GUI en Java

1. FUNDAMENTOS

1.1. Arquitectura de Swing

Java Swing es una biblioteca de herramientas gráficas (**GUI**, *graphical user interface*) ligera y multiplataforma que forma parte de las **Java Foundation Classes** (**JFC**). Fue desarrollada como una evolución de la librería **AWT**, ofreciendo componentes más sofisticados, personalizables y flexibles para crear interfaces de usuario en aplicaciones de escritorio.

Sus principales características son:

- **Independencia de plataforma**: Al estar escrito completamente en Java, sus componentes son *livianos* y no dependen de elementos nativos del sistema operativo.
- **Arquitectura MVC**: Implementa el patrón **Modelo-Vista-Controlador**, separando la lógica de datos de la presentación visual.
- **Look and Feel/Aspecto y Tacto intercambiable**: Permite cambiar la apariencia visual de la aplicación (temas) sin modificar el código subyacente.
- **Componentes avanzados**: Ofrece elementos como tablas, árboles, paneles con pestañas y listas, además de los básicos como botones y etiquetas.
- **Paquete estándar**: Está incluido en el paquete `javax.swing` y está presente en la SDK estándar de Java desde la versión 1.2

```
JFrame (Ventana principal)
    |--- ContentPane (Panel contenedor)
            |--- JPanel (Paneles)
            |--- JButton (Botones)
            |--- JLabel (Etiquetas)
            |--- JTextField (Campos de texto)
            |--- Otros componentes...
```

1.1.1. Conceptos

- `JFrame`: **La ventana principal**. Todo GUI comienza aquí.
- `JPanel`: **Contenedor que agrupa componentes**. Es como un *lienzo*.
- `ContentPane`: **El área interior** del `JFrame` donde se colocan los componentes.

1.1.2. Ejemplo

```
JFrame ventana = new JFrame("Mi Ventana");
ventana.setSize(400, 300);
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
ventana.setVisible(true);
```

1.2. Layout Managers (Gestores de Diseño)

Son objetos que **organizan** automáticamente los componentes de una interfaz gráfica (GUI) dentro de un contenedor según reglas específicas. Su función principal es controlar el posicionamiento, el tamaño mínimo/máximo y el reacomodo de los elementos cuando se redimensiona la ventana, evitando que la interfaz se descomponga.

Layout	Característica	Cuándo usarlo
`FlowLayout`	Es el predeterminado para JPanel y organiza los componentes en un flujo lineal, de izquierda a derecha. Coloca componentes en línea (como palabras)	Formularios simples, botones
`BorderLayout`	Es el predeterminado para JFrame y divide el contenedor en cinco áreas: NORTH, SOUTH, EAST, WEST, CENTER	Ventanas con áreas definidas
`GridLayout`	Dispone los componentes en una cuadrícula de filas y columnas de igual tamaño	Calculadoras, teclados numéricos
`GridBagLayout`	EL MÁS POTENTE: Permite un control granular colocando componentes en celdas de una cuadrícula, ajustando su tamaño según sus preferencias. Control total de posición/tamaño	Formularios complejos (registro de usuarios)
`BoxLayout`	Alinea los componentes vertical u horizontal	Menús, barras de herramientas
`null layout`	Posicionamiento absoluto (NO RECOMENDADO)	Solo para casos muy específicos

¡No use `null layout`! Utilice siempre use un `LayoutManager`!!

1.3. Eventos y Listeners (El corazón de la interacción)

En **Java Swing**, los **Eventos** *son acciones que realiza el usuario al interactuar con la interfaz gráfica*, como hacer clic en un botón, presionar una tecla o mover el ratón. Estos eventos son notificaciones generadas por componentes específicos (*fuentes de evento*) que informan que ha ocurrido una acción.

Los **Listeners** (o escuchadores) son interfaces en el paquete `java.awt.event` que *se encargan de capturar y procesar estos eventos*. Para responder a una interacción, se debe registrar un *listener* en el componente correspondiente mediante métodos como `addActionListener()`, `addMouseListener()` o `addKeyListener()`. Cuando ocurre el evento, el sistema invoca automáticamente el método específico del listener implementado (por ejemplo, `actionPerformed()` para `ActionListener`).

Los *listeners* más comunes incluyen:

- `ActionListener`: Maneja acciones de alto nivel, como clics en botones (`JButton`), selección de ítems de menú (`JMenuItem`) o pulsar **Enter** en campos de texto (`JTextField`).
- `MouseListener`: **Detecta interacciones con el ratón**, tales como clics (`mouseClicked`), presiones (`mousePressed`), entradas o salidas del cursor del componente (`mouseEntered`/`mouseExited`).
- `KeyListener`: **Captura eventos del teclado**, diferenciando entre teclas pulsadas (`keyPressed`), soltadas (`keyReleased`) o tipadas (`keyTyped`).
- `FocusListener`: Responde cuando un componente **gana** (`focusGained`) o **pierde** (`focusLost`) el foco de selección.

1.4. Modelo de delegación de eventos

Este modelo se basa en el patrón de diseño **Observer**, donde los componentes generan eventos y los **listeners** actúan como observadores que ejecutan la lógica definida por el desarrollador en respuesta a dichas interacciones.

- El usuario **genera** un evento (clic, tecla, etc.).
- El **listener** (*oyente*) captura el evento y ejecuta una acción.

```
// Patrón básico de eventos
boton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // Código que se ejecuta al hacer clic
        System.out.println("¡Botón presionado!");
    }
});
```

1.4.1. Listeners más comunes

Listener	Evento	Método principal
`ActionListener`	Clic en botón, Enter en campo	`actionPerformed()`
`MouseListener`	Clic, entrada/salida del mouse	`mouseClicked()`, `mouseEntered()`
`KeyListener`	Teclas presionadas	`keyPressed()`, `keyTyped()`
`WindowListener`	Cerrar/minimizar ventana	`windowClosing()`
`ItemListener`	Selección en checkbox/combo	`itemStateChanged()`

2. COMPONENTES ESENCIALES

2.1. Contenedores Principales

- `JFrame`: Representa la ventana principal de la aplicación, con título, bordes y botones de control.
- `JPanel`: Es un contenedor liviano utilizado para agrupar y organizar otros componentes dentro de una ventana.
- `JDialog`: Ventana emergente o modal utilizada para mostrar mensajes o solicitar confirmación.

2.2. Componentes básicos

Los componentes básicos de ***Java Swing*** se dividen en **contenedores** (ventanas y paneles) y **componentes interactivos** (botones, etiquetas, campos de texto), todos pertenecientes al paquete `javax.swing` y heredando de `java.awt.Component`.

Componente	Propósito	Ejemplo de uso
`JLabel`	Muestra texto plano, imágenes o resultados de procesos sin permitir edición	`Nombre`: (etiqueta)
`JTextField`	Campo de entrada de texto para una sola línea	Nombre de usuario
`JPasswordField`	Entrada de contraseña (oculta caracteres)	Contraseña
`JTextArea`	Área de edición para múltiples líneas de texto	Descripción, comentarios
`JButton`	Botón de acción estándar que dispara eventos al ser presionado	`Enviar`, `Cancelar`
`JCheckBox`	Casilla de verificación para selecciones independientes.	`Acepto términos`
`JRadioButton`	Botón de opción para selecciones exclusivas (en grupo)	Género (Masculino/Femenino)
`JComboBox`	Cuadro combinado o lista desplegable para seleccionar una opción	Selección de país
`JList`	Lista de selección	Selección de múltiples elementos
`JTable`	Componentes avanzados para mostrar datos estructurados en tablas	Mostrar registros de BD
`JScrollPane`	Barras de desplazamiento	Envolver `JTextArea` o `JTable`

Ejemplo de combo y radio:

```
// Radio buttons (exclusivos)
ButtonGroup grupo = new ButtonGroup();
JRadioButton rb1 = new JRadioButton("Masculino");
JRadioButton rb2 = new JRadioButton("Femenino");
grupo.add(rb1);
grupo.add(rb2);

// Combo box
String[] paises = {"México", "España", "Argentina"};
JComboBox<String> combo = new JComboBox<>(paises);
```

2.3. Diálogos (Mensajes emergentes)

```
// Mensaje informativo
JOptionPane.showMessageDialog(null, "¡Operación exitosa!");

// Confirmación (Sí/No)
int respuesta = JOptionPane.showConfirmDialog(null, "¿Estás seguro?");
if (respuesta == JOptionPane.YES_OPTION) {
    // Eliminar archivo
}

// Entrada de texto
String nombre = JOptionPane.showInputDialog("Ingresa tu nombre:");
```

3. Buenas Prácticas y Patrones

3.1. Patrón MVC (Modelo-Vista-Controlador)

El **patrón MVC** (**Modelo-Vista-Controlador**) en **Java Swing** es una arquitectura de software que estructura la interfaz de usuario separando la lógica en tres componentes interconectados pero independientes para facilitar el mantenimiento y la extensibilidad.

3.1.1. Componentes Principales

- **Modelo**: Representa la *abstracción de los datos y la lógica de negocio*. En *Swing*, existen modelos de datos (como *TableModel* o *ListModel*) que abstraen la información estructurada, y modelos de GUI (como *ButtonModel*) que abstraen el estado de los componentes visuales.
- **Vista**: *Es el conjunto de componentes gráficos* (clases que heredan de `java.awt.Component`) responsables de presentar los datos al usuario. Se encarga de la disposición, el dibujo y la apariencia visual, obteniendo la información necesaria del modelo.
- **Controlador**: *Encapsula la lógica que responde a las interacciones del usuario* (clics, teclas, etc.). En **Java Swing**, esta función suele ser realizada por el ***UI Delegate*** (delegado de interfaz de usuario) asociado a cada componente, el cual captura eventos y actualiza el modelo o la vista según corresponda.

3.2. Implementación en Java Swing

A diferencia de la implementación clásica de MVC, **Java Swing** *adopta un enfoque por componente*. Cada elemento visual tiene *asociado* un delegado de interfaz que actúa como controlador y está vinculado a un modelo de aplicación. Esto permite que los cambios en el modelo del dominio notifiquen automáticamente a las vistas asociadas, manteniendo la sincronización sin necesidad de un controlador centralizado genérico.

```
-------------------------------------------------------------------------
MODELO (Datos)              VISTA (GUI)             CONTROLADOR (Lógica)
-------------------------------------------------------------------------
 Automovil      <- ->    VentanaAutomovil   <- ->   AutomovilController
(Atributos)                  (JFrame)               (Maneja eventos)
-------------------------------------------------------------------------
```

Obligatorio para aplicaciones profesionales!

Ventajas:

- Separa la lógica de negocio de la interfaz.
- Facilita el mantenimiento y pruebas.
- Permite cambiar la vista sin tocar el modelo.

3.3. Threading en Swing (Hilos)

El **threading** en **Java Swing** se refiere al *manejo de hilos de ejecución* para garantizar que la interfaz gráfica de usuario (GUI) permanezca responsiva y nunca se *congele* durante operaciones largas. La arquitectura de **Swing** es esencialmente `single-threaded` para los componentes de la interfaz, lo que significa que la mayoría del código que interactúa con los componentes debe ejecutarse en un hilo especial llamado **Event Dispatch Thread** (**EDT**).

Existen tres tipos principales de hilos en las aplicaciones **Swing**:

- **Hilos iniciales**: Ejecutan el código de arranque inicial y son responsables de crear y mostrar la GUI en el EDT.
- **Event Dispatch Thread** (**EDT**): Es el hilo dedicado donde se ejecuta todo el código de manejo de eventos y la actualización de la interfaz; bloquear este hilo (por ejemplo, con `Thread.sleep`) hace que la GUI deje de responder.
- **Hilos de trabajo** (*Worker Threads*): Se utilizan para ejecutar tareas intensivas o de larga duración en segundo plano, manteniendo la GUI fluida.

Para gestionar estos hilos correctamente, se deben seguir reglas estrictas: **nunca se debe bloquear el EDT con operaciones lentas**, y las actualizaciones de la interfaz desde hilos de trabajo deben realizarse de forma segura. Las herramientas estándar para lograr esto son:

- **SwingWorker**: Una clase abstracta introducida en Java 6 que facilita la ejecución de tareas en segundo plano y la publicación de resultados en el EDT.
- **SwingUtilities.invokeLater()** y **invokeAndWait()**: Métodos para programar la ejecución de código en el EDT de manera segura.
- **javax.swing.Timer**: Útil para realizar operaciones repetitivas o con retardos sin bloquear la interfaz.

Toda actualización de la GUI debe hacerse en el `Event Dispatch Thread (EDT)`!

```
// CORRECTO: Usar SwingUtilities
SwingUtilities.invokeLater(new Runnable() {
    @Override
    public void run() {
        label.setText("Texto actualizado");
    }
});

// INCORRECTO (puede causar bloqueos):
new Thread(() -> {
    label.setText("Texto"); // ¡MAL! No actualices la GUI desde otro hilo
}).start();
```

Conceptos:

- `EDT (Event Dispatch Thread)`: Hilo que maneja todos los eventos de la GUI.
- `SwingUtilities.invokeLater()`: Ejecuta código en el EDT.
- `SwingWorker`: Para tareas largas (descargas, procesamiento) sin congelar la GUI.

3.4. Manejo de Excepciones en GUI

El **Manejo de Excepciones en GUI** con **Java Swing** es el proceso de capturar y gestionar errores de ejecución (como entradas inválidas, fallos de conexión o errores lógicos) para evitar que la interfaz gráfica se detenga abruptamente o se estrelle/***crashee***.

Dado que las aplicaciones **Swing** son interactivas y dependen de eventos del usuario, este manejo es crucial para garantizar la robustez y la usabilidad del software.

3.4.1. Conceptos Clave

- **Estructura Try-Catch**: Se utiliza la instrucción `try-catch` dentro de los *Listeners* (como `ActionListener`) para envolver el código que podría generar errores. Si ocurre una excepción, el bloque `catch` intercepta el error y permite ejecutar una acción alternativa en lugar de terminar el programa.
- **Excepciones vs. Validaciones**: Java no detecta automáticamente todos los errores lógicos (por ejemplo, un número fuera de rango válido pero técnicamente entero). Por ello, se debe combinar el manejo de excepciones con validaciones explícitas del estado de los componentes **Swing** (como `JTextField` o `JComboBox`).
- **Bloque Finally**: Se usa opcionalmente para asegurar que recursos o estados se limpien o actualicen, independientemente de si ocurrió un error o no.

Ejemplo Práctico

Al procesar una entrada de usuario en un campo de texto (`JTextField`), es común encontrar errores de formato o división por cero.

```
botonProcesar.addActionListener(e -> {
    try {
        String input = campoTexto.getText();
```

```
        int valor = Integer.parseInt(input); // Puede lanzar NumberFormatException
        int resultado = 100 / valor;         // Puede lanzar ArithmeticException

        // Actualizar GUI solo si es exitoso
        etiquetaResultado.setText("Resultado: " + resultado);

    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(this, "Ingrese un número válido", "Error",
                                      JOptionPane.ERROR_MESSAGE);
    } catch (ArithmeticException ex) {
        JOptionPane.showMessageDialog(this, "No se puede dividir por cero", "Error",
                                      JOptionPane.ERROR_MESSAGE);
    }
});
```

3.5. Importancia en el Desarrollo

- **Prevención de Bloqueos**: Sin manejo de excepciones, un error no capturado detiene el hilo de la interfaz, congelando la ventana.
- **Gestión de Recursos**: Asegura que conexiones a bases de datos o archivos se cierren correctamente incluso si falla la lógica de negocio.
- **Experiencia de Usuario**: Permite mostrar mensajes de error claros y amigables mediante JOptionPane, en lugar de mostrar trazas de error técnicas en la consola.

Aquí otro ejemplo:

```
btnGuardar.addActionListener(e -> {
    try {
        // Validar y guardar datos
        int edad = Integer.parseInt(txtEdad.getText());
        // ...
    } catch (NumberFormatException ex) {
        // Mostrar error en la GUI, no solo en consola
        JOptionPane.showMessageDialog(null,
            "Error: Ingresa un número válido",
            "Error de formato",
            JOptionPane.ERROR_MESSAGE);
    }
});
```

4. CONCEPTOS AVANZADOS

4.1. Componentes personalizados

- Extender `JPanel` para crear componentes reutilizables.
- Sobrescribir `paintComponent()` para dibujos personalizados.

```
class PanelDibujo extends JPanel {
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.RED);
        g.fillOval(10, 10, 100, 100); // Dibujar círculo
    }
}
```

4.2. Modelos de datos (TableModel)

Para `JTable`, **NUNCA** uses datos directamente:

```
// MAL: Directo (no se actualiza automáticamente)
String[][] datos = {{"Juan", "25"}};
JTable tabla = new JTable(datos, columnas);



// BIEN: Usar DefaultTableModel
DefaultTableModel modelo = new DefaultTableModel();
modelo.addColumn("Nombre");
modelo.addColumn("Edad");
modelo.addRow(new Object[]{"Juan", 25});
JTable tabla = new JTable(modelo);
```

4.3. JavaFX (El futuro de GUIs en Java)

Aunque **Swing** sigue vigente, **JavaFX** es la tecnología moderna.

- **JavaFX** es el *framework* moderno y activo para el desarrollo de interfaces gráficas de usuario (GUI) en Java, diseñado para reemplazar a **Java Swing**, que es una tecnología heredada ya congelada en cuanto a nuevas características.
- **Arquitectura**: Utiliza un **Scene Graph** (grafo de escenas) jerárquico, lo que facilita la manipulación de elementos visuales y ofrece mejor rendimiento gráfico.
- **Diseño y Estilizado**: Soporta nativamente CSS para la personalización visual y utiliza FXML (XML) para separar la interfaz de usuario de la lógica de negocio, facilitando el patrón MVC.
- **Multimedia y Gráficos**: Incluye soporte integrado para animaciones fluidas, gráficos 3D, aceleración por hardware y reproducción de multimedia, capacidades que son limitadas o inexistentes en Swing.
- **Estado actual**: A partir de Java 11, **JavaFX** se separó del JDK principal como **OpenJFX**, siendo la opción recomendada para cualquier proyecto nuevo, mientras que **Swing** se mantiene solo para compatibilidad con **sistemas legacy**[1].

[1] Un **sistema legacy** (o *sistema heredado*) es una aplicación, software o infraestructura tecnológica **obsoleta** que continúa utilizándose dentro de de una organización porque sigue desempeñando funciones críticas para la operación diaria, a pesar de no contar con soporte, mantenimiento ni actualizaciones de los fabricantes

4.4. Comparación

Aspecto	Swing	JavaFX
Interfaz	Programática (código Java)	FXML (XML) + CSS
Estilo	Limitado	CSS potente
Animaciones	Complejas	Nativas
Curva de aprendizaje	Media	Media-Alta
Mantenimiento	Oracle no lo actualiza activamente	Activo

4.5. Ejemplo mínimo en JavaFX

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HolaJavaFX extends Application {
    @Override
    public void start(Stage stage) {
        Button btn = new Button("¡Haz clic!");
        StackPane root = new StackPane(btn);
        Scene scene = new Scene(root, 300, 200);
        stage.setScene(scene);
        stage.show();
    }
}
```

5. Lista de Verificación (para Estudiantes)

1. **Crear** una ventana (`JFrame`) con componentes.
2. **Organizar** componentes con `LayoutManager`.
3. **Manejar** eventos (clic, teclado, mouse).
4. **Usar** diálogos (`JOptionPane`).
5. **Aplicar** el patrón MVC.
6. **Conectar** GUI con base de datos (JDBC).
7. **Actualizar** la GUI desde otros hilos correctamente.
8. **Validar** datos de entrada.
9. **Mostrar** datos en `JTable`
10. **Estructurar** el código en paquetes lógicos.

6. ERRORES COMUNES QUE DEBE EVITAR

Error	Solución
Usar `null layout`	Siempre usar un `LayoutManager`
Poner lógica en la Vista	Usar MVC, separar en Controlador
No usar `SwingUtilities.invokeLater()`	Usarlo para actualizaciones GUI
No manejar excepciones	Usar `try-catch` y mostrar mensajes
Hacer tareas pesadas en EDT	Usar `SwingWorker`
No cerrar conexiones a BD	Usar `try-with-resources`

7. RECURSOS RECOMENDADOS

1. **Documentación oficial**: [Oracle Swing Tutorial]
2. **Libro**: **Java: Cómo Programar** (Deitel & Deitel) - Capítulo de GUI.