Interfaces gráficas

Programación de interfaces gráficas en la actualidad

Cuando se presentó Java, este tenía una biblioteca de clases que Sun Microsystem denominó AWT (Abstract Window Toolkit), para programar interfaces gráficas de usuario.

La biblioteca AWT aborda los elementos de la interfaz, delegando la creación y comportamiento a la máquina donde vaya a ejecutarse el código.

Con aplicaciones sencillas funcionaba bastante bien, pero a la hora de construir programas más complejos, los elementos tales como ventanas, barras de menús, barras de desplazamiento, botones etc, tenían un comportamiento diferente en distintas plataformas rompiendo la regla más importante de Java: la portabilidad, ya saben, “write once, run everywhere”.

Para solucionar este problema se creó otra biblioteca de clases más potente, la biblioteca **Swing**.

Los elementos de la biblioteca Swing tardan más en cargar que los elementos de la biblioteca AWT, pero esto en las máquinas de hoy en día no supone ningún problema.

Las ventajas de utilizar Swing podrían ser las siguientes:

- Swing posee un conjunto de elementos de interfaz de usuario rico y cómodo.
- Swing tiene pocas dependencias de la plataforma donde se ejecuten los programas.
- Swing es coherente entre plataformas.

Sin embargo, en el momento de escribir estas líneas, Oracle (actual dueño de la tecnología Java) no realiza ningún tipo de actualizaciones en la biblioteca Swing. Si bien no se considera a Swing una tecnología obsoleta (a día de hoy. Es probable que en un futuro cercano lo sea), Oracle ha desarrollado una biblioteca de clases más potente y acorde con las necesidades de hoy en día: **JavaFx.**

Las principales características que ofrece **JavaFx** en el desarrollo de interfaces gráficas son las siguientes:

- Permite integrar gráficos vectoriales, animación, sonido y vídeo.
- Se puede combinar con cualquier otra biblioteca de Java
- Separa de forma clara la parte del diseño de una aplicación de la parte del desarrollo. De esta forma los diseñadores pueden trabajar en el diseño sin interferir en el desarrollo de la aplicación y viceversa, los desarrolladores podrán trabajar sin interferir en el diseño.

Como en el momento de escribir estas líneas Swing se considera parte del estándar de Java y se sigue utilizando plenamente, en este manual comenzaremos por ver todo lo referente a esta biblioteca, dejando JavaFx para más adelante en este manual.

Creación de un JFrame o marco:

Las ventanas reciben el nombre de Marco en Java y son la base de cualquier interfaz gráfica. Haciendo un símil un marco es a la interfaz gráfica lo que el lienzo a una pintura. El JFrame o marco es el soporte de todo lo que alberga una interfaz gráfica: botones, menús desplegables, cuadros de texto, casillas de verificación etc.

La biblioteca AWT posee una clase llamada Frame para la construcción de este tipo de objetos. La versión Swing para la construcción de ventanas es **JFrame**. Todos los componentes pertenecientes a esta clase tales como botones por ejemplo empiezan por J, como por ejemplo JButton, JTextArea, JPanel etc.

El siguiente ejemplo muestra un marco o JFrame vacío en pantalla:

```
import javax.swing.*;

public class CreandoMarcos {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        MiMarco marco1=new MiMarco();

        marco1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        marco1.setVisible(true);

    }

}


class MiMarco extends JFrame{

    public MiMarco(){

        setSize(500,300);

        setLocation(500,300);

        setTitle("Mi ventana");

    }

}
```

Es muy importante revisar este primer programa línea por línea:

Las clases de Swing se encuentran en el paquete **javax.swing**. El nombre del paquete, javax, indica que se trata de un paquete de extensión Java, y no de un paquete de base.

Por defecto, los marcos tienen un tamaño de 0 x 0 píxeles. Al definir la subclase MiMarco se define en su constructor el tamaño que tendrá el marco o JFrame: 500 x 300. Se utiliza para ello el método **setSize()** perteneciente a la clase JFrame, estableciendo la medida en píxeles, siendo

500 el tamaño horizontal (eje x) y 300 el tamaño vertical (eje y). En el método main de la clase CreandoMarcos, empezamos por construir un objeto de tipo MiMarco.

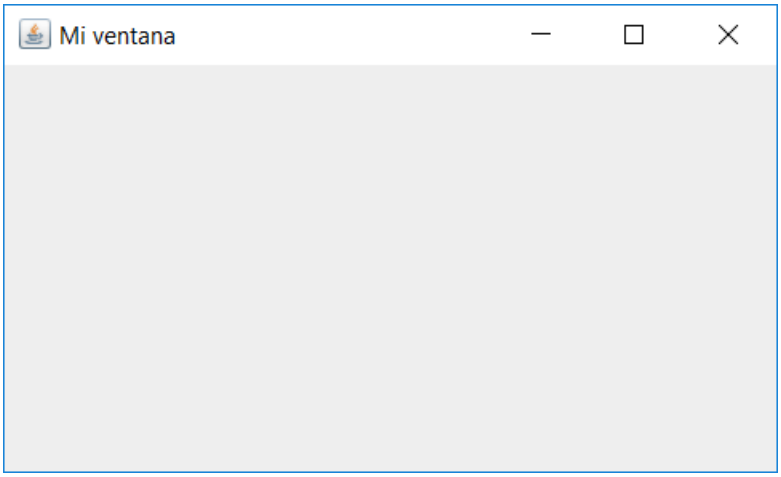
A continuación, se define lo que tiene que ocurrir si se cierra este marco. Para este programa en particular lo que queremos es que el programa concluya. Para este comportamiento empleamos la sentencia **marco1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);**

En otros programas con más marcos seguramente no interese que el programa finalice solo porque se cierre uno de los marcos. De forma predeterminada el marco se oculta cuando el usuario lo cierra, pero el programa no concluye.

Los marcos o JFrames por defecto aparecen invisibles. Esta característica permite añadir al marco elementos como botones, barras etc y no mostrarlas hasta que aparezca el marco. Para mostrar el marco, el método main llama al método setVisible del marco pasándole un parámetro true. Es muy importante que esta instrucción **setVisible(true)** esté siempre al final después de todas las instrucciones. De otra forma, el marco no mostrara su contenido al iniciar el programa. Esto se debe a una cuestión de "refresco" de pantalla ya que una vez que se ha hecho visible un marco, este no podrá mostrar después contenido a no ser que se utilice una instrucción de actualización o refresco de pantalla de forma intencionada.

El método main concluye. Al salir del main no salimos del programa, sino del hilo principal.

Al ejecutar el programa en una plataforma Windows el aspecto que esta muestra es el de la clásica ventana Windows con los elementos característicos. Si se ejecutase en otra plataforma, el aspecto del marco variaría.



Aspecto del JFrame o marco en Windows

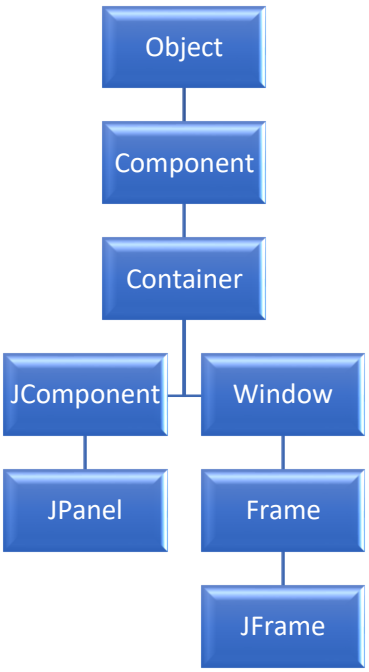
Colocación de un marco

Gracias a la herencia, la mayoría de los métodos que sirven para trabajar con el tamaño y la posición del marco provienen de las distintas superclases de JFrame. Los métodos más importantes son los siguientes:

- **dispose**: cierra la ventana y recupera recursos.
- **setIconImage**: admite por parámetro un objeto de tipo Image y lo utiliza como icono cuando minimiza la ventana.

- **setTitle**: cambia el texto que aparece en la barra del título.
- **setResizable**: admite por parámetro un boolean y determina si el usuario podrá cambiar el tamaño del marco.

Herencia de JFrame:



La clase Component y la clase Window es donde se deben buscar los métodos necesarios para modificar el tamaño y la forma de los marcos. El método **setLocation** de la clase Component es una forma de modificar la ubicación de un componente con la llamada **setLocation(500, 300)**.

La esquina superior izquierda se sitúa a 500 píxeles del borde izquierdo y a 300 píxeles del borde superior, siendo (0,0) la esquina superior izquierda de la pantalla.

El método **setBounds** de Component permite modificar el tamaño y ubicación de un componente (en particular de un JFrame) en un solo paso con la siguiente sentencia:

setBounds (x, y, anchura, altura);

En nuestro programa de ejemplo, quedaría así:

```
class MiMarco extends JFrame{

    public MiMarco(){

        setBounds(500,300,500,300);

        setTitle("Mi ventana");

    }

}
```

A continuación, se expone un ejemplo interesante para examinar línea a línea. Sería un buen ejercicio de aprendizaje que el alumno consultara la API de Java para buscar todos aquellos métodos y clases nuevos:

```
package test;

import java.awt.*;

import javax.swing.*;

public class EjemploMarcoCentrado {

    public static void main (String args[])  {

        MarcoCentrado marco = new MarcoCentrado();

        marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        marco.setVisible(true);

        }

}
```

(continúa en la siguiente página)

```
class MarcoCentrado extends JFrame {

    public MarcoCentrado()  {

        //  se obtienen las dimensiones de la pantalla

        Toolkit kit = Toolkit.getDefaultToolkit();

        Dimension tamanopantalla = kit.getScreenSize();

        int alturaPantalla = tamanopantalla.height;

        int anchuraPantalla = tamanopantalla.width;

        //  se centra el marco en la pantalla

        setSize(anchuraPantalla / 2, alturaPantalla / 2);

        setLocation (anchuraPantalla / 4, alturaPantalla / 4);

        //  se especifica el titulo e icono del marco

        Image img = kit.getImage("icono.gif");

        setIconImage (img);

        setTitle("Marco centrado");

        }

    }
```

Visualización de la información en una lámina o JPanel

En Java, los marcos están diseñados para ser contenedores de componentes como las barras de menú y otros elementos.

Cuando se quiere añadir un elemento de texto en un marco se hace a través de otro componente, **la lámina o JPanel**, que se añadirá al marco.

Cuando se diseña un marco, se van añadiendo elementos a la lámina de contenido con un código similar al siguiente:

```
Container laminaDeContenido=marco.getContentPane();

Component c=new JButton();

laminaDeContenido.add(c);
```

Deseamos añadir una sola lámina al marco en que vamos a dibujar nuestro mensaje. Las láminas se implementan mediante la clase **JPanel**. Se trata de elementos de interfaz de usuario que poseen dos características útiles:

- Tienen una superficie en la que se puede dibujar.
- Son a su vez contenedores.

De esta forma pueden contener otros elementos de interfaz como botones, barras de desplazamiento etc.

Para dibujar sobre una lámina:

- Se define una clase que extiende a **JPanel**.
- Se invalida el método **PaintComponent** de esa clase.

El método PaintComponent se encuentra en realidad en JComponent, que es la superclase de todos los componentes de Swing que no son ventanas. Admite un parámetro de tipo **Graphics**. Los objetos de tipo Graphics recuerdan una colección de ajustes para dibujar imágenes y texto, tipo de letra, color etc.

Todos los dibujos que se hagan en Java tienen que pasar por un objeto de tipo Graphics. La clase Graphics posee métodos para dibujar patrones imágenes y texto. La forma de crear una lámina en la que dibujar podría ser la siguiente:

```
class MiLamina extends JPanel  {

    public void paintComponent(Graphics g)  {

//  aquí iría el código para dibujar

        }
}
```

El método **paintComponent** admite un solo parámetro de tipo Graphics. Las medidas de los objetos de tipo Graphics se hacen en píxeles. La coordenada (0,0) es la esquina superior izquierda.

La visualización de texto se considera un caso especial de dibujo. La clase Graphics posee un método que es **drawsString** que tiene la siguiente sintaxis: **g.drawString(texto, x, y);**

Ejemplo: (en página siguiente)

```
class LaminaHolaMundo extends JPanel  {

    public void paintComponent (Graphics g) {

        super.paintComponent(g);

        g.drawString ("Hola Mundo ", X_MENSAJE, Y_MENSAJE );

    }

    public static final int X_MENSAJE = 75;

    public static final int Y_MENSAJE = 100;
}
```

La clase **JPanel**, que es la superclase de LaminaHolaMundo tiene su forma de dibujar una lámina. Por esta razón utilizamos la instrucción **super.paintComponent(g)**, llamando al constructor de la clase JPanel.

Ejemplo completo:

```
package test;

import javax.swing.*;

import java.awt.*;

public class HolaMundo  {

    public static void main (String args[])  {

        MarcoHolaMundo marco = new MarcoHolaMundo();

        marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        marco.setVisible(true);

        }
    }


class MarcoHolaMundo extends JFrame  {

    public MarcoHolaMundo ()  {

        setTitle ("Hola Mundo");

        setSize(300, 200);

        //   añadimos al marco una lámina

        LaminaHolaMundo lamina = new LaminaHolaMundo();

        add(lamina);
    }
}
```

```
//  una lámina que muestra un mensaje
class LaminaHolaMundo extends JPanel  {
    public void paintComponent (Graphics g) {
        super.paintComponent(g);
        g.drawString ("Hola Mundo ", 75, 100 );
    }

}
```

Trabajando con formas 2D

Para dibujar formas en la biblioteca 2D en Java, es necesario obtener un objeto de tipo **Graphics2D**. Esta clase es una subclase de la clase Graphics. Los métodos recibirán un objeto de tipo Graphics2D:

```
public void paintComponent (Graphics g) {
    Graphics2D g2 = (Graphics2D) g;
    // aquí iría el resto del código
}
```

La biblioteca Java2D organiza las formas geométricas de una manera orientada a objetos. Hay clases para representar líneas, rectángulos y elipses. Ejemplo de estas clases son:

- **Line2D**
- **Rectangle2D**
- **Ellipse2D**

Estas clases implementan la interfaz **Shape.**

Para dibujar una forma lo primero que se hace es crear un objeto de una clase que implementa la interfaz Shape y después se hace una llamada al método draw de la clase Graphics2D.

Ejemplo:

```
public void paintComponent (Graphics g) {

    Graphics2D g2 = (Graphics2D) g;

    Rectangle2D rectangulo=new Rectangle2D.Double(100,100,200,150);

    g2.draw(rectangulo);

    // aquí iría el resto del código

}
```

Las formas de Java2D utilizan coordenadas en coma flotante. Sin embargo, la manipulación de valores float es incómoda para el programador en ciertas ocasiones, porque el lenguaje Java es tajante en lo que se refiere en la conversión de valores double en float.

Como las conversiones son una molestia, los diseñadores de la biblioteca Java2D decidieron proporcionar dos versiones de cada clase de forma: una con las coordenadas float y otra con las coordenadas double.

Ejemplo:

```
package test;

import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class EjemploDibujo  {

    public static void main (String args[])  {

        MarcoDeDibujo marco = new MarcoDeDibujo();

        marco.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        marco.setVisible(true);

        }
    }
```

(Continúa en la página siguiente)

```
class MarcoDeDibujo extends JFrame  {

    public MarcoDeDibujo()  {

        setTitle("Prueba de Dibujo");

        setSize(400, 400);

//   se añade la lámina al marco

        LaminaDeDibujo lamina=new LaminaDeDibujo();

        add(lamina);

    }

}

class LaminaDeDibujo extends JPanel  {

    public void paintComponent(Graphics g)  {

        super.paintComponent(g);

        Graphics2D g2= (Graphics2D) g;

//  dibujar un rectángulo

        double XIzquierda = 100;

        double Ysuperior = 100;

        double anchura = 200;

        double altura = 150;

        Rectangle2D rectangulo = new Rectangle2D.Double(XIzquierda,
                Ysuperior, anchura, altura);

        g2.draw(rectangulo);

//  dibujar elipse

        Ellipse2D elipse= new Ellipse2D.Double();

        elipse.setFrame(rectangulo);

        g2.draw(new Line2D.Double(XIzquierda,   Ysuperior,  XIzquierda  +
                anchura, Ysuperior + altura));
```

(Continúa en la página siguiente)

```
//  dibujar un círculo con el mismo centro
        double xCentro = rectangulo.getCenterX();

        double yCentro = rectangulo.getCenterY();

        double radio = 150;

        Ellipse2D circulo= new Ellipse2D.Double();

        circulo.setFrameFromCenter(xCentro, yCentro,
                xCentro+radio, yCentro + radio);

        g2.draw(circulo);

    }
}
```

Uso del color

El método **setPaint()** de la clase Graphics2D permite seleccionar un color que se utilizará para todas las operaciones de trazado.

Para dibujar en múltiples colores, se selecciona un color, se dibuja, se selecciona después otro color y se dibuja de nuevo.

Los colores se definen mediante la clase Color. La clase java.awt.Color ofrece constantes para los trece colores estándar que se enumeran a continuación:

BLACK	**GREEN**	**RED**
BLUE	**LIGHT_GRAY**	**WHITE**
CYAN	**MAGENTA**	**YELLOW**
DARK_GRAY	**ORANGE**	**GRAY**
PINK		

Ejemplo:

```
Graphics2D g2= (Graphics2D) g;

g2.setPaint(Color.RED);

g2.drawString("Atención", 100, 100);
```

Se puede especificar un color o personalizar un color creando un objeto de tipo Color mediante sus componentes de rojo, verde y azul. Si utilizamos una escala de 0 a 255 para cada componente, la llamada al constructor será como sigue:

```
g2.setPaint(new Color(0,128,128)); //   un verde azulado
g2.drawString("Bienvenido", 75, 125);
```

Si se desea especificar un color de fondo, utilizaremos el método **setBackground()** de la clase Component, que es un predecesor JPanel.

```
MiLamina p = new MiLamina();
p.setBackground(Color.PINK);
```

También existe el método **setForeground()**. Establece el color predeterminado que se utilizará para dibujar el componente.

Llenado de formas

Se puede rellenar el interior de las formas cerradas con un color.

```
Rectangle2D rectangulo=new Rectangle(100,150,300,250);
g2.setPaint(Color.RED);
g2.fill(rectangulo); // rellena de color rojo el rectángulo
```

Uso de fuentes especiales para texto

Con frecuencia es necesario mostrar un texto con una fuente diferente a la predeterminada. El tipo de letra se especifica mediante el nombre de la fuente.

Para determinar cuáles son los tipos de letra disponibles en una determinada computadora, se llama al método **getAvailableFontFamilyNames()** de la clase **GraphicsEnvironment**. El método proporciona una matriz de cadenas que contiene los nombres de todos los tipos de letras disponibles.

Para obtener un ejemplar de la clase GraphicsEnvironment que describa el entorno gráfico del sistema del usuario, se utiliza el método estático **getLocalGraphicsEnvironment()**.

Ejemplo:

```
import java.awt.*;

public class ListaDeFuentes  {

    public static void main(String args[])  {

        String [] nombresDeFuentes = GraphicsEnvironment.getLocalGraphicsEnvironment().getAvailableFontFamilyNames();

        for (String nombreDeFuente : nombresDeFuentes)

            System.out.println(nombreDeFuente);
    }
}
```

Para dibujar caracteres empleando un determinado tipo de letra lo primero es crear un objeto de la clase Font. Se especifica el tipo, el tamaño y el estilo:

```
Font letra=new Font("Helvetica", Font.BOLD, 14); // El tercer argumento es el tamaño
                                                 // de la letra en puntos
```

Uso de imágenes

Las imágenes una vez que están guardadas en algún directorio del ordenador o en algún lugar de Internet, se pueden leer desde una aplicación Java empleando objetos de tipo **Graphics**.

Si la imagen está almacenada en un objeto local, se hace la llamada:

```
String nombrefichero="icono.jpg";

try {

    Image imagen = ImageIO.read (new URL("www.pildorasinformaticas.es"));

} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Ahora la variable imagen contiene una referencia a un objeto que encapsula los datos de la imagen. Se puede visualizar la imagen empleando el método **drawImage()** de la clase Graphics:

```
        String nombrefichero="icono.jpg";

        Image imagen=null;

        try {

            imagen = ImageIO.read (new URL("www.pildorasinformaticas.es"));

        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        g.drawImage (imagen, 50, 30, null);


    }
```