ACTIVIDAD GUIADA 1
Creación de dibujos y componentes genéricos

Creación de componentes a partir de componentes concretos 2
Mostrando dibujos 5
Método paintComponent() 5
Comandos gráficos 5
Una aplicación gráfica sencilla 6
Creación de componentes a partir de componentes genéricos 8

En la primera unidad hemos aprendido a utilizar los principales componentes que ofrece Swing, así como distribuirlos en la interfaz y responder a acciones de usuario utilizando el modelo de eventos.

Si bien la jerarquía de componentes de Swing es muy potente, en muchos casos nos va a interesar modificar el aspecto que presentan los elementos para adaptarlos a las características de nuestra aplicación o incluso para añadir nuevas funcionalidades a los mismos.

En esta unidad vamos a ver cómo crear componentes propios, totalmente equivalentes a los componentes estándar de la API Swing y capaces de generar eventos de alto nivel según la funcionalidad de la aplicación concreta.

A la hora de crear componentes a medida tenemos dos opciones:

- *Crear el componente a partir de un componente concreto*
- *Crear el componente a partir de un componente genérico.*

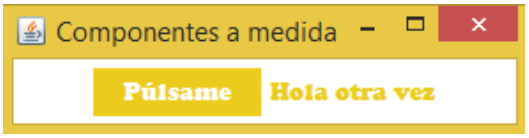
Antes de ver cómo crear componentes personalizados, vamos a ver cómo podemos realizar dibujos simples en Java, pues utilizaremos los conceptos aquí explicados más adelante.

Creación de componentes a partir de componentes concretos

El procedimiento en este caso consiste en crear una subclase de la clase componente que deseemos (**`JButton`**, **`JLabel`**, etc.) y utilizar directamente dicha subclase en nuestra aplicación.

El ejemplo mostrado a continuación crea los siguientes elementos a medida:

- Un botón que llamaremos `BotonMedida`
- Una etiqueta que llamaremos `EtiquetaMedida`
- Un marco que llamaremos `FrameMedida`



Además crearemos un nuevo marco que hace uso del marco a medida.
A continuación se muestra el código del botón:

```
public class BotonMedida extends JButton implements MouseListener{
    public BotonMedida(){
      super();
      estableceAspecto();
    }

    public BotonMedida(String texto){
        super(texto);
        estableceAspecto();
    }

    private void estableceAspecto(){
        setFont(new Font("Cooper Black", Font.PLAIN, 14));
        setForeground(Color.WHITE);
        setBackground(new Color(235,204,28));

        setBorderPainted(false);
        setFocusPainted(false);
        addMouseListener(this);
    }
    @Override
    public void mouseClicked(MouseEvent me) {}

    @Override
    public void mousePressed(MouseEvent me) {}

    @Override
    public void mouseReleased(MouseEvent me) {}

    @Override
    public void mouseEntered(MouseEvent me) {
        setBackground(new Color(255,217,3));
    }

    @Override
    public void mouseExited(MouseEvent me) {
        setBackground(new Color(235,204,28));
    }
}
```

Además de los aspectos relativos a color de fondo y tipografía, se determinan los siguientes elementos:

- **`setBorderPainted(false)`**. Determinamos que el botón no utiliza borde.
- **`setFocusPainted(false)`**. Determinamos que no queremos que el borde del botón cambie cuando el elemento tiene el foco.

Por otro lado, al hacer que el propio componente sea un escuchador de ratón, podemos responder a los eventos de ratón. En este caso hacemos que el fondo del componente cambie al pasar el ratón sobre él.

Para determinar los colores RGB correspondientes a cada color se ha utilizado la paleta de la página http://colorpicker.com.

El código de **`LabelMedida`** es mucho más sencillo, sólo establecemos la tipografía y los colores.

```
public class LabelMedida extends JLabel{
    public LabelMedida(String texto){
        super(texto);
        estableceAspecto();
    }

    public LabelMedida(){
        super();
        estableceAspecto();
    }

    private void estableceAspecto(){
        setFont(new Font("Cooper Black", Font.PLAIN, 14));
        setForeground(new Color(235,204,28));
    }
}
```

Del mismo modo, en nuestro marco a medida establecemos los parámetros a los que estamos habituados.

```
public class FrameMedida extends JFrame{

    public FrameMedida(){
        super("Componentes a medida");

        setLocationRelativeTo(null);
        setLayout(new FlowLayout());
        getContentPane().setBackground(Color.white);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public void muestra(){
        setVisible(true);
    }
}
```

Establecemos el color de fondo del *Content Pane* (que es donde se agregan los componentes) como blanco). Además, hemos definido un método público **`muestra()`** que se utilizará para hacer el marco visible.

Por último vamos a crear un marco que haga uso de nuestro **`FrameMedida`** a través de la herencia y donde estableceremos los elementos concretos que tenemos en nuestra interfaz. Como es lógico, desde la clase hija vamos a poder acceder tanto a los métodos que hemos definido como a los de **`JFrame`**, además de implementar oyentes de eventos como hacíamos hasta ahora:

```
public class NuevoFrame extends FrameMedida implements ActionListener{
        BotonMedida btnPulsa;
        LabelMedida lblTexto;

    public NuevoFrame(){
        btnPulsa=new  BotonMedida("Púlsame");
        btnPulsa.addActionListener(this);
        lblTexto=new LabelMedida("Etiqueta");

        add(btnPulsa);
        add(lblTexto);
        setSize(300,75);
    }
    @Override
    public void actionPerformed(ActionEvent ae) {
      lblTexto.setText("Hola otra vez");
    }
    public static void main(String[] args){
        new NuevoFrame().muestra();
    }
}
```

Mostrando dibujos

Para mostrar un dibujo, deberemos usar un panel como si se tratase de un lienzo (en inglés, *canvas*). Para dibujar en un panel, debemos sobrescribir el método **`paintComponent()`** de la clase **`JPanel`**: Éste redibujará el panel.

`paintComponent()` es llamado de forma automática cuando el panel necesita ser redibujado, ya sea por un cambio de tamaño o porque alguna de sus partes ha vuelto a quedar al descubierto tras estar oculta por otra ventana o control. El algoritmo usado por el gestor de ventanas para decidir cuándo deben ser redibujadas ha variado con el tiempo. En Windows XP y los gestores de ventanas más antiguos, el gestor no mostraba las ventanas que estaban tapadas, ya que éstas no eran visibles. Con la aparición de gestores compuestos actuales (con transparencias) se hace necesario que todas las ventanas se dibujen de forma simultánea, con el sobrecoste que ello supone para el S.O.

Nunca debemos llamar de manera directa a **paintComponent**(). En su lugar podemos usar al método **repaint**() del panel, que se encargará de gestionar el repintado de manera ordenada. Como podemos imaginarnos, **repaint**() no pinta tal cual, sino que avisa a la JVM de que nuestro formulario debe ser repintado.

Método `paintComponent()`

Su sintaxis es:

```
public void paintComponent(Graphics g)
```

El objeto **`Graphics`** recibido contiene una superficie de dibujo, pudiendo consultar su ancho y su alto, así como usar diversos métodos para pintar sobre el control, de forma que se encargará de conectar directamente con el S.O. y pintar todos los objetos que le hayamos indicado. El objeto **`Graphics`** no lo construimos directamente, sino que es Java quien lo proporciona de acuerdo a las características del control sobre el cual vamos a pintar.

Dado que vamos a sobrescribir el método **`paintCompontent()`** y no queremos perder la funcionalidad de la clase base, usaremos al comienzo del método una llamada al de la clase base para así inicializar correctamente la superficie de dibujo:

```
public void paintComponent(Graphics g){
   super.paintComponent(g); //Inicializamos el panel
  //Aquí irían las instrucciones de dibujo
}
```

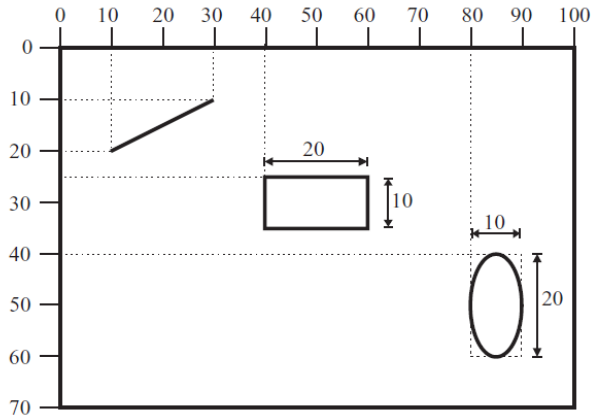
Comandos gráficos

`Graphics` contiene una serie de métodos que nos permiten crear dibujo. Por ejemplo:

- **`drawLine(int xstart, int ystart, int xend, int yend)`**
  - Dibuja un segmento de línea entre los puntos **`(xstart,ystart)`** y **`(xend,yend)`**.
- **`drawRect(int xleft, int ytop, int width, int height)`**
  - Dibuja el contorno de un rectángulo. La esquina superior izquierda está en las coordenadas (**`xleft`**, **`ytop`**) y el ancho y el alto son **`width`** y **`height`**.

- **`drawOval(int xleft, int ytop, int width, int height)`**
  - Dibuja el contorno de una elipse. Los ejes de la elipse corren paralelos a los ejes de coordenadas. La esquina superior izquierda del rectángulo que contiene a la elipse se sitúa en las coordenadas **`(xleft, ytop)`** y es denominado punto de referencia. El eje horizontal de la elipse tiene longitud **`width`** y el eje vertical **`height`**
- **`drawString(String text, int xleft, int ybottom)`**
  - Dibuja un texto. La esquena inferior izquierda del rectángulo contenedor se sitúa en **`(xleft, ybottom)`**, que es su punto de referencia.
- **`fillRect(int xleft, int ytop, int width, int height)`**
  - Dibuja un rectángulo relleno de forma análoga a como lo hacía **`drawRect`**()
- **`fillOval(int xleft, int ytop, int width, int height)`**
  - Dibuja una elipse rellena de forma análoga a como lo hacía **`drawOval`**().
- **`setColor(Color color)`**
  - Fija el color que usamos para pintar. Este color es usado en el borde de los dibujos de contorno, o en el relleno de los dibujos rellenos.

El siguiente ejemplo muestra el resultado de usar **`drawLine(10,20,30,10)`**, **`drawRect(40,25,20,10)`** y **`drawOval(80,40,10,20)`**:



Una aplicación gráfica sencilla

Nuestra aplicación solamente pretende mostrar algunos dibujos sobre un panel.

Como se ha visto, en este caso necesitamos heredar de la clase **`JPanel`** para poder utilizar las funciones de dibujo.

El código de la clase es sencillo. En primer lugar llamamos al método de la clase base para luego ir dibujando una serie de objetos geométricos sobre la pantalla. Los objetos al ser dibujados son sobrescritos, luego cuando un objeto se sitúa sobre otro lo tapa.

Como vemos, establecemos el tamaño del panel usando el método **`setPreferredSize()`**, que es el que debemos utilizar en el caso de querer fijar tamaño de componentes internos como es el panel. El objeto recibe un objeto de tipo **`Dimension`** al que le pasamos el ancho y el alto, si bien podía recibir directamente las coordenadas.

```
public class GraficosPanel extends JPanel{
    public GraficosPanel(){
        setBackground(Color.white);
        setPreferredSize(new Dimension(300,300));
    }
    @Override
    public void paintComponent(Graphics g){
        super.paintComponent(g);

        g.setColor(Color.black);
        g.drawLine(10,10,100,100);

        g.setColor(Color.red);
        g.drawLine(10,100,100,10);

        g.setColor(Color.green);
        g.drawOval(120,0,70,40);

        g.setColor(Color.yellow);
        g.fillOval(230,150,30,30);

        g.setColor(Color.red);
        g.fillOval(245,150,30,30);

        g.setColor(Color.black);
        g.fillOval(238,160,30,30);

        g.setColor(Color.cyan);
        g.fillOval(10, 120, 100, 60);

        g.setColor(getBackground());
        g.fillOval(50, 140, 100, 60);

        g.setColor(Color.blue);
        g.drawString("Swing es fácil!", 100, 200);
    }
}
```

En cuanto al frame contenedor, simplemente crea la ventana y muestra el contenido. Hay que hacer énfasis en el método **pack()** que se encarga de que los componentes del frame tengan sus tamaños completos.

```
public class GraficosFrame extends JFrame {
      public GraficosFrame(){
          super("Graficos");
          add(new GraficosPanel());
          pack();
          setVisible(true);
      }

    public static void main(String[] args) {
      GraficosFrame graficos=new GraficosFrame();
    }
}
```

Creación de componentes a partir de componentes genéricos

A modo de ejemplo, vamos a ver cómo podemos crear un componente a medida que herede de **`JComponent`**. Básicamente tenemos que implementar los siguientes métodos:

- Métodos **`getPreferredSize()`**, **`getMaximumSize()`** y **`getMinimumSize()`**. Son obligatorios aunque no siempre respetados por el administrador de distribución que usemos.
- Método **`paintComponent(Graphics g)`**. Se encarga de dibujar el componente con las indicaciones dadas.

En el siguiente ejemplo vamos a implementar un botón a medida. Podemos hacer las siguientes consideraciones:

- Se declaran como atributos privados las dimensiones del botón, el texto que contendrá (que le pasamos en el constructor) y una variable booleana que nos servirá para comprobar si el ratón está dentro del componente.
- El componente implementa la interfaz **`MouseListener`**. La variable booleana **`ratonDentro`** cambia su valor en los métodos **`mouseEntered()`** y **`mouseExited()`**. Desde éstos también se llama a **`repaint()`** que se encarga de llamar a **`paintCompontent()`**.
- En dicho método pintamos el botón y el contenido con distinto formato dependiendo de la posición del ratón. Para simular el borde, pintamos un rectángulo dentro de otro.

```
public class OtroBotonMedida extends JComponent implements MouseListener {
      private Dimension medidas=new Dimension(90,30);
      private String texto;
      private boolean ratonDentro;

      public OtroBotonMedida(String texto){
          this.texto=texto;
          addMouseListener(this);
      }

       @Override
        public Dimension getPreferredSize() {
            return medidas;
        }

        @Override
        public Dimension getMaximumSize() {
            return medidas;
        }

        @Override
        public Dimension getMinimumSize() {
            return medidas;
        }
```

```
    @Override
    protected void paintComponent(Graphics g){
    if(ratonDentro){
       g.setColor(Color.black);
       g.fillRect(0,0,medidas.width,medidas.height);
       g.setColor(new Color(255,217,3));
       g.fillRect(3, 3,medidas.width-6,medidas.height-6);
       g.setColor(Color.white);
       setFont(new Font("Cooper Black", Font.PLAIN, 14));
       g.drawString(texto,20,20);

    }
      else{
       g.setColor(Color.black);
       g.fillRect(0,0,medidas.width,medidas.height);
       g.setColor(new Color(235,204,28));
       g.fillRect(2, 2,medidas.width-4,medidas.height-4);
       g.setColor(Color.white);
       setFont(new Font("Cooper Black", Font.PLAIN, 14));
       g.drawString("Pulsar",20,20);
      }
    }
    @Override
    public void mouseClicked(MouseEvent me) { }
    @Override
    public void mousePressed(MouseEvent me) { }
    @Override
    public void mouseReleased(MouseEvent me) { }
    @Override
    public void mouseEntered(MouseEvent me) {
       ratonDentro=true;
       repaint();
    }
    @Override
    public void mouseExited(MouseEvent me) {
       ratonDentro=false;
       repaint();
    }
}
```

NOTA: La posición del texto y el tamaño del ratón se han establecido mediante valores numéricos constantes. En realidad, deberíamos haber determinado ambos en función del tamaño de la cadena de texto que va a contener el ratón. Para no complicar el ejemplo se ha omitido dicha parte, si bien el alumno que tenga interés puede consultar el siguiente enlace donde se explica cómo hacerlo.

La siguiente captura muestra el ejemplo de **`NuevoFrame`** donde hemos modificado el tipo del botón por **`OtroBotonMedida:`**

Si bien el resultado es muy parecido, la ventaja de este enfoque es que nos estamos librando de todo el código de **`JButton`** que no usamos en la clase hija. Esto hace el componente a medida más eficiente y liviano. Sin embargo, la programación del componente también implica mayor dificultad y esfuerzo.

Independientemente del enfoque utilizado, la creación de componentes personalizados mediante herencia es un mecanismo que facilita mucho la reutilización de código y el mantenimiento del mismo, por eso se recomienda utilizarla siempre que se pueda.