UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL CÓRDOBA
DEPARTAMENTO DE SISTEMAS DE INFORMACIÓN
CÁTEDRA:
PARADIGMAS DE PROGRAMACIÓN

APUNTE TEORICO-PRACTICO

UNIDAD 2

PROGRAMACIÓN ORIENTADA A EVENTOS

AUTORES Y COLABORADORES:

Gustavo García
Soledad Alborno
Jorge Tymoschuk
Javier Ferreyra
Karina Ligorria

AÑO 2004

INDICE

1 Introducción 1
2 Los Applets 1
2.1 La Clase JApplet 2
2.1.1 El Ciclo de Vida de un Applet 2
2.2 “Hola Applet” 3
2.3 Páginas HTML y Etiquetas 3
2.3.1 Cargando el applet en el navegador 4
3 Componentes y Swing 5
3.1 Componentes y Contenedores 6
3.1.1 La clase `JComponent` 6
3.1.2 La clase `JPanel` 7
3.2 Componentes Básicos 7
3.2.1 Etiquetas (`JLabel`) 7
3.2.2 Botones (`JButton`) 7
3.2.3 Campos de edición (`JTextField`) 8
3.2.4 Casillas de verificación (`JCheckBox`) 8
3.2.5 Listas desplegables (`JComboBox`) 8
3.2.6 Listas (`JList`) 8
3.3 Contenedores Básicos 10
3.3.1 Paneles (`JPanel`) 10
3.3.2 Paneles Desplazables (`JScrollPanel`) 10
3.3.3 Ventanas (`JFrame`) 10
3.3.4 Ventanas de Diálogo (`JDialog`) 11
3.3.5 Diálogos de Opción (`JOptionPane`) 11
3.4 Un Applet con Componentes 12
4 Manejo de Eventos 14
4.1 El Modelo de Delegación 14
4.2 Respondiendo al Ratón 15
4.3 Escuchas y Adaptadores 17
4.4 Principales Tipos de Eventos 18
4.4.1 Eventos de ratón (`MouseEvent`) 18
4.4.2 Eventos de teclado (`KeyEvent`) 18
4.4.3 Eventos de acción (`ActionEvent`) 19
4.4.4 Eventos de ventana (`WindowEvent`) 21
4.5 Un Applet con Eventos 22
4.6 Una Ventana con Eventos 23
Parte Práctica 27
Enunciados 27
Ejercicio 1 27
Ejercicio 2 27
Ejercicio 3 27
Ejercicio 4 27

Ejercicio 5 27
Ejercicio 6 27
Ejercicio 7 28
Ejercicio 8 28
Ejercicio 9 28
Resolución de Ejercicios 29
Resolución para Ejercicio 1 29
Resolución para Ejercicio 2 29
Resolución para Ejercicio 3 30
Resolución para Ejercicio 4 31
Resolución para Ejercicio 8 32
Resolución para Ejercicio 9 33

Unidad 2 – Programación de Interfaces Gráficas de Usuario
Parte Teórica

1 Introducción

El mundo de la computación se ha convertido a las interfaces GUI. GUI significa interfaz gráfica de usuario por las siglas en inglés de Graphical User Interface. Una interfaz gráfica es la que ofrece al usuario diferentes componentes estándares que facilitan el uso de los programas. Uno de los ejemplos más difundidos de GUIs es el sistema operativo Windows de Microsoft. En este sistema el usuario no se enfrenta a una poco amigable línea de comando, como ocurría con el D.O.S. o las viejas versiones de Unix/Linux. En cambio, se le brinda un conjunto de elementos gráficos como ventanas, botones, áreas de edición de texto, menús, íconos, etc. que hacen más intuitivo el manejo de los programas.

La uniformidad en el tipo de componentes y en su forma de uso, aumenta la usabilidad de los programas. Por ejemplo, todo usuario de un sistema de ventanas sabe qué es un menú y cómo debe ser usado, o qué es un botón y cómo es accionado.

Asimismo, aparece la posibilidad de utilizar nuevos periféricos de entrada que no existían (o tenían una aplicación mucho más limitada) en los sistemas de texto. El periférico más común, y que en el día de hoy encontramos en todas las computadoras es el ratón o mouse.

Java provee un conjunto de APIs[1] muy bien equipadas para proporcionar estos elementos en los programas. La API de GUI se encuentra en un paquete llamado `javax.swing` (que a partir de ahora llamaremos simplemente Swing).

Sólo será posible utilizar GUIs en entornos gráficos que permitan el uso de ventanas y el manejo de eventos de usuario. Esto significa que no es posible utilizar interfaces gráficas de Java en sistemas operativos de línea de comando.

2 Los Applets

Antes de la aparición de las tecnologías de streaming[2] en Internet (como Flash, Real Audio y demás) las páginas Web eran páginas estáticas que contenían tan sólo texto y gráficos. Los hipervínculos eran lo único que les daba algún tipo de funcionamiento. Hasta que Java hizo posible escribir pequeños programas que se ejecutaban en el navegador, dando vida a la Web. Estas pequeñas aplicaciones recibieron el nombre de applets.

Un applet es una pequeña aplicación accesible en un servidor web, que se transporta por la red, se instala automáticamente y se ejecuta in situ como parte de una página web dentro del navegador. O sea que sin importar en qué servidor fue accedida la página web, el applet es transferido a la computadora del usuario y allí es ejecutada.

Como un applet puede estar en cualquier página a la que una persona desprevenida pudiera llegar, la arquitectura las limita en muchos aspectos como ser el acceso al disco local. Esta seguridad inherente junto a la innecesidad de instalación son las principales ventajas de los applets. La gran desventaja, que hizo que quedaran un poco relegadas, es la necesidad de transmitir todo el programa a través de la red cada vez que se deseara acceder al mismo, con las consiguientes demoras.

[1] API significa interfaz de programación de aplicaciones (por las siglas en inglés de Application Programmer Interface). Una API es un conjunto de objetos, clases, métodos y constantes agrupados en librerías de objetivos determinados.

[2] Transmisión de flujos de bytes. Esto se utiliza para poder transmitir cualquier tipo de información binaria dinámica, como audio y video.

Para escribir un applet debemos extender la clase `javax.swing.JApplet`, y se implementa como un panel dentro de la página HTML que lo contiene.

2.1 La Clase JApplet

La clase `JApplet` contiene un constructor sin parámetros que simplemente inicializa atributos privados. Posee 35 métodos que permiten mostrar imágenes, reproducir archivos de audio y responder a eventos, entre otras cosas. Por ejemplo, los métodos `getAudioClip()` y `getImage()` permiten cargar un clip de sonido y una imagen respectivamente, identificados por una URL[3].

2.1.1 El Ciclo de Vida de un Applet

Como los applets son cargados, instanciados y ejecutados dentro de un navegador web, y es éste quien controla toda su vida. Asimismo, Sun nos provee junto con la SDK de un visor de applets (`appletviewer.exe`) que puede ser utilizado para visualizar y probar applets sin la necesidad de un navegador.

Los estados y transiciones por los que pasa un applet durante su vida son los de la siguiente figura:

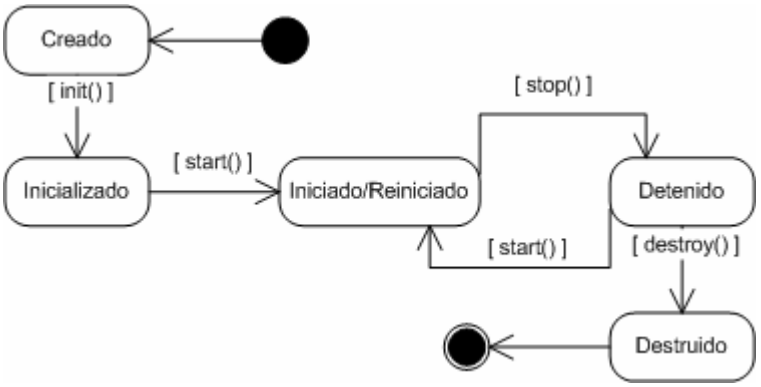


Figura 1 - Ciclo de vida de JApplet

Cada vez que se va a realizar la transición hacia alguno de estos estados se invoca al método indicado en el gráfico. Estos métodos son:

- **`init()`**: es invocado automáticamente por el entorno de ejecución (el navegador) cuando el applet se carga por primera vez, inmediatamente después de ser creado. Este método debe ser redefinido para realizar cualquier tipo de inicialización que se precise efectuar una única vez.
- **`start()`**: es llamado cada vez que el applet vuelve al área visible de la página HTML en el navegador web para permitirle comenzar con sus operaciones normales (particularmente aquellas que son detenidas por `stop()`). También invocado luego del `init()`. Este método debe ser redefinido cuando alguna actividad deba comenzar al mostrarse el applet.
- **`stop()`**: es llamado cada vez que el applet sale del área visible dentro del navegador, para permitirle detener operaciones costosas en procesamiento (como ser operaciones gráficas o de animación). También es llamado justo antes del `destroy()`. Este método debe ser redefinido cuando alguna actividad iniciada en el método `start()` debe detenerse al dejar de estar visible el applet.
- **`destroy()`**: es invocado cuando se descarga el applet del navegador web, para realizar la liberación de los recursos porque el applet no se usa más. Esto ocurre, por ejemplo, cuando el usuario cierra el navegador. Este método debe ser redefinido cuando se deben liberar recursos tomados por el applet, justo antes de su destrucción.

Estos son los métodos que muchas veces deberemos redefinir para hacer nuestra propia implementación del applet.

[3] Por las siglas en inglés de Localizador Uniforme de Recursos – Uniform Resource Locator.

2.2 "Hola Applet"

Ningún texto que enseñe a programar estará completo sin el clásico ejemplo del "Hola Mundo", así que aquí va la versión applet. A continuación vamos a transcribir el código de un applet que lo único que hace es mostrar el mensaje "Hola Applet". La explicación de cada paso va embebida en forma de comentarios.

Listado 1 – Código fuente del applet `HolaApplet`

```
import javax.swing.JApplet;
import java.awt.Graphics;

/**
 * Clase que representa el applet que mostrará el mensaje "Hola Applet" en la
 * pantalla del navegador. Esta clase extiende a javax.swing.JApplet.
 */
public class HolaApplet extends JApplet {
   /**
    * Cadena que contendrá el mensaje de texto a mostrar. Se
    * inicializará al valor "Hola Applet!!" en el método init().
    */
   private String mensaje = null;
   /**
    * Método invocado justo después de crear el applet. Aquí debemos
    * hacer todas las inicializaciones necesarias.
    */
   public void init() {
      mensaje = "Hola Applet!!";
   }
   /**
    * Este método es el encargado de dibujar el área gráfica del applet
    * en la pantalla del navegador. Lo redefinimos para dibujar el mensaje.
    */
   public void paint(Graphics areaGrafica) {
      areaGrafica.drawString(mensaje, 10, 40);
   }
}
```

En este caso sólo fue necesario redefinir `init()`, para inicializar el atributo que contiene el mensaje a mostrar. Como se puede ver, no es obligatorio redefinir todos los métodos que controlan el ciclo de vida, sino sólo aquellos que precisemos en nuestra implementación.

Para ver cómo funciona este applet, tenemos que crear una pequeña página HTML. Esta le indica al navegador, o al visor de applets, qué applet se debe cargar cuando se vea esa página web. Para poder hacer esto, vamos a ver algunos conceptos de páginas HTML y las etiquetas (tags) que utilizaremos.

2.3 Páginas HTML y Etiquetas

Las páginas Web se escriben en Lenguaje de Marcado de Hipertexto[4]. Éste es un formato de archivos de texto que utilizan etiquetas para describir la estructura y los componentes de la página como ser títulos, listas, vínculos, imágenes y applets. En esta sección veremos algunas de las etiquetas básicas que deberemos utilizar para poder mostrar un applet en una página.

Toda etiqueta HTML debe ir encerrada entre "**<**" y "**>**". Pueden escribirse indistintamente en mayúscula o minúscula. Hay dos tipos de etiquetas: separadoras y circundantes.

- Una *etiqueta separadora* se ubica entre los elementos a los cuales se aplica. Por ejemplo, la etiqueta
 provoca un salto de línea (break) entre dos líneas de texto.

[4] HTML por las siglas del inglés "HyperText Markup Language".

- Una *etiqueta circundante* es un par de etiquetas que se ubican alrededor del elemento que afectan. La primera etiqueta se llama etiqueta de apertura y la segunda etiqueta de cierre. La etiqueta de cierre posee un / entre el < y el nombre de la etiqueta. Por ejemplo, <HTML> y </HTML> que encierra todo el texto que debe ser interpretado como parte de la página por el navegador.

La siguiente es una lista incompleta de las etiquetas más importantes:

<HTML> </HTML>	Encierran todo el contenido de la página HTML.	<HEAD> </HEAD>	Encierran la cabecera de la página.
<BODY> </BODY>	Encierran el cuerpo de la página.	<TITLE> </TITLE>	Encierran el texto que será título de la página.
<B></B>	Caracteres en negrita entre los marcadores.	<I></I>	Caracteres en cursiva entre los marcadores.
<H#></H#>	Donde # es un número del 1 al 6 con el nivel del encabezado.	<APPLET> </APPLET>	Etiqueta utilizada para incluir un applet en una página HTML.
 	Salto de línea.	<P>	Comienzo de párrafo.

Algunas etiquetas pueden poseer atributos. En caso de ser aplicados a una etiqueta circundante, estos se colocan sólo en la etiqueta de apertura. Por ejemplo, en la etiqueta <BODY> se puede definir un color de fondo para la página distinto del blanco que se configura por defecto. Haciendo <BODY BGCOLOR="000000"> … </BODY> tendremos una página de fondo negro.

2.3.1 Cargando el applet en el navegador

Como mencionamos, la etiqueta <APPLET></APPLET> es la que se utiliza para incluir un applet en una página HTML. La etiqueta <APPLET> puede poseer, entre otros, los siguientes atributos:

- CODE: URL relativa a la dirección de la página HTML contenedora, que indica la ubicación del applet. Por ejemplo: CODE="HolaApplet.class"
- CODEBASE: cuando la dirección del applet difiera de aquella de la página web contenedora, deberá definirse este atributo con dicha URL.
- HEIGHT: define el alto, en píxeles, con el que se mostrará el applet.
- WIDTH: define el ancho, en píxeles, con el que se mostrará el applet.
- ALT: define el texto alternativo que se mostrará en los navegadores que aceptan applets pero no tengan el visor de applets habilitado.
- NAME: indica el nombre que tendrá el applet dentro de la página HTML. Puede haber múltiples applets dentro de una página HTML, y se los puede distinguir por su nombre.

Asimismo, la etiqueta <APPLET></APPLET> puede encerrar:

- Parámetros pasados por nombre al applet mediante la etiqueta <PARAM NAME="nombre" VALUE="valor">. Estos parámetros pueden ser recuperados desde el applet usando la función getParameter("nombre"), que retorna el valor en forma de cadena dentro de un String.
- Texto que será mostrado por los navegadores que no aceptan applets.

Ahora veamos el código fuente del archivo HolaApplet.html que muestra nuestro applet.

Listado 2 – Código fuente de la página HolaApplet.html

```
<HTML>
  <HEAD>
    <TITLE>Applet "Hola Applet"</TITLE>
  </HEAD>
  <BODY BGCOLOR="BBBBBB">
    <H1>Mi Primer Applet</H1>
    El cuadro blanco que sigue encierra el área de pintado del applet.<BR>
    <APPLET
      CODE   = "HolaApplet.class"
```

```
      HEIGHT = "100"
      WIDTH  = "200"
      ALT    = "Debe habilitar applets en su navegador para poder ver
                este applet."
      NAME   = "HolaApplet"
    >
      SU NAVEGADOR NO SOPORTA APPLETS.
    </APPLET>
    <P>El <I>tamaño</I> y la <I>ubicación</I> del applet son definidas por el
    código de la <B>página HTML</B> que lo contiene. Las características del
    área de pintado del applet son definidas dentro del código del applet.
  </BODY>
</HTML>
```

La figura que sigue muestra el resultado de ejecutar la página HTML del listado anterior usando el navegador de Microsoft. Repase atentamente las etiquetas del listado y vea qué efectos visuales tienen en la página en ejecución. El color de fondo del cuerpo de la página (el hexadecimal BBBBBB) es el gris que se ve en la figura.

Figura 2 – El applet HolaApplet en acción

3 Componentes y Swing

Swing es el nombre comercial que Sun Microsystems le dio a la parte GUI de las JFC (Java Foundation Classes – Clases Fundamentales de Java[5]). Esta API nos da la capacidad de desarrollar aplicaciones con interfaces basadas en ventanas y componentes.

Todos los componentes de Swing están escritos en Java puro. Esto ofrece varias ventajas, como ser independencia de la plataforma. De esta forma se puede mantener la apariencia de una aplicación en

[5] El concepto de *fundamental* se toma con su acepción de cimiento o basamento.

distintos sistemas operativos. También se independiza totalmente de las limitaciones del sistema de ventanas subyacente[6]. Además, somos libres de extenderlos y ampliarlos a nuestras necesidades.

Además, se trata de componentes *lightweight* (livianos). Esto significa que están implementados completamente en Java y su pintado es más rápido, ya que sólo se refrescan las partes que quedan visibles de los distintos componentes. Así se evita pintar un componente que se encuentre debajo de otro, o que quede fuera del área visible de la pantalla.

3.1 Componentes y Contenedores

Un componente es un elemento que puede formar parte de una interfaz gráfica. En Java, este elemento será instancia de alguna clase derivada de `javax.swing.JComponent`, y en casi todos los casos tendrá una representación visual. Ejemplos típicos de componentes gráficos son los botones, campos de edición y menús.

Los componentes no pueden existir y ser mostrados por sí mismos, sino que deben ser ubicados dentro de algún contenedor que los aloje. Ejemplos típicos de contenedores son los paneles, las ventanas, los diálogos y los applets. De estos, las ventanas, los diálogos y los applets son los únicos contenedores de nivel máximo y que pueden, por ende, ser mostrados independientemente de cualquier otro contenedor.

Para poder utilizar un componente, éste debe ser ubicado en un contenedor independiente como una ventana, que provea un lugar para que sus componentes internos se pinten. De esta manera, se crea una jerarquía de componentes que tiene su raíz en un contenedor independiente y forma una estructura de árbol con los componentes que contiene.

3.1.1 La clase `JComponent`

Como mencionamos, todos los componentes derivan de la clase `JComponent`[7]. Esta clase es la que agrupa todo el comportamiento estándar de un componente gráfico que no pueda ser independiente. Es la clase base para todos los componentes estándares y los desarrollados por el programador que utilicen la arquitectura de Swing. Poseen un "aspecto" o L&F (Look and Feel, en inglés) reemplazable. Esto significa que podemos ver una aplicación desarrollada utilizando Swing como si fuera una aplicación Windows, Mac o Solaris cambiando sólo el conjunto de definiciones que se utilizan para pintar, sin tocar el código. `JComponent` también tiene soporte para tool tips: pequeñas descripciones que se despliegan cuando el cursor del ratón se detiene sobre un componente.

La clase `JComponent` extiende a `java.awt.Container`. Por ende, cualquier componente de Swing puede funcionar a su vez como un contenedor y así contener otros componentes. Un ejemplo de esto es el botón que posee una etiqueta, un borde y hasta puede contener una imagen.

Su protocolo ofrece más de 100 métodos. Todos ellos vinculados con el manejo gráfico del componente. Algunos de los más comúnmente usados son:

- `setSize(int,int)`, `getWidth()` y `getHeight()`, que define el tamaño y retornan el ancho y el alto del componente en píxeles, respectivamente.
- `setLocation(int,int)`, `getX()` y `getY()`, que define la posición y retornan las coordenadas x e y donde se encuentra ubicado el componente, respectivamente.
- `setBounds(int,int,int,int)`, que define al mismo tiempo la posición (coordenadas x e y) y el tamaño (ancho y alto) del componente, resumiendo en un método los dos "*set*" anteriores. El

[6] Se entiende por sistema de ventanas subyacente al conjunto de facilidades (ventanas, componentes y manejo de eventos) que provee el sistema operativo sobre el que se ejecuta la aplicación.

[7] Java posee, además del conjunto de APIs de Swing, unas librerías gráficas más primitivas que salieron al mercado con la primera versión del SDK de Java: las AWT (del inglés Abstract Windowing Toolkit – Conjunto de Herramientas Abstractas de Ventanas). Sin embargo, estas librerías eran tan pobres gráficamente, que fueron completamente reemplazadas por el nuevo conjunto de clases incluidas en Swing. El paquete `java.awt` sigue existiendo por compatibilidad hacia atrás, y algunas de sus clases siguen siendo ancestros de clases de Swing como `java.awt.Component` es ancestra de `javax.swing.JComponent`. Para distinguirlas y evitar confusiones entre los nombres es que todos los nombres de las clases de los componentes de Swing empiezan con **J**.

método `getBounds()` retorna un objeto rectángulo que representa el área que ocupa el componente dentro de su contenedor.

- `isVisible()` y `setVisible(boolean)`, que indica y define si el componente debe mostrarse cuando se pinta su contenedor.
- `isEnabled()` y `setEnabled(boolean)`, que indica y define si un componente está habilitado, lo que afectará a su funcionamiento y pintado (muy importante para botones y menús).
- `requestFocusInWindow()`, que intenta darle el foco al componente que recibe el mensaje, siempre y cuando la ventana que lo contiene posea el foco. Lo de "intenta" se debe a que es una tarea muy dependiente de la plataforma.

Para una enumeración y descripción completa de todos los métodos de `JComponent` y sus ancestras, vaya a la documentación de la API de Java.

3.1.2 La clase `JPanel`

La clase `javax.swing.JPanel` deriva de `JComponent`, o sea que es un componente de Swing y como tal es un contenedor. `JPanel` es el contenedor más comúnmente usado en una interfaz gráfica. Representa un panel donde se insertan y distribuyen componentes. Todo contenedor independiente o de máximo nivel, como las ventanas y los applets, poseen un panel en el que se ubicarán todos los componentes que queremos incluir en la interfaz. Este panel se obtiene en todos los casos mediante el método `getContentPane()` (retorna el panel de contenidos) y existe desde que se crea la ventana o el applet.

El principal método que se usará de esta clase es `add(Component)`. Mediante este método se agregará un componente con el tamaño y en la ubicación absoluta que dicho componente tenga definida.

3.2 Componentes Básicos

En las siguientes secciones mencionamos las características principales de algunos de los componentes más comúnmente utilizados en la programación de interfaces gráficas. Todas las clases que se describen pertenecen al paquete `javax.swing`, por lo que se obviará el nombre del paquete y sólo se indicará le nombre de cada clase.

3.2.1 Etiquetas (`JLabel`)

Las etiquetas son usadas normalmente para mostrar texto no modificable al usuario. Un componente `JLabel` permite mostrar una cadena de texto, una imagen o ambos. Es derivado de la clase `JComponent`, por lo que comparte todo el comportamiento definido en ésta. Los principales métodos que agrega son:

- `setText(String)` y `getText()`, que define y retorna, respectivamente, la cadena de caracteres que mostrará la etiqueta.
- `setIcon(Icon)` y `getIcon()`, que define y retorna el ícono que mostrará la etiqueta.
- `setHorizontalAlignment(int)` y `getHorizontalAlignment()`, que define y retorna la alineación horizontal de la etiqueta.

3.2.2 Botones (`JButton`)

Los botones son útiles para recibir comandos dados por el usuario a través del ratón. La clase que nos permite implementar este tipo de componentes es llamada `JButton`. Esta clase deriva de `JComponent` y los principales métodos que agrega son:

- `setText(String)` y `getText()`, que define y retorna, respectivamente, la cadena de caracteres que mostrará el botón.
- `setIcon(Icon)` y `getIcon()`, que define y retorna el ícono que mostrará el botón.
- `setDisabledIcon(Icon)` y `getDisabledIcon()`, que define y retorna el ícono que mostrará el botón cuando esté deshabilitado. Si esa propiedad no se define, cuando el botón sea deshabilitado mostrará por defecto el mismo ícono que muestra cuando está habilitado.
- `setSelected(boolean)` e `isSelected()`, que define e indica si el botón se encuentra seleccionado. Esto es aplicable a los botones que permanecen presionados hasta el siguiente clic.

- `setMnemonic(int)` y `getMnemonic()`, que define e indica cuál es el método abreviado de tecla del botón, por ejemplo se puede configurar que un botón con la etiqueta Aceptar sea *presionado* al oprimir las teclas `Alt+A`.

3.2.3 Campos de edición (`JTextField`)

Cuando el usuario debe proveer información al programa, lo más común es ofrecerle un pequeño cuadro donde pueda ingresar texto. Esto puede ser realizado en Java instanciando la clase `JTextField`, que también es derivada de `JComponent`. Los principales métodos que define son:

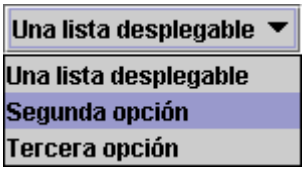
- `setText(String)` y `getText()`, que define y retorna, respectivamente, la cadena de caracteres que están contenidas dentro del campo de edición.
- `setEditable(boolean)` e `isEditable()`, que define e indica si el contenido del campo de edición es modificable (no confundir con habilitado/deshabilitado).
- `getSelectedText()`, que retorna el texto dentro del campo que se encuentra seleccionado.
- `setFont(Font)` y `getFont()`, que define y retorna la fuente en que se mostrará el texto dentro del campo de edición.

3.2.4 Casillas de verificación (`JCheckBox`)

A la hora de pedirle al usuario que seleccione opciones, cuando varias de ellas pueden ser seleccionadas simultáneamente, los `JCheckBox` le permiten ir tildando aquellas opciones deseadas. Este componente derivado de `JComponent` ofrece el mismo comportamiento especializado que un botón, agregando:

- `isSelected()` y `setSelected(boolean)`, que indica y define, respectivamente, si la casilla de verificación se encuentra seleccionada.

3.2.5 Listas desplegables (`JComboBox`)

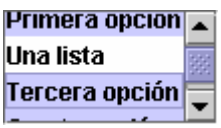


Una lista desplegable, instancia de `JComboBox`, nos permite seleccionar una de varias opciones posibles. Es parecido a un campo de edición con la particularidad de que nos puede mostrar un conjunto de líneas predeterminadas de las cuales el usuario podrá elegir. Así le ahorramos tener que escribir cuando sabemos cuáles son las posibles alternativas.

Este componente derivado de `JComponent` ofrece el siguiente comportamiento especializado:

- `addItem(Object)` y `removeItem(Object)`, que agrega y quita, respectivamente, un ítem a la lista desplegable.
- `getItemAt(int)` y `removeItemAt(int)`, que retorna una referencia al ítem y quita el ítem que se encuentra en la posición indicada por parámetro.
- `getItemCount()`, que retorna la cantidad de ítems existentes en la lista.
- `getSelectedIndex()` y `getSelectedItem()`, que retornan respectivamente la posición y una referencia al ítem de la lista seleccionado.
- `setSelectedIndex(int)` y `setSelectedItem(Object)`, que definen respectivamente la posición y el ítem seleccionado de la lista.
- `isEditable()` y `setEditable(boolean)`, que indica y define si el campo de edición del `JComboBox` es editable. Cuando no es editable sólo puede cambiar seleccionando un ítem de la lista. Cuando sí lo es, además de seleccionar cualquier ítem de la lista se puede introducir texto diferente al propuesto.

3.2.6 Listas (`JList`)



Cuando deseamos tener la posibilidad de ver y seleccionar más de una opción por vez, podemos utilizar objetos `JList`. Es el componente usado para tener listas de selección múltiple. Este componente derivado de `JComponent` ofrece el siguiente

comportamiento especializado:

- `clearSelection()`, deselecciona todos los elementos seleccionados de la lista.
- `addSelectionInterval(int,int)`, agrega a la selección existente todos los ítems que se encuentren entre los índices pasados por parámetro.
- `getselectedIndices()`, que retorna un vector con los índices de todos los ítems actualmente seleccionados.
- `setSelectedIndices(int[])`, que define cuáles van a ser todos los elementos seleccionados de la lista.
- `getselectedValues()`, que retorna un vector con las referencias a todos los ítems actualmente seleccionados.
- `isSelectedIndex(int)`, que indica por `true` o `false` si el elemento que se encuentra en la posición indicada por parámetro está actualmente seleccionado.

Además de este comportamiento directamente asociado a la selección, será necesario utilizar otro objeto cuando queramos agregar o quitar elementos de una lista durante la ejecución del programa. En la clase `JList` se aplica el patrón de diseño llamado MVC[8] (Modelo-Vista-Controlador). Este patrón separa en distintos objetos las responsabilidades de manejar el modelo de datos, de la visualización y pintado en pantalla, y de control del comportamiento. Así cada lista posee un modelo de datos asociado, que es a quien se le solicita que maneje (agregue o elimine) los elementos de la lista. Mientras que es la misma clase `JList` la encargada de la visualización y el lanzamiento de eventos. Podemos entonces utilizar una instancia de `DefaultListModel` de la siguiente manera:

```
DefaultListModel listModel;
JList            list;
.........
listModel = new DefaultListModel();
listModel.addElement("Gustavo García ");
listModel.addElement("Ana González");
listModel.addElement("Graciela Franchescini ");
listModel.addElement("Juan Pérez");

list = new JList(listModel);
```

Lo que crea una lista de nombres con los valores agregados inicialmente. Si ante algún evento se debe eliminar le elemento seleccionado, se podrá escribir:

```
int index = list.getSelectedIndex();
listModel.remove(index);
```

Igualmente, si se debe agregar un elemento la posición siguiente a la actualmente seleccionada, se deberá escribir:

```
int index = list.getSelectedIndex();
if (index == -1) {  // no hay un ítem seleccionado
    index = 0;
} else {
    index++;
}
listModel.insertElementAt(nombre, index); // se agrega otro nombre
```

Veamos entonces los principales métodos del protocolo de un modelo de datos para una lista.

- `addElement(Object)`, agrega el elemento al final de la lista.
- `clear()`, elimina todos los elementos, dejando la lista vacía.
- `contains(Object)`, retorna un `boolean` indicando si la lista contiene el objeto indicado.
- `get(int)`, retorna el objeto almacenado en la posición indicada de la lista.
- `getSize()`, retorna la cantidad de elementos en la lista.

[8] En realidad, las iniciales MVC provienen del inglés *Model-View-Controller*.

- `insertElementAt(Object,int)`, inserta el elemento pasado por parámetro en la posición indicada.
- `isEmpty()`, retorna un `boolean` indicando si la lista está vacía.
- `remove(int)`, elimina el enésimo elemento de la lista.

3.3 Contenedores Básicos

Como ya mencionamos, todos los componentes vistos necesitan un contenedor para poder ser mostrados. O sea que no poseen la independencia gráfica necesaria como para pintarse sin estar dentro de otro componente. A continuación, veremos algunos de los contenedores más comunes. Los dos primeros no son contenedores independientes (de máximo nivel), sino que se utilizan para mostrar de forma organizada los componentes anteriormente vistos.

3.3.1 Paneles (`JPanel`)

Este contenedor básico es, tal como dijimos, el más comúnmente usado. Si bien no es un contenedor independiente, tanto las ventanas como los diálogos poseen una instancia de la clase `JPanel` como raíz de la estructura de componentes de GUI que posee.

Los métodos particulares que definen el comportamiento de las instancias de esta clase son:

- `add(Component)`, que agrega un componente dentro del panel.
- `remove(Component)`, que quita un componente del panel.
- `removeAll()`, que quita todos los componentes del panel.

3.3.2 Paneles Desplazables (`JScrollPanel`)

Cuando queremos mostrar algún componente que es más grande que el área que tiene disponible para mostrarse, como por ejemplo una `JList`, tenemos que dar al usuario la posibilidad de que desplace el contenido del panel hacia arriba y abajo o hacia la derecha e izquierda. Para hacer esto tenemos un panel especial llamando `JScrollPane` (para una descripción de listas ver la sección 3.2.6) que puede mostrar barras de desplazamiento tanto vertical como horizontal.

Este contenedor derivado de `JComponent` ofrece un comportamiento especializado que es muy dependiente de lo que está mostrando. Dejaremos para el ejemplo de la sección 4.6 una demostración de cómo trabaja un panel de desplazamiento.

3.3.3 Ventanas (`JFrame`)

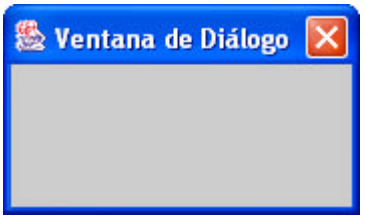
Una `JFrame` es el contenedor de máximo nivel por antonomasia. Toda aplicación gráfica independiente tendrá una instancia de esta clase donde se mostrarán los elementos principales del programa. Como se puede ver en el ejemplo que acompaña este párrafo, un `JFrame` es una ventana del sistema operativo sobre el que estemos trabajando. Por ende, la apariencia será siempre la que defina el sistema subyacente (Windows XP en el ejemplo).

Esta clase no deriva de `JComponent`, sino de `Window` por lo que su comportamiento difiere mucho de lo visto hasta el momento. Los principales métodos que nos ofrece son:

- `show()`, que muestra la ventana con todos los componentes que contenga.
- `hide()`, que oculta una ventana.
- `dispose()`, que libera todos los recursos de sistema que la ventana estaba utilizando, provocando que desaparezca si estaba siendo mostrada.
- `isActive()` e `isFocused()`, que indican si la ventana es la ventana activa del sistema, o sea si posee el foco.
- `toBack()` y `toFront()`, que envía a la ventana debajo de todas las que están siendo mostradas y la lleva hacia delante, respectivamente.

- `getTitle()` y `setTitle(String)`, que retorna y define la cadena que se muestra en la barra de título de la ventana.
- `isResizable()` y `setResizable(boolean)`, que indica y define si la ventana puede ser cambiada de tamaño por el usuario, arrastrando alguno de los bordes.
- `getContentPane()`, que retorna el panel que es la raíz de toda la jerarquía de componentes que se muestran en la ventana. Este método es muy importante ya que todo lo que queramos que sea mostrado en la ventana debe ser agregado al panel obtenido.

3.3.4 Ventanas de Diálogo (`JDialog`)



Cuando en lugar de una ventana independiente queremos construir una ventana de diálogo, usamos la clase `JDialog`. Una ventana de diálogo es muy parecida a una ventana con la diferencia de que no puede ser minimizada ni maximizada. Se utiliza cuando se requiere una respuesta o interacción precisa del usuario, de allí su nombre de ventana de "diálogo".

Ofrece el mismo comportamiento que una ventana `JFrame`, agregando la capacidad de tomar el control del foco para una aplicación hasta ser cerrada. Cuando una ventana de diálogo posee esta capacidad se dice que es una ventana *modal*. Los métodos propios que agrega son:

- `isModal()` y `setModal(boolean)`, que indica y define si la ventana es modal.
- Redefine el método `show()` de modo que si la ventana está definida como modal al momento de ser mostrada, el método no retorna hasta que la ventana no sea cerrada (usando `hide()` o `dispose()`).

3.3.5 Diálogos de Opción (`JOptionPane`)

Swing también ofrece al programador una clase de ventanas utilitarias. `JOptionPane` facilita la presentación de ventanas de diálogo que muestren un mensaje, o pidan algún valor o confirmación al usuario. Posee varios métodos estáticos de la forma `showXxxDialog()` que permiten mostrar ventanas de diálogo de distinto tipo. Los métodos más comúnmente usados son:

- `showMessageDialog(Component, Object, String, int)`, que muestra una ventana con un mensaje. Por ejemplo, de la siguiente sentencia

```
JOptionPane.showMessageDialog(this, "Se produjo un error", "Error",
                              JOptionPane.ERROR_MESSAGE);
```

se obtiene la ventana que se ve a continuación:

Figura 3 – Un mensaje de error

Cambiando el valor del último parámetro por algunas otras constantes se pueden obtener mensajes de precaución (`WARNING_MESSAGE`) o de información (`INFORMATION_MESSAGE`).

- `showConfirmDialog(Component, Object, String, int)`, que muestra una ventana con un mensaje de confirmación, retornando el código correspondiente al botón presionado.

```
JOptionPane.showConfirmDialog(this,
                        "Está seguro que desea salir de la aplicación?",
                        "Confirmación", JOptionPane.YES_NO_OPTION);
```

que produce:

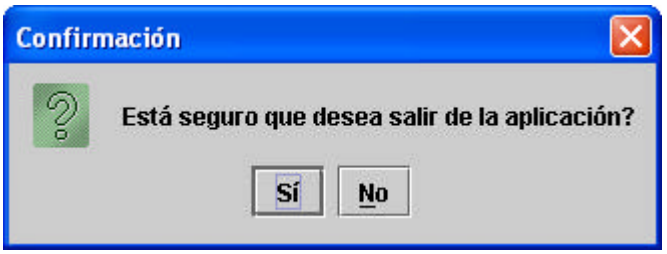


Figura 4 – Un mensaje de confirmación

- `showInputDialog(Component, String)`, que muestra una ventana con un campo de edición, retornando la cadena de caracteres introducida por el usuario.

```
JOptionPane.showInputDialog(this, "Introduzca su nombre");
```

que produce:

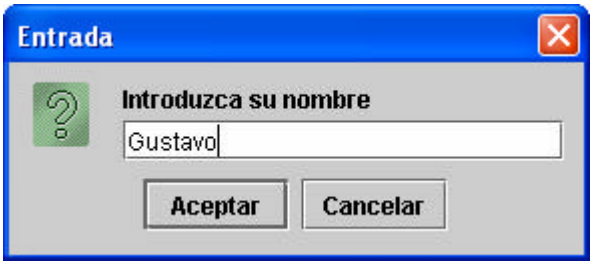


Figura 5 – Una entrada de datos

Como se puede ver, los métodos de esta clase construyen ventanas predefinidas que son útiles en muchas situaciones generales. En todos los casos, el primer parámetro (`Component`) es la ventana que será padre del diálogo. Si este parámetro es distinto de `null`, la ventana de diálogo será modal.

3.4 Un Applet con Componentes

A continuación tenemos el listado de un applet muy sencillo. Este applet posee simplemente una etiqueta de texto y 3 botones. Todavía no posee ningún funcionamiento (esperemos hasta la siguiente sección para ver cómo se manejan los eventos) pero muestra cómo se pueden agregar componentes de Swing a nuestra interfaz de usuario.

Listado 3 – Código fuente del applet `TresBotones`

```
import javax.swing.JApplet;
import javax.swing.JLabel;
import javax.swing.JButton;

public class TresBotones extends JApplet {

  private JLabel etiqueta;
  private JButton botonIzq;
  private JButton botonCtro;
  private JButton botonDer;

  public void init() {
    // elimina la distribución automática de componentes
    this.getContentPane().setLayout(null);
    // creación y configuración de la etiqueta
    etiqueta = new JLabel("Etiqueta Predeterminada");
    etiqueta.setHorizontalAlignment(JLabel.CENTER);
    etiqueta.setSize(340, 20);
    etiqueta.setLocation(20, 30);
    this.getContentPane().add(etiqueta);
    // creación y configuración del botón izquierdo
    botonIzq = new JButton("Izquierda");
    botonIzq.setSize(100, 27);
    botonIzq.setLocation(20, 70);
```

```
    this.getContentPane().add(botonIzq);
    // creación y configuración del botón central
    botonCtro = new JButton("Centro");
    botonCtro.setSize(100, 27);
    botonCtro.setLocation(140, 70);
    this.getContentPane().add(botonCtro);
    // creación y configuración del botón derecho
    botonDer = new JButton("Derecha");
    botonDer.setSize(100, 27);
    botonDer.setLocation(260, 70);
    this.getContentPane().add(botonDer);
  }
}
```

Y mediante la siguiente página HTML podremos ver el applet en acción.

Listado 4 – Código fuente de la página `TresBotones.html`

```
<HTML>
  <HEAD>
    <TITLE>Applet con Tres Botones</TITLE>
  </HEAD>
  <BODY BGCOLOR="000000">
    <CENTER>
      <APPLET CODE = "TresBotones.class" WIDTH = "380" HEIGHT = "130" >
      </APPLET>
    </CENTER>
  </BODY>
</HTML>
```

El navegador mostrará la ventana de la siguiente figura.

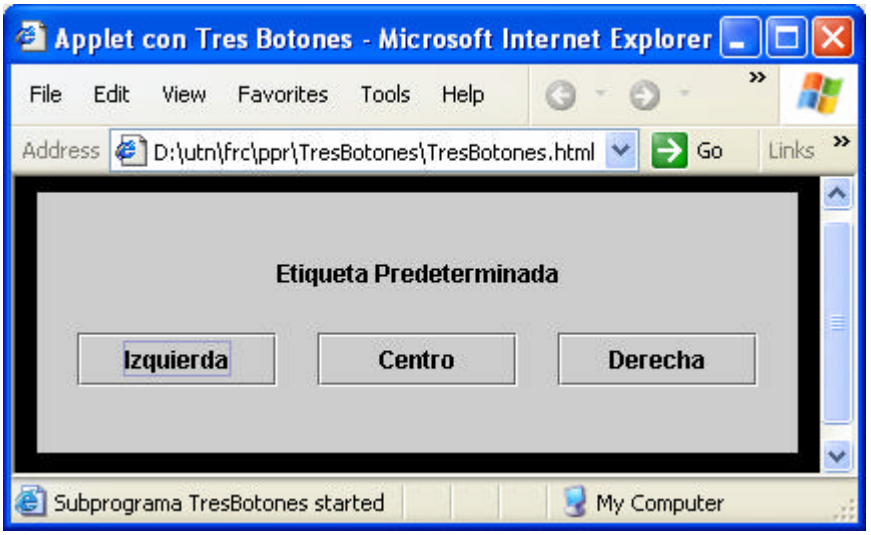


Figura 6 – El applet TresBotones en acción

Por defecto el foco aparece en el botón izquierdo por ser éste el primer componente enfocable que se agregó al applet. Si se presiona cualquier botón, simplemente no pasará nada porque todavía no le hemos dicho al applet que haga algo ante estos eventos.

La apariencia con que se muestran por defecto los componentes de Swing es propietaria de Java. Si lo deseamos, podemos ver el applet con la apariencia de Windows o Solaris. Agregando las siguientes líneas al comienzo del método `init()`

```
    // setea el L&F del applet en estilo Windows
    try {
      javax.swing.UIManager.setLookAndFeel(
          "com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
```

```
    } catch (Exception e) { }
```

se logrará un estilo de Windows, como se ve en la ventana de la izquierda de la figura. Y escribiendo

```
    // setea el L&F del applet en estilo Solaris
    try {
      javax.swing.UIManager.setLookAndFeel(
          "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
    } catch (Exception e) { }
```

se obtiene un estilo Motif (la interfaz gráfica de Solaris) que se ve en la ventana de la derecha de la figura. Como se puede advertir, la cadena que se pasa como parámetro a método `setLookAndFeel()` es la que define cómo se van a pintar los componentes del applet (o de la ventana).

Figura 7 – El applet TresBotones en estilo Windows y Solaris

Hasta aquí todo bien. Tenemos un applet, le agregamos una etiqueta y botones, y lo mostramos como si estuviéramos en distintas plataformas. Pero todavía no hicimos que el applet tenga ninguna utilidad. En la sección que sigue veremos cómo podemos reaccionar a los eventos de interfaz.

4 Manejo de Eventos

El usuario se comunica con los programas con interfaz gráfica ejecutando acciones como la de pulsar un botón del ratón o pulsar una tecla del teclado. Estas acciones tienen como resultado la generación de eventos. El proceso de respuesta a eventos se conoce como *manejo de eventos*. Se dice que los programas de ventanas son *conducidos por eventos*[9], ya que funcionan realizando acciones en respuesta a eventos.

En Java, desde la versión 1.1 de la JDK, se utiliza un modelo de delegación de eventos. Éste ofrece la posibilidad de entregar un evento a un objeto específico que se encargará de realizar las acciones necesarias.

4.1 El Modelo de Delegación

El modelo de delegación de eventos se basa en un patrón de diseño muy conocido: el *observador*[10]. La intención de este patrón es definir una relación de dependencia entre objetos (de uno a muchos) de manera que cuando un objeto cambie de estado, todos sus dependientes son notificados y actúan automáticamente.

En nuestro caso, el objeto central es un componente, digamos un botón, y su objeto dependiente[11] es el observador o escucha que recibirá las notificaciones de eventos y actuará en consecuencia. O sea que el escucha depende del botón, y debe enterarse de sus eventos.

[9] En textos en inglés son llamados *event driven*.

[10] Este modelo fue publicado en el muy conocido libro "Patrones de Diseño: Elementos de Software Orientado a Objetos Reutilizable", Gamma E. et al. Ed. Addison Wesley, 1995.

[11] Si bien en el párrafo anterior habíamos dicho que la intención del patrón de diseño era definir una relación de dependencia de uno a muchos objetos, por simplicidad vamos a continuar la explicación con un caso de dependencia uno a uno.

La estructura de clases de esta relación puede verse en la siguiente figura:

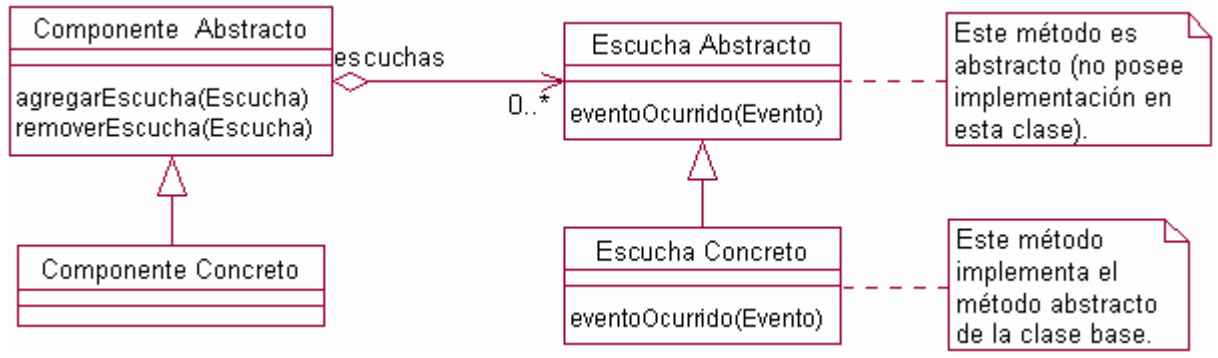


Figura 8 – Diagrama de clases del patrón observador

En el diagrama tenemos lo siguiente:

- Una clase *Componente Abstracto* que posee métodos para agregar y quitar escuchas u observadores de la lista que mantiene internamente.
- Una clase *Componente Concreto* que extiende al anterior, heredando los métodos necesarios para manejar escuchas asociados.
- Una clase *Escucha Abstracto* que define el protocolo del método que deberá ser llamado por cualquier componente que le quiera informar de un evento. Por ser esta definición sólo de protocolo, lo que esta clase posee es un método abstracto (sin implementación).
- Una clase *Escucha Concreto* que extiende a la anterior, definiéndole una implementación al método abstracto. Aquí es donde se define qué sucede cuando la ocurrencia del evento sobre el componente concreto es informada al escucha.

Ahora supongamos que nuestro componente concreto es un botón, instancia de la clase `JButton`, y que nos interesa responder a los clics del ratón hechos sobre dicho botón. Para esto definiremos una clase escucha concreta que implemente el método necesario[12]. Una instancia de esta clase será agregada como escucha de los métodos de ratón del botón. Ante la ocurrencia de un evento, el funcionamiento será el siguiente:

1. El usuario realiza un clic con el puntero del ratón ubicado sobre el botón.
2. El evento es capturado por el sistema operativo, que se lo envía a la máquina virtual de Java, que a su vez se lo envía al botón. El botón recibe dicho evento.
3. El botón notifica a todos los escuchas que tenga registrados (uno solo en este caso), invocando para cada uno de ellos el método redefinido.
4. El escucha del botón recibe el mensaje que le hace ejecutar el método redefinido.
5. La implementación de este método redefinido lleva adelante los pasos necesarios para responder al evento de clic del ratón.

Veamos todos estos pasos en un ejemplo. Vamos a agregarle al applet `TresBotones` el código necesario para capturar los eventos de clic sobre el botón izquierdo.

4.2 Respondiendo al Ratón

La API de Java nos provee lo siguiente para poder realizar esta tarea:

- Un método `addMouseListener(MouseListener)` implementado en la clase `Component`, que es heredado por todos los componentes en la jerarquía. Este método nos permite agregar un escucha de eventos de ratón a cualquier componente, incluyendo un botón.[13]

[12] Más adelante veremos que este método es el `mouseClicked(MouseEvent)` que habrá que redefinir.

[13] Como se puede advertir, escucha en inglés se escribe *listener*, y es por ello que los métodos para agregar escuchas a los componentes tienen la forma `addXxxListener()`, dependiendo del tipo de evento observado. Asimismo, los observadores de distintos tipos de eventos son llamados `XxxListener`.

- Una *interfaz* `MouseListener`, que define los siguientes métodos:
  - ⇨ `public void mouseClicked(MouseEvent e)`: método invocado por los componentes cuando reciben un clic del ratón.
  - ⇨ `public void mouseEntered(MouseEvent e)`: método invocado por los componentes cuando el puntero del ratón ingresa a su área de pantalla.
  - ⇨ `public void mouseExited(MouseEvent e)`: método invocado por los componentes cuando el puntero del ratón sale de su área de pantalla.
  - ⇨ `public void mousePressed(MouseEvent e)`: método invocado por los componentes cuando algún botón del ratón es presionado.
  - ⇨ `public void mouseReleased(MouseEvent e)`: método invocado por los componentes cuando algún botón del ratón es liberado, luego de haber sido presionado.

De lo anterior vemos que vamos a tener que invocar el método `addMouseListener()` del botón, pasando como parámetro el escucha. Este escucha deberá ser una instancia de una clase implementadora de la interfaz `MouseListener`, que defina el método `mouseClicked()` con el código que se debe ejecutar al presionar el botón usando el ratón.

El diagrama de clases para este caso particular será:

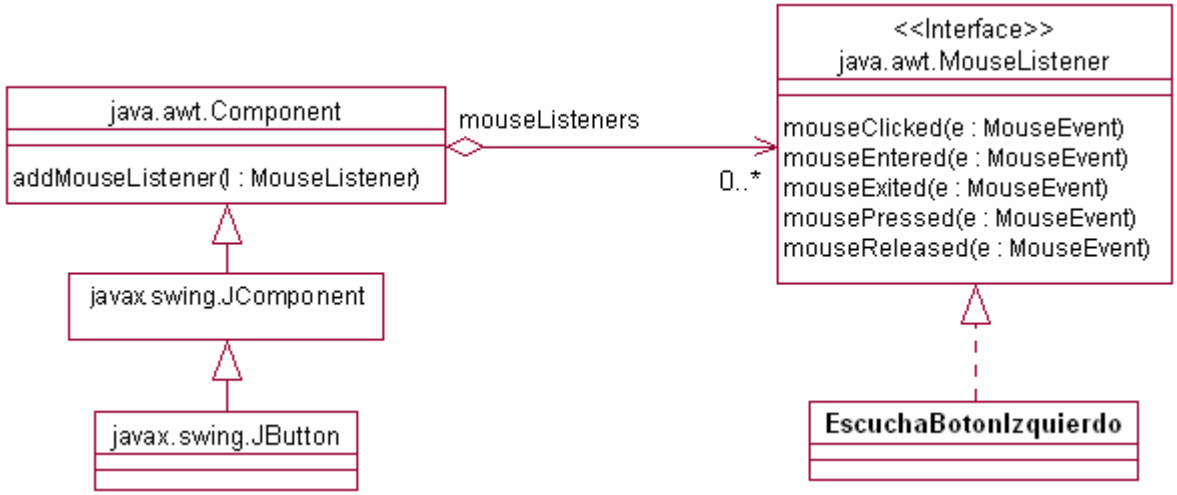


Figura 9 – Diagrama de clases del escucha del botón

Queremos que la etiqueta cambie su texto a "*Ud. ha presionado el botón izquierdo*" cuando hagamos clic sobre dicho botón. Para esto, el código que deberemos ejecutar dentro del applet será:

```
    etiqueta.setText("Ud. ha presionado el botón izquierdo");
```

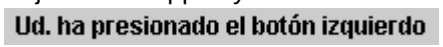
Pero considerando que esta línea debe ser el cuerpo de la redefinición del método `mouseClicked()` en la clase `EscuchaBotonIzquierdo`, nos queda el siguiente código donde hemos marcado con **negrita** el texto agregado.

Listado 5 – Código fuente del applet `TresBotones`, con manejo de eventos de ratón.

```
import javax.swing.JApplet;
import javax.swing.JLabel;
import javax.swing.JButton;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class TresBotones extends JApplet {

  private JLabel etiqueta;
  private JButton botonIzq;
  private JButton botonCtro;
  private JButton botonDer;

  public void init() {
```

```
    // elimina la distribución automática de componentes
    this.getContentPane().setLayout(null);
    // creación y configuración de la etiqueta
    etiqueta = new JLabel("Etiqueta Predeterminada");
    etiqueta.setHorizontalAlignment(JLabel.CENTER);
    etiqueta.setSize(340, 20);
    etiqueta.setLocation(20, 30);
    this.getContentPane().add(etiqueta);
    // creación y configuración del botón izquierdo
    botonIzq = new JButton("Izquierda");
    botonIzq.setSize(100, 27);
    botonIzq.setLocation(20, 70);
    this.getContentPane().add(botonIzq);
    // creación y configuración del botón central
    botonCtro = new JButton("Centro");
    botonCtro.setSize(100, 27);
    botonCtro.setLocation(140, 70);
    this.getContentPane().add(botonCtro);
    // creación y configuración del botón derecho
    botonDer = new JButton("Derecha");
    botonDer.setSize(100, 27);
    botonDer.setLocation(260, 70);
    this.getContentPane().add(botonDer);
    // se agrega el escucha al botón izquierdo
    botonIzq.addMouseListener(new EscuchaBotonIzquierdo());
  }

  class EscuchaBotonIzquierdo implements MouseListener {
    public void mouseClicked(MouseEvent e) {
      etiqueta.setText("Ud. ha presionado el botón izquierdo");
    }
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
  }
}
```

En la implementación se debieron importar una interfaz y una clase nuevas, que son las que nos van a permitir capturar los eventos. Se agregó un escucha de eventos de ratón al botón izquierdo, y se implementó la clase para este escucha. La clase `EscuchaBotonIzquierdo` debe implementar todos los métodos abstractos declarados en la interfaz que implementa, aunque sólo uno tenga alguna sentencia de código útil.

Ahora, al ejecutar el applet y hacer clic sobre el botón izquierdo el texto predeterminado será reemplazado por 

4.3 Escuchas y Adaptadores

Como hemos visto, los escuchas (listeners) son interfaces de Java que definen el protocolo de llamada a los distintos eventos. Cuando una de estas interfaces debe ser implementada, todos los métodos abstractos que declara deben ser definidos. De otro modo, la clase implementadora sería una clase abstracta y no podría ser instanciada.

A fin de evitarnos la necesidad de dar implementaciones vacías a los métodos declarados en la interfaz del escucha que no nos interesan, Java nos provee de clases implementadoras para todas las interfaces de escuchas que poseen más de un método. Estas clases definen todos los métodos abstractos de su interfaz, con implementaciones vacías. Por ejemplo, la clase `MouseAdapter` posee el siguiente código:

Listado 6 – Código fuente de la clase `MouseAdapter`

```
package java.awt.event;
```

```
public abstract class MouseAdapter implements MouseListener {
  public void mouseClicked(MouseEvent e) {}
  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}
}
```

Si partimos de esta clase adaptadora en lugar de la interfaz para definir `EscuchaBotonIzquierdo`, tendremos el siguiente código de clase (otra vez marcando los cambios en **negrita**):

Listado 7 – Código fuente de la clase `EscuchaBotonIzquierdo` extendiendo a `MouseAdapter`

```
// la instrucción "import java.awt.event.MouseListener;" fue reemplazada por
import java.awt.event.MouseAdapter;

  class EscuchaBotonIzquierdo extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
      etiqueta.setText("Ud. ha presionado el botón izquierdo");
    }
  }
```

Así nos evitamos hacer el código confuso con métodos vacíos que no utilizamos.

4.4 Principales Tipos de Eventos

Java provee soporte para muchos tipos de eventos. Algunos son los eventos de ratón que ya vimos, eventos de teclado, eventos de ventana, eventos de foco, eventos de contenedores, eventos de ítems de listas, eventos de cambio, eventos de menú, etc.

A continuación, describiremos algunos de los tipos de eventos más utilizados, y lo que Java nos provee para capturarlos.

4.4.1 Eventos de ratón (`MouseEvent`)

Estos son los que vimos en la sección '*4.2 – Respondiendo al Ratón*'. Lo que nos provee la API de Java es:

- Una *clase* `MouseEvent`, que posee toda la información referida al evento de ratón que provocó la llamada al método del escucha: tecla presionada (izquierda, central o derecha), posición donde se presionó, etc.
- Una *interfaz* `MouseListener`, que define el protocolo de los métodos para eventos de clic, presión de botones, liberación de botones, etc.
- Una *clase* `MouseAdapter`, que da una implementación vacía para todos los métodos de la interfaz anterior.
- Un método `addMouseListener(MouseListener)` implementado en la clase `Component`, que es heredado por todos los componentes en la jerarquía.

Estos elementos nos permiten crear objetos escuchas de cualquier evento disparado por el ratón.

4.4.2 Eventos de teclado (`KeyEvent`)

Cuando queremos capturar eventos de teclado, tenemos las siguientes utilidades provistas:

- Una *clase* `KeyEvent`, que posee toda la información referida al evento de teclado que provocó la llamada al método del escucha: tecla presionada, modificadores (Shift o Ctrl) presionados junto con la tecla, etc.
- Una *interfaz* `KeyListener`, que define el protocolo de los métodos:
  - ⇨ `public void keyPressed(KeyEvent e)`: método invocado por un componente cuando se presiona una tecla sobre él.

- ⇨ `public void keyReleased(KeyEvent e)`: método invocado por un componente cuando se suelta una tecla que había sido presionada sobre él.
- ⇨ `public void keyTyped(KeyEvent e)`: método invocado por un componente cuando se presiona y se suelta una tecla sobre él. Este evento implica los dos anteriores.

- Una *clase* `KeyAdapter`, que da una implementación vacía para todos los métodos de la interfaz anterior.
- Un método `addKeyListener(KeyListener)` implementado en la clase `Component`, que es heredado por todos los componentes en la jerarquía. Este método puede ser utilizado sobre cualquier componente para agregarle observadores de eventos de teclado.

Estos elementos nos permiten crear objetos escuchas de cualquier evento disparado mediante el teclado.

4.4.3 Eventos de acción (`ActionEvent`)

Si volvemos al ejemplo del applet `TresBotones`, vemos que implementamos un escucha que responde a los eventos del ratón sobre el botón izquierdo. ¿Qué ocurre si posicionamos el foco sobre el botón izquierdo y presionamos la barra espaciadora? El comportamiento propio de la clase `JButton` se encargará de representar en la pantalla el efecto de presión sobre el botón. Veremos lo mismo que se muestra cuando hacemos clic con el ratón sobre él. Sin embargo, el texto de la etiqueta no cambiará porque no implementamos ningún escucha para eventos de teclado.

Una solución a este problema puede ser agregar un escucha de eventos de teclado (`KeyListener`) que implemente el método `keyTyped()` filtrando las teclas. Si la tecla presionada es la barra espaciadora, entonces cambiará el texto de la etiqueta al igual que el escucha de eventos de ratón.

Como se puede deducir, esta solución es muy engorrosa ya que implica duplicar el código necesario para el evento. Si tenemos un sistema con varias ventanas, cada una de las cuales incluye diversos componentes y múltiples botones, la duplicación de código sería tremenda. Pasaría de ser un engorro a convertirse en código frágil y difícil de mantener.

Es por ello que la API de Java nos ofrece otro tipo de evento: el evento de acción. Un evento de acción es un evento de alto nivel que es disparado por un componente cuando se realiza la acción principal que puede ocurrir en dicho componente. La acción principal dependerá de cada componente. Por ejemplo, para un botón su "acción" principal es ser presionado. Para un menú: ser seleccionado. Para un campo de edición: cambiar su contenido. Para una lista de selección: cambiar la selección. Cada componente define cuál es su acción principal.

Para capturar este tipo de eventos tenemos:

- Una *clase* `ActionEvent`, que posee toda la información referida al evento de acción que provocó la llamada al método del escucha: componente sobre el que ocurrió, modificadores (Shift o Ctrl) presionados durante el evento, argumentos particulares del evento, etc.
- Una *interfaz* `ActionListener`, que define el protocolo del método:
  - ⇨ `public void actionPerformed(ActionEvent e)`: método invocado por un componente cuando ocurre un evento de acción sobre aquél.
- Un método `addActionListener(ActionListener)` implementado en la clase `Component`, que es heredado por todos los componentes en la jerarquía. Este método puede ser utilizado sobre cualquier componente para agregarle observadores de eventos de acción.

Podemos ver que en este caso no tenemos una clase adaptadora con implementación vacía para el método. Esto se debe a que la interfaz posee un solo método, y no tiene sentido que alguna clase implemente la interfaz dejando el único método vacío.

Agreguemos entonces al applet `TresBotones` un escucha de eventos de acción para los botones central y derecho. Este escucha cambiará el texto de la etiqueta a "Ud. ha presionado el botón central" o bien "Ud. ha presionado el botón derecho" dependiendo de qué botón presione el usuario, ya sea con el ratón o con el teclado.

Listado 8 – Código fuente del applet `TresBotones`

```
import javax.swing.JApplet;
import javax.swing.JLabel;
import javax.swing.JButton;
import java.awt.event.MouseEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class TresBotones extends JApplet {

  private JLabel etiqueta;
  private JButton botonIzq;
  private JButton botonCtro;
  private JButton botonDer;

  public void init() {
    // elimina la distribución automática de componentes
    this.getContentPane().setLayout(null);
    // creación y configuración de la etiqueta
    etiqueta = new JLabel("Etiqueta Predeterminada");
    etiqueta.setHorizontalAlignment(JLabel.CENTER);
    etiqueta.setSize(340, 20);
    etiqueta.setLocation(20, 30);
    this.getContentPane().add(etiqueta);
    // creación y configuración del botón izquierdo
    botonIzq = new JButton("Izquierda");
    botonIzq.setSize(100, 27);
    botonIzq.setLocation(20, 70);
    this.getContentPane().add(botonIzq);
    // creación y configuración del botón central
    botonCtro = new JButton("Centro");
    botonCtro.setSize(100, 27);
    botonCtro.setLocation(140, 70);
    this.getContentPane().add(botonCtro);
    // creación y configuración del botón derecho
    botonDer = new JButton("Derecha");
    botonDer.setSize(100, 27);
    botonDer.setLocation(260, 70);
    this.getContentPane().add(botonDer);
    // se agrega el escucha al botón izquierdo
    botonIzq.addMouseListener(new EscuchaBotonIzquierdo());
    // se crea el escucha para los eventos de acción
    ActionListener escuchaAccion = new EscuchaBotones();
    // se agrega el mismo escucha al botón derecho e izquierdo
    botonCtro.addActionListener(escuchaAccion);
    botonDer.addActionListener(escuchaAccion);
  }
  class EscuchaBotonIzquierdo extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
      etiqueta.setText("Ud. ha presionado el botón izquierdo");
    }
  }
  class EscuchaBotones implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if (e.getSource() == botonCtro) {
        etiqueta.setText("Ud. ha presionado el botón central");
      } else if (e.getSource() == botonDer) {
        etiqueta.setText("Ud. ha presionado el botón derecho");
      }
    }
  }
```

```
}
```

Concentrándonos en los cambios podemos advertir lo siguiente:

- Se importan la clase y la interfaz asociadas a los eventos de acción.
- Se define la clase `EscuchaBotones` que implementa la interfaz necesaria para eventos de acción. La implementación del método `actionPerformed()` utiliza el evento recibido por parámetro para definir cuál fue el objeto origen del evento. Dependiendo de cuál de los botones se trate será el texto que pondrá en la etiqueta.
- En el método `init()` se crea un único escucha de eventos de acción, que es agregado como observador de ambos botones.

Así queda completo el código del applet `TresBotones`. Se puede advertir que sólo el botón central y el derecho responderán tanto a los clics del ratón como a las presiones realizadas con el teclado.

4.4.4 Eventos de ventana (`WindowEvent`)

Cuando estamos trabajando con ventanas (`JFrame` o `JDialog`) podemos necesitar capturar los eventos que son originados por los cambios de estado de aquéllas. Para esto es que la API de Java nos provee lo siguiente:

- Una *clase* `WindowEvent`, que posee toda la información referida al cambio de estado de la ventana que provocó la llamada al método del escucha: si fue minimizada, maximizada, seleccionada, etc.
- Una *interfaz* `WindowListener`, que define el protocolo de los métodos:
  - ⇨ `public void windowActivated(WindowEvent e)`: método invocado por una ventana cuando es seleccionada y recibe el foco.
  - ⇨ `public void windowClosed(WindowEvent e)`: método invocado por una ventana inmediatamente después de haber sido cerrada.
  - ⇨ `public void windowClosing(WindowEvent e)`: método invocado por una ventana mientras se está cerrando. Este es el evento que debe ser implementado para salir de una aplicación basada en ventanas, al cerrar la ventana principal. Deberá contener la sentencia `System.exit(0)` para finalizar la máquina virtual sin código de error.
  - ⇨ `public void windowDeactivated(WindowEvent e)`: método invocado por una ventana cuando pierde el foco.
  - ⇨ `public void windowDeiconified(WindowEvent e)`: método invocado por una ventana cuando estaba minimizada y es restaurada.
  - ⇨ `public void windowIconified(WindowEvent e)`: método invocado por una ventana cuando es minimizada.
  - ⇨ `public void windowOpened(WindowEvent e)`: método invocado por una ventana cuando es abierta y mostrada por primera vez.
- Una *clase* `WindowAdapter`, que da una implementación vacía para todos los métodos de la interfaz anterior.
- Un método `addWindowListener(WindowListener)` implementado en la clase `Window`, que es heredado por todas sus derivadas. Este método puede ser utilizado sobre cualquier ventana para agregarle observadores de eventos.

Con estos eventos podemos enterarnos de lo que ocurre en todo momento con nuestra aplicación de ventanas.

Hemos visto que en Java podemos capturar muchos tipos de eventos, de los cuales describimos los más comúnmente utilizados. Para mayor detalle y para conocer una lista completa de los eventos disponibles, por favor diríjase a la documentación de la API que provee Sun o a la bibliografía recomendada por la cátedra.

4.5 Un Applet con Eventos

Veamos ahora el código de un applet que posea componentes y haga algo básico. Vamos a construir el applet `Copiador` que tendrá la interfaz de la siguiente figura.

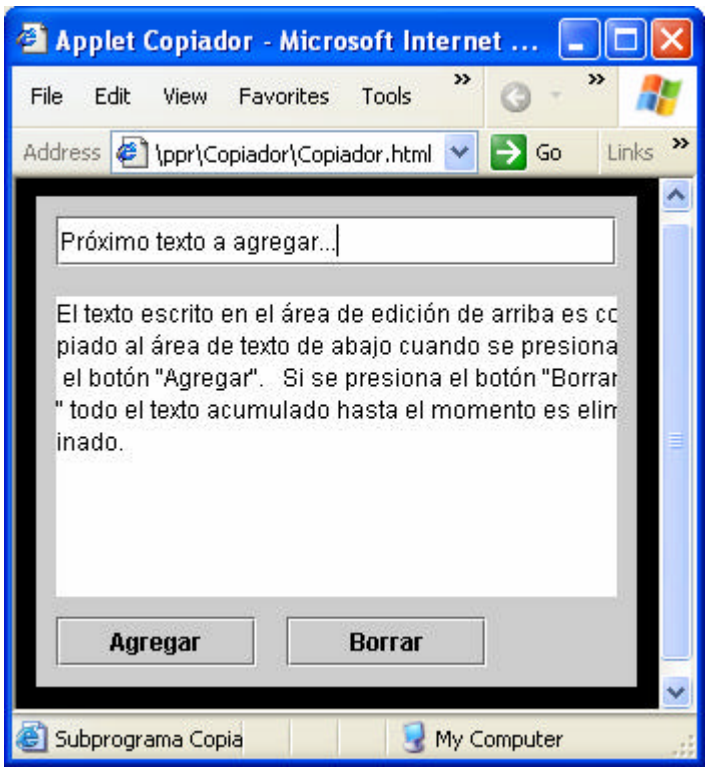


Figura 10 – El applet `Copiador` en acción

Como el mismo texto explica, la intención del applet es permitir escribir texto en el campo de edición de arriba. Cuando se presiona el botón "Agregar" el texto es copiado al área de texto inferior, agregándola al final del texto existente y borrando el campo de edición. Cuando se presiona el botón "Borrar" todo el texto existente tanto en el campo de edición, como en el área de texto es eliminado.

El código del applet es sencillo y se deja como ejercicio consultar en la bibliografía cualquier clase o método utilizado que no haya sido explicado hasta el momento.

Listado 9 – Código fuente del applet `Copiador`

```
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JTextField;
import javax.swing.JTextArea;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Copiador extends JApplet {

  private JTextField textField = new JTextField();
  private JTextArea textArea = new JTextArea();
  private JButton btnAgregar = new JButton("Agregar");
  private JButton btnBorrar = new JButton("Borrar");

  public void init() {
    getContentPane().setLayout(null);
    // se configura la posición y el tamaño de los componentes
    textField.setBounds(10, 10, 280, 25);
    textArea.setBounds(10, 50, 280, 150);
    textArea.setLineWrap(true);
    btnAgregar.setBounds(10, 210, 100, 25);
```

```
    btnBorrar.setBounds(125, 210, 100, 25);
    // se agregan los componentes al applet
    getContentPane().add(textField);
    getContentPane().add(textArea);
    getContentPane().add(btnAgregar);
    getContentPane().add(btnBorrar);
    // se agregan los escuchas de los botones
    btnAgregar.addActionListener(new AgregarAction());
    btnBorrar.addActionListener(new BorrarAction());
  }
  class AgregarAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String texto = textArea.getText() + textField.getText();
      textArea.setText(texto);
      textField.setText("");
    }
  }
  class BorrarAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      textArea.setText("");
      textField.setText("");
    }
  }
}
```

Sigue el código de la página HTML utilizada para mostrar este applet.

Listado 10 – Código fuente de la página `Copiador.html`

```
<HTML>
  <HEAD>
    <TITLE>Applet Copiador</TITLE>
  </HEAD>
  <BODY BGCOLOR="000000">
    <CENTER>
      <APPLET CODE = "Copiador.class" WIDTH  = "300" HEIGHT = "245" >
      </APPLET>
    </CENTER>
  </BODY>
</HTML>
```

4.6 Una Ventana con Eventos

A continuación desarrollamos una aplicación independiente basada en ventanas en lugar de usar applets. Esta aplicación permitirá realizar la copia de texto como se estaba haciendo en el ejemplo de la sección anterior, con tres nuevas capacidades:

- Se agrega una casilla de verificación que nos permite indicar si queremos agregar el nuevo texto al final del que se había ingresado anteriormente. Si la casilla no está seleccionada, el texto del área inferior es reemplazado por el nuevo texto introducido cada vez que se presiona el botón copiar.
- Se permite desplazar el área de texto hacia arriba y hacia abajo para poder ver todas las líneas de texto agregadas.
- Se pide confirmación cuando se presiona el botón borrar.

El resultado obtenido es el de la siguiente figura:

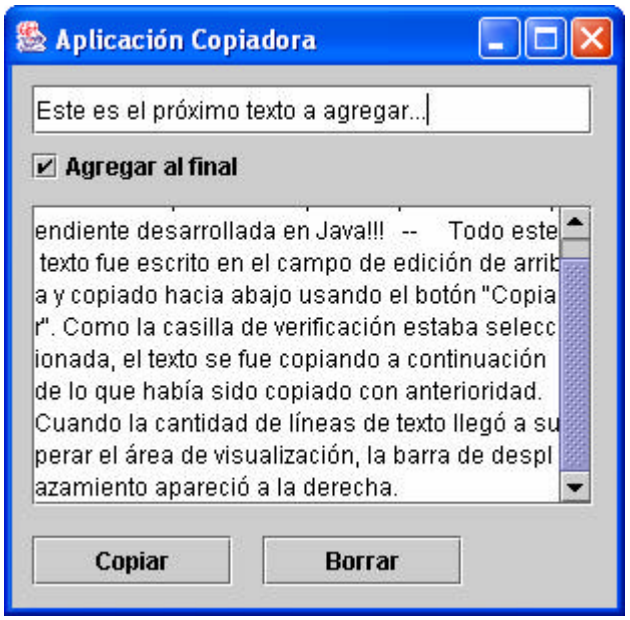


Figura 11 – La Aplicación Copiadora en acción

Cuando se presione el botón borrar, aparecerá el siguiente diálogo de confirmación:

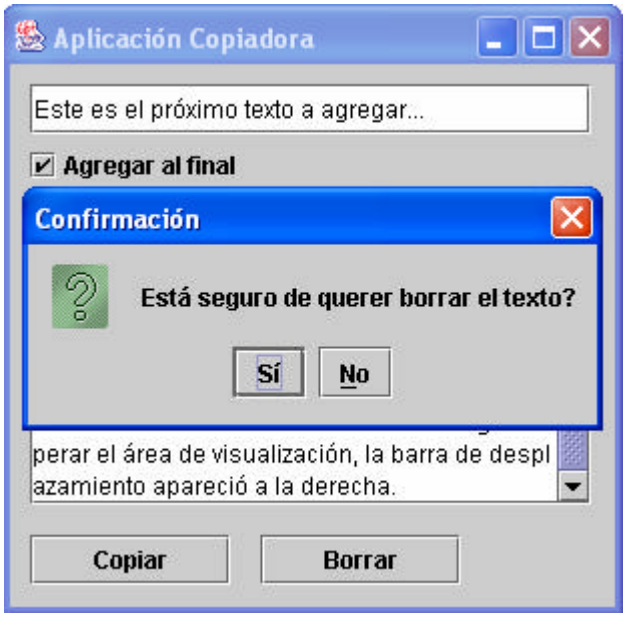


Figura 12 – Confirmación de borrado

Si presionamos **Sí**, el texto será borrado. Caso contrario, no pasará nada. A continuación vemos el código de esta aplicación.

Listado 11 – Código fuente de la ventana `Aplicacion`

```
import javax.swing.*;
import java.awt.event.*;

class Aplicacion extends JFrame {

  private JTextField textField = new JTextField();
  private JTextArea textArea = new JTextArea();
  private JCheckBox checkbox = new JCheckBox("Agregar al final");
  private JButton btnCopiar = new JButton("Copiar");
  private JButton btnBorrar = new JButton("Borrar");

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

```
    Aplicacion ventana = new Aplicacion();
    ventana.setSize(310, 305);
    ventana.setTitle("Aplicación Copiadora");
    ventana.setVisible(true);
  }

  public Aplicacion() {
   this.getContentPane().setLayout(null);
    getContentPane().setLayout(null);
    // se configura la posición y el tamaño de los componentes
    textField.setBounds(10, 10, 280, 25);
    checkbox.setBounds(10, 40, 200, 20);
    checkbox.setSelected(true);
    textArea.setLineWrap(true);
    // se ubica el área de texto dentro de un panel desplazable
    JScrollPane scrollPane = new JScrollPane(textArea);
    scrollPane.setBounds(10, 70, 280, 150);
    btnCopiar.setBounds(10, 235, 100, 25);
    btnBorrar.setBounds(125, 235, 100, 25);
    // se agregan los componentes al applet
    getContentPane().add(textField);
    getContentPane().add(checkbox);
    getContentPane().add(scrollPane);
    getContentPane().add(btnCopiar);
    getContentPane().add(btnBorrar);
    // se agregan los escuchas de los botones
    btnCopiar.addActionListener(new CopiarAction());
    btnBorrar.addActionListener(new BorrarAction());
    // se agrega el escucha de la ventana
    this.addWindowListener(new AplicacionWindow());
  }
  class AplicacionWindow extends WindowAdapter {
    // al cerrar la ventana se termina la aplicación
    public void windowClosing(WindowEvent e) {
      dispose();
      System.exit(0);
    }
  }
  class CopiarAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String texto = "";
      if (checkbox.isSelected()) {
        texto = textArea.getText();
      }
      texto += textField.getText();
      textArea.setText(texto);
      textField.setText("");
    }
  }
  class BorrarAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int ret;
      ret = JOptionPane.showConfirmDialog(Aplicacion.this,
                    "Está seguro de querer borrar el texto?",
                    "Confirmación", JOptionPane.YES_NO_OPTION);
      if (ret == JOptionPane.YES_OPTION) {
        textArea.setText("");
        textField.setText("");
      }
    }
  }
}
```

Debe advertirse la presencia de un `main()`, por tratarse de una aplicación independiente. Asimismo, en el escucha de eventos de ventana puede verse un ejemplo de cómo se debe hacer para finalizar la máquina virtual cuando se cierra la ventana.

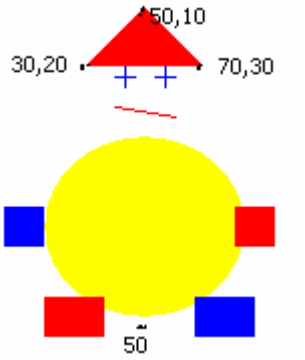
Parte Práctica

Enunciados

A continuación se presentan varios enunciados para desarrollar programas con interfaz gráfica.

Ejercicio 1

Crear un applet que dibuje un payaso como el de la figura:



Ejercicio 2

Desarrollar un applet para verificación de contraseñas, para ello debe solicitar un usuario y una contraseña y al presionar el botón Ingresar que se muestre si el usuario es conocido o no.

Ejercicio 3

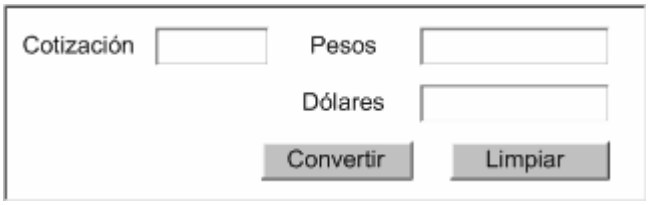
Desarrollar un applet que posea un botón y una etiqueta en la cual se vaya contando la cantidad de veces que se presiona el botón.

Ejercicio 4

Desarrollar un applet que tenga dos áreas de texto y un botón de modo que al presionar el mismo se copie el texto de un campo al otro.

Ejercicio 5

Desarrollar un applet que posea una apariencia como la de la figura, y que permita convertir una cantidad de dinero en pesos a dólares.



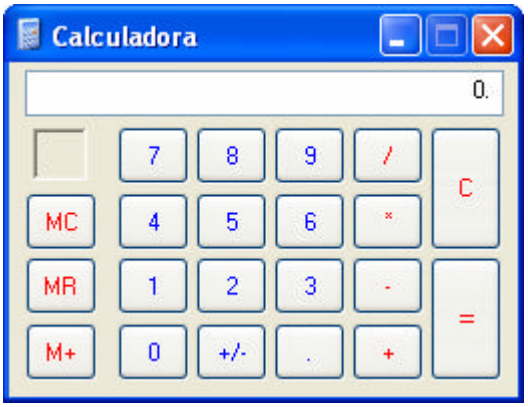
Cuando se presiona el botón “Limpiar” todos los campos de edición, a excepción de la cotización, deberán ser borrados.

Ejercicio 6

Extender el applet del punto anterior para que pueda también convertir de dólares a pesos. Para esto, se deberá agregar un nuevo botón.

Ejercicio 7

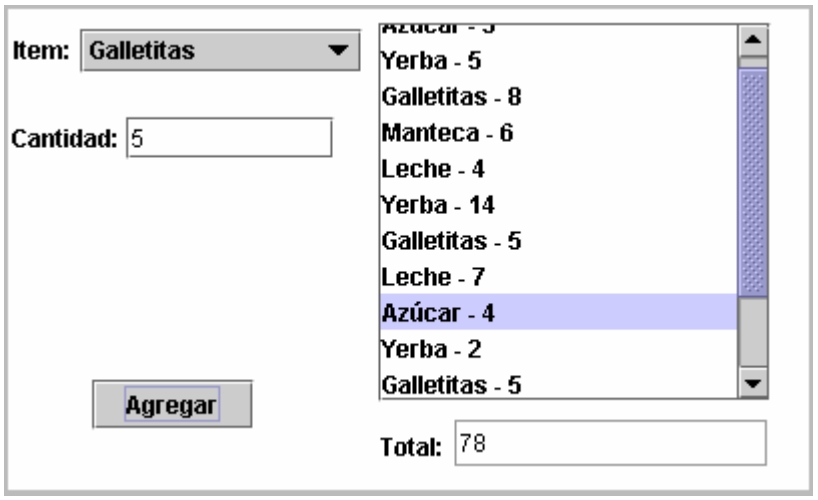
Desarrollar un applet o una aplicación que funcione como una calculadora sencilla. Las operaciones que debe incluir son las de la figura.



Como guía para el funcionamiento de cada elemento de la interfaz, se puede utilizar la calculadora de Windows.

Ejercicio 8

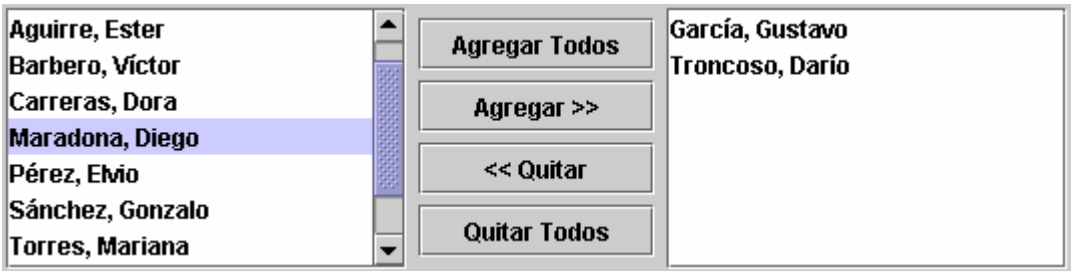
Desarrollar un applet para llevar el inventario de un pequeño almacén. Este applet deberá permitir seleccionar productos de una lista desplegable e indicar la cantidad que se posee. Luego se los deberá incluir en una lista, e ir sumando el total de productos más abajo, como se muestra en la figura.



Ejercicio 9

Desarrollar un applet que implemente la funcionalidad necesaria para intercambiar elementos entre dos listas. En este caso es una lista con nombres de personas a la izquierda que pueden se agregadas o quitadas de la lista de participantes de un curso, que se muestra a la derecha.

En la figura se puede apreciar la apariencia deseada.



Resolución de Ejercicios

Resolución para Ejercicio 1

Código fuente Java:

```
import javax.swing.JApplet;
import java.awt.Graphics;
import java.awt.Color;

public class Payaso extends JApplet
{
  public void paint(Graphics g) {
    g.setColor(Color.white); //establece el color
    g.fillOval(70, 30, 60, 50); //dibuja un círculo
    g.setColor(Color.red);
    int [] xs = {70, 100, 130};
    int [] ys = {40, 10, 40};
    g.fillPolygon(xs, ys, 3); //dibuja el triángulo
    g.setColor(Color.yellow);
    g.fillOval(50, 75, 100, 90);
    g.setColor(Color.blue);
    g.fillRect(30, 110, 20, 20); //dibuja rectángulo relleno
    g.setColor(Color.red);
    g.fillRect(145, 110, 20, 20);
    g.fillRect(50, 155, 30, 20);
    g.setColor(Color.blue);
    g.fillRect(125, 155, 30, 20);
    g.drawLine(90, 40, 90, 50);
    g.drawLine(110, 40, 110, 50);
    g.drawLine(85, 45, 95, 45); //dibuja las líneas de la cara
    g.drawLine(105, 45, 115, 45);
    g.setColor(Color.red);
    g.drawLine(85, 60, 115, 65);
  }
}
```

Código fuente HTML:

```
<HTML>
  <HEAD>  <TITLE>Applet "Payaso"</TITLE>  </HEAD>
  <BODY>
    <H1>Payaso</H1>
    <APPLET CODE = "Payaso.class" HEIGHT = "240" WIDTH = "400"> </APPLET>
  </BODY>
</HTML>
```

Resolución para Ejercicio 2

Código fuente Java:

```
import java.awt.event.*;
import javax.swing.*;
import java.awt.*;
public class Password extends JApplet
{
  private JLabel nomLbl;
  private JTextField nombre;
  private JLabel passLbl;
  private JPasswordField pass;
  private JButton ing;
  private JLabel res;

  public void init()
```

```
  {
    nomLbl = new JLabel("Usuario ");
    passLbl = new JLabel();
    passLbl.setText("Password");
    nombre = new JTextField("", 30);
    pass = new JPasswordField("", 30); //que el texto de la contraseña
                                       // escriba * en lugar de las letras
    res = new JLabel("Ingrese su nombre y contrasenia");
    ing = new JButton("Ingresar");
    getContentPane().setLayout(new FlowLayout());
    getContentPane().add(nomLbl);
    getContentPane().add(nombre);
    getContentPane().add(passLbl);
    getContentPane().add(pass);
    getContentPane().add(res);
    getContentPane().add(ing);
    // agregar el escucha de eventos al botón
    ing.addActionListener(new AccionIng());
  }
  class AccionIng implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      //controla el acceso
      if (nombre.getText().equals("Soledad") &&
          pass.getText().equals("Alborno")) {
        res.setText("Acceso Permitido");
      } else {
        if (nombre.getText().equals("Guillermo") &&
            pass.getText().equals("Colazo")) {
          res.setText("Acceso Permitido");
        } else {
          res.setText("Acceso Denegado");
        }
      }
    }
  }
}
```

Código fuente HTML:

```
<HTML>
  <HEAD>  <TITLE>Applet "Password"</TITLE>  </HEAD>
  <BODY>
    <H2>Ingreso de Password</H2>
    <APPLET CODE = "Password.class" HEIGHT = "120" WIDTH = "400"/>
  </BODY>
</HTML>
```

Resolución para Ejercicio 3

Código fuente Java:

```
import javax.swing.*;
import java.awt.FlowLayout;
import java.awt.event.*;

public class Contador extends JApplet {
  int contador = 0;
  JLabel lblCuenta;
  JButton btnCuenta;
  // cuando se carga el applet se le agregan los componentes
  public void init() {
    // crear e incializar los componentes
    lblCuenta = new JLabel("Clicks: 0");
    btnCuenta = new JButton ("Contar");
    btnCuenta.addActionListener(new ContarAction());
```

```
    // agregar los componentes al panel de contenidos del applet
    this.setSize(300, 80);
    this.getContentPane().setLayout(new FlowLayout());
    this.getContentPane().add(btnCuenta);
    this.getContentPane().add(lblCuenta);
  }
  // cuando se reinicia el applet, empezar a contar de 0
  public void start() {
    contador = 0;
    lblCuenta.setText("Clicks: " + contador);
  }
  // manejo del evento de acción del botón
  class ContarAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      contador++;
      lblCuenta.setText("Clicks: " + contador);
    }
  }
}
```

Código fuente HTML:

```
<HTML>
  <HEAD>  <TITLE>Applet Contador de Clicks</TITLE>  </HEAD>
  <BODY>
    <H3>Contador de Clicks</H3>
    <APPLET CODE = "Contador.class" WIDTH = "170" HEIGHT = "40"> </APPLET>
  </BODY>
</HTML>
```

Resolución para Ejercicio 4

Código fuente Java:

```
import javax.swing.*;
import java.awt.FlowLayout;
import java.awt.event.*;

public class CopiaTexto extends JApplet {
  JTextArea textOrigen;
  JTextArea textDestino;
  JButton btnCopia;

  public void init() {
    String s = "Java es un nuevo lenguaje\n" +
               "de programación orientado\n" +
               "a objetos con unas\n" +
               "características que lo\n" +
               "hacen especialmente interesante\n" +
               "para el desarrollo de\n" +
               "miniaplicaciones, denominadas\n" +
               "applets, que pueden integrarse\n" +
               "dentro de páginas Web.";
    // crear e incializar los componentes
    textOrigen = new JTextArea(10, 20);
    textOrigen.setLineWrap(true);
    textOrigen.setWrapStyleWord(true);
    textOrigen.setText(s);
    textDestino = new JTextArea(10, 20);
    textDestino.setLineWrap(true);
    textDestino.setWrapStyleWord(true);
    btnCopia = new JButton ("Copiar >>");
    btnCopia.addActionListener(new CopiarAction());
    // agregar los componentes al panel de contenidos del applet
    this.setSize(560, 180);
```

```
    this.getContentPane().setLayout(new FlowLayout());
    this.getContentPane().add(new JScrollPane(textOrigen));
    this.getContentPane().add(btnCopia);
    this.getContentPane().add(new JScrollPane(textDestino));
  }
  // manejo del evento de acción del botón
  class CopiarAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      textDestino.setText(textOrigen.getText());
    }
  }
}
```

Código fuente HTML:

```
<HTML>
  <HEAD>  <TITLE>Applet Copiador de Texto</TITLE>  </HEAD>
  <BODY>
    <H3>Copiador de Texto</H3>
    <APPLET CODE = "CopiaTexto.class" WIDTH = "560" HEIGHT = "180"> </APPLET>
  </BODY>
</HTML>
```

Resolución para Ejercicio 8

Código fuente Java:

```
import java.awt.Color;
import java.awt.event.*;
import javax.swing.*;

public class Inventario extends JApplet {

  int total;  // acumulador de cantidaded ingresadas
  // componentes visuales
  JComboBox comboItems;
  JLabel labelItem;
  JLabel labelCant;
  JTextField txtCant;
  JButton btnAgregar;
  JScrollPane scroll;
  DefaultListModel listModel;
  JList lista;
  JLabel labelTotal;
  JTextField txtTotal;

  public void init() {
    total = 0;    // se inicializa el acumulador
    // se inicializa el applet
    this.setSize(390, 240);
    this.getContentPane().setLayout(null);
    // se crean los componentes
    comboItems = new JComboBox();
    labelItem = new JLabel();
    labelCant = new JLabel();
    txtCant = new JTextField();
    btnAgregar = new JButton();
    scroll = new JScrollPane();
    listModel = new DefaultListModel();
    lista = new JList(listModel);
    labelTotal = new JLabel();
    txtTotal = new JTextField();
    // se inicializan los componentes
    comboItems.setBounds(36, 10, 141, 22);
    labelItem.setText("Item:");
```

```
    labelItem.setBounds(3, 10, 35, 22);
    labelCant.setText("Cantidad:");
    labelCant.setBounds(2, 55, 59, 22);
    txtCant.setBounds(59, 55, 104, 22);
    btnAgregar.setBounds(42, 185, 81, 25);
    btnAgregar.setText("Agregar");
    scroll.setBounds(185, 7, 196, 189);
    labelTotal.setText("Total:");
    labelTotal.setBounds(186, 205, 42, 22);
    txtTotal.setBackground(Color.white);
    txtTotal.setEditable(false);
    txtTotal.setBounds(223, 205, 156, 22);
    // se agregan los componentes al panel de contenidos del applet
    this.getContentPane().add(txtTotal);
    this.getContentPane().add(comboItems);
    this.getContentPane().add(labelCant);
    this.getContentPane().add(txtCant);
    this.getContentPane().add(scroll);
    this.getContentPane().add(labelTotal);
    this.getContentPane().add(btnAgregar);
    this.getContentPane().add(labelItem);
    scroll.getViewport().add(lista);
    // se agregan items a la lista desplegable
    comboItems.addItem("Azúcar");
    comboItems.addItem("Yerba");
    comboItems.addItem("Galletitas");
    comboItems.addItem("Leche");
    comboItems.addItem("Manteca");
    // se agrega un escucha de eventos de acción al botón Agregar
    btnAgregar.addActionListener(new AgregarAction());
  }
  // definición del escucha de eventos de acción para el botón agregar
  class AgregarAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      int cant;
      try {
        cant = Integer.parseInt(txtCant.getText());
        listModel.addElement(comboItems.getSelectedItem().toString() +
                             " - " + cant);
        total += cant;
        txtTotal.setText(String.valueOf(total));
      } catch (Exception ex) {
        // si ocurre una excepción, el elemento no se agrega
      }
      txtCant.setText("");
    }
  }
}
```

Código fuente HTML:

```
<HTML>
  <HEAD>  <TITLE>Applet "Inventario"</TITLE>  </HEAD>
  <BODY>
    <H1>Control de Inventario</H1>
    <APPLET CODE = "inventario.Inventario.class"
            HEIGHT = "240" WIDTH = "400"></APPLET>
  </BODY>
</HTML>
```

Resolución para Ejercicio 9

Código fuente Java:

```
import java.awt.event.*;
```

```
import javax.swing.*;
import java.awt.FlowLayout;
import java.awt.Dimension;

public class Listas extends JApplet {

  DefaultListModel datosOrigen;
  JList listaOrigen;
  JButton btnAgregarTodo;
  JButton btnAgregar;
  JButton btnQuitar;
  JButton btnQuitarTodo;
  DefaultListModel datosDestino;
  JList listaDestino;

  public void init() {
    // se inicializa el applet
    this.getContentPane().setLayout(new FlowLayout());
    // se crean los componentes
    datosOrigen = new DefaultListModel();
    listaOrigen = new JList(datosOrigen);
    btnAgregarTodo = new JButton("Agregar Todos");
    btnAgregar = new JButton("Agregar >>");
    btnQuitar = new JButton("<< Quitar");
    btnQuitarTodo = new JButton("Quitar Todos");
    datosDestino = new DefaultListModel();
    listaDestino = new JList(datosDestino);
    // se inicializan los componentes
    listaOrigen.setSelectionMode(
                            ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
    JScrollPane scrollOrigen = new JScrollPane(listaOrigen);
    scrollOrigen.setPreferredSize(new Dimension(200, 130));
    listaDestino.setSelectionMode(
                            ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
    JScrollPane scrollDestino = new JScrollPane(listaDestino);
    scrollDestino.setPreferredSize(new Dimension(200, 130));
    JPanel panelBotones = new JPanel();
    btnAgregar.setPreferredSize(btnAgregarTodo.getPreferredSize());
    btnQuitar.setPreferredSize(btnAgregarTodo.getPreferredSize());
    btnQuitarTodo.setPreferredSize(btnAgregarTodo.getPreferredSize());
    panelBotones.add(btnAgregarTodo);
    panelBotones.add(btnAgregar);
    panelBotones.add(btnQuitar);
    panelBotones.add(btnQuitarTodo);
    panelBotones.setPreferredSize(new Dimension(120, 130));
    // se agrega el escucha de eventos de acción
    ActionListener escucha = new BotonesAction();
    btnAgregarTodo.addActionListener(escucha);
    btnAgregar.addActionListener(escucha);
    btnQuitar.addActionListener(escucha);
    btnQuitarTodo.addActionListener(escucha);
    // se agregan los componentes al panel de contenidos del applet
    this.getContentPane().add(scrollOrigen);
    this.getContentPane().add(panelBotones);
    this.getContentPane().add(scrollDestino);
    // se llena la lista de origen
    llenarOrigen();
  }
  private void llenarOrigen() {
    datosOrigen.addElement("Agote, Julio");
    datosOrigen.addElement("Aguirre, Ester");
    datosOrigen.addElement("Barbero, Víctor");
```

```
    datosOrigen.addElement("Carreras, Dora");
    datosOrigen.addElement("García, Gustavo");
    datosOrigen.addElement("Maradona, Diego");
    datosOrigen.addElement("Pérez, Elvio");
    datosOrigen.addElement("Sánchez, Gonzalo");
    datosOrigen.addElement("Torres, Mariana");
    datosOrigen.addElement("Troncoso, Darío");
    datosOrigen.addElement("Vaca, Hugo");
    datosOrigen.addElement("Zapata, Andrea");
  }
  // manejo del evento de acción de los botones
  class BotonesAction implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      // botón  "Agregar Todo"
      if (e.getSource() == btnAgregarTodo) {
        Object [] elementos = new Object[20];
        datosOrigen.copyInto(elementos);
        for (int i = 0; i < elementos.length; i++) {
          if (elementos[i] != null) {
            datosDestino.addElement(elementos[i]);
          }
        }
        datosOrigen.clear();
      // botón  "Agregar >>"
      } else if (e.getSource() == btnAgregar) {
        int [] indices = listaOrigen.getSelectedIndices();
        for (int i = 0; i < indices.length; i++) {
          datosDestino.addElement(datosOrigen.remove(indices[i] - i));
        }
      // botón  "<< Quitar"
      } else if (e.getSource() == btnQuitar) {
        int [] indices = listaDestino.getSelectedIndices();
        for (int i = 0; i < indices.length; i++) {
          datosOrigen.addElement(datosDestino.remove(indices[i] - i));
        }
      // botón  "Quitar Todo"
      } else if (e.getSource() == btnQuitarTodo) {
        Object [] elementos = new Object[20];
        datosDestino.copyInto(elementos);
        for (int i = 0; i < elementos.length; i++) {
          if (elementos[i] != null) {
            datosOrigen.addElement(elementos[i]);
          }
        }
        datosDestino.clear();
      }
    }
  }
}
```

Código fuente HTML:

```
<HTML>
  <HEAD>  <TITLE>Applet Manejador de Listas</TITLE>  </HEAD>
  <BODY>
    <H1>Manejador de Listas</H1>
    <APPLET CODE = "Listas.class" WIDTH = "560" HEIGHT = "140"> </APPLET>
  </BODY>
</HTML>
```