PRÁCTICA 7

Ejercicio “Paint Básico 2D”

Descripción del ejercicio

El objetivo de esta práctica es realizar una versión mejorada del “Paint Básico” realizado en la práctica 4. En este caso, usaremos `Graphics2D` e incluiremos las siguientes nuevas funcionalidades:

- Entorno multiventana (i.e., la ventana principal tendrá un escritorio con ventanas internas, cada una de ella con su propio lienzo de dibujo).
- El lienzo mantendrá todas las figuras que se vayan dibujando
- Desplazamiento de las figuras previamente pintadas
- Grosor del trazo
- Transparencia
- Alisado de las formas

El aspecto visual de la aplicación será el mostrado en la Figura 1. En el escritorio se podrán tener tantas ventanas internas como se quiera, cada una de ellas con su propio lienzo de dibujo.

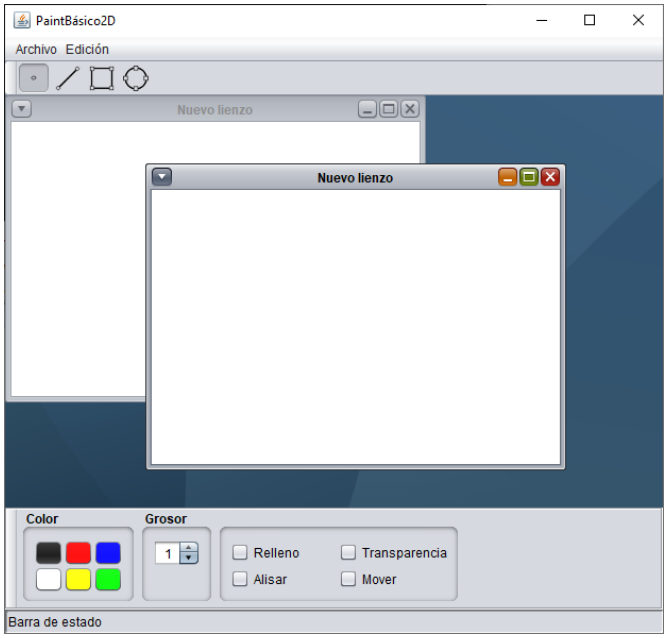


Figura 1: Aspecto de la aplicación

Al igual que en la práctica 4, el usuario podrá elegir entre cuatro formas de dibujo: **punto, línea, rectángulo y elipse**, que seleccionará mediante botones situados en la barra de herramientas (la barra de estado mostrará la forma activa). Además, habrá otra barra de herramientas (`JToolBar`) en la parte inferior para poder seleccionar los atributos de dibujo; concretamente, se podrá elegir entre un conjunto de **colores** predeterminados, el **grosor** del trazo (seleccionable mediante un *spinner*), si la forma está o no **rellena**, si se aplica o no **transparencia**[1] y si se **alisan** o no las formas en el renderizado (asociadas a estas tres últimas opciones, incluiremos tres casillas de validación).

[1] Entendida como semitransparencia correspondiente a un alfa 0.5

El lienzo de cada ventana interna mostrará **el conjunto de formas dibujadas** en ese lienzo (en este caso, y al contrario que en la práctica 4, hay que mantener las figuras dibujadas usando un vector de formas). Todas **las formas de un mismo lienzo se mostrarán con los mismos atributos** de color, grosor, relleno, transparencia y alisado (seleccionados en el panel inferior), de forma que un cambio en un atributo implicará el cambio de dicha propiedad en todas las formas dibujadas en dicho lienzo[2]. No obstante, dados dos lienzos distintos, correspondientes a dos ventanas internas distintas, los atributos de uno y otro lienzo pueden ser diferentes entre sí[3].

El usuario podrá **mover las figuras** que ya estén dibujadas. Para ello, se incluirá en la ventana principal una casilla de validación (`JCheckBox`), de forma que, si está seleccionada, indicará que la interacción del usuario será para mover[4] las figuras (en caso contrario, dicha interacción implicará la incorporación de nuevas formas).

En el menú se incluirán dos opciones: "Archivo" y "Edición". La primera tendrá a su vez tres opciones: "Nuevo", "Abrir" y "Guardar". La opción "Nuevo" deberá crear una nueva ventana interna con un lienzo vacío (sobre el que se podrá empezar a dibujar), mientras que "Abrir" y "Guardar" deberán lanzar el diálogo correspondiente (si bien no se abrirán ni guardarán archivos). El menú edición tendrá tres opciones: "Ver barra de estado", "Ver barra de formas" y "Ver barra de atributos" que activen/desactiven respectivamente la barra de estado, la de formas y la de atributos.

Algunas recomendaciones

1. Crear una biblioteca propia que contenga paquetes en los que incluir las clases de nueva creación que puedan ser necesarias en futuras prácticas. En particular, se recomienda incluir en esa biblioteca la clase correspondiente al lienzo y, si las hubiese, las clases de creación propia correspondientes a formas. Para ello:
   - Crear un proyecto en NetBeans de tipo "*Java Class Library*" de título *SM.XXX.Biblioteca*, con "XXX" las iniciales del estudiante
   - En el proyecto, dentro de la carpeta "Paquetes de fuentes", se crearán tantas subcarpetas como paquetes tenga nuestra biblioteca (en el menú contextual del proyecto, seleccionamos *Nuevo → Java Package*). En particular, se propone crear dos paquetes:
     - *sm.xxx.iu*, en la que se irán incluyendo componente y contendores de propósito general que puedan ser útiles a la hora de diseñar interfaces de usuario (por ejemplo la clase `Lienzo2D` que comentaremos posteriormente).
     - *sm.xxx.graficos*, en la que se incluirán clases de diseño propio relativas a gráficos (por ejemplo, aquellas correspondientes a clases de formas).
   - Una vez creada la biblioteca, hay que recordar incluirla en aquellos proyectos en los que vaya a utilizarse (a través de *Propiedades → Biblioteca → Añadir proyecto*). En particular, hay que añadirla en el proyecto de esta práctica 7.
2. Ya centrados en el proyecto NetBeans propio de la práctica 7, y para realizar un entorno multiventana, aplicaremos lo ya visto en la práctica 3:

---

[2] Esto simplifica notablemente la aplicación, ya que, si se considerase como requisito que cada forma mantuviese los atributos con los que se dibujó, implicaría la necesidad de nuevas clases que agruparan información geométrica y de atributos (recordemos con los objetos *Shape* sólo contienen datos geométricos).

[3] Es decir, en una ventana interna se podría estar dibujando en color rojo, mientras que en otra en color azul. Esto será así porque se asume que los atributos con los que se dibuja en un lienzo son propiedades de la instancia (es decir, cada lienzo tiene las suyas).

[4] Recordar que, como se indicó en la práctica 5, para mover líneas será necesario crearse una clase propia que herede de *Line2D* (más concretamente, de *Line2D.Float* o de *Line2D.Double*). Véase práctica 5 para más detalles.

- Anadir un escritorio en el centro de la ventana principal (donde se tuviese el lienzo en la práctica 4). Para ello, incorporar un `JDesktopPane` usando el NetBeans (área "contenedores swing").
- Crear una clase propia `VentanaInterna` que herede de `JInternalFrame` (para ello, usar NetBeans). Activar las propiedades "*closable*", "*iconifiable*", "*maximizable*" y "*resizable*".
- Para incorporar una ventana interna, hay que (i) crear el objeto, (ii) añadirlo al escritorio y (iii) mandar el mensaje para visualizarla. Así, por ejemplo, en el manejador del evento asociado a la opción "Nuevo", tendríamos:

```
private void menuNuevoActionPerformed(ActionEvent evt) {
  VentanaInterna vi = new VentanaInterna();
  escritorio.add(vi);
  vi.setVisible(true);
}
```

- Si en un momento dado queremos acceder a la ventana que esté activa en el escritorio (p.e., desde un método gestor de eventos), el siguiente código devuelve la ventana activa (`null` si no hay ninguna):

```
VentanaInterna vi;
vi = (VentanaInterna)escritorio.getSelectedFrame();
```

  A través de `vi` podemos acceder, por ejemplo, al lienzo de la ventana activa y a sus métodos y variables[5].

3. Al igual que en la práctica 4, para el área de dibujo se recomienda crear una clase propia `Lienzo2D` que herede de `JPanel` (que incluiremos en la librería que hemos creado).En este caso, el lienzo estará situado dentro de la ventana interna (y no en la principal). Dicha clase gestionará todo lo relativo al dibujo: vector de formas, atributos, método `paint`, gestión de eventos de ratón vinculados al proceso de dibujo, etc. En esta clase:

   - El método `paint` deberá contener código centrado sólo en el dibujo de formas (mediante llamadas a métodos `draw` y `fill`), no de creación de formas (objetos `Shape`) o de atributos (de no ser así, **se consideraría erróneo**) Por ejemplo:

```
public void paint(Graphics g){
  super.paint(g);
  Graphics2D g2d = (Graphics2D)g;
  g2d.setPaint(color);
  g2d.setStroke(stroke);
  for(Shape s:vShape) {
    if(relleno) g2d.fill(s);
    g2d.draw(s);
  }
}
```

   - Se recomienda definir métodos `createShape` y `updateShape` para la creación y modificación de formas; será dentro de dichos métodos donde se considerarán las diferentes casuísticas en función de la forma[6]. Estos métodos serán llamados

[5] Por ejemplo, si en la clase `VentanaInterna` tenemos definido un método `getLienzo()` que devuelva el lienzo asociado a dicha ventana, para acceder a él sería `vi.getLienzo()`

[6] En la práctica 5 no hacía falta distinguir entre unas formas u otras ya que sólo creábamos rectángulos (`r=new Rectangle(…)`) y los modificábamos (`r.setFrameFromDiagonal(…)`); ahora, el "*new*" dependerá de la forma a crear y los métodos para modificar de la clase correspondiente a la forma. Por ello se recomienda definir dos métodos `createShape` y `updateShape` que consideren las diferentes casuísticas en función de la forma (en un caso pare crear, en el otro para modificar); en particular, las sentencias `switch` (o `if-else`) que en la práctica 4 estaban localizadas en el método `paint`, ahora se encontrarán en este tipo de métodos.

desde los manejadores de eventos: el de creación desde el `mousePressed` y el de actualización desde el `mouseDragged` y `mouseReleased`.

En el caso del `updateShape`, dependiendo de la clase a la que pertenezca la forma habrá que invocar a un método u a otro. Por ejemplo, en el caso de la línea, vimos en la práctica 5 que para modificar sus puntos de inicio y final usábamos el método `setLine`; para el caso del rectángulo, usamos el método `setFrameFromDiagonal`. Por este motivo, es necesario preguntarle al objeto "a qué clase pertenece" para poder hacer el "casting" que permita invocar al método. Por ejemplo, el siguiente código podría ser el cuerpo del método `updateShape`:

```
if(s instanceof Line2D) ((Line2D)s).setLine(p1,p2);
else if(s instanceof RectangularShape)
  ((RectangularShape)s).setFrameFromDiagonal(p1, p2);
```

donde `s` es una variable tipo `Shape` (en este caso, la figura que se está pintando), y `p1` y `p2` los puntos extremos de la línea o de la diagonal del rectángulo/elipse. Nótese que el if/else no se hace en base a la figura seleccionada en la barra de herramientas, sino considerando la clase de la forma que estamos modificando.

- Al igual que en la práctica 5, se recomienda definir un método `getSelectedShape` para gestionar la selección de una forma. En dicho método se comprobará si el punto donde se ha hecho el clic está contenido dentro de alguna de las formas del lienzo[7].
- Para desplazar una forma, ya vimos en la práctica 5 que no existe un método en la clase `Shape` común para todas las formas (tipo `setLocation`); de hecho, en muchas clases no está definido un método específico que permita reubicar la forma (p.e., en la práctica 5 se explicó el caso de la `Line2D`) y hay que acudir a otro tipo de métodos (que dependen de la clase). Por ello, para esta práctica se recomienda definir un método `setLocation(Shape s, Point2D pos)` que considere las diferentes casuísticas en función de la forma[8]. En este método, al igual que pasaba en el `updateShape`, habrá que tener en cuenta a qué clase pertenece la figura para hacer el casting y llamar a uno u otro método.

Entrega del ejercicio

Esta práctica se incluye dentro de la evaluación de la asignatura. Por ello, antes del comienzo de la siguiente clase de prácticas, se deberá de entregar la versión final a través de PRADO. Concretamente, al comenzar la sesión de prácticas correspondiente a esta práctica 7, se activará una entrega en PRADO; dicha **entrega estará activa hasta el comienzo de la siguiente sesión de prácticas** (siguiente semana), no siendo necesario hacer una entrega parcial (solo la final).

En la entrega deberá incluirse un fichero comprimido (zip o rar) que contenga el proyecto (carpeta Neatbeans tanto de la biblioteca como de la aplicación) y el ejecutable[9] (.jar).

[7] En el caso de líneas, no sería "contenido" sino "cerca de". Para este caso, ha de considerarse lo explicado en la práctica 5.

[8] La solución correcta, desde el punto de vista del diseño, sería la creación de clases propias que solventaran las carencias de la clase `Shape` y subclases (como se apuntaba en la práctica 5 para el caso de la línea).

[9] Véase apéndice.

Posibles mejoras para trabajar en casa…

Una vez realizada la práctica, se proponen una serie de mejoras para darle mayor funcionalidad y mejorar el interfaz:

- Mejorar las barras de herramientas, simplificándolas de cara a próximas prácticas (téngase en cuenta que iremos ampliando nuestro entorno gráfico, incluyendo nuevas barras y funcionalidades, por lo que se aconseja hacer un diseño sencillo y práctico). Se recomienda un diseño en línea similar al siguiente[10]:

- Incluir descripciones emergentes ("*tooltips*")
- Cuando se cambie de una ventana interna a otra, hacer que los botones de forma y atributos de la ventana principal se activen conforme a la forma y atributos del correspondiente lienzo.
- Mostrar en la barra de estado las coordenadas del puntero al desplazarse sobre el lienzo[11].

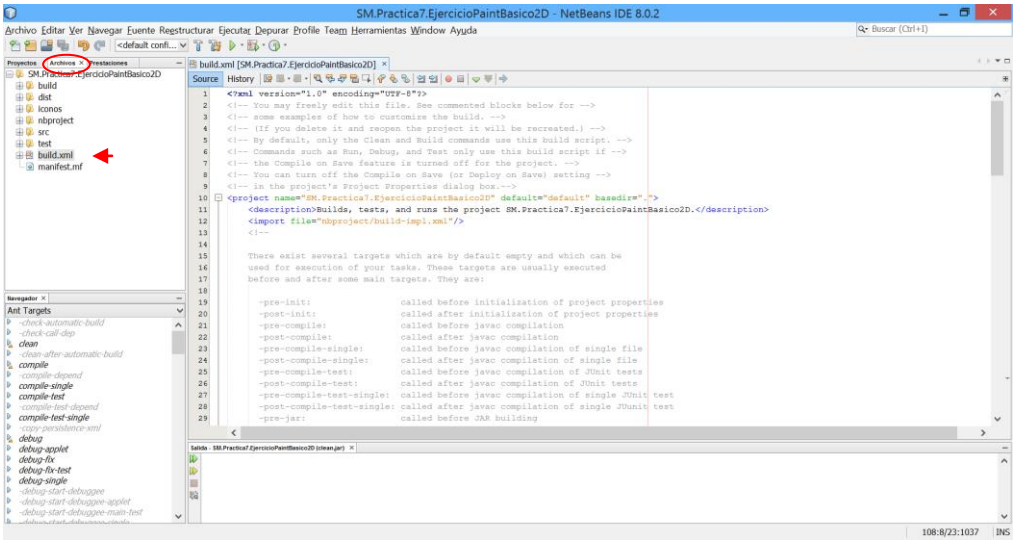
[10] En la web se incluyen los iconos mostrados en la figura. Para el caso de la lista desplegable, se aconseja leer cómo hacer celdas personalizadas mediante ListCellRender.

[11] La gestión de los eventos de movimiento de ratón no se hará desde la clase *Lienzo2D*, sino desde la ventana principal (o la interna). Téngase en cuenta que en este caso el lienzo es el generador.

Apéndice 1: Incluir todas las bibliotecas en un único JAR

Como sabemos, para un proyecto dado, NetBeans genera el correspondiente fichero *.jar* en la carpeta */dist*. No obstante, si el proyecto usa bibliotecas externas (como ocurre en esta práctica), éstas no se incluyen en el fichero *.jar*, sino que, en su lugar, crea una carpeta */lib* (dentro de */dist*) en la que incorpora todas estas bibliotecas. Esto implica que, para ejecutar el fichero *.jar*, tiene que estar siempre presente la carpeta[12] */lib* (haciendo más "engorrosa" la posible distribución de nuestro programa -véase el fichero *README.TXT* generado por NetBeans-). Si quisiéramos generar un *.jar* que empaquetase todas las bibliotecas en un único fichero, habría que hacer lo siguiente:

1. Nos vamos a la sección "Archivos" (junto a "Proyectos", en el panel superior izquierdo) y seleccionamos el fichero *build.xml*:



   Como vemos, hay muy poco código (la mayoría del fichero es una sección comentada que explica cómo ampliarlo)

2. Incorporamos el siguiente código XML al final del archivo *build.xml* (antes del *tag* de cierre *</project>*):

```
<target name="-post-jar">
  <property name="pack.jar" value="dist/${application.title}.pack.jar"/>
  <echo message="Packaging into a single JAR at ${pack.jar}"/>
  <jar jarfile="${pack.jar}">
    <zipfileset src="${dist.jar}" excludes="META-INF/*" />
    <zipgroupfileset dir="dist/lib" includes="*.jar" excludes="META-INF/*"/>
    <manifest>
      <attribute name="Main-Class" value="${main.class}"/>
    </manifest>
  </jar>
</target>
```

3. Ahora, al "Limpiar y construir" nuestro proyecto, además del *.jar* que se obtenía antes, se generará otro fichero *.pack.jar* que incluirá todas las bibliotecas y, por lo tanto, se podrá ejecutar de forma autónoma sin necesidad de estar junto a la carpeta */lib*. En caso de tener que distribuir vuestro programa, usad este fichero.

[12] Si no está la carpeta */lib*, no tendrá acceso a las clases de las bibliotecas y lanzará excepciones cuando trate de usarlas. Para poder ver el trazado de estas excepciones, habrá que ejecutar nuestra aplicación desde una ventana de comandos. Por este motivo, se aconseja que antes de distribuir una aplicación a un cliente final, ésta se ejecute desde una ventana de comandos para asegurarnos que no se lanzan excepciones de inicio.

Apéndice 2: Definir eventos propios

En ocasiones es necesario que objetos que son instancias de clases propias (definidas por nosotros) notifiquen sucesos específicos de su clase (por ejemplo, en el caso del lienzo, notificar que se ha añadido una nueva figura o que se ha cambiado alguna propiedad). En estos casos, los eventos estándar de Java no nos sirven, siendo necesario definir (1) clases propias de eventos, así como (2) los interfaces asociados que deberán de implementar los manejadores del evento; además, (3) habrá que incorporar la capacidad de generación de eventos en nuestra clase.

A continuación se muestra un ejemplo en el que se define una clase evento propia para la clase `Lienzo` desarrollada en esta práctica, así como el interface del manejador (se aconseja definirlas en nuestra biblioteca dentro de un paquete específico *sm.xxx.eventos*).

• Definición de la clase *LienzoEvent*

En primer lugar definimos la clase `LienzoEvent` que representará un evento lanzado por un objeto `Lienzo`. Dicha clase deberá de heredar de `EventObject` y tendrá definidas variables miembro para guardar la información asociada a dicho evento. Por ejemplo:

```
public class LienzoEvent extends EventObject{

    private Shape forma;
    private Color color;

    public LienzoEvent(Object source, Shape forma, Color color) {
        super(source);
        this.forma = forma;
        this.color = color;
    }
    public Shape getForma() {
        return forma;
    }
    public Color getColor() {
        return color;
    }
}
```

representa un evento que llevará asociada como información una figura y un color.

• Definición de la interface *LienzoListener*

A continuación definimos el interface `LienzoListener` que deberán cumplir los manejadores de eventos. Dicha interface deberá de heredar de `EventListener` y tendrá definidos tantos métodos como "motivos" puedan generar ese evento. Por ejemplo:

```
public interface LienzoListener extends EventListener{
    public void shapeAdded(LienzoEvent evt);
    public void propertyChange(LienzoEvent evt);
}
```

que indica que todo manejador tipo `LienzoListener` deberá de implementar dos métodos, uno asociado a añadir una nueva figura al lienzo, y otro a modificar una propiedad (p.e., el color). Siguiendo la costumbre de Java, podemos definir el "adapter" asociado[13]:

```
public class LienzoAdapter implements LienzoListener{
    public void shapeAdded(LienzoEvent evt){}
    public void propertyChange(LienzoEvent evt){}
}
```

[13] Un manejador de eventos lienzo podrá definirse (1) implementando el interface `LienzoListener` o (2) heredando de `LienzoAdapter` y sobrecargando el método que nos interese.

• Incorporando en *Lienzo* la capacidad de lanzar eventos

Una vez creadas las clases anteriores, surge la siguiente pregunta: ¿cómo hacemos que el lienzo lance los eventos? Recordemos que, en este caso, el lienzo hace las veces de generador. Además, ¿cómo le asociamos manejadores al lienzo? Para abordar los puntos anteriores habrá que añadir las siguientes funcionalidades a la clase `Lienzo`:

1. Declarar como variable miembro de la clase `Lienzo` una lista de elementos `LienzoListener`. Dicha lista tendrá almacenados los manejadores asociados al lienzo:

```
ArrayList<LienzoListener> lienzoEventListeners = new ArrayList();
```

2. Añadir a la clase `Lienzo` un método `addLienzoListener` que permita añadir manejadores a la lista

```
public void addLienzoListener(LienzoListener listener){
  if (listener != null) {
     lienzoEventListeners.add(listener);
  }
}
```

3. Desde el lienzo habrá que notificar a los manejadores asociados el lanzamiento de un determinado evento; esto implicará mandar el correspondiente mensaje (según motivo que lo haya ocasionado) a cada uno de los manejadores de la lista anterior (por ejemplo, mandarle el mensaje `shapeAdded` cuando se añada un elemento nuevo en la lista de figuras).

   Para simplificar esta tarea, se aconseja añadir a la clase `Lienzo` métodos para notificar a los manejadores asociados el lanzamiento de un determinado evento; concretamente, se aconseja definir uno por cada "motivo" que puede ocasionar el evento:

```
private void notifyShapeAddedEvent(LienzoEvent evt) {
  if (!lienzoEventListeners.isEmpty()) {
    for (LienzoListener listener : lienzoEventListeners) {
      listener.shapeAdded(evt);
    }
  }
}

private void notifyPropertyChangeEvent(LienzoEvent evt) {
  if (!lienzoEventListeners.isEmpty()) {
    for (LienzoListener listener : lienzoEventListeners) {
       listener.propertyChange(evt);
    }
  }
}
```

4. A los métodos definidos en el punto anterior los llamaremos cuando queramos que se lance el evento. Esto se hará desde otros métodos de la clase, según se cumpla o no el motivo que genera el evento. Por ejemplo, el evento que notifique que se ha añadido una nueva figura debería llamarse después de añadir el correspondiente elemento a la lista de figuras:

```
vShape.add(...);
notifyShapeAddedEvent( new LienzoEvent(this,s,color) );
```

   donde `s` sería la figura (objeto `Shape`) añadido y `color` el color activo en el lienzo en ese momento.

- **Y no olvidar…**

Recordar que, como vimos en la práctica 2, para manejar eventos es necesario (1) definir la clase manejadora, en este caso implementando el interface *LienzoListener*, (2) crear el objeto manejador y (3) enlazar el manejador con el generador, en este caso con el lienzo mediante la llamada a *addLienzoListener*.

Para los eventos estándar (*MouseEvent*, *KeyEvent*, etc.), NetBeans realiza las tres tareas anteriores de una forma sencilla; sin embargo, en el caso de eventos propios, no contamos con NetBeans para "hacerlo por nosotros", así que habrá que incluir el código por nuestra cuenta (que, recordemos, es lo habitual en programación avanzada). Por ejemplo, si queremos en nuestra *VentanaPrincipal* muestre un mensaje (en consola o en la barra de estado) cada vez que se añada una figura en algún lienzo, en primer lugar tendremos que definir la clase manejadora[14]:

```
public class MiManejadorLienzo extends LienzoAdapter{
    public void shapeAdded(LienzoEvent evt){
       System.out.println("Figura "+evt.getForma()+" añadida");
    }
}
```

y posteriormente crear un objeto de la clase anterior y asociarlo con el correspondiente manejador:

```
MiManejadorLienzo manejador = new MiManejadorLienzo();
lienzo.addLienzoListener(manejador);
```

donde *lienzo* es una referencia al lienzo que queremos manejar (por ejemplo, el de una ventana interna al que accedemos mediante *vi.getLienzo()*)

[14] En este ejemplo, se podría declarar como una clase interna de *VentanaPrincipal*.