IES DR LLUÍS SIMARRO

ENSENYAMENT PÚBLIC

XÀTIVA

CFGS Desarrollo de aplicaciones multiplataforma

Módulo profesional: Programación

GENERALITAT VALENCIANA

Conselleria d'Educació, Investigació, Cultura i Esport

GOBIERNO DE ESPAÑA

MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

Unió Europea

Fons Social Europeu

L'FSE inverteix en el teu futur

Material elaborado por:

Edu Torregrosa Llácer

(aulaenlanube.com)

Aplicaciones con Interfaz gráfica en JAVA

1. La API de Java para GUIs
2. AWT
3. Swing
4. Maven
5. JavaFX

Introducción

- A lo largo de los años han ido surgiendo distintos paquetes que nos han permitido desarrollar aplicaciones con interfaces gráficas en JAVA.
- En primer lugar surgió AWT (JDK 1.0), que realmente no estaba programado en JAVA, era el propio SO el encargado de proporcionar los distintos componentes a las aplicaciones JAVA.
- Más tarde apareció JAVA Swing (JDK 1.2), cuyos componentes estaban escritos en JAVA y ya no dependían del SO.
- Con el tiempo Swing se empezó a quedar anticuado ante la aparición de múltiples tipos de dispositivos, entre ellos, los táctiles. Para satisfacer las necesidades de este tipo de dispositivos surgió JavaFX (JDK 8)

Introducción

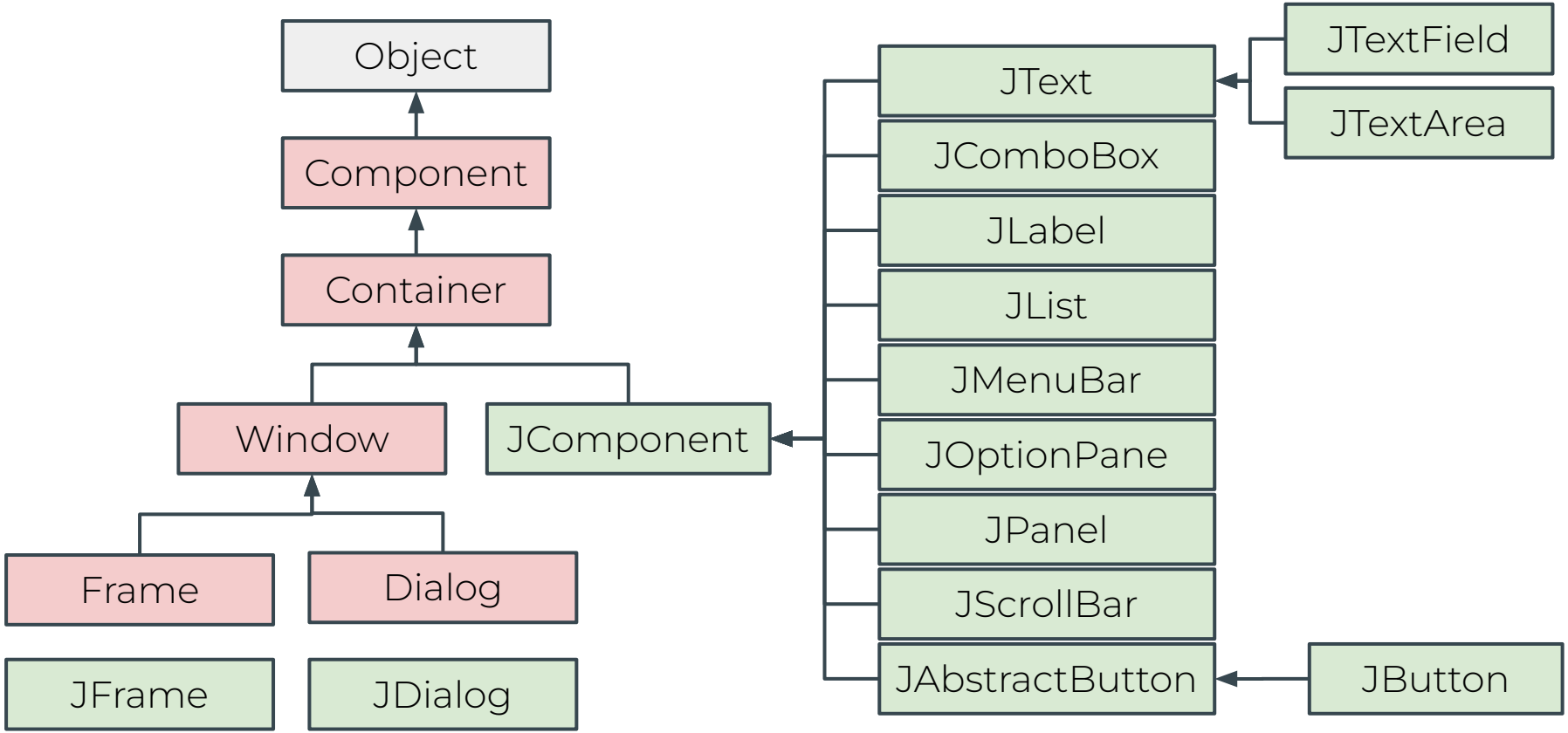
- Con la evolución de JavaFX y su inclusión en el JDK, se convirtió en la elección recomendada para el desarrollo de nuevas aplicaciones de escritorio en Java. Sin embargo, AWT y Swing siguen siendo mantenidos y son utilizados en muchas aplicaciones existentes.
- JavaFX se utiliza actualmente para diseñar interfaces gráficas JAVA en todo tipo de dispositivos. Cuenta con soporte para animaciones, acciones táctiles, hojas de estilo, ficheros de propiedades, etc.

JAVA Swing

- Java Swing es una herramienta de interfaz gráfica de usuario (GUI) que incluye un amplio conjunto de componentes. Incluye paquetes que le permiten crear componentes de GUI en Java (JButton, JLabel, etc.)
- A diferencia de AWT, los componentes de Swing son renderizados por JAVA, lo que permite una mayor consistencia en diferentes sistemas operativos. Swing ofrece una mayor variedad de componentes y permite una mayor personalización en comparación con AWT.
- En Swing, el componente principal que almacena la interfaz gráfica del usuario se llama frame (clase **Jframe**). Un frame es una ventana vacía donde puedes añadir un panel (clase Jpanel), que sirve como contenedor para los elementos gráficos de la pantalla del usuario (campos de texto, botones). Incluso al trabajar con JAVA Swing se necesitan paquetes de AWT.

Principales clases de JAVA Swing

Jerarquía de clases en Java Swing



JAVA Swing:
Contenedores y Layouts

Contenedores

Los contenedores son clases que pueden tener otros componentes dentro. En JAVA Swing hay 3 tipos de contenedores principales:

- **JFrame**: contenedor principal, se trata de una ventana completa con su título, icono y botones específicos. Un JFrame no admite ninguna otra ventana como padre.
- **JPanel**: un contendor que se incluye en un marco para organizar un grupo de elementos. El un contenedor secundario, y por ello se debe estar dentro de un contenedor principal (JFrame)
- **JDialog**: ventana emergente que se suele utilizar para mostrar mensajes. No es una ventana completamente funcional como el Marco. Un JDialog admite como padres tanto otro JDialog como un JFrame. Esto es importante porque una ventana hija siempre quedará por encima de su padre.

Cerrar ventana

Para poder cerrar de forma adecuada cada una de las ventanas de nuestra aplicación debemos indicarlo de forma explícita. De lo contrario podemos cerrar todas las ventanas, pero nuestra aplicación continuará en marcha. Esto se hace a través del método **setDefaultCloseOperation(...)**, como parámetro debemos pasarle el tipo de acción.

Para indicar el tipo de acción al cerrar una ventana tenemos disponibles una serie de variables estáticas de la clase **JFrame**, entre ellas podemos destacar:

- **EXIT_ON_CLOSE:** Cierra la aplicación al cerrarla
- **DISPOSE_ON_CLOSE**: Elimina la ventana al cerrarla
- **HIDE_ON_CLOSE**: Oculta la ventana al cerrarla

```
public static void main(String[] args) {

    JFrame ventana = new JFrame("Mi primera ventana");
    ...
    ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

Layout manager

Un layout manager es un objeto que implementa la interfaz **LayoutManager** y controla la forma en la que se distribuyen los componentes dentro de un contenedor. En otras palabras, define cómo se organizan los componentes gráficos en un contenedor, tales como JFrame, JPanel, etc. Con ello podemos especificar la apariencia que tendrán los componentes a hora de colocarlos sobre un contenedor, controlando tamaño y posición (layout) automáticamente. Entre las distintas implementaciones del LayoutManager podemos destacar:

- FlowLayout
- **BorderLayout**
- CardLayout
- GridLayout
- GridBagLayout
- BoxLayout
- OverlayLayout
- CustomLayout

```
JFrame ventana = new JFrame();
ventana.setLayout(new BorderLayout());

ventana.add(new JButton("Norte"), BorderLayout.PAGE_START);
ventana.add(new JButton("Sur"), BorderLayout.PAGE_END);
ventana.add(new JButton("Este"), BorderLayout.LINE_END);
ventana.add(new JButton("Oeste"), BorderLayout.LINE_START);
ventana.add(new JButton("Centro"), BorderLayout.CENTER);
```

Layout manager

FlowLayout: Este es el administrador de diseño por defecto para los paneles en Java Swing. Organiza los componentes en una dirección (de izquierda a derecha por defecto) y pasa a la siguiente línea cuando no hay más espacio. Puedes configurar la alineación (izquierda, centro, derecha) y el espaciado entre los componentes.

```
ventana.setLayout(new FlowLayout());
```

BorderLayout: Este layout divide el contenedor en cinco regiones: Norte, Sur, Este, Oeste y Centro. Cada componente que añades se coloca en una de estas regiones y se expande para llenar todo el espacio disponible en esa región.

```
ventana.setLayout(new BorderLayout());
```

GridLayout: Este layout divide el contenedor en una cuadrícula de celdas de igual tamaño, todas las cuales tienen la misma anchura y altura. Cada componente que añades ocupa una celda de la cuadrícula.

```
ventana.setLayout(new GridLayout());
```

Contenedores y layout

Si quieres un control absoluto sobre la posición y el tamaño de los componentes, puedes establecer el layout manager en **null** y utilizar el método **setBounds(..)** para cada componente. Sin embargo, se desaconseja generalmente hacer esto, ya que puede llevar a interfaces de usuario que no se ven bien en todas las plataformas o con todas las configuraciones de tamaño de ventana.

```
public static void main(String args[]) {
    //creamos ventana
    JFrame ventana = new JFrame("Mi primera ventana");
    ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // características ventana
    ventana.setSize(500, 500);
    ventana.setLayout(null);
    // creamos botón, posición x:100, y:200 - 200px de ancho y 40px de alto
    JButton boton = new JButton("Pulsa");
    ventana.add(boton);
    boton.setBounds(100, 200, 200, 40);
    ventana.setVisible(true);
}
```

JAVA Swing: Componentes

Componentes en JAVA Swing

Existen multitud de componentes que podemos agregar a nuestras interfaces, entre ellos podemos destacar

- **JLabel**: permite crear etiquetas para mostrar texto en nuestras aplicaciones.
- **JButton**: permite crear botones.
- **JTextField**: permite ingresar una cadena de caracteres por teclado.
- **JTextArea**: permite ingresar múltiples líneas de caracteres por teclado.
- **JMenuBar**: permite crear un menú horizontal en la parte superior de un JFrame, además se debe combinar con objetos de la clase **JMenu** y por último objetos de la clase **JMenuItem**
- **JCheckBox:** permite seleccionar varias opciones de una lista al mismo tiempo.
- **JRadioButton:** permite seleccionar una opción de una lista, todas están visibles.
- **JComboBox:** permite seleccionar una opción de una lista, está visible la opción activa.

Ejemplo componentes JAVA Swing

```
JLabel jLabel = new JLabel("Esto es un JLabel"); //creamos JLabel
ventana.add(jLabel);
JButton jButton = new JButton("Botón"); //creamos JButton
ventana.add(jButton);
JTextField jTextField = new JTextField("Esto es un JTextField"); //creamos JTextField
ventana.add(jTextField);
JCheckBox jCheckBox = new JCheckBox("Esto es JCheckBox"); //creamos JCheckBox
ventana.add(jCheckBox);
JRadioButton jRadioButton = new JRadioButton("Esto es JRadioButton"); //creamos JRadioButton
ventana.add(jRadioButton);

//menu de opciones
JMenuBar barraMenu = new JMenuBar();
JMenu menu = new JMenu("Esto es un JMenu");
menu.add(new JMenuItem("Esto la opción 1 del primer menú"));
menu.add(new JMenuItem("Esto la opción 2 del primer menú"));

JMenu menu2 = new JMenu("Esto es otro JMenu");
menu2.add(new JMenuItem("Esto la opción 1 del segundo menú"));
menu2.add(new JMenuItem("Esto la opción 2 del segundo menú"));

barraMenu.add(menu);
barraMenu.add(menu2);
ventana.setJMenuBar(barraMenu);
```

JAVA Swing y AWT:
Eventos

Eventos

- En programas de consola, las instrucciones se ejecutan de forma secuencial, una tras otra.
- En aplicaciones con interfaz gráfica no existe un orden predefinido a la hora de ejecutar las instrucciones. Dependiendo de las acciones del usuario, se ejecutarán unas instrucciones u otras. Es lo que se denomina una **programación orientada a eventos.**
- Un evento es una acción que debe ocurrir para que se ejecute una función determinada. El evento más típico es hacer click sobre un botón.
- Para poder utilizar eventos se utilizan **interfaces**. Dichas interfaces permanecen a la escucha y solo ejecutan la función cuando se produce dicho evento. Una de las más utilizadas en **ActionListener.**

addActionListener

- Cuando usamos el **addActionListener()** de un componente, nos estamos suscribiendo a la "acción típica" o que java considera más importante para ese componente. Por ejemplo, la acción más típica de un **JButton** es pulsarlo. Para un **JTextField**, java considera que es pulsar <INTRO> indicando que hemos terminado de escribir el texto, para un **JComboBox** es seleccionar una opción, etc.
- **Como parámetro, debemos pasar un objeto de una clase que implementa ActionListener**. Por tanto, tenemos tres opciones, crear una instancia de una clase que implemente la interface ActionListener, crear una instancia de una clase anónima que implemente la interface ActionListener, o utilizar una lambda que implemente la interface funcional ActionListener.

Método de implementación con una clase

```
// creamos una clase para implementar ActionListener
public class BotonSaludo implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    //acción al hacer click
    System.out.println("Hola mundo desde un botón");
  }
}
```

```
public class Ventana {
  public static void main(String args[]) {
    ...
    // creamos un botón
    JButton boton = new JButton("Pulsa");
    // asociamos instancia de una clase que implementa ActionListener
    boton.addActionListener(new BotonSaludo());
  }
}
```

El modelo de eventos en JAVA

El modelo de eventos en Java se basa en el patrón de diseño Observer, que permite que ciertos objetos sean notificados cuando ocurre un evento. En el modelo de eventos de Java, hay fuentes de eventos, listeners de eventos y objetos de eventos.

- **Fuentes de Eventos**: Estos son los objetos que generan eventos (**Jbutton**). Las fuentes de eventos tienen un registro de los listeners de eventos que están interesados en ser notificados cuando ocurre un evento.
- **Listeners de Eventos**: Estos son los objetos que están interesados en ser notificados cuando ocurre un evento. Estos objetos implementan interfaces listener, que definen uno o más métodos que serán llamados cuando ocurra el evento (**BotonSaludo**)
- **Objetos de Eventos**: Cuando ocurre un evento, la fuente de eventos crea un objeto de evento que encapsula información sobre el evento, como cuándo ocurrió y qué tipo de evento fue. Este objeto de evento se pasa al listener de eventos como un argumento de los métodos de manejo de eventos(**ActionEvent**)

Método de implementación creando instancia y con lambda

```
public static void main(String[] args) {
     ...
    // creamos un botón
    JButton boton = new JButton("Presionar");
    // asociamos evento al botón creando una instancia de una clase anónima
    boton.addActionListener(new ActionListener() {
        //implementamos método abstracto actionPerformed
        public void actionPerformed(ActionEvent e) {
         //acción al hacer click
         System.out.println("Hola mundo desde un botón");
        }
     });
}
```

```
public static void main(String[] args) {
     ...
    // creamos un botón
    JButton boton = new JButton("Presionar");
    // asociamos evento al botón creando una lambda, interfaz funcional
    boton.addActionListener(e -> {
         //acción al hacer click
         System.out.println("Hola mundo desde un botón");
     });
}
```

JAVA Swing:
Eventos en componentes

Ejemplo de uso JLabel

```
JFrame ventana = new JFrame("Ejemplo JLabel");
ventana.setSize(400, 300);
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// creamos un JLabel
JLabel label = new JLabel("Intenta pulsar el botón");

// creamos un JButton
JButton boton = new JButton("Es aquí");

// al presionar el botón, modifica el texto del JLabel
boton.addActionListener(e -> label.setText("Enhorabuena, has pulsado el botón"));

// agregamos el JLabel y JButton al JFrame
ventana.setLayout(new FlowLayout());
ventana.add(label);
ventana.add(boton);

// mostramos ventana
ventana.setVisible(true);
```

Ejemplo de uso JTextField

```
JFrame ventana = new JFrame("Ejemplo de uso JTextField");
ventana.setSize(300, 200);
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
ventana.setLayout(new FlowLayout());

// Crea un JTextField
JTextField textField = new JTextField(10);  // 10 es el ancho

// Crea un JButton
JButton boton = new JButton("Pulsa");
// Agrega un ActionListener al botón para manejar el clic
boton.addActionListener(e -> {
    // Cuando el botón se presiona, imprime el texto actual del JTextField
    System.out.println("Texto introducido: " + textField.getText());
});
// Agrega el JTextField y JButton al frame
ventana.add(textField);
ventana.add(boton);
ventana.setVisible(true);
```

Ejemplo de uso JComboBox

```
JFrame ventana = new JFrame("Ejemplo ComboBox");
ventana.setSize(400, 300);
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// creamos un JComboBox
String[] opciones = { "Opción 1", "Opción 2", "Opción 3" };
JComboBox<String> comboBox = new JComboBox<>(opciones);

// creamos un JButton
JButton boton = new JButton("Pulsa");

// al presionar el botón, imprime por consola el elemento seleccionado en el JComboBox
boton.addActionListener(e -> {   System.out.println(comboBox.getSelectedItem());  });

// agregamos el JComboBox y JButton al JFrame
ventana.setLayout(new FlowLayout());
ventana.add(comboBox);
ventana.add(boton);

// mostramos ventana
ventana.setVisible(true);
```

Ejercicios componentes JAVA Swing

1. Crea una clase en JAVA Swing que reciba dos entero en dos JTextField y muestre en un JLabel el resultado de sumar ambos enteros. Se debe comprobar previamente si el contenido introducido en ambas cajas de texto se corresponde con un entero válido.
2. Crea una clase en JAVA Swing que reciba un número real en un JTextField y realice una conversión entre €/$ y $/€. Se tomará como ejemplo el valor del dólar en 0,91€. Se puede crear un botón por cada conversión, o con un comboBox, ambas opciones serán válidas. El resultado se mostrará en un JLabel.
3. Crea una clase en JAVA Swing que simula una ventana de login, con nombre de usuario y contraseña. La contraseña debe contener como mínimo 8 caracteres, mínimo una mayúscula, una minúscula y un número. Además se debe añadir una caja de texto adicional para la verificación de la contraseña y que ambas coincidan. Si todo es correcto, se deben vaciar las cajas de texto y mostrar un mensaje de retroalimentación por consola. Si hay algún tipo de error, se deberá mostrar un mensaje de error personalizado por consola.

JAVA Swing:
Eventos de teclado

Eventos de teclado

- Para leer del teclado es necesario registrar un objeto (Listener) que se encargue de escuchar si una tecla es presionada. Para escuchar este tipo de eventos JAVA nos ofrece la interfaz **KeyListener.** Necesitamos implementar tres métodos abstractos de esta interfaz en nuestra clase.
- El método **keyPressed(KeyEvent e)** se invocará cuando se presione una tecla cualquiera.
- El método **keyReleased(KeyEvent e)** se invocará cuando se suelte la tecla.
- El método **keyTyped(KeyEvent e)** se invocará cuando una tecla escriba un carácter Unicode, no se invocará con caracteres especiales, por ejemplo tecla Ctrl.

Finalmente, debemos añadir dicho KeyListener a nuestra aplicación a través del método **addKeyListener()**. Dicho método recibe como parámetro una instancia de un objeto que implementa la interfaz KeyListener.

Método de implementación en la propia clase y con instancia

```
//debemos crear una clase para implementar KeyListener
public class Ventana implements KeyListener {
    // creamos una ventana
    JFrame ventana = new JFrame();
    // agregamos evento de escucha del teclado
    ventana.addKeyListener(this);

    //finalmente debemos implementar los métodos abstractos de KeyListener
    public void keyTyped(KeyEvent e)  {  …  }
    public void keyPressed(KeyEvent e)  {  …  }
    public void keyReleased(KeyEvent e)  {  …  }
}
```

```
// creamos una ventana
JFrame ventana = new JFrame();
// agregamos evento de escucha del teclado
ventana.addKeyListener(new KeyListener() {
     //finalmente debemos implementar los métodos abstractos de KeyListener
     public void keyTyped(KeyEvent e)     {  …  } //cuando se teclea algún carácter
     public void keyPressed(KeyEvent e)   {  …  } //cuando se presiona cualquier tecla
     public void keyReleased(KeyEvent e)  {  …  } //cuando se suelta la tecla
}); //orden de los eventos: keyPressed -> keyTyped -> keyReleased
```

Comprobación de la tecla pulsada

```
public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == KeyEvent.VK_RIGHT)
        System.out.println("Has pulsado la tecla → ");
    if (e.getKeyCode() == KeyEvent.VK_LEFT)
         System.out.println("Has pulsado la tecla ← ");
    if (e.getKeyCode() == KeyEvent.VK_DOWN)
         System.out.println("Has pulsado la tecla ↓ ");
    if (e.getKeyCode() == KeyEvent.VK_UP)
         System.out.println("Has pulsado la tecla ↑ ");
}
```

JAVA Swing:
Imágenes e iconos

Iconos e imágenes en JAVA Swing

El uso de imágenes en Java Swing se basa en la clase **ImageIcon**. Esta clase permite cargar imágenes en formatos como PNG, JPEG, entre otros. La imagen se puede mostrar directamente en componentes de Swing como JLabel, JButton, etc., o se puede pintar en un JPanel con la ayuda del método drawImage() de la clase **Graphics**. Veamos algunos ejemplos:

```
//mostrar imagen en un JLabel
ImageIcon imagen = new ImageIcon("ruta_imagen");
JLabel etiqueta = new JLabel(imagen);
//mostrar icono en un JButton
ImageIcon icono = new ImageIcon("ruta_icono");
JButton boton = new JButton("Pulsa", icono);

//dibujar imagen en un JPanel
ImageIcon imageIcon = new ImageIcon("ruta_imagen");
Image imagen = imageIcon.getImage();
JPanel panel = new JPanel() {
     @Override
     protected void paintComponent(Graphics g) {
          super.paintComponent(g);
          g.drawImage(imagen, 0, 0, this);
     }
};
```

Clase Image en AWT

La clase java.awt.Image es una clase abstracta en Java que representa una imagen. Las imágenes son datos inmutables que se pueden dibujar en componentes. Al ser una clase abstracta, no podemos instanciar directamente un objeto Image. En cambio, podemos obtener una instancia a través de métodos proporcionados por otras clases, como Toolkit.getImage(), ImageIO.read(), o creando un ImageIcon y luego llamando a getImage(). Algunos de sus métodos son los siguientes:

- **getScaledInstance(int width, int height, int hints)**: este método se utiliza para escalar la imagen a un nuevo tamaño. Retorna una nueva imagen. Utiliza constantes para definir el modo de escalado, parámetro “hints” (SCALE_DEFAULT, SCALE_FAST, SCALE_SMOOTH, SCALE_REPLICATE, SCALE_AREA_AVERAGING)
- **getWidth() y getHeight()**: estos métodos se utilizan para obtener las dimensiones de la imagen.
- **flush()**: se utiliza para limpiar los recursos utilizados por la imagen.

Clase Image en AWT

Debido a que la clase Image es abstracta, a menudo se suele trabajar con subclases de Image. Las más comunes son BufferedImage y VolatileImage.

- La clase **BufferedImage** es una subclase de Image que permite la manipulación directa de píxeles de imagen y es utilizada para imágenes en memoria.
- La clase **VolatileImage** se utiliza para imágenes que pueden ser aceleradas por hardware. Esto significa que la imagen se almacena en la memoria de la tarjeta gráfica en lugar de la memoria del sistema, lo que puede proporcionar un mejor rendimiento según el tipo de hardware.

```
// creamos una imagen
BufferedImage imagenOriginal = ImageIO.read(new File("ruta_imagen"));

// redimensionar la imagen
Image imagenEscalada = imagenOriginal.getScaledInstance(100, 100, Image.SMOOTH);

// creamos un JLabel y fijamos la imagen
JLabel labelImagen = new JLabel(new ImageIcon(imagenEscalada));
```

JAVA Swing:
Eventos de ratón

Eventos de ratón - MouseListerner

MouseListener es una interface en Java que se utiliza para recibir eventos de ratón en un componente. Forma parte del paquete **java.awt.event**. Los eventos de ratón son generados por acciones del usuario como clics, movimientos del ratón y arrastrar y soltar. La interfaz MouseListener define los siguientes cinco métodos que deben ser implementados por cualquier clase que implemente la interfaz:

- **mouseClicked(MouseEvent e)**: se invoca cuando se ha hecho clic en un componente. Un clic de ratón es considerado como la acción de presionar y soltar el botón del ratón en la misma posición del componente sin mover el ratón.
- **mousePressed(MouseEvent e)**: se invoca cuando se ha presionado un botón del ratón en un componente. No es necesario soltar el botón del ratón.
- **mouseReleased(MouseEvent e)**: se invoca cuando se ha soltado un botón del ratón, no tiene por qué seguir a una presión del botón del ratón en el mismo componente.
- **mouseEntered(MouseEvent e)**: se invoca cuando el cursor entra en la zona del componente.
- **mouseExited(MouseEvent e)**: se invoca cuando el cursor sale de la zona del componente.

Finalmente, debemos añadir dicho MouseListener a nuestra aplicación a través del método **addMouseListener()**. Dicho método recibe como parámetro una instancia de un objeto que implementa la interfaz MouseListener.

Ejemplos MouseListener

```
// creamos un JLabel y fijamos una imagen
JLabel labelImagen = new JLabel(new ImageIcon("ruta_imagen"));
labelImagen.addMouseListener(new MouseListener() {

     public void mouseClicked(MouseEvent e) {
          System.out.println("Has pulsado en la imagen: ("+e.getX()+", "+e.getY()+")");
     }

     public void mousePressed(MouseEvent e) {
          System.out.println("Has pulsado con el foco en la imagen");
     }

     public void mouseReleased(MouseEvent e) {
          System.out.println("Has soltado el botón del ratón");
     }

     public void mouseEntered(MouseEvent e) {
          System.out.println("Has entrado en la imagen");
     }

     public void mouseExited(MouseEvent e) {
          System.out.println("Has salido de la imagen");
     }
});  //orden de los eventos: mousePressed -> mouseReleased-> mouseClicked
```

JAVA Swing: paintComponent y repaint
Clase Graphics

paintComponent y Graphics

El método **paintComponent(Graphics g)** es una parte crucial de la arquitectura Swing en Java. Esta función se llama automáticamente por Swing cuando cree que es necesario volver a dibujar un componente, y es donde debes poner tu lógica de dibujo personalizada. Algunas situaciones en las que se llama a paintComponent incluyen:

- Cuando se hace visible el componente por primera vez.
- Cuando el usuario redimensiona la ventana que contiene el componente.
- Cuando el componente se ha invalidado (por ejemplo, llamando al método repaint()).

El parámetro **Graphics g** que se pasa a paintComponent es el contexto gráfico en el que se realizarán las operaciones de dibujo. Dicha clase nos proporciona métodos para dibujar figuras y texto, configurar colores, fuentes y otros atributos gráficos. Además tenemos el método **repaint()**, dicho método se utiliza para indicar al sistema que un componente necesita ser redibujado. Invocar al método no redibuja inmediatamente el componente, Swing decidirá cuándo hacerlo, y cuando lo haga, llamará al método paintComponent().

clase Graphics

La clase java.awt.Graphics es una clase abstracta que proporciona una API para dibujar formas, texto y otros elementos en un objeto java.awt.Component en Java.

- **setColor(Color c)**: Establece el color actual en el contexto gráfico. Todos los métodos de dibujo que siguen utilizarán este color.
- **getColor()**: Devuelve el color actual en el contexto gráfico.
- **setFont(Font font)**: Establece la fuente actual en el contexto gráfico. Todos los métodos de dibujo de texto que siguen utilizarán esta fuente.
- **getFont()**: Devuelve la fuente actual en el contexto gráfico.
- **drawString(String str, int x, int y)**: Dibuja el texto representado por str en la posición especificada por x y y.
- **drawLine(int x1, int y1, int x2, int y2)**: Dibuja una línea desde la posición (x1, y1) hasta la posición (x2, y2).
- **drawRect(int x, int y, int width, int height)**: Dibuja un rectángulo con las esquinas superior izquierda e inferior derecha especificadas por (x, y) y la anchura y altura especificadas.
- **fillRect(int x, int y, int width, int height)**: Dibuja un rectángulo lleno con las esquinas superior izquierda e inferior derecha especificadas por (x, y) y la anchura y altura especificadas.

clase Graphics

La clase java.awt.Graphics es una clase abstracta que proporciona una API para dibujar formas, texto y otros elementos en un objeto java.awt.Component en Java.

- **drawOval(int x, int y, int width, int height)**: Dibuja el contorno de una elipse que se ajusta en un rectángulo especificado por (x, y) para la esquina superior izquierda y la anchura y altura especificadas.
- **fillOval(int x, int y, int width, int height)**: Dibuja una elipse llena que se ajusta en un rectángulo especificado por (x, y) para la esquina superior izquierda y la anchura y altura especificadas.
- **drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)**: Dibuja el contorno de un sector circular, definido por un rectángulo de encuadre, un ángulo de inicio y un ángulo de arco.
- **fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)**: Dibuja un sector circular lleno, definido por un rectángulo de encuadre, un ángulo de inicio y un ángulo de arco.
- **drawPolygon(Polygon p)**: Dibuja el contorno de un polígono definido por la clase Polygon.
- **fillPolygon(Polygon p)**: Dibuja un polígono lleno definido por la clase Polygon.

Ejemplo de uso - paintComponent y Graphics

```
public class PintarFiguras extends JPanel {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        // dibuja una línea recta horizontal desde (20,30) hasta (200,30)
        g.drawLine(20, 30, 200, 30);

        // dibuja un rectángulo de 200x100
        g.drawRect(50, 50, 200, 100);

        // dibuja un óvalo de 150x100
        g.drawOval(220, 50, 150, 100);

        // dibuja un triángulo de 3 puntos
        Polygon triangulo = new Polygon();
        triangulo.addPoint(50, 200);
        triangulo.addPoint(100, 250);
        triangulo.addPoint(150, 200);
        g.drawPolygon(triangulo);
    }
}
```

```
public static void main(String[] args) {
      JFrame ventana = new JFrame();
      ventana.add(new PintarFiguras());
      ventana.setSize(400, 400);
      ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      ventana.setVisible(true);
}
```

Ejemplo de uso - repaint

```
public class EjemploRepaintHora extends JPanel {

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        LocalDateTime horaActual = LocalDateTime.now();
        DateTimeFormatter formatoHora = DateTimeFormatter.ofPattern("HH:mm:ss");
        String textoHora = horaActual.format(formatoHora);
        g.setFont(new Font("Arial", Font.BOLD, 18));
        g.drawString("Hola, Mundo! son las " + textoHora, 50, 50);
    }

    public static void main(String[] args) {
        JFrame ventana = new JFrame("Hola Mundo con reloj");
        EjemploRepaintHora panel = new EjemploRepaintHora();
        ventana.add(panel);
        ventana.setSize(400, 200);
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ventana.setVisible(true);

        Timer reloj = new Timer(1000, e -> panel.repaint());
        reloj.start();
    }
} //analizar primero el código, antes de ver el vídeo
```

JAVA Swing: paintComponent y repaint
Clase Graphics2D

clase Graphics2D

La clase Graphics2D en Java es una subclase abstracta de Graphics y es parte del paquete java.awt. Fue introducida con la API de Java 2D en Java 1.2 y es fundamental para la representación de gráficos 2D, ofrece capacidades más extensas y sofisticadas para el dibujo y renderizado que la clase Graphics .

- **rotate(double theta):** rota según el ángulo proporcionado alrededor del punto dado. Todos los métodos de dibujo que siguen utilizarán dicha rotación.
- **rotate(double theta, double x, double y):** rota según el ángulo proporcionado alrededor del punto dado. Todos los métodos de dibujo que siguen utilizarán dicha rotación.
- **scale(double sx, double sy):** escala las multiplicando las coordenadas existentes por los factores de escala sx y sy. Todos los métodos de dibujo que siguen utilizarán dicha escala.
- **setStroke(Stroke s):** se utiliza para establecer el atributo de trazo actual, que define cómo se dibujarán las líneas y las curvas. Stroke es una interfaz, cuya implementación es BasicStroke, dicha clase nos permite generar distintos tipos de trazos, tanto en las líneas como en las terminaciones. Todos los métodos de dibujo que siguen utilizarán dicho trazo.
- **GradientPaint(float x1, float y1, Color color1, float x2, float y2, Color color2):** crea un gradiente lineal a lo largo de la línea definida por los puntos dados con los colores dados. Se utiliza para pintar una transición de color entre dos puntos.

Ejemplo de uso - Graphics y Graphics2D

```
public class PintarFigurasBordes extends JPanel {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        Graphics2D g2d = (Graphics2D)g;

        // dibuja una línea en rojo con un ancho de 5
        g2d.setColor(Color.RED);
        g2d.setStroke(new BasicStroke(5));
        g2d.drawLine(20, 30, 200, 30);

        // dibuja un rectángulo verde de 300x200 rotado 5 grados en la posición (100,100)
        // y con borde verde con un ancho de 3
        g2d.setColor(Color.GREEN);
        g2d.setStroke(new BasicStroke(3));
        g2d.rotate(Math.toRadians(5));
        g2d.scale(2,2);
        g2d.fillRect(50, 50, 150, 100);
        g2d.setColor(Color.BLACK);
        g2d.drawRect(50, 50, 150, 100);
    }
}
```

```
public static void main(String[] args) {
      JFrame ventana = new JFrame();
      ventana.add(new PintarFigurasBordes());
      ventana.setSize(800, 800);
      ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      ventana.setVisible(true);
}
```

Ejemplo de uso - repaint

```
public class EjemploRepaintFiguras extends JPanel {

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        int centroX = getWidth() / 2;
        int centerY = getHeight() / 2;
        int size = 200;
        switch (new Random().nextInt(3)) {
            case 0 ->  g.drawOval(centerX - size/2, centerY - size/2, size, size);
            case 1 ->  g.drawRect(centerX - size/2, centerY - size/2, size, size);
            case 2 ->  { … }
       }
    }
    public static void main(String[] args) {
        JFrame ventana = new JFrame("Figuras random");
        EjemploRepaintFiguras panel = new EjemploRepaintFiguras();
        ventana.add(panel);
        ventana.setSize(400, 200);
        ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        ventana.setVisible(true);
        Timer timer = new Timer(1000, e -> panel.repaint());
        timer.start();
    }
}
```

JAVA Swing:
Ventanas emergentes

Ventanas emergentes - JOptionPane

En Swing, existen varios tipos de ventanas emergentes que se pueden usar para interactuar con el usuario. Los principales son **JOptionPane**, **JDialog** y **JPopupMenu**.

- **JOptionPane**: permite mostrar un mensaje, un conjunto de botones y un icono al usuario. También se puede utilizar para recibir una entrada del usuario. Hay varios tipos que puedes utilizar:
  - **showMessageDialog(Component parentComponent, Object message)**: muestra un mensaje al usuario. Puede ser solo texto o incluir un icono.
  - **showConfirmDialog(Component parentComponent, Object message)**: muestra un mensaje al usuario y proporciona botones para que el usuario confirme o niegue el mensaje.
  - **showInputDialog(Component parentComponent, Object message)**: muestra un mensaje al usuario y proporciona un campo de texto para que el usuario introduzca una entrada.
  - **showOptionDialog(Component parentComponent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object initialValue)**: proporciona una ventana emergente personalizable que puede incluir un mensaje, un título, un icono y una serie de botones que se especifican en el parámetro options.

Ventanas emergentes - JOptionPane

Ejemplos de uso de JOptionPane:

```
JOptionPane.showMessageDialog(null, "Mensaje de información", "Info", JOptionPane.INFORMATION_MESSAGE);
JOptionPane.showMessageDialog(null, "Mensaje de error", "Error", JOptionPane.ERROR_MESSAGE);
JOptionPane.showMessageDialog(null, "Mensaje de advertencia", "Advertencia", JOptionPane.WARNING_MESSAGE);
JOptionPane.showMessageDialog(null, "¿Estás seguro?", "Pregunta", JOptionPane.QUESTION_MESSAGE);
```

```
// usar las opciones SI/NO para que el usuario seleccione una
int respuesta = JOptionPane.showConfirmDialog(null, "¿Deseas continuar?");
if (respuesta == JOptionPane.YES_OPTION)          System.out.println("Has elegido SI");
else if (respuesta == JOptionPane.NO_OPTION)      System.out.println("Has elegido NO");
else    System.out.println("No has elegido ninguna opción");
```

```
// solicitar entrada de datos
String nombre = JOptionPane.showInputDialog("Introduce tu nombre:");
System.out.println("Hola " + nombre + "!");
```

```
// usar un conjunto de opciones para que el usuario seleccione una
Object[] opciones = {"Opción 1", "Opción 2", "Opción 3"};
int seleccion = JOptionPane.showOptionDialog(null, "Elige una opción", "Opciones",
        JOptionPane.DEFAULT_OPTION, JOptionPane.INFORMATION_MESSAGE, null, opciones, opciones[0]);

System.out.println("Has elegido: " + opciones[seleccion]);
```

Ventanas emergentes - JDialog

JDialog es una ventana emergente que puede contener cualquier tipo de componente Swing. A diferencia de JOptionPane, que es una ventana emergente predefinida, JDialog es una ventana emergente completamente personalizable. Puedes agregarle cualquier componente Swing, como botones, etiquetas, paneles, etc. También puedes hacer que sea **modal** o no modal. Una ventana modal es una ventana que bloquea el acceso a otras ventanas hasta que se cierra.

```
JDialog dialog = new JDialog();
dialog.setTitle("Simple JDialog");
dialog.setSize(300, 200);
dialog.setLayout(new FlowLayout());
dialog.add(new JLabel("Esto es un JDialog sencillo y modal");
dialog.setLocationRelativeTo(null); // centra el JDialog en la pantalla
dialog.add(new JLabel("Haz clic en un botón"));

JButton botonOK = new JButton("OK");
JButton botonCancelar = new JButton("Cancelar");
botonOK.addActionListener( e -> System.out.println("Has pulsado OK"));
botonCancelar.addActionListener( e -> System.out.println("Has pulsado Cancelar"));
dialog.add(botonOK );
dialog.add(botonCancelar);
dialog.setModal(true);
dialog.setVisible(true);
```

Ventanas emergentes - JPopupMenu

JPopupMenu es una ventana emergente que se utiliza generalmente para proporcionar un menú contextual al usuario. Un menú contextual es un menú que aparece cuando el usuario realiza una acción específica, como hacer clic con el botón derecho. Un JPopupMenu puede contener cualquier tipo de componente Swing, pero generalmente contiene elementos de menú (JMenuItem), que son básicamente botones.

```
// crear el JPopupMenu y las opciones de color
JPopupMenu colorMenu = new JPopupMenu();

JMenuItem rojo = new JMenuItem("Rojo");
rojo.addActionListener(e -> colorActual = Color.RED);
JMenuItem verde = new JMenuItem("Verde");
verde.addActionListener(e -> colorActual = Color.GREEN);
JMenuItem azul  = new JMenuItem("Azul");
azul.addActionListener(e -> colorActual = Color.BLUE);

colorMenu.add(rojo);
colorMenu.add(verde);
colorMenu.add(azul);
```

JAVA Swing:
JFileChooser y JColorChooser

JColorChooser

JColorChooser: Este es un componente que proporciona una interfaz de usuario para la selección de colores. Puede agregarse a cualquier contenedor, pero a menudo se usa en un diálogo emergente. El método estático **showDialog** de JColorChooser muestra un diálogo de selección de color. A su vez, dicho método nos devolverá una instancia de la clase Color, que será el color seleccionado por el usuario. Si por el contrario, el usuario cancela el diálogo cerrando la ventana, el método devolverá un null.

```
public static Color showDialog(Component parent, String title, Color initialColor) { ...
}
```

```
Color colorSeleccionado = Color.BLACK;

...

// botón para abrir el JColorChooser
JButton boton = new JButton("Elegir color");
boton.addActionListener(e -> {
    colorSeleccionado = JColorChooser.showDialog(null, "Elige un color", colorSeleccionado);
    // si elegimos algún color
    if (colorSeleccionado != null) {
        //acción a realizar teniendo almacenado un color en colorSeleccionado
        ...
    }
});
```

JFileChooser

JFileChooser: Este es un componente de Swing que proporciona una ventana para navegar por el sistema de archivos del usuario y seleccionar un archivo o directorio. El método **showOpenDialog** de JFileChooser muestra un diálogo de apertura de archivos, y el método **showSaveDialog** muestra un diálogo de guardado de archivos. Estos diálogos emergentes son modal y bloquean el acceso a la aplicación hasta que el usuario haga una selección o cancele el diálogo.

```
//añadimos JButton en la parte inferior
JButton botonSeleccionarImagen = new JButton("Selecciona una Imagen");
add(botonSeleccionarImagen, BorderLayout.SOUTH);

//evento al pulsar el botón
botonSeleccionarImagen.addActionListener(e -> {
    JFileChooser chooser = new JFileChooser();
    int opcion = chooser.showOpenDialog(this);
    if (opcion == JFileChooser.APPROVE_OPTION) {
        File file = chooser.getSelectedFile();
        cargarImagen(file); //método que almacena una imagen en un JLabel
    }
});
```

JAVA Swing:
Ejercicios finales

Ejercicios finales JAVA Swing

1. Crear el juego del 3 en raya en JAVA Swing, para ello se deben utilizar 9 botones que se corresponden con las 9 posibles jugadas. El juego será para 2 judadores(azul y rojo). Al pulsar por primera vez en uno de los 9 posibles botones, se fijará el fondo del botón en azul(jugador1) y se deberá bloquear el botón, más tarde al pulsar en uno de los 8 posibles botones restantes, se fijará el fondo del botón en color rojo(jugador2) y se bloqueará de nuevo dicho botón. De este modo, se deberán ir intercambiando los turnos hasta que se completen los posiciones o alguno de los jugadores haga tres en raya. Si alguno de los jugadores gana, o la partida finaliza sin un vencedor(empate), se deberá mostrar un mensaje de retroalimentación y se reiniciará el juego.
2. Crear un cronómetro con segundos y centésimas. Deberá tener 2 JButton, para iniciar y reiniciar el tiempo. Al pulsar el botón de iniciar, dicho botón permitirá pausar el tiempo, mientras que al pulsar en reiniciar, el contador pasará a estar a cero y se detendrá. Ampliar la aplicación para que si pausamos con el contador de centésimas en 00, nos muestre un mensaje de diálogo.

Ejercicios finales JAVA Swing

3. Desarrollar una aplicación de mecanografía básica que permita a los usuarios comprobar su habilidad al teclear. La aplicación consiste en mostrar al usuario letras aleatorias que deberá teclear correctamente, se podrá definir la cantidad de letras a mostrar a través de una constante en la propia clase. Las características de la aplicación serán las siguientes:
   - 3.1. Mostrar una letra aleatoria en una etiqueta en el centro de la ventana.
   - 3.2. Incluir un botón que permita iniciar o reiniciar el juego.
   - 3.3. Cronometrar el tiempo que tarda el usuario en teclear las letras.
   - 3.4. Contar los aciertos del usuario (número de letras tecleadas correctamente).
   - 3.5. Si el usuario se equivoca al teclear una letra, mostrar la siguiente sin dar opción de corrección.

AMPLIACIÓN: Al terminar la partida, mostrar un cuadro de diálogo solicitando el nombre del usuario para así poder almacenar la puntuación en una BD. Además, tras introducir el nombre, luego se mostrará otro cuadro de diálogo que muestre su puntuación y tiempo y un botón para que el usuario pueda ver las 10 mejores puntuaciones registradas. Se mostrarán en primer lugar los jugadores con más aciertos, si dos jugadores tienen los mismos aciertos, se mostrará primero el que haya tardado menos tiempo

Ejercicios finales JAVA Swing

4. Crear una aplicación en Swing que permita observar el funcionamiento del algoritmo de ordenación quicksort visto en la unidad 4 del curso. Para ello se pide que se muestren por pantalla 40 rectángulos de alturas distintas. A partir de dichos rectángulos, se deberá proceder a ordenar dichos rectángulos en base a su altura, de modo que a la izquierda quede el rectángulo de menos altura y a la derecha el de mayor altura. Cada modificación sobre la colección de rectángulos deberá ser visible. Se recomiendo utilizar un delay de 100ms en cada modificación sobre la colección, para ello se puede utilizar **Thread.sleep(long ms)**.

Datos del ejemplo de prueba:

- 40 rectángulos
- 20x20 rectángulo más pequeño
- 20x800 rectángulo más grande
- separación entre rectángulos: 5
- Intervalo: 100ms

Introducción a Maven

Introducción a Maven

Maven es una herramienta de gestión de proyectos y comprensión de software ampliamente utilizada en el ecosistema Java. Proporciona un marco estándar para construir, empaquetar, y desplegar aplicaciones Java, y también ayuda en la gestión de dependencias y plugins.

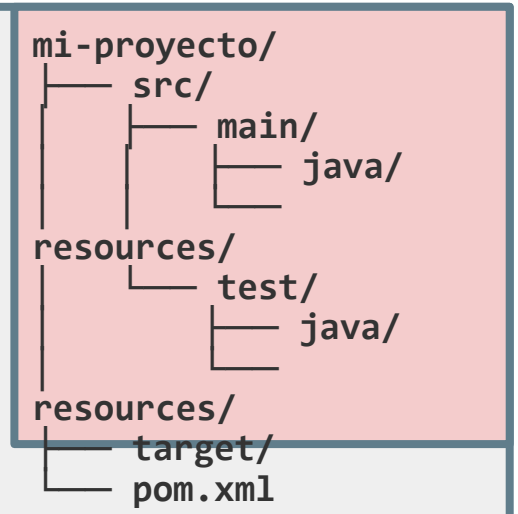
Conceptos clave de Maven:

- **Archivo pom.xml**: Es el núcleo de la configuración de Maven. Cada proyecto Maven tiene un archivo llamado pom.xml que describe el proyecto, sus dependencias, los plugins que utiliza, entre otros detalles.
- **Gestión de dependencias**: Una de las características de Maven es su capacidad para manejar dependencias. Puedes especificar las bibliotecas de las que tu proyecto depende en el pom.xml y Maven las descargará y las incluirá en el classpath automáticamente.
- **Ciclo de vida**: Maven tiene un "ciclo de vida de construcción" que es una secuencia de fases que definen los pasos para construir y desplegar el proyecto.

Estructura de carpetas de un proyecto Maven

En un proyecto Maven, la organización de los archivos y carpetas sigue una estructura estándar para facilitar la comprensión y el manejo del proyecto. Los archivos .java se colocan según su tipo, es decir, si son archivos de código fuente o archivos de prueba.

```
NombreDelProyecto/              // raíz del proyecto
|-- src/                        // carpeta de código fuente
|   |-- main/
|   |   |-- java/               // código fuente del proyecto
|   |   |   |-- com/
|   |   |       |-- ejemplo/
|   |   |           |-- MiClase.java
|   |   |-- resources/          // recursos de la aplicación
|   |-- test/
|   |   |-- java/               // código fuente de las pruebas
|   |   |   |-- com/
|   |   |       |-- ejemplo/
|   |   |           |-- MiClaseTest.java
|   |   |-- resources/          // recursos para las pruebas
|-- target/                     // carpeta donde se almacenarán los archivos compilados
|-- pom.xml                     // archivo de configuración de Maven
```



Archivo pom.xml

El archivo pom.xml (Project Object Model) es un archivo XML que contiene información sobre el proyecto y detalles de configuración utilizados por Maven para construir el proyecto.

- **<modelVersion>**: es la versión del POM que se está utilizando.
- **<groupId>**: Identifica el ID del grupo del proyecto, a menudo una versión "invertida" del dominio de la organización (por ejemplo, "com.aulaenlanube").
- **<artifactId>**: El nombre del "artefacto" (proyecto), que también se convierte en el nombre del archivo JAR resultante (por ejemplo, "MiAplicacion").
- **<version>**: Versión del artefacto, útil para la gestión de dependencias.
- **<packaging>**: Tipo de empaquetado del proyecto. Puede ser "jar", "war", "ear", etc. Si se omite, por defecto es "jar".
- **<properties>**: Un conjunto de propiedades que se pueden utilizar en el pom.xml o en el código fuente. Por ejemplo, puedes especificar la versión de Java aquí.
- **<dependencies>**: Contiene todas las dependencias necesarias para el proyecto.

Ejemplos pom.xml

Gestión de Dependencias: Contiene todas las dependencias necesarias para construir y ejecutar el proyecto. Cada dependencia se especifica en un bloque **<dependency>**.

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.26</version>
    </dependency>
</dependencies>
```

Repositorios: Define de dónde Maven debe descargar las dependencias. Por defecto, Maven utilizará el repositorio central de Maven.

```
<repositories>
    <repository>
        <id>central</id>
        <url>https://repo.maven.apache.org/maven2</url>
    </repository>
</repositories>
```

Fases del ciclo de vida de Maven

Maven tiene un "ciclo de vida de construcción" que es una secuencia de fases que definen los pasos para construir y desplegar el proyecto. Algunas fases clave son:

- **validate**: Valida que el proyecto es correcto y toda la información necesaria está disponible.
- **compile**: Compila el código fuente del proyecto.
- **test**: Prueba el código fuente compilado.
- **package**: Empaqueta el código en un formato distribuible (como JAR).
- **install**: Instala el paquete en el repositorio local para su uso como dependencia en otros proyectos.
- **deploy**: Copia el paquete final en un repositorio remoto para compartirlo con otros desarrolladores.
- **clean**: Elimina todos los archivos compilados de la carpeta target.

Descargar e instalar Maven

Introducción a JAVA FX

Introducción a JavaFX

JavaFX es un framework que facilita el desarrollo de aplicaciones de escritorio con interfaz gráfica en Java. En el núcleo de JavaFX se encuentra el conjunto de bibliotecas de gráficos que permiten a los desarrolladores diseñar y construir interfaces de usuario con una amplia gama de características visuales. Esto incluye la capacidad de crear formas 2D y 3D, aplicar efectos especiales, trabajar con imágenes y texto, así como incorporar contenido multimedia, como audio y vídeo.

JavaFX fue inicialmente desarrollado y mantenido por Oracle a partir de Java 8(2014), pero a partir de Java 11, Oracle descontinuó su soporte para JavaFX y lo separó del JDK. Desde entonces, JavaFX se ha convertido en un proyecto de código abierto bajo el nombre de **OpenJFX**, y es mantenido por la comunidad.

Repositorio de GitHub de OpenJFX

OpenJFX API

OpenJFX dispone de una API que proporciona clases e interfaces que son el núcleo para crear aplicaciones, las clases principales son las siguientes:

- **Stage**: Es la ventana principal de la aplicación.
- **Scene**: Contiene todos los elementos que se muestran en una Stage.
- **Node**: Elementos individuales como botones, tablas, etc.

```
public class HolaMundo extends Application {
    @Override
    public void start(Stage ventanaPrincipal) {
        Button boton = new Button("Saludar");
        boton.setOnAction(e -> System.out.println("Hola mundo desde JavaFX!"));

        Scene escena = new Scene(boton, 200, 100);
        ventanaPrincipal.setScene(escena);
        ventanaPrincipal.setTitle("HolaMundoFX");
        ventanaPrincipal.show();
    }
    public static void main(String[] args) {  launch(args);  }
} // ejemplo de cómo crear una simple ventana con un botón
```

Aquitectura MVC

MVC es el acrónimo de Model-View-Controller (Modelo-Vista-Controlador). Es un patrón de diseño de software que divide una aplicación en tres componentes interconectados. Este patrón es particularmente útil en aplicaciones con interfaces gráficas de usuario, como las desarrolladas con JavaFX. Veamos cada uno de los componentes principales del MVC:

- **Model (Modelo):**
  - Representa la lógica de negocio y los datos en la aplicación.
  - No sabe nada sobre la interfaz de usuario.
  - Cualquier cambio en el modelo se comunica a la vista para que se actualice.
- **View (Vista):**
  - Muestra la información al usuario.
  - Obtiene los datos del modelo para presentarlos.
  - Envía las acciones del usuario (como clics de botón, inserción de texto) al controlador.
- **Controller (Controlador):**
  - Intermediario entre el modelo y la vista.
  - Toma las acciones del usuario desde la vista, procesa (o pide procesar) esos datos a través del modelo y devuelve la salida para que la vista la muestre.

Diseño de la UI - FXML

FXML es un lenguaje de marcado basado en XML que se utiliza para construir la interfaz de usuario de una aplicación JavaFX. Utilizar FXML separa la lógica de la interfaz de usuario de la lógica de la aplicación, veamos un ejemplo de archivo FXML que describe una interfaz de usuario con un botón:

```
<?xml version="1.0" encoding="UTF-8"?>
<AnchorPane xmlns:fx="http://javafx.com/fxml"
            fx:controller="MiControlador">
    <Button fx:id="miBoton" text="Haz clic aquí"/>
</AnchorPane>
```

El controlador es una clase Java asociada a un archivo FXML. Utiliza anotaciones para enlazar los componentes de la interfaz de usuario con variables y métodos. En el ejemplo anterior, fx:controller="MyController" indica que MyController es la clase controladora.

```
public class MiControlador {
    @FXML private Button miBoton;
    @FXML public void initialize() {
        miBoton.setOnAction(e -> System.out.println("Botón pulsado"));
    }
}
```

Ejemplo de MVC con JavaFX

Supongamos un simple contador que aumenta cada vez que se pulsa un botón.

```
// modelo
public class Contador{
    private int n = 0;
    public void incrementar() {   n++;          }
    public int get()          {   return n;     }
}
// vista(FXML)
<Button fx:id="botonIncrementar" text="Incrementar" />
// controlador
public class ContadorControlador {
    @FXML private Button botonIncrementar;
    private Contador contador;
    @FXML
    public void initialize() {
        contador = new Contador();
        botonIncrementar.setOnAction(e -> {
            contador.incrementar();
            System.out.println("Contador: " + contador.get());
        });
    }
}
```

JAVA FX: Ejemplos adicionales

Ejemplo App sin FXML

Veamos un ejemplo adicional de una App típica HolaMundo con OpenJFX sin el uso de FXML.

```
public class App extends Application {
    public void start(Stage ventanaPrincipal) {
        TextField texto = new TextField();
        Button boton = new Button("Saludar");

        boton.setOnAction(e -> {
            String nombre = texto.getText();
            Alert alerta = new Alert(AlertType.INFORMATION);
            alerta.setTitle("Saludo Personalizado");
            alerta.setContentText("Hola mundo " + nombre);
            alerta.show();
        });

        VBox vbox = new VBox(10);
        vbox.setAlignment(Pos.CENTER);
        vbox.setPadding(new Insets(20, 20, 20, 20));
        vbox.getChildren().addAll(texto, boton);

        Scene scene = new Scene(vbox, 300, 150);
        ventanaPrincipal.setTitle("Ejemplo sin FXML");
        ventanaPrincipal.setScene(scene);
        ventanaPrincipal.show();
    }
    //... main
}
```

Ejemplo App con FXML

Veamos el mismo ejemplo de la diapositiva anterior pero utilizando un fichero FXML.

```
public class ControladorVentana {
    @FXML private TextField texto;
    @FXML private Button boton;
    @FXML public void saludar() {
        String nombre = texto.getText();
        Alert alert = new Alert(AlertType.INFORMATION);
        alert.setTitle("Saludo Personalizado");
        alert.setHeaderText(null);
        alert.setContentText("Hola mundo " + nombre);
        alert.show();
    }
}
```

```
public class AppConFXML extends Application {
    @Override
    public void start(Stage ventanaPrincipal) throws Exception {
        Parent root = FXMLLoader.load(getClass().getResource("ventana.fxml"));
        Scene scene = new Scene(root, 300, 150);
        ventanaPrincipal.setTitle("Ejemplo con FXML");
        ventanaPrincipal.setScene(scene);
        ventanaPrincipal.show();
    }
    //... main
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<VBox alignment="CENTER" spacing="10.0" xmlns="http://javafx.com/javafx"
xmlns:fx="http://javafx.com/fxml" fx:controller="ControladorVentana">
    <TextField fx:id="texto" promptText="Introduce tu nombre" />
    <Button fx:id="boton" text="Saludar" onAction="#saludar" />
</VBox>
```

ventana.fxml

JAVA FX: Clase Node

La clase Node

La clase **Node** en JavaFX es la superclase de todos los elementos que pueden ser renderizados en una escena de JavaFX. Esto incluye formas básicas como círculos, rectángulos y líneas, así como controles de usuario más complejos como botones, tablas y gráficos. Incluso los contenedores como Pane y GridPane son subclases de Node. Veamos sus características principales:

- **Posición y Tamaño**: propiedades que determinan su posición (x, y, z) y tamaño (width, height).
- **Estilo y Apariencia**: Los nodos pueden ser estilizados usando JavaFX CSS. También puedes aplicar todo tipos de efectos gráficos avanzados, como sombras, desenfoques, etc.
- **Eventos**: La clase Node tiene una rica API para manejar eventos, como clics del mouse, entrada del teclado, y otros tipos de interacciones del usuario.
- **Transformaciones**: Puedes aplicar transformaciones geométricas como traslación, rotación y escalado a nodos.
- **Visibilidad**: Los nodos pueden ser visibles o invisibles, y también pueden ser deshabilitados.
- **Jerarquía de Nodos**: Un nodo puede tener un nodo padre y varios nodos hijos, formando una estructura de árbol llamada gráfico de escena.

Componentes de la clase Node

Cada uno de estos componentes hereda de Node, lo que significa que todos ellos pueden ser añadidos a una escena, transformados, estilizados, y pueden interactuar con eventos. Esta herencia común facilita la consistencia y la reutilización en el diseño de interfaces de usuario en JavaFX.

Contenedores (Layout Panes)

Pane: Clase base para todos los paneles de diseño.
AnchorPane: Permite anclar nodos a los bordes.
HBox / VBox: Contenedores para diseño horizontal y vertical, respectivamente.
GridPane: Para diseño en forma de cuadrícula.
BorderPane: Para diseño con áreas específicas como TOP, BOTTOM, LEFT, RIGHT, y CENTER.
StackPane: Apila nodos uno encima del otro.
FlowPane: Coloca nodos en un flujo, similar a como el texto fluye en un párrafo.
TilePane: Coloca nodos en una cuadrícula de azulejos.

Controles de Usuario

Button: Botón estándar.
Label: Etiqueta de texto.
TextField / TextArea: Campos de texto para entrada del usuario.
CheckBox / RadioButton: Para selecciones múltiples y únicas, respectivamente.
ComboBox / ChoiceBox: Para una lista desplegable de elementos.
Slider: Para seleccionar un valor de un rango.
ProgressBar / ProgressIndicator: Para mostrar el progreso de una tarea.
TableView / ListView / TreeView: Para mostrar datos en forma tabular, lista o jerárquica.

Componentes de la clase Node

Cada uno de estos componentes hereda de Node, lo que significa que todos ellos pueden ser añadidos a una escena, transformados, estilizados, y pueden interactuar con eventos. Esta herencia común facilita la consistencia y la reutilización en el diseño de interfaces de usuario en JavaFX.

Formas Geométricas

Circle: Un círculo.
Rectangle: Un rectángulo.
Polygon: Una figura geométrica con más de tres lados.
Line: Una línea recta.
Ellipse: Una elipse.
Path: Una ruta compuesta por múltiples segmentos de línea y curvas.

Otros

ImageView: Para mostrar imágenes.
MediaView: Para mostrar contenido multimedia como videos.
WebView: Para mostrar contenido web.
Text: Para mostrar texto con estilos más avanzados que Label.
Canvas: Para dibujo de bajo nivel.
Group: Un contenedor especial que simplemente agrupa otros nodos sin aplicar un diseño específico.

Métodos importantes clase Node

- **setTranslateX()**, **setTranslateY()**: Para mover el nodo a lo largo de los ejes X y Y.
- **setRotate()**: Para rotar el nodo.
- **setScaleX()**, **setScaleY()**: Para escalar el nodo en los ejes X y Y.
- **setVisible(boolean)**: Para establecer la visibilidad del nodo.
- **setDisable(boolean)**: Para habilitar o deshabilitar el nodo para la interacción del usuario.

```
public class EjemploNode extends Application {
    @Override
    public void start(Stage stage) {
        Button boton = new Button("Pulsa");  // button es un Node
        boton.setTranslateX(50);  // mover 50 píxeles a la derecha
        boton.setTranslateY(50);  // mover 50 píxeles hacia abajo
        boton.setRotate(45);      // rotar 45 grados
        boton.setScaleX(1.5);     // escalar 1.5 veces en el eje X

        Scene scene = new Scene(boton, 500, 500);
        stage.setScene(scene);
        stage.show();
    }
}
```

La clase Parent

La clase Parent en JavaFX es una subclase de la clase Node y sirve como un elemento contenedor para otros nodos. En otras palabras, un objeto Parent puede tener nodos hijos, lo que facilita la creación de jerarquías de elementos en la interfaz gráfica de usuario (GUI). La clase Parent es abstracta, es por ello que lo más común es usar alguna de sus subclases, como Group, Region, o las diferentes clases de contenedores (Pane, AnchorPane, VBox, HBox, etc.).

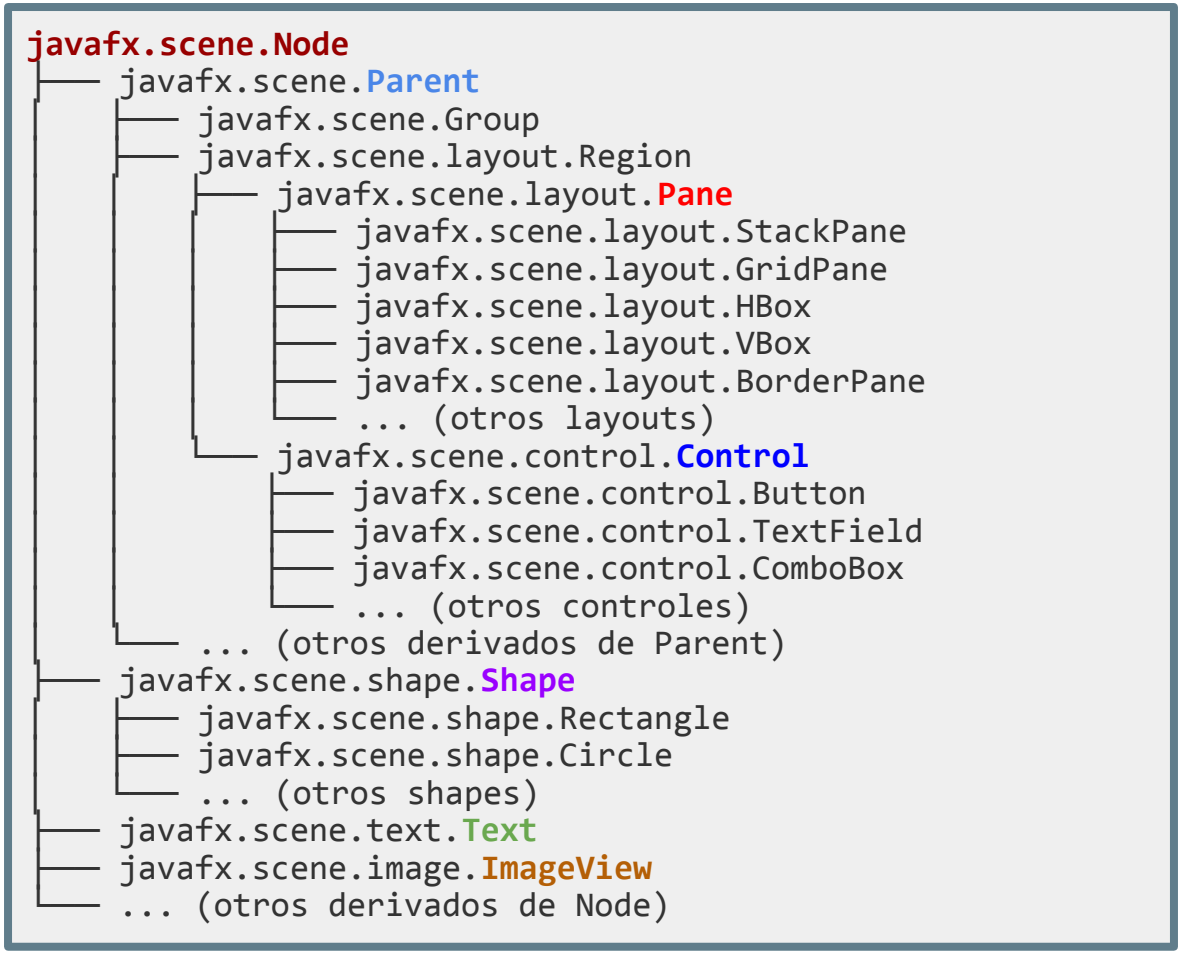
Características principales:

- **Jerarquía de Nodos**: Un Parent puede contener múltiples nodos hijos, que a su vez pueden ser también objetos Parent, permitiendo así crear una jerarquía de nodos.
- **Transformaciones**: Las transformaciones aplicadas a un objeto Parent afectan a todos sus hijos.
- **Estilo CSS**: Un Parent y sus nodos hijos pueden ser estilizados utilizando hojas de estilo en cascada (CSS).
- **Eventos**: Un objeto Parent puede capturar eventos que se propagan a través de la jerarquía de nodos, lo que permite manejar eventos en un nivel más general si es necesario.

La clase Parent

```
public class EjemploParent extends Application {

    @Override
    public void start(Stage ventanaPrincipal) {
        Button boton1 = new Button("Botón 1");
        Button boton2 = new Button("Botón 2");

        VBox vbox = new VBox();     // VBox es una subclase de Parent
        vbox.getChildren().addAll(boton1, boton2);
        boton1.setRotate(45); //aplicamos una rotación de 45 al boton1
        vbox.setRotate(45); //aplicamos una rotación de 45 grados al vbox

        Scene scene = new Scene(vbox, 200, 100); // La raíz de la escena es un Parent
        ventanaPrincipal.setTitle("Ejemplo de Parent");
        ventanaPrincipal.setScene(scene);
        ventanaPrincipal.show();
    }

    public static void main(String[] args) {
        launch();
    }
}
```

Jerarquía de clases de Node



JAVA FX: Layouts

Java FX: Layouts

Los layouts en JavaFX son contenedores que organizan los componentes o nodos de una forma particular en la interfaz gráfica.

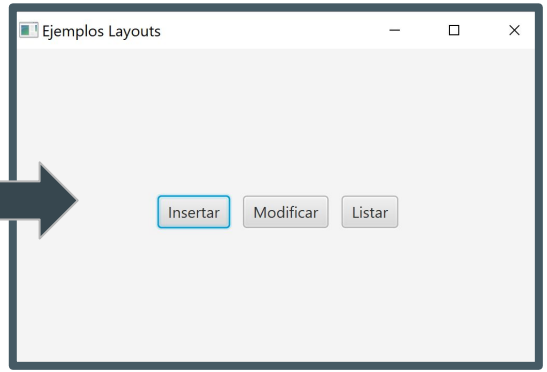
- **Pane:** Un Pane es el layout más simple. No tiene un esquema de diseño específico y permite el posicionamiento y el dimensionamiento manual de los nodos.
- **StackPane**: Organiza los nodos en una pila, uno encima del otro.
- **VBox**: Organiza los nodos verticalmente, uno debajo del otro.
- **HBox**: Organiza los nodos horizontalmente, uno al lado del otro.
- **GridPane**: Organiza los nodos en una rejilla bidimensional, con filas y columnas.
- **BorderPane**: Divide el área en cinco regiones: superior, inferior, izquierda, derecha y centro. Cada región puede contener un solo nodo, como otro layout, que a su vez puede contener múltiples nodos.
- **FlowPane**: Coloca los elementos en una fila o en una columna y los mueve a la siguiente fila o columna cuando se alcanza el borde del layout.
- **TilePane**: Coloca los nodos en una cuadrícula de baldosas, donde cada nodo tiene el mismo tamaño.
- **AnchorPane**: Permite "anclar" los nodos a los bordes del layout. Esto puede ser útil para crear diseños más flexibles que se ajustan bien al cambio de tamaño de la ventana.

Java FX: Layouts

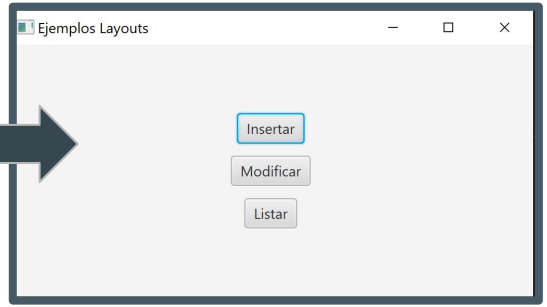
Ejemplos:

```
Button btn1 = new Button("Insertar");
Button btn2 = new Button("Eliminar");
Button btn3 = new Button("Listar");
```

```
HBox hbox1 = new HBox(10);
hbox1.getChildren().addAll(btn1, btn2, btn3);
hbox1.setAlignment(Pos.CENTER);
hbox1.setPadding(new Insets(20, 20, 20, 20));
```




```
VBox vbox1 = new VBox(10);
vbox1.getChildren().addAll(btn1, btn2, btn3);
vbox1.setAlignment(Pos.CENTER);
vbox1.setPadding(new Insets(20, 20, 20, 20));
```



Java FX: Layouts

En Java FX es bastante típico anidar diferentes tipos de layouts para lograr diseños de interfaz de usuario más complejos. Un layout puede contener otros layouts como hijos, permitiendo una gran flexibilidad. Veamos un ejemplo:

Supongamos que quieres tener una barra de botones en la parte superior y un área principal debajo de ella. Podrías usar un VBox como layout principal y añadir un HBox para la barra de botones y un StackPane (o cualquier otro layout) para el área principal.

```
Button btn1 = new Button("Insertar");
Button btn2 = new Button("Eliminar");
Button btn3 = new Button("Mostrar");
HBox hbox = new HBox();
hbox.getChildren().addAll(btn1, btn2, btn3);
Label etiqueta = new Label("Etiqueta dentro de un FlowPane");
FlowPane flowPane = new FlowPane();
flowPane.getChildren().add(etiqueta);
VBox vbox = new VBox();
vbox.getChildren().addAll(hbox, flowPane);
Scene scene = new Scene(vbox, 300, 250);
```

JAVA FX: Binding y propiedades

Binding y propiedades

En JavaFX, el **binding** (enlace) y las **propiedades** son conceptos poderosos que permiten que las variables y los elementos de la UI estén sincronizados. Estos conceptos son esenciales para crear interfaces de usuario reactivas y fácilmente mantenibles.

Propiedades: una propiedad en JavaFX no es más que un contenedor especial que puede contener un valor, como un int, un String, un objeto, etc. Las propiedades son objetos que envuelven un valor y permiten ser observados, lo que significa que puedes ser notificado cuando su valor cambia.

```
//podemos declarar propiedades para encapsular Strings, enteros, reales y booleanos
StringProperty s1 = new SimpleStringProperty("Hola");
StringProperty s2 = new SimpleStringProperty("Mundo");
IntegerProperty a = new SimpleIntegerProperty(5);
DoubleProperty euros = new SimpleDoubleProperty(1.123);
BooleanProperty condicion = new SimpleBooleanProperty(true);
```

Binding: permite mantener dos propiedades sincronizadas.

```
// sincronizamos las propiedades s1 y s2
s1.bind(s2);
```

Tipos de Binding

El binding te permite mantener dos propiedades sincronizadas, existen varios tipos de binding en JavaFX:

- **Unidireccional** (bind): Cuando una propiedad cambia, la otra también lo hace, pero no al revés.

```
StringProperty s1 = new SimpleStringProperty("Hola");
StringProperty s2 = new SimpleStringProperty("Mundo");
System.out.println("Antes del binding: " + s2.get());  // imprime "Mundo"
s2.bind(s1); // aquí es donde el binding ocurre
s1.set("Adiós"); // cambia el valor de s1
System.out.println("Después del binding: " + s2.get()); // imprime "Adiós", s2 está vinculado a s1
s2.set("Fin"); // ERROR, tras en vínculo, no se puede cambiar directamente s2
```

- **Bidireccional** (bindBidirectional): Ambas propiedades se actualizan cuando cualquiera de ellas cambia.

```
StringProperty s1 = new SimpleStringProperty("Hola");
StringProperty s2 = new SimpleStringProperty("Mundo");
System.out.println("s1 antes del cambio: " + s1.get()); // imprime "Hola"
s1.bindBidirectional(s2); // binding bidireccional
s2.set("Adiós"); // cambia el valor de s2
System.out.println("s1 después del cambio: " + s1.get()); // imprime "Adiós"
s1.set("Fin");
System.out.println("s1 después del cambio: " + s2.get()); // imprime "Fin"
```

Tipos de Binding

Binding de expresiones: Puedes crear expresiones más complejas, como sumar propiedades, multiplicarlas, crear condiciones, etc.

```
IntegerProperty a = new SimpleIntegerProperty(5);
IntegerProperty b = new SimpleIntegerProperty(10);
IntegerProperty sum = new SimpleIntegerProperty();

// sum se actualizará automáticamente cuando 'a' o 'b' cambien
sum.bind(a.add(b));

// imprime 15 (5 + 10)
System.out.println("Suma inicial: " + sum.get());

// cambia el valor de 'a'
a.set(20);

// imprime 30 (20 + 10) porque sum está vinculado a la suma de 'a' y 'b'
System.out.println("Suma después del cambio: " + sum.get());

sum.set(45);  // ERROR
```

Especializaciones Clase Binding

Las clases NumberBinding, DoubleBinding, StringBinding y BooleanBinding son especializaciones de la clase abstracta Binding en JavaFX. Estas clases permiten crear vínculos entre propiedades.

- **NumberBinding** es una clase abstracta que representa un valor numérico calculado que depende de uno o más observables.
- **DoubleBinding** es una especialización de NumberBinding para valores de punto flotante.
- **StringBinding** crea cadenas de texto calculadas que dependen de uno o más observables.
- **BooleanBinding** está diseñada específicamente para manejar propiedades booleanas.

```
IntegerProperty a = new SimpleIntegerProperty(5);
IntegerProperty b = new SimpleIntegerProperty(10);
NumberBinding suma = a.add(b);
System.out.println(suma.getValue());  // 15
a.setValue(20);
System.out.println(suma.getValue());  // 30

DoubleProperty x = new SimpleDoubleProperty(2.5);
DoubleProperty y = new SimpleDoubleProperty(4.5);
DoubleBinding resultado = x.multiply(y);
System.out.println(resultado.get());  // 11.25
```

Especializaciones Clase Binding

```
// ejemplos con String
StringProperty nombre = new SimpleStringProperty("Juan");
StringProperty apellido = new SimpleStringProperty("Pérez");
StringBinding nombreCompleto = (StringBinding)Bindings.concat(nombre, " ", apellido);
System.out.println(nombreCompleto.get());  // Juan Pérez
nombre.set("Ana");
System.out.println(nombreCompleto.get());  // Ana Pérez

// ejemplos con Boolean
BooleanProperty condicionA = new SimpleBooleanProperty(true);
BooleanProperty condicionB = new SimpleBooleanProperty(false);
BooleanBinding ambasVerdaderas = condicionA.and(condicionB);
System.out.println(ambasVerdaderas.get());  // false
condicionB.set(true);
System.out.println(ambasVerdaderas.get());  // true

BooleanBinding cualquieraVerdadera = condicionA.or(condicionB);
BooleanBinding noA = condicionA.not();

System.out.println(cualquieraVerdadera.get());  // true
System.out.println(noA.get());  // false
```

Clase Bindings

La clase **Bindings** en JavaFX es una clase de utilidad que ofrece varios métodos estáticos para crear y manipular instancias de enlace (binding). Veamos algunos métodos útiles:

- **add(ObservableNumberValue a, ObservableNumberValue b)**: Realiza la suma de a y b.
- **subtract(ObservableNumberValue a, ObservableNumberValue b)**: Realiza la resta de a y b.
- **multiply(ObservableNumberValue a, ObservableNumberValue b)**: Realiza la multiplicación de a y b.
- **divide(ObservableNumberValue a, ObservableNumberValue b)**: Realiza la división de a y b.
- **when(BooleanExpression condition)**: Crea un enlace condicional. Puedes usar then y otherwise para definir los dos posibles resultados.

```
IntegerProperty a = new SimpleIntegerProperty(5);
IntegerProperty b = new SimpleIntegerProperty(2);
NumberBinding suma = Bindings.add(a, b);
System.out.println("Suma: " + suma.getValue()); // Suma: 7

BooleanProperty condicion = new SimpleBooleanProperty(true);
StringProperty resultado = new SimpleStringProperty("");
resultado.bind(Bindings.when(condicion).then("Verdadero").otherwise("Falso"));
System.out.println("Resultado: " + resultado.getValue()); // Resultado: Verdadero
```

Ejemplos Clase Bindings

```
IntegerProperty num1 = new SimpleIntegerProperty(10);
IntegerProperty num2 = new SimpleIntegerProperty(20);
IntegerProperty valorCondicion = new SimpleIntegerProperty(50);

NumberBinding multiplication = num1.multiply(num2);

NumberBinding result = Bindings.when(multiplication.greaterThan(100))
                                .then(multiplication.add(valorCondicion))
                                .otherwise(multiplication);

System.out.println("Resultado inicial: " + result.getValue());

num1.set(5);
System.out.println("Después de cambiar num1: " + result.getValue());

num2.set(25);
System.out.println("Después de cambiar num2: " + result.getValue());

valorCondicion.set(100);
System.out.println("Después de cambiar valorCondicion: " + result.getValue());
```

Ejemplos Clase Bindings

```
IntegerProperty num1 = new SimpleIntegerProperty(10);
IntegerProperty num2 = new SimpleIntegerProperty(20);
IntegerProperty valorCondicion = new SimpleIntegerProperty(50);

NumberBinding multiplication = num1.multiply(num2); // operación de multiplicación entre num1 y num2

NumberBinding result = Bindings.when(multiplication.greaterThan(100)) // expresión condicional
                              .then(multiplication.add(valorCondicion))
                              .otherwise(multiplication);

System.out.println("Resultado inicial: " + result.getValue()); // (10*20)=200; 200>100 → 200+50 = 250

num1.set(5); // modificamos num1
System.out.println("Después de cambiar num1: " + result.getValue()); // (5*20)= 100; 100 no > 100 → 100

num2.set(25); // modificamos num2
System.out.println("Después de cambiar num2: " + result.getValue()); // (5*25)=125; 125>100 → 125+50 =
175

valorCondicion.set(100); // modificamos valorCondicion
System.out.println("Después de cambiar valorCondicion: " + result.getValue()); // (5*25)=125; 125>100 →
125+100 = 225
```

Ejemplo de Binding con Barra de progreso

Barra de progreso: Vamos a ver cómo usar propiedades y binding en un pequeño ejemplo con una barra de progreso y un contador.

```
ProgressBar progressBar = new ProgressBar(0);
Label label = new Label("Procesando...");

DoubleProperty progressValue = new SimpleDoubleProperty(0);
progressBar.progressProperty().bind(progressValue); //vinculamos

// aumentamos el contador con un hilo (simulando una tarea en segundo plano)
new Thread(() -> {
    for (int i = 0; i <= 100; i++) {
         progressValue.set(i / 100.0);
         if (i == 100) {
             label.setText("Completado!!");
             progressBar.setStyle("-fx-accent: green");
         }
        try { Thread.sleep(100); } catch (InterruptedException e) { e.printStackTrace(); }
    }
}).start();
```

Ejemplo de Binding con conversor de monedas

Conversor de moneda: Supongamos que tienes un campo de texto para ingresar una cantidad en euros y quieres mostrar su equivalente en dólares en tiempo real.

```
VBox root = new VBox(10);
root.setPadding(new Insets(20));

TextField campoEuros = new TextField();
Label etiquetaDolares = new Label();

DoubleProperty euros = new SimpleDoubleProperty();
DoubleProperty tasaCambio = new SimpleDoubleProperty(1.08);

euros.bind(Bindings.createDoubleBinding(() -> {
            try { return Double.parseDouble(campoEuros.getText()); }
            catch (NumberFormatException e) { return 0.0; }
        }, campoEuros.textProperty()));

DoubleBinding dolares = euros.multiply(tasaCambio);
etiquetaDolares.textProperty().bind(Bindings.format("En dólares: %.2f", dolares));
root.getChildren().addAll(new Label("Euros:"), campoEuros, etiquetaDolares);
```

JAVA FX: Manejo de eventos

Eventos en JAVA FX

En JavaFX, los eventos son instancias de la clase javafx.event.Event o de sus subclases, como ActionEvent, MouseEvent, KeyEvent, entre otros. JavaFX utiliza un modelo de propagación de eventos de tipo "burbuja", en el que un evento se propaga desde el nodo donde se originó hasta la raíz del gráfico de escena (Scene Graph).

1. **Captura**: Primero, el evento viaja desde el nodo raíz hasta el nodo objetivo, dando a cada nodo en el camino la oportunidad de "capturar" el evento antes de que llegue a su destino.
2. **Objetivo**: El nodo objetivo tiene la siguiente oportunidad de manejar el evento.
3. **Burbuja**: Finalmente, el evento "burbujea" de nuevo al nodo raíz, dando a cada nodo en el camino otra oportunidad para manejarlo.

Puedes registrar un manejador de eventos utilizando los métodos **setOn[Evento]** proporcionados por la clase Node o sus subclases. Por ejemplo:

```
Button btn = new Button("Haz clic en el botón");
btn.setOnAction(e -> {
    System.out.println("Botón presionado!");
});
```

Eventos en JAVA FX

Filtrado de Eventos: También puedes registrar "filtros de eventos" que se ejecutan durante la fase de captura. Los filtros te permiten interceptar un evento antes de que llegue a su manejador de eventos final.

```
vbox.addEventFilter(MouseEvent.MOUSE_CLICKED, e -> {
    System.out.println("Has pulsado en el vbox: no has llegado al destino");
});
vbox.addEventHandler(MouseEvent.MOUSE_CLICKED, e -> {
    System.out.println("Has pulsado en el vbox: has llegado al destino");
});
```

Buenas Prácticas

1. Utilizar Lambdas: Utiliza expresiones lambda para hacer que el registro del manejador de eventos sea más conciso.
2. Separar la Lógica: Para aplicaciones más grandes, considera separar los manejadores de eventos en sus propias clases o métodos para mejorar la mantenibilidad.
3. Usar Constantes para Tipos de Eventos: Utiliza constantes para representar diferentes tipos de eventos si los mismos se utilizan en múltiples lugares.

Eventos de ratón

```
// click izquierdo
escena.setOnMouseClicked(e -> {
    if (e.getButton() == MouseButton.PRIMARY)  label.setText("Has hecho click izquierdo!");
});
// click derecho
escena.setOnMouseClicked(e -> {
    if (e.getButton() == MouseButton.SECONDARY) label.setText("Has hecho click derecho!");
});
// arrastrar
escena.setOnMouseDragged(e -> {
   label.setText("Estás arrastrando el ratón. Posición: X=" + e.getX() + ", Y=" + e.getY());
});
// girar la rueda del ratón
escena.setOnScroll(e -> {
    label.setText("Has girado la rueda. Cambio en Y: " + e.getDeltaY());
});
// evento al mover el ratón
escena.setOnMouseMoved(e -> {
    label.setText("Estás moviendo el ratón. Posición: X=" + e.getX() + ", Y=" + e.getY());
});
```

Eventos de teclado

```
// evento para detectar si se ha pulsado la tecla "ENTER"
escena.setOnKeyPressed(evento -> {
     if (evento.getCode() == KeyCode.ENTER) {
          System.out.println("Se presionó ENTER");
     }
});
// evento para detectar si se ha pulsado la LETRA "A"
escena.setOnKeyPressed(evento -> {
     if (evento.getCode() == KeyCode.A) {
          System.out.println("Se presionó la tecla A");
     }
});
// evento para detectar si se ha soltado la "Barra Espaciadora"
escena.setOnKeyReleased(evento -> {
     if (evento.getCode() == KeyCode.SPACE) {
          System.out.println("Se soltó la barra espaciadora");
     }
});
```

Eventos de teclado con combinaciones de teclas

```
// evento para detectar si se ha pulsado la combinación "CTRL + S"
escena.setOnKeyPressed(evento -> {
     if (evento.isControlDown() && evento.getCode() == KeyCode.S) {
          System.out.println("Se presionó CTRL + S");
     }
});
// evento para detectar si se ha pulsado la combinación "CTRL + S" con KeyCombination
final KeyCombination combo = new KeyCodeCombination(KeyCode.S, KeyCombination.CONTROL_DOWN);
escena.addEventHandler(KeyEvent.KEY_PRESSED, evento -> {
     if (combo.match(evento)) {
          System.out.println("Combinación CTRL + S detectada");
     }
});
// evento para detectar si se ha pulsado la combinación "CTRL + SHIFT + K"
escena.setOnKeyPressed(evento -> {
     if (evento.isControlDown() && evento.isShiftDown() && evento.getCode() == KeyCode.K) {
          System.out.println("Se presionaron las teclas CTRL + SHIFT + K al mismo tiempo");
     }
});
```

Eventos de teclado con combinaciones de teclas

```
// creamos un Set para almacenar las teclas que están pulsadas
private final Set<KeyCode> teclasActivas = new HashSet<>();

@Override
public void start(Stage stage) {
     // … creación de la escena → scene
     scene.setOnKeyPressed(event -> {
          teclasActivas.add(event.getCode());  // añadir la tecla presionada al Set

          if (teclasActivas.contains(KeyCode.A) && teclasActivas.contains(KeyCode.B) &&
               teclasActivas.contains(KeyCode.C)) {
               System.out.println("Las teclas 'A', 'B', y 'C' han sido presionadas!");
          }
     });
     scene.setOnKeyReleased(event -> {
          teclasActivas.remove(event.getCode());  // eliminar la tecla liberada del Set
     });
     // … resto del método start
}
```

JAVA FX: uso de CSS

JAVA FX: uso de CSS

JavaFX permite el uso de hojas de estilo en cascada (CSS) para personalizar la apariencia y el comportamiento de la interfaz gráfica. El CSS en JavaFX es muy similar al CSS que se utiliza en el diseño web, pero tiene algunas particularidades específicas de JavaFX.

Puedes vincular una hoja de estilo a una Scene o Parent:

```
Scene escena = new Scene(contenido);
escena.getStylesheets().add(getClass().getResource("fichero.css").toExternalForm());
```

Un archivo CSS para JavaFX tiene selectores y reglas, similares a los de la web.

```
.boton-personalizado {
     -fx-background-color: #3498db;
     -fx-text-fill: white;
}
#texto-destacado {
     -fx-font-size: 20px;
     -fx-font-weight: bold;
}
```

JAVA FX: uso de CSS

Una vez definidos los estilos en el archivo CSS, puedes aplicarlos a los componentes de JavaFX mediante el método getStyleClass(). Por ejemplo:

```css
.boton-personalizado {
    -fx-background-color: #3498db;
    -fx-text-fill: white;
    -fx-border-color: red;
    -fx-border-width: 5px;
}
#texto-destacado {
    -fx-font-size: 20px;
    -fx-font-weight: bold;
    -fx-text-fill: green;
    -fx-font-family: "Arial";
}
```

```java
Button boton = new Button("Haz clic en mí");
boton.getStyleClass().add("boton-personalizado");

Text texto = new Text("Texto destacado");
texto.setId("texto-destacado");
```

Aunque JavaFX utiliza CSS, no es exactamente igual que CSS web. Las propiedades comienzan generalmente con -fx-. Por ejemplo, -fx-background-color define el color de fondo de un componente.

JAVA FX: uso de CSS

Puedes definir estilos por defecto para componentes específicos utilizando el nombre del componente como selector en tu archivo CSS. Al hacer esto, cualquier instancia del componente en tu aplicación que no tenga un estilo específicamente definido tomará el estilo por defecto definido en el CSS.

Si en algún momento deseas anular el estilo por defecto para una instancia específica de Label o Button, simplemente puedes asignarle una clase de estilo diferente o definirle estilos en línea. Por ejemplo:

```
Button boton = new Button("Pulsa");
Label label1 = new Label("Soy normal");
Label label2 = new Label("Soy especial");
label2.getStyleClass().add("etiqueta-especial");
```

```
Label {
      -fx-text-fill: #2c3e50;
      -fx-font-size: 14px;
}
Button {
      -fx-background-color: #3498db;
      -fx-text-fill: white;
      -fx-border-radius: 5px;
      -fx-background-radius: 5px;
      -fx-padding: 5px 15px;
}
.etiqueta-especial {
      -fx-text-fill: red;
}
```

Para acceder a la documentación oficial de CSS para JavaFX en el contexto del proyecto OpenJFX, puedes visitar: **JavaFX CSS Reference Guide - OpenJFX**

JAVA FX: estilos en línea

En JavaFX, puedes definir estilos en línea para un componente específico utilizando el método **setStyle**(). Los estilos definidos de esta manera se aplican directamente al componente y anulan cualquier estilo que se haya definido para ese componente en una hoja de estilo CSS vinculada.

```
Button b1 = new Button("Pulsa1");
Button b2 = new Button("Pulsa2");
Button b3 = new Button("Pulsa3");
Button b4 = new Button("Pulsa4");
b1.setStyle("-fx-background-color: red; -fx-text-fill: white;");
b2.getStyleClass().add("boton-especial");
b2.setStyle("-fx-background-color: blue;");
b3.setStyle("-fx-background-color: blue;");
b3.getStyleClass().add("boton-especial");
```

```
Button {
    -fx-background-color: green;
    -fx-text-fill: white;
    -fx-border-radius: 5px;
    -fx-background-radius: 5px;
    -fx-padding: 5px 15px;
}
.boton-especial {
    -fx-text-fill: red;
    -fx-background-color: black;
}
```

CSS

Por lo general, es recomendable usar estilos en línea solo en situaciones específicas donde se requiere una sobrescritura directa y específica. Para la mayoría de los casos, es mejor definir estilos en hojas de estilo externas. Esto facilita la reutilización y el mantenimiento a largo plazo.

JAVA FX: Proyecto final

Proyecto final Java FX: Gestor de contactos

Objetivo: Desarrollar una aplicación en JavaFX que permita gestionar una agenda de contactos, donde cada contacto tiene un nombre, correo, imagen de perfil, sitio web personal y teléfono. La aplicación deberá ser expansible para añadir más información al contacto en el futuro.

Interfaz de Usuario:

- La pantalla principal debe contener una tabla (consultar en la API el componente TableView) que muestre los nombres, teléfonos y correos de todos los contactos.
- Al seleccionar un contacto en la tabla, se debe mostrar su imagen de perfil en un área designada de la ventana, junto con toda su información, incluida su página web.
- Debe haber un botón que permita al usuario añadir un nuevo contacto. Al hacer clic en este botón, se abrirá una ventana secundaria con un formulario para ingresar la información del nuevo contacto.
- El formulario tendrá un botón que permitirá añadir una imagen al Contacto (consultar en la API el componente FileChooser). La imagen del contacto únicamente podrá ser un fichero de tipo ‘jpg’ o ‘png’. Si no se elige ninguna imagen para el contacto, se deberá establecer una por defecto.
- Al crear un Contacto se deberá crear una copia de la imagen seleccionada y guardarla en una carpeta específica del proyecto donde se deben guardar todas las imágenes de los contactos.

Proyecto final Java FX: Gestor de contactos

Vista y Controlador:

- Utilizar FXML para diseñar las vistas de la aplicación. Debes tener al menos dos archivos FXML: uno para la vista principal y otro para el formulario de añadir contacto.
- Crear clases controladoras adecuadas para manejar la lógica de la vista principal y del formulario.

Modelo de Datos:

- Crear una clase Contacto que represente la información de un contacto individual. La clase debe contener atributos para el nombre, correo, imagen de perfil, sitio web personal y teléfono.

Estilización:

- Utilizar como mínimo un archivo CSS externo para definir y aplicar estilos a los componentes de la interfaz. Se deberá crear una interfaz con un diseño uniforme.

Funcionalidad Adicional (para estudiantes avanzados):

- Implementar la función de editar y eliminar contactos.
- Incorporar persistencia de datos, para guardar y cargar contactos desde una base de datos.
- Diseñar la tabla para que como máximo muestre 10 contactos. En caso de tener una BD con más de 10 contactos de deberá paginar dicha tabla para mostrar los contactos de 10 en 10.
- Añadir un botón para cambiar el diseño de la interfaz, tipo modo Oscuro o similar.

JAVA FX: Ejercicios adicionales

Ejercicios adicionales JavaFX

1. Crea un mini-juego donde debes intentar pulsar un botón que se mueve por toda la ventana, el botón se debe mover primero 10 veces por segundo, luego 20, y así sucesivamente hasta que la última vez se mueva 50 veces por segundo. Al conseguir pulsarlo 5 veces la partida termina indicando el tiempo que se has necesitado para pulsar el botón las 5 veces.
2. Crea el juego del Tres en raya (como en JAVA Swing).
3. Crear el juego de mecanografía (como en JAVA Swing).
4. Crear el juego del Buscaminas.
5. Crear el juego de Hundir la flota.

Bibliografía:

Allen Weiss, M. (2007). *Estructuras de datos en JAVA*. Madrid: Pearson

Froufe Quintas, A. (2002). *JAVA 2: Manual de usuario y tutorial*. Madrid: RA-MA

J. Barnes, D. *Programación orientada a objetos en JAVA*. Madrid: Pearson

Desing Patterns. Elements of Reusable. OO Software

JAVA Limpio. Pello Altadi. Eugenia Pérez

Apuntes de Programación de Anna Sanchis Perales

Apuntes de Programación de Lionel Tarazón Alcocer

Ilustraciones:

https://pixabay.com/

https://freepik.es/

https://lottiefiles.com/

Preguntas