; esto puede ser útil si más adelante se tiene al intención de volver a mostrar este mismo diálogo con `setVisible(true)`, lo cual será más rápido que volver a crear el diálogo y mostrarlo. También es útil en el caso de que la ventana padre necesite acceder a alguna variable del diálogo una vez cerrado, para saber, por ejemplo, si se ha abortado o no.
- `System.exit(0)`: este es el método habitual de cerrar un `Frame`, pero en el caso de un `Dialog`, si lo cerramos de esta forma se cerrará también el `Frame`, es decir, la aplicación terminará. Puesto que probablemente esta no sea nuestra intención, este método no es válido para cerrar un `Dialog`.

**Ejemplo** Todo lo anterior se verá más claro con el siguiente ejemplo, que consiste simplemente en pedir confirmación antes de salir del programa cuando se intenta cerrar la ventana.

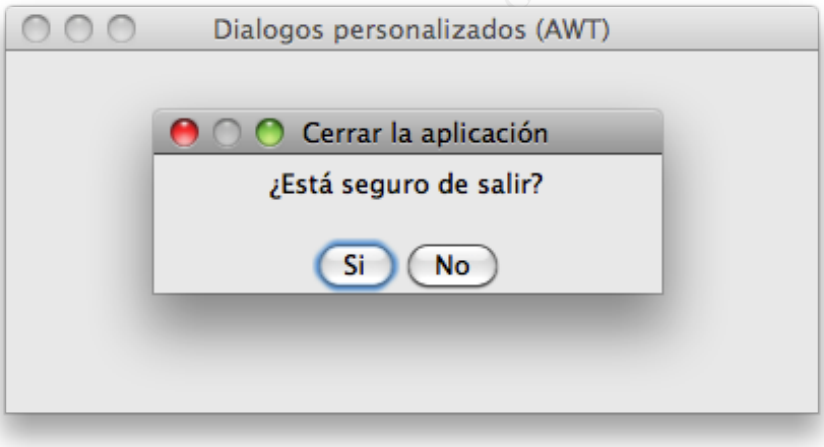



Figura 8.10: Diálogos: ejemplo de solicitar confirmación para salir

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class AWT_Dialogs1 {
5     public static void main (String args []) {
6        // Creamos objeto ventana
7        AWT_Dialogs1_Frame f = new AWT_Dialogs1_Frame ("Dialogos (AWT)");
8        // Establecemos tamaño de la ventana
9        f.setSize (400,200);
```

```
10          // Mostramos la ventana
11          f.setVisible (true);
12       }
13    }
14
15    // ActionListener: para las opciones de menus
16    class AWT_Dialogs1_Frame extends Frame implements WindowListener {
17
18       public AWT_Dialogs1_Frame(String titulo) {
19          // Establecemos el titulo
20          super(titulo);
21          // Captura eventos de la ventana
22          this.addWindowListener(this);
23       }
24
25       /* WindowListener */
26       public void windowClosing (WindowEvent e) {
27          // Creamos para pedir confirmacion (será modal)
28          DialogConfirmar dlg = new DialogConfirmar(this,
29                   "Cerrar la aplicación", "¿Está seguro de salir?");
30          // Mostramos dialogo (de forma modal)
31          dlg.setVisible(true);
32          // Una vez cerrado el diálogo consultamos si hemos confirmado o no
33          if (dlg.confirmed) {
34             System.exit(0);
35          }
36          // Descargamos el diálogo (aun sigue en memoria)
37          dlg.dispose();
38       }
39
40       public void windowOpened      (WindowEvent e) { }
41       public void windowClosed      (WindowEvent e) { }
42       public void windowActivated   (WindowEvent e) { }
43       public void windowDeactivated (WindowEvent e) { }
44       public void windowIconified   (WindowEvent e) { }
45       public void windowDeiconified (WindowEvent e) { }
46
47    }
48
49    /* Dialogo para pedir confirmacion antes de salir */
50    class DialogConfirmar extends Dialog
51                          implements ActionListener, WindowListener {
52       Label   lblQuestion;
53       Button  ok, cancel;
54       boolean confirmed = false;
55
```

```
56      public DialogConfirmar (Frame owner, String title, String question) {
57          super (owner, title, true);
58
59          // Establecemos el tamaño
60          setSize (250, 90);
61          // Establecemos la posición del diálogo centrado respecto el Frame
62          setLocationRelativeTo(owner);
63          // Construimos la ventana
64          setLayout (new GridLayout(2,1));
65
66          Panel p1 = new Panel();
67          add (p1);
68          p1.setLayout (new FlowLayout());
69          lblQuestion = new Label(question);
70          p1.add (lblQuestion);
71
72          Panel p2 = new Panel();
73          add (p2);
74          p2.setLayout (new FlowLayout());
75          ok = new Button("Si");
76          p2.add (ok);
77          cancel = new Button("No");
78          p2.add (cancel);
79          ok.addActionListener (this);
80          cancel.addActionListener (this);
81
82          // Registramos manejador de eventos
83          addWindowListener(this);
84      }
85
86      /* ActionListener */
87      public void actionPerformed (ActionEvent e) {
88          if (e.getSource()==ok) {
89              confirmed = true;
90          } else if (e.getSource()==cancel) {
91              confirmed = false;
92          }
93          // Ocultamos el dialogo (no lo descargamos!)
94          setVisible(false);  // dispose();
95      }
96
97      /* WindowListener */
98      public void windowClosing (WindowEvent e) {
99          confirmed = false;
100         // Ocultamos el dialogo (no lo descargamos!)
101         setVisible(false);  // dispose();
```

```
102     }
103     public void windowOpened      (WindowEvent e) { }
104     public void windowClosed      (WindowEvent e) { }
105     public void windowActivated   (WindowEvent e) { }
106     public void windowDeactivated (WindowEvent e) { }
107     public void windowIconified   (WindowEvent e) { }
108     public void windowDeiconified (WindowEvent e) { }
109 }
```

Analizando este ejemplo podemos observar lo siguiente:

- Desde una ventana `Frame` se crea un diálogo y se muestra mediante `setVisible(true)` (líneas 28 a 31).
- A la hora de crear un diálogo es necesario especificar la ventana padre. El diálogo se ha diseñado de tal forma que sea genérico, es decir, especificando un título y la pregunta que se quiere formular al invocar al constructor con `new` podremos utilizarlo para preguntar cualquier cosa al usuario (líneas 28 y 56).
- Utilizando el método `setLocationRelativeTo(owner)` conseguimos que el diálogo se muestre centrado respecto a la ventana `Frame`; por defecto aparece sobre su esquina superior izquierda (línea 62).
- En el diálogo, cuando pulsamos cualquiera de los botones, asignamos `true` o `false` a una variable antes de ocultarlo mediante `setVisible(false)` (líneas 88 a 94).
- Para ocultar un diálogo también puede descargarse mediante `dispose()` desde dentro de su código, pero no nos interesa en este caso porque queremos consultar el valor de la variable desde la ventana `Frame` para actuar en consecuencia (línea 33): si habíamos confirmado entonces cerraremos la aplicación mediante `System.exit(0)` (línea 34), en caso contrario podemos descargar el diálogo mediante `dispose()` (línea 37), y la aplicación seguirá abierta.

### Cuadros de diálogo predefinidos: `FileDialog`

La clase `FileDialog` consiste en un cuadro de diálogo predefinido que se utiliza para solicitar al usuario un nombre de archivo a leer o escribir. El usuario podrá seleccionar un archivo existente explorando por los distintos directorios del sistema de ficheros, o bien, escribir su nombre. El uso de la clase `FileDialog` tiene la ventaja de que el programador no necesita construir su propio diálogo para ello, y el cuadro de diálogo tiene un aspecto coherente con el que utiliza su sistema operativo.

Se trata de un diálogo **modal**, es decir, que hasta que el usuario no seleccione el nombre del archivo (o cancele la operación) el resto de ventanas de la aplicación permanecerán bloqueadas.

La tabla 8.46 resume los constructores y métodos más interesantes. En cuanto a los constructores mostrados, comentar que ocurre como con la clase `Dialog`: existen 2 versiones para el caso de que la ventana padre sea un `Frame` u otro `Dialog`, pero por abreviar, aquí escribimos `Window`, por heredar ambas clases de ella.

| Constructor/Método | Descripción |
|---|---|
| `Dialog(Window parent)` | Constructor de un diálogo para leer un archivo, cuyo padre puede ser un `Frame` o un `Dialog`. |
| `Dialog(Window parent, String title)` | Constructor de un diálogo con el título especificado para leer un archivo. |
| `Dialog(Window parent, String title, int mode)` | Constructor de un diálogo con el título especificado para leer o escribir un archivo, según el modo especificado `FileDialog.LOAD` o `FileDialog.SAVE`, respectivamente. |
| `getDirectory()` | Obtener el directorio del archivo seleccionado. |
| `getFile()` | Obtener el nombre del archivo seleccionado. En caso de que el usuario cancele retornará `null`. |
| `getFilenameFilter()` | Obtener el objeto `FilenameFilter` utilizado para filtrar los archivos que se muestran en el diálogo. |
| `getMode()` | Obtener el modo de lectura (`FileDialog.LOAD`) o escritura (`FileDialog.SAVE`) del diálogo. |
| `setDirectory(String dir)` | Establecer el directorio inicial que utilizará el diálogo para mostrar archivos. |
| `setFile(String file)` | Establecer el nombre del archivo por defecto que se seleccionará, aunque el usuario luego podrá seleccionar cualquier otro. |
| `setFilenameFilter( FilenameFilter filter)` | Establecer el filtro de los archivos que se mostrarán en el diálogo. |
| `setMode(int mode)` | Establecer el modo del diálogo, es decir, si se utilizará para seleccionar un archivo a leer (`FileDialog.LOAD`) o a escribir (`FileDialog.SAVE`). |

Cuadro 8.46: Menús: constructores y métodos de la clase `MenuBar`

La clase dispone de dos **constantes** estáticas para especificar si vamos a seleccionar un archivo para leer o para escribir, lo cual afecta ligeramente a su comportamiento. Lo más habitual es pasar estas constantes al constructor en el momento de crear el diálogo, aunque también puede establecerse posteriormente mediante el método `setMode()`, lo cual permitiría crear un único diálogo que podría utilizarse tanto para pedir el nombre de un archivo a leer como para escribir, cambiando simplemente el modo.

- `FileDialog.LOAD`: se pretende seleccionar un archivo para leerlo, por lo que deberá existir.
- `FileDialog.SAVE`: se pretende seleccionar un archivo para escribir, por lo que podría no existir.

Para más detalles sobre establecer **filtros** en los nombres de los archivos consultar en la documentación la clase `FilenameFilter`.

**Ejemplo** En el siguiente ejemplo utilizamos un `FileDialog` para solicitar el nombre de un archivo a leer o a escribir, según la opción de menú seleccionada. El nombre seleccionado se mostrará en la parte inferior, aunque no se leerá ni escribirá nada. Según la siguiente captura de pantalla, el archivo seleccionado para lectura será: `...\Documents\Java\Notas.txt`. Lo que obtenemos con el método `getFile()` será únicamente el nombre del fichero `Notas.txt`; para obtener el nombre con la ruta completa (absoluta) habrá que concatenarle delante lo que retorna el método `getDirectory()`, tal y como se hace en el código que se muestra a continuación.

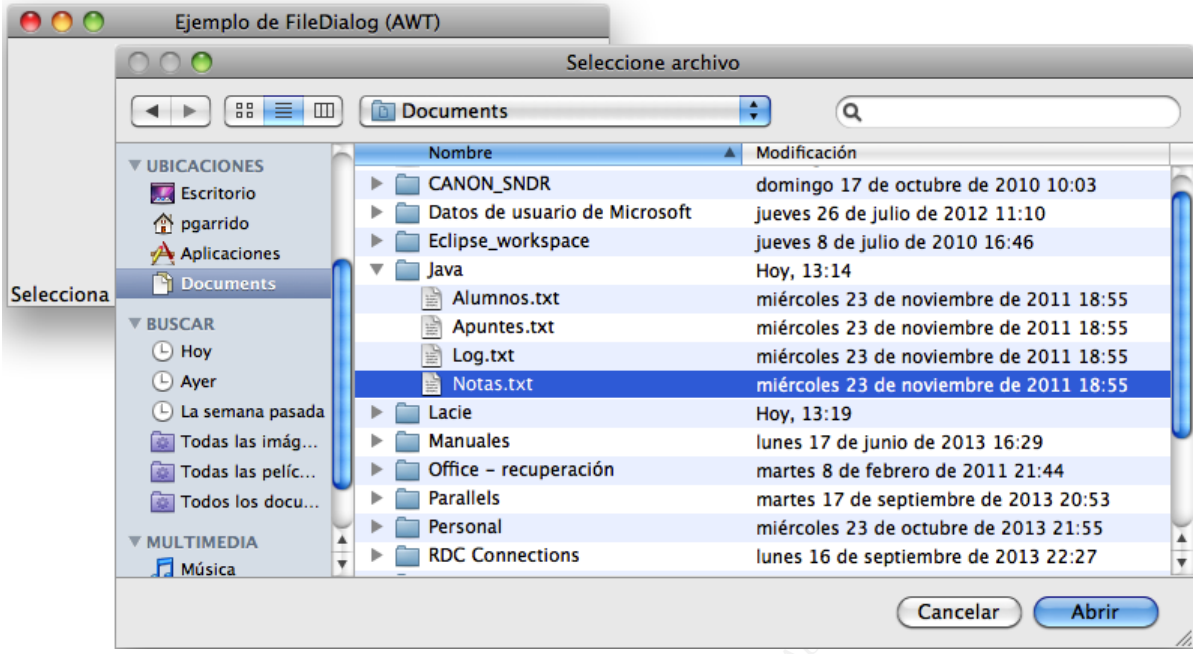



Figura 8.11: Diálogos: ejemplo de `FileDialog`

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class AWT_Dialogs2 {
5     public static void main (String args[]) {
6        // Creamos objeto ventana y establecemos el titulo
7        AWT_Dialogs2_Frame f =
8             new AWT_Dialogs2_Frame ("Ejemplo de FileDialog (AWT)");
9        // Establecemos tamaño de la ventana
10       f.setSize (400,200);
11       // Mostramos la ventana
12       f.setVisible (true);
13    }
14 }
15
16 class AWT_Dialogs2_Frame extends Frame
17                         implements WindowListener, ActionListener
18 {
19    MenuBar  menubar;
20    Menu     menu;
21    MenuItem leer, grabar, salir;
22    MenuShortcut shortcut;
23    Label label;
24
```

```
25      public AWT_Dialogs2_Frame (String titulo) {
26          super(titulo);
27          setLayout(new BorderLayout());
28
29          // Añadimos componentes al contenedor
30          label = new Label("Selecciona una opción del menu ...");
31          add(label, BorderLayout.SOUTH);
32
33          menubar = new MenuBar();
34          setMenuBar(menubar);
35
36          menu = new Menu("Archivo");
37          menubar.add(menu);
38
39          shortcut = new MenuShortcut(KeyEvent.VK_L); // CTRL+L o Comando+L
40          leer = new MenuItem("Leer", shortcut);
41          menu.add(leer);
42          leer.addActionListener(this);
43
44          shortcut = new MenuShortcut(KeyEvent.VK_G); // CTRL+G o Comando+G
45          grabar = new MenuItem("Guardar", shortcut);
46          menu.add(grabar);
47          grabar.addActionListener(this);
48
49          menu.addSeparator();
50
51          shortcut = new MenuShortcut(KeyEvent.VK_S); // CTRL+S o Comando+S
52          salir = new MenuItem("Salir", shortcut);
53          menu.add(salir);
54          salir.addActionListener(this);
55
56          // Registramos gestor de eventos de ventana
57          addWindowListener(this);
58      }
59
60      /* WindowListener */
61      public void windowClosing     (WindowEvent e) {
62          System.exit(0);  // dispose();
63      }
64      public void windowOpened      (WindowEvent e) { }
65      public void windowClosed      (WindowEvent e) { }
66      public void windowActivated   (WindowEvent e) { }
67      public void windowDeactivated (WindowEvent e) { }
68      public void windowIconified   (WindowEvent e) { }
69      public void windowDeiconified (WindowEvent e) { }
70
```

```
71      // Opciones de menu
72      public void actionPerformed (ActionEvent event) {
73         FileDialog dlg;
74         String filename;
75
76         if (event.getSource() == leer) {
77            // Creamos dialogo para pedir nombres de ficheros
78            dlg = new FileDialog(this, "Seleccione archivo", FileDialog.LOAD);
79            //dlg.setMode (FileDialog.LOAD);
80            dlg.setVisible(true);
81            filename = dlg.getFile();
82            if (filename != null) {
83               filename = dlg.getDirectory() + dlg.getFile();
84               label.setText("Fichero a leer: " + filename);
85            }
86         } else if (event.getSource() == grabar) {
87            dlg = new FileDialog(this, "Seleccione archivo", FileDialog.SAVE);
88            //dlg.setMode (FileDialog.SAVE);
89            dlg.setVisible(true);
90            filename = dlg.getFile();
91            if (filename != null) {
92               filename = dlg.getDirectory() + dlg.getFile();
93               label.setText("Fichero a grabar: " + filename);
94            }
95         } else if (event.getSource() == salir) {
96            System.exit(0);
97         }
98      }
99   }
```

Algunos comentarios finales sobre el ejemplo:

- Se crea un `FileDialog` cada vez que el usuario selecciona la opción Leer o Grabar, pero en cada caso especificando el modo correspondiente `FileDialog.LOAD` o `FileDialog.SAVE` (líneas 78 y 87).
- Una vez creado el diálogo, podemos volver a cambiar el modo con `setMode()` o cambiar el directorio inicial con `setDirectory()`, y después se puede mostrar mediante `setVisible(true)` (líneas 80 y 89).
- Una vez mostrado el diálogo, la ejecución se bloqueará hasta que el usuario seleccione un archivo o cancele. La forma de saber el archivo seleccionado es mediante el método `getFile()`, el cual retorna `null` en caso de que el usuario haya cancelado.
- Finalmente, en caso de que el usuario no cancele, se muestra en la parte inferior el nombre completo del archivo seleccionado, concatenando lo que retornan los métodos `getDirectory()` y `getFile()`.

## 8.9. Gráficos

En esta sección se van a presentar las clases necesarias para dibujar objetos gráficos en un componente (líneas, círculos, imágenes, etc.), utilizar distintos colores, cambiar la fuente de letra de los textos, cambiar el icono de la aplicación, y también cómo reproducir sonidos.

Las clases que se van a utilizar son las que se muestran en la figura 8.12, las cuales se describen a continuación.

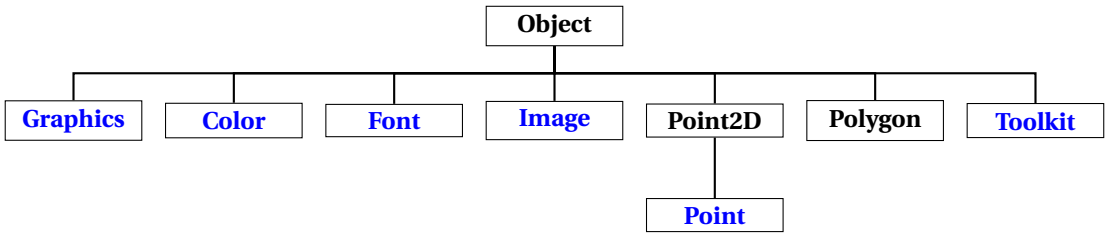



Figura 8.12: Gráficos: jerarquía de clases



- `Graphics`: es la clase base que se utiliza para poder dibujar cualquier figura en un componente sobreescribiendo el método `paint()`.
- `Color`: clase que se utiliza para dibujar figuras de un determinado color, establecer el color del fondo de componentes, ...; encapsula una serie de constantes con colores predefinidos y permite crear otros nuevos según el espacio RGB (*red-green-blue*).
- `Font`: nos permite cambiar la fuente de letra por defecto de los componentes, nombre de fuente, negrita, cursiva, o modificar su tamaño.
- `Image`: clase utilizada para cargar imágenes gráficas desde archivos en diversos formatos gráaficos.
- `Point`: encapsulan las coordenadas de un punto (x,y).
- `Polygon`: encapsula una serie de puntos (x,y) que forman un polígono cerrado.
- `Toolkit`: Nos proporciona métodos para obtener información gráfica del sistema: resolución, gestión del portapapeles, fuentes disponibles, obtener imágenes desde ficheros gráficos (GIF, JPEG, PNG), emitir un sonido (beep), etc. Para utilizar esta clase es necesario obtener la referencia al *toolkit* del sistema mediante: `Toolkit tk = Toolkit.getDefaultToolkit()`.

### Sistema de coordenadas

Para dibujar con los métodos de la clase `Graphics` que veremos a continuación es necesario conocer que el **origen del sistema de coordenadas** (0,0) se sitúa en la esquina superior izquierda del contexto gráfico del componente en el que se dibuja, creciendo el eje X hacia la derecha y el eje Y hacia abajo, tal y como se muestra en la figura 8.13.

Por otro lado, los **ángulos** se expresan en grados sexagesimales 0º..360º, considerándose que el 0º está en la posición *este*. Los ángulos crecen en sentido antihorario, y se expresan con signo positivo; si queremos expresar un ángulo de arco en sentido horario, lo pondremos en negativo.

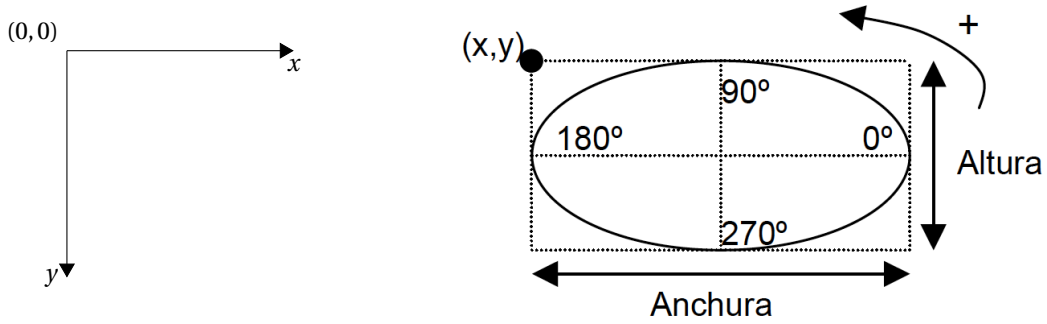



Figura 8.13: Gráficos: sistema de coordenadas

### Clase `Graphics`

La clase `Graphics` dispone de una serie de métodos para dibujar (ver tabla 8.47). Todos esos métodos se utilizan dentro de la función `paint(Graphics g)`, la cual recibe un **objeto** de la clase `Graphics`; este objeto se conoce como *contexto gráfico del dispositivo*, y se crea automáticamente, es decir, el programador no tiene que crearlo.

Más adelante se hablará algo más sobre la función `paint()`; de momento simplemente comentar que es una función que hay que escribir tal cual se muestra a continuación, dentro del componente en el que se quiere dibujar:

```
1 class xxx extends Frame {
2    ...
3    public void paint(Graphics g) {
4       // Dibujamos una cadena de texto utilizando el objeto g
5       g.drawString("Hola Mundo!",50,50);
6    }
7 }
```

Muchos de los métodos de la clase `Graphics` están duplicados como `drawXXX()` y `fillXXX()`; para abreviar los exponemos juntos puesto que se trata del mismo método, con la diferencia de que los primeros dibujan sólo el borde de la figura concreta mientras que los segundos dibujan la misma figura pero rellena del color actual.

| Método | Descripción |
|---|---|
| `clearRect(int x, int y, int width, int height)` | Borrar el rectángulo especificado. |
| `drawArc / fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)` | Dibujar un arco circular desde el ángulo inicial hasta el ángulo final del óvalo definido por el rectángulo especificado. El ángulo del arco es una cantidad tal que sumada al ángulo inicial nos determina el ángulo final. |
| `drawImage(Image img, int x, int y [, int width, int height,] ImageObserver observer)` | Dibujar una imagen, la cual, por ejemplo, ya se habrá leido desde un archivo gráfico .gif o .png. En caso de especificar el ancho y alto opcional se escalará la imagen para ajustarse a dicho tamaño. |
| `drawLine(int x1, int y1, int x2, int y2)` | Dibujar una línea recta entre los puntos (x1,y1) y (x2,y2). |
| `drawOval / fillOval(int x, int y, int width, int height)` | Dibujar una elipse contenida por el rectángulo especificado; en caso de que el ancho y el alto sean iguales se dibujará un círculo. |
| `drawPolygon / fillPolygon(int[] xPoints, int[] yPoints, int nPoints)` | Dibujar un polígono, es decir, una figura geométrica cerrada que pasa por la serie de puntos especificados. También es posible especificar un objeto `Polygon` como único argumento (consultar la documentación). |
| `drawPolyline(int[] xPoints, int[] yPoints, int nPoints)` | Dibujar una polilínea o línea con varios tramos o segmentos entre los puntos especificados. |
| `drawRect / fillRect(int x, int y, int width, int height)` | Dibujar un rectángulo con la esquina superior izquierda en las coordenadas (x,y) y el ancho y alto especificados. |
| `drawRoundRect / fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)` | Dibujar un rectángulo con los esquinas redondeadas. Los argumentos son los mismos que para un rectángulo normal, especificando también el radio de curvatura de las esquinas. |
| `drawString(String str, int x, int y)` | Escribir un texto en las coordenadas (x,y) especificadas. Se utilizará la fuente de letra actual (ver `setFont()`). |
| `getColor()` | Obtener el color actual del contexto gráfico, que es aquel con el que se dibuja en cada momento, pudiéndose cambiar con `setColor()`. |
| `getFont()` | Obtener la fuente del contexto gráfico, que es aquella con la que se dibujan los textos, pudiéndose cambiar con `setFont()`. |
| `getFontMetrics([Font f])` | Obtiene las medidas de la fuente actual (o la especificada como parámetro) del contexto gráfico. Puede ser útil para calcular lo que puede ocupar un texto de ancho o alto en pixeles. |
| `setColor(Color c)` | Establecer el color actual del contexto gráfico, que será aquel con el que se dibujará a partir de ese momento, hasta que se vuelva a cambiar (ver más adelante). |
| `setFont(Font font)` | Establecer la fuente del contexto gráfico, que será aquella con la que se dibujarán los textos a partir de ese momento, hasta que se vuelva a cambiar (ver más adelante). |
| `translate(int x, int y)` | Trasladar el origen de coordenadas al punto (x,y) especificado; a partir de ese momento, todas las coordenadas serán relativas al nuevo punto especificado. |

Cuadro 8.47: Clase `Graphics` - métodos

**Ejemplo** El siguiente ejemplo muestra el uso de varios de los métodos de la clase `Graphics`, dibujando varias figuras de distinto color, y texto con fuentes diferentes. Se puede observar que todo el código se coloca dentro del método `paint()`, el cual se invoca automáticamente al abrirse la aplicación, y cada vez que el sistema detecta que debe volver a redibujarse. Cada vez que eso sucede, se borra todo y se vuelve a dibujar de nuevo.

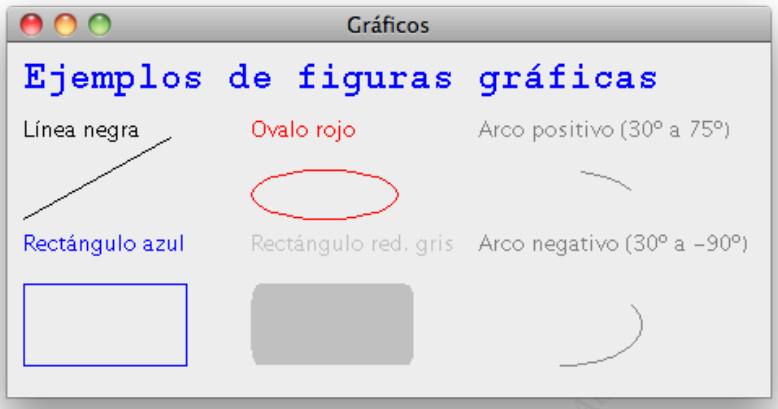



Figura 8.14: Gráficos: ejemplo de dibujo de figuras

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class Graphics1
5  {
6     public static void main (String args []) {
7        // Creamos objeto ventana y establecemos el titulo
8        Graphics1_Frame f = new Graphics1_Frame ("Gráficos");
9        // Establecemos tamaño de la ventana
10       f.setSize(470,240);
11       // Mostramos la ventana
12       f.setVisible (true);
13    }
14 }
15
16 /** Clase ventana de nuestra aplicación. */
17 class Graphics1_Frame extends Frame implements WindowListener
18 {
19    public Graphics1_Frame (String title) {
20       // Establecemos el título del formulario, llamando al padre
21       super(title);
22       // Registramos gestor de eventos de ventana
```

```
23          addWindowListener(this);
24      }
25
26      /* WindowListener */
27      public void windowClosing     (WindowEvent e) {
28          System.exit(0);  // dispose();
29      }
30      public void windowOpened      (WindowEvent e) { }
31      public void windowClosed      (WindowEvent e) { }
32      public void windowActivated   (WindowEvent e) { }
33      public void windowDeactivated (WindowEvent e) { }
34      public void windowIconified   (WindowEvent e) { }
35      public void windowDeiconified (WindowEvent e) { }
36
37      public void paint(Graphics g) {
38          Font fontInicial = g.getFont();
39          Font fontTitulo;
40
41          // Como en un Frame, el origen de coordenadas es la esquina
42          // superior izquierda de la ventana, incluyendo la barra de título
43          // desplazamos el origen de coordenadas
44          g.translate(0,30);
45
46          // Dibujamos un título con una nueva fuente de letra
47          fontTitulo = new Font("Courier", Font.BOLD + Font.ITALIC, 24);
48          g.setFont(fontTitulo);
49          g.setColor(Color.blue);
50          g.drawString("Ejemplos de figuras gráficas",10,20);
51          // Volvemos a establecer la fuente inicial
52          g.setFont(fontInicial);
53
54          // Dibujamos una línea de color negro
55          // Sintáxis: drawLine(x1,y1,x2,y2)
56          g.setColor(Color.black);
57          g.drawString("Línea negra",10,50);
58          g.drawLine(10,100,100,50);
59
60          // Dibujamos un rectángulo de color azul
61          // Sintáxis: drawRect(x,y,width,height)
62          g.setColor(Color.blue);
63          g.drawString("Rectángulo azul",10,120);
64          g.drawRect(10,140,100,50);
65
66          // Dibujamos un óvalo de color rojo
67          // Sintáxis: drawOval(x,y,width,height)
68          g.setColor(Color.red);
```

```
69          g.drawString("Ovalo rojo",150,50);
70          g.drawOval(150,70,90,30);
71
72          // Dibujamos un rectángulo redondeado y relleno
73          // Sintáxis: drawRoundRect(x,y,width,height,arcWidth,arcHeight)
74          g.setColor(Color.lightGray);
75          g.drawString("Rectángulo red. gris",150,120);
76          g.fillRoundRect(150,140,100,50,10,20);
77
78          // Dibujamos un arco en sentido antihorario (30º .. 75º).
79          // Sintáxis: drawArc(x,y,width,height,startAngle,arcAngle)
80          g.setColor(Color.gray);
81          g.drawString("Arco positivo (30º a 75º)",290,50);
82          g.drawArc(290,70,100,50,30,45);
83
84          // Dibujamos un arco en sentido horario (30º .. -90º).
85          // Notar que en este caso, el incremento de ángulo es negativo.
86          // Sintáxis: drawArc(x,y,width,height,startAngle,arcAngle)
87          g.setColor(Color.gray);
88          g.drawString("Arco negativo (30º a -90º)",290,120);
89          g.drawArc(290,140,100,50,30,-120);
90      }
91  }
```

Más adelante se ampliará información sobre el uso de colores y fuentes de texto. Antes se va a explicar algo más sobre el método `paint()`.

### Método `paint()`

El método `paint()` en el que escribimos las instrucciones para dibujar es un método que se invoca automáticamente al mostrarse la ventana por primera vez, y cada vez que el sistema detecta que se debe redibujar por haberse "dañado" alguna parte de la imagen mostrada. Tal sería el caso de las siguientes situaciones:

- cuando la ventana se ha minimizado y luego se vuelve a maximizar.
- cuando la ventana cambia de tamaño.
- cuando parte de la ventana se oculta por alguna otra ventana y luego se vuelve a activar.

Cada vez que se ejecuta de nuevo el método `paint()` se borra todo el contenido anterior; esto, en realidad no lo hace la función `paint()` sino otra función que se invoca previamente, el método `update()`.

Sin embargo, hay ocasiones en las que nos puede interesar forzar un refresco de la imagen, por ejemplo, si el programa requiere dibujar alguna nueva figura o imagen, ya que el sistema no detectará que debe redibujarla. Puesto que no podemos invocar manualmente a la función `paint()` al recibir un objeto `Graphics` que puede que no tengamos, existe un método para ello, el método `repaint()`, el cual no recibe ningún parámetro. En realidad, `repaint()` invoca al método `update()` para borrar la ventana, y éste, a su vez, a `paint()`, aunque esto es transparente para el programador.

```
1     ...
2     public void actionPerformed() {
3         ...
4         // Forzamos un refresco de la imagen
5         repaint();
6         ...
7     }
8
9     public void paint(Graphics g) {
10        ...
11    }
12    ...
```

**Ejemplo** El siguiente ejemplo muestra el uso del método `repaint()`, el cual es necesario invocar manualmente. Cada vez que el usuario hace click con el ratón en algún lugar, se dibujará un nuevo punto. Los puntos se van almacenando en un array de tamaño limitado de objetos `Point`, que son objetos que contienen coordenadas (x,y). En la función `paint()` se recorre dicho array y se dibuja un punto, pero es necesario forzar el refresco mediante `repaint()` para que se redibuje todo de nuevo. También sería el caso de pulsar el botón de borrar todos los puntos, que lo que hace es reiniciar la cuenta de puntos, y forzar el refresco para que se borre inmediatamente la ventana. De lo contrario, tendríamos que provocar una de las situaciones mencionadas en las que se redibuja automáticamente, como por ejemplo, minimizar y maximizar de nuevo. Para completar el ejemplo, haremos que conforme vayamos moviendo el ratón nos indique las coordenadas del cursor en la parte inferior derecha.

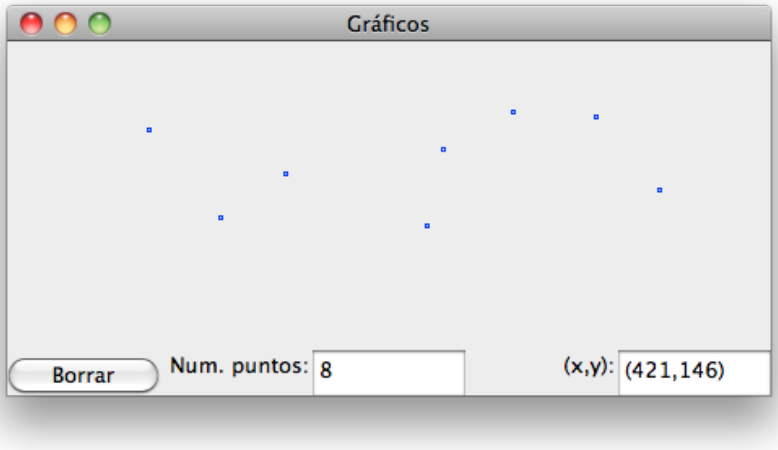



Figura 8.15: Gráficos: ejemplo de invocación a `repaint()`

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class Graphics2
5  {
6     public static void main (String args []) {
7        // Creamos objeto ventana y establecemos el titulo
8        Graphics2_Frame f = new Graphics2_Frame ("Gráficos");
9        // Establecemos tamaño de la ventana
10       f.setSize(470,240);
11       // Mostramos la ventana
12       f.setVisible (true);
13    }
14 }
15
16 class Graphics2_Frame extends Frame implements WindowListener, ActionListener,
17                                                 MouseListener, MouseMotionListener
18 {
19    /** Numero maximo de puntos */
20    private static final int NUM_MAX_PUNTOS = 20;
21
22    Panel     sur;
23    Button    borrar;
24    Label     lblNumPuntos, lblCoord;
25    TextField txtNumPuntos, txtCoord;
26
27    /** Lista de puntos */
28    Point puntos[];
29    /** Numero de puntos */
30    int numPuntos;
31
32    public Graphics2_Frame (String titulo) {
33       super(titulo);
34       setLayout(new BorderLayout());
35
36       // Panel sur
37       sur = new Panel();
38       add (sur, BorderLayout.SOUTH);
39       sur.setLayout(new GridLayout(1,5));
40
41       borrar = new Button("Borrar");
42       sur.add (borrar);
43       borrar.addActionListener (this);
44
45       lblNumPuntos = new Label("Num. puntos:", Label.RIGHT);
```

```
46          txtNumPuntos = new TextField();
47          txtNumPuntos.setEditable (false);
48          sur.add (lblNumPuntos);
49          sur.add (txtNumPuntos);
50
51          lblCoord = new Label("(x,y):", Label.RIGHT);
52          txtCoord = new TextField();
53          txtCoord.setEditable (false);
54          sur.add (lblCoord);
55          sur.add (txtCoord);
56
57          // Lista de puntos
58          puntos = new Point[NUM_MAX_PUNTOS];
59          numPuntos = 0;
60
61          addWindowListener(this);
62          addMouseListener (this);
63          addMouseMotionListener (this);
64      }
65
66      /* WindowListener */
67      public void windowClosing    (WindowEvent e) {
68          System.exit(0);  // dispose();
69      }
70      public void windowOpened     (WindowEvent e) { }
71      public void windowClosed     (WindowEvent e) { }
72      public void windowActivated  (WindowEvent e) { }
73      public void windowDeactivated(WindowEvent e) { }
74      public void windowIconified  (WindowEvent e) { }
75      public void windowDeiconified(WindowEvent e) { }
76
77      /* ActionListener */
78      public void actionPerformed (ActionEvent e) {
79          if (e.getSource() == borrar) {
80              numPuntos = 0;
81              repaint();
82          }
83      }
84
85      /* MouseListener: eventos de pulsaciones botones del ratón */
86      public void mouseClicked(MouseEvent e) {
87          Point p = new Point (e.getX(), e.getY());
88          if (numPuntos < NUM_MAX_PUNTOS) {
89              puntos[numPuntos] = p;
90              numPuntos++;
91              repaint();
```

```
92          }
93      }
94      public void mousePressed (MouseEvent e)  {}
95      public void mouseReleased(MouseEvent e) {}
96      public void mouseEntered (MouseEvent e)  {}
97      public void mouseExited  (MouseEvent e)   {}
98
99      /* MouseMotionListener: eventos de movimiento del ratón */
100     public void mouseMoved(MouseEvent e) {
101         txtCoord.setText( "(" + e.getX() + "," + e.getY() + ")" );
102         repaint();
103     }
104     public void mouseDragged(MouseEvent e) {}
105
106     /* Acciones de dibujo */
107     public void paint (Graphics g) {
108         g.setColor(Color.BLUE);
109         for (int i=0; i<numPuntos; i++) {
110             Point p = (Point) puntos[i];
111             g.drawRect (p.x, p.y, 2, 2);
112         }
113         // Mostramos el número de puntos
114         txtNumPuntos.setText (""+numPuntos);
115     }
116 }
```

### Colores

Anteriormente hemos visto algún ejemplo en el que la forma de cambiar el color con el que se dibuja es utilizando el método `setColor()` y especificando una de las constantes con unos **colores predefinidos** que tiene la clase `Color`, los cuales se enumeran en la tabla 8.48. Pueden escribirse tanto en mayúsculas como en minúsculas; por ejemplo `Color.BLACK` y `Color.black` hacen referencia al mismo color, aunque algunos se escriben de forma ligeramente distinta al escribirlos en minúsculas (consultar la documentación). En la tabla 8.48 utilizamos mayúsculas, puesto que suele ser lo recomendado para las constantes.

| Constante | Color | Valores RGB |
|---|---|---|
| `Color.BLACK` | Negro | 0-0-0 |
| `Color.BLUE` | Azul | 0-0-255 |
| `Color.CYAN` | Cian | 0-255-255 |
| `Color.DARK_GRAY` | Gris oscuro | 64-64-64 |
| `Color.GRAY` | Gris | 128-128-128 |
| `Color.GREEN` | Verde | 0-255-0 |
| `Color.LIGHT_GRAY` | Gris claro | 192-192-192 |
| `Color.MAGENTA` | Magenta | 255-0-255 |
| `Color.ORANGE` | Naranja | 255-200-0 |
| `Color.PINK` | Rosa | 255-175-175 |
| `Color.RED` | Rojo | 255-0-0 |
| `Color.WHITE` | Blanco | 255-255-255 |
| `Color.YELLOW` | Amarillo | 255-255-0 |

Cuadro 8.48: Clase `Color` - constantes de colores predefinidos

Sin embargo, también podemos crear otros colores nuevos, puesto que no hay muchos colores predefinidos; esto se hace creando objetos de la clase `Color`. La clase `Color` dispone de unos constructores, que se decriben en la tabla 8.49, en los que los colores se especifican en el formato RGB (*red-green-blue*), es decir, como una combinación de los tres colores básicos *rojo-verde-azul*. Por tanto, cada color es una combinación de tres valores enteros 0 a 255; también pueden escribirse como un porcentaje, con valores `float` 0.0 a 1.0.

De esta forma es posible construir cualquier otro color no contemplado en los colores predefinidos, especificando los 3 valores RGB, y opcionalmente un valor *alfa* que nos indica el nivel de transparencia. Un valor de *alfa* mínimo 0 (ó 0.0) indica que es totalmente transparente; un valor máximo 255 (ó 1.0) indica que es opaco.

**Ejemplo** El siguiente fragmento crea un nuevo color para poder pintar figuras de color marrón.

```
1    ...
2    public void paint(Graphics g) {
3       Color marron = new Color(102,0,0);
4
5       g.setColor(marron);
6       g.drawString("Este texto aparecerá de color marrón",50,50);
7    }
8    ...
```

| Constructor/Método | Descripción |
|---|---|
| `Color(float r, float g, float b [, float alfa])` | Constructor de un color con valores `float` 0.0 a 1.0. |
| `Color(int r, int g, int b [, int alfa])` | Constructor de un color con valores enteros 0 a 255. |
| `getRed()` | Obtener el valor del componente rojo del espacio RGB en el rango 0..255. |
| `getGreen()` | Obtener el valor del componente verde del espacio RGB en el rango 0..255. |
| `getBlue()` | Obtener el valor del componente azul del espacio RGB en el rango 0..255. |
| `getAlpha()` | Obtener el valor *alfa* o nivel de transparencia del color. |
| `brighter()` | Crear un nuevo color más brillante que el color actual. |
| `darker()` | Crear un nuevo color más oscuro que el color actual. |

Cuadro 8.49: Clase `Color` - constructores y métodos

Para conocer los valores RGB de un nuevo color a crear podemos hacer varias cosas:

- abrir el diálogo de color de ciertas aplicaciones, como por ejemplo MS-Office (ver figura 8.16)
- ejecutar este applet (botón *[Background color]* o *[Text color]*) [1]
- visitar este enlace.



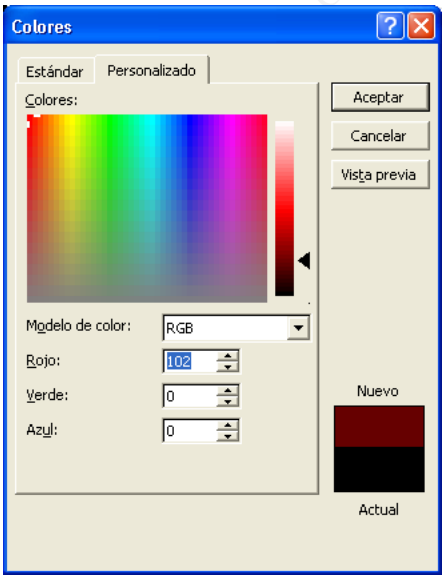



Figura 8.16: Dialogo de color de MS-Office

**NOTA**: a la hora de crear un **nuevo color** especificando los valores RGB es posible que un ordenador no pueda mostrar justamente ese color. En ese caso, se mostrará el color más cercano.

[1] NOTA: si se utiliza Internet Explorer puede que el applet no funcione; probar con otro navegador como Opera o Safari

### Fuentes de letra

Para mostrar texto en alguna parte de la ventana, se utiliza la función `drawString(texto,x,y)`. Antes de eso, podemos establecer el color y la fuente de letra con la que queremos dibujar la cadena con `setColor()` y `setFont()`, respectivamente. Con respecto a la fuente, se utiliza la clase `Font`, de la cual veremos un único constructor y varios métodos (ver tabla 8.50).

| Constructor/Método | Descripción |
|---|---|
| `Font(String name, int style, int size)` | Constructor de una fuente especificando su nombre, su estilo y un tamaño. En cuanto al estilo, se puede utilizar alguna de las constantes siguientes: `Font.BOLD` (negrita), `Font.ITALIC` (cursiva), o `Font.PLAIN` (normal). Es posible especificar varias al mismo tiempo mediante el operador suma + o el operador Or-lógico `|`, por ejemplo: `BOLD+ITALIC`. |
| `getFamily()` | Obtener el nombre de la familia de la fuente. |
| `getFontName()` | Obtener el nombre de la fuente. |
| `getName()` | Obtener el nombre lógico de la fuente. |
| `getSize()` | Obtener tamaño de la fuente. |
| `isBold()` | Averiguar si la fuente es negrita (estilo `Font.BOLD`). |
| `isItalic()` | Averiguar si la fuente es cursiva (estilo `Font.ITALIC`). |
| `isPlain()` | Averiguar si la fuente es normal (estilo `Font.PLAIN`). |

Cuadro 8.50: Clase `Font` - constructores y métodos

**Ejemplo** El siguiente fragmento establece el color rojo y cambia la fuente de letra antes de dibujar la cadena de texto.

```
1 public void paint (Graphics g) {
2    Font fuente = new Font ("Courier", Font.BOLD, 15);
3    g.setColor (Color.RED);
4    g.setFont (fuente);
5    g.drawString ("Hola", 50, 50);
6 }
```

**IMPORTANTE**: Debemos tener en cuenta que no están disponibles todas las fuentes en todas las plataformas. En caso de que una fuente especificada no esté disponible en una plataforma concreta no obtendremos un error, sino que el intérprete de Java la sustituirá por la fuente por defecto del sistema, lo cual podría afectar al aspecto de la aplicación.
Para asegurarnos de que nuestro programa sea lo más portable posible se aconseja utilizar sólo las fuentes de letra más comunes, como por ejemplo:
*Courier*, *Dialog*, *DialogInput*, *Helvética* y *TimesNewRoman*.

Si lo que nos interesa es mostrar un **texto centrado** en una ventana, necesitamos conocer:

- El ancho o alto de la ventana o panel: `getWidth()` y `getHeight()`.
- El ancho del texto a mostrar. Puesto que esto es más complicado, abajo se proporciona una función que recibe el texto a mostrar, la fuente y el contexto gráfico, y retorna lo que ocupa en pixels.

La siguiente función `anchoTexto()` calcula los **pixeles necesarios para dibujar un texto** según la fuente utilizada. Requiere que se le pase como argumento el contexto gráfico (el objeto `Graphics` que recibe el propio método `paint()`), la fuente a utilizar, y el texto que queremos dibujar. Notar que requiere importar dos clases que están dentro de dos subpaquetes de `java.awt`.

```
1  import java.awt.font.FontRenderContext;
2  import java.awt.geom.Rectangle2D
3  ...
4  public class xxx {
5     ...
6     /** Averiguar el ancho de un texto en pixels según una fuente. */
7     public int anchoTexto (Graphics g, Font f, String texto) {
8        Graphics2D g2 = (Graphics2D) g;
9        FontRenderContext frc = g2.getFontRenderContext();
10       Rectangle2D r2 = f.getStringBounds(texto, frc);
11       int width = (int) r2.getWidth();
12       return (width);
13    }
14 }
```



### Imágenes e iconos

Para **visualizar una imagen** se utiliza el método `drawImage()`, el cual recibe un objeto `Image` que ya debe haberse creado, normalmente leyéndolo desde un archivo gráfico mediante la función `getImage()`. Por rendimiento, conviene cargar las imágenes sólo una vez al principio (en el constructor de la clase `Frame` principal), y posteriormente visualizarlo con el método `drawImage()` cada vez que sea necesario dentro de la función `paint(Graphics g)` del contenedor en el que queremos mostrarlo (`Frame`, `Dialog`, `Panel`, ...).

De forma parecida, podemos cambiar el **icono de nuestra aplicación** para que no aparezca el icono por defecto de cualquier aplicación Java (una taza de café). Por ejemplo, podríamos utilizar una imagen llamada "icono.gif" como icono de la aplicación, como se muestra en el siguiente ejemplo.

**Ejemplo** El siguiente fragmento de código realiza ambas cosas, establece el icono de la aplicación y muestra una imagen, cargándolas ambas previamente en el constructor.

```
1  import java.net.*;
2
3  class MiFrame extends Frame {
4     URL   url;
5     Image imagen;
6
7     public MiFrame (String titulo) {
8        super (titulo);
9
10       Toolkit tk  = Toolkit.getDefaultToolkit();
11       // Cargamos una imagen para dibujarla después
12       url = this.getClass().getResource("imagen.png");
13       if (url != null) {
14          imagen = tk.getImage(url);
15       }
16       // Establecemos el icono de la aplicación
17       url = this.getClass().getResource("icono.gif");
18       if (url != null) {
19          Image icono = tk.getImage(url);
20          if (icono != null) this.setIconImage(icono);
21       }
22    }
23
24    public void paint (Graphics g) {
25       if (image != null) {
26          g.drawImage (image, 50, 50, this);
27       }
28    }
29 }
```

No debemos olvidar importar el paquete `java.net.*`, o al menos la clase `URL`, ya que es necesaria. Por otro lado, no es necesario insertar ningún componente ni panel para poder dibujar; toda la zona del `Frame` no ocupada por componentes es la que quedará disponible para dibujar.

Los archivos de imágenes deben estar en el directorio donde residan los archivos .class, por debajo del directorio *build* y el directorio correspondiente al paquete (si se utiliza). Debe además aparecer en la solapa *Files* del entorno de NetBeans. Si hay que mover dicho archivo, se aconseja hacerlo desde dicho explorador de la solapa *Files* en lugar de hacerlo desde el explorador de archivos de Windows.

IMPORTANTE: No se admiten todos los formatos de archivos gráficos. A la hora de utilizar la función `getImage()` se admiten archivos GIF, JPEG o PNG, aunque a partir de Java 5.0 también podemos cargar los típicos archivos de mapa de bits de Windows BMP y WBMP.

### Reproducir sonidos

Para reproducir un sonido podemos utilizar una de las dos posibilidades que se muestran a continuación.

```
1  // Al principio del fichero (después de package)
2  import java.applet.*;           // AudioClip
3
4  class MiFrame extends Frame {
5
6     // Dentro de una función ...
7     void reproducirSonido_v1 () {
8        /* Si el sonido reside en el directorio del programa: "flash.wav"
9         * Si el sonido reside en un subdirectorio 'sonidos' por
10        *       debajo del directorio del programa: "/sonidos/flash.wav"
11        * Si el sonido reside en el directorio raiz: "C:\\flash.wav"  */
12       URL url = this.getClass().getResource("flash.wav");
13       if (url != null) {
14          AudioClip sonido = Applet.newAudioClip(url);
15          if (sonido != null) sonido.play();
16       }
17    }
18
19    // Dentro de una función ...
20    void reproducirSonido_v2 () {
21       URL url = null;
22
23       try {
24          url = new URL("file", "localhost", "flash.wav");
25       } catch (MalformedURLException e) {
26          System.out.println("No se encuentra archivo de sonido");
27       }
28       AudioClip sonido = Applet.newAudioClip(url);
29       if (sonido != null) sonido.play();
30    }
31 }
```

## 8.10. Swing

A lo largo de todo el capítulo se ha trabajado con la librería gráfica *Abstract Window Toolkit* (AWT), pero existe la librería Swing, la cual está también intregada en el propio Java Development Kit (JDK). Existe alguna más, como la librería *Standard Widget Toolkit* (SWT) incluida en el entorno de desarrollo (IDE) Eclipse. El entorno de desarrollo NetBeans dispone de un diseñador gráfico de ventanas, el cual puede generar código automáticamente tanto para AWT como Swing, que son los más estándar.

En este apartado vamos a describir muy brevemente la librería Swing, presentando una comparativa con respecto AWT para facilitar la migración desde AWT.

### Diferencias entre AWT y Swing

Aunque Swing se basa en AWT, algunas **diferencias** entre ambas librerías serían las siguientes:

- Los componentes AWT consumen muchos más **recursos** que los Swing, ya que en AWT cada componente utiliza su propio gestor de eventos, mientras que en Swing, tan sólo hay un gestor de eventos para toda la ventana. Por ello, a los componentes AWT se les llama *peso-pesado* (*heavy-weight*), mientras que a los componentes Swing se les llama *peso-ligero* (*light-weight*).
- Swing permite cambiar la **apariencia** de la aplicación: AWT utiliza componentes propios del sistema, por lo que las aplicaciones AWT siempre tendrán el mismo aspecto que el resto de aplicaciones de la plataforma en la que se ejecutan. Por otro lado, puesto que los componentes Swing son redibujados (excepto las ventanas de mayor nivel), una aplicación podría tener un aspecto diferente (apariencia), por ejemplo, una aplicación con aspecto de Unix ejecutándose sobre un sistema operativo Windows, aunque esto no suele utilizarse.
- Tanto AWT como Swing comparten muchas características de las que se han explicado, como los gestores de esquemas, la gestión de eventos, los gráficos, etc. Sin embargo, Swing tiene un mayor repertorio de componentes, añade nuevos gestores de esquemas, tiene nuevos tipos de eventos, y ofrece más diálogos predefinidos (por ejemplo, para seleccionar un color o una fuente, o pedir confirmación o algún valor al usuario).
- Como inconvenientes de ambos, AWT tiene problemas de portabilidad, es decir, es necesario testear una aplicación en diferentes plataformas para comprobar que su aspecto es el correcto. En cuanto a Swing, es necesario aprender más cosas.

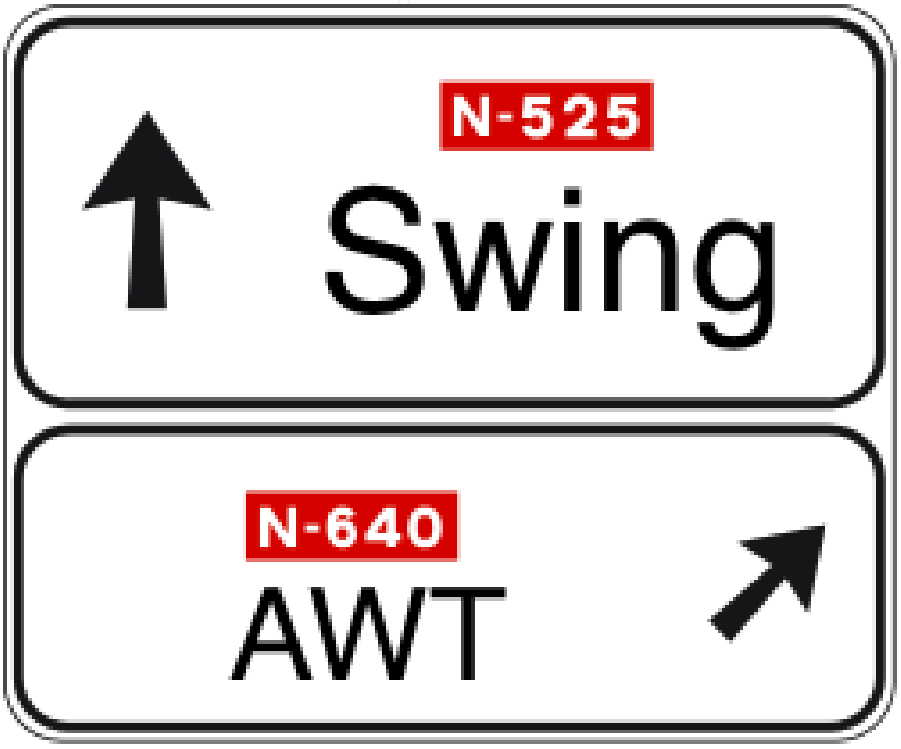
### Migración desde AWT a Swing

Por las razones expuestas, se aconseja utilizar Swing. No obstante, el haber aprendido AWT no es inconveniente, puesto que Swing se basa en AWT y, como se ha explicado, tienen muchas cosas en común. Desde el punto de vista del programador, para **migrar desde AWT a Swing** hay que tener en cuenta lo siguiente:

- Es posible tener una aplicación con ventanas que utilicen AWT y otras que utilicen Swing. No sólo eso, es posible también que coexistan en una misma ventana componentes AWT y Swing, por lo que la migración puede ir haciéndose de forma progresiva.

- Los componentes existentes en Swing tienen nombre muy parecido a los de AWT, en muchos casos basta con poner delante del nombre una letra `'J'`. Por ejemplo, en AWT tenemos un `Button` o `Panel`, y en Swing les corresponde un `JButton` o `JPanel`, aunque éstos tienen más características, como la posibilidad de mostrar un icono sobre un botón (cosa que no se permitía en AWT de forma directa).
- En AWT podemos añadir componentes directamente a un `Frame` o un `Dialog`, y también a otros paneles (`Panel`), mientras que en Swing es necesario obtener siempre su panel contenedor mediante la función `getContentPane()`, y es en ese panel (`JPanel` en este caso) donde ya se añadirían los componentes de la forma habitual.





# GLOSARIO DE ACRÓNIMOS

| Siglas | Significado |
|---|---|
| ANSI | American National Standards Institute |
| AOT | Ahead-Of-Time (técnica de compilación) |
| API | Application Programming Interface |
| ASCII | American Standard Code for Information Interchange |
| AWT | Abstract Window Toolkit |
| CISC | Complex Instruction Set Computing |
| DBMS | DataBase Management System |
| DLL | Dynamic Linked Library |
| ETSI | European Telecommunications Standards Institute |
| GNU | GNU Not Unix |
| GPL | GNU General Public License |
| GUI | Graphical User Interface |
| IDE | Integrated Development Environment |
| IEEE | Institute of Electrical and Electronic Engineer |
| ISO | International Standards Organization |
| JAR | Java ARchive |
| J2SE | Java 2 Platform, Standard Edition (~ Java SE) |
| J2EE | Java 2 Platform, Enterprise Edition (~ Java EE) |
| J2ME | Java 2 Platform, Micro Edition (~ Java ME) |
| JCP | Java Community Process |
| JDBC | Java DataBase Conectivity |
| JDK | Java Development Kit (~ Java 2 SDK y J2SDK) |
| JFC | Java Foundation Classes |
| JIT | Just-In-Time (técnica de compilación) |
| JNDI | Java Naming and Directory Interface |
| JNI | Java Native Interface |
| JRE | Java Runtime Environment |
| JVM | Java Virtual Machine |
| KNI | K Native Interface (~ JNI) |
| ODBC | Open DataBase Connectivity |
| OSI | International Organization of Standarization |
| PC | Personal Computer |

| | |
|---|---|
| PDA | Personal Digital/Data Assistant |
| RDBMS | Relational DataBase Management System |
| RISC | Reduced Instruction Set Computing |
| ROM | Read Only Memory |
| SDK | Standard Development Kit |
| SQL | Structured Query Language |
| SRAM | Static Random Access Memory |
| URL | Uniform Resource Locator |
| UTF-16 | Unicode Transformation Format (16-bit) |
| WORA | Write Once, Run Anywhere |