

Comunicándonos con el usuario. Interfaces.

Caso práctico



Ministerio de Educación y FP ([CC BY-NC](#))

Ana está cursando el módulo de Programación.

En el aula suele sentarse junto a su compañero **José Javier**.

En clase llevan unos días explicándoles cómo construir aplicaciones Java, utilizando interfaces gráficas de usuario (GUI).

Pero, ¿qué es una interfaz de usuario? A grandes rasgos, les han comentado que una interfaz de usuario es el medio con que el usuario puede comunicarse con una computadora. Las interfaces gráficas de usuario incluyen elementos tales como: menús, ventanas,

paneles, etc., en general, elementos gráficos que facilitan la comunicación entre el ser humano y la computadora.

Hasta ahora, las aplicaciones de ejemplo que habían realizado, estaban en modo consola, o modo carácter, y están contentos porque están viendo las posibilidades que se les abren ahora. Están comprobando que podrán dar a sus aplicaciones un aspecto mucho más agradable para el usuario.

Ana le comenta a **José Javier**:

—Así podremos dar un aspecto profesional a nuestros programas.



[Ministerio de Educación y Formación Profesional](#). (Dominio público)

Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.

[Aviso Legal](#)

1.- Introducción.

Hoy en día las **interfaces** de los programas son cada vez más sofisticadas y atractivas para el usuario. Son intuitivas y cada vez más fáciles de usar: pantallas táctiles, etc.

Sin embargo, no siempre ha sido así. No hace muchos años, antes de que surgieran y se popularizaran las interfaces gráficas de usuario para que el usuario interactuara con el sistema operativo con sistemas como Windows, etc., se trabajaba en **modo consola**, o **modo carácter**, es decir, se le daban las ordenes al ordenador con comandos por teclado, de hecho, por entonces no existía el ratón.



[jdiasica](#) ([CC BY-NC](#))

Así que, con el tiempo, con la idea de simplificar el uso de los ordenadores para extender el uso a un cada vez mayor espectro de gente, de usuarios de todo tipo, y no sólo para los expertos, se ha convertido en una práctica habitual utilizar **interfaces gráficas de usuario (IGU ó GUI** en inglés), para que el usuario interactúe y establezca un contacto más fácil e intuitivo con el ordenador.

En ocasiones verás otras definiciones de interfaz, como la que define una interfaz como un dispositivo que permite comunicar dos sistemas que no hablan el mismo lenguaje. También se emplea el término interfaz para definir juego de conexiones y dispositivos que hacen posible la comunicación entre dos sistemas.

Aquí en este módulo, cuando hablamos de interfaz nos referimos a la cara visible de los programas tal y como se presenta a los usuarios para que interactúen con la máquina. La interfaz gráfica implica la presencia de un monitor de ordenador, en el que veremos la interfaz constituida por una serie de **menús e iconos que representan las opciones que el usuario** puede tomar dentro del sistema.

Las interfaces gráficas de usuario proporcionan al usuario ventanas, cuadros de diálogo, barras de herramientas, botones, listas desplegables y muchos otros elementos. Las aplicaciones son conducidas por eventos y se desarrollan haciendo uso de las clases que para ello nos ofrece el API de Java.

En 1981 aparecieron los primeros ordenadores personales, los llamados PC, pero hasta 1993 no se generalizaron las interfaces gráficas de usuario. El escritorio del sistema operativo Windows de Microsoft y su sistema de ventanas sobre la pantalla se ha estandarizado y universalizado, pero fueron los ordenadores Macintosh de la compañía Apple los pioneros en introducir las interfaces gráficas de usuario.

Para saber más

En el siguiente enlace puedes ver la evolución en las interfaces gráficas de diverso software, entre ellos, el de las GUI de MAC OS, o de Windows en las sucesivas versiones

[Galería de interfaces gráficas](#)

Autoevaluación

Señala la opción correcta acerca de las interfaces gráficas de usuario:

- Son sinónimos de ficheros de texto.
- Las ventanas de una aplicación no serían un ejemplo de elemento de interfaz gráfica de usuario.
- Surgen con la idea de facilitar la interacción del usuario con la máquina.
- Ninguna es correcta.

¡No! Esos dos conceptos no tienen nada ver.

¡Incorrecto! Las ventanas sí son un ejemplo de interfaz gráfica.

¡Exacto! Se pretendía con ellas que cada vez más gente usara las computadoras.

¡No es correcto! Hay una correcta.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

2.- Librerías de Java para desarrollar GUI.

Caso práctico



José Javier Bermúdez Hernández
[\(CC BY-NC\)](#)

José Javier está un poco abrumado por la cantidad de componentes que tiene Java para desarrollar interfaces gráficos. Piensa que hay tantos, que son tantas clases, que nunca podrá aprendérselos. **Ana** le dice:

—José Javier, no te preocupes, seguro que haciendo programas, enseguida ubicarás los que más utilices, y el resto, siempre tienes la documentación de Java para cuando dudes, consultarla.

Hemos visto que la interfaz gráfica de usuario es la parte del programa que permite al usuario interaccionar con él. Pero, ¿cómo la podemos crear en Java?

El API de Java proporciona una librería de clases para el desarrollo de interfaces gráficas de usuario (en realidad son dos: AWT y Swing). Esas librerías se engloban bajo los nombres de **AWT** y **Swing**, que a su vez forman parte de las **Java Foundation Classes o JFC**. AWT es una librería ya en desuso y en la actualidad Swing está siendo desplazado por **JavaFX**, plataforma que permite el desarrollo de interfaces gráficas de usuario con grandes prestaciones y ejecutables en cualquier plataforma (pc, móviles, web, etc). JavaFX es una tecnología de Oracle que no está incluida en JDK pero está a disposición de los usuarios que deseen utilizarla. Sin duda JavaFX marcará el futuro de las interfaces gráficas en Java, pues además es compatible con componentes Swing.

Entre las clases de las **JFC** hay un grupo de elementos importante **que ayuda a la construcción de interfaces gráficas de usuario (GUI)** en Java.

Los elementos que componen las **JFC** son:

- ✓ **Componentes Swing**: encontramos componentes tales como botones, cuadros de texto, ventanas o elementos de menú.
- ✓ **Soporte de diferentes aspectos y comportamientos (Look and Feel)**: permite la elección de diferentes apariencias de entorno. Por ejemplo, el mismo programa puede adquirir un aspecto **Metal** Java (multiplataforma, igual en cualquier entorno), **Motif** (el aspecto estándar para entornos Unix) o **Windows** (para entornos Windows). De esta forma, el aspecto de la aplicación puede ser independiente de la plataforma, lo cual está muy bien para un lenguaje que lleva la seña de identidad de multiplataforma, y se puede elegir el que se deseé en cada momento.
- ✓ **Interfaz de programación Java 2D**: permite incorporar gráficos en dos dimensiones, texto e imágenes de alta calidad.
- ✓ **Soporte de arrastrar y soltar** (Drag and Drop) entre aplicaciones Java y aplicaciones nativas. Es decir, se implementa un portapapeles. Llamamos aplicaciones nativas a las que están desarrolladas en un entorno y una plataforma concretos (por ejemplo Windows o Unix), y sólo funcionarán en esa plataforma.
- ✓ **Soporte de impresión**.
- ✓ **Soporte sonido**: captura, reproducción y procesamiento de datos de audio y **MIDI**
- ✓ **Soporte de dispositivos de entrada distintos del teclado**, para japonés, chino, etc.
- ✓ **Soporte de funciones de Accesibilidad, para crear interfaces para discapacitados**: permite el uso de tecnologías como los lectores de pantallas o las pantallas Braille adaptadas a las personas discapacitadas.

Con estas librerías, Java proporciona un conjunto de herramientas para la construcción de interfaces gráficas que tienen una apariencia y se comportan de forma semejante en todas las plataformas en las que se ejecuten.

El objetivo de esta unidad es hacer una introducción a la construcción de interfaces gráficas con Java utilizando las dos plataformas más utilizadas: Swing y JavaFX. Se trabajarán los conceptos básicos que permitirán una primera toma de contacto con el mundo de las interfaces gráficas de usuario. Todos estos contenidos serán trabajados en profundidad en el Módulo Profesional "Interfaces de Usuario" de segundo curso.

Autoevaluación

Señala la opción incorrecta. JFC se consta de los siguientes elementos:

- Componentes Swing.
- Soporte de diversos "look and feel".
- Soporte de impresión.
- Interfaz de programación Java 3D.

¡Incorrecto! Claro que sí consta de controles Swing.

¡No es correcto! Sí que proporciona soporte de diferentes aspectos y comportamientos.

¡No es la opción correcta! Sí que proporciona soporte para impresión.

¡Exacto! Proporciona soporte para 2D pero no para 3D.

Solución

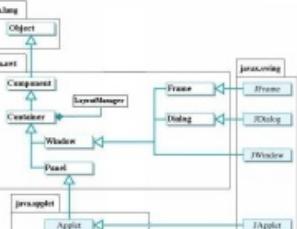
1. Incorrecto
2. Incorrecto
3. Incorrecto
4. Opción correcta

2.1.- Swing.

Cuando se vio que era necesario mejorar las características que ofrecía AWT, distintas empresas empezaron a sacar sus propios controles para mejorar algunas de las características de AWT. Así, Netscape sacó una librería de clases llamada **Internet Foundation Classes** para usar con Java, y eso obligó a Sun (todavía no adquirida por Oracle) a reaccionar para adaptar el lenguaje a las nuevas necesidades.

Se desarrolló en colaboración con Netscape todo el conjunto de componentes Swing que se añadieron a la JFC.

Swing es una librería de Java para la generación del GUI en aplicaciones.



José Javier Bermúdez Hernández ([CC BY-NC](#))

Swing se apoya sobre AWT y añade JComponents. La arquitectura de los componentes de Swing facilita la personalización de apariencia y comportamiento, si lo comparamos con los componentes AWT.

Por cada componente AWT (excepto canvas) existe un componente Swing equivalente, cuyo nombre empieza por J, que permite más funcionalidad siendo menos pesado. Así, por ejemplo, para el componente AWT Button existe el equivalente Swing JButton, que permite como funcionalidad adicional la de crear botones con distintas formas (rectangulares, circulares, etc), incluir imágenes en el botón, tener distintas representaciones para un mismo botón según esté seleccionado, o bajo el cursor, etc.

La razón por la que no existe **JCanvas** es que los paneles de la clase **JPanel** ya soportan todo lo que el componente **Canvas** de AWT soportaba. No se consideró necesario añadir un componente Swing **JCanvas** por separado.

Algunas características más de Swing, podemos mencionar:

- ✓ Es independiente de la arquitectura (metodología no nativa propia de Java)
- ✓ Proporciona todo lo necesario para la creación de entornos gráficos, tales como diseño de menús, botones, cuadros de texto, manipulación de eventos, etc.
- ✓ **Los componentes Swing no necesitan una ventana propia del sistema operativo cada uno**, sino que son visualizados dentro de la ventana que los contiene mediante métodos gráficos, por lo que requieren bastantes menos recursos.
- ✓ **Las clases Swing están completamente escritas en Java, con lo que la portabilidad es total**, a la vez que no hay obligación de restringir la funcionalidad a los mínimos comunes de todas las plataformas
- ✓ Por ello las clase Swing aportan una considerable gama de funciones que haciendo uso de la funcionalidad básica propia de AWT aumentan las posibilidades de diseño de interfaces gráficas.
- ✓ Debido a sus características, los componentes AWT se llaman componentes “**de peso pesado**” por la gran cantidad de recursos del sistema que usan, y los componentes **Swing** se llaman componentes “**de peso ligero**” por no necesitar su propia ventana del sistema operativo y por tanto consumir muchos menos recursos.
- ✓ Aunque todos los componentes Swing derivan de componentes AWT y de hecho se pueden mezclar en una misma aplicación componentes de ambos tipos, se desaconseja hacerlo. Es preferible desarrollar aplicaciones enteramente Swing, que requieren menos recursos y son más portables.

2.2.- JavaFX

JavaFX es una tecnología Java para el diseño de aplicaciones con interfaces gráficas interactivas multiplataforma. Las aplicaciones JavaFx pueden ser ejecutadas con el mismo resultado en multitud de dispositivos: pc, móviles, tv, consolas, etc. JavaFX amplía la potencia de Java permitiendo a los desarrolladores utilizar cualquier biblioteca de Java en aplicaciones JavaFX. Los desarrolladores pueden ampliar sus capacidades en Java y utilizar la tecnología de presentación que JavaFX proporciona para crear experiencias visuales que resulten atractivas.

El JavaFX no está incluido en el SDK estándar desde la version 11. Sin embargo, proporciona dispone de su propia kit de desarrollo, en concreto, JavaFX SDK 14 en la última versión. Está formada por un conjunto de clases y API junto con un editor gráfico **Scene Builder** para crear las interfaces visualmente.

- **Java APIs**: las APIs están escritas en código nativo Java compatibles con otros lenguajes soportados por la máquina virtual.
- **FXML and Scene Builder**: FXML es un lenguaje de marcado que describe las interfaces de usuario. Se pueden escribir directamente o usar la herramienta JavaFX Scene Builder para crearlos con una interfaz gráfica.
- **WebView**: permite embeber páginas HTML en las aplicaciones JavaFX. Ofrece soporte para JavaScript.
- **Built-in UI controls and CSS**: proporciona cantidad de controles para construir aplicaciones completas. El estilo de los controles puede ser modificado con CSS.
- **Canvas API**: para dibujar directamente en la pantalla.
- **Multitouch Support**: soporte para gestos táctiles múltiples en función de las posibilidades de la plataforma subyacente.
- **Hardware-accelerated graphics pipeline**: haciendo uso de la GPU se consiguen animaciones gráficas fluidas en las tarjetas gráficas soportadas, si la gráfica no está soportada de hace uso de la pila de software Java2D.
- **High-performance media engine**: soporta la reproducción de contenido multimedia con baja latencia basándose en GStreamer.
- **Self-contained application deployment model**: las aplicaciones contenidas tiene todos los recursos y una copia privada de los entornos de ejecución de Java y JavaFX. Son distribuidos como paquetes instalables y proporcionan la misma experiencia de instalación e inicio que las aplicaciones nativas del sistema operativo.

3.- Diseño de Interfaces con Swing.

Trabajaremos en este capítulo con Swing, la librería de JFC para la construcción de interfaces gráficas de usuario. Para escribir código Swing tan solo necesitamos un editor de texto, como para escribir otro código Java. Sin embargo, los IDEs actuales proporcionan herramientas que permiten la construcción y diseño de interfaces de manera mas intuitiva, a través de entornos gráficos. Netbeans incluye esta herramienta (**editor gráfico**) de forma nativa, sin embargo, Eclipse las proporciona a través de plugins.

Debes conocer

Aunque utilicemos herramientas de diseño de interfaces Swing, independientemente del entorno de desarrollo utilizado, debemos entender mínimamente el código Java generado. En ciertas ocasiones puede ser útil hacer pequeñas modificaciones a través de código.

Debemos saber también que el código escrito a mano por un programador SIEMPRE será más eficiente que el generado por cualquier herramienta de diseño que genere código automáticamente.

Recomendación

La mejor forma de aprender a diseñar, construir y añadir funcionalidad a interfaces Java es diseñándolas. Por eso, te animamos a que pruebes cada uno de los ejemplos que se utilizan además de tratar de implementar funcionalidades que se te ocurran.

3.1- Creación de interfaces gráficos de usuario utilizando asistentes y herramientas del entorno integrado.

Caso práctico

Ana le ha tomado el gusto a hacer programas con la librería Swing de Java y utilizando el IDE NetBeans. Le parece tan fácil que no lo puede creer. Pensaba que sería difícil el uso de los controles de las librerías, pero ha descubierto que es poco menos que cogerlos de la paleta y arrastrarlos hasta el formulario donde quiere situarlos.

—Pero si esto lo podría hacer hasta mi sobrino pequeño,... — piensa para sí.

Crear aplicaciones que incluyan interfaces gráficos de usuario, con NetBeans, es muy sencillo debido a las facilidades que nos proporcionan sus asistentes y herramientas.

El IDE de programación nos proporciona un **diseñador o editor gráfico** para crear aplicaciones de una manera fácil, sin tener que preocuparnos demasiado del código. Simplemente nos debemos centrar en el funcionamiento de las mismas.

Un programador experimentado, probablemente no utilizará los asistentes, sino que escribirá, casi siempre, todo el código de la aplicación. De este modo, podrá indicar por código la ubicación de los diferentes controles, de su interfaz gráfica, en el contenedor (panel, marco, etc.) en el que se ubiquen.

Pero en el caso de programadores novatos, o simplemente si queremos ahorrar tiempo y esfuerzo, tenemos la opción de aprovecharnos de las capacidades que nos brinda un entorno de desarrollo visual para diseñar e implementar formularios gráficos.

Algunas de las características de NetBeans, que ayudan a la generación rápida de aplicaciones, y en particular de interfaces son:

- ✓ **Modo de diseño libre (Free Design):** permite mover libremente los componentes de interfaz de usuario sobre el panel o marco, sin atenerse a uno de los layouts por defecto. Un layout define la posición de los elementos de un panel.
- ✓ **Independencia de plataforma:** diseñando de la manera más fácil, es decir, arrastrando y soltando componentes desde la paleta al área de diseño visual, el NetBeans sugiere alineación, posición, y dimensión de los componentes, de manera que se ajusten para cualquier plataforma, y en tiempo de ejecución el resultado sea el óptimo, sin importar el sistema operativo donde se ejecute la aplicación.
- ✓ **Soporte de internacionalización:** Se pueden internacionalizar las aplicaciones Swing, aportando las traducciones de cadenas de caracteres, imágenes, etc., sin necesidad de tener que reconstruir el proyecto, sin tener que compilarlo según el país al que vaya dirigido. Se puede utilizar esta características empleando ficheros de recursos (ResourceBundle files). En ellos, deberíamos aportar todos los textos visibles, como el texto de etiquetas, campos de texto, botones, etc. NetBeans proporciona una asistente de internacionalización desde el menú de herramientas (Tools).

Autoevaluación

NetBeans ayuda al programador de modo que pueda concentrarse en implementar la lógica de negocio que se tenga que ejecutar por un evento dado.

- Verdadero Falso

Verdadero

En efecto, NetBeans ayuda de esa manera, siendo de gran ayuda en la gestión de eventos de las aplicaciones.

3.1.1.- Diseñando con asistentes.

Los pasos para crear y ejecutar aplicaciones, se pueden resumir en:

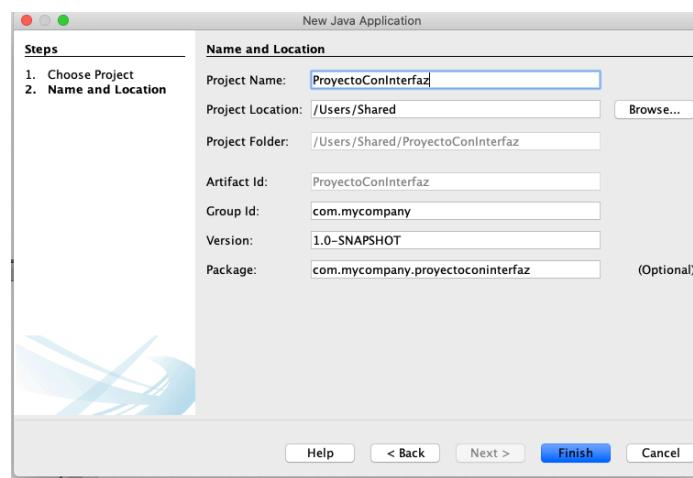
- ✓ Crear un proyecto.
- ✓ Construir la interfaz con el usuario.
- ✓ Añadir funcionalidad, en base a la lógica de negocio que se requiera.
- ✓ Ejecutar el programa.

Veamos un ejemplo con estos pasos:

- ✓ Primero, crea el proyecto de la manera tan simple como puedes ver en:

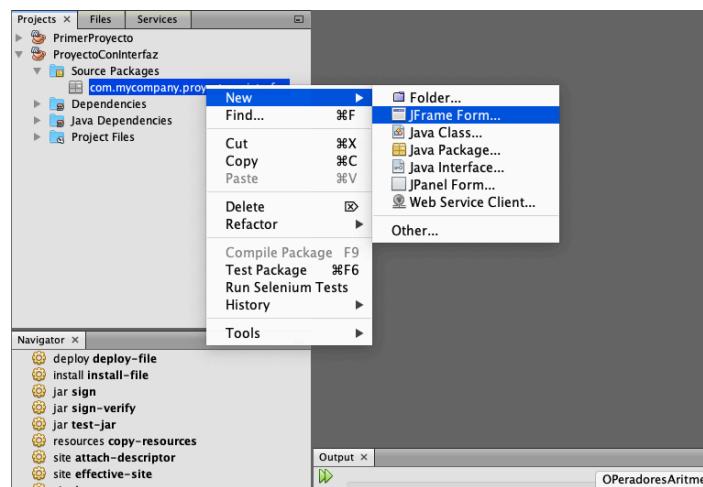
Creando proyecto con interfaz Swing

En primer lugar crea un nuevo proyecto Java tal y como lo hemos venido haciendo hasta ahora.



Creando proyecto con interfaz Swing II

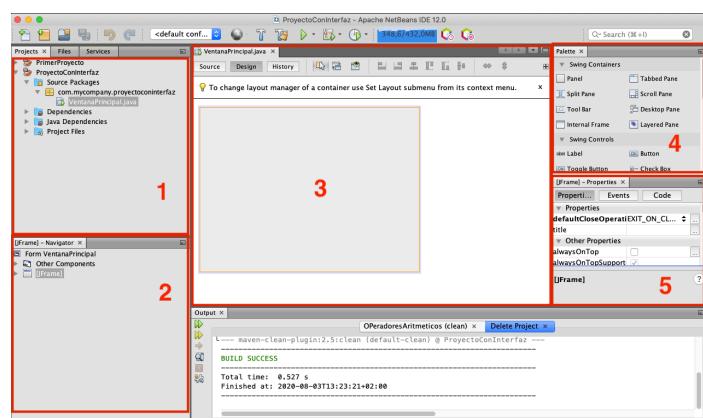
En segundo lugar crearemos una clase que represente la ventana principal de nuestra aplicación. Para ello, dentro del paquete creado por defecto al crear el proyecto, creamos una clase **JFrame** (conocerás mucho más de ella en capítulos posteriores). Tan solo tendrás que asignarle un nombre, como a cualquier otra clase.



Creando proyecto con interfaz Swing III

Después de crear nuestra primera ventana, observa cómo se abre automáticamente el diseñador de Netbeans. En la ventana del entorno tenemos:

1. Clase Java en el **Editor de Proyectos**, en el ejemplo, Ventanaprinicipal.java.
2. Justo debajo del editor de proyecto, tenemos el panel **Navigator**, que nos permitirá navegar por el conjunto de elementos que vayamos incluyendo en nuestra ventana.
3. **Editor gráfico** en la parte central: permite la edición gráfica de los elementos (botón **Design**) o a través de código Java (botón **Source**), situados ambos en la parte superior de diseñador.
4. **Paleta de elementos**: Importante, porque contiene cada uno de los elementos de diseño que podemos añadir a nuestra ventana arrastrando con el ratón (etiquetas, botones, listas, menús, etc).
5. Panel de propiedades (**Properties**): muestra y permite la modificación de las propiedades de un elemento seleccionado en el editor. Además, permite gestionar los eventos asociados a dicho elemento.



¡Si ejecutas la aplicación aparecerá la ventana de nuestra ventana, que no tendrá contenido!

Creando proyecto con interfaz Swing IV

Todas las imágenes utilizadas pertenecen al Ministerio de Educación y FP con licencia CC BY-NC y son capturas de pantalla de la aplicación Netbeans, propiedad de Oracle.

[Resumen textual alternativo](#)

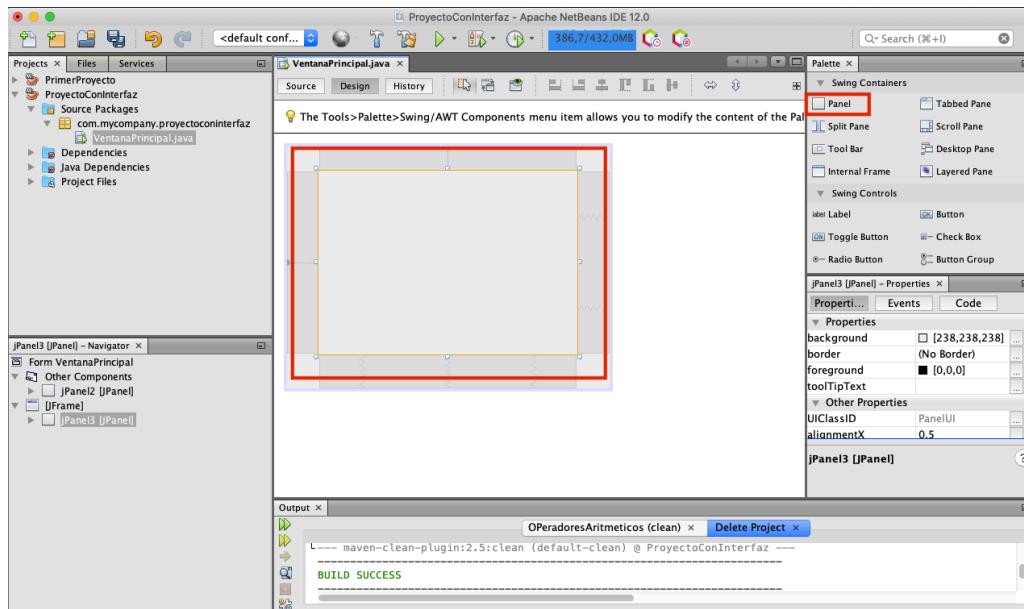
- ✓ A continuación, vamos a crear la interfaz. Se tratará de una ventana con tres campos de texto y varios botones. Obsévalo en la siguiente presentación:

Añadiendo componentes a la interfaz

Sobre la aplicación creada en el punto anterior, vamos a añadir elementos a nuestra ventana para crear una minicalculadora.

En primer lugar, añade un **JPanel** a la ventana. Para ello solo tendrás que pulsar sobre el elemento **Panel** en la paleta y a continuación colocarlo en la ventana. Un panel es un elemento contenedor donde colocamos otros elementos.

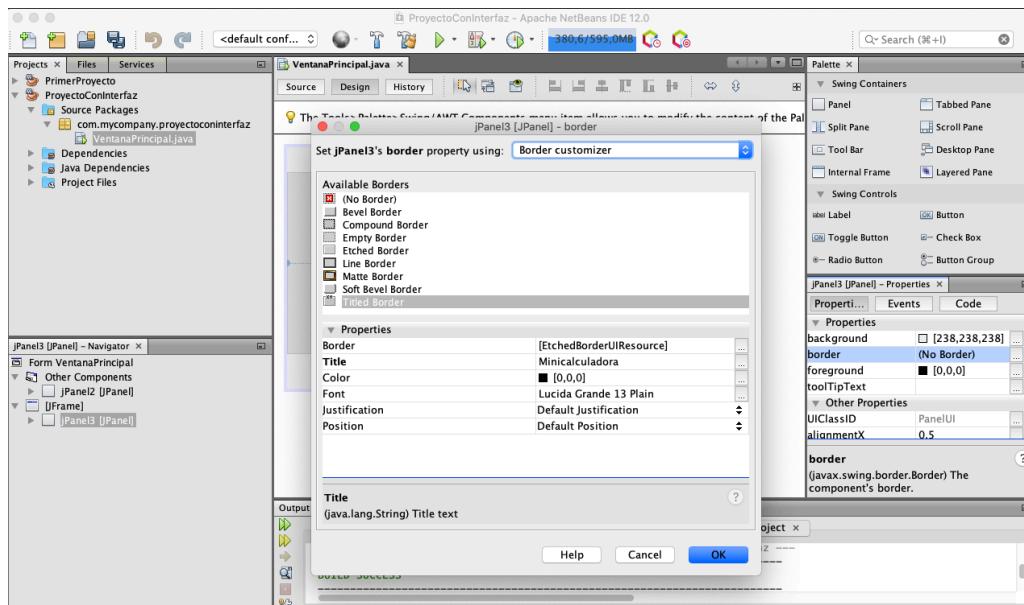
- Una vez colocado, puedes ajustar el tamaño y reajustarlo si es necesario. Incluso puedes ajustar el tamaño de la ventana (**JFrame**).



Añadiendo componentes a la interfaz II

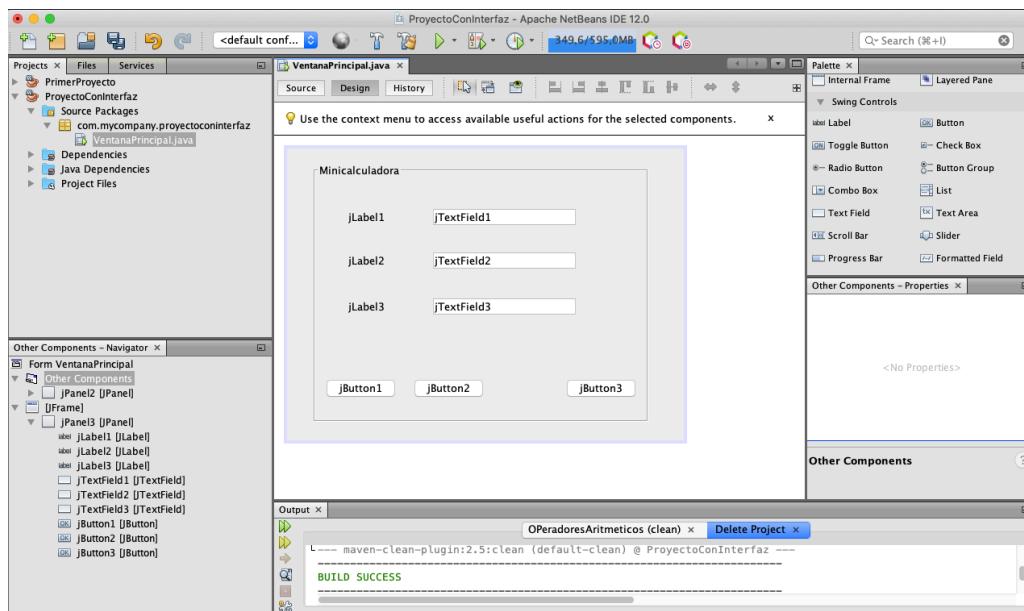
La siguiente tarea será añadir un borde y un título al panel. En este caso accedemos a esas propiedades a través del panel de propiedades.

- Selecciona **Title Border** de los tipos de bordes disponibles.
- Añade un texto a la propiedad **Title**.
- Observa el efecto en el panel tras aceptar.



Añadiendo componentes a la interfaz III

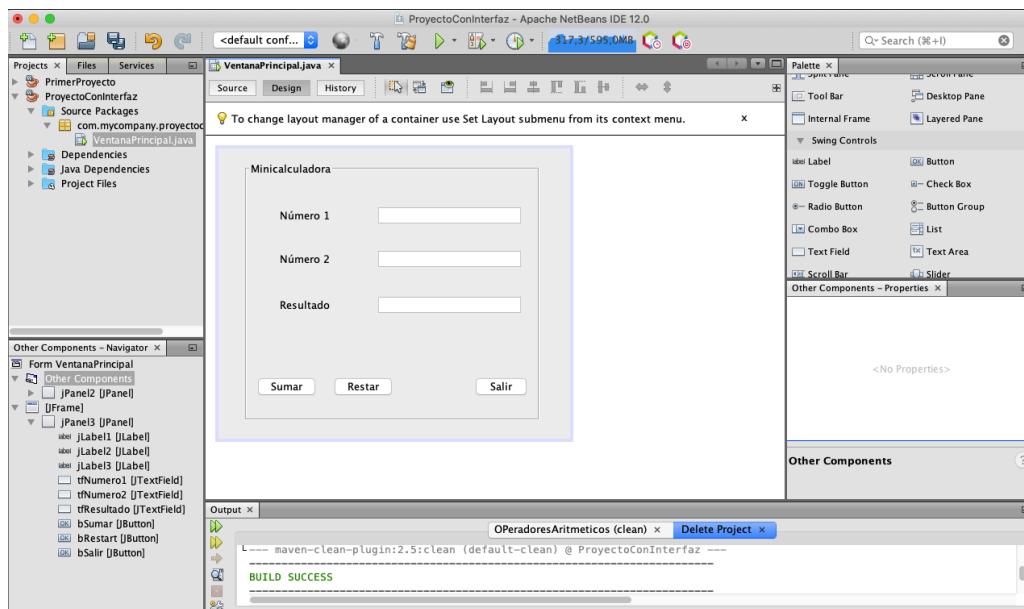
Ya tenemos nuestro panel. Has comprobado que añadir elementos es muy fácil desde el editor gráfico de Netbeans. Añade componentes **JTextField** (campos de texto), **Labels** (Etiquetas) y **Button** (botones) para conseguir un diseño parecido al que se ve en la imagen. Podrás comprobar que alinearlos es relativamente fácil con la ayuda del diseñador.



Añadiendo componentes a la interfaz IV

Vamos a personalizar las etiquetas y el texto mostrado en los campos de texto. Debemos tener en cuenta cada elemento que añadimos a nuestro diseño estará representado por un objeto Java que tendrá un identificador (puedes observarlo en el código fuente). **Nunca debemos confundir el texto que muestra un elemento en la ventana con su identificador asociado en el código fuente**. Para acceder a estas propiedades utilizamos el navegador situado en la parte inferior izquierda de la pantalla. Si pulsas con el botón derecho sobre cualquier elemento, podrás:

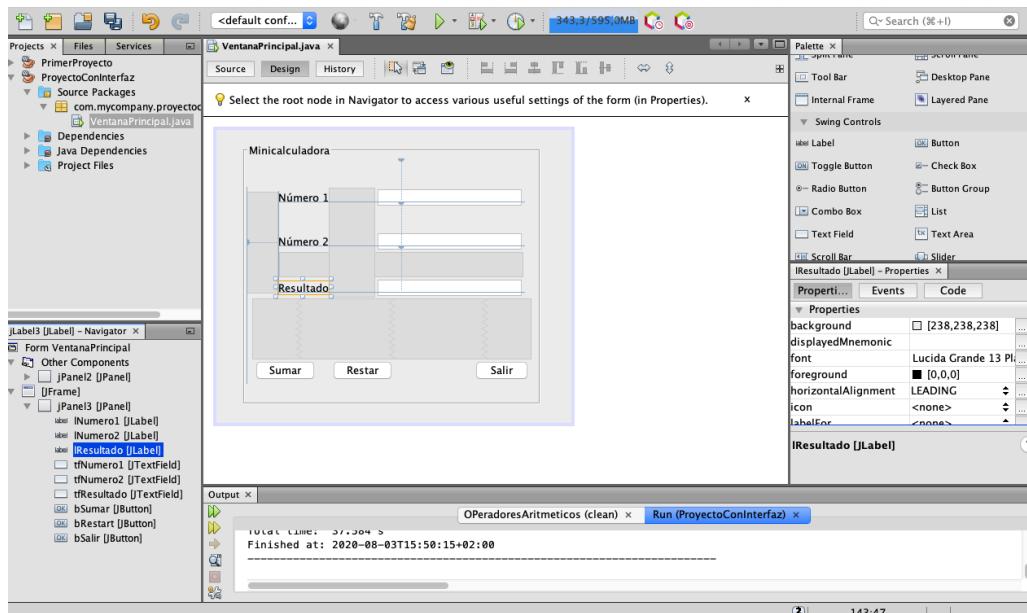
- **Edit Text**: Cambiar el texto que muestra el componente en la pantalla.
- **Change Variable Name**: Cambiar el identificador del objeto en código Java.



Añadiendo componentes a la interfaz V

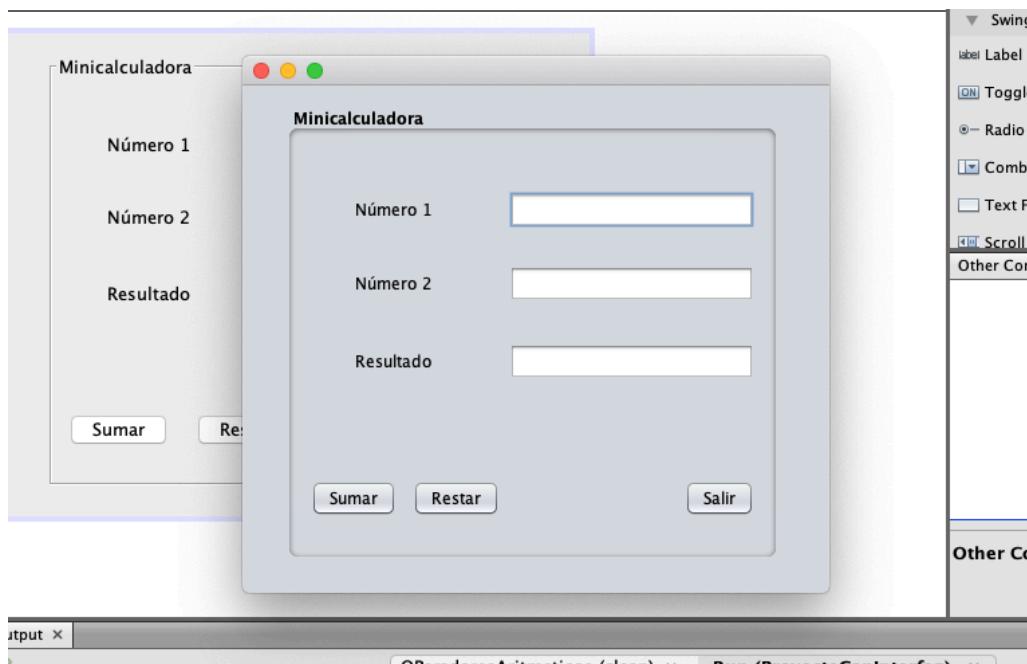
Edita el texto mostrado por cada elemento y su identificador para obtener algo parecido a lo que se observa en la imagen. Observa como se han cambiado los identificadores de los elementos. Reajusta la ventana y el panel si lo consideras oportuno.

- Los campos de texto no deben mostrar nada, por lo tanto, solo tienes que eliminar el texto.



Añadiendo componentes a la interfaz VI

Ya tenemos nuestra interfaz, aunque no implementa ninguna funcionalidad. Ejecuta la aplicación y verás algo parecido a lo que se observa en pantalla. Si tu color del fondo es diferente, trata de modificar la propiedad apropiada para cambiarlo



Añadiendo componentes a la interfaz VII

¿Qué ocurre cuando añadimos elementos a nuestra a través del editor gráfico? Netbeans está generando código java que automáticamente va añadiendo al código fuente.

```

25  /*
26  * @suppressWarnings("unchecked")
27  * // <editor-fold defaultstate="collapsed" desc="Generated Code">
28  *
29  * private void initComponents() {
30  *
31  *     jPanel2 = new javax.swing.JPanel();
32  *     jPanel3 = new javax.swing.JPanel();
33  *     jLabel1 = new javax.swing.JLabel();
34  *     jLabel2 = new javax.swing.JLabel();
35  *     jLabel3 = new javax.swing.JLabel();
36  *     tfNumero1 = new javax.swing.JTextField();
37  *     tfNumero2 = new javax.swing.JTextField();
38  *     tfResultado = new javax.swing.JTextField();
39  *     bSumar = new javax.swing.JButton();
39  *     bRestart = new javax.swing.JButton();
39  *     bSalir = new javax.swing.JButton();
39  *
40  *     javax.swing.GroupLayout jPanel2Layout = new javax.swing.GroupLayout(jPanel2);
41  *     jPanel2.setLayout(jPanel2Layout);
42  *     jPanel2Layout.setHorizontalGroup(
43  *         jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
44  *         .add(jPanel2Layout.createSequentialGroup()
45  *             .addContainerGap()
46  *             .addGroup(jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
47  *                 .addGroup(jPanel2Layout.createSequentialGroup()
48  *                     .addComponent(jLabel1)
49  *                     .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
50  *                     .addComponent(tfNumero1, javax.swing.GroupLayout.PREFERRED_SIZE, 100, javax.swing.GroupLayout.PREFERRED_SIZE))
51  *                 .addGroup(jPanel2Layout.createSequentialGroup()
52  *                     .addComponent(jLabel2)
53  *                     .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
54  *                     .addComponent(tfNumero2, javax.swing.GroupLayout.PREFERRED_SIZE, 100, javax.swing.GroupLayout.PREFERRED_SIZE))
55  *                 .addGroup(jPanel2Layout.createSequentialGroup()
56  *                     .addComponent(jLabel3)
57  *                     .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
58  *                     .addComponent(tfResultado, javax.swing.GroupLayout.PREFERRED_SIZE, 100, javax.swing.GroupLayout.PREFERRED_SIZE))
59  *                 .addGroup(jPanel2Layout.createSequentialGroup()
60  *                     .addComponent(bSumar)
61  *                     .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
62  *                     .addComponent(bRestart)
63  *                     .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
64  *                     .addComponent(bSalir))
65  *             .addContainerGap())
66  *         .addGap(0, 0, Short.MAX_VALUE)
67  *     );
68  *     jPanel2Layout.setVerticalGroup(
69  *         jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
70  *         .addGroup(jPanel2Layout.createSequentialGroup()
71  *             .addContainerGap()
72  *             .addGroup(jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
73  *                 .addComponent(jLabel1)
74  *                 .addComponent(tfNumero1, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
75  *             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
76  *             .addGroup(jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
77  *                 .addComponent(jLabel2)
78  *                 .addComponent(tfNumero2, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
79  *             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
80  *             .addGroup(jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
81  *                 .addComponent(jLabel3)
82  *                 .addComponent(tfResultado, javax.swing.GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
83  *             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
84  *             .addGroup(jPanel2Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
85  *                 .addComponent(bSumar)
86  *                 .addComponent(bRestart)
87  *                 .addComponent(bSalir))
88  *             .addContainerGap())
89  *     );
90  * }
91  */

```

Creando proyecto con interfaz Swing IV

Todas las imágenes utilizadas pertenecen al Ministerio de Educación y FP con licencia CC BY-NC y son capturas de pantalla de la aplicación Netbeans, propiedad de Oracle.

[Resumen textual alternativo](#)

Has podido comprobar que el diseño de interfaces con el editor gráfico de Netbeans es relativamente sencillo. En otros entornos de desarrollo la forma de proceder es parecida. En las siguientes unidades veremos cómo dar funcionalidad a nuestra aplicación.

3.2- Eventos.

Caso práctico



Ministerio de Educación y FP ([CC BY-NC](#))

Ana sabe que entender cómo funciona la programación por eventos es algo relativamente fácil, el problema está en utilizar correctamente los eventos más adecuados en cada momento. **Ana** tiene muy claro que el evento se asocia a un botón cuando se pulsa, pero **José Javier** la pone en duda, cuando la llama por teléfono para preguntarle unas dudas y le dice, que él cree, que el evento se produce cuando el botón se suelta. Además, le recuerda que en clase dijeron que al ser enfocado con el ratón, o al accionar una combinación de teclas asociadas, también se pueden producir eventos. Tras hablarlo, piensan que realmente no es tan complicado, porque se repiten muchos eventos y si nos paramos a pensarlo, todos ellos son predecibles y bastante lógicos.

3.2.1.- Introducción.

¿Qué es un **evento**?

Es todo hecho que ocurre mientras se ejecuta la aplicación. Normalmente, llamamos evento a cualquier interacción que realiza el usuario con la aplicación, como puede ser:

- ✓ pulsar un botón con el ratón;

- ✓ hacer doble clic;
- ✓ pulsar y arrastrar;
- ✓ pulsar una combinación de teclas en el teclado;
- ✓ pasar el ratón por encima de un componente;
- ✓ salir el puntero de ratón de un componente;
- ✓ abrir una ventana;
- ✓ etc.

¿Qué es la **programación guiada por eventos**?

Imagina la ventana de cualquier aplicación, por ejemplo la de un procesador de textos. En esa ventana aparecen multitud de elementos gráficos interactivos, de forma que no es posible que el programador haya previsto todas las posibles entradas que se pueden producir por parte del usuario en cada momento.

Con el control de flujo de programa de la **programación imperativa**, el programador tendría que estar continuamente leyendo las entradas (de teclado, o ratón, etc) y comprobar para cada entrada o interacción producida por el usuario, de cual se trata de entre todas las posibles, usando estructuras de flujo condicional (**if-then-else, switch**) para ejecutar el código conveniente en cada caso. Si piensas que para cada opción del menú, para cada botón o etiqueta, para cada lista desplegable, y por tanto para cada componente de la ventana, incluyendo la propia ventana, habría que comprobar todos y cada uno de los eventos posibles, nos damos cuenta de que las posibilidades son casi infinitas, y desde luego impredecibles. Por tanto, de ese modo es imposible solucionar el problema.

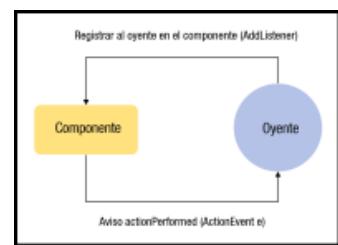
Para abordar el problema de tratar correctamente las interacciones del usuario con la interfaz gráfica de la aplicación hay que cambiar de estrategia, y la **programación guiada por eventos es una buena solución**, veamos cómo funciona el modelo de gestión de eventos.

3.2.2.- **Modelo de gestión de eventos.**

¿Qué sistema operativo utilizas? ¿Posee un entorno gráfico? Hoy en día, la mayoría de sistemas operativos utilizan interfaces gráficas de usuario. Este tipo de sistemas operativos están **continuamente monitorizando el entorno para capturar y tratar los eventos** que se producen.

El sistema operativo informa de estos eventos a los programas que se están ejecutando y entonces cada programa decide, según lo que se haya programado, qué hace para dar respuesta a esos eventos.

Cada vez que el usuario realiza una determinada acción sobre una aplicación que estamos programando en Java, un clic sobre el ratón, presionar una tecla, etc., se produce un evento que el sistema operativo transmite a Java.



Ministerio de Educación y FP ([CC BY-NC](#))

Java crea un objeto de una determinada clase de evento, y este evento se transmite a un determinado método para que lo gestione.

El **modelo de eventos de Java está basado en delegación**, es decir, la responsabilidad de gestionar un evento que ocurre en un objeto fuente la tiene otro objeto **oyente**.

Las **fuentes de eventos** (event sources) son objetos que detectan eventos y notifican a los receptores que se han producido dichos eventos. Ejemplos de fuentes:

- ✓ Botón sobre el que se pulsa o pincha con el ratón.
- ✓ Campo de texto que pierde el foco.
- ✓ Campo de texto sobre el que se presiona una tecla.
- ✓ Ventana que se cierra.
- ✓ Etc.

En el apartado anterior de creación de interfaces con ayuda de los asistentes del IDE, vimos lo fácil que es realizar este tipo de programación, ya que el IDE hace muchas cosas, genera código automáticamente por nosotros.

Pero también podríamos hacerlo nosotros todo, si no tuviéramos un IDE como NetBeans, o porque simplemente nos apeteciera hacerlo todo desde código, sin usar asistentes ni diseñadores gráficos. En este caso, **los pasos a seguir** se pueden resumir en:

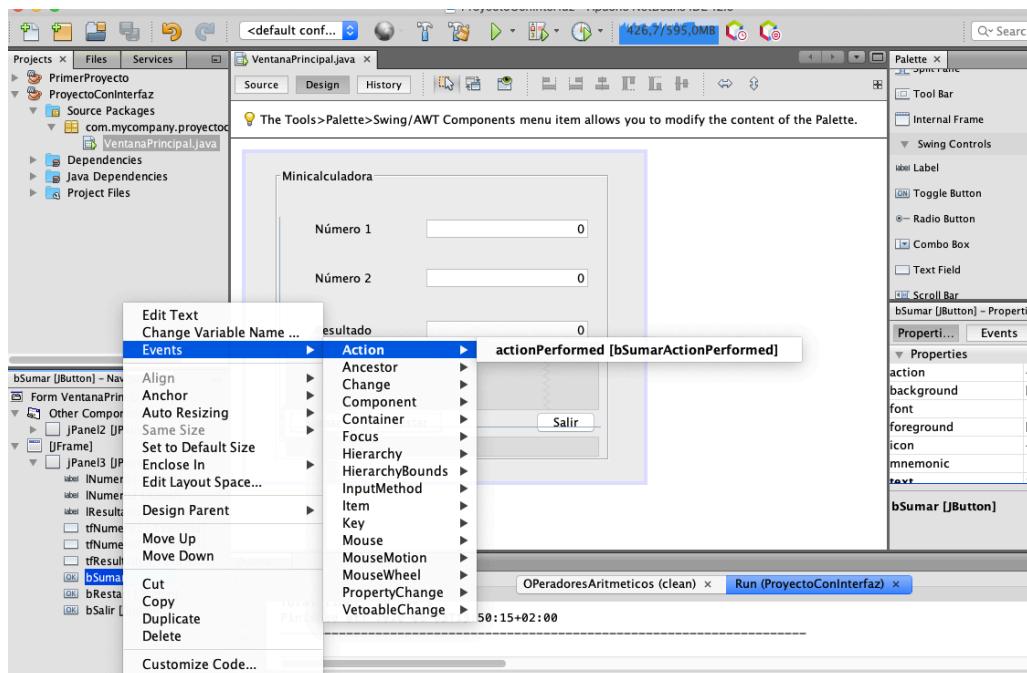
1. Crear la clase oyente que implemente la interfaz.
 - ✓ Ej. `ActionListener`: pulsar un botón.
2. Implementar en la clase oyente los métodos de la interfaz.
 - ✓ Ej. `void actionPerformed(ActionEvent)`.

3. Crear un objeto de la clase oyente y registrarlo como oyente en uno o más componentes gráficos que proporcionen interacción con el usuario.

Vamos a añadir funcionalidad a nuestra minicalculadora. Cuando el usuario pulse el botón Sumar se debe realizar la suma de los dos números y mostrar el resultado en el campo resultado.

Añadiendo funcionalidad a nuestra calculadora

Para añadir funcionalidad a nuestra calculadora necesitamos trabajar con eventos. En este caso, el elemento que lanzará el evento será el botón **Sumar**: cuando sea pulsado se deben realizar una serie de acciones que permitan sumar los operandos y mostrar el resultado en el campo **Resultado**. Para registrar el evento en el botón accedemos a sus propiedades:



Añadiendo funcionalidad a nuestra calculadora II

En nuestro caso, nos interesa el evento **ActionPerformed**: este evento será lanzado cuando su pulse sobre el botón. Para asociar el evento al botón tan solo tendremos que seleccionarlo. Otra forma de hacer lo mismo sería haciendo doble click con el ratón sobre el elemento en el diseñador.

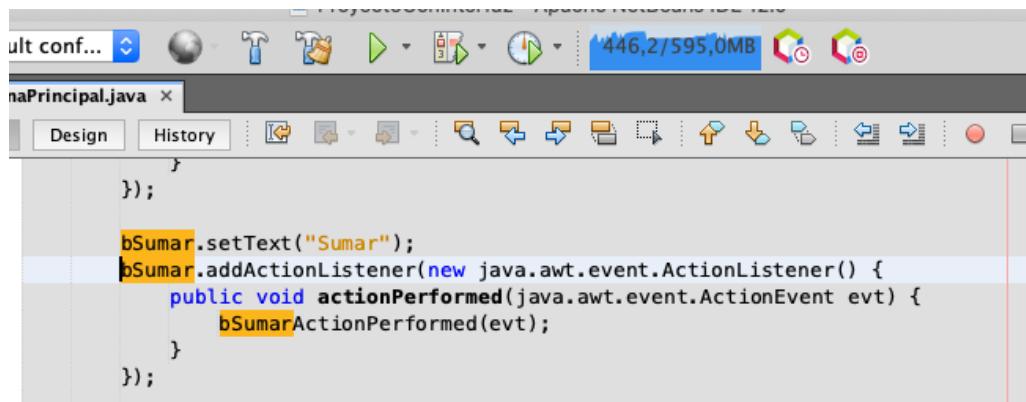
Inmediatamente se abrirá el editor de código fuente para incluir el código a ejecutar cuando se produzca el evento, es decir, el código del manejador de eventos. Obsérvalo en la siguiente imagen:

```
70
71  private void bSumarActionPerformed(java.awt.event.ActionEvent evt) {
72      // TODO add your handling code here:
73
74      //Declaramos variables para realizar los cálculos.
75      int num1, num2, resul;
76
77      num1= Integer.parseInt(tfNumero1.getText());
78      num2= Integer.parseInt(tfNumero2.getText());
79
80      //Ya hemos recogido el valor de los operandos de los campos de texto de la interfaz. Realizamos la sum
81      resul=num1+num2;
82
83      tfResultado.setText(String.valueOf(resul));
84  }
85
```

1. Observa como en el código del manejador de evento recogemos el valor de los campos de texto que contienen los números a sumar (`tfNumero1` y `tfNumero2`) a través de su método `getText()`. Los valores son convertidos a tipo entero.
2. A continuación se realiza la suma.
3. Por último y a través del método `setText()`, se muestra el resultado en el campo de texto `tfResultado`, previa conversión a cadena de caracteres.

Añadiendo funcionalidad a nuestra calculadora III

¿Pero cómo se ha asociado el escuchador de eventos al botón? Observa el código de la imagen.

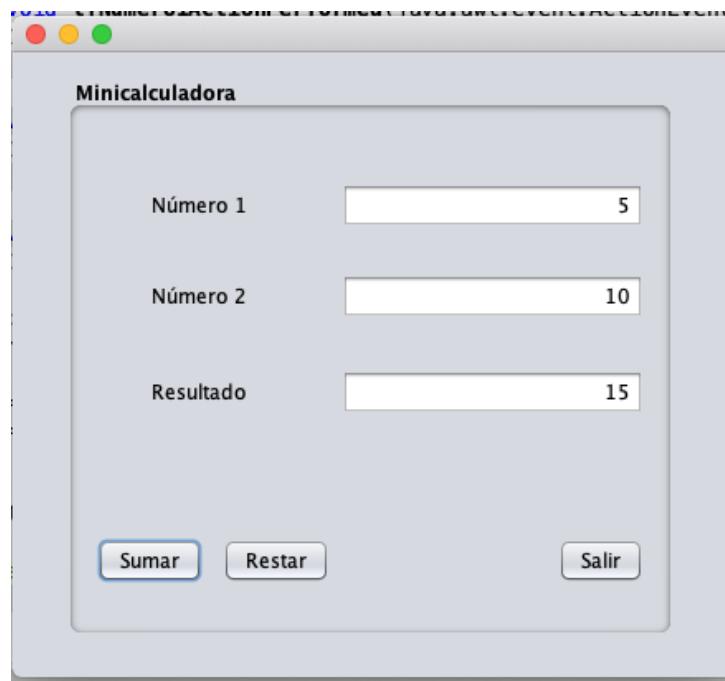


```
        });
        bSumar.setText("Sumar");
        bSumar.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                bSumarActionPerformed(evt);
            }
        });
    }
```

Al botón **bSumar** se le ha añadido un escuchador de eventos a través del método `addActionListener()`. Ese escuchador es un objeto `ActionListener` que debe implementar el método `actionPerformed` (se crea una clase sin referencia porque no necesitamos referenciarla en ningún momento a través de su identificador). Realmente la implementación de dicho método es el manejador de eventos, es decir, el código que se ejecutará cuando se produzca el evento. En este ejemplo, se invoca al método que hemos analizado en el paso anterior.

Añadiendo funcionalidad a nuestra calculadora IV

Ejecutamos la aplicación y comprobamos su funcionamiento.



Añadiendo funcionalidad a nuestra calculadora V

Todas las imágenes utilizadas pertenecen al Ministerio de Educación y FP con licencia CC BY-NC y son capturas de pantalla de la aplicación Netbeans, propiedad de Oracle.

[Resumen textual alternativo](#)

Autoevaluación

Con la programación guiada por eventos, el programador se concentra en estar continuamente leyendo las entradas de teclado, de ratón, etc., para comprobar cada entrada o interacción producida por el usuario.

- Verdadero Falso

Falso

El programador o programadora sólo debe preocuparse de la lógica de negocio.

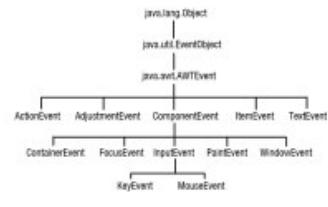
3.2.3.- Tipos de eventos.

En la mayor parte de la literatura escrita sobre Java, encontrarás dos tipos básicos de eventos:

- ✓ **Físicos** o de **bajo nivel**: que corresponden a un evento hardware claramente identificable. Por ejemplo, se pulsó una tecla (*KeyStrokeEvent*). Destacar los siguientes:
 - ◆ En componentes: *ComponentEvent*. Indica que un componente se ha movido, cambiado de tamaño o de visibilidad
 - ◆ En contenedores: *ContainerEvent*. Indica que el contenido de un contenedor ha cambiado porque se añadió o eliminó un componente.
 - ◆ En ventanas: *WindowEvent*. Indica que una ventana ha cambiado su estado.
 - ◆ *FocusEvent*, indica que un componente ha obtenido o perdido la entrada del foco.
- ✓ **Semánticos** o de mayor nivel de abstracción: se componen de un conjunto de eventos físicos, que se suceden en un determinado orden y tienen un significado más abstracto. Por ejemplo: el usuario elige un elemento de una lista desplegable (*ItemEvent*).
 - ◆ *ActionEvent*, *ItemEvent*, *TextEvent*, *AdjustmentEvent*.

Los eventos en Java se organizan en una jerarquía de clases:

- ✓ La clase *java.util.EventObject* es la clase base de todos los eventos en Java.
- ✓ La clase *java.awt.AWTEvent* es la clase base de todos los eventos que se utilizan en la construcción de GUI.
- ✓ Cada tipo de evento *loqueseaEvent* tiene asociada una interfaz *loqueseaListener* que nos permite definir manejadores de eventos.
- ✓ Con la idea de simplificar la implementación de algunos manejadores de eventos, el paquete *java.awt.event* incluye clases *loqueseaAdapter* que implementan las interfaces *loqueseaListener*.



José Javier Bermúdez Hernández ([CC BY-NC](#))

Autoevaluación

El evento que se dispara cuando le llega el foco a un botón es un evento de tipo físico.

- Verdadero Falso

Verdadero

En efecto es un evento físico, tanto a un botón como a cualquier otro componente.

3.2.4.- Eventos de teclado.

Los eventos de teclado se generan como respuesta a que el usuario pulsa o libera una tecla mientras un componente tiene el foco de entrada.



Long Zheng (CC BY-NC-SA)

KeyListener (oyente de teclas).

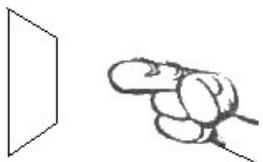
Método	Causa de la invocación
keyPressed (KeyEvent e)	Se ha pulsado una tecla.
keyReleased (KeyEvent e)	Se ha liberado una tecla.
keyTyped (KeyEvent e)	Se ha pulsado (y a veces soltado) una tecla.

KeyEvent (evento de teclas)

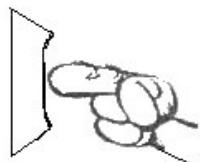
Métodos más usuales	Explicación
char getKeyChar()	Devuelve el carácter asociado con la tecla pulsada.
int getKeyCode()	Devuelve el valor entero que representa la tecla pulsada.
String getKeyText()	Devuelve un texto que representa el código de la tecla.
Object getSource()	Método perteneciente a la clase EventObject . Indica el objeto que produjo el evento.

La clase **KeyEvent**, define muchas constantes así:

- ✓ **KeyEvent.VK_A** especifica la tecla A.
- ✓ **KeyEvent.VK_ESCAPE** especifica la tecla ESCAPE.



Botón en estado normal.



Al pulsar la tecla se disparará el evento **KeyPressed**.



Al liberar la tecla se genera el evento **KeyReleased**.

José Javier Bermúdez Hernández ([CC BY-NC](#))

En el siguiente enlace tienes el código del proyecto que te puedes descargar. En él se puede ver un ejemplo del uso eventos. En concreto vemos cómo se están capturando los eventos que se producen al pulsar una tecla y liberarla. El programa escribe en un área de texto las teclas que se oprimen.

[Código Java comentado de un oyente de teclado](#)

3.2.5.- Eventos de ratón.

Similarmente a los eventos de teclado, los eventos del ratón se generan como respuesta a que el usuario pulsa o libera un botón del ratón, o lo mueve sobre un componente.

MouseListener (oyente de ratón)

Método	Causa de la invocación
mousePressed (MouseEvent e)	Se ha pulsado un botón del ratón en un componente.
mouseReleased (MouseEvent e)	Se ha liberado un botón del ratón en un componente.
mouseClicked (MouseEvent e)	Se ha pulsado y liberado un botón del ratón sobre un componente.
mouseEntered (KeyEvent e)	Se ha entrado (con el puntero del ratón) en un componente.
mouseExited (KeyEvent e)	Se ha salido (con el puntero del ratón) de un componente.

MouseMotionListener (oyente de ratón)

Método	Causa de la invocación
mouseDragged (MouseEvent e)	Se presiona un botón y se arrastra el ratón.
mouseMoved (MouseEvent e)	Se mueve el puntero del ratón sobre un componente.

MouseWheelListener (oyente de ratón)

Método	Causa de la invocación
MouseWheelMoved (MouseWheelEvent e)	Se mueve la rueda del ratón.

En el siguiente proyecto podemos ver una demostración de un formulario con dos botones. Implementamos un oyente `MouseListener` y registramos los dos botones para detectar tres de los cinco eventos del interface.

[Código Java comentado de un oyente de ratón](#)

Como se ve en el código, se deja en blanco el cuerpo de `mouseEntered` y de `mouseExited`, ya que no nos interesan en este ejemplo. Cuando se desea escuchar algún tipo de evento, de deben implementar todos los métodos del interface para que la clase no tenga que ser definida como abstracta. Para evitar tener que hacer esto, podemos utilizar adaptadores.



Gianfranco Degrandi (CC BY-NC-SA)

Para saber más

En el enlace que ves a continuación, hay también un ejemplo interesante de la programación de eventos del ratón.

[La programación del ratón](#)

En el siguiente vídeo puede ver el primero de una serie de cinco vídeos en los que se realiza una calculadora con la ayuda de los asistentes de NetBeans.

<https://www.youtube.com/embed/A9ZX5rWcDOE>

Autoevaluación

Cuando el usuario deja de pulsar una tecla se invoca a `keyReleased(KeyEvent e)`.

- Verdadero Falso

Verdadero

La afirmación es correcta, ese es el evento que se dispara.

Debes conocer

En enlace este tienes un documento muy interesante, paso a paso, sobre construcción de interfaces gráficos con NetBeans.

[Construcción de interfaces gráficos de usuario con NetBeans \(399 KB\)](#)

3.3.- Generación de programas en entorno gráfico.

Caso práctico



José Javier Bermúdez Hernández [\(CC BY-NC\)](#)

José Javier va de camino a la clase práctica en la que él, y el resto de la clase, van a probar a hacer sus primeros pasos en programación visual con entornos gráficos, el profesor les explica que al principio parece que todo es muy fácil sobre el papel, pero en realidad crear un proyecto con componentes gráficos no siempre es fácil. Pero el profesor lo tiene claro, hay que empezar por los contenedores que, como su propio nombre indica, se emplean para contener o ubicar al resto de componentes.

En este mismo tema, más arriba, has visto la lista de los principales componentes Swing que se incluyen en la mayoría de las aplicaciones, junto con una breve descripción de su uso, y una imagen que nos da una idea de cuál es su aspecto.

También hemos visto un ejemplo de cómo crear con la ayuda de NetBeans unos sencillos programas. Ahora, vamos a ver con mayor detalle, los componentes más típicos utilizados en los programas en Java y cómo incluirlos dentro de una aplicación.



[DraXus \(CC BY\)](#)

Citas para pensar

"Algo sólo es imposible hasta que alguien lo dude y termine probando lo contrario".

Albert Einstein

3.3.1.- Contenedores.

Por los ejemplos vistos hasta ahora, ya te habrás dado cuenta de la necesidad de que cada aplicación "contenga" de alguna forma esos componentes. ¿Qué componentes se usan para contener a los demás?

En Swing esa función la desempeñan un grupo de componentes llamados **contenedores** Swing.

Existen dos tipos de elementos contenedores:

- ✓ **Contenedores de alto nivel** o "peso pesado".
 - ◆ Marcos: **JFrame** y **JDialog** para aplicaciones
 - ◆ **JApplet**, para applets.
- ✓ **Contenedores de bajo nivel** o "peso ligero". Son los paneles: **JRootPane** y **JPanel**.



José Javier Bermúdez Hernández. ([CC BY-NC](#))

Cualquier aplicación, con interfaz gráfica de usuario típica, comienza con la apertura de una ventana principal, que suele contener la barra de título, los botones de minimizar, maximizar/restaurar y cerrar, y unos bordes que delimitan su tamaño.

Esa ventana constituye un marco dentro del cual se van colocando el resto de componentes que necesita el programador: menú, barras de herramientas, barra de estado, botones, casillas de verificación, cuadros de texto, etc.

Esa ventana principal o marco sería el contenedor de alto nivel de la aplicación.

Toda aplicación de interfaz gráfica de usuario Java tiene, al menos, un contenedor de alto nivel.

Los contenedores de alto nivel extienden directamente a una clase similar de AWT, es decir, **JFrame** extiende de **Frame**. Es decir, realmente necesitan crear una ventana del sistema operativo independiente para cada uno de ellos.

Los demás componentes de la aplicación no tienen su propia ventana del sistema operativo, sino que se dibujan en su objeto contenedor.

En los ejemplos anteriores del tema, hemos visto que podemos añadir un **JFrame** desde el diseñador de NetBeans, o bien escribiéndolo directamente por código. De igual forma para los componentes que añadamos sobre el.

Para saber más

Te recomendamos que mires la siguiente web para ver información sobre **JFrame** y **JDialog**.

[JFrame y JDialog](#)

Autoevaluación

Cualquier componente de gráfico de una aplicación Java necesita tener su propia ventana del sistema operativo.

- Verdadero Falso

Falso

Sólo lo necesitan los contenedores de alto nivel.

3.3.2.- Cerrar la aplicación.

Cuando quieras terminar la ejecución de un programa, ¿qué sueles hacer? Pues normalmente pinchar en el ícono de cierre de la ventana de la aplicación.

En Swing, una cosa es cerrar una ventana, y otra es que esa ventana deje de existir completamente, o cerrar la aplicación completamente.

- ✓ **Se puede hacer que una ventana no esté visible**, y sin embargo que ésta siga existiendo y ocupando memoria para todos sus componentes, usando el método `setVisible(false)`. En este caso bastaría ejecutar para el `JFrame` el método `setVisible(true)` para volver a ver la ventana con todos sus elementos.
- ✓ **Si queremos cerrar la aplicación**, es decir, que no sólo se destruya la ventana en la que se mostraba, sino que se destruyan y liberen todos los recursos (memoria y CPU) que esa aplicación tenía reservados, tenemos que invocar al método `System.exit(0)`.
- ✓ **También se puede invocar para la ventana JFrame al método dispose()**, heredado de la clase `Window`, que no requiere ningún argumento, y que borra todos los recursos de pantalla usados por esta ventana y por sus componentes, así como cualquier otra ventana que se haya abierto como hija de esta (dependiente de esta). Cualquier memoria que ocupara esta ventana y sus componentes se libera y se devuelve al sistema operativo, y tanto la ventana como sus componentes se marcan como "no representables". Y sin embargo, el objeto ventana sigue existiendo, y podría ser reconstruido invocando al método `pack()` o la método `show()`, aunque deberían construir de nuevo toda la ventana.

Las ventanas `JFrame` de Swing permiten establecer una operación de cierre por defecto con el método `setDefaultCloseOperation()`, definido en la clase `JFrame`.

¿Cómo se le indica al método el modo de cerrar la ventana?

Los valores que se le pueden pasar como parámetros a este método son una serie de constantes de clase:

- ✓ **DO NOTHING ON CLOSE**: **no hace nada**, necesita que el programa maneje la operación en el método `windowClosing()` de un objeto `WindowListener` registrado para la ventana.
- ✓ **HIDE ON CLOSE**: **Oculta** de ser mostrado en la pantalla pero no destruye el marco o ventana después de invocar cualquier objeto `WindowListener` registrado.
- ✓ **DISPOSE ON CLOSE**: **Oculta y termina (destruye) automáticamente el marco o ventana** después de invocar cualquier objeto `WindowListener` registrado.
- ✓ **EXIT ON CLOSE**: Sale de la aplicación usando el método `System.exit(0)`. Al estar definida en `JFrame`, se puede usar con aplicaciones, pero no con applets.

Autoevaluación

`System.exit(0)` oculta la ventana pero no libera los recursos de CPU y memoria que la aplicación tiene reservados.

- Verdadero Falso

Falso

En efecto, sí que libera también esos recursos.

3.3.3.- Organizadores de contenedores: layout managers.

Los layout managers son fundamentales en la creación de interfaces de usuario, ya que determinan las posiciones de los controles en un contenedor.

En lenguajes de programación para una única plataforma, el problema sería menor porque el aspecto sería fácilmente controlable. Sin embargo, dado que Java está orientado a la portabilidad del código, éste es uno de los aspectos más complejos de la creación de interfaces, ya que las medidas y posiciones dependen de la máquina en concreto.

En algunos entornos los componentes se colocan con coordenadas absolutas. En Java se desaconseja esa práctica porque en la mayoría de casos es imposible prever el tamaño de un componente.

Por tanto, en su lugar, se usan organizadores o también llamados **administradores de diseño** o **layout managers** o **gestores de distribución** que permiten colocar y maquetar de forma independiente de las coordenadas.

Debemos hacer un buen diseño de la interfaz gráfica, y así tenemos que elegir el mejor gestor de distribución para cada uno de los contenedores o paneles de nuestra ventana.

Esto podemos conseguirlo con el método `setLayout()`, al que se le pasa como argumento un objeto del tipo de Layout que se quiere establecer.

En NetBeans, una vez insertado un `JFrame`, si nos situamos sobre él y pulsamos botón derecho, se puede ver, como muestra la imagen, que aparece un menú, el cual nos permite elegir el layout que queramos.



José Javier Bermúdez Hernández ([CC BY-NC](#))

Debes conocer

En la siguiente web puedes ver gráficamente los distintos layouts:

[LayOuts \(en inglés\).](#)

Este otro enlace contiene ejemplos de uso de los distintos layout.

[Ejemplos de layouts](#)

Autoevaluación

Cuando programamos en Java es aconsejable establecer coordenadas absolutas, siempre que sea posible, en nuestros componentes.

Verdadero Falso

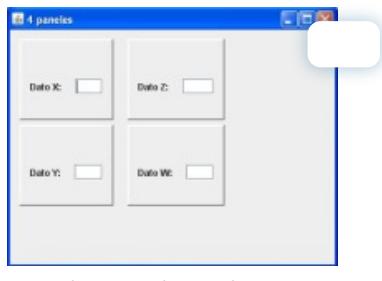
Falso

¡No se aconseja utilizar coordenadas absolutas!

3.3.4.- Contenedor ligero: JPanel.

Es la clase utilizada como contenedor genérico para agrupar componentes. Normalmente cuando una ventana de una aplicación con interfaz gráfica cualquiera presenta varios componentes, para hacerla más atractiva y ordenada al usuario se suele hacer lo siguiente:

- ✓ Crear un marco, un **JFrame**.
- ✓ Para organizar mejor el espacio en la ventana, añadiremos varios paneles, de tipo **JPanel**. (Uno para introducir los datos de entrada, otro para mostrar los resultados y un tercero como zona de notificación de errores.)
- ✓ Cada uno de esos paneles estará delimitado por un borde que incluirá un título. Para ello se usan las clases disponibles en **BorderFactory** (**BevelBorder**, **CompoundBorder**, **EmptyBorder**, **EtchedBorder**, **LineBorder**, **LoweredBevelBorder**, **MatteBorder** y **TitledBorder**) que nos da un surtido más o menos amplio de tipos de bordes a elegir.
- ✓ En cada uno de esos paneles se incluyen las etiquetas y cuadros de texto que se necesitan para mostrar la información adecuadamente.



José Javier Bermúdez Hernández (CC BY-NC)

Con NetBeans es tan fácil como arrastrar tantos controles **JPanel** de la paleta hasta el **JFrame**. Por código, también es muy sencillo, por ejemplo podríamos crear un panel rojo, darle sus características y añadirlo al **JFrame** del siguiente modo:

Cuando en Sun desarrollaron Java, los diseñadores de Swing, por alguna circunstancia, determinaron que para algunas funciones, como añadir un **JComponent**, los programadores no pudiéramos usar **JFrame.add**, Sino que en lugar de ello, primeramente, tuviéramos que obtener el objeto **Container** asociado con **JFrame.getContentPane()**, y añadirlo.

Sun se dio cuenta del error y ahora permite utilizar **JFrame.add**, desde Java 1.5 en adelante. Sin embargo, podemos tener el problema de que un código desarrollado y ejecutado correctamente en 1.5 falle en máquinas que tengan instalado 1.4 o anteriores.

Para saber más

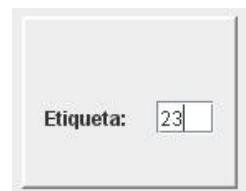
En la siguiente web puedes ver algo más sobre paneles.

[Paneles.](#)

3.3.5.- Etiquetas y campos de texto.

Los **cuadros de texto** Swing vienen implementados en Java por la clase **JTextField**.

Para insertar un campo de texto, el procedimiento es tan fácil como: **seleccionar el botón correspondiente a JTextField en la paleta de componentes**, en el diseñador, y **pinchar sobre el área de diseño** encima del panel en el que queremos situar ese campo de texto. El tamaño y el lugar en el que se sitúe, dependerá del Layout elegido para ese panel.



José Javier Bermúdez Hernández (CC BY-NC)

El componente Swing etiqueta **JLabel**, se utiliza para crear etiquetas de modo que podamos insertarlas en un marco o un panel para visualizar un texto estático, que no puede editar el usuario.

Los constructores son:

- ✓ **JLabel()**. Crea un objeto **JLabel** sin nombre y sin ninguna imagen asociada.
- ✓ **JLabel(Icon imagen)**. Crea un objeto sin nombre con una imagen asociada.
- ✓ **JLabel(Icon imagen, int alineacionHorizontal)**. Crea una etiqueta con la imagen especificada y la centra en horizontal.
- ✓ **JLabel(String texto)**. Crea una etiqueta con el texto especificado.
- ✓ **JLabel(String texto, Icon icono, int alineacionHorizontal)**. Crea una etiqueta con el texto y la imagen especificada y alineada horizontalmente.
- ✓ **JLabel(String texto, int alineacionHorizontal)**. Crea una etiqueta con el texto especificado y alineado horizontalmente.

Si quieras conocer en profundidad las clases **JLabel** y **JTextField**, puedes consultar la documentación de Java.

[Propiedades de JLabel](#)

Para saber más

En el siguiente enlace puedes ver cómo usar `DecimalFormat` para presentar un número en un `JTextField` o recoger el texto del `JTextField` y reconstruir el número.

[Formatear cuadro de texto.](#)

Autoevaluación

Un componente `JLabel` permite al usuario de la aplicación en ejecución cambiar el texto que muestra dicho componente.

- Verdadero Falso

Falso

No se puede cambiar puesto que es un texto estático.

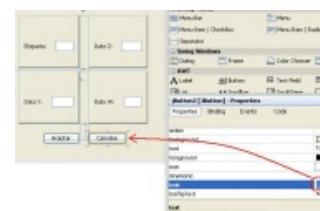
3.3.6.- Botones.

Ya has podido comprobar que prácticamente todas las aplicaciones incluyen botones que al ser pulsados efectúan alguna acción: hacer un cálculo, dar de alta un libro, aceptar modificaciones, etc.

Estos botones se denominan **botones de acción**, precisamente porque realizan una acción cuando se pulsan. En Swing, la clase que los implementa en Java es la `JButton`.

Los principales métodos son:

- ✓ `void setText(String)`. Asigna el texto al botón.
- ✓ `String getText()`. Recoge el texto.



José Javier Bermúdez Hernández ([CC BY-NC](#))

Hay un tipo especial de botones, que se comportan como **interruptores de dos posiciones** o estados (pulsados-on, no pulsados-off). Esos botones especiales se denominan botones on/off o `JToggleButton`.

Para saber más

A continuación puedes ver un par de enlaces: uno en el que se diseña una interfaz gráfica de usuario sencilla, con los controles que hemos visto hasta ahora, y el segundo enlace, en inglés, un ejemplo para añadir un botón en Java por código.

[Diseño de una GUI sencilla.](#) (0.70 MB)

[JButton.](#)

3.3.7.- Casillas de verificación y botones de radio.

Las casillas de verificación en Swing están implementadas para Java por la clase `JCheckBox`, y los botones de radio o de opción por la clase `JRadioButton`. Los grupos de botones, por la clase `ButtonGroup`.

La funcionalidad de ambos componentes es en realidad la misma.

- ✓ Ambos tienen **dos "estados"**: seleccionados o no seleccionados (marcados o no marcados).
- ✓ Ambos **se marcan o desmarcan** usando el método `setSelected(boolean estado)`, que establece el valor para su propiedad `selected`. (El estado toma el valor `true` para seleccionado y `false` para no seleccionado).
- ✓ A ambos le **podemos preguntar si están seleccionados o no**, mediante el método `isSelected()`, que devuelve `true` si el componente está seleccionado y `false` si no lo está.
- ✓ Para ambos **podemos asociar un ícono distinto** para el estado de **seleccionado** y el de **no seleccionado**.



José Javier Bermúdez Hernández (CC BY-NC)

`JCheckBox` pueden usarse en menús mediante la clase `JCheckBoxMenuItem`.

`ButtonGroup` permite agrupar una serie de casillas de verificación (`JRadioButton`), de entre las que sólo puede seleccionarse una. Marcar una de las casillas implica que el resto sean desmarcadas automáticamente. La forma de hacerlo consiste en añadir un `ButtonGroup` y luego, agregarle los botones.

Cuando en **un contenedor** aparezcan **agrupados** varios **botones de radio** (o de opción), **entenderemos** que no son opciones independientes, sino que sólo uno de ellos podrá estar activo en cada momento, y necesariamente uno debe estar activo. Por tanto en ese contexto entendemos que son opciones excluyentes entre sí.

Autoevaluación

Los componentes radiobotones y las casillas de verificación tienen ambos dos estados: seleccionado y no seleccionado.

Verdadero Falso

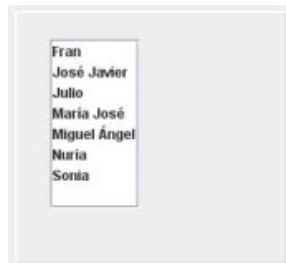
Verdadero

Así es, presentan esos dos estados.

3.3.8.- Listas.

En casi todos los programas, nos encontramos con que se pide al usuario que introduzca un dato, y además un dato de entre una lista de valores posibles, no un dato cualquiera.

La clase `JList` constituye un componente lista sobre el que se puede ver y seleccionar uno o varios elementos a la vez. En caso de haber más elementos de los que se pueden visualizar, es posible utilizar un componente `JScrollPane` para que aparezcan barras de desplazamiento.



En los componentes `JList`, un modelo `ListModel` representa **los contenidos de la lista**. Esto significa que los datos de la lista se guardan en una estructura de datos independiente, denominada modelo de la lista. Es fácil mostrar en una lista los elementos de un vector o array de objetos, usando un constructor de `JList` que cree una instancia de `ListModel` automáticamente a partir de ese vector.

Vemos a continuación un ejemplo para crear una lista `JList` que muestra las cadenas contenidas en el vector `info[]`:

```
String[] info = {"Pato", "Loro", "Perro", "Cuervo"};
```

```

JList listaDatos = new JList(info);

/* El valor de la propiedad model de JList es un objeto que proporciona una visión de sólo lectura del vector info[].

El método getModel() permite recoger ese modelo en forma de Vector de objetos, y utilizar con los métodos de la clase

Vector, como getElementAt(i), que proporciona el elemento de la posición i del Vector. */

for (int i = 0; i < listaDatos.getModel().getSize(); i++) {

System.out.println(listaDatos.getModel().getElementAt(i));

}

```

Para saber más

A continuación puedes ver como crear un **JList**, paso a paso, en el enlace siguiente.

[Crear un JList paso a paso.](#)

En el enlace que tienes a continuación puedes ver un vídeo que muestra cómo trabajar con un componente lista en Netbeans.

<https://www.youtube.com/embed/7eb-3bKz7js>

[Resumen textual alternativo](#)

3.3.9.- Listas (II).

Cuando trabajamos con un componente **JList**, podemos seleccionar un único elemento, o varios elementos a la vez, que a su vez pueden estar contiguos o no contiguos. La posibilidad de hacerlo de una u otra manera la establecemos con la propiedad **selectionMode**.

Los valores posibles para la propiedad **selectionMode** para cada una de esas opciones son las siguientes constantes de clase del interface **ListSelectionModel**:

- ✓ **MULTIPLE_INTERVAL_SELECTION**: Es el valor por defecto. Permite seleccionar múltiples intervalos, manteniendo pulsada la tecla **CTRL** mientras se seleccionan con el ratón uno a uno, o la tecla de mayúsculas, mientras se pulsa el primer elemento y el último de un intervalo.
- ✓ **SINGLE_INTERVAL_SELECTION**: Permite seleccionar un único intervalo, manteniendo pulsada la tecla mayúsculas mientras se selecciona el primer y último elemento del intervalo.
- ✓ **SINGLE_SELECTION**: Permite seleccionar cada vez un único elemento de la lista.



Ministerio de Educación y FP. IdITE=111238 (CC BY-NC)

Podemos establecer el valor de la propiedad **selectedIndex** con el método **setSelectedIndex()** es decir, el índice del elemento seleccionado, para seleccionar el elemento del índice que se le pasa como argumento.

Como hemos comentado, los datos se almacenan en un modelo que al fin y al cabo es un vector, por lo que tiene sentido hablar de índice seleccionado.

También se dispone de un método **getSelectedIndex()** con el que podemos averiguar el índice del elemento seleccionado.

El método **getSelectedValue()** devuelve el objeto seleccionado, de tipo **Object**, sobre el que tendremos que aplicar un "casting" explícito para obtener el elemento que realmente contiene la lista (por ejemplo un **String**). Observa que la potencia de usar como modelo un vector de **Object**, es que en el **JList** podemos mostrar realmente cualquier cosa, como por ejemplo, una imagen.

El método `setSelectedValue()` permite establecer cuál es el elemento que va a estar seleccionado.

Si se permite hacer selecciones múltiples, contamos con los métodos:

- ✓ `setSelectedIndices()`, al que se le pasa como argumento un vector de enteros que representa los índices a seleccionar.
- ✓ `getSelectedIndices()`, que devuelve un vector de enteros que representa los índices de los elementos o ítems que en ese momento están seleccionados en el `JList`.
- ✓ `getSelectedValues()`, que devuelve un vector de `Object` con los elementos seleccionados en ese momento en el `JList`.

Autoevaluación

Los componentes `JList` permiten la selección de elementos de una lista, siempre que estén uno a continuación del otro de manera secuencial.

Verdadero Falso

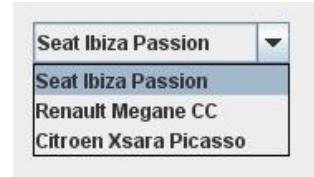
Falso

Se puede usar `MULTIPLE_INTERVAL_SELECTION` para seleccionar varios elementos no contiguos.

3.3.10.- Listas desplegables.

Una **lista desplegable** se representa en Java con el componente Swing `JComboBox`. Consiste en una lista en la que sólo se puede elegir una opción. Se pueden crear `JComboBox` tanto editables como no editables.

Una lista desplegable es una mezcla entre un campo de texto editable y una lista. Si la propiedad `editable` de la lista desplegable la fijamos a verdadero, o sea a `true`, el usuario podrá seleccionar uno de los valores de la lista que se despliega al pulsar el botón de la flecha hacia abajo y dispondrá de la posibilidad de teclear directamente un valor en el campo de texto.



José Javier Bermúdez Hernández. ([CC BY-NC](#))

Establecemos la propiedad `editable` del `JComboBox` el método `setEditable()` y se comprueba con el método `isEditable()`. La clase `JComboBox` ofrece una serie de métodos que tienen nombres y funcionalidades similares a los de la clase `JList`.

Para saber más

En el siguiente enlace tienes algún ejemplo comentado para crear `JComboBox` por código.

[Ejemplo JComboBox](#)

3.3.11.- Menús.

En las aplicaciones informáticas siempre se intenta que el usuario disponga de un menú para facilitar la localización una operación. La filosofía, al crear un menú, es que contenga todas las acciones que el usuario pueda realizar en la aplicación. Lo más normal y útil es hacerlo clasificando o agrupando las operaciones por categorías en submenús.

José Javier Bermúdez Hernández
([CC BY-NC](#))

En Java usamos los componentes **JMenu** y **JMenuItem** para crear un menú e insertarlo en una barra de menús. La barra de menús es un componente **JMenuBar**. Los constructores son los siguientes:

- ✓ **JMenu()**. Construye un menú sin título.
- ✓ **JMenu(String s)**. Construye un menú con título indicado por s.
- ✓ **JMenuItem()**. Construye una opción sin ícono y sin texto.
- ✓ **JMenuItem(Icon icono)**. Construye una opción con ícono y con texto.



Podemos construir por código un menú sencillo como el de la imagen con las siguientes sentencias:

```
// Crear la barra de menú

JMenuBar barra = new JMenuBar();

// Crear el menú Archivo

JMenu menu = new JMenu("Archivo");

// Crear las opciones del menú

JMenuItem opcionAbrir = new JMenuItem("Abrir");

menu.add(opcionAbrir);

JMenuItem opcionguardar = new JMenuItem("Guardar");

menu.add(opcionguardar);

JMenuItem opcionSalir = new JMenuItem("Salir");

menu.add(opcionSalir);

// Añadir las opciones a la barra

barra.add(menu);

// Establecer la barra

setJMenuBar(barra);
```

Frecuentemente, dispondremos de alguna opción dentro de un menú, que al elegirla, nos dará paso a un conjunto más amplio de opciones posibles.

Cuando en un menú un elemento del mismo es a su vez un menú, se indica con una flechita al final de esa opción, de forma que se sepa que, al seleccionarla, nos abrirá un nuevo menú.

Para incluir un menú como submenú de otro basta con incluir como ítem del menú, un objeto que también sea un menú, es decir, incluir dentro del **JMenu** un elemento de tipo **JMenu**.

Autoevaluación

Un menú se crea utilizando el componente **JMenu** dentro de un **JList**.

Verdadero Falso

Falso

Para crear un menú no nos hace falta ningún **JList**.

Para saber más

En el siguiente enlace tienes un ejemplo de trabajo con menús en Java.

[Menús en Java](#)

3.10.12.- Separadores.

A veces, en un menú pueden aparecer distintas opciones. Por ello, nos puede interesar destacar un determinado grupo de opciones o elementos del menú como relacionados entre sí por referirse a un mismo tema, o simplemente para separarlos de otros que no tienen ninguna relación con ellos.

El **componente Swing que tenemos en Java para esta funcionalidad es: JSeparator**, que dibuja una línea horizontal en el menú, y así separa visualmente en dos partes a los componentes de ese menú. En la imagen podemos ver cómo se han separado las opciones de Abrir y Guardar de la opción de Salir, mediante este componente.

Al código anterior, tan sólo habría que añadirle una línea, la que resaltamos en negrita:

...

```
menu.add(opcionguardar);  
  
menu.add(new JSeparator());  
  
JMenuItem opcionSalir = new JMenuItem("Salir");  
  
...
```



José Javier Bermúdez Hernández
(CC BY-NC)

Autoevaluación

En un menú en Java se debe introducir siempre un separador para que se pueda compilar el código.

Verdadero Falso

Falso

Para crear un menú no es necesario que haya separador forzosamente.

3.4.- Catálogo de ejemplo de componentes Swing

En el siguiente enlace tienes a tu disposición decenas de ejemplos de componentes Swing elaborados por Oracle. Para cada ejemplo contiene un proyecto Netbeans, los ficheros fuentes Java o incluso pueden ser lanzados de forma remota para ver su aspecto.

[Ejemplos de componentes Swing](#)

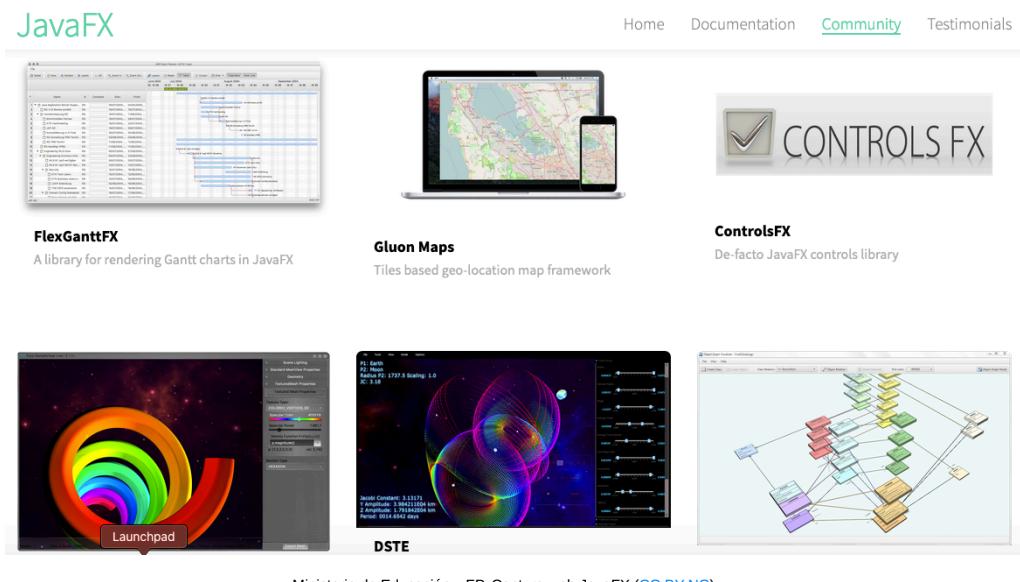
4.- JavaFX

JavaFX es un conjunto de paquetes de gráficos y medios que permite a los desarrolladores diseñar, crear, probar, depurar e implementar aplicaciones de cliente enriquecido que ejecutan y se visualizan perfectamente en diversas plataformas.

Con JavaFX se pueden crear muchos tipos de aplicaciones. Por lo general, son aplicaciones que cumplen con los requisitos de ejecución en múltiples plataformas y muestran información en una interfaz de usuario moderna de alto rendimiento que incluye audio, vídeo, gráficos y animación. Esta tecnología aumenta la productividad y mejora la mantenibilidad de nuestras interfaces de usuario, manteniendo de forma separada los ficheros que implementan la lógica de negocio de los que contienen el diseño de las interfaces. En este sentido, JavaFX supera con creces las posibilidades con respecto Swing, donde crear una simple animación o dar un aspecto semiprofesional es una tarea tediosa y no demasiado simple.

En el siguiente enlace tienes acceso a la web del proyecto JavaFX.

[JavaFX](#)



4.1.- Instalación del JavaFX SDK.

Como se comentó anteriormente, desde el JDK 11, JavaFX no está incluido en el mismo. JavaFX se convirtió en un módulo separado e independiente del JDK 11, lo que no quiere decir que no tenga soporte, de hecho incluye numerosas mejoras desde versiones anteriores.

Antes de trabajar con JavaFX debemos instalar el JavaFX SDK en nuestro sistema operativo. Observa en la siguiente presentación los pasos a seguir para la descarga e instalación:

Instalación de JavaFx SDK

El primer paso será descargar el paquete apropiado según nuestro sistema operativo. Existen dos paquetes diferentes: versión no modular y versión modular del proyecto. En nuestro caso utilizaremos la versión no modular. Además podemos descargar la versión 11 LTS o la última versión estable que es la 14.

[Descarga de JavaFX SDK](#)

Por lo tanto, si utilizamos Windows, puedes descargar el paquete que se observa en la imagen:

JavaFX 11 is the first long term support release of JavaFX by Gluon. For commercial, long term support of JavaFX 11, please review our [JavaFX Long Term Support options](#).

The JavaFX 11 runtime is available as a platform-specific SDK, as a number of jmods, and as a set of artifacts in maven central.

The OpenJFX page at openjfx.io is a great starting place to learn more about JavaFX 11.

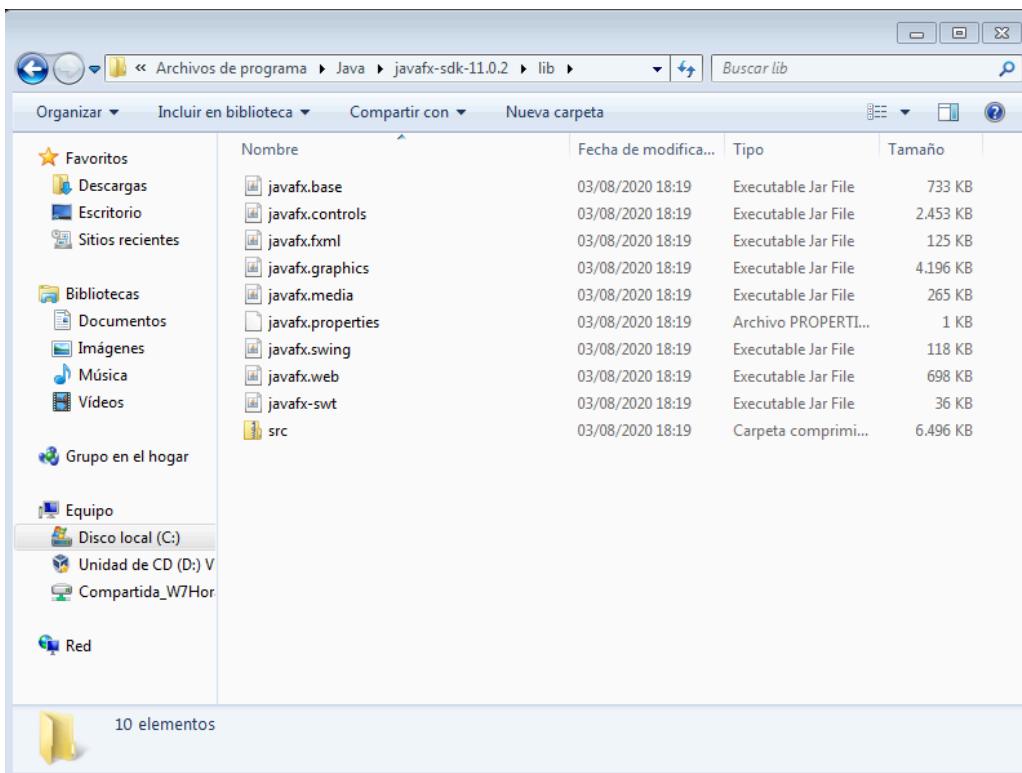
The Release Notes for JavaFX 11 are available in the OpenJFX GitHub repository: [Release Notes](#).

This software is licensed under GPL v2 + Classpath (see <http://openjdk.java.net/legal/gplv2+ce.html>).

Product	Public version	LTS version	Platform	Download
JavaFX Windows SDK	11.0.2	11.0.8 More info	Windows	Download [SHA256]
JavaFX Windows jmods	11.0.2	11.0.8 More info	Windows	Download [SHA256]

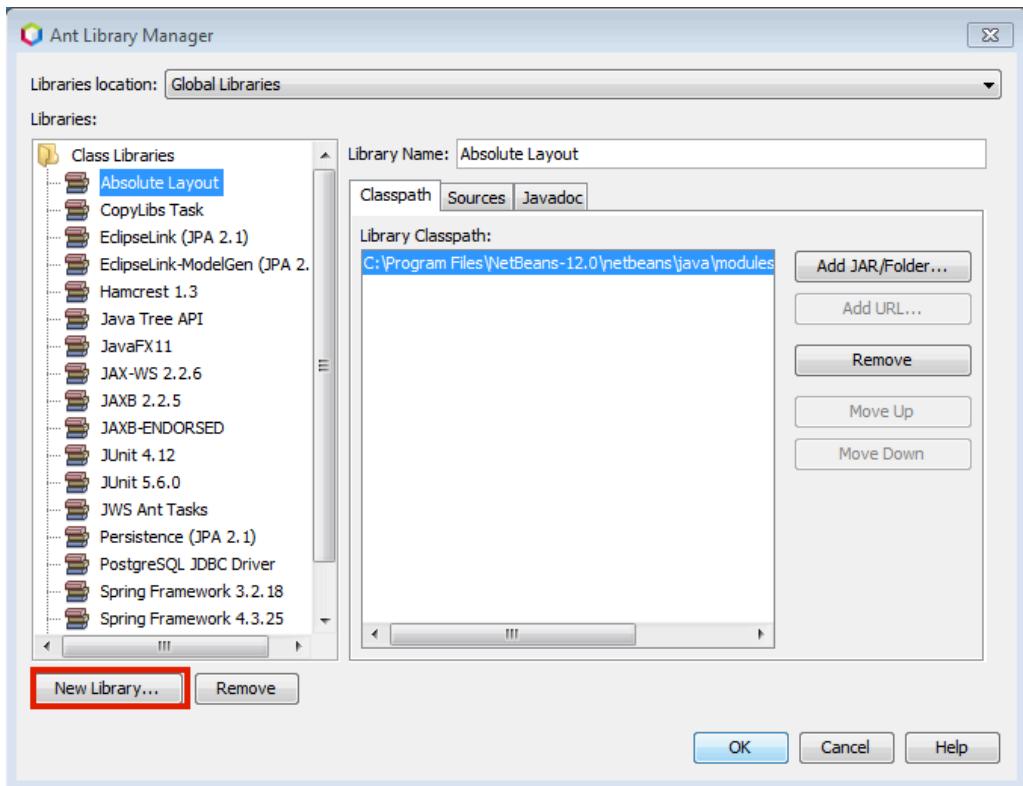
Instalación de JavaFx SDK II

No será necesible ejecutar ningún archivo, tan solo descomprimir el paquete zip descargado en la ruta que consideres oportuna. Un lugar apropiado puede ser la carpeta de instalación de Java.



Instalación de JavaFx SDK III

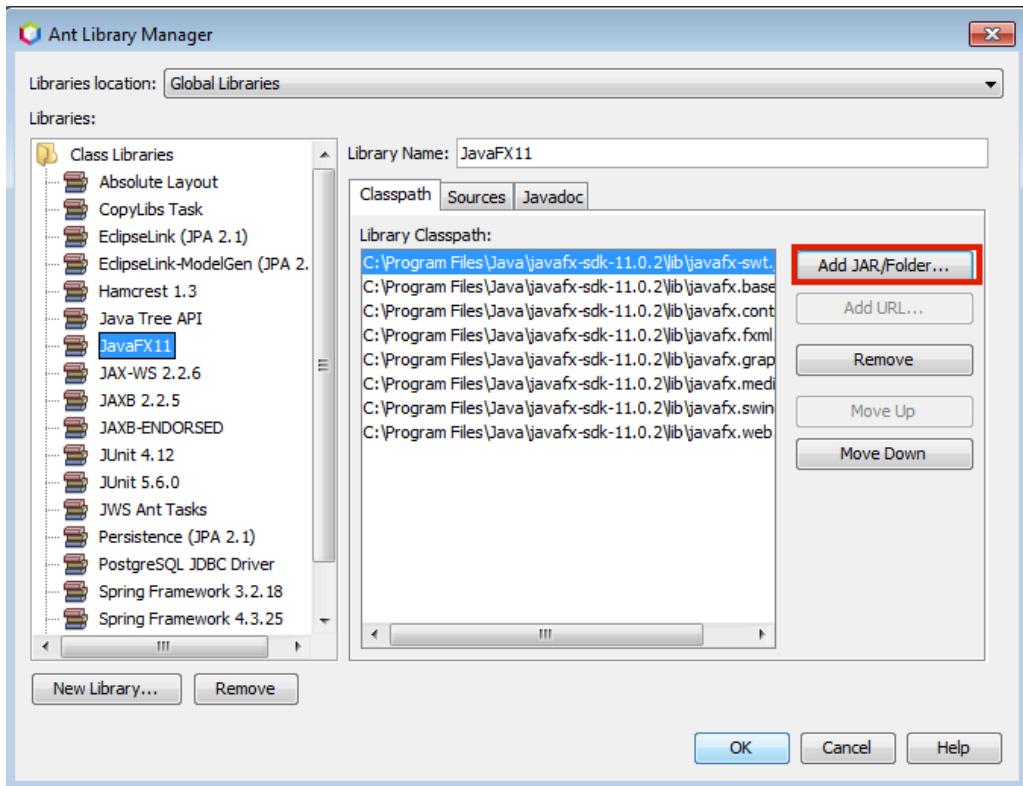
Aunque ya estamos en disposición de utilizar el SDK de JavaFS, tal y como hicimos al principio de curso con el SDK de Java, en esta ocasión vamos a utilizarlo directamente desde Netbeans. Para ello, accedemos al menú de Netbeans **Tools - Library**. Vamos a crear una nueva librería que contenga los ficheros del JavaFX SDK. Pulsamos en el botón **New Library**: tan solo tendremos que asignarle un nombre, por ejemplo, **JavaFX11**.



Instalación de JavaFX SDK IV

Por último, vamos a añadir a la librería creada los ficheros .jar contenidos en la carpeta **lib** de la carpeta de instalación del jdk.

IMPORTANTE: No incluir el fichero **src.zip** que también está contenido en esa carpeta, puede generar errores en la ejecución.



Instalación de JavaFX SDK V

[Resumen textual alternativo](#)

Si utilizas otro sistema operativo (Linux o Mac), el proceso de instalación es exactamente el mismo.

YA TENEMOS NUESTRA LIBRERÍA DE JAVAFX DISPONIBLE EN NETBEANS.

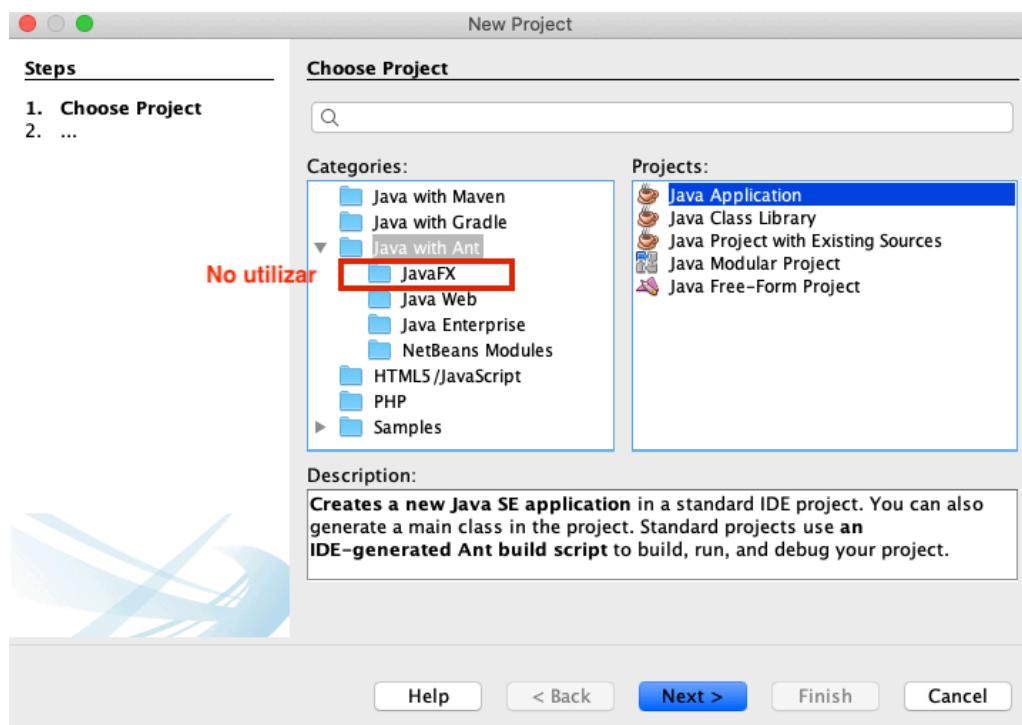
4.2.- Primera aplicación JavaFX.

Una vez instalado el JavaFX SDK en Netbeans, vamos a crear un nuevo proyecto que haga uso del mismo: HolamundoFX.

Primera aplicación JavaFX en Netbeans

El primer paso será una aplicación tradicional Java. Asignale el nombre HolamundoFX.

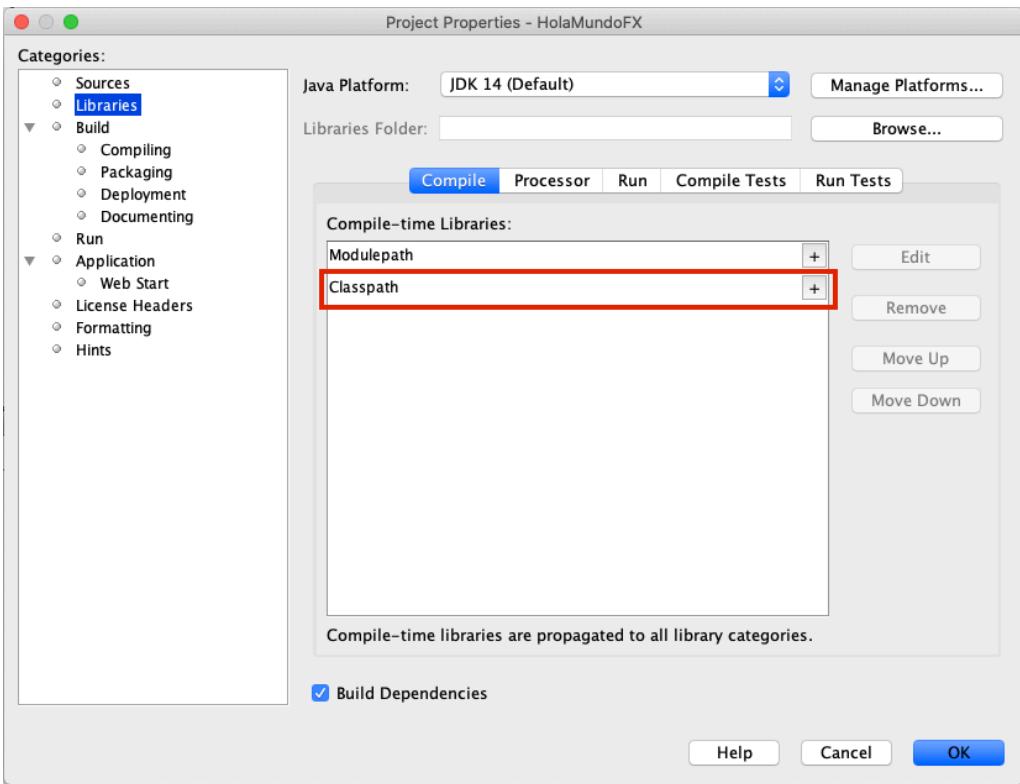
No intentes crear un proyecto FX directamente. Esa tarea aún no está implementada en ANT para JavaFX 11 y posteriores.



Primera aplicación JavaFX en Netbeans II

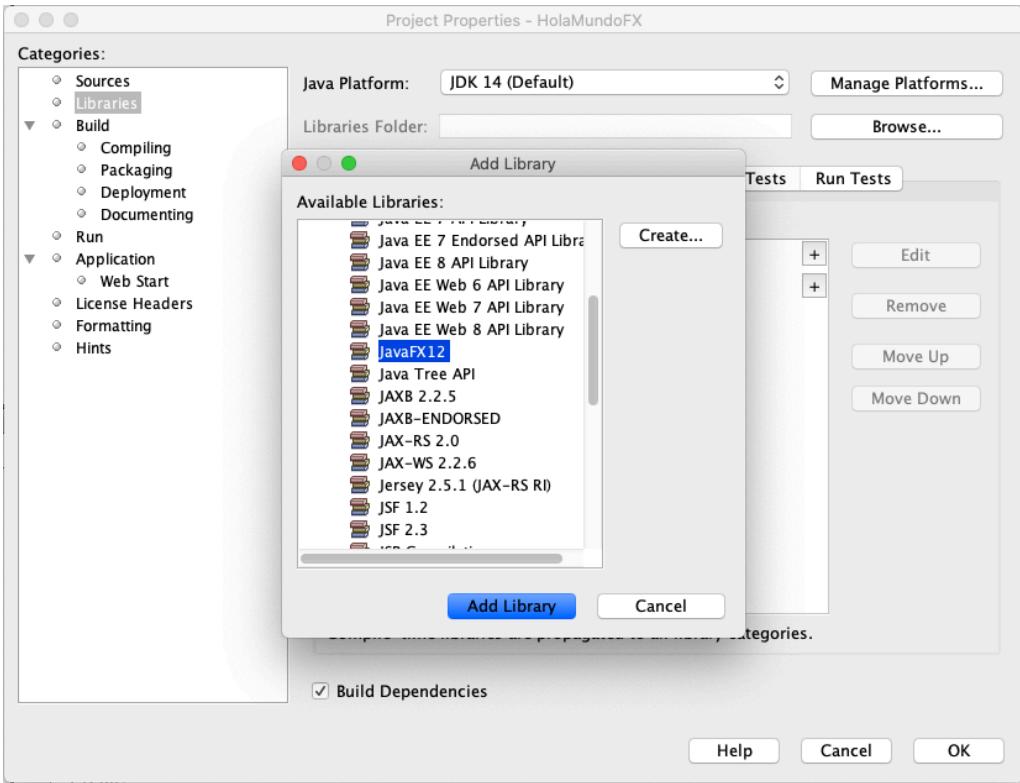
El segundo paso será dar soporte JavaFX a nuestro proyecto. Para ello:

1. Hacemos click con el botón derecho sobre el proyecto creado y seleccionamos la opción **Properties** (Propiedades). Se abrirá una ventana donde vamos a seleccionar **Libraries** en el panel izquierdo.
2. Añadimos una librería al **classpath** a través del botón + que se muestra en la imagen. Debemos seleccionar **Add Library**.



Primera aplicación JavaFX en Netbeans III

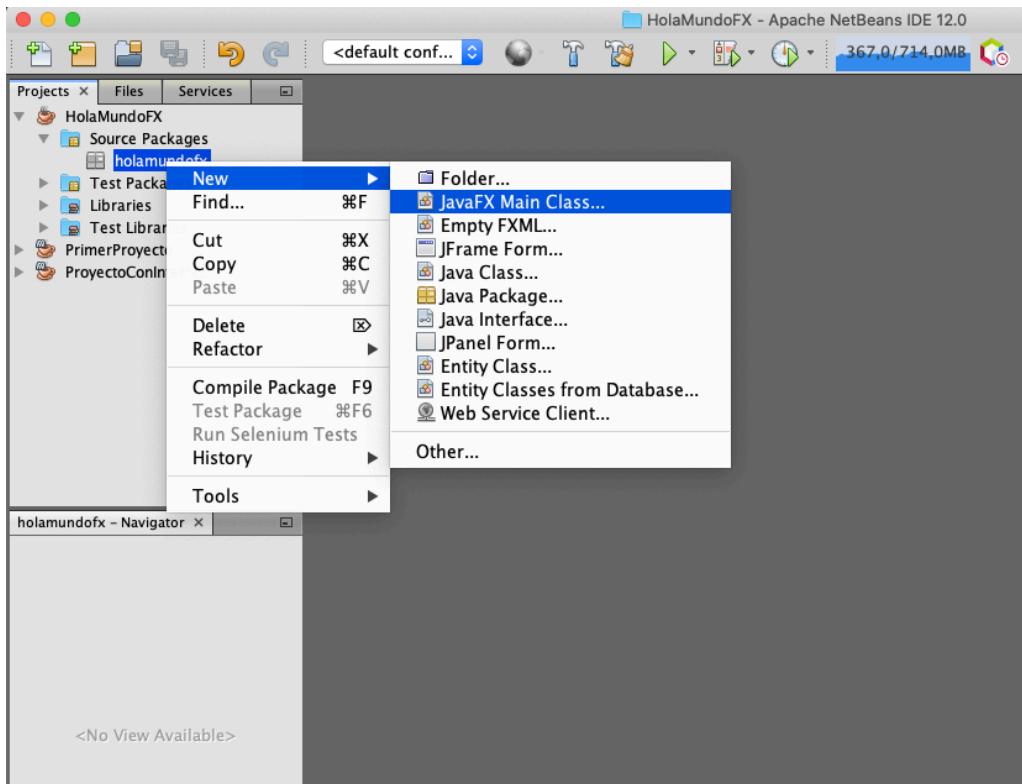
En la ventana que aparece podemos añadir al classpath cualquier librería que no esté soportada actualmente. En nuestro caso será la biblioteca JavaFX creada en el punto 4.1. Obsérvalo.



Primera aplicación JavaFX en Netbeans IV

Nuestra aplicación ya tiene soporte para JavaFX. Vamos a crear una clase principal en el paquete por defecto que implementa la ventana principal de nuestra aplicación. Tan solo tendrás que darle un nombre. La ventana

contendrá un botón que al pulsar mostrará "Hello World" en la consola. Observa la opción a elegir en la siguiente imagen:

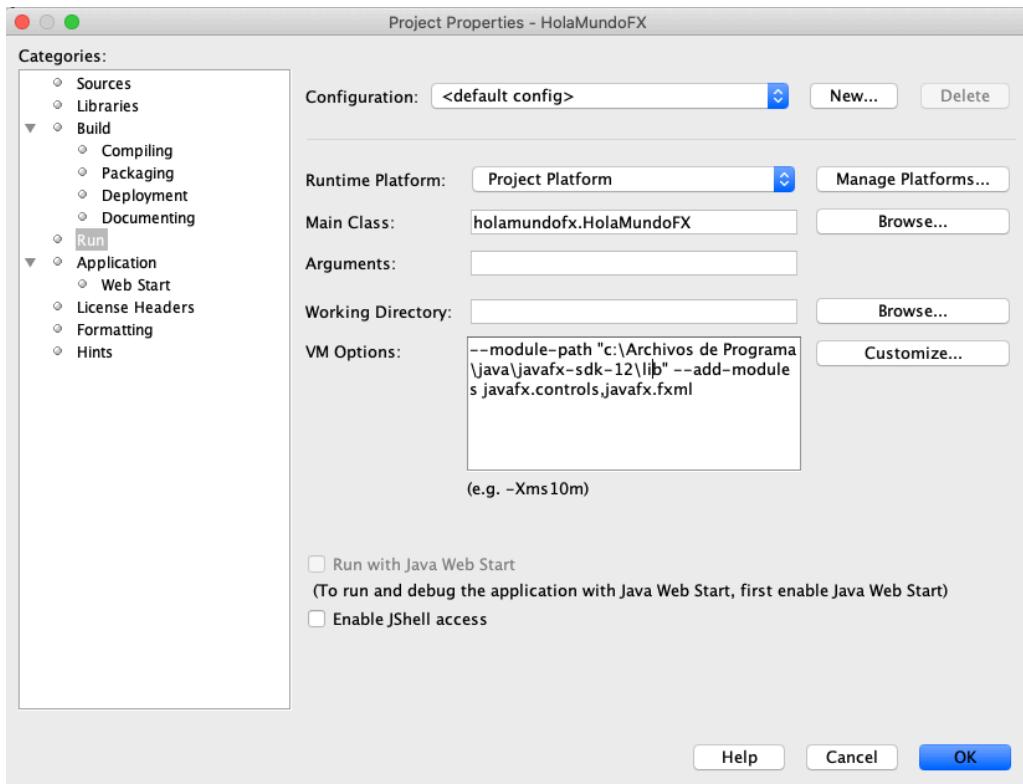


Primera aplicación JavaFX en Netbeans V

Podrás observar que ya tienes la nueva clase creada. Llega la hora de ejecutar la aplicación. Sin embargo, si ejecutamos la aplicación como un aplicación tradicional obtendremos errores.

Debemos lanzar la aplicación parametrizando la jvm, indicándole las librerías de JavaFX que debe utilizar. ¿Porqué?. Si te fijas, la clase principal hereda de `Application`, por lo tanto necesita el soporte de JavaFx para lanzarla a ejecución.

Para ello, accedemos de nuevo a las propiedades de nuestro proyecto y seleccionamos la opción run del panel izquierdo. Observa en la imagen los valores que le indicamos a la JVM.

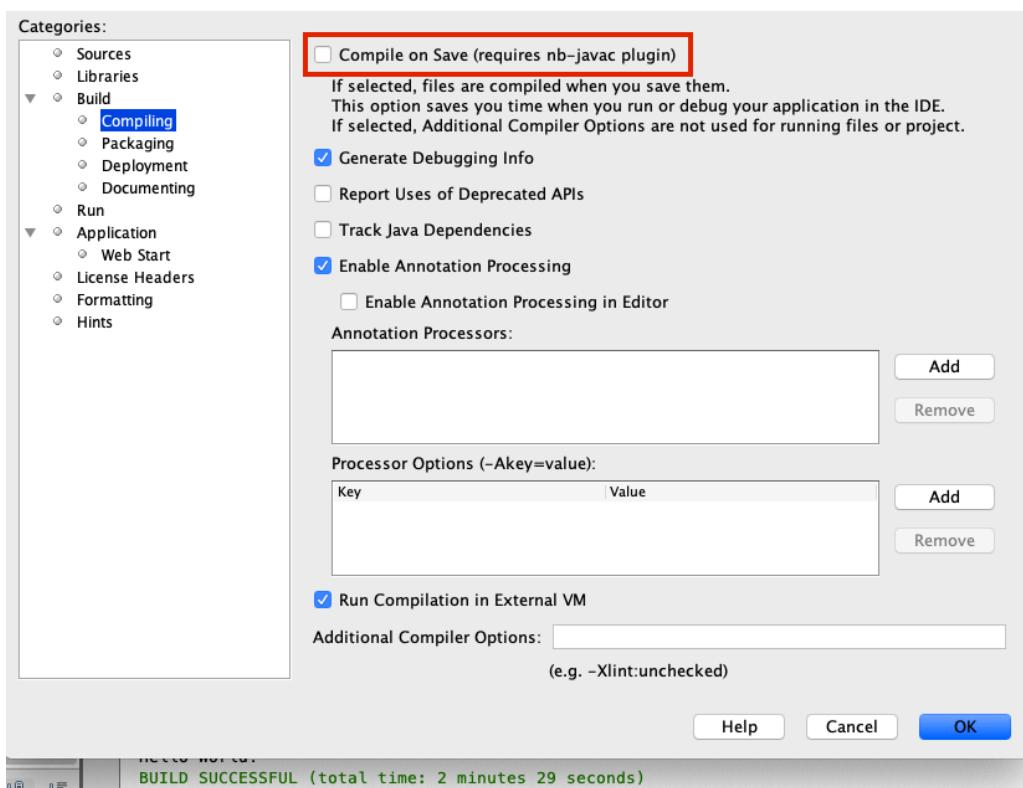


El valor pasado es (en Windows):

```
--module-path "\path\to\javafx-sdk-12\lib" --add-modules javafx.controls,javafx.fxml
```

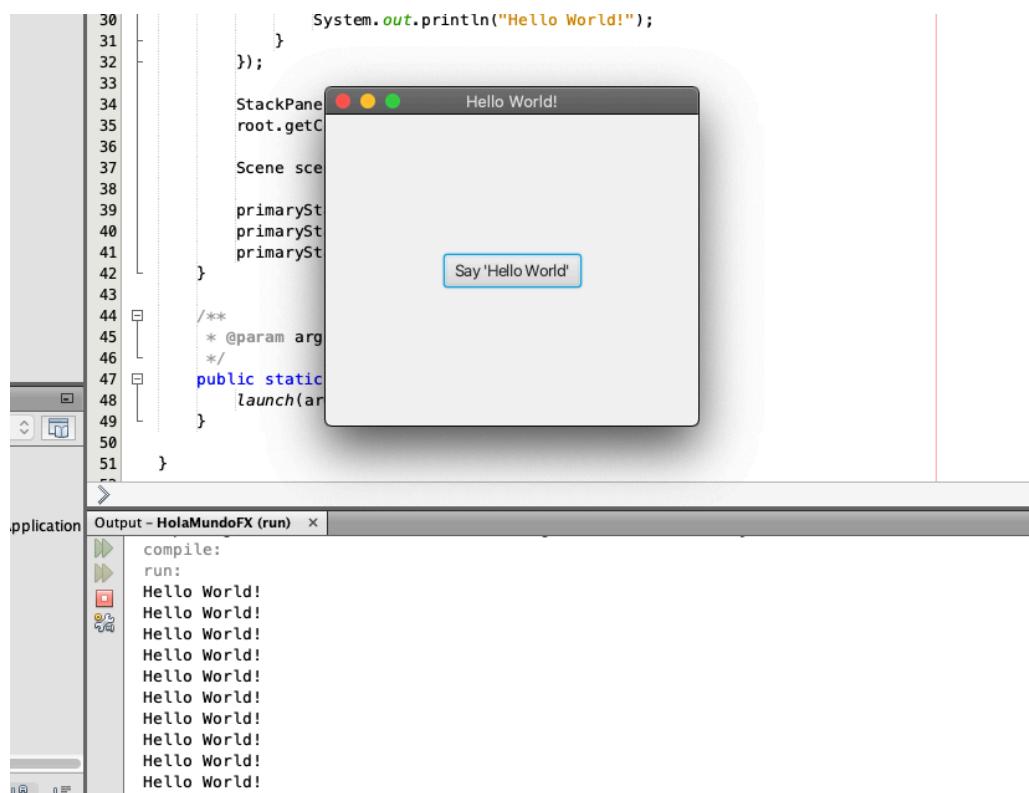
Primera aplicación JavaFX en Netbeans VI

Si al lanzar la ejecución te encuentras con un error en la consola de Netbeans, deselecciona la opción indicada en la imagen en las propiedades del proyecto. Eso solucionará el problema.



Primera aplicación JavaFX en Netbeans VII

Ya podemos lanzar la ejecución de nuestra primera aplicación JavaFX.



Primera aplicación JavaFX en Netbeans VIII

Todas las imágenes utilizadas son propiedad del Ministerio de Educación y FP bajo licencia CC BY-NC y se corresponden con capturas de pantalla de la aplicación Netbeans.

[Resumen textual alternativo](#)

4.3.- Estructura de una aplicación JavaFX.

Si editamos el fichero `HolaMundoFX.java` creado en el ejemplo anterior podremos observar el código que Netbeans ha generado automáticamente. Vamos a analizar dicho código para entender la estructura de una aplicación JavaFX.

```
public class HolaMundoFX extends Application {  
  
    @Override  
    public void start(Stage primaryStage) {  
        Button btn = new Button();  
        btn.setText("Say 'Hello World'");  
        btn.setOnAction(new EventHandler<ActionEvent>() {  
  
            @Override  
            public void handle(ActionEvent event) {  
                System.out.println("Hello World!");  
            }  
        });  
  
        StackPane root = new StackPane();  
        root.getChildren().add(btn);  
  
        Scene scene = new Scene(root, 300, 250);  
  
        primaryStage.setTitle("Hello World!");  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
}
```

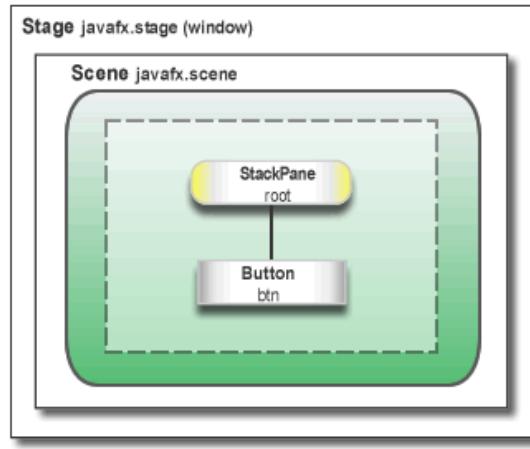
```

}

/**
 * @param args the command line arguments
 */
public static void main(String[] args) {
    launch(args);
}

```

1. La clase creada herera de `javax.application.Application`. El método `start()` es el punto de entrada a cualquier aplicación JavaFX.
2. La aplicación JavaFX define un contenedor para la aplicación que contiene un `stage` (recibido por parámetro en el método `start()`) y UN `scene`.
 - o La clase `Stage` representa en JavaFX un contenedor de máximo nivel.
 - o La clase `Scene` representa un contenedor donde incluimos todos los demás elementos de la ventana (botones, etiquetas, gráficos, etc).
3. En las líneas del 20 a 23 se crea una escena con un determinado tamaño. Se añade un título al `stage`, se le añade la escena creada y se hace visible.
4. En JavaFX, el contenido de una escena es representado como una jerarquía de nodos gráficos. En el ejemplo, el nodo raíz es un objeto de tipo `StackPane`, el cual es un nodo contenedor redimensionable. Es decir, que modificará su tamaño automáticamente si es redimensionado el `stage`.
5. El nodo raíz contiene un nodo hijo, un botón con texto, con un manejador de eventos que imprime un mensaje cuando el botón es pulsado.
6. El método `main` no es necesario para lanzar aplicaciones JavaFX. Sin embargo, es aconsejable utilizarlo e invocar desde él el método `launch()`. Eso permite por ejemplo, que aplicaciones Swing (necesitan el método `main`), pueden embeber código Java FX.



Oracle (Todos los derechos reservados)

4.4.- Instalación del Scene Builder.

Scene Builder es una herramienta que trabaja con JavaFX y permite el diseño de interfaces gráficas de usuario de forma gráfica utilizando un potente entorno, evitando la tediosa y no fácil tarea de construir interfaces a través de código. El editor gráfico utilizado por **Scene Builder** utiliza el principio WYSIWYG (lo que ves es lo que obtienes). Además, para mayor productividad en el diseño y mejor mantenibilidad de las aplicaciones, Scene Builder separa los ficheros que contienen la lógica de aplicación de los que contienen el diseño de las interfaces. Scene Builder es gratis y libre, aunque tiene soporte comercial.

La instalación de Scene Builder y su integración en Netbeans es un proceso sencillo. Obsérvalo en la siguiente presentación:

Instalación de Scene Builder

Descarga el fichero de instalación de Scene Builder teniendo en cuenta la versión del sistema operativo que estás utilizando.

[Descarga de Scene Builder](#)

 GLUON

- Products ▾
- Developers
- Pricing
- Services
- Insights ▾
- Contact ▾



Integrated

Scene Builder works with the JavaFX ecosystem – official controls, community projects, and Gluon offerings including [Gluon Mobile](#), [Gluon Desktop](#), and [Gluon CloudLink](#).



Simple

Drag & Drop user interface design allows for rapid iteration. Separation of design and logic files allows for team members to quickly and easily focus on their specific layer of application development.



Supported

Scene Builder is free and open source, but is backed by Gluon. [Commercial support offerings](#) are available, including [training](#) and [custom consultancy services](#).

Download Scene Builder for Java 11

The latest version of Scene Builder for Java 11 is **11.0.0**.

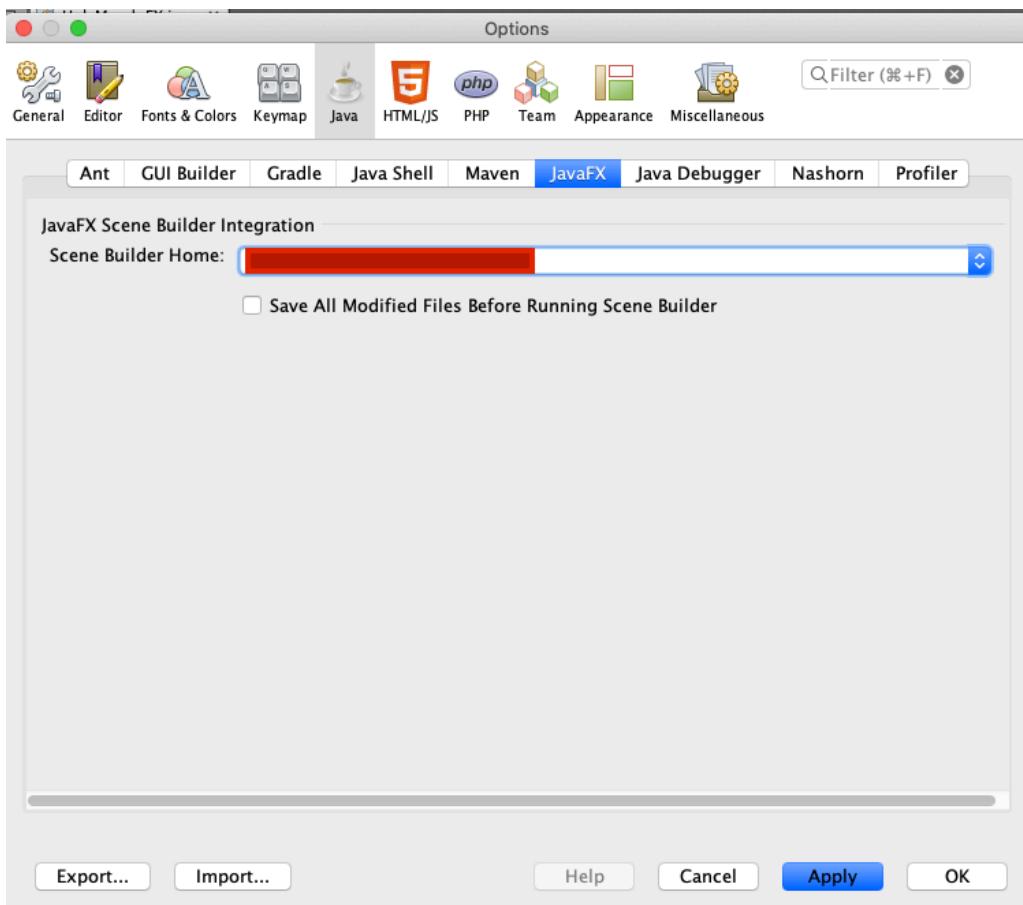
To be kept informed of Scene Builder releases, consider subscribing to the [Gluon Newsletter](#).

Product	Platform	Download
Scene Builder	Windows Installer	Download
Scene Builder	Mac OS X dmg	Download
Scene Builder	Linux RPM	Download
Scene Builder	Linux Deb	Download
Scene Builder Kit <small>(info)</small>	Jar File	Download

Una vez descargado, ejecuta el fichero de instalación y sigue los pasos. Es un proceso muy sencillo.

Integración de Scene Builder en Netbeans

Integrar Scene Builder en Netbeans es fácil. Una vez instalado, tan solo tendremos que decirle a Netbeans en qué ruta se encuentra instalado. Para ello, debemos acceder a las preferencias de Netbeans y configurar el parámetro que se observa en la imagen:



¡Listo, ya tienes Scene Builder en Netbeans!

Integración de Scene Builder en Netbeans

Todas las imágenes son propiedad del Ministerio de Educación y FP bajo licencia CC BY-NC.

[Resumen textual alternativo](#)

4.5.- Primera aplicación JavaFX con Scene Builder

Como práctica introductoria del Scene Builder de JavaFX, vamos a desarrollar la minicalculadora que implementamos con la librería Swing. Como ya hemos comentado, JavaFX permite separar los ficheros que contienen la lógica de negocio de las aplicaciones (llamados **controladores**) de los ficheros que contienen la interfaz de usuario (ficheros **.fxml**). Se puede decir que JavaFX implementa el tan conocido patrón **MVC**, ampliamente utilizada en el desarrollo web actual. Todo quedará mas claro a través del ejemplo.

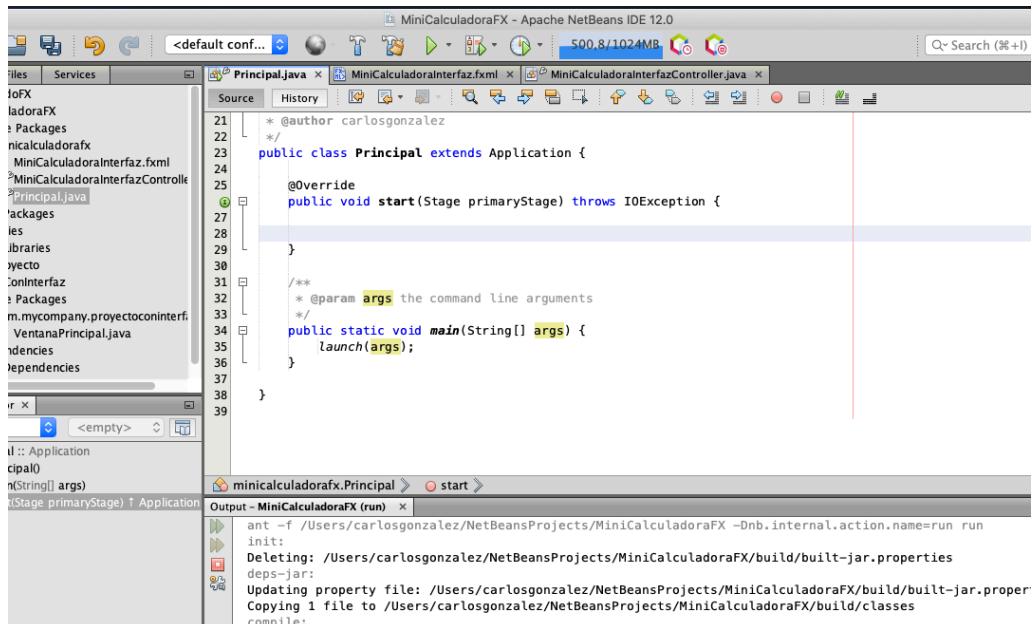
Para ello, observa la siguiente presentación:

Primera aplicación JavaFX con Scene Builder

El primer paso será crear un proyecto con soporte para JavaFX (nómbrale **MiniCalculadoraFX**) tal y como hicimos en el punto 4.2:

- Crearemos una aplicación java tradicional.
- Añadimos la librería de JavaFX.
- Parametrizamos la jvm para poder lanzar a ejecución la aplicación.

El siguiente paso será crear, tal y como hicimos en el proyecto anterior, una clase principal JavaFX. Una vez creada, eliminamos el código del método **start**.

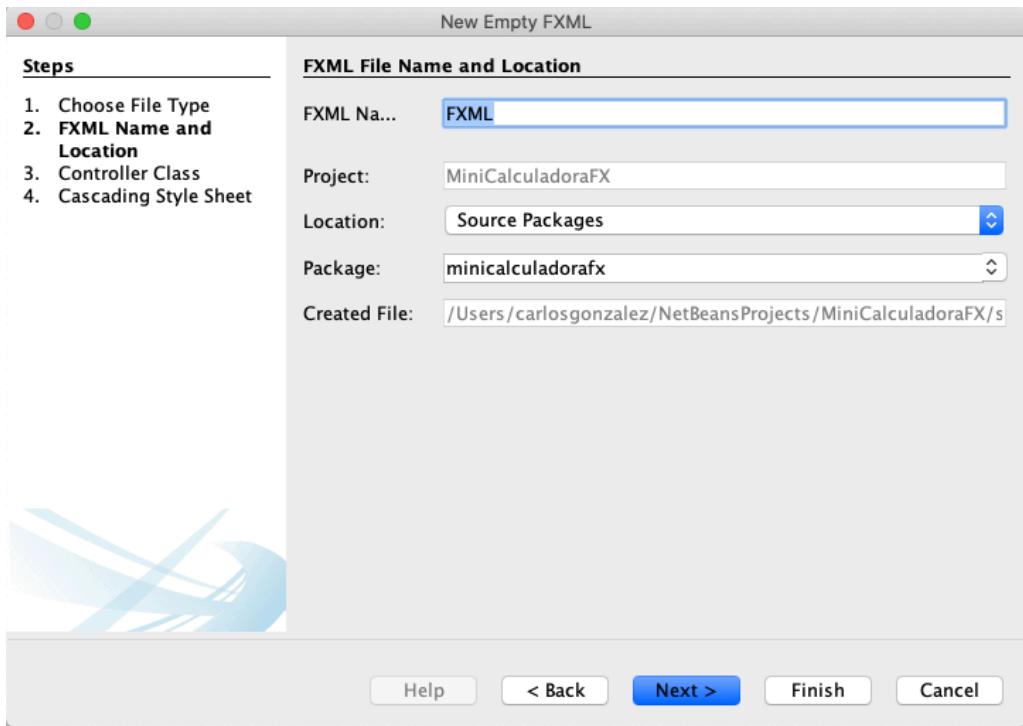


Primera aplicación JavaFX con Scene Builder II

El siguiente paso consistirá en crear dos ficheros que darán soporte a la interfaz de usuario de nuestra aplicación:

1. **MiniCalcVentana.fxml**: Fichero declarativo que contiene la definición de la interfaz de usuario: controles, posición, características, etc. Este fichero no se editará a mano sino a través del Scene Builder.
2. **MiniCalcVentanaControlador.java**: Fichero java que contendrá el código asociado a la ventana, es decir, toda la lógica de negocio. Por ejemplo, los manejadores de eventos de la ventana MiniCalcVentana estarán implementados en este controlador.

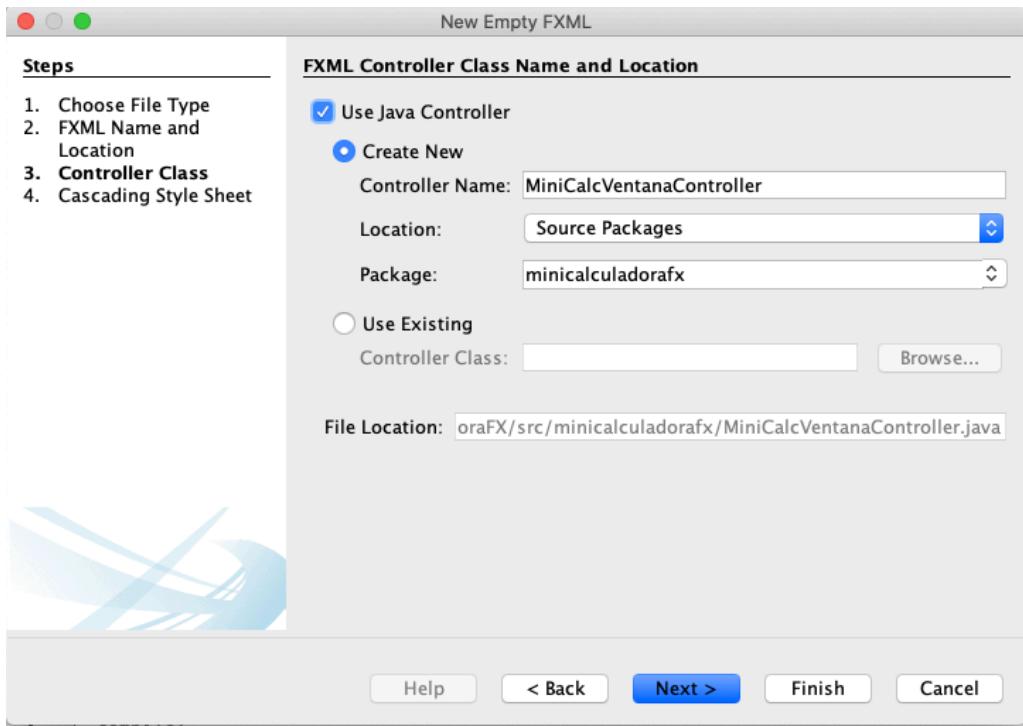
Para ello, en el paquete por defecto, seleccionamos en Netbeans **New - Empty FXML**. Aparecerá la siguiente ventana:



Indicamos el nombre para nuestro fichero FXML: **MiniCalcVentana**. Pulsamos **Next**.

Primera aplicación JavaFX con Scene Builder III

En el siguiente paso, indicamos a Netbeans que genere automáticamente el controlador asociado a la vista.



Dejamos el nombre por defecto y pulsamos **Next**. En el siguiente paso podemos comprobar como incluso podemos asociar un fichero de estilos (CSS) a nuestra vista. Esto no es posible hacerlo en Swing pero si en JavaFX. En nuestro caso no lo hacemos. Pulsamos en **finalizar** y ya tenemos los dos ficheros creados en nuestro paquete.

Primera aplicación JavaFX con Scene Builder IV

Antes de diseñar nuestra interfaz, vamos a incluir el código java necesario en la clase principal de nuestra aplicación para que cuando lancemos a ejecutar, cargue la vista creada **MiniCalcVentana**. Observa el código del método **start**:

```

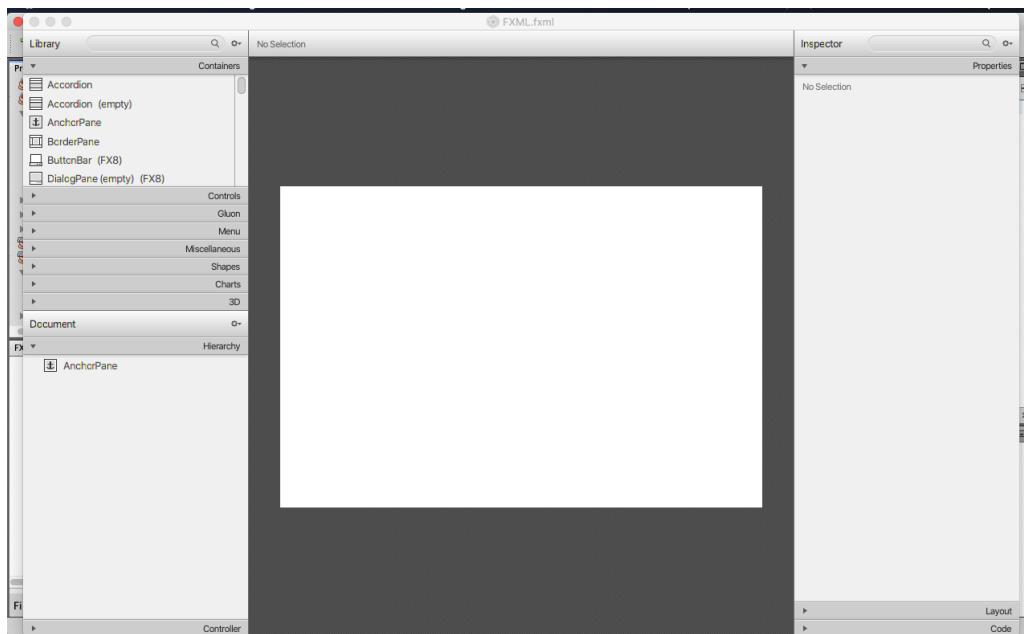
public void start(Stage primaryStage) throws IOException {
    Parent root= FXMLLoader.load(getClass().getResource("MiniCalculadoraInterfaz.fxml"));
    Scene scene= new Scene(root);
    primaryStage.setTitle("Mini Calculadora JavaFX");
    primaryStage.setScene(scene);
    primaryStage.show();
}

```

Parte de este código ya nos es familiar, pues se crea la escena y se asocia al **Stage**. En la primera línea del código, a través del **FXMLLoader**, indicamos que cargue nuestro fichero fxml. En nuestra "primera aplicación JavaFX" creamos la interfaz a través de código Java y en este caso lo haremos a través de nuestro fichero fxml, que editamos con Scene Builder.

Primera aplicación JavaFX con Scene Builder V

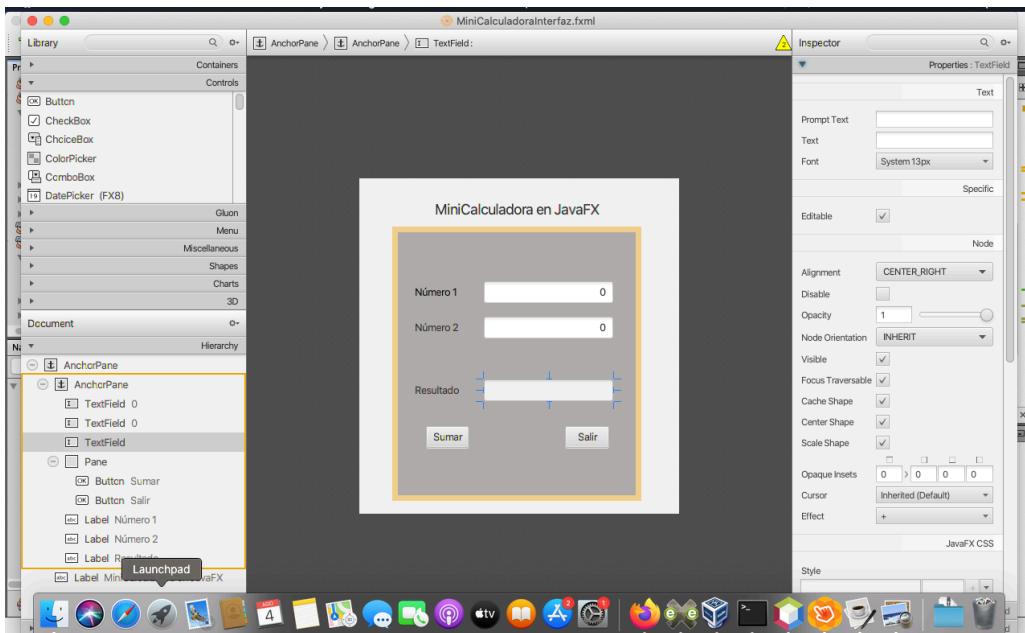
Llega el momento de diseñar la interfaz. Para ello, pulsamos sobre el fichero **FXML** con el botón derecho y pulsamos **Open**: si la instalación del Scene Builder se realizó correctamente, se abrirá y cargará el fichero **MiniCalcVentana.fxml**, que estará vacío.



Primera aplicación JavaFX con Scene Builder VI

La interfaz de Scene Builder es parecida al editor de Swing de Netbeans.

- El panel superior izquierdo contiene todo tipo de controles y layouts.
- Justo debajo se encuentra el navegador, donde se irán colocando de forma jerárquica todos los elementos que añadamos a nuestra interfaz.
- En la parte central tendremos el editor: podemos añadir elementos arrastrando con el ratón.
- El panel de la parte derecha contiene las propiedades de un elemento seleccionado.



Utilizando tus conocimientos e intuición, trata de crear una interfaz parecida a la de la imagen, sin añadirle funcionalidad.

Primera aplicación JavaFX con Scene Builder VII

Una vez que tenemos creada nuestra interfaz, es hora de darle funcionalidad. Recuerda que todo el código Java asociado se incluirá en el controlador **MiniCalcVentanaController.java**. Observa el código que vamos a añadir a nuestro controlador.

```
public class MiniCalculadoraInterfazController implements Initializable {

    @FXML
    private TextField tfNumero1;

    @FXML
    private TextField tfNumero2;

    @FXML
    private TextField tfResultado;

    @FXML
    private void buttonSumarHandler(ActionEvent event){
        //Variables para almacenar los operandos a sumar.
        int num1, num2, resul;

        num1= Integer.parseInt(tfNumero1.getText());
        num2= Integer.parseInt(tfNumero2.getText());

        //Ya hemos recogido el valor de los operandos de los campos de texto de la interfaz. Realizamos la suma.
        resul=num1+num2;

        tfResultado.setText(String.valueOf(resul));
    }

    @FXML
    private void buttonSalirHandler(ActionEvent event){
        System.exit(0);
    }

    @Override
    public void initialize(URL url, ResourceBundle rb) {
        // TODO
    }
}
```

}

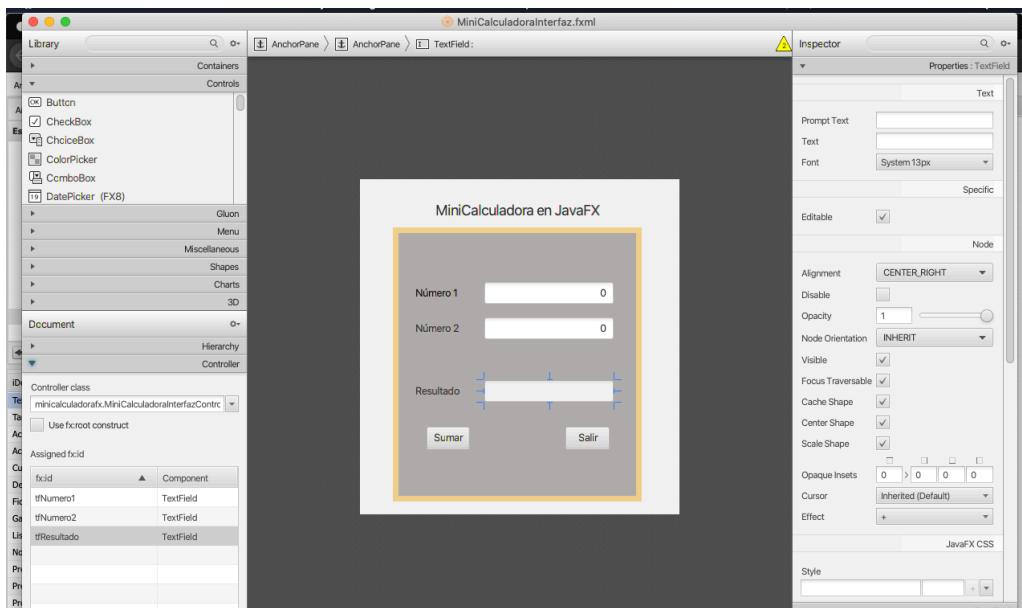
1. Declaramos una referencia a cada uno de los elementos de las interfaz a los que tenemos que acceder desde el controlador. Para ello utilizamos la etiqueta **@FXML** (incluir el import si hay errores de compilación). Evidentemente, hemos de asignar un nombre a cada referencia. En nuestro caso, necesitamos acceder a los tres campos de texto: a dos de ellos para recuperar los valores de los números a sumar (**tfNumero1** y **tfNumero2**) y el tercero para mostrar el resultado (**tfResultado**). Observa que esas referencia son del tipo de objeto que utilizamos al incluir el elemento en la interfaz, en nuestro caso, **TextField**. Líneas 3-10.
2. A continuación, declaramos un método que va a actuar como manejador de eventos del botón **Sumar**. Se encargará de realizar la suma y mostrar el resultado en el campo de texto apropiado. El nombre del método lo decidimos nosotros.
3. Creamos un tercer método que se encargará de manejar eventos producidos por el botón **Salir**.

Primera aplicación JavaFX con Scene Builder VIII

¿Cómo asociamos el controlador al fichero fxml? Esta asociación consiste en:

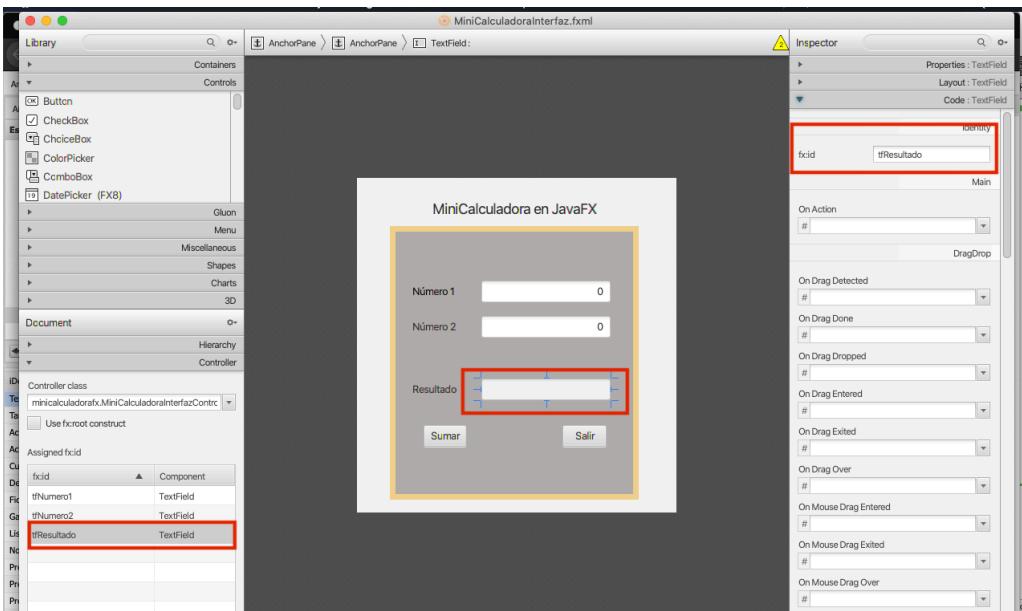
1. Asociar el fichero **MiniCalcVentanaController.java** al fichero **MiniCalcVentana.fxml**.
2. Asociar cada campo de texto en el controlador a su homólogo en el fichero fxml.
3. Asociar el manejador creado a cada uno de los botones en el fichero fxml.

La asociación de ficheros ya está hecha, pues al generar el fichero fxml indicamos a Netbeans que generara su fichero controlador. De todas formas, se podría modificar esa asociación en Scene Builder.



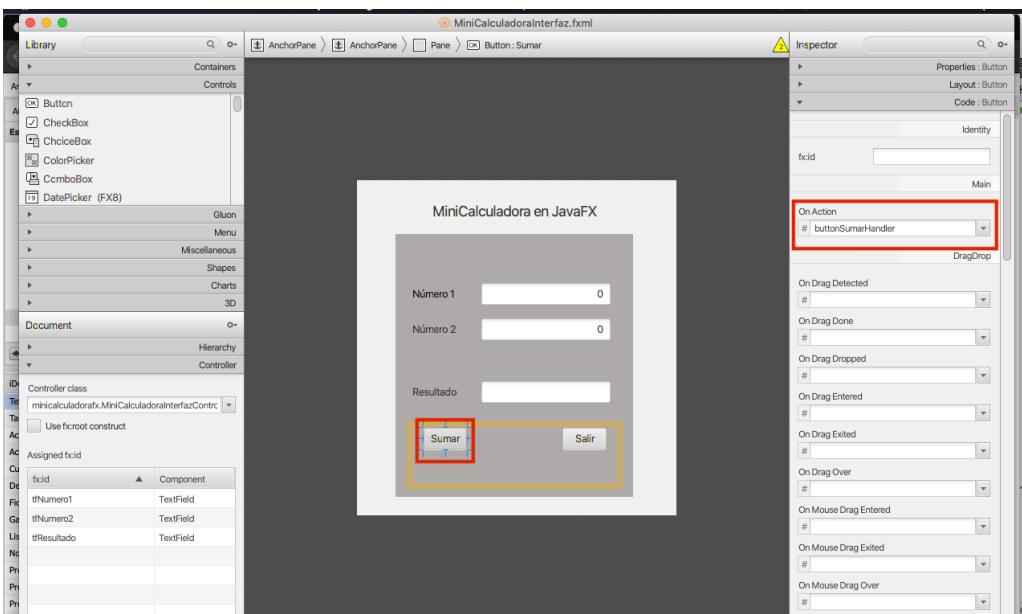
Primera aplicación JavaFX con Scene Builder IX

Para asociar los campos de texto utilizamos el panel **Code** situado en la parte derecha. Observa como en la imagen, para el elemento seleccionado (campo de texto con el resultado), asociamos el id **tfResultado** que previamente hemos definido en el controlador: esta asociación permite que desde el controlador se pueda recuperar el valor de este campo de texto. Habría que lo mismo para los otros dos campos de texto.



Primera aplicación JavaFX con Scene Builder X

Solo nos queda asociar los manejadores de eventos a los botones. Utilizamos el mismo panel. Obsévalo en la imagen:

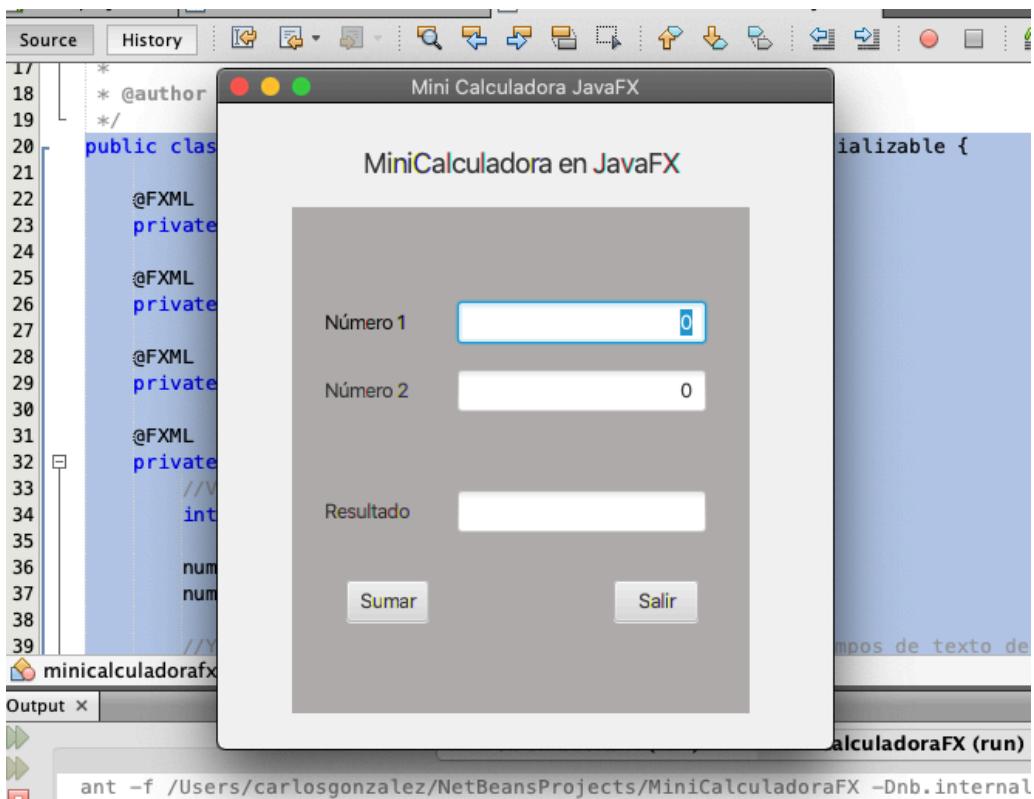


Al evento **OnAction** del botón, que se lanzará cuando el botón sea pulsado, le asociamos el manejador **buttonSumarHandler**, que previamente hemos definido en el controlador.

Puedes asociar el manejador apropiado al botón Salir.

Primera aplicación JavaFX con Scene Builder XI

Momento de ejecutar nuestra aplicación. Si todo es correcto, éste debería ser el resultado.



Primera aplicación JavaFX con Scene Builder X

Todas las imágenes utilizadas son propiedad del Ministerio de Educación y FP bajo licencia CC BY-NC y se corresponden con capturas de pantalla de la aplicación Netbeans.

[Resumen textual alternativo](#)

En el siguiente enlace puedes descargar el proyecto.

[Minicalculadora JavaFX](#)

4.6.- Recursos JavaFX.

Los ejemplos trabajados en la unidad son únicamente un mínimo ejemplo de funcionamiento de JavaFX y de la potente herramienta Scene Builder. Si investigas un poco por la web, podrás comprobar que las posibilidades de estas herramientas para la construcción de interfaces son muchísimas. A continuación tienes algunos recursos que son soporte a lo que hemos trabajado en esta unidad, que solo os sirve de punto de partida para empezar a trabajar con estas tecnologías.

Para saber más

Los siguientes videotutoriales muestran la configuración del entorno Netbeans para trabajar con JavaFX.

<https://www.youtube.com/embed/Bk42DbVBzXM>

[Resumen textual alternativo](#)

El siguiente videotutorial muestra cómo utilizar el Scene Builder con un ejemplo sencillo.

<https://www.youtube.com/embed/iOxu0LAOWk>

[Resumen textual alternativo](#)

El siguiente videotutorial muestra un ejemplo mas completo.

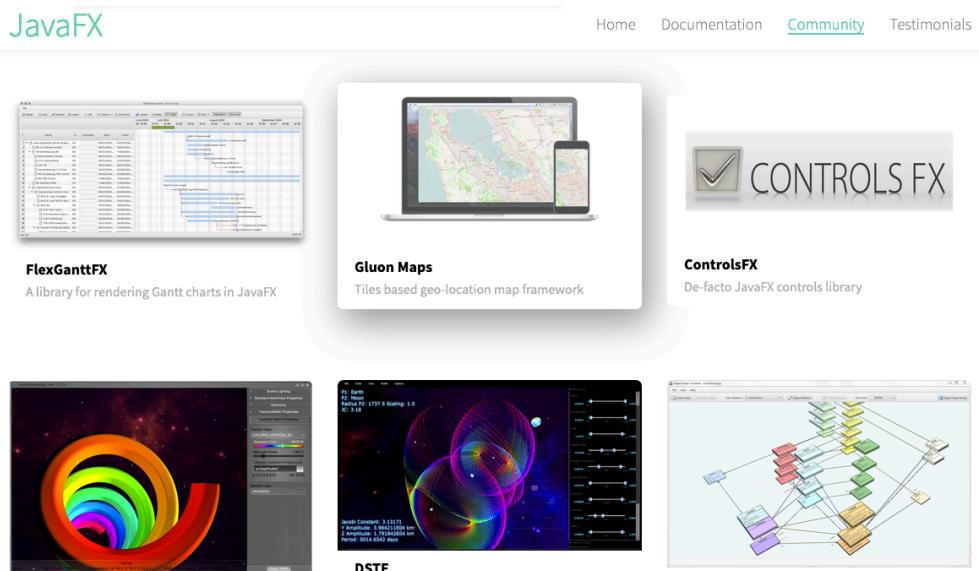
https://www.youtube.com/embed/OT_qBWKiRfY

[Resumen textual alternativo](#)

Recomendación

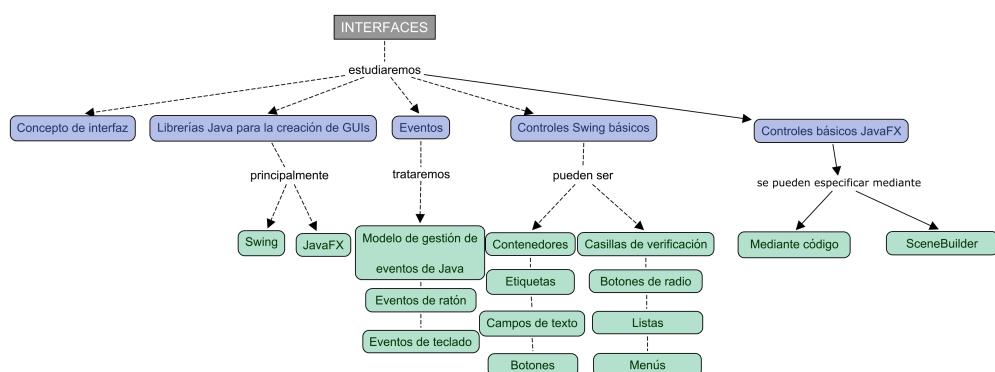
En la web de JavaFX, hay multitud de proyectos JavaFX que pueden ser integrados en nuestros propios proyectos. Puede ser muy interesantes pues aportar funcionalidad que no tendríamos que implementar.

[Web de JavaFX](#)



5.- Conclusiones

A lo largo de esta unidad hemos trabajado con una parte importante de cualquier aplicación software, la interfaz de usuario. Hemos aprendido las dos librerías de Java para el desarrollo de interfaces: Swing, ya en desuso y JavaFX, que permite crear interfaces más modernas que la anterior. Evidentemente aprender en profundidad estas librerías supondría dedicar casi la totalidad de horas de un módulo. El objetivo es hacer una introducción al funcionamiento de estas librerías.



Para terminar el curso, trabajaremos con bases de datos relacionales y haremos una pequeña introducción a las bases de datos orientadas a objeto.