Comunicándonos con el usuario. Interfaces.

Caso práctico

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.

1.- Introducción.

Hoy en día las **interfaces** de los programas son cada vez más sofisticadas y atractivas par 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.

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 el 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 interactuación del usuario con la máquina.

- Ninguna es correcta.

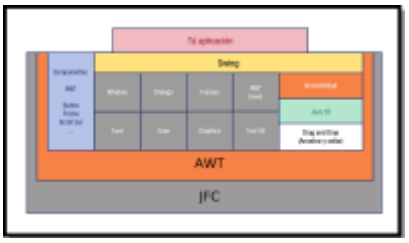
2.- Librerías de Java para desarrollar GUI.

Caso práctico

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.**

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 desee 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.

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.

2.1.- AWT.

Cuando apareció Java, los componentes gráficos disponibles para el desarrollo de GUI se encontraban en la librería denominada Abstract Window Toolkit (**AWT**).

Las clases AWT se desarrollaron usando código nativo (o sea, código asociado a una plataforma concreta). Eso **dificultaba la portabilidad** de las aplicaciones. Al usar código nativo, para poder conservar la portabilidad era necesario restringir la funcionalidad a los mínimos comunes a todas las plataformas donde se pretendía usar AWT. Como consecuencia, AWT es una librería con una funcionalidad muy pobre.

La estructura básica de la librería gira en torno a componentes y contenedores.
Los contenedores contienen componentes y son componentes a su vez, de forma que los eventos pueden tratarse tanto en contenedores como en componentes.

AWT es adecuada para interfaces gráficas sencillas, pero no para proyectos más complejos.
Más tarde, cuando apareció Java 2 surgió una librería más robusta, versátil y flexible: **Swing**. AWT aún sigue estando disponible, de hecho se usa por los componentes de Swing.

Puedes ver la lista de los principales componentes de la clase AWT en el siguiente enlace:

Principales componentes de la clase AWT

Para saber más

Página oficial de Sun – Oracle sobre AWT (en inglés).

AWT (Abstract Window Toolkit)

AWT sigue siendo imprescindible, ya que todos **los componentes Swing se construyen haciendo uso de clases de AWT**. De hecho, como puedes comprobar en el API, todos los componentes Swing, como por ejemplo **`JButton`** (es la clase Swing que usamos para crear cualquier botón de acción en una ventana), derivan de la clase **`JComponent`**, que a su vez deriva de la clase AWT **`Container`**.

Las clases asociadas a cada uno de los componentes AWT se encuentran en el paquete **`java.awt.`**
Las clases relacionadas con el manejo de eventos en AWT están en el paquete **`java.awt.event`**.

AWT fue la primera forma de construir las ventanas en Java, pero:

limitaba la portabilidad,
restringía la funcionalidad y
requería demasiados recursos.

Autoevaluación

AWT está indicado para proyectos muy grandes y de gran complejidad.

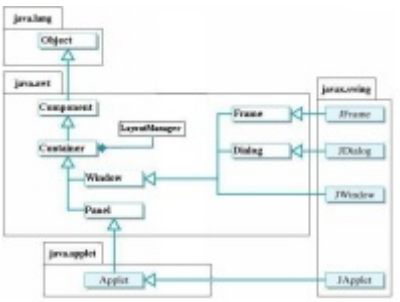
Verdadero. ○ Falso. ○

Para saber más

Decíamos más arriba, que las JFC proporcionan soporte para impresión. En el siguiente enlace tienes más información sobre impresión en Java.

Impresión en Java

2.2.- Swing.



Cuando se vio que era necesario mejorar las características que ofrecía AWT, distintas empresas empezaron a sacar sus controles propios 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.

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.

Debes conocer

A continuación tienes una tabla con la lista de los controles Swing más habituales. Debes tener en cuenta que las imágenes están obtenidas eligiendo el aspecto (LookAndFeel) multiplataforma propio de Java.

Resumen textual alternativo

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.

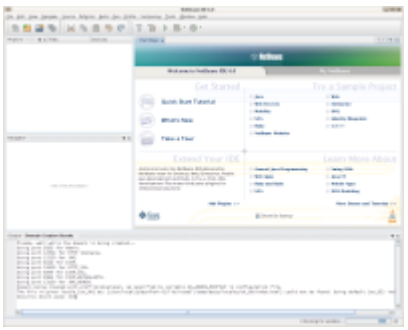
Algunas características más de Swing, podemos mencionar:

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.

3.- 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.

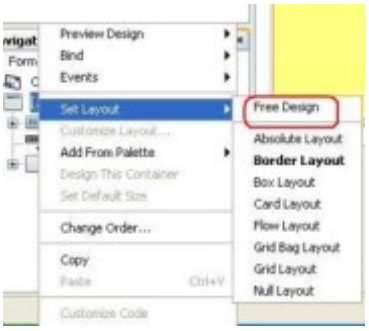
Crear aplicaciones que incluyan interfaces gráficos de usuario, con NetBeans, es muy sencillo debido a las facilidades que nos proporcionan sus asistentes y herramientas.

El IDE de programación nos proporciona un diseñador 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:



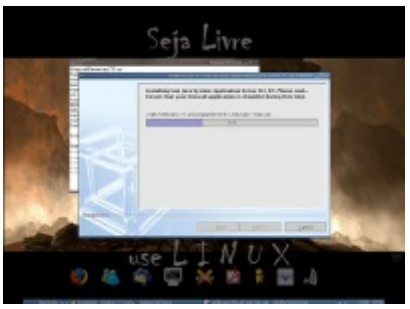
Modo de diseño libre (Free Design): permite mover libremente los componentes de interfaz de usuario sobre el panel o marco, sin atenerse a uno de los layouts por defecto.
Independencia de plataforma: diseñando de la manera más fácil, es decir, arrastrando y soltando componentes desde la paleta al área 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. ○

3.1.- 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:

Resumen textual alternativo

A continuación, crea la interfaz de usuario tal y como ves en:

Resumen textual alternativo

Ahora, añade la funcionalidad como en:

Resumen textual alternativo

Por último, ejecuta el proyecto:

Resumen textual alternativo

Es posible que te estés empezando a dar cuenta de cómo se gestionan los eventos en Java. Vamos a analizar un poco más, en la siguiente presentación, como funciona la gestión de eventos, al hilo del proyecto que acabamos de crear:

Resumen textual alternativo

Debes conocer

En el siguiente enlace puedes ver, paso a paso, la creación de un proyecto, que utiliza interfaz gráfica de usuario, con NetBeans

Proyecto para crear un interfaz gráfico paso a paso (644 KB)

En 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)

Para saber más

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.

Java NetBeans Calculator 1 of 5

Resumen textual alternativo

4.- Eventos.

Caso práctico

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.

4.1.- 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:

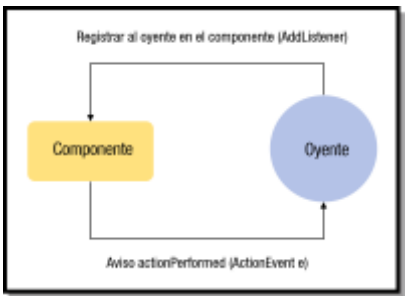
pulsar un botón con el ratón;
hacer doble clic;
pulsar y arrastrar;
pulsar una combinación de teclas en el teclado;
pasar el ratón por encima de un componente;
salir el puntero de ratón de un componente;
abrir una ventana;
etc.

¿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.

4.2.- 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.

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.

Observa un ejemplo muy sencillo para ver estos tres pasos:

Resumen textual alternativo

Explicación del modelo de eventos con un ejemplo sencillo:

Resumen textual alternativo

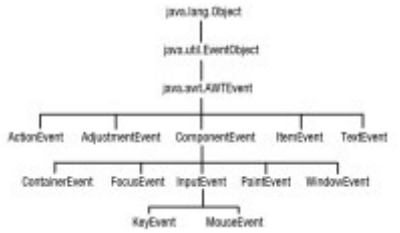
Autoevaluación

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. ○

4.3.- 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`.

Autoevaluación

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

Verdadero. ○ Falso. ○

4.4.- 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.

`KeyListener` (oyente de teclas).

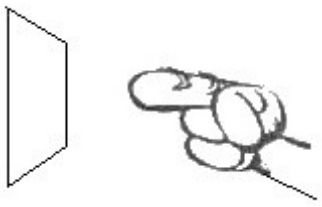
Método	Causa de la invocación
`keyPressed (KeyEvent e)`	Se ha pulsado una tecla.
`keyReleased (KeyEvent e)`	Se ha liberado una tecla.
`keyTyped (KeyEvent e)`	Se ha pulsado (y a veces soltado) una tecla.

`KeyEvent` (evento de teclas)

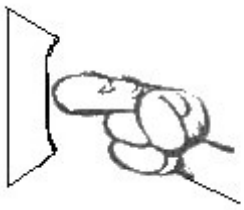
Métodos más usuales	Explicación
`char getKeyChar()`	Devuelve el carácter asociado con la tecla pulsada.
`int getKeyCode()`	Devuelve el valor entero que representa la tecla pulsada.
`String getKeyText()`	Devuelve un texto que representa el código de la tecla.
`Object getSource()`	Método perteneciente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

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

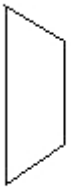
`KeyEventVK_A` especifica la tecla A.
`KeyEventVK_ESCAPE` especifica la tecla ESCAPE.



Botón en estado normal.



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



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

En la siguiente presentación tienes el código del proyecto que te puedes descargar también a continuación. 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.

Resumen textual alternativo

Código Java comentado de un oyente de teclado (17.8 KB)

4.5.- 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.
`mouseClickerd (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 (22.3 KB)

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.

Debes conocer

Tienes un ejemplo de uso de los adaptadores en el siguiente enlace:

Ejemplo de Adaptadores

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

Autoevaluación

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

Verdadero. ○ Falso. ○

4.6.- Creación de controladores de eventos.

A partir del JDK1.4 se introdujo en Java la clase **EventHandler** para soportar oyentes de evento muy sencillos.

La utilidad de estos controladores o manejadores de evento es:

> Crear oyentes de evento sin tener que incluirlos en una clase propia.
> Esto aumente el rendimiento, ya que no "añade" otra clase.

Como inconveniente, destaca la dificultad de construcción: **los errores no se detectan en tiempo de compilación**, sino en tiempo de ejecución.
Por esta razón, es mejor crear controladores de evento con la ayuda de un asistente y documentarlos todo lo posible.

El uso más sencillo de **EventHandler** consiste en instalar un oyente que llama a un método, en el objeto objetivo sin argumentos. En el siguiente ejemplo creamos un **ActionListener** que invoca al método *dibujar* en una instancia de **javax.Swing.JFrame**.

```
miBoton.addActionListener(
        (ActionListener)EventHandler.create(ActionListener.class, frame, "dibujar"));
```

Cuando se pulse miBoton, se ejecutará la sentencia **frame.dibujar()**. Se obtendría el mismo efecto, con mayor seguridad en tiempo de compilación, definiendo una nueva implementación al interface **ActionListener** y añadiendo una instancia de ello al botón:

```
// Código equivalente empleando una clase interna en lugar de EventHandler.
miBoton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
                frame.dibujar();
        }
});
```

Probablemente el uso más típico de **EventHandler** es extraer el valor de una propiedad de la fuente del objeto evento y establecer este valor como el valor de una propiedad del objeto destino. En el siguiente ejemplo se crea un **ActionListener** que establece la propiedad "**label**" del objeto destino al valor de la propiedad "**text**" de la fuente (el valor de la propiedad "**source**") del evento.

```
EventHandler.create(ActionListener.class, miBoton, "label", "source.text")
```

Esto correspondería a la implementación de la siguiente clase interna:

```
// Código equivalente utilizando una clase interna en vez de EventHandler.
new ActionListener {
        public void actionPerformed(ActionEvent e) {
                miBoton.setLabel(((JTextField)e.getSource()).getText());
        }
}
```

Autoevaluación

El uso de `EventHandler` tiene como inconveniente, que los errores no se detectan en tiempo de ejecución.

Verdadero. ○ Falso. ○

5.- Generación de programas en entorno gráfico.

Caso práctico

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 de una idea de cual es su aspecto.

También hemos visto un ejemplo de cómo crear con al 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.

Citas para pensar

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

Albert Einstein

5.1.- 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`.

Cualquier aplicación, con interfaz gráfico 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. ○

5.2.- Cerrar la aplicación.

Cuando quieres terminar la ejecución de un programa, ¿qué sueles hacer? Pues normalmente pinchar en el icono 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. ○

5.3.- 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.

Debes conocer

En la siguiente web puedes ver gráficamente los distintos layouts

LayOuts (en inglés)

LayOuts con código comentado

En la siguiente demostración se muestra el uso de **`FlowLayout`** para posicionar varios botones. Pulsa sobre cada opción para ver una imagen de su comportamiento con las propiedades anteriores:

Resumen textual alternativo

Autoevaluación

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

Verdadero. ○ Falso. ○

5.4.- 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.

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:

```
…
JPanel panelRojo = new JPanel();
panelRojo.setBackground(Color.RED);
panelRojo.setSize(300,300);
// Se crea una ventana
JFrame ventana=new JFrame("Prueba en rojo");
ventana.setLocation(100,100);
ventana.setVisible(true);
// Se coloca el JPanel en el content pane
Container contentPane=ventana.getContentPane();
contentPane.add(panelRojo);
…
```

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.

5.5.- 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.

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 quieres ver las principales propiedades asociadas a **`JTextField`**, y los métodos que permiten modificarlas, puedes utilizar la presentación siguiente:

Resumen textual alternativo

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. ○

5.6.- 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.

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.

5.7.- 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 icono distinto** para el estado de **seleccionado** y el de **no seleccionado**.

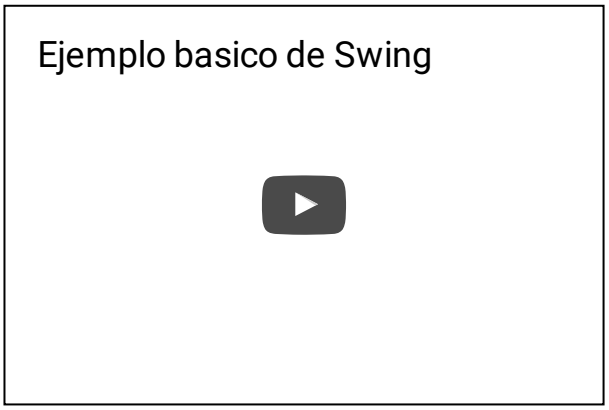
`JCheckBox` pueden usarse en menús mediante la clase **`JCheckBoxMenuItem`**.

`JButtonGroup` 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 **`JButtonGroup`** 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.

Para saber más

En el siguiente enlace puedes ver un vídeotutorial para crear un ejemplo básico de Java con interfaz gráfica.



Resumen textual alternativo

Autoevaluación

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

Verdadero. ○ Falso. ○

5.8.- 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 **JScrollPanel** para que aparezcan barras de desplazamiento.

Podemos ver cómo añadir un **JList** en NetBeans en la siguiente presentación:

Resumen textual alternativo

En los componentes **JList**, un modelo code>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 parar 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 sc
El método getModel() permite recoger ese modelo en forma de Vector de objetos, y utiliza
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 una presentación, que aunque con audio en inglés, es lo suficientemente explicativa para entenderla.

Resumen textual alternativo

5.8.1.- 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.

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 **getSelectecValue()** 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`.

Para saber más

En el siguiente enlace tienes algún ejemplo comentado para crear **JList**.

Ejemplo **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. ○

5.9.- 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.

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`**.

Debes conocer

Puedes ver en el siguiente videotutorial cómo se crea una aplicación con NetBeans, en la que se van añadiendo los controles que hemos visto, entre ellos una lista desplegable.

Resumen textual alternativo

Para saber más

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

Ejemplo **`JComboBox`**

5.10.- 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.

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 icono y sin texto.
`JMenuItem(Icon icono)`. Construye una opción con icono 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**.

Para saber más

En este enlace podrás encontrar un ejemplo de construcción de un menú con varios elementos.

Construcción de menús.

Autoevaluación

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

Verdadero. ○ Falso. ○

5.10.1.- 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");
...
```

Autoevaluación

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

Verdadero. ○ Falso. ○

Para saber más

En este enlace podrás encontrar más información sobre diseño de menús.

Diseño de menús. (3.75 MB)

5.10.2.- Aceleradores de teclado y mnemónicos.

A veces, tenemos que usar una aplicación con interfaz gráfica y no disponemos de ratón, porque se nos ha roto, o cualquier causa.

Además, cuando diseñamos una aplicación, debemos preocuparnos de las características de accesibilidad.

Algunas empresas de desarrollo de software obligan a todos sus empleados a trabajar sin ratón al menos un día al año, para obligarles a tomar conciencia de la importancia de que todas sus aplicaciones deben ser accesibles, usables sin disponer de ratón.

Ya no sólo en menús, sino en cualquier componente interactivo de una aplicación: campo de texto, botón de acción, lista desplegable, etc., es muy recomendable que pueda seleccionarse sin el ratón, haciendo uso exclusivamente del teclado.

Para conseguir que nuestros menús sean accesibles mediante el teclado, la idea es usar aceleradores de teclado o **atajos de teclado** y de **mnemónicos**.

Un acelerador o **atajo de teclado** es una combinación de teclas que se asocia a una opción del menú, de forma que pulsándola se consigue el mismo efecto que abriendo el menú y seleccionando esa opción.

Esa combinación de teclas **aparece escrita a la derecha de la opción del menú**, de forma que el usuario pueda tener conocimiento de su existencia.

Para añadir un atajo de teclado, lo que se hace es emplear la propiedad **accelerator** en el diseñador

Los atajos de teclado **no suelen** asignarse a todas las opciones del menú, sino **sólo a** las que más se usan.

Un **mnemónico** consiste en resaltar una tecla dentro de esa opción, mediante un subrayado, de forma que pulsando Alt + <el mnemónico> se abra el menú correspondiente. Por ejemplo en la imagen con Alt + A abriríamos ese menú que se ve, y ahora con Ctrl+G se guardaría el documento.

Para añadir mediante código un mnemónico a una opción del menú, se hace mediante la **propiedad mnemonic**. Los mnemónicos sí que deberían ser incluidos para todas las opciones del menú, de forma que todas puedan ser elegidas haciendo uso sólo del teclado, mejorando así la accesibilidad de nuestra aplicación. Para ver cómo se añadiría mediante el diseñador de NetBeans, mira el siguiente Debes conocer.

Debes conocer

En el siguiente enlace se puede ver cómo añadir a una aplicación que estemos construyendo con NetBeans, entre otras cosas, mnemónicos y aceleradores.

Diseñando con NetBeans

Anexo I.- Principales clases AWT.

Principales clases AWT

NOMBRE DE LA CLASE AWT	UTILIDAD DEL COMPONENTE
`Applet`	Ventana para una applet que se incluye en una página web.
`Button`	Crea un botón de acción.
`Canvas`	Crea un área de trabajo en la que se puede dibujar. Es el único componente AWT que no tiene un equivalente Swing.
`Checkbox`	Crea una casilla de verificación.
`Label`	Crea una etiqueta.
`Menu`	Crea un menú.
`ComboBox`	Crea una lista desplegable.
`List`	Crea un cuadro de lista.
`Frame`	Crea un marco para las ventanas de aplicación.
`Dialog`	Crea un cuadro de diálogo.
`Panel`	Crea un área de trabajo que puede contener otros controles o componentes.
`PopupMenu`	Crea un menú emergente.
`RadioButton`	Crea un botón de radio.
`ScrollBar`	Crea una barra de desplazamiento.
`ScrollPane`	Crea un cuadro de desplazamiento.
`TextArea`	Crea un área de texto de dos dimensiones.
`TextField`	Crea un cuadro de texto de una dimensión.
`Window`	Crea una ventana.

Anexo.- Licencias de recursos.

Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos del recurso (2)
	Autoría: jdlasica. Licencia: CC-by-nc. Procedencia: http://www.flickr.com/photos/jdlasica/4258294714/		Autoría: Ilya K. Licencia: Free license. Procedencia: http://commons.wikimedia.c iki/File:NetBeans_IDE_6.0.p
	Autoría: Hélio Costa. Licencia: . CC-by-nc-sa. Procedencia: http://www.flickr.com/photos/hlegius/85231013/		Autoría: Robert Gaal. Licencia: CC-by. Procedencia: http://www.flickr.com/photos lueace/304306531/
	Autoría: Long Zheng. Licencia: CC-by-nc-sa. Procedencia: http://www.flickr.com/photos/longzheng/2889562804/		Autoría: Gianfranco Degranc Licencia: CC-by-nc-sa. Procedencia: http://www.flickr.com/photos egra/2501752605/
	Autoría: Beverly & Pack. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/walkadog/3353936487/		Autoría: DraXus. Licencia: CC-by. Procedencia: http://www.flickr.com/photos raxus/205876866/
	Autoría: Notfrancois. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/frenchy/30585495/		