13: Crear ventanas y applets

Una guía de diseño fundamental es "haz las cosas simples de forma sencilla, y las cosas difíciles hazlas posibles."[1]

La meta original de diseño de la biblioteca de interfaz gráfico de usuario (IGU) en Java 1.0 era permitir al programador construir un IGU que tuviera buen aspecto en todas las plataformas. Esa meta no se logró. En su lugar, el *Abstract Window Toolkit* (AWT) de Java 1.0 produce un IGU que tiene una apariencia igualmente mediocre en todos los sistemas. Además, es restrictivo: sólo se pueden usar cuatro fuentes y no se puede acceder a ninguno de los elementos de IGU más sofisticados que existen en el sistema operativo. El modelo de programación de AWT de Java 1.0 también es siniestro y no es orientado a objetos. En uno de mis seminarios, un estudiante (que había estado en Sun durante la creación de Java) explicó el porqué de esto: el AWT original había sido conceptualizado, diseñado e implementado en un mes. Ciertamente es una maravilla de la productividad, además de una lección de por qué el diseño es importante.

La situación mejoró con el modelo de eventos del AWT de Java 1.1, que toma un enfoque orientado a objetos mucho más claro, junto con la adición de JavaBeans, un modelo de programación basado en componentes orientado hacia la creación sencilla de entornos de programación visuales. Java 2 acaba esta transformación alejándose del AWT de Java 1.0 esencialmente, reemplazando todo con las *Java Foundation Classes* (JFC), cuya parte IGU se denomina "Swing". Se trata de un conjunto de JavaBeans fáciles de usar, y fáciles de entender que pueden ser arrastrados y depositados (además de programados a mano) para crear un IGU con el que uno se encuentre finalmente satisfecho. La regla de la "revisión 3" de la industria del software (un producto no es bueno hasta su tercera revisión) parece cumplirse también para los lenguajes de programación.

Este capítulo no cubre toda la moderna biblioteca Swing de Java 2, y asume razonablemente que Swing es la biblioteca IGU destino final de Java. Si por alguna razón fuera necesario hacer uso del "viejo" AWT (porque se está intentando dar soporte a código antiguo o se tienen limitaciones impuestas por el navegador), es posible hacer uso de la introducción que había en la primera edición de este libro, descargable de *http://www.BruceEckel.com* (incluida también en el CD ROM que se adjunta a este libro).

Al principio de este capítulo veremos cómo las cosas son distintas si se trata de crear un *applet* o si se trata de crear una aplicación ordinaria haciendo uso de Swing, y cómo crear programas que son tanto *applets* como aplicaciones, de forma que se pueden ejecutar bien dentro de un *browser*, o bien desde línea de comandos. Casi todos los ejemplos IGU de este libro serán ejecutables bien como *applet*, o como aplicaciones.

---

[1] Hay una variación de este dicho que se llama "el principio de asombrarse al mínimo", que dice en su esencia: "No sorprenda al usuario".

Hay que ser conscientes de que este capítulo no es un vocabulario exhaustivo de los componentes Swing o de los métodos de las clases descritas. Todo lo que aquí se presenta es simple a propósito. La biblioteca Swing es vasta y la meta de este capítulo es sólo introducirnos con la parte esencial y más agradable de los conceptos. Si se desea hacer más, Swing puede proporcionar lo que uno desee siempre que uno se enfrente a investigarlo.

Asumimos que se ha descargado e instalado la documentación HTML de las bibliotecas de Java (que es gratis) de *http://java.sun.com* y que se navegará a lo largo de las clases **javax.swing** de esa documentación para ver los detalles completos y los métodos de la biblioteca Swing. Debido a la simplicidad del diseño Swing, generalmente esto será suficiente información para solucionar todos los problemas. Hay numerosos libros (y bastante voluminosos) dedicados exclusivamente a Swing y son altamente recomendables si se desea mayor nivel de profundidad, o cuando se desee cambiar el comportamiento por defecto de la biblioteca Swing.

A medida que se aprendan más cosas sobre Swing se descubrirá que:

1. Swing es un modelo de programación mucho mejor que lo que se haya visto probablemente en otros lenguajes y entornos de desarrollo. Los JavaBeans (que no se presentarán hasta el final de este capítulo) constituyen el marco de esa biblioteca.

2. Los "constructores IGU" (entornos de programación visual) son un aspecto *de rigueur* de un entorno de desarrollo Java completo. Los JavaBeans y Swing permiten al constructor IGU escribir código por nosotros a medida que se ubican componentes dentro de formularios utilizando herramientas gráficas. Esto no sólo acelera rápidamente el desarrollo utilizando construcción IGU, sino que permite un nivel de experimentación mayor, y por consiguiente la habilidad de probar más diseños y acabar generalmente con uno mejor.

3. La simplicidad y la tan bien diseñada naturaleza de Swing significan que incluso si se usa un constructor IGU en vez de codificar a mano, el código resultante será más completo —esto soluciona un gran problema con los constructores IGU del pasado, que podían generar de forma sencilla código ilegible.

Swing contiene todos los componentes que uno espera ver en un IU moderno, desde botones con dibujos hasta árboles y tablas. Es una gran biblioteca, pero está diseñada para tener la complejidad apropiada para la tarea a realizar —es algo simple, no hay que escribir mucho código, pero a medida que se intentan hacer cosas más complejas, el código se vuelve proporcionalmente más complejo. Esto significa que nos encontramos ante un punto de entrada sencillo, pero se tiene a mano toda la potencia necesaria.

A mucho de lo que a uno le gustaría de Swing se le podría denominar "ortogonalidad de uso". Es decir, una vez que se captan las ideas generales de la biblioteca, se pueden aplicar en todas partes. En primer lugar, gracias a las convenciones estándar de denominación, muchas de las veces, mientras se escriben estos ejemplos, se pueden adivinar los nombres de los métodos y hacerlo bien a la primera, sin investigar nada más. Ciertamente éste es el sello de un buen diseño de biblioteca. Además, generalmente se pueden insertar componentes a los componentes ya existentes de forma que todo funcione correctamente.

En lo que a velocidad se refiere, todos los componentes son "ligeros", y Swing se ha escrito completamente en Java con el propósito de lograr portabilidad.

La navegación mediante teclado es automática —se puede ejecutar una aplicación Java sin usar el ratón, y no es necesaria programación extra. También se soporta desplazamiento sin esfuerzo —simplemente se envuelven los componentes en un **JScrollPane** a medida que se añaden al formulario. Aspectos como etiquetas de aviso simplemente requieren de una línea de código.

Swing también soporta una faceta bastante más radical denominada "*apariencia* conectable" que significa que se puede cambiar dinámicamente la apariencia del IU para que se adapte a las expectativas de los usuarios que trabajen en plataformas y sistemas operativos distintos. Incluso es posible (aunque difícil) inventar una apariencia propia.

El applet básico

Una de las metas de diseño de Java es la creación de *applets*, que son pequeños programas que se ejecutan dentro del navegador web. Dado que tienen que ser seguros, se limitan a lo que pueden lograr. Sin embargo, los *applets* son una herramienta potente que soporta programación en el lado cliente, uno de los aspectos fundamentales de la Web.

Restricciones de applets

La programación dentro de un *applet* es tan restrictiva que a menudo se dice que se está "dentro de una caja de arena" puesto que siempre se tiene a alguien —es decir, el sistema de seguridad de tiempo de ejecución de Java— vigilando.

Sin embargo, uno también se puede salir de la caja de arena y escribir aplicaciones normales en vez de *applets*, en cuyo caso se puede acceder a otras facetas del S.O. Hemos estado escribiendo aplicaciones normales a lo largo de todo este libro, pero han sido *aplicaciones de consola* sin componentes gráficos. También se puede usar Swing para construir interfaces IGU para aplicaciones normales.

Generalmente se puede responder a la pregunta de qué es lo que un *applet* puede hacer mirando a lo que *se supone* que hace: extender la funcionalidad de una página web dentro de un navegador. Puesto que, como navegador de la Red, nunca se sabe si una página web proviene de un sitio amigo o no, se desea que todo código que se ejecute sea seguro. Por tanto, las mayores restricciones que hay que tener en cuenta son probablemente:

1. *Un applet no puede tocar el disco local.* Esto significa escribir *o* leer, puesto que no se desearía que un *applet* pudiera leer y transmitir información privada a través de Internet sin permiso. Se evita la escritura, por supuesto, dado que supondría una invitación abierta a los virus. Java ofrece *firmas digitales* para *applets*. Muchas restricciones de *applets* se suavizan cuando se elige la ejecución de *applets de confianza* (los firmados por una fuente de confianza) para tener acceso a la máquina.

2. *Puede llevar más tiempo mostrar los applets*, puesto que hay que descargarlos por completo cada vez, incluyendo una solicitud distinta al servidor por cada clase diferente. El navegador puede introducir el *applet* en la caché, pero no hay ninguna garantía de que esto ocurra. Debido a esto, probablemente habría que empaquetar los *applets* en un JAR (Java ARchivo) que combina todos los componentes del *applet* (incluso otros archivos **.class** además de imágenes y sonidos) dentro de un único archivo comprimido que puede descargarse en una única transacción servidora. El "firmado digital" está disponible para cada entrada digital de un archivo JAR.

Ventajas de los *applets*

Si se puede vivir con las restricciones, los *applets* tienen también ventajas significativas cuando se construyen aplicaciones cliente/servidor u otras aplicaciones en red:

1. *No hay aspectos de instalación*. Un *applet* tiene independencia completa de la plataforma (incluyendo la habilidad de reproducir sin problemas archivos de sonido, etc.) por lo que no hay que hacer ningún cambio en el código en función de la plataforma ni hay que llevar a cabo ninguna "adaptación" en el momento de la instalación. De hecho, la instalación es automática cada vez que el usuario carga la página web que contiene *applets*, de forma que las actualizaciones se darán de forma inadvertida y automáticamente. En los sistemas cliente/servidor tradicionales, construir e instalar nuevas versiones del software cliente es siempre una auténtica pesadilla.

2. *No hay que preocuparse de que el código erróneo cause ningún mal a los sistemas de alguien*, gracias a la propia seguridad implícita en la esencia de Java y en la estructura de los *applets*. Esto, junto con el punto anterior, convierte a Java en un lenguaje popular para las denominadas aplicaciones cliente/servidor de *intranet* que sólo residen dentro de un campo de operación restringido dentro de una compañía donde se puede especificar y/o controlar el entorno del usuario (el navegador web y sus añadidos).

Debido a que los *applets* se integran automáticamente con el HTML, hay que incluir un sistema de documentación embebido independiente de la plataforma para dar soporte al *applet*. Se trata de un problema interesante, puesto que uno suele estar acostumbrado a que la documentación sea parte del programa en vez de al revés.

Marcos de trabajo de aplicación

Las bibliotecas se agrupan generalmente dependiendo de su funcionalidad. Algunas bibliotecas, por ejemplo, se usan como tales, independientemente. Las clases **String** y **ArrayList** de la biblioteca estándar de Java son ejemplos de esto. Otras bibliotecas están diseñadas específicamente como bloques que permiten construir otras clases. Cierta categoría de biblioteca es el *marco de trabajo de aplicación*, cuya meta es ayudar en la construcción de aplicaciones proporcionando una clase o un conjunto de clases que producen el comportamiento básico deseado para toda aplicación de un tipo particular. Posteriormente, para adaptar el comportamiento a nuestras propias necesidades, hay que heredar de la clase aplicación y superponer los métodos que interesen. El marco de trabajo de apli-

cación es un buen ejemplo de "separar las cosas que cambian de las que permanecen invariables", puesto que intenta localizar todas las partes únicas de un programa en los métodos superpuestos[2].

Los *applets* se construyen utilizando un marco de trabajo de aplicación. Se hereda de **JApplet** y se superponen los métodos apropiados. Hay unos pocos métodos que controlan la creación y ejecución de un *applet* en una página web:

Método	Operación
init( )	Se invoca automáticamente para lograr la primera inicialización del *applet*, incluyendo la disposición de los componentes. Este método siempre se superpone.
start( )	Se invoca cada vez que se visualiza un *applet* en el navegador para permitirle empezar sus operaciones normales (especialmente las que se apagan con **stop( )**).También se invoca tras **init( )**.
stop( )	Se invoca cada vez que un *applet* se aparta de la vista de un navegador web para permitir al *applet* apagar operaciones caras. Se invoca también inmediatamente antes de **destroy( )**.
destroy( )	Se invoca cada vez que se está descargando un *applet* de una página para llevar a cabo la liberación final de recursos cuando se deja de usar el *applet*.

Con esta información ya se puede crear un *applet* simple:

```
//: c13:Applet1.java
// Un applet muy simple.
import javax.swing.*;
import java.awt.*;

public class Applet1 extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("¡Applet!"));
  }
} ///:~
```

Nótese que no se exige a los *applets* tener un método **main( )**. Todo se incluye en el marco de trabajo de aplicación; el código de arranque se pone en el **init( )**.

En este programa, la única actividad que se hace es poner una etiqueta de texto en el *applet*, vía la clase **JLabel** (la vieja AWT se apropió del nombre **Label** además de otros nombres de componen-

[2] Éste es un ejemplo del patrón de diseño denominado *método plantilla*.

tes, por lo que es habitual ver que los componentes Swing empiezan por una "**J**"). El constructor de esta clase toma un **String** y lo usa para crear la etiqueta. En el programa de arriba se coloca esta etiqueta en el formulario.

El método **init( )** es el responsable de poner todos los componentes en el formulario haciendo uso del método **add( )**. Se podría pensar que debería ser posible invocar simplemente a **add( )** por sí mismo, y de hecho así solía ser en el antiguo AWT. Sin embargo, Swing requiere que se añadan todos los componentes al "panel de contenido" de un formulario, y por tanto hay que invocar a **getContentPane( )** como parte del proceso **add( )**.

Ejecutar applets dentro de un navegador web

Para ejecutar este programa, hay que ubicarlo dentro de una página web y ver esa página dentro de un navegador web con Java habilitado. Para ubicar un *applet* dentro de una página web se pone una etiqueta especial dentro de la fuente HTML de esa página web para indicar a la misma[3] cómo cargar y ejecutar el *applet*.

Este proceso era muy simple cuando Java en sí era simple y todo el mundo estaba en la misma línea e incorporaba el mismo soporte Java en sus navegadores web. Se podría haber continuado simplemente con un fragmento muy pequeño de código HTML dentro de la página web, así:

```
<applet code=Applet1 width=100 height=50>
</applet>
```

Después vinieron las guerras de navegadores y lenguajes y perdimos nosotros (los programadores y los usuarios finales). Tras cierto periodo de tiempo, JavaSoft se dio cuenta de que no podía esperar a que los navegadores siguieran soportando el buen gusto de Java, y la única solución era proporcionar algún tipo de añadido que se relacionara con el mecanismo de extensión del navegador. Utilizando el mecanismo de extensión (que los vendedores de navegadores no pueden deshabilitar —en un intento de ganar ventaja competitiva— sin romper con todas las extensiones de terceros), JavaSoft garantiza que un vendedor antagonista no pueda arrancar Java de su navegador web.

Con Internet Explorer, el mecanismo de extensión es el control ActiveX, y con Netscape, los *plug-ins*. He aquí el aspecto que tiene la página HTML más sencilla para **Applet1:**[4]

```
//:! c13:Applet1.html
<html><head><title>Applet1</title></head><hr>
<OBJECT
  classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
  width="100" height="50" align="baseline"
```

[3] Se asume que el lector está familiarizado con lo básico de HTML. No es demasiado difícil, y hay cientos de libros y otros recursos.

[4] Esta página —en particular la porción "clsid"— parecía funcionar bien tanto con JDK 1.2.2 como JDK 1.3 rc-1. Sin embargo, puede ser que en el futuro haya que cambiar algo la etiqueta. Pueden encontrar detalles al respecto en *java.sun.com*.

```
codebase="http://java.sun.com/products/plugin/1.2.2/jinstall-1_2_2-
win.cab#Version=1,2,2,0">
<PARAM NAME="code" VALUE="Applet1.class">
<PARAM NAME="codebase" VALUE=".">
<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2.2">
<COMMENT>
  <EMBED type=
    "application/x-java-applet;version=1.2.2"
    width="200" height="200" align="baseline"
    code="Applet1.class" codebase="."
pluginspage="http://java.sun.com/products/plugin/1.2/plugin-
install.html">
  <NOEMBED>
</COMMENT>
   No Java 2 support for APPLET!!
  </NOEMBED>
</EMBED>
</OBJECT>
<hr></body></html>
///:~
```

Algunas de estas líneas eran demasiado largas por lo que fue necesario envolverlas para que encajaran en la página. El código del código fuente de este libro (que se encuentra en el CD ROM de este libro, y se puede descargar de *http://www.BruceEckel.com*) funcionará sin que haya que preocuparse de corregir estos envoltorios de líneas.

El valor **code** da el nombre del archivo **.class** en el que reside el *applet*. Los valores **width** y **height** especifican el tamaño inicial del *applet* (en píxeles, como antiguamente). Hay otros elementos que se pueden ubicar dentro de la etiqueta *applet*: un lugar en el que encontrar otros archivos **.class** en Internet (**codebase**), información de alineación (**align**), un identificador especial que posibilita la intercomunicación de los *applets* (**name**), y parámetros de *applet* que proporcionan información sobre lo que ese *applet* puede recuperar. Los parámetros son de la forma:

```
<param name="identificador" value = "información">
```

y puede haber tantos como uno desee.

El paquete de código fuente de este libro proporciona una página HTML por cada *applet* de este libro, y por consiguiente muchos ejemplos de la etiqueta *applet*. Se pueden encontrar descripciones completas de los detalles de ubicación de los *applets* en las páginas web de *http://java.sun.com*.

Utilizar *Appletviewer*

El JDK de Sun (descargable gratuitamente de *http://www.sun.com*) contiene una herramienta denominada el *Appletviewer* que extrae las etiquetas <**applet**> del archivo HTML y ejecuta los *applets* sin

mostrar el texto HTML que les rodea. Debido a que el *Appletviewer* ignora todo menos las etiquetas APPLET, se puede poner esas etiquetas en el archivo fuente Java como comentarios:

```
// <applet code=MiApplet width=200 height=100>
// </applet>
```

De esta forma, se puede ejecutar "**appletviewer MiApplet.java**" y no hay que crear los pequeños archivos HTML para ejecutar pruebas. Por ejemplo, se pueden añadir las etiquetas HTML comentadas a **Applet1.java**:

```
//: c13:Applet1b.java
// Embebiendo la etiqueta applet para Appletviewer.
// <applet code=Applet1b width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;

public class Applet1b extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("¡Applet!"));
  }
} ///:~
```

Ahora, se puede invocar al *applet* con el comando:

```
appletviewer Applet1b.java
```

En este libro, se usará esta forma para probar los *applets* de manera sencilla. En breve, se verá otro enfoque de codificación que permite ejecutar *applets* desde la línea de comandos sin el *Appletviewer*.

Probar applets

Se puede llevar a cabo una prueba sencilla sin tener ninguna conexión de red, arrancando el navegador web y abriendo el archivo HTML que contiene la etiqueta *applet*. Al cargar el archivo HTML, el navegador descubrirá la etiqueta *applet* y tratará de acceder al archivo **.class** especificado por el valor **code**. Por supuesto, miremos primero la variable CLASSPATH para saber dónde buscar, y si el archivo **.class** no está en el CLASSPATH aparecerá un mensaje de error en la línea de estado del navegador indicando que no se puede encontrar ese archivo **.class**.

Cuando se desea probar esto en un sitio web, las cosas son algo más complicadas. En primer lugar, hay que *tener* un sitio web, lo que para la gran mayoría de la gente es un Proveedor de Servicios de Internet[5], un tercero, en una ubicación remota. Puesto que el *applet* es simplemente un archivo o un conjunto de archivos, el ISP no tiene que proporcionar ningún soporte especial para Java. También

[5] N. del traductor: En inglés. *Internet Service Provider* o *ISP*.

hay que tener alguna forma de mover los archivos HTML y **.class** desde un sistema al directorio correcto de la máquina ISP. Esto se suele hacer con un programa de FTP (File Transfer Protocol), de los que existen muchísimos disponibles gratuitamente o *shareware*. Por tanto, podría parecer que simplemente hay que mover los archivos a la máquina ISP con FTP, después conectarse al sitio y solicitar el archivo HTML utilizando el navegador; si aparece el *applet* y funciona, entonces todo probado, ¿verdad?

He aquí donde uno puede equivocarse. Si el navegador de la máquina cliente no puede localizar el archivo **.class** en el servidor, buscará en el CLASSPATH de la máquina *local*. Por consiguiente, podría ocurrir que no se esté cargando adecuadamente el *applet* desde el servidor, pero todo parece ir bien durante la prueba porque el navegador lo encuentra en la máquina local. Sin embargo, al conectarse otro, puede que su navegador no lo encuentre. Por tanto, al probar hay que asegurarse de borrar los archivos **.class** relevantes (o el archivo **.jar**) de la máquina local para poder verificar que existan en la ubicación adecuada dentro del servidor.

Uno de los lugares más insidiosos en los que me ocurrió esto es cuando inocentemente ubiqué un *applet* dentro de un **package**. Tras ubicar el archivo HTML y el *applet* en el servidor, resultó que el servidor no encontraba la trayectoria al *applet* debido al nombre del paquete. Sin embargo, mi navegador lo encontró en el CLASSPATH local. Por tanto, yo era el único que podía cargar el *applet* adecuadamente. Me llevó bastante tiempo descubrir que el problema era la sentencia **package**. En general, es recomendable no incorporar sentencias **package** con los *applets*.

Ejecutar *applets* desde la línea de comandos

Hay veces en las que se desearía hacer un programa con ventanas para algo más que para que éste resida en una página web. Quizás también se desearía hacer alguna de estas cosas en una aplicación "normal" a la vez que seguir disponiendo de la portabilidad instantánea de Java. En los capítulos anteriores de este libro hemos construido aplicaciones de línea de comandos, pero en algunos entornos operativos (como Macintosh, por ejemplo) no hay línea de comandos. Por tanto, por muchas razones se suele desear construir programas con ventanas pero sin *applets* haciendo uso de Java. Éste es verdaderamente un deseo muy codiciado.

La biblioteca Swing permite construir aplicaciones que preserven la apariencia del entorno operativo subyacente. Si se desea construir aplicaciones con ventanas, tiene sentido hacerlo[6] sólo si se puede hacer uso de la última versión de Java y herramientas asociadas de forma que se puedan entregar las aplicaciones sin necesidad de liar a los usuarios. Si por alguna razón uno se ve forzado a usar versiones antiguas de Java, que se lo piense bien antes de acometer la construcción de una aplicación con ventanas, especialmente, si tiene un tamaño mediano o grande.

---

[6] En mi opinión, y después de aprender Swing, no se deseará perder el tiempo con versiones anteriores.

A menudo se deseará ser capaz de crear clases que puedan invocarse bien como ventanas o bien como *applets*. Esto es especialmente apropiado cuando se prueban los *applets*, pues es generalmente mucho más rápido y sencillo ejecutar la aplicación-*applet* resultante desde la línea de comandos que arrancar el navegador web o el Appletviewer.

Para crear un *applet* que pueda ejecutarse desde la línea de comandos de la consola, simplemente hay que añadir un método **main( )** al *applet* que construya una instancia del *applet* dentro de un **JFrame**[7]. Como ejemplo sencillo, he aquí **Applet1b.java** modificado para que funcione como *applet* y como aplicación:

```
//: c13:Applet1c.java
// Un applet y una aplicación.
// <applet code=Applet1c width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Applet1c extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("¡Applet!"));
  }
  // Un main() para la aplicación:
  public static void main(String[] args) {
    JApplet applet = new Applet1c();
    JFrame frame = new JFrame("Applet1c");
    // Para cerrar la aplicación:
    Console.setupClosing(frame);
    frame.getContentPane().add(applet);
    frame.setSize(100,50);
    applet.init();
    applet.start();
    frame.setVisible(true);
  }
} ///:~
```

Sólo se ha añadido al *applet* el método **main( )**, y el resto permanece igual. Se crea el *applet* y se añade a un **JFrame** para que pueda ser mostrado.

La línea:

```
Console.setupClosing(frame);
```

[7] Como se describió anteriormente, el AWT ya tenía el término "Frame" por lo que Swing usa el nombre JFrame.

hace que se cierre convenientemente la ventana. **Console** proviene de **com.bruceekcel.swing** y será explicada algo más tarde.

Se puede ver que en el método **main( )** se minimiza explícitamente el *applet* y se arranca, puesto que en este caso no tenemos un navegador que lo haga automáticamente. Por supuesto, así no se logra totalmente el comportamiento del navegador, que también llama a **stop( )** y **destroy( )**, pero es aceptable en la mayoría de situaciones. Si esto constituye un problema es posible forzar uno mismo estas llamadas[8].

Nótese la última línea:

```
frame.setVisible(true);
```

Sin ella, no se vería nada en pantalla.

Un marco de trabajo de visualización

Aunque el código que convierte programas en *applets* y aplicaciones a la vez produce resultados de gran valor, si se usa en todas partes, se convierte en una distracción y un derroche de papel. En vez de esto, se usará el siguiente marco de trabajo de visualización para los ejemplos Swing de todo el resto del libro:

```
//: com:bruceeckel:swing:Console.java
// Herramienta para ejecutar demos Swing desde la
// consola, tanto applets como JFrames.
package com.bruceeckel.swing;
import javax.swing.*;
import java.awt.event.*;

public class Console {
  // Crear un string título a partir del nombre de la clase:
  public static String título(Object o) {
    String t = o.getClass().toString();
    // Eliminar la palabra "class":
    if(t.indexOf("class") != -1)
      t = t.substring(6);
    return t;
  }
  public static void setupClosing(JFrame frame) {
    // La solución de JDK 1.2 como una
    // clase interna anónima:
    frame.addWindowListener(new WindowAdapter() {
```

[8] Esto tendrá sentido una vez que se haya avanzado en este capítulo. En primer lugar, se hace la referencia **JApplet** miembro **static** de la clase (en vez de una variable local del **main( )**), y después se invoca a **applet.stop( )** y **applet.destroy( )** dentro de **WindowsAdapter.windowClosing( )** antes de invocar a **System.exit( )**.

```
        public void windowClosing(WindowEvent e) {
          System.exit(0);
        }
      });
      // La solución mejorada del JDK 1.3:
      // frame.setDefaultCloseOperation(
      //       EXIT_ON_CLOSE);
    }
    public static void
    run(JFrame frame, int width, int height) {
      setupClosing(frame);
      frame.setSize(width, height);
      frame.setVisible(true);
    }
    public static void
    run(JApplet applet, int width, int height) {
      JFrame frame = new JFrame(título(applet));
      setupClosing(frame);
      frame.getContentPane().add(applet);
      frame.setSize(width, height);
      applet.init();
      applet.start();
      frame.setVisible(true);
    }
    public static void
    run(JPanel panel, int width, int height) {
      JFrame frame = new JFrame(title(panel));
      setupClosing(frame);
      frame.getContentPane().add(panel);
      frame.setSize(width, height);
      frame.setVisible(true);
    }
} ///:~
```

Esta herramienta puede usarse cuando se desee, por ello está en **com.bruceeckel.swing**. La clase **Console** consiste únicamente en métodos **static**. El primero se usa para extraer el nombre de la clase (usando RTTI) del objeto y para eliminar la palabra "class", que suele incorporarse en **getClass( )**. Éste usa los métodos **String indexOf( )** para determinar si está o no la palabra "class", y **substring( )** para producir la nueva cadena de caracteres sin "class" o el espacio del final. Este nombre se usa para etiquetar la ventana que mostrarán los métodos **run( )**.

El método **setupClosing( )** se usa para esconder el código que hace que un **JFrame** salga del programa al cerrarlo. El comportamiento por defecto es no hacer nada, por lo que si no se llama a **setupClosing( )** o se escribe un código equivalente para el **JFrame**, la aplicación no se cerrará. La razón por la que este código está oculto va más allá de ubicarlo directamente en los métodos **run( )** subsecuentes se debe en parte a que nos permite usar el método por sí mismo cuando lo que se desea

hacer es más complicado que lo proporcionado por **run( )**. Sin embargo, también aísla un factor de cambio: Java 2 tiene dos formas de hacer que se cierren ciertos tipos de ventanas. En el JDK 1.2, la solución es crear una nueva clase **WindowAdapter** e implementar **windowClosing( )** como se vio anteriormente (se explicará el significado completo de hacer esto algo más adelante en este capítulo). Sin embargo, durante la creación de JDK 1.3 los diseñadores de la biblioteca observaron que generalmente sería necesario cerrar las ventanas siempre que se cree algo que no sea un *applet*, por lo que añadieron el método **setDefaultCloseOperation( )** tanto a **JFrame** como a **JDialog**. Desde el punto de vista de la escritura de código, el método nuevo es mucho más sencillo de usar pero este libro se escribió mientras seguía sin implementarse JDK 1.3 en Linux y en otras plataformas, por lo que en pos de la portabilidad entre versiones, se aisló el cambio dentro de **setupClosing( )**.

El método **run( )** está sobrecargado para que funcione con **JApplet**s, **JPanel**s y **JFrame**s. Nótese que sólo se invoca a **init( )** y **start( )** en el caso de que se trate de un **JApplet**.

Ahora, es posible ejecutar cualquier *applet* desde la consola creando un método **main( )** que contenga un método como:

```
Console.run(new MiClase(), 500, 300);
```

en la que los dos últimos argumentos son la anchura y altura de la visualización. He aquí **Applet1c.java** modificado para usar **Console**:

```
//: c13:Applet1d.java
// Console ejecuta applets desde la línea de comandos.
// <applet code=Applet1d width=100 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Applet1d extends JApplet {
  public void init() {
    getContentPane().add(new JLabel("¡Applet!"));
  }
  public static void main(String[] args) {
    Console.run(new Applet1d(), 100, 50);
  }
} ///:~
```

Esto permite la eliminación de código repetido a la vez que proporciona la mayor flexibilidad posible a la hora de ejecutar los ejemplos.

Usar el Explorador de Windows

Si se usa Windows, se puede simplificar el proceso de ejecutar un programa Java de línea de comandos configurando el Explorador de Windows —el navegador de archivos de Windows, *no* el

Internet Explorer— de forma que se pueda simplemente pulsar dos veces en un **.class** para ejecutarlo. Este proceso conlleva varios pasos.

Primero, descargar e instalar el lenguaje de programación Perl de *http://www.Perl.org*. En esta misma ubicación hay documentación e instrucciones relativas al lenguaje.

A continuación, hay que crear el siguiente script sin la primera y última líneas (este script es parte del paquete de código fuente de este libro):

```
//:! C13:EjecutarJava.bat
@rem = '--*-Perl-*--
@echo off
perl -x -S "%0" %1 %2 %3 %4 %5 %6 %7 %8 %9
goto endofperl
@rem ';
#!perl
$file = $ARGV[0];
$file =~ s/(.*)\..*/\1/;
$file =~ s/(.*\\)*(.*)/$+/;
`java $file`;
__END__
:endofperl
///:~
```

Ahora, se abre el Explorador de Windows, se selecciona "Ver", "Opciones de Carpeta", y después se pulsa en "Tipos de Archivo". Se presiona el botón "Nuevo Tipo". En "Descripción del tipo" introducir "Fichero de clase Java". En el caso de "Extensiones Asociadas", introducir "class". Bajo "Acciones", presionar el botón "Nueva". En "Acción" introducir "Open" y en "Aplicación a utilizar para realizar la acción" introducir una línea como:

```
"c:\aaa\Perl\RunJava.bat" "%L"
```

Hay que personalizar la trayectoria ubicada antes de "EjecutarJava.bat" para que contenga la localización en la que cada uno ubique el archivo **.bat**.

Una vez hecha esta instalación, se puede ejecutar cualquier programa Java simplemente pulsando dos veces en el archivo **.class** que contenga un método **main( )**.

Hacer un botón

Hacer un botón es bastante simple: simplemente se invoca al constructor **JButton** con la etiqueta que se desee para el botón. Se verá más adelante que se pueden hacer cosas más elegantes, como poner imágenes gráficas en los botones.

Generalmente se deseará crear un campo para el botón dentro de la clase, de forma que podamos referirnos al mismo más adelante.

El **JButton** es un componente —su propia pequeña ventana— que se repintará automáticamente como parte de la actualización. Esto significa que no se pinta explícitamente ningún botón, ni ningún otro tipo de control; simplemente se ubican en el formulario y se deja que se encarguen ellos mismos de pintarse. Por tanto, para ubicar un botón en un formulario, se hace dentro de **init( )**:

```
//: c13:Boton1.java
// Poniendo botones en un applet.
// <applet code=Boton1 width=200 height=50>
// </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Boton1 extends JApplet {
  JButton
    b1 = new JButton("Boton 1"),
    b2 = new JButton("Boton 2");
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
  }
  public static void main(String[] args) {
    Console.run(new Button1(), 200, 50);
  }
} ///:~
```

En este caso, se ha añadido algo nuevo: antes de ubicar los elementos en el panel de contenidos, se ha asignado a éste un "gestor de disposición", de tipo **FlowLayout**. Un gestor de disposiciones permite organizar el control dentro de un formulario. El comportamiento normal para un *applet* es usar el **BorderLayout**, pero éste no funcionaría aquí pues (se aprenderá algo más adelante cuando se explique en detalle cómo controlar la disposición de un formulario) por defecto cubre cada control completamente con cada uno que se añada. Sin embargo, **FlowLayout** hace que los controles fluyan por el formulario, de izquierda a derecha y de arriba hacia abajo.

Capturar un evento

Uno se dará cuenta de que si se compila y ejecuta el *applet* de arriba, no ocurre nada cuando se presionan los botones. Es aquí donde hay que escribir algún tipo de código para ver qué ocurrirá. La base de la programación conducida por eventos, que incluye mucho de lo relacionado con IGU, es atar los eventos al código que responda a los mismos.

La forma de hacer esto en Swing es separando limpiamente la interfaz (los componentes gráficos) y la implementación (el código que se desea ejecutar cuando se da cierto evento en un componen-

te). Cada componente Swing puede reportar todos los eventos que le pudieran ocurrir, y puede reportar también cada tipo de evento individualmente. Por tanto, si uno no está interesado, por ejemplo, en ver si se está moviendo el ratón por encima del botón, no hay por qué registrar interés en ese evento. Se trata de una forma muy directa y elegante de manejar la programación dirigida por eventos, y una vez que se entienden los conceptos básicos se puede usar componentes Swing de forma tan sencilla que nunca antes se podría imaginar —de hecho, el modelo se extiende a todo lo que pueda clasificarse como JavaBean (sobre los que se aprenderá más adelante en este Capítulo).

En primer lugar, nos centraremos simplemente en el evento de interés principal para los componentes que se usen. En el caso de un **JButton**, este "evento de interés" es que se presione el botón. Para registrar interés en que se presione un botón, se invoca al método **addActionListener( )** del **JButton**. Este método espera un parámetro que es un objeto que implemente la interfaz **ActionListener**, que contiene un único método denominado **actionPerformed( )**. Por tanto, todo lo que hay que hacer para adjuntar código a un **JButton** es implementar la interfaz **ActionListener** en una clase y registrar un evento de esa clase con el **JButton** vía **addActionListener( )**. El método será invocado al presionar el botón (a esto se le suele denominar *retrollamada*).

Pero, ¿cuál debería ser el resultado de presionar ese botón? Nos gustaría ver que algo cambia en la pantalla, por tanto, se introducirá un nuevo componente Swing: el **JTextField**. Se trata de un lugar en el que se puede escribir texto, o en este caso, ser modificado por el programa. Aunque hay varias formas de crear un **JTextField**, la más sencilla es decir al constructor lo ancho que se desea que sea éste. Una vez ubicado el **JTextField** en el formulario, se puede modificar su contenido utilizando el método **setText( )** (hay muchos otros métodos en **JTextField**, que pueden encontrarse en la documentación HTML del JDK disponible en *http://java.sun.com*). Ésta es la apariencia que tiene:

```
//: c13:Boton2.java
// Respondiendo al presionar un botón.
// <applet code=Boton2 width=200 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Boton2 extends JApplet {
  JButton
    b1 = new JButton("Boton 1"),
    b2 = new JButton("Boton 2");
  JTextField txt = new JTextField(10);
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e){
      String nombre =
        ((JButton)e.getSource()).getText();
      txt.setText(nombre);
    }
  }
```

```
  BL al = new BL();
  public void init() {
    b1.addActionListener(al);
    b2.addActionListener(al);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(txt);
  }
  public static void main(String[] args) {
    Console.run(new Button2(), 200, 75);
  }
} ///:~
```

Crear un **JTextFiled** y ubicarlo en el lienzo conlleva los mismos pasos que en los **JButtons**, o cualquier componente Swing. La diferencia en el programa de arriba radica en la creación de **BL** de la clase **ActionLitener** ya mencionada. El argumento al **actionPerformed( )** es de tipo **ActionEvent**, que contiene toda la información sobre el evento y su procedencia. En este caso, quería describir el botón que se presionaba: **getSource( )** produce el objeto en el que se originó el evento, y asumí que **JButton.getText( )** devuelve el texto que está en el botón, y que éste está en el **JTextField** para probar que sí que se estaba invocando al código al presionar el botón.

En **init( )** se usa **addActionListener( )** para registrar el objeto **BL** con ambos botones.

Suele ser más conveniente codificar el **ActionListener** como una clase interna anónima, especialmente desde que se tiende sólo a usar una instancia de cada clase oyente. Puede modificarse **Boton2.java** para usar clases internas anónimas como sigue:

```
//: c13:Boton2b.java
// Utilizando clases anónimas internas.
// <applet code=Boton2b width=200 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Boton2b extends JApplet {
  JButton
    b1 = new JButton("Boton 1"),
    b2 = new JButton("Boton 2");
  JTextField txt = new JTextField(10);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      String nombre =
```

```
            ((JButton)e.getSource()).getText();
          txt.setText(nombre);
        }
      };
      public void init() {
        b1.addActionListener(al);
        b2.addActionListener(al);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
        cp.add(txt);
      }
      public static void main(String[] args) {
        Console.run(new Button2b(), 200, 75);
      }
} ///:~
```

Se preferirá el enfoque de usar clases anónimas internas (siempre que sea posible) para los ejemplos de este libro.

Áreas de texto

Un **JTextArea** es como un **JTextField** excepto en que puede tener múltiples líneas y tiene más funcionalidad. Un método particularmente útil es **append( )**; con él se puede incorporar alguna salida de manera sencilla a la **JTextArea**, logrando así una mejora de Swing (dado que se puede hacer desplazamiento hacia delante y hacia atrás) sobre lo que se había logrado hasta la fecha haciendo uso de programas de línea de comandos que imprimían a la salida estándar. Como ejemplo, el siguiente programa rellena una **JTextArea** con la salida del generador **geografía** del Capítulo 9:

```
//: c13:AreaTexto.java
// Utilizando el control JTextArea.
// <applet code=TextArea width=475 height=425>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.util.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class AreaTexto extends JApplet {
  JButton
    b = new JButton("Añadir datos"),
    c = new JButton("Limpiar datos");
```

```
  JTextArea t = new JTextArea(20, 40);
  Map m = new HashMap();
  public void init() {
    // Usar todos los datos:
    Colecciones2.fill(m,
      Colecciones2.geografia,
      CapitalesPaises.pares.length);
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        for(Iterator it= m.entrySet().iterator();
            it.hasNext();){
          Map.Entry me = (Map.Entry)(it.next());
          t.append(me.getKey() + ": "
            + me.getValue() + "\n");
        }
      }
    });
    c.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        t.setText("");
      }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JScrollPane(t));
    cp.add(b);
    cp.add(c);
  }
  public static void main(String[] args) {
    Console.run(new TextArea(), 475, 425);
  }
} ///:~
```

En el método **init( )**, se rellena **Map** con todos los países y sus capitales. Nótese que para ambos botones se crea y añade el **ActionListener** sin definir una variable intermedia, puesto que nunca es necesario hacer referencia a ese oyente en todo el programa. El botón "Añadir Datos" da formato e incorpora todos los datos, mientras que el botón "Borrar Datos" hace uso de **setText( )** para eliminar todo el texto del **JTextArea**.

Al añadir el **JTextArea** al *applet*, se envuelve en un **JScrollPane** para controlar el desplazamiento cuando se coloca demasiado texto en la pantalla. Esto es todo lo que hay que hacer para lograr capacidades de desplazamiento al completo. Habiendo intentado averiguar cómo hacer el equivalente en otros entornos de programación, estoy muy impresionado con la simplicidad y el buen diseño de componentes como **JScrollPane**.

Controlar la disposición

La forma de colocar los componentes en un formulario en Java probablemente sea distinta de cualquier otro sistema IGU que se haya usado. En primer lugar, es todo código; no hay "recursos" que controlen la ubicación de los componentes. Segundo, la forma de colocar los componentes en un formulario no está controlada por ningún tipo de posicionado absoluto sino por un "gestor de disposición" que decide cómo disponer los componentes basándose en el orden en que se invoca a **add( )** para ellos. El tamaño, forma y ubicación de los componentes será notoriamente diferente de un gestor de disposición a otro. Además, los gestores de disposición se adaptan a las dimensiones del *applet* o a la ventana de la aplicación, por lo que si se cambia la dimensión de la ventana, cambiarán como respuesta el tamaño, la forma y la ubicación de los componentes.

JApplet, **JFrame**, **JWindow** y **JDialog** pueden todos producir un **Container** con **getContentPane( )** que puede contener y mostrar **Componentes**. En **Container**, hay un método denominado **setLayout( )** que permite elegir entre varios gestores de disposición distintos. Otras clases, como **JPanel**, contienen y muestran los componentes directamente, de forma que también se establece el gestor de disposición directamente, sin usar el cuadro de contenidos.

En esta sección se explorarán los distintos gestores de disposición ubicando botones en los mismos (puesto que es lo más sencillo que se puede hacer con ellos). No habrá ninguna captura de eventos de los botones puesto que simplemente se pretende que estos ejemplos muestren cómo se disponen los botones.

BorderLayout

El *applet* usa un esquema de disposición por defecto: el **BorderLayout** (varios de los ejemplos anteriores variaban éste al **FlowLayout**). Sin otra instrucción, éste toma lo que se **add( )** (añade) al mismo y lo coloca en el centro, extendiendo el objeto hasta los bordes.

Sin embargo, hay más. Este gestor de disposición incorpora los conceptos de cuatro regiones limítrofes en torno a un área central. Cuando se añade algo a un panel que está haciendo uso de un **BorderLayout** se puede usar el método sobrecargado **add( )** que toma como primer parámetro un valor constante. Este valor puede ser cualquiera de los siguientes:

BorderLayout.NORTH(superior)
BorderLayout.SOUTH (inferior)
BorderLayout.EAST(derecho)
BorderLayout.WEST(izquierdo)
BorderLayout.CENTER(rellenar el centro hasta los bordes de todos los demás componentes)

Si no se especifica un área al ubicar un objeto, éste ira por defecto a **CENTER**.

He aquí un simple ejemplo. Se usa la disposición por defecto, puesto que **JApplet** se pone por defecto a **BorderLayout**:

```
//: c13:BorderLayout1.java
// Demuestra BorderLayout.
// <applet code=BorderLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class BorderLayout1 extends JApplet {
  public void init() {
    Container cp = getContentPane();
    cp.add(BorderLayout.NORTH,
        new JButton ("Norte"));
    cp.add(BorderLayout.SOUTH,
        new JButton ("Sur"));
    cp.add(BorderLayout.EAST,
        new JButton ("Este"));
    cp.add(BorderLayout.WEST,
        new JButton ("Oeste"));
    cp.add(BorderLayout.CENTER,
        new JButton ("Centro"));
  }
  public static void main(String[] args) {
    Console.run(new BorderLayout1(), 300, 250);
  }
} ///:~
```

Para todas las ubicaciones excepto **CENTER**, el elemento que se añada se comprime hasta caber en la menor cantidad de espacio posible en una de las dimensiones, extendiéndose al máximo en la otra dimensión. Sin embargo, **CENTER** se extiende en ambas dimensiones para ocupar todo el centro.

FlowLayout

Simplemente "hace fluir" los componentes del formulario de izquierda a derecha hasta que se llena el espacio de arriba, después se mueve una fila hacia abajo y continúa fluyendo.

He aquí un ejemplo que establece el gestor de disposición a **FlowLayout** y después ubica botones en el formulario. Se verá que con **FlowLayout** los componentes tomarán su tamaño "natural". Por ejemplo un **JButton**, será del tamaño de su cadena de caracteres:

```
//: c13:FlowLayout1.java
// Demuestra FlowLayout.
// <applet code=FlowLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;
```

```
public class FlowLayout1 extends JApplet {
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < 20; i++)
      cp.add(new JButton("Boton " + i));
  }
  public static void main(String[] args) {
    Console.run(new FlowLayout1(), 300, 250);
  }
} ///:~
```

Se compactarán todos los componentes a su menor tamaño posible en un **FlowLayout**, de forma que se podría obtener un comportamiento ligeramente sorprendente. Por ejemplo, dado que un **JLabel** será del tamaño de su cadena de caracteres, intentar justificar el texto a la derecha conduce a que lo mostrado utilizando **FlowLayout** no varíe.

GridLayout

Un **GridLayout** permite construir una tabla de componentes, y al añadirlos se ubican de izquierda a derecha y de arriba abajo en la rejilla. En el constructor se especifica el número de filas y columnas que se necesiten y éstas se distribuyen en proporciones iguales:

```
//: c13:GridLayout1.java
// Demuestra GridLayout.
// <applet code=GridLayout1
// width=300 height=250> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class GridLayout1 extends JApplet {
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(7,3));
    for(int i = 0; i < 20; i++)
      cp.add(new JButton("Boton " + i));
  }
  public static void main(String[] args) {
    Console.run(new GridLayout1(), 300, 250);
  }
} ///:~
```

En este caso hay 21 casillas y sólo 20 botones. La última casilla se deja vacía porque en un **GridLayout** no se produce ningún "balanceo".

GridBagLayout

El **GridBagLayout** proporciona un control tremendo a la hora de decidir exactamente cómo distribuir en sí las regiones de la ventana además de cómo reformatear cada región cuando se redimensionan las ventanas. Sin embargo, también es el gestor de disposiciones más complicado, y bastante difícil de entender. Inicialmente está pensado para generación automática de código por parte de un constructor de IGU (los buenos constructores de IGU utilizarán **GridBagLayout** en vez de posicionamiento absoluto). Si uno tiene un diseño tan complicado que cree que debe usar **GridBagLayout**, habría que usar una buena herramienta de construcción de IGU para ese diseño. Si siente la necesidad de saber detalles intrínsecos, le recomiendo la lectura de *Core Java 2*, de Hostmann & Cornell (Prentice-Hall, 1999), o un libro dedicado a la Swing, como punto de partida.

Posicionamiento absoluto

También es posible establecer la posición absoluta de los componentes gráficos así:

1. Poner a **null** el gestor de disposiciones del **Container**: **setLayout(null)**.
2. Invocar a **setBounds( )** o **reshape( )** (en función de la versión del lenguaje) por cada componente, pasando un rectángulo circundante con sus coordenadas en puntos. Esto se puede hacer en el constructor o en **paint( )**, en función de lo que se desee lograr.

Algunos constructores de IGU usan este enfoque de forma extensiva, pero ésta no suele ser la mejor forma de generar código. Los constructores de IGU más útiles usan en vez de éste el **GridBagLayout**.

BoxLayout

Debido a que la gente tiene tantas dificultades a la hora de entender y trabajar con **GridBagLayout**, Swing también incluye el **BoxLayout**, que proporciona muchos de los beneficios de **GridBagLayout** sin la complejidad, de forma que a menudo se puede usar cuando se precisen disposiciones codificadas a mano (de nuevo, si el diseño se vuelve demasiado complicado, es mejor usar un constructor de IGU que genere **GridBagLayout**s automáticamente). **BoxLayout** permite controlar la ubicación de los componentes vertical u horizontalmente, y controlar el espacio entre componentes utilizando algo que denomina "puntales y pegamento". En primer lugar, veremos cómo usar directamente **BoxLayout** de la misma forma que se ha demostrado el uso de los demás gestores de disposiciones:

```
//: c13:BoxLayout1.java
// BoxLayouts vertical y horizontal.
// <applet code=BoxLayout1
// width=450 height=200> </applet>
import javax.swing.*;
```

```
import java.awt.*;
import com.bruceeckel.swing.*;

public class BoxLayout1 extends JApplet {
  public void init() {
    JPanel jpv = new JPanel();
    jpv.setLayout(
      new BoxLayout(jpv, BoxLayout.Y_AXIS));
    for(int i = 0; i < 5; i++)
      jpv.add(new JButton("" + i));
    JPanel jph = new JPanel();
    jph.setLayout(
      new BoxLayout(jph, BoxLayout.X_AXIS));
    for(int i = 0; i < 5; i++)
      jph.add(new JButton("" + i));
    Container cp = getContentPane();
    cp.add(BorderLayout.EAST, jpv);
    cp.add(BorderLayout.SOUTH, jph);
  }
  public static void main(String[] args) {
    Console.run(new BoxLayout1(), 450, 200);
  }
} ///:~
```

El constructor de **BoxLayout** es un poco diferente del de los otros gestores de disposición —se proporciona el **Container** que va a ser controlado por el **BoxLayout** como primer argumento, y la dirección de la disposición como segundo argumento.

Para simplificar las cosas, hay un contenedor especial denominado **Box** que usa **BoxLayout** como gestor nativo. El ejemplo siguiente distribuye los componentes horizontal y verticalmente usando **Box**, que tiene dos métodos **static** para crear cajas con alineación vertical y horizontal:

```
//: c13:Box1.java
// BoxLayout vertical y horizontal.
// <applet code=Box1
// width=450 height=200> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box1 extends JApplet {
  public void init() {
    Box bv = Box.createVerticalBox();
    for(int i = 0; i < 5; i++)
      bv.add(new JButton("" + i));
```

```
    Box bh = Box.createHorizontalBox();
    for(int i = 0; i < 5; i++)
      bh.add(new JButton("" + i));
    Container cp = getContentPane();
    cp.add(BorderLayout.EAST, bv);
    cp.add(BorderLayout.SOUTH, bh);
  }
  public static void main(String[] args) {
    Console.run(new Box1(), 450, 200);
  }
} ///:~
```

Una vez que se tiene una **Box**, se pasa como segundo parámetro al añadir componentes al panel contenedor.

Los puntales añaden espacio entre componentes, midiéndose este espacio en puntos. Para usar un puntal, simplemente se añade éste entre la adición de los componentes que se desea separar:

```
//: c13:Box2.java
// Añadiendo puntales.
// <applet code=Box2
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box2 extends JApplet {
  public void init() {
    Box bv = Box.createVerticalBox();
    for(int i = 0; i < 5; i++) {
      bv.add(new JButton("" + i));
      bv.add(Box.createVerticalStrut(i*10));
    }
    Box bh = Box.createHorizontalBox();
    for(int i = 0; i < 5; i++) {
      bh.add(new JButton("" + i));
      bh.add(Box.createHorizontalStrut(i*10));
    }
    Container cp = getContentPane();
    cp.add(BorderLayout.EAST, bv);
    cp.add(BorderLayout.SOUTH, bh);
  }
  public static void main(String[] args) {
    Console.run(new Box2(), 450, 300);
  }
} ///:~
```

Los puntales separan los componentes en una cantidad fija, mientras que la cola actúa al contrario: separa los componentes tanto como sea posible. Por consiguiente, es más un elástico que "pegamento" (y el diseño en el que se basó se denominaba "elásticos y puntales" por lo que la elección del término es algo misteriosa).

```
//: c13:Box3.java
// Usando Pegamento.
// <applet code=Box3
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box3 extends JApplet {
  public void init() {
    Box bv = Box.createVerticalBox();
    bv.add(new JLabel("Hola"));
    bv.add(Box.createVerticalGlue());
    bv.add(new JLabel("Applet"));
    bv.add(Box.createVerticalGlue());
    bv.add(new JLabel("Mundo"));
    Box bh = Box.createHorizontalBox();
    bh.add(new JLabel("Hola"));
    bh.add(Box.createHorizontalGlue());
    bh.add(new JLabel("Applet"));
    bh.add(Box.createHorizontalGlue());
    bh.add(new JLabel("Mundo"));
    bv.add(Box.createVerticalGlue());
    bv.add(bh);
    bv.add(Box.createVerticalGlue());
    getContentPane().add(bv);
  }
  public static void main(String[] args) {
    Console.run(new Box3(), 450, 300);
  }
} ///:~
```

Un puntal funciona de forma inversa, pero un área rígida fija los espacios entre componentes en ambas direcciones:

```
//: c13:Box4.java
// Las áreas rígidas son como pares de puntales.
// <applet code=Box4
// width=450 height=300> </applet>
import javax.swing.*;
import java.awt.*;
```

```
import com.bruceeckel.swing.*;

public class Box4 extends JApplet {
  public void init() {
    Box bv = Box.createVerticalBox();
    bv.add(new JButton("Arriba"));
    bv.add(Box.createRigidArea(
      new Dimension(120, 90)));
    bv.add(new JButton("Abajo"));
    Box bh = Box.createHorizontalBox();
    bh.add(new JButton("Izquierda"));
    bh.add(Box.createRigidArea(
      new Dimension(160, 80)));
    bh.add(new JButton("Derecha"));
    bv.add(bh);
    getContentPane().add(bv);
  }
  public static void main(String[] args) {
    Console.run(new Box4(), 450, 300);
  }
} ///:~
```

Habría que ser consciente de que las áreas rígidas son algo controvertidas. Puesto que usan valores absolutos, algunos piensan que pueden causar demasiados problemas como para merecer la pena.

¿El mejor enfoque?

Swing es potente; puede lograr mucho con unas pocas líneas de código. Los ejemplos mostrados en este libro son razonablemente simples, y por propósitos de aprendizaje tiene sentido escribirlos a mano. De hecho se puede lograr, simplemente, combinando disposiciones. En algún momento, sin embargo, deja de tener sentido codificar a mano formularios IGU —se vuelve demasiado complicado y no constituye un buen uso del tiempo de programación. Los diseñadores de Java y Swing orientaron el lenguaje y las bibliotecas de forma que soportaran herramientas de construcción de IGU, creadas con el propósito específico de facilitar la experiencia de programación. A medida que se entiende todo lo relacionado con disposiciones y con cómo tratar con esos eventos (como se describe a continuación), no es particularmente importante que se sepan los detalles de cómo distribuir componentes a mano —deje que la herramienta apropiada haga el trabajo por usted (Java, después de todo, está diseñado para incrementar la productividad del programador).

El modelo de eventos de Swing

En el modelo de eventos de Swing, un componente puede iniciar ("disparar") un evento. Cada tipo de evento está representado por una clase distinta. Cuando se dispara un evento, éste es recibido

por uno o más "oyentes", que actúan sobre ese evento. Por consiguiente, la fuente de un evento y el lugar en el que éste es gestionado podrían ser diferentes. Por tanto, generalmente se usan los componentes Swing tal y como son, aunque es necesario escribir el código al que se invoca cuando los componentes reciben un evento, de forma que nos encontramos ante un excelente ejemplo de la separación entre la interfaz y la implementación.

Cada oyente de eventos es un objeto de una clase que implementa un tipo particular de **interfaz** oyente. Por tanto, como programador, todo lo que hay que hacer es crear un objeto oyente y registrarlo junto con el componente que dispara el evento. El registro se lleva a cabo invocando a un método **addXXXListener( )** en el componente que dispara el evento, donde "**XXX**" representa el tipo de evento por el que se escucha. Se pueden conocer fácilmente los tipos de eventos que pueden gestionarse fijándose en los nombres de los métodos "addListener", y si se intenta escuchar por eventos erróneos se descubrirá el error en tiempo de compilación. Más adelante en este capítulo se verá que los JavaBeans también usan los nombres de los métodos "addListener" para determinar qué eventos puede manejar un Bean.

Después, toda la lógica de eventos irá dentro de una clase oyente. Cuando se crea una clase oyente, su única restricción es que debe implementar la interfaz apropiada. Se puede crear una clase oyente global, pero ésta es una situación en la que tienden a ser bastante útiles las clases internas, no sólo por proporcionar una agrupación lógica de las clases oyentes dentro del IU o de las clases de la lógica de negocio a las que sirven, sino (como se verá más adelante) por el hecho de que una clase interna proporciona una forma elegante más allá de una clase y los límites del subsistema.

Todos los ejemplos mostrados en este capítulo hasta el momento han usado el modelo de eventos de Swing, pero el resto de la sección acabará de describir los detalles de ese modelo.

Tipos de eventos y oyentes

Todos los componentes Swing incluyen **addXXXListener( )** y **removeXXXListener( )** de forma que se pueden añadir y eliminar los tipos de oyentes apropiados de cada componente. Se verá que en cada caso el "**XXX**" representa el parámetro del método, por ejemplo: **addMiOyente (MiOyente m)**. La tabla siguiente incluye los eventos básicos asociados, los oyentes y los métodos, junto con los componentes básicos que soportan esos eventos particulares proporcionando los métodos **addXXXListener( )** y **removeXXXListener( )**. Debería tenerse en mente que el modelo de eventos fue diseñado para ser extensible, de forma que se podrían encontrar otros tipos de eventos y oyentes que no estén en esta tabla.

Evento, interfaz oyente y métodos add- y remove	**Componentes que soportan este evento**
ActionEvent **ActionListener** **addActionListener( )** **removeActionListener( )**	**JButton, JList, JTextField,** **JMenuItem** y sus derivados incluyendo **JCheckBoxMenuItem, JMenu,** y **JPopupMenu.**

Evento, interfaz oyente y métodos add- y remove	Componentes que soportan este evento
AdjustmentEvent **AdjustmentListener** **addAdjustementListener( )** **removeAdjustmentListener( )**	**JScrollBar** y cualquier cosa que se cree que implemente la **interfaz Adjustable**.
ComponentEvent **ComponentListener** **addComponentListener( )** **removeComponentListener( )**	***Component** y sus derivados incluyendo **JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea** y **JTextField**.
ContainerEvent **ContainerListener** **addContainerListener( )** **removeContainerListener( )**	**Container** y sus derivados incluyendo **JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog** y **JFrame**.
FocusEvent **FocusListener( )** **addFocusListener( )** **removeFocusListener( )**	**Component** y sus derivados*.
KeyListener **addKeyListener ( )** **removeKeyListener( )**	**Component** y sus derivados*.
MouseEvent (para ambos clicks y movimiento) **MouseListener** **addMouseListener( )** **removeMouseListener( )**	**Component** y sus derivados*.
MouseEvent[9] (para ambos clics y movimiento) **MouseMotionListener** **addMouseMotionListener( )** **removeMouseMotionLitener( )**	**Component** y sus derivados*.
WindowEvent **WindowListener** **addWindowListener( )** **removeWindowListener( )**	**Window** y sus derivados, incluyendo **JDialog, JFileDialog** y **JFrame**.

[9] No existe evento **MouseMotionEvent** incluso aunque parece que debería haberlo. Hacer clic y el movimiento se combinan en **MouseEvent**, por lo que esta segunda ocurrencia de **MouseEvent** en la tabla no es ningún error.

Evento, interfaz oyente y métodos add- y remove	Componentes que soportan este evento
ItemEvent **ItemListener** **addItemListener( )** **removeItemListener( )**	**JCheckBox,** **JCheckBoxMenuItem,** **JComboBox, JList** y cualquier cosa que implemente la **interfaz ItemSelectable.**
TextEvent **TextListener** **addTextListener( )** **removeTextListener( )**	Cualquier cosa derivada de **JTextComponent** incluyendo **JTextArea** y **JTextField.**

Se puede ver que cada tipo de componente sólo soporta ciertos tipos de eventos. Se vuelve bastante difícil mirar todos los eventos que soporta cada componente. Un enfoque más sencillo es modificar el programa **MostrarLimpiarMetodos.java** del Capítulo 12, de forma que muestre todos los oyentes de eventos soportados por cualquier componente Swing que se introduzca.

El Capítulo 12 introdujo la *reflectividad* y usaba esa faceta para buscar los métodos de una clase particular —bien la lista de métodos al completo, o bien un subconjunto de aquéllos cuyos nombres coincidan con la palabra clave que se proporcione. La magia de esto es que puede mostrar automáticamente *todos* los métodos de una clase sin forzarla a recorrer la jerarquía de herencias hacia arriba examinando las clases base en cada nivel. Por consiguiente, proporciona una valiosa herramienta para ahorrar tiempo a la hora de programar: dado que la mayoría de los nombres de los métodos de Java son bastante descriptivos, se pueden buscar los nombres de métodos que contengan una palabra de interés en particular. Cuando se encuentre lo que uno cree estar buscando, compruebe la documentación en línea.

Sin embargo, cuando llegamos al Capítulo 12 no se había visto Swing, por lo que la herramienta de ese capítulo se desarrolló como aplicación de línea de comandos. Aquí está la versión más útil de IGU, especializada en buscar los métodos "addListener" de los componentes Swing:

```
//: c13:MostrarAddListeners.java
// Mostrar los metodos "addXXXListener" de cualquier
// clase Swing.
// <applet code = MostrarAddListeners
// width=500 height=400></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.io.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;
```

```
public class MostrarAddListeners extends JApplet {
  Class cl;
  Method[] m;
  Constructor[] ctor;
  String[] n = new String[0];
  JTextField nombre = new JTextField(25);
  JTextArea resultados = new JTextArea(40, 65);
  class NombreL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String nm = nombre.getText().trim();
      if(nm.length() == 0) {
        resultados.setText("No coinciden");
        n = new String[0];
        return;
      }
      try {
        cl = Class.forName("javax.swing." + nm);
      } catch(ClassNotFoundException ex) {
        resultados.setText("No coinciden");
        return;
      }
      m = cl.getMethods();
      // Convertir en un array de Strings:
      n = new String[m.length];
      for(int i = 0; i < m.length; i++)
        n[i] = m[i].toString();
      representar();
    }
  }
  void representar() {
    // Crear el conjunto resultado:
    String[] rs = new String[n.length];
    int j = 0;
    for (int i = 0; i < n.length; i++)
      if(n[i].indexOf("add") != -1 &&
        n[i].indexOf("Listener") != -1)
          rs[j++] =
            n[i].substring(n[i].indexOf("add"));
    resultados.setText("");
    for (int i = 0; i < j; i++)
      resultados.append(
        EliminarCalificadores.strip(rs[i]) + "\n");
  }
  public void init() {
    nombre.addActionListener(new NombreL());
```

```
    JPanel cima = new JPanel();
    cima.add(new JLabel(
      "Nombre de clase Swing (presionar ENTER):"));
    cima.add(nombre);
    Container cp = getContentPane();
    cp.add(BorderLayout.NORTH, top);
    cp.add(new JScrollPane(resultados));
  }
  public static void main(String[] args) {
    Console.run(new MostrarAddListeners(), 500,400);
  }
} ///:~
```

Aquí se vuelve a usar la clase **EliminarCalificadores** definida en el Capítulo 12 importando la biblioteca **com.bruceeckel.util**.

La IGU contiene un **JTextField nombre** en el que se puede introducir el nombre de la clase Swing que se desea buscar. Los resultados se muestran en una **JTextArea**.

Se verá que no hay botones en los demás componentes mediante los que indicar que se desea comenzar la búsqueda. Esto se debe a que **JTextField** está monitorizada por un **ActionListener**. Siempre que se haga un cambio y se presione ENTER se actualiza la lista inmediatamente. Si el texto no está vacío, se usa dentro de **Class.forName( )** para intentar buscar la clase. Si el nombre es incorrecto, **Class.forName( )** fallará, lo que significa que lanzará una excepción. Ésta se atrapa y se pone la **JTextArea** a "No coinciden". Pero si se teclea un nombre correcto (teniendo en cuenta las mayúsculas y minúsculas), **Class.forName( )** tiene éxito y **getMethods( )** devolverá un array de objetos **Method**. Cada uno de los objetos del array se convierte en un **String** vía **toString( )** (esto produce la signatura completa del método) y se añade a **n**, un array de **String**s. El array **n** es un miembro de **class ShowAddListeners** y se usa en la actualización de la pantalla siempre que se invoca a **representar( )**.

El método **representar( )** crea un array de **String**s llamado **rs** (de "result set" —"conjunto resultado" en inglés). De este conjunto se copian en **n** aquellos elementos que contienen "add" y "Listener". Después se usan **indexOf( )** y **substring( )** para eliminar los calificadores como **public**, **static**, etc. Finalmente, **EliminarCalificadores.strip( )** remueve los calificadores de nombre extra.

Este programa constituye una forma conveniente de investigar las capacidades de un componente Swing. Una vez que se conocen los eventos soportados por un componente en particular, no es necesario buscar nada para que reaccione ante el evento. Simplemente:

1. Se toma el nombre de la clase evento y se retira la palabra "**Event**". Se añade la palabra "**Listener**" a lo que queda. Ésta es la interfaz oyente a implementar en la clase interna.

2. Implementar la interfaz de arriba y escribir los métodos de los eventos a capturar. Por ejemplo, se podría estar buscando movimientos de ratón, por lo que se escribe código para el método **mouseMoved( )** de la interfaz **MouseMotionListener**. (Hay que implementar los otros métodos, por supuesto, pero a menudo hay un atajo que veremos pronto.)

3. Crear un objeto de la clase oyente del paso 2. Registrarlo con el componente con el método producido al prefijar "**add**" al nombre del Oyente. Por ejemplo, **addMouseMotionListener( )**.

He aquí algunos de las interfaces Oyente:

Listener interface oyente y adaptadores	Métodos de la interfaz
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged (AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)

Éste no es un listado exhaustivo, en parte porque el modelo de eventos permite crear tipos de eventos y oyentes personalizados. Por consiguiente, frecuentemente se tendrá acceso a bibliotecas que han inventado sus propios eventos, de forma que el conocimiento que se adquiera en este capítulo te permitirá adivinar como utilizar esos eventos.

Utilizar adaptadores de oyentes por simplicidad

En la tabla de arriba, se puede ver que algunas interfaces oyentes sólo tienen un método. Éstas son triviales de implementar puesto que se implementarán sólo cuando se desee escribir ese método en particular. Sin embargo, las interfaces oyentes que tienen múltiples métodos pueden ser menos agradables de usar. Por ejemplo, algo que hay que hacer siempre que se cree una aplicación es proporcionar un **WindowListener** al **JFrame** de forma que cuando se logre el evento **windowClosing( )** se llame a **System.exit( )** para salir de la aplicación. Pero dado que **WindowListener** es una **interfaz**, hay que implementar todos los demás métodos incluso aunque no hagan nada. Esto puede resultar molesto.

Para solucionar el problema, algunas (aunque no todas) de las interfaces oyentes que tienen más de un método se suministran con *adaptadores*, cuyos nombres pueden verse en la tabla de arriba. Cada adaptador proporciona métodos por defecto vacíos para cada uno de los métodos de la interfaz. Después, todo lo que hay que hacer es heredar del adaptador y superponer sólo los métodos que se necesiten cambiar. Por ejemplo, el **WindowListener** típico a usar tendrá la siguiente apariencia (recuérdese que se ha envuelto en la clase **Console** de **com.bruceeckel.swing**):

```
class MiWindowListener extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

La única razón para la existencia de los adaptadores es facilitar la creación de las clases Oyente.

Sin embargo, los adaptadores también tienen un inconveniente. Imagínese que se escribe un **WindowAdapter** como el de arriba:

```
class MiWindowListener extends WindowAdapter {
  public void WindowClosing(WindowEvent e) {
    System.exit(0);
  }
}
```

No funciona, pero uno se volverá loco tratando de averiguar por qué, pues compila y se ejecuta correctamente —excepto por el hecho de que cerrar la ventana no saldrá del programa. ¿Se ve el problema? Está en el nombre del método: **WindowClosing( )** en vez de **windowClosing( )**. Un simple error con una mayúscula produce la adición de un método completamente nuevo. Sin embargo, no es el método que se invoca cuando se cierra la ventana, por lo que no se obtendrán los resultados deseados. Con excepción de este inconveniente, una **interfaz** garantizará que los métodos estén correctamente implementados.

Seguimiento de múltiples eventos

Para probar que estos eventos se están disparando verdaderamente, y como experimento interesante, merece la pena crear un *applet* que haga un seguimiento de comportamiento extra en un **JButton** (que no sea si está o no presionado). Este ejemplo también muestra cómo heredar tu propio objeto botón puesto que se usa como destino de todos los eventos de interés. Para lograrlo, simplemente se puede heredar de **JButton**[10].

La clase **MiBoton** es una clase interna de **RastrearEvento**, por lo que **MiBoton** puede llegar a la ventana padre y manipular sus campos de texto, que es lo necesario para poder escribir la información de estado en los campos del padre. Por supuesto ésta es una solución limitada, puesto que **MiBoton** puede usarse sólo en conjunción con **RastrearEvento**. A este tipo de código se le suele denominar "altamente acoplado":

```
//: c13:RastrearEvento.java
// Mostrar eventos a medida que ocurren.
// <applet code=RastrearEvento
//  width=700 height=500></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class RastrearEvento extends JApplet {
  HashMap h = new HashMap();
  String[] evento = {
    "focusGained", "focusLost", "keyPressed",
    "keyReleased", "keyTyped", "mouseClicked",
    "mouseEntered", "mouseExited","mousePressed",
    "mouseReleased", "mouseDragged", "mouseMoved"
  };
  MiBoton
    b1 = new MiBoton(Color.blue, "test1"),
    b2 = new MiBoton(Color.red, "test2");
  class MiBoton extends JButton {
    void informa(String campo, String msg) {
      ((JTextField)h.get(campo)).setText(msg);
    }
    FocusListener fl = new FocusListener() {
      public void focusGained(FocusEvent e) {
        informar("focusGained", e.paramString());
```

[10] En Java 1.0/1.1 *no* se podía heredar de manera útil del objeto botón. Esto no era sino uno de los muchos fallos de diseño de que adolecía.

```
    }
    public void focusLost(FocusEvent e) {
      informar("focusLost", e.paramString());
    }
  };
  KeyListener kl = new KeyListener() {
    public void keyPressed(KeyEvent e) {
      informar("keyPressed", e.paramString());
    }
    public void keyReleased(KeyEvent e) {
      informar("keyReleased", e.paramString());
    }
    public void keyTyped(KeyEvent e) {
      informar("keyTyped", e.paramString());
    }
  };
  MouseListener ml = new MouseListener() {
    public void mouseClicked(MouseEvent e) {
      informar("mouseClicked", e.paramString());
    }
    public void mouseEntered(MouseEvent e) {
      informar("mouseEntered", e.paramString());
    }
    public void mouseExited(MouseEvent e) {
      informar("mouseExited", e.paramString());
    }
    public void mousePressed(MouseEvent e) {
      informar("mousePressed", e.paramString());
    }
    public void mouseReleased(MouseEvent e) {
      informar("mouseReleased", e.paramString());
    }
  };
  MouseMotionListener mml =
    new MouseMotionListener() {
    public void mouseDragged(MouseEvent e) {
      informar("mouseDragged", e.paramString());
    }
    public void mouseMoved(MouseEvent e) {
      informar("mouseMoved", e.paramString());
    }
  };
  public MiBoton(Color color, String etiqueta) {
    super(etiqueta);
    setBackground(color);
```

```
        addFocusListener(fl);
        addKeyListener(kl);
        addMouseListener(ml);
        addMouseMotionListener(mml);
      }
    }
    public void init() {
      Container c = getContentPane();
      c.setLayout(new GridLayout(evento.length+1,2));
      for(int i = 0; i < evento.length; i++) {
        JTextField t = new JTextField();
        t.setEditable(false);
        c.add(new JLabel(event[i], JLabel.RIGHT));
        c.add(t);
        h.put(evento[i], t);
      }
      c.add(b1);
      c.add(b2);
    }
    public static void main(String[] args) {
      Console.run(new RastrearEvento(), 700, 500);
    }
} ///:~
```

En el constructor de **MiBoton**, se establece el color del botón con una llamada a **setBackground( )**. Los oyentes están todos instalados con simples llamadas a métodos.

La clase **RastrearEvento** contiene un **HashMap** para guardar las cadenas que representan el tipo de evento y **JTextFields** donde se guarda la información sobre el evento. Por supuesto, éstos podrían haberse creado de forma estática en vez de ponerlos en un **HashMap**, pero creo que estará de acuerdo en que es mucho más fácil de usar y cambiar. En particular, si se necesita añadir o retirar un nuevo tipo de evento en **RastrearEvento**, simplemente se añadirá o retirará un String en el array **evento** —todo lo demás sucede automáticamente.

Cuando se invoca a **informar( )** se le da el nombre del evento y el parámetro String del evento. Usa el **HashMap h** en la clase externa para buscar el **JTextField** asociado con ese nombre de evento y después coloca el parámetro String en ese campo.

Es divertido ejecutar este ejemplo puesto que realmente se puede ver lo que está ocurriendo con los eventos del programa.

Un catálogo de componentes Swing

Ahora que se entienden los gestores de disposición y el modelo de eventos, ya podemos ver cómo se pueden usar los componentes Swing. Esta sección es un viaje no exhaustivo por los componen-

tes de Swing y las facetas que probablemente se usarán la mayoría de veces. Cada ejemplo pretende ser razonablemente pequeño, de forma que se pueda tomar el código y usarlo en los programas que cada uno desarrolle.

Se puede ver fácilmente qué aspecto tiene cada uno de los ejemplos al ejecutarlo viendo las páginas HTML en el código fuente descargable para este capítulo.

Mantenga en mente:

1. La documentación HTML de *java.sun.com* contiene todas las clases y métodos Swing (aquí sólo se muestran unos pocos).
2. Debido a la convención de nombres usada en los eventos Swing, es bastante fácil adivinar cómo escribir e instalar un manipulador para un tipo particular de evento. Se puede usar el programa de búsqueda **MostrarAddListeners.java** visto antes en este capítulo para ayudar a investigar un componente particular.
3. Cuando las cosas se complican habría que pasar a un constructor de IGU.

Botones

Swing incluye varios tipos de botones. Todos los botones, casillas de verificación, botones de opción e incluso los elementos de menú se heredan de **AbstractButton** (que, dado que incluye elementos de menú, se habría llamado probablemente "AbstractChooser" o algo igualmente general). En breve veremos el uso de elementos de menú, pero el ejemplo siguiente muestra los distintos tipos de botones disponibles:

```
//: c13:Botones.java
// Varios botones Swing.
// <applet code=Botones
//  width=350 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.plaf.basic.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Botones extends JApplet {
  JButton jb = new JButton("JButton");
  BasicArrowButton
    arriba = new BasicArrowButton(
      BasicArrowButton.NORTH),
    abajo = new BasicArrowButton(
      BasicArrowButton.SOUTH),
    derecha = new BasicArrowButton(
```

```
        BasicArrowButton.EAST),
      izquierda = new BasicArrowButton(
        BasicArrowButton.WEST);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(jb);
    cp.add(new JToggleButton("JToggleButton"));
    cp.add(new JCheckBox("JCheckBox"));
    cp.add(new JRadioButton("JRadioButton"));
    JPanel jp = new JPanel();
    jp.setBorder(new TitledBorder("Direcciones"));
    jp.add(arriba);
    jp.add(abajo);
    jp.add(izquierda);
    jp.add(derecha);
    cp.add(jp);
  }
  public static void main(String[] args) {
    Console.run(new Buttons(), 350, 100);
  }
} ///:~
```

Éste comienza con el **BasicArrowButton** de **javax.swing.plaf.basic**, después continúa con los diversos tipos específicos de botones. Al ejecutar el ejemplo, se verá que el botón de conmutación guarda su última posición, dentro o fuera. Pero las casillas de verificación y los botones de opción se comportan exactamente igual simplemente pulsando para activarlos o desactivarlos (ambos se heredan de **JToggleButton**).

Grupos de botones

Si se desea que varios botones de opción se comporten en forma de "or exclusivo o XOR", hay que añadirlos a un "grupo de botones". Pero, como demuestra el ejemplo de debajo, se puede añadir cualquier **AbstractButton** a un **ButtonGroup**.

Para evitar repetir mucho código, este ejemplo usa la reflectividad para generar los grupos de distintos tipos de botones. Esto se ve en **hacerBPanel( )**, que crea un grupo de botones y un **JPanel**. El segundo parámetro a **hacerBPanel( )** es un array de **String**. Por cada **String**, se añade un botón de la clase indicada por el primer argumento al **Janel**:

```
//: c13:GrupoBotones.java
// Usa la reflectividad para crear grupos
// de diferentes tipos de GrupoBotones.
// <applet code=GrupoBotones
//  width=500 height=300></applet>
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import java.lang.reflect.*;
import com.bruceeckel.swing.*;

public class GrupoBotones extends JApplet {
  static String[] ids = {
    "June", "Ward", "Beaver",
    "Wally", "Eddie", "Lumpy",
  };
  static JPanel
  hacerBPanel(Class bClass, String[] ids) {
    ButtonGroup bg = new ButtonGroup();
    JPanel jp = new JPanel();
    String titulo = bClass.getName();
    titulo = titulo.substring(
      titulo.lastIndexOf('.') + 1);
    jp.setBorder(new TitledBorder(titulo));
    for(int i = 0; i < ids.length; i++) {
      AbstractButton ab = new JButton("Fallo");
      try {
        // Lograr el método constructor dinámico
        // que toma un argumento String:
        Constructor ctor = bClass.getConstructor(
          new Class[] { String.class });
        // Crear un objeto nuevo:
        ab = (AbstractButton)ctor.newInstance(
          new Object[]{ids[i]});
      } catch(Exception ex) {
        System.err.println("no se puede crear " +
          bClass);
      }
      bg.add(ab);
      jp.add(ab);
    }
    return jp;
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(hacerBPanel(JButton.class, ids));
    cp.add(hacerBPanel(JToggleButton.class, ids));
    cp.add(hacerBPanel(JCheckBox.class, ids));
    cp.add(hacerBPanel(JRadioButton.class, ids));
```

```
  }
  public static void main(String[] args) {
    Console.run(new GrupoBotones(), 500, 300);
  }
} ///:~
```

El título del borde se toma del nombre de la clase, eliminando la información de trayectoria. El **AbstractButton** se inicializa a **JButton**, que tiene la etiqueta "Fallo" por lo que si se ignora el mensaje de excepción, se seguirá viendo el problema en pantalla. El método **getConstructor( )** produce un objeto **Constructor** que toma el array de argumentos de los tipos del array **Class** pasado a **getConstructor( )**. Después todo lo que se hace es llamar a **newInstance( )**, pasándole un array de **Object** que contiene los parámetros actuales —en este caso, simplemente el **String** del array **ids**.

Esto añade un poco de complejidad a lo que es un proceso simple. Para lograr comportamiento XOR con botones, se crea un grupo de botones y se añade cada botón para el que se desea ese comportamiento en el grupo. Cuando se ejecuta el programa, se verá que todos los botones excepto **JButton** exhiben este comportamiento "or exclusivo".

Iconos

Se puede usar un **Icon** dentro de un **JLabel** o cualquier cosa heredada de **AbstractButton** (incluyendo **JButton**, **JCheckBox**, **JRadioButton**, y los distintos tipos de **JMenuItem**). Utilizar **Icon**s con **JLabel**s es bastante directo (se verá un ejemplo más adelante). El ejemplo siguiente explora todas las formas adicionales de usar **Icon**s con botones y sus descendientes.

Se puede usar cualquier archivo **gif** que se desee, pero los que se usan en este ejemplo son parte de la distribución de código de este libro, disponible en *http://www.BruceEckel.com*. Para abrir un archivo e incorporar la imagen, simplemente se crea un **ImageIcon** y se le pasa el nombre del archivo. A partir de ese momento, se puede usar el **Icon** resultante en el programa.

Nótese que en este ejemplo, la información de trayectoria está codificada a mano; se necesitará cambiar la trayectoria para hacerla corresponder con la ubicación de los archivos de imágenes:

```
//: c13:Caras.java
// Comportamiento de Icon en JButtons.
// <applet code=Caras
//  width=250 height=100></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Caras extends JApplet {
  // La información de ruta siguiente es necesaria
  // para ejecutarse via un applet directamente desde el disco:
  static String ruta =
```

```
    "C:/aaa-TIJ2-distribution/code/c13/";
  static Icon[] caras = {
    new ImageIcon(rutas + "face0.gif"),
    new ImageIcon(rutas + "face1.gif"),
    new ImageIcon(rutas + "face2.gif"),
    new ImageIcon(rutas + "face3.gif"),
    new ImageIcon(rutas + "face4.gif"),
  };
  JButton
    jb = new JButton("JButton", caras[3]),
    jb2 = new JButton("Deshabilitar");
  boolean loco = false;
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    jb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(loco) {
          jb.setIcon(caras[3]);
          loco = false;
        } else {
          jb.setIcon(caras[0]);
          loco = true;
        }
        jb.setVerticalAlignment(JButton.TOP);
        jb.setHorizontalAlignment(JButton.LEFT);
      }
    });
    jb.setRolloverEnabled(true);
    jb.setRolloverIcon(caras[1]);
    jb.setPressedIcon(caras[2]);
    jb.setDisabledIcon(caras[4]);
    jb.setToolTipText("¡Yow!");
    cp.add(jb);
    jb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(jb.isEnabled()) {
          jb.setEnabled(false);
          jb2.setText("Habilitar");
        } else {
          jb.setEnabled(true);
          jb2.setText("Deshabilitar");
        }
      }
    });
```

```
    cp.add(jb2);
  }
  public static void main(String[] args) {
    Console.run(new Caras(), 400, 200);
  }
} ///:~
```

Se puede usar un **Icon** en muchos constructores, pero también se puede usar **setIcon( )** para añadir o cambiar un **Icon**. Este ejemplo también muestra cómo un **JButton** (o cualquier **AbstractButton**) puede establecer los distintos tipos de iconos que aparecen cuando le ocurren cosas a ese botón: cuando se presiona, se deshabilita o se "pasa sobre él" (el ratón se mueve sobre él sin hacer clic). Se verá que esto da a un botón una imagen animada genial.

Etiquetas de aviso

El ejemplo anterior añadía una "etiqueta de aviso" al botón. Casi todas las clases que se usen para crear interfaces de usuario se derivan de **JComponent**, que contiene un método denominado **setToolTipText(String)**. Por tanto, para casi todo lo que se coloque en un formulario, todo lo que se necesita hacer es decir (siendo **jc** el objeto de cualquier clase derivada **JComponent**):

```
jc.setToolTipText("Mi etiqueta");
```

y cuando el ratón permanece sobre ese **JComponent**, durante un periodo de tiempo predeterminado, aparecerá una pequeña caja con el texto junto al puntero del ratón.

Campos de texto

Este ejemplo muestra el comportamiento extra del que son capaces los **JTextField**s:

```
//: c13:CamposTexto.java
// Campos de texto y eventos Java.
// <applet code=CamposTexto width=375
// height=125></applet>
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class CamposTexto extends JApplet {
  JButton
    b1 = new JButton("Leer Texto"),
    b2 = new JButton("Poner Texto");
  JTextField
    t1 = new JTextField(30),
```

```
    t2 = new JTextField(30),
    t3 = new JTextField(30);
  String s = new String();
  DocumentoMayusculas
    ucd = new DocumentoMayusculas();
  public void init() {
    t1.setDocument(ucd);
    ucd.addDocumentListener(new T1());
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    DocumentListener dl = new T1();
    t1.addActionListener(new T1A());
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(b1);
    cp.add(b2);
    cp.add(t1);
    cp.add(t2);
    cp.add(t3);
  }
  class T1 implements DocumentListener {
    public void changedUpdate(DocumentEvent e){}
    public void insertUpdate(DocumentEvent e){
      t2.setText(t1.getText());
      t3.setText("Texto: "+ t1.getText());
    }
    public void removeUpdate(DocumentEvent e){
      t2.setText(t1.getText());
    }
  }
  class T1A implements ActionListener {
    private int conteo = 0;
    public void actionPerformed(ActionEvent e) {
      t3.setText("t1 Action Event " + conteo++);
    }
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      if(t1.getSelectedText() == null)
        s = t1.getText();
      else
        s = t1.getSelectedText();
      t1.setEditable(true);
    }
  }
```

```
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      ucd.setUpperCase(false);
      t1.setText("Insertado por el Boton 2: " + s);
      ucd.setUpperCase(true);
      t1.setEditable(false);
    }
  }
  public static void main(String[] args) {
    Console.run(new CamposTexto(), 375, 125);
  }
}

class DocumentoMayusculas extends PlainDocument {
  boolean mayusculas = true;
  public void setUpperCase(boolean flag) {
    mayusculas = flag;
  }
  public void insertString(int offset,
    String string, AttributeSet attributeSet)
    throws BadLocationException {
      if(mayusculas)
        string = string.toUpperCase();
      super.insertString(offset,
        string, attributeSet);
  }
} ///:~
```

El **JTextField t3** se incluye como un lugar a reportar cuando se dispare el oyente del **JTextField t1**. Se verá que el oyente de la acción de un **JTextField** se dispara sólo al presionar la tecla "enter".

El **JTextField t1** tiene varios oyentes asignados. El **T1** es un **DocumentListener** que responde a cualquier cambio en el "documento" (en este caso los contenidos de **JTextField**). Copia automáticamente todo el texto de **t1** a **t2**. Además, el documento de **t1** se pone a una clase derivada de **PlainDocument**, llamada **DocumentoMayusculas**, que fuerza a que todos sus caracteres sean mayúsculas. Detecta automáticamente los espacios en blanco y lleva a cabo los borrados, ajustando los intercalados y gestionando todo como cabría esperar.

Bordes

JComponent contiene un método denominado **setBorder( )**, que permite ubicar varios bordes interesantes en cualquier componente visible. El ejemplo siguiente demuestra varios de los distintos bordes disponibles, utilizando un método denominado **showBorder( )** que crea un **JPanel** y pone el borde en cada caso. También usa RTTI para averiguar el nombre del borde que se está usando

(eliminando toda información de trayectoria), y pone después ese nombre en un **JLabel** en el medio del panel:

```
//: c13:Bordes.java
// Bordes Swing diferentes.
// <applet code=Bordes
//  width=500 height=300></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Bordes extends JApplet {
  static JPanel mostrarBorde(Border b) {
    JPanel jp = new JPanel();
    jp.setLayout(new BorderLayout());
    String nm = b.getClass().toString();
    nm = nm.substring(nm.lastIndexOf('.') + 1);
    jp.add(new JLabel(nm, JLabel.CENTER),
      BorderLayout.CENTER);
    jp.setBorder(b);
    return jp;
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.setLayout(new GridLayout(2,4));
    cp.add(mostrarBorde(new TitledBorder("Titulo")));
    cp.add(mostrarBorde(new EtchedBorder()));
    cp.add(mostrarBorde(new LineBorder(Color.blue)));
    cp.add(mostrarBorde(
      new MatteBorder(5,5,30,30,Color.green)));
    cp.add(mostrarBorde(
      new BevelBorder(BevelBorder.RAISED)));
    cp.add(mostrarBorde(
      new SoftBevelBorder(BevelBorder.LOWERED)));
    cp.add(mostrarBorde(new CompoundBorder(
      new EtchedBorder(),
      new LineBorder(Color.red))));
  }
  public static void main(String[] args) {
    Console.run(new Bordes(), 500, 300);
  }
} ///:~
```

También se pueden crear bordes personalizados y ponerlos dentro de botones, etiquetas, etc. —cualquier cosa derivada de **JComponent**.

JScrollPanes

La mayoría de las veces simplemente se deseará dejar que un **JScrollPane** haga su trabajo, pero también se pueda controlar qué barras de desplazamiento están permitidas —la vertical, la horizontal, ambas o ninguna:

```
//: c13:JScrollPanes.java
// Controlando las barras de desplazamiento en un JScrollPane.
// <applet code=JScrollPanes width=300 height=725>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class JScrollPanes extends JApplet {
  JButton
    b1 = new JButton("Area Texto 1"),
    b2 = new JButton("Area Texto 2"),
    b3 = new JButton("Reemplazar Texto"),
    b4 = new JButton("Insertar Texto");
  JTextArea
    t1 = new JTextArea("t1", 1, 20),
    t2 = new JTextArea("t2", 4, 20),
    t3 = new JTextArea("t3", 1, 20),
    t4 = new JTextArea("t4", 10, 10),
    t5 = new JTextArea("t5", 4, 20),
    t6 = new JTextArea("t6", 10, 10);
  JScrollPane
    sp3 = new JScrollPane(t3,
      JScrollPane.VERTICAL_SCROLLBAR_NEVER,
      JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
    sp4 = new JScrollPane(t4,
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
      JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
    sp5 = new JScrollPane(t5,
      JScrollPane.VERTICAL_SCROLLBAR_NEVER,
      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS),
    sp6 = new JScrollPane(t6,
      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
```

```
  class B1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t5.append(t1.getText() + "\n");
    }
  }
  class B2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t2.setText("Insertado por el Boton 2");
      t2.append(": " + t1.getText());
      t5.append(t2.getText() + "\n");
    }
  }
  class B3L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String s = " Reemplazo ";
      t2.replaceRange(s, 3, 3 + s.length());
    }
  }
  class B4L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t2.insert(" Insertado ", 10);
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    // Crear Bordes para los componentes:
    Border brd = BorderFactory.createMatteBorder(
      1, 1, 1, 1, Color.black);
    t1.setBorder(brd);
    t2.setBorder(brd);
    sp3.setBorder(brd);
    sp4.setBorder(brd);
    sp5.setBorder(brd);
    sp6.setBorder(brd);
    // Inicializar los oyentes y añadir componentes:
    b1.addActionListener(new B1L());
    cp.add(b1);
    cp.add(t1);
    b2.addActionListener(new B2L());
    cp.add(b2);
    cp.add(t2);
    b3.addActionListener(new B3L());
    cp.add(b3);
    b4.addActionListener(new B4L());
```

```
    cp.add(b4);
    cp.add(sp3);
    cp.add(sp4);
    cp.add(sp5);
    cp.add(sp6);
  }
  public static void main(String[] args) {
    Console.run(new JScrollPanes(), 300, 725);
  }
} ///:~
```

Utilizar argumentos distintos en el constructor **JScrollPane** permite controlar las barras de desplazamiento disponibles. Este ejemplo también adorna un poco los distintos elementos usando bordes.

Un minieditor

El control **JTextPane** proporciona un gran soporte a la edición sin mucho esfuerzo. El ejemplo siguiente hace un uso muy sencillo de esto, ignorando el grueso de la funcionalidad de la clase:

```
//: c13:PanelTexto.java
// El control JTextPane es un pequeño editor.
// <applet code=PanelTexto width=475 height=425>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;
import com.bruceeckel.util.*;

public class PanelTexto extends JApplet {
  JButton b = new JButton("Añadir Texto");
  JTextPane tp = new JTextPane();
  static Generator sg =
    new Arrays2.RandStringGenerator(7);
  public void init() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        for(int i = 1; i < 10; i++)
          tp.setText(tp.getText() +
            sg.next() + "\n");
      }
    });
    Container cp = getContentPane();
    cp.add(new JScrollPane(tp));
```

```
    cp.add(BorderLayout.SOUTH, b);
  }
  public static void main(String[] args) {
    Console.run(new PanelTexto(), 475, 425);
  }
} ///:~
```

El botón simplemente añade texto generado al azar. La intención del **JTextPane** es permitir editar texto *in situ*, de forma que se verá que no hay método **append( )**. En este caso (hay que admitir que se trata de un uso pobre de las capacidades de **JTextPane**), debemos capturar el texto, modificarlo y volverlo a ubicar en su sitio utilizando **setText( )**.

Como se mencionó anteriormente, el comportamiento de la disposición por defecto de un *applet* es usar el **BorderLayout**. Si se añade algo al panel sin especificar más detalles, simplemente se rellena el centro del panel hasta los bordes. Sin embargo, si se especifica una de las regiones que le rodean (NORTH, SOUTH, EAST o WEST) como se hace aquí, el componente se encajará en esa región —en este caso, el botón se anidará abajo, en la parte inferior de la pantalla.

Fíjese en las facetas incluidas en **JTextPane**, como la envoltura automática de líneas. Usando la documentación JDK es posible buscar otras muchas facetas.

Casillas de verificación

Una casilla de verificación proporciona una forma sencilla de hacer una elección de tipo activado/desactivado; consiste en una pequeña caja y una etiqueta. La caja suele guardar una pequeña "x" (o alguna otra indicación que se establezca), o está vacía, en función de si el elemento está o no seleccionado.

Normalmente se creará un **JCheckBox** utilizando un constructor que tome la etiqueta como parámetro. Se puede conseguir y establecer su estado, y también la etiqueta si se desea leerla o cambiarla una vez creado el **JCheckBox**.

Siempre que se asigne valor o se limpie un **JCheckBox**, se da un evento que se puede capturar de manera análoga a como se hace con un botón, utilizando un **ActionListener**. El ejemplo siguiente usa un **JTextArea** para enumerar todas las casillas de verificación seleccionadas:

```
//: c13:CasillasVerificacion.java
// Uso de JCheckBox.
// <applet code=CasillasVerificacion width=200 height=200>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class CasillasVerificacion extends JApplet {
```

```
  JTextArea t = new JTextArea(6, 15);
  JCheckBox
    cb1 = new JCheckBox("Check Box 1"),
    cb2 = new JCheckBox("Check Box 2"),
    cb3 = new JCheckBox("Check Box 3");
  public void init() {
    cb1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        rastrear("1", cb1);
      }
    });
    cb2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        rastrear("2", cb2);
      }
    });
    cb3.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        rastrear("3", cb3);
      }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JScrollPane(t));
    cp.add(cb1);
    cp.add(cb2);
    cp.add(cb3);
  }
  void rastrear(String b, JCheckBox cb) {
    if(cb.isSelected())
      t.append("Box " + b + " Establecido\n");
    else
      t.append("Box " + b + " Limpiado\n");
  }
  public static void main(String[] args) {
    Console.run(new CasillasVerificacion(), 200, 200);
  }
} ///:~
```

El método **Rastrear( )** envía el nombre del **JCheckBox** seleccionado y su estado actual al **JTextArea** utilizando **append( )**, por lo que se verá una lista acumulativa de casillas de verificación seleccionadas y cuál es su estado.

Botones de opción

El concepto de botón de opción en programación de IGU proviene de las radios de coche preelectrónicas con botones mecánicos: cuando se oprimía un botón cualquier otro botón que estuviera pulsado, saltaba. Por consiguiente, permite forzar una elección única entre varias.

Todo lo que se necesita para establecer un grupo asociado de **JRadioButton**s es añadirlos a un **ButtonGroup** (se puede tener cualquier número de **ButtonGroup**s en un formulario). Uno de los botones puede tener su valor inicial por defecto a **true** (utilizando el segundo argumento del constructor). Si se intenta establecer más de un botón de opción a **true** sólo quedará con este valor el último al que se le asigne.

He aquí un ejemplo simple del uso de botones de opción. Nótese que se pueden capturar eventos de botones de opción, al igual que con los otros:

```
//: c13:BotonesOpcion.java
// Usando JRadioButtons.
// <applet code=BotonesOpcion
// width=200 height=100> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class BotonesOpcion extends JApplet {
  JTextField t = new JTextField(15);
  ButtonGroup g = new ButtonGroup();
  JRadioButton
    rb1 = new JRadioButton("uno", false),
    rb2 = new JRadioButton("dos", false),
    rb3 = new JRadioButton("tres", false);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      t.setText("Radio button " +
        ((JRadioButton)e.getSource()).getText());
    }
  };
  public void init() {
    rb1.addActionListener(al);
    rb2.addActionListener(al);
    rb3.addActionListener(al);
    g.add(rb1); g.add(rb2); g.add(rb3);
    t.setEditable(false);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
```

```
    cp.add(rb1);
    cp.add(rb2);
    cp.add(rb3);
  }
  public static void main(String[] args) {
    Console.run(new BotonesOpcion(), 200, 100);
  }
} ///:~
```

Para mostrar el estado, se usa un campo de texto. Este campo se pone a no editable pues se usa para mostrar datos, no para recogerlos. Por consiguiente, es una alternativa al uso de una **JLabel**.

Combo boxes (listas desplegables)

Al igual que un grupo de botones de opción, una lista desplegable es una forma de obligar al usuario a seleccionar sólo un elemento a partir de un grupo de posibles elementos. Sin embargo, es una forma más compacta de lograrlo, y es más fácil cambiar los elementos de la lista sin soprender al usuario. (Se pueden cambiar botones de opción dinámicamente, pero esto tiende a ser demasiado visible.)

El **JComboBox** de Java no es como el cuadro combinado de Windows, que permite seleccionar a partir de una lista *o* tipo de la propia selección. Con un **JComboBox** se puede elegir uno y sólo un elemento de la lista. En el ejemplo siguiente, la **JComboBox** comienza con cierto número de entradas, añadiéndosele otras cuando se presiona un botón:

```
//: c13:CuadrosCombinados.java
// Usando listas desplegables.
// <applet code=CuadrosCombinados
// width=200 height=100> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class CuadrosCombinados extends JApplet {
  String[] descripcion = { "Bullicioso", "Obtuso",
    "Recalcitrante", "Brillante", "Somnoliento",
    "Temoroso", "Florido", "Putrefacto" };
  JTextField t = new JTextField(15);
  JComboBox c = new JComboBox();
  JButton b = new JButton("Añadir elementos");
  int conteo = 0;
  public void init() {
    for(int i = 0; i < 4; i++)
      c.addItem(descripcion[conteo++]);
    t.setEditable(false);
    b.addActionListener(new ActionListener() {
```

```
        public void actionPerformed(ActionEvent e){
          if(conteo < descripcion.length)
            c.addItem(descripcion[conteo++]);
        }
      });
      c.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
          t.setText("indice: "+ c.getSelectedIndex()
            + "    " + ((JComboBox)e.getSource())
            .getSelectedItem());
        }
      });
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      cp.add(t);
      cp.add(c);
      cp.add(b);
    }
    public static void main(String[] args) {
      Console.run(new CuadrosCombinados(), 200, 100);
    }
} ///:~
```

La **JTextField** muestra los "índices seleccionados", que es la secuencia del elemento actualmente seleccionado, así como la etiqueta del botón de opción.

Listas

Estas cajas son significativamente diferentes de las **JComboBox**, y no sólo en apariencia. Mientras que una **JComboBox** se despliega al activarla, una **JList** ocupa un número fijo de líneas en la pantalla todo el tiempo y no cambia. Si se desea ver los elementos de la lista, simplemente se invoca a **getSelectedValues( )**, que produce un array de **String** de los elementos seleccionados.

Una **JList** permite selección múltiple: si se hace control-clic en más de un elemento (manteniendo pulsada la tecla "control" mientras que se llevan a cabo varios clics de ratón) el elemento original sigue resaltado y se puede seleccionar tantos como se desee. Si se selecciona un elemento, y después se pulsa mayúsculas-clic en otro elemento, se seleccionan todos los elementos comprendidos entre ambos. Para eliminar un elemento de un grupo se puede hacer control-clic sobre él.

```
//: c13:Lista.java
// <applet code=Lista width=250
// height=375> </applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.border.*;
import com.bruceeckel.swing.*;

public class Lista extends JApplet {
  String[] sabores = { "Chocolate", "Fresa",
    "Fundido de Vanilla", "Galleta de Menta",
    "Moka con Almendras", "Amanecer de Ron",
    "Crema de Praline", "Pastel de Barro" };
  DefaultListModel elementos=new DefaultListModel();
  JList lst = new JList(elementos);
  JTextArea t = new JTextArea(sabores.length,20);
  JButton b = new JButton("Añadir elemento");
  ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      if(conteo < sabores.length) {
        lElementos.add(0, sabores[conteo++]);
      } else {
        // Deshabilitar, puesto que no hay
        // más sabores que añadir a la Lista
        b.setEnabled(false);
      }
    }
  };
  ListSelectionListener ll =
    new ListSelectionListener() {
      public void valueChanged(
        ListSelectionEvent e) {
          t.setText("");
          Object[] elementos=lst.getSelectedValues();
          for(int i = 0; i < elementos.length; i++)
            t.append(elementos[i] + "\n");
        }
    };
  int conteo = 0;
  public void init() {
    Container cp = getContentPane();
    t.setEditable(false);
    cp.setLayout(new FlowLayout());
    // Crear Bordes para los componentes:
    Border brd = BorderFactory.createMatteBorder(
      1, 1, 2, 2, Color.black);
    lst.setBorder(brd);
    t.setBorder(brd);
    // Añadir los primeros cuatro elementos a la lista
    for(int i = 0; i < 4; i++)
```

```
        lElementos.addElement(sabores[conteo++]);
      // Añadir elementos al Panel Contenedor para ser mostrados
      cp.add(t);
      cp.add(lst);
      cp.add(b);
      // Registrar los oyentes de eventos
      lst.addListSelectionListener(ll);
      b.addActionListener(bl);
    }
    public static void main(String[] args) {
      Console.run(new Lista(), 250, 375);
    }
} ///:~
```

Cuando se presiona el botón, añade elementos a la parte *superior* de la lista (porque el segundo argumento de **addItem( )** es 0).

Podemos ver que también se han añadido bordes a las listas.

Si se desea poner un array de **String**s en una **JList**, hay una solución mucho más simple: se pasa el array al constructor **JList**, y construye la lista automáticamente. La única razón para usar el "modelo lista" en el ejemplo de arriba es que la lista puede ser manipulada durante la ejecución del programa.

Las **JList**s no proporcionan soporte directo para el desplazamiento. Por supuesto, todo lo que hay que hacer es envolver la **JList** en un **JScrollPane** y se logrará la gestión automática de todos los detalles.

Paneles Tabulados

El **JTabbedPane** permite crear un "diálogo con lengüetas", que tiene lengüetas de carpetas de archivos ejecutándose a través de un borde, y todo lo que hay que hacer es presionar una lengüeta para presentar un diálogo diferente:

```
//: c13:PanelTabulado1.java
// Demuestra el Panel Tabulado.
// <applet code=PanelTabulado1
// width=350 height=200> </applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class PanelTabulado1 extends JApplet {
  String[] sabores = { "Chocolate", "Fresa",
    "Fundido de Vainilla", "Galleta de Menta",
```

```
      "Moka con Almendras", "Amanecer de Ron",
      "Crema de Praline", "Pastel de Barro" };
  JTabbedPane tabs = new JTabbedPane();
  JTextField txt = new JTextField(20);
  public void init() {
    for(int i = 0; i < sabores.length; i++)
      tabs.addTab(sabores[i],
        new JButton("Panel Tabulado " + i));
    tabs.addChangeListener(new ChangeListener(){
      public void stateChanged(ChangeEvent e) {
        txt.setText("Tab seleccionado: " +
          tabs.getSelectedIndex());
      }
    });
    Container cp = getContentPane();
    cp.add(BorderLayout.SOUTH, txt);
    cp.add(tabs);
  }
  public static void main(String[] args) {
    Console.run(new PanelTabulado1(), 350, 200);
  }
} ///:~
```

En Java, el uso de algún tipo de mecanismo de "paneles tabulados" es bastante importante pues en programación de *applets* se suele intentar desmotivar el uso de diálogos emergentes añadiendo automáticamente un pequeño aviso a cualquier diálogo emergente de un *applet*.

Cuando se ejecute el programa se verá que el **JTabbedPane** comprime las lengüetas si hay demasiadas, de forma que quepan en una fila. Podemos ver esto redimensionando la ventana al ejecutar el programa desde la línea de comandos de la consola.

Cajas de mensajes

Los entornos de ventanas suelen contener un conjunto estándar de cajas de mensajes que permiten enviar información al usuario rápidamente, o capturar información proporcionada por éste. En Swing, estas cajas de mensajes se encuentran en **JOptionPane**. Hay muchas posibilidades diferentes (algunas bastante sofisticadas), pero las que más habitualmente se usan son probablemente el diálogo mensaje y el diálogo confirmación, invocados usando el **static JOptionPanel.showMessageDialog( )** y **JOptionPane.showConfirmDialog( )**. El ejemplo siguiente muestra un subconjunto de las cajas de mensajes disponibles con **JOptionPane**:

```
//: c13:CajasMensajes.java
// Demuestra JOptionPane.
// <applet code=CajasMensajes
// width=200 height=150> </applet>
import javax.swing.*;
```

```
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class CajasMensajes extends JApplet {
  JButton[] b = { new JButton("Alerta"),
    new JButton("Si/No"), new JButton("Color"),
    new JButton("Entrada"), new JButton("3 Vals")
  };
  JTextField txt = new JTextField(15);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      String id =
        ((JButton)e.getSource()).getText();
      if(id.equals("Alerta"))
        JOptionPane.showMessageDialog(null,
          "¡Ahí hay un fallo!", "¡Hey!",
          JOptionPane.ERROR_MESSAGE);
      else if(id.equals("Si/No"))
        JOptionPane.showConfirmDialog(null,
          "o no", "elegir si",
          JOptionPane.YES_NO_OPTION);
      else if(id.equals("Color")) {
        Object[] opciones = { "Rojo", "Verde" };
        int sel = JOptionPane.showOptionDialog(
          null, "¡Elija un Color!", "Precaucion",
          JOptionPane.DEFAULT_OPTION,
          JOptionPane.WARNING_MESSAGE, null,
          opciones, opciones[0]);
          if(sel != JOptionPane.CLOSED_OPTION)
            txt.setText(
              "Color Seleccionado: " + opciones[sel]);
      } else if(id.equals("Entrada")) {
        String val = JOptionPane.showInputDialog(
            "¿Cuantos dedos ves?");
        txt.setText(val);
      } else if(id.equals("3 Vals")) {
        Object[] selecciones = {
          "Primero", "Segundo", "Tercero" };
        Object val = JOptionPane.showInputDialog(
          null, "Elegir una", "Entrada",
          JOptionPane.INFORMATION_MESSAGE,
          null, selecciones, selecciones[0]);
        if(val != null)
          txt.setText(
```

```
                val.toString());
        }
      }
    };
    public void init() {
      Container cp = getContentPane();
      cp.setLayout(new FlowLayout());
      for(int i = 0; i < b.length; i++) {
        b[i].addActionListener(al);
        cp.add(b[i]);
      }
      cp.add(txt);
    }
    public static void main(String[] args) {
      Console.run(new CajasMensajes(), 200, 200);
    }
} ///:~
```

Para poder escribir un único **ActionListener**, he usado un enfoque con riesgo, comprobando las etiquetas **String** de los botones. El problema de todo esto es que es fácil provocar algún fallo con las etiquetas en el uso de mayúsculas, y este fallo podría ser difícil de localizar.

Nótese que **showOptionDialog( )** y **showInputDialog( )** proporcionan objetos de retorno que contienen el valor introducido por el usuario.

Menús

Cada componente capaz de guardar un menú, incluyendo **JApplet**, **JFrame**, **JDialog** y sus descendientes, tiene un método **setMenuBar( )** que acepta un **JMenuBar** (sólo se puede tener un **JMenuBar** en un componente particular). Se añaden **JMenus** al **JMenuBar**, y **JMenuItems** a los **JMenus**. Cada **JMenuItem** puede tener un **ActionListener** asociado, que se dispara al seleccionar ese elemento del menú.

A diferencia de un sistema basado en el uso de recursos, con Java y Swing hay que ensamblar a mano todos los menús en código fuente. He aquí un ejemplo de menú sencillo:

```
//: c13:MenusSimples.java
// <applet code=MenusSimples
// width=200 height=75> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;
```

```
public class MenusSimples extends JApplet {
  JTextField t = new JTextField(15);
  ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent e){
      t.setText(
        ((JMenuItem)e.getSource()).getText());
    }
  };
  JMenu[] menus = { new JMenu("Winken"),
    new JMenu("Blinken"), new JMenu("Nod") };
  JMenuItem[] elementos = {
    new JMenuItem("Fee"), new JMenuItem("Fi"),
    new JMenuItem("Fo"),  new JMenuItem("Zip"),
    new JMenuItem("Zap"), new JMenuItem("Zot"),
    new JMenuItem("Olly"), new JMenuItem("Oxen"),
    new JMenuItem("Free") };
  public void init() {
    for(int i = 0; i < elementos.length; i++) {
      elementos[i].addActionListener(al);
      menus[i%3].add(elementos[i]);
    }
    JMenuBar mb = new JMenuBar();
    for(int i = 0; i < menus.length; i++)
      mb.add(menus[i]);
    setJMenuBar(mb);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
  }
  public static void main(String[] args) {
    Console.run(new MenusSimples(), 200, 75);
  }
} ///:~
```

El uso del operador módulo en "**i%3**" distribuye los elementos de menú entre los tres **JMenus**. Cada **JMenuItem** debe tener un **ActionListener** adjunto; aquí se usa el mismo **ActionListener** en todas partes pero generalmente será necesario uno individual para cada **JMenuItem**.

JMeuItem hereda de **AbstractButton**, por lo que tiene algunos comportamientos propios de los botones. Por sí mismo, proporciona un elemento que se puede ubicar en un menú desplegable. También hay tres tipos heredados de **JMenuItem**: **JMenu** para albergar otros **JMenuItems** (de forma que se pueden tener menús en cascada), **JCheckBoxMenuItem**, que produce una marca que indica si se ha seleccionado o no ese elemento de menú, y **JRadioButtonMenuItem**, que contiene un botón de opción.

Como ejemplo más sofisticado, he aquí el de los sabores de helado de nuevo, utilizado para crear menús. Este ejemplo también muestra menús en cascada, mnemónicos de teclado, **JCheckBox MenuItems**, y la forma de cambiar menús dinámicamente:

```
//: c13:Menus.java
// Submenús, elementos de menú casillas de verificación, menús intercambiables,
// mnemónicos (atajos) y comandos de acción.
// <applet code=Menus width=300
// height=100> </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Menus extends JApplet {
  String[] sabores = { "Chocolate", "Fresa",
    "Fundido de Vainilla", "Galleta de Menta",
    "Moka con Almendras", "Amanecer de Ron",
    "Crema de Praline", "Pastel de Barro" };
  JTextField t = new JTextField("No sabor", 30);
  JMenuBar mb1 = new JMenuBar();
  JMenu
    f = new JMenu("Fichero"),
    m = new JMenu("Sabores"),
    s = new JMenu("Seguridad");
  // Enfoque alternativo:
  JCheckBoxMenuItem[] seguridad = {
    new JCheckBoxMenuItem("Guardar"),
    new JCheckBoxMenuItem("Ocultar")
  };
  JMenuItem[] archivo = {
    new JMenuItem("Abrir"),
  };
  // Una segunda barra de menú a la que cambiar a:
  JMenuBar mb2 = new JMenuBar();
  JMenu fooBar = new JMenu("fooBar");
  JMenuItem[] otro = {
    // Añadir un atajo de menú (mnemónico) es muy
    // simple, pero sólo los JMenuItems pueden tenerlos en
    // sus constructores:
    new JMenuItem("Foo", KeyEvent.VK_F),
    new JMenuItem("Bar", KeyEvent.VK_A),
    // Sin atajo:
    new JMenuItem("Baz"),
  };
```

```
  JButton b = new JButton("Intercambiar Menus");
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuBar m = getJMenuBar();
      setJMenuBar(m == mb1 ? mb2 : mb1);
      validate(); // Refrescar el frame
    }
  }
  class ML implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuItem objetivo = (JMenuItem)e.getSource();
      String comandoAccion =
        objetivo.getActionCommand();
      if(comandoAccion.equals("Abrir")) {
        String s = t.getText();
        boolean elegido = false;
        for(int i = 0; i < sabores.length; i++)
          if(s.equals(sabores[i])) elegido = true;
        if(!elegido)
          t.setText("¡Elegir un sabor primero!");
        else
          t.setText("Abriendo "+ s +". Mmm, mm!");
      }
    }
  }
  class FL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JMenuItem objetivo = (JMenuItem)e.getSource();
      t.setText(objetivo.getText());
    }
  }
  // Alternativamente, se puede crear una clase
  // diferente por cada MenuItem diferente. Después no hay
  // que averiguar cuál es:
  class FooL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Foo seleccionado");
    }
  }
  class BarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText("Bar seleccionado");
    }
  }
  class BazL implements ActionListener {
```

```
    public void actionPerformed(ActionEvent e) {
      t.setText("Baz seleccionado");
    }
  }
  class CMIL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      JCheckBoxMenuItem objetivo =
        (JCheckBoxMenuItem)e.getSource();
      String comandoAccion =
        target.getActionCommand();
      if(comandoAccion.equals("Guardar"))
        t.setText("¡Guardar el helado! " +
          "Guardando esta a " + objetivo.getState());
      else if(comandoAccion.equals("Ocultar"))
        t.setText("¡Esconder el Helado! " +
          "¿Esta frio? " + objetivo.getState());
    }
  }
  public void init() {
    ML ml = new ML();
    CMIL cmil = new CMIL();
    seguridad[0].setActionCommand("Guardar");
    seguridad[0].setMnemonic(KeyEvent.VK_G);
    seguridad[0].addItemListener(cmil);
    seguridad[1].setActionCommand("Ocultar");
    seguridad[0].setMnemonic(KeyEvent.VK_H);
    seguridad[1].addItemListener(cmil);
    otro[0].addActionListener(new FooL());
    otro[1].addActionListener(new BarL());
    otro[2].addActionListener(new BazL());
    FL fl = new FL();
    for(int i = 0; i < sabores.length; i++) {
      JMenuItem mi = new JMenuItem(sabores[i]);
      mi.addActionListener(fl);
      m.add(mi);
      // Añadir separadores a intervalos:
      if((i+1) % 3 == 0)
        m.addSeparator();
    }
    for(int i = 0; i < seguridad.length; i++)
      s.add(seguridad[i]);
    s.setMnemonic(KeyEvent.VK_A);
    f.add(s);
    f.setMnemonic(KeyEvent.VK_F);
    for(int i = 0; i < archivo.length; i++) {
```

```
        archivo[i].addActionListener(fl);
        f.add(archivo[i]);
      }
      mb1.add(f);
      mb1.add(m);
      setJMenuBar(mb1);
      t.setEditable(false);
      Container cp = getContentPane();
      cp.add(t, BorderLayout.CENTER);
      // Establecer el sistema para menús intercambiables:
      b.addActionListener(new BL());
      b.setMnemonic(KeyEvent.VK_S);
      cp.add(b, BorderLayout.NORTH);
      for(int i = 0; i < otro.length; i++)
        fooBar.add(otro[i]);
      fooBar.setMnemonic(KeyEvent.VK_B);
      mb2.add(fooBar);
    }
    public static void main(String[] args) {
      Console.run(new Menus(), 300, 100);
    }
} ///:~
```

En este programa hemos ubicado los elementos del menú en arrays y después se recorre cada array invocando a **add( )** por cada **JMenuItem**. De esta forma se logra que la adición o substracción de un elemento de menú sea bastante menos tediosa.

Este programa crea no uno, sino dos **JMenuBar** para demostrar que se pueden intercambiar barras de menú activamente mientras se ejecuta el programa. Se puede ver cómo un **JMenuBar** está hecho de **JMenus**, y cada **JMenu** está hecho de **JMenuItems**, **JCheckBoxMenuItems**, o incluso otros **JMenus** (logrando así submenús). Cuando se ensambla un **JMenuBar**, éste puede instalarse en el programa actual con el método **setJMenuBar( )**. Nótese que al presionar el botón, se comprueba qué menú está actualmente instalado invocando a **getJMenuBar( )**, y pone la otra barra de menú en su sitio.

Al probar "Abrir" nótese que el deletreo y el uso de mayúsculas es crítico, pero Java no señala ningún error si no hay coincidencia exacta con "Abrir". Este tipo de comparación de cadenas de caracteres es una fuente de errores de programación.

La comprobación y la no comprobación de los elementos de menú se lleva a cabo automáticamente. El código que gestiona los **JCheckBoxMenuItems** muestran dos formas de determinar lo que se comprobó: la comprobación de cadenas de caracteres (que, como se mencionó arriba, no es un enfoque muy seguro, aunque se verá a menudo) y la coincidencia de todos los objetos destino de eventos. Como se ha mostrado, el método **getState( )** puede usarse para revelar el estado. También se puede cambiar el estado de un **JCheckBoxMenuItem** con **setState( )**.

Los eventos de los menús son un poco inconsistentes y pueden conducir a confusión: los **JMenuItems** usan **ActionListeners**, pero los **JCheckboxMenuItems** usan **ItemListeners**. Los ob-

jetos **JMenu** también pueden soportar **ActionListener**s, pero eso no suele ser de ayuda. En general, se adjuntarán oyentes a cada **JMenuItem**, **JCheckBoxMenuItem** o **JRadioButtonMenuItem**, pero el ejemplo muestra **ItemListener**s y **ActionListener**s adjuntados a los distintos componentes menú.

Swing soporta mnemónicos o "atajos de teclado", de forma que se puede seleccionar cualquier cosa derivada de **AbstractButton** (botón, elemento de menú, etc.) utilizando el teclado en vez del ratón. Éstos son bastante simples: para **JMenuItem** se puede usar el constructor sobrecargado que toma como segundo argumento el identificador de la clave. Sin embargo, la mayoría de **AbstractButton**s no tiene constructores como éste, por lo que la forma más general de solucionar el problema es usar el método **setMnemonic( )**. El ejemplo de arriba añade mnemónicos al botón y a algunos de los elementos de menús; los indicadores de atajos aparecen automáticamente en los componentes.

También se puede ver el uso de **setActionCommand( )**. Éste parece un poco extraño porque en cada caso el "comando de acción" es exactamente el mismo que la etiqueta en el componente del menú. ¿Por qué no usar simplemente la etiqueta en vez de esta cadena de caracteres alternativa? El problema es la internacionalización. Si se vuelve a direccionar el programa a otro idioma, sólo se deseará cambiar la etiqueta del menú, y no cambiar el código (lo cual podría sin duda introducir nuevos errores). Por ello, para que esto sea sencillo para el código que comprueba la cadena de texto asociada a un componente de menú, se puede mantener inmutable el "comando de acción" a la vez que se puede cambiar la etiqueta de menú. Todo el código funciona con el "comando de acción", pero no se ve afectada por los cambios en las etiquetas de menú. Nótese que en este programa, no se examinan los comandos de acción de todos los componentes del menú, por lo que los no examinados no tienen su comando de acción.

La mayoría del trabajo se da en los oyentes. **BL** lleva a cabo el intercambio de **JMenuBar**. En **ML**, se toma el enfoque de "averigua quién llama" logrando la fuente del **ActionEvent** y convirtiéndola en un **JMenuItem**, consiguiendo después la cadena de caracteres del comando de acción para pasarlo a través de una sentencia **if** en cascada.

El oyente **FL** es simple, incluso aunque esté gestionando todos los sabores distintos del menú de sabores. Este enfoque es útil para tomar el enfoque usado con **FooL**, **BarL** y **BazL**, en los que sólo se les ajunta un componente menú de forma que no es necesaria ninguna lógica extra de detección y se sabe exactamente quién invocó al oyente. Incluso con la profusión de clases generadas de esta manera, el código interno tiende a ser menor y el proceso es más a prueba de torpes.

Se puede ver que el código de menú se vuelve largo y complicado rápidamente. Éste es otro caso en el que la solución apropiada es usar un constructor de IGU. Una buena herramienta también gestionará el mantenimiento de menús.

Menús emergentes

La forma más directa de implementar un **JPopupMenu** es crear una clase interna que extienda **MouseAdapter**, después añadir un objeto de esa clase interna a cada componente que se desea para producir un comportamiento emergente:

```
//: c13:Emergente.java
// Creando menús emergentes con Swing.
// <applet code=Emergente
//  width=300 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class Emergente extends JApplet {
  JPopupMenu emergente = new JPopupMenu();
  JTextField t = new JTextField(10);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    ActionListener al = new ActionListener() {
      public void actionPerformed(ActionEvent e){
        t.setText(
          ((JMenuItem)e.getSource()).getText());
      }
    };
    JMenuItem m = new JMenuItem("Aquí");
    m.addActionListener(al);
    emergente.add(m);
    m = new JMenuItem("Yon");
    m.addActionListener(al);
    emergente.add(m);
    m = new JMenuItem("Afar");
    m.addActionListener(al);
    emergente.add(m);
    emergente.addSeparator();
    m = new JMenuItem("Permanecer aqui");
    m.addActionListener(al);
    emergente.add(m);
    OyenteEmergente pl = new OyenteEmergente();
    addMouseListener(pl);
    t.addMouseListener(pl);
  }
  class OyenteEmergente extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      quizasMostrarEmergente(e);
    }
    public void mouseReleased(MouseEvent e) {
      quizasMostrarEmergente(e);
```

```
    }
    private void quizasMostrarEmergente(MouseEvent e) {
      if(e.isPopupTrigger()) {
        emergente.show(
          e.getComponent(), e.getX(), e.getY());
      }
    }
  }
  public static void main(String[] args) {
    Console.run(new Emergente(), 300, 200);
  }
} ///:~
```

Se añade el mismo **ActionListener** a cada **JMenuItem**, de forma que tome el texto de la etiqueta de menú y lo inserte en el **JTextField**.

Generación de dibujos

En un buen marco de trabajo de IGU, la generación de dibujos debería ser razonablemente sencilla —y lo es, en la biblioteca Swing. El problema del ejemplo de dibujo es que los cálculos que determinan dónde van las cosas son bastante más complicados que las rutinas a las llamadas de generación de dibujos, y estos cálculos suelen mezclarse junto con las llamadas a dibujos de forma que puede parecer que la interfaz es más complicada que lo que es en realidad.

Por simplicidad, considérese el problema de representar datos en la pantalla —aquí, los datos los proporcionará el método **Math.sin( )** incluido que es la función matemática seno. Para hacer las cosas un poco más interesantes, y para demostrar más allá lo fácil que es utilizar componentes Swing, se puede colocar un deslizador en la parte de abajo del formulario para controlar dinámicamente el número de ondas cíclicas sinoidales que se muestran. Además, si se redimensiona la ventana, se verá que la onda seno se reajusta al nuevo tamaño de la ventana.

Aunque se puede pintar cualquier **JComponent**, y usarlo, por consiguiente, como lienzo, si simplemente se desea una superficie de dibujo, generalmente se heredará de un **JPanel** (es el enfoque más directo). Sólo hay que superponer el método **paintComponent( )**, que se invoca siempre que hay que repintar ese componente (generalmente no hay que preocuparse por esto pues se encarga Swing). Cuando es invocado, Swing le pasa un objeto **Graphics**, que puede usarse para dibujar o pintar en la superficie.

En el ejemplo siguiente, toda la inteligencia de pintado está en la clase **DibujarSeno**; la clase **OndaSeno** simplemente configura el programa y el control de deslizamiento. Dentro de **DibujarSeno**, el método **establecerCiclos( )** proporciona la posiblidad de permitir a otro objeto —en este caso el control de deslizamientos— controlar el número de ciclos.

```
//: c13:OndaSeno.java
// Dibujando con Swing, usando un JSlider.
// <applet code=OndaSeno
```

```
//  width=700 height=400></applet>
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class DibujarSeno extends JPanel {
  static final int FACTORESCALADO = 200;
  int ciclos;
  int puntos;
  double[] senos;
  int[] pts;
  DibujarSeno() { establecerCiclos(5); }
  public void establecerCiclos(int numCiclos) {
    ciclos = numCiclos;
    puntos = FACTORESCALADO * ciclos * 2;
    senos = new double[puntos];
    pts = new int[puntos];
    for(int i = 0; i < puntos; i++) {
      double radianes = (Math.PI/FACTORESCALADO) * i;
      senos[i] = Math.sin(radianes);
    }
    repaint();
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    int anchuraMax = getWidth();
    double pasoH = (double)anchuraMax/(double)puntos;
    int alturaMax = getHeight();
    for(int i = 0; i < puntos; i++)
      pts[i] = (int)(senos[i] * alturaMax/2 * .95
                     + alturaMax/2);
    g.setColor(Color.red);
    for(int i = 1; i < puntos; i++) {
      int x1 = (int)((i - 1) * pasoH);
      int x2 = (int)(i * pasoH);
      int y1 = pts[i-1];
      int y2 = pts[i];
      g.drawLine(x1, y1, x2, y2);
    }
  }
}

public class OndaSeno extends JApplet {
  DibujarSeno senos = new DibujarSeno();
```

```
  JSlider ciclos = new JSlider(1, 30, 5);
  public void init() {
    Container cp = getContentPane();
    cp.add(senos);
    ciclos.addChangeListener(new ChangeListener(){
      public void stateChanged(ChangeEvent e) {
        senos.establecerCiclos(
          ((JSlider)e.getSource()).getValue());
      }
    });
    cp.add(BorderLayout.SOUTH, ciclos);
  }
  public static void main(String[] args) {
    Console.run(new OndaSeno(), 700, 400);
  }
} ///:~
```

Todos los miembros de datos y arrays se usan en el cálculo de los puntos de la onda senoidal: **ciclos** indica el número de ondas senoidales completas deseadas; **puntos** contiene el número total de puntos a dibujar, **senos** contiene los valores de la función seno, y **pts** contiene las coordenadas *y* de los puntos a dibujar en el **JPanel**. El método **establecerCiclos( )** crea los arrays de acuerdo con el número de puntos deseados y rellena el array **senos** con números. Llamar a **repaint( )**, fuerza a **establecerCiclos( )** a que se invoque a **paintComponent( )**, de forma que se lleven a cabo el resto de cálculos y redibujado.

Lo primero que hay que hacer al superponer **paintComponent( )** es invocar a la versión de la clase base del método. Después se puede hacer lo que se desee; normalmente, esto implica usar los métodos **Graphics** que se pueden encontrar en la documentación de **java.awt.Graphics** (en la documentación HTML de *http://java.sun.com*) para dibujar y pintar píxeles en el **JPanel**. Aquí, se puede ver que casi todo el código está involucrado en llevar a cabo los cálculos; de hecho, las dos únicas llamadas a métodos que manipulan la pantalla son **setColor( )** y **drawLine( )**. Probablemente se tendrá una experiencia similar al crear programas que muestren datos gráficos —se invertirá la mayor parte del tiempo en averiguar qué es lo que se desea dibujar, mientras que el proceso de dibujado en sí será bastante simple.

Cuando creamos este programa, la mayoría del tiempo se invirtió en mostrar la onda seno en la pantalla. Una vez que lo logramos, pensamos que sería bonito poder cambiar dinámicamente el número de ciclos. Mis experiencias programando al intentar hacer esas cosas en otros lenguajes, me hicieron dudar un poco antes de acometerlo, pero resultó ser la parte más fácil del proyecto. Creamos un **JSlider** (los argumentos son valores más a la izquierda del **JSlider**, los de más a la derecha y el valor de comienzo, respectivamente, pero también hay otros constructores) y lo volcamos en el **JApplet**. Después miramos en la documentación HTML y descubrimos que el único oyente era **addChangeListener**, que fue disparado siempre que se cambiase el *deslizador* lo suficiente como para producir un valor diferente. El único método para esto era el **stateChanged( )**, de nombre obvio, que proporcionó un objeto **ChangeEvent**, de forma que podía mirar hacia atrás a la fuente del cambio y encontrar el nuevo valor. Llamando a **establecerCiclos( )** del objeto **senos**, se incorporó el nuevo valor y se redibujó el **JPanel**.

En general, se verá que la mayoría de los problemas Swing se pueden solucionar siguiendo un proceso similar, y se averiguará que suele ser bastante simple, incluso si nunca antes se ha usado un componente particular.

Si el problema es más complejo, hay otras alternativas de dibujo más sofisticadas, incluyendo componentes JavaBeans de terceras partes y el API Java 2D. Estas soluciones se escapan del alcance de este libro, pero habría que investigarlo cuando el código de dibujo se vuelve demasiado oneroso.

Cajas de diálogo

Una caja de diálogo es una ventana que saca otra ventana. Su propósito es tratar con algún aspecto específico sin desordenar la ventana original con esos detalles. Las cajas de diálogo se usan muy intensivamente en entornos de programación con ventanas, pero se usan menos frecuentemente en los *applets*.

Para crear una caja de diálogo, se hereda de **JDialog**, que es simplemente otra clase de **Window**, como **JFrame**. Un **JDialog** tiene un gestor de disposiciones (por defecto **BorderLayout**) y se le añaden oyentes para que manipulen eventos. Una diferencia significativa al llamar a **windowClosing( )** es que no se desea apagar la aplicación. En vez de esto, se liberan los recursos usados por la ventana de diálogos llamando a **dispose( )**. He aquí un ejemplo muy sencillo:

```
//: c13:Dialogos.java
// Creando y usando Cajas de Diálogo.
// <applet code=Dialogos width=125 height=75>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

class MiDialogo extends JDialog {
  public MiDialogo(JFrame parent) {
    super(parent, "Mi dialogo", true);
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JLabel("He aqui mi dialogo"));
    JButton ok = new JButton("OK");
    ok.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        dispose(); // Cierra el diálogo
      }
    });
    cp.add(ok);
    setSize(150,125);
```

```
    }
}

public class Dialogos extends JApplet {
  JButton b1 = new JButton("Caja de dialogo");
  MiDialogo dlg = new MiDialogo(null);
  public void init() {
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        dlg.show();
      }
    });
    getContentPane().add(b1);
  }
  public static void main(String[] args) {
    Console.run(new Dialogos(), 125, 75);
  }
} ///:~
```

Una vez que se crea el **JDialog**, se debe llamar al método **show( )** para mostrarlo y activarlo. Para que se cierre el diálogo hay que llamar a **dispose( )**.

Veremos que cualquier cosa que surja de un objeto, incluyendo las cajas de diálogo, "no es de confianza". Es decir, se obtienen advertencias al usarlas. Esto es porque, en teoría, sería posible confundir al usuario y hacerle pensar que está tratando con una aplicación nativa regular y hacer que tecleen el número de su tarjeta de crédito que viajará después por la Web. Un *applet* siempre viaja adjunto a una página web, y será visible desde un navegador, mientas que las cajas de diálogo se desasocian —por lo que sería posible, en teoría. Como resultado no suele ser frecuente ver *applets* que hagan uso de cajas de diálogo.

El ejemplo siguiente es más complejo; la caja de diálogo consta de una rejilla (usando **GridLayout**) de un tipo especial de botón definido como clase **BotonToe**. Este botón dibuja un marco en torno a sí mismo y, dependiendo de su estado, un espacio en blanco, una "x" o una "o" en el medio. Comienza en blanco y después, dependiendo de a quién le toque, cambia a "x" o a "o". Sin embargo, también cambiará entre "x" y "o" al hacer clic en el botón. (Esto convierte el concepto tic-tac-toe en sólo un poco más asombroso de lo que ya es.) Además, se puede establecer que la caja de diálogo tenga un número cualquiera de filas y columnas, cambiando los números de la ventana de aplicación principal.

```
//: c13:TicTacToe.java
// Demostración de las cajas de diálogo
// y creación de componentes propios.
// <applet code=TicTacToe
//  width=200 height=100></applet>
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class TicTacToe extends JApplet {
  JTextField
    filas = new JTextField("3"),
    cols = new JTextField("3");
  static final int BLANCO = 0, XX = 1, OO = 2;
  class DialogoToe extends JDialog {
    int turno = XX; // Comenzar poniendo a "x"
    // w = número de celdas de ancho
    // h = número de celdas de alto
    public DialogoToe(int w, int h) {
      setTitle("El juego en si mismo");
      Container cp = getContentPane();
      cp.setLayout(new GridLayout(w, h));
      for(int i = 0; i < w * h; i++)
        cp.add(new BotonToe());
      setSize(w * 50, h * 50);
      // Diálogo de cierre de JDK 1.3:
      //#setDefaultCloseOperation(
      //#   DISPOSE_ON_CLOSE);
      // Diálogo de cierre de JDK 1.2:
      addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e){
          dispose();
        }
      });
    }
    class BotonToe extends JPanel {
      int estado = BLANCO;
      public BotonToe() {
        addMouseListener(new ML());
      }
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int x1 = 0;
        int y1 = 0;
        int x2 = getSize().width - 1;
        int y2 = getSize().height - 1;
        g.drawRect(x1, y1, x2, y2);
        x1 = x2/4;
        y1 = y2/4;
        int ancho = x2/2;
        int alto = y2/2;
```

```
        if(estado == XX) {
          g.drawLine(x1, y1,
            x1 + ancho, y1 + alto);
          g.drawLine(x1, y1 + alto,
            x1 + ancho, y1);
        }
        if(estado == OO) {
          g.drawOval(x1, y1,
            x1 + ancho/2, y1 + alto/2);
        }
      }
      class ML extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
          if(estado == BLANK) {
            estado = turno;
            turno = (turno == XX ? OO : XX);
          }
          else
            estado = (estado == XX ? OO : XX);
          repaint();
        }
      }
    }
  }
  class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JDialog d = new DialogoToe(
        Integer.parseInt(pilas.getText()),
        Integer.parseInt(cols.getText()));
      d.setVisible(true);
    }
  }
  public void init() {
    JPanel p = new JPanel();
    p.setLayout(new GridLayout(2,2));
    p.add(new JLabel("Filas", JLabel.CENTER));
    p.add(Pilas);
    p.add(new JLabel("Columnas", JLabel.CENTER));
    p.add(cols);
    Container cp = getContentPane();
    cp.add(p, BorderLayout.NORTH);
    JButton b = new JButton("Comenzar");
    b.addActionListener(new BL());
    cp.add(b, BorderLayout.SOUTH);
  }
```

```
  public static void main(String[] args) {
    Console.run(new TicTacToe(), 200, 100);
  }
} ///:~
```

Dado que los **static**s sólo pueden estar en el nivel externo de la clase, las clases internas no pueden tener datos **static** o clases internas **static**.

El método **paintComponent( )** dibuja el cuadrado alrededor del panel y la "x" o la "o". Esto implica muchos cálculos tediosos pero es directo.

El **MouseListener** captura los eventos de ratón, y comprueba en primer lugar si el panel ha escrito algo. Si no, se invoca a la ventana padre para averiguar a quién le toca, y qué se usa para establecer el estado del **BotonToe**. Vía el mecanismo de clases internas, posteriormente el **BotonToe** vuelve al padre y cambia el turno. Si el botón ya está mostrando una "x" o una "o", se cambia. En estos cálculos se puede ver el uso del "if-else" ternario descrito en el Capítulo 3. Después de un cambio de estado, se repinta el **BotonToe**.

El constructor de **DialogoToe** es bastante simple: añade en un **GridLayout** tantos botones como se solicite, y después lo redimensiona a 50 píxeles de lado para cada botón.

TicTacToe da entrada a toda la aplicación creando los **JTextFields** (para la introducción de las filas y columnas de la rejilla de botones) y el botón "comenzar" con su **ActionListener**. Cuando se presiona este botón se recogen todos los datos de los **JTextField**s, y puesto que se encuentran en forma de **String**s, se convierten en **int**s usando el método **static Integer.parseInt( )**.

Diálogos de archivo

Algunos sistemas operativos tienen varias cajas de diálogo pre-construidas para manejar la selección de ciertos elementos como fuentes, colores, impresoras, etc. Generalmente, todos los sistemas operativos gráficos soportan la apertura y salvado de archivos, sin embargo, el **JFileChooser** de Java encapsula estas operaciones para facilitar su uso.

La aplicación siguiente ejercita dos formas de diálogos **JFileChooser**, uno para abrir y otro para guardar. La mayoría del código debería parecer ya familiar al lector, y es en los oyentes de acciones de ambos clics sobre los botones donde se dan las operaciones más interesantes:

```
//: c13:PruebaElectorArchivo.java
// Demostración de cajas de diálogo Archivo.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class PruebaElectorArchivo extends JFrame {
  JTextField
    nombreArchivo = new JTextField(),
```

```
    dir = new JTextField();
  JButton
    abrir = new JButton("Abrir"),
    salvar = new JButton("Salvar");
  public PruebaElectorArchivo() {
    JPanel p = new JPanel();
    abrir.addActionListener(new AbrirL());
    p.add(abrir);
    salvar.addActionListener(new SalvarL());
    p.add(salvar);
    Container cp = getContentPane();
    cp.add(p, BorderLayout.SOUTH);
    dir.setEditable(false);
    nombreArchivo.setEditable(false);
    p = new JPanel();
    p.setLayout(new GridLayout(2,1));
    p.add(nombreArchivo);
    p.add(dir);
    cp.add(p, BorderLayout.NORTH);
  }
  class AbrirL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JFileChooser c = new JFileChooser();
      // Demostrar el diálogo "Abrir":
      int rVal =
        c.showOpenDialog(PruebaElectorArchivo.this);
      if(rVal == JFileChooser.APPROVE_OPTION) {
        nombreArchivo.setText(
          c.getSelectedFile().getName());
          dir.setText(
            c.getCurrentDirectory().toString());
      }
      if(rVal == JFileChooser.CANCEL_OPTION) {
        nombreArchivo.setText("Presiono Cancelar");
        dir.setText("");
      }
    }
  }
  class SalvarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      JFileChooser c = new JFileChooser();
      // Demostrar el diálogo "Salvar" :
      int rVal =
        c.showSaveDialog(PruebaElectorArchivo.this);
      if(rVal == JFileChooser.APPROVE_OPTION) {
```

```
            nombreArchivo.setText(
              c.getSelectedFile().getName());
              dir.setText(
                c.getCurrentDirectory().toString());
          }
          if(rVal == JFileChooser.CANCEL_OPTION) {
            nombreArchivo.setText("Presiono Cancelar");
            dir.setText("");
          }
        }
      }
      public static void main(String[] args) {
        Console.run(new PruebaElectorArchivo(), 250, 110);
      }
    } ///:~
```

Nótese que se pueden aplicar muchas variaciones a **JFileChooser**, incluyendo filtros para limitar los nombres de archivo permitidos.

Para un diálogo abrir archivo se puede invocar a **showOpenDialog( )**, y en el caso de salvar archivo el diálogo al que se invoca es **showSaveDialog( )**. Estos comandos no devuelven nada hasta cerrar el diálogo. El objeto **JFileChooser** sigue existiendo, pero se pueden leer datos del mismo. Los métodos **getSelectedFile( )** y **getCurrentDirectory( )** son dos formas de interrogar por los resultados de la operación. Si devuelven **null** significa que el usuario canceló el diálogo.

HTML en componentes Swing

Cualquier componente que pueda tomar texto, también puede tomar texto HTML, al que dará formato de acuerdo a reglas HTML. Esto significa que se puede añadir texto elegante a los componentes Swing fácilmente. Por ejemplo:

```
/: c13:BotonHTML.java
// Poniendo texto HTML en componentes Swing.
// <applet code=BotonHTML width=200 height=500>
// </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class BotonHTML extends JApplet {
  JButton b = new JButton("<html><b><font size=+2>" +
    "<center>¡Hola!<br><i>¡Presioneme ahora!");
  public void init() {
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
```

```
            getContentPane().add(new JLabel("<html>"+
              "<i><font size=+4>Kapow!"));
            // Forzar la redistribución para
            // incluir la nueva etiqueta:
            validate();
          }
        });
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b);
      }
      public static void main(String[] args) {
        Console.run(new BotonHTML(), 200, 500);
      }
} ///:~
```

Hay que empezar el texto con "<html>" y después se pueden usar etiquetas HTML normales. Nótese que no hay obligación de incluír las etiquetas de cierre habituales.

El **ActionListener** añade una etiqueta **JLabel** nueva al formulario, que también contiene texto HTML. Sin embargo, no se añade esta etiqueta durante **init( )** por lo que hay que llamar al método **validate( )** del contenedor para forzar una redistribución de los componentes (y por consiguiente mostrar la nueva etiqueta).

También se puede usar texto HTML para **JTabbedPane**, **JMenuItem**, **JToolTip**, **JRadioButton** y **JCheckBox**.

Deslizadores y barras de progreso

Un deslizador (que ya se ha usado en el ejemplo de la onda senoidal) permite al usuario introducir datos moviéndose hacia delante y hacia atrás, lo que es intuitivo en algunas situaciones (por ejemplo, controles de volumen). Una barra de progreso muestra datos en forma relativa, desde "lleno" hasta "vacío" de forma que el usuario obtiene una perspectiva. Nuestro ejemplo favorito para éstos es simplemente vincular el deslizador a la barra de progreso de forma que al mover uno el otro varíe en consecuencia:

```
//: c13:Progreso.java
// Usando barras de progreso y deslizadoras.
// <applet code=Progreso
//  width=300 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;
import javax.swing.border.*;
import com.bruceeckel.swing.*;
```

```
public class Progreso extends JApplet {
  JProgressBar pb = new JProgressBar();
  JSlider sb =
    new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(2,1));
    cp.add(pb);
    sb.setValue(0);
    sb.setPaintTicks(true);
    sb.setMajorTickSpacing(20);
    sb.setMinorTickSpacing(5);
    sb.setBorder(new TitledBorder("Deslizame"));
    pb.setModel(sb.getModel()); // Compartir modelo
    cp.add(sb);
  }
  public static void main(String[] args) {
    Console.run(new Progreso(), 300, 200);
  }
} ///:~
```

La clave para vincular juntos ambos elementos es compartir su modelo en la línea:

```
pb.setModel(sb.getModel());
```

Por supuesto, también se podría controlar ambos usando un oyente, pero esto es más directo en situaciones simples.

El **JProgressBar** es bastante directo, pero el **JSlider** tiene un montón de opciones, como la orientación y las marcas de mayor y menor. Nótese lo directo que es añadir un borde.

Árboles

Utilizar un **JTree** puede ser tan simple como decir:

```
add(new JTree(
  new Object[] {"este", "ese", "aquel"}));
```

Esto muestra un árbol primitivo. Sin embargo, el API de los árboles es vasto —ciertamente uno de los mayores de Swing. Parece que casi se puede hacer cualquier cosa con los árboles, pero tareas más sofisticadas podrían requerir de bastante información y experimentación.

Afortunadamente, hay un terreno neutral en la biblioteca: los componentes árbol "por defecto", que generalmente hacen lo que se necesita. Por tanto, la mayoría de las veces se pueden usar estos componentes, y sólo hay que profundizar en los árboles en casos especiales.

El ejemplo siguiente usa los componentes árbol "por defecto" para mostrar un árbol en un *applet*. Al presionar el botón, se añade un nuevo subárbol bajo el nodo actualmente seleccionado (si no se selecciona ninguno se usa el nodo raíz):

```
//: c13:Arboles.java
// Árbol ejemplo simple de Árbol Swing. Se pueden hacer
// árboles mucho más complejos que éste.
// <applet code=Arboles
//  width=250 height=250></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.tree.*;
import com.bruceeckel.swing.*;

// Toma un array de Strings y convierte
// el primer elemento en nodo y los demás en hojas:
class Rama {
  DefaultMutableTreeNode r;
  public Rama (String[] datos) {
    r = new DefaultMutableTreeNode(data[0]);
    for(int i = 1; i < datos.length; i++)
      r.add(new DefaultMutableTreeNode(datos[i]));
  }
  public DefaultMutableTreeNode nodo() {
    return r;
  }
}

public class Arboles extends JApplet {
  String[][] datos = {
    { "Colores", "Rojo", "Azul", "Verde" },
    { "Sabores", "Acido", "Dulce", "Suave" },
    { "Longitud", "Corto", "Medio", "Largo" },
    { "Volumen", "Alto", "Medio", "Bajo" },
    { "Temperatura", "Alta", "Media", "Baja" },
    { "Intensidad", "Alta", "Media", "Baja" },
  };
  static int i = 0;
  DefaultMutableTreeNode raiz, hijo, elegido;
  JTree arbol;
  DefaultTreeModel modelo;
  public void init() {
    Container cp = getContentPane();
    raiz = new DefaultMutableTreeNode("raiz");
```

```
    arbol = new JTree(raiz);
    // Añadirlo y hacer que se encargue del desplazamiento:
    cp.add(new JScrollPane(arbol),
      BorderLayout.CENTER);
    // Capturar el modelo de árbol:
    modelo =(DefaultTreeModel)arbol.getModel();
    JButton prueba = new JButton("Pulsame");
    prueba.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e){
        if(i < datos.length) {
          hijo = new rama(datos[i++]).nodo();
          // ¿Cuál es el último sobre el que se hizo clic?
          elegido = (DefaultMutableTreeNode)
            hijo.getLastSelectedPathComponent();
          if(elegido == null) elegido = raiz;
          // El modelo creara el evento
          // apropiado. En respuesta, el
          // árbol se actualizará a sí mismo:
          modelo.insertNodeInto(hijo, elegido, 0);
          // Esto pone el nuevo nodo en el nodo
          // actualmente seleccionado.
        }
      }
    });
    // Cambiar los colores de los botones:
    prueba.setBackground(Color.blue);
    prueba.setForeground(Color.white);
    JPanel p = new JPanel();
    p.add(prueba);
    cp.add(p, BorderLayout.SOUTH);
  }
  public static void main(String[] args) {
    Console.run(new Arboles(), 250, 250);
  }
} ///:~
```

La primera clase, **Rama**, es una herramienta para tomar un array de **String** y construir un **DefaultMutableTreeNode** con el primer **String** como raíz y el resto de los **String**s del array como hojas. Después se puede llamar a **nodo( )** para producir la raíz de esta "rama".

La clase **Arboles** contiene un array bi-dimensional de **String**s a partir del cual se pueden construir **Ramas**, y una **static int i** para recorrer este array. Los objetos **DefaultMutableTreeNode** guardan los nodos, pero la representación física en la pantalla la controla el **JTree** y su modelo asociado, el **DefaultTreeModel**. Nótese que cuando se añade el **JTree** al *applet*, se envuelve en un **JScrollPane** —esto es todo lo necesario para el desplazamiento automático.

El **JTree** lo controla su *modelo*. Cuando se hace un cambio al modelo, éste genera un evento que hace que el **JTree** desempeñe cualquier actualización necesaria a la representación visible del árbol. En **init( )**, se captura el modelo llamando a **getModel( )**. Cuando se presiona el botón, se crea una nueva "rama". Después, se encuentra el componente actualmente seleccionado (o se usa la raíz si es que no hay ninguno) y el método **insertNodeInto( )** del modelo hace todo el trabajo de cambiar el árbol y actualizarlo.

Un ejemplo como el de arriba puede darnos lo que necesitamos de un árbol. Sin embargo, los árboles tienen la potencia de hacer casi todo lo que se pueda imaginar —en todas partes donde se ve la expresión "por defecto" dentro del ejemplo, podría sustituirse por otra clase para obtener un comportamiento diferente. Pero hay que ser conscientes: casi todas estas clases tienen grandes interfaces, por lo que se podría invertir mucho tiempo devanándonos los sesos para entender los aspectos intrínsecos de los árboles. Independientemente de esto, constituyen un buen diseño y cualquier alternativa es generalmente mucho peor.

Tablas

Al igual que los árboles, las tablas de Swing son vastas y potentes. Su intención principal era ser la interfaz "rejilla" popular para bases de datos vía Conectividad de Bases de Datos Java (JDBC, *Java DataBase Connectivity*, que se estudiará en el Capítulo 15), y por consiguiente, tienen una gran flexibilidad, a cambio de una carga de complejidad. Aquí hay materia suficiente como para un gran rompecabezas, e incluso para justificar la escritura de un libro entero. Sin embargo, también es posible crear una **JTable** relativamente simple si se entienden las bases.

La **JTable** controla la forma de mostrar los datos, pero el **TableModel** controla los datos en sí. Por ello, para crear una tabla, se suele crear primero el **TableModel**. Se puede implementar la interfaz **TableModel** al completo pero suele ser más simple heredar de la clase ayudante **AbstractTable Model**:

```
//: c13:Tabla.java
// Demostración simple de JTable.
// <applet code=Tabla
//  width=350 height=200></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.table.*;
import javax.swing.event.*;
import com.bruceeckel.swing.*;

public class Tabla extends JApplet {
  JTextArea txt = new JTextArea(4, 20);
  // El TableModel controla todos los datos:
  class ModeloDatos extends AbstractTableModel {
    Object[][] datos = {
```

```
      {"uno", "dos", "tres", "cuatro"},
      {"cinco", "seis", "siete", "ocho"},
      {"nueve", "diez", "once", "doce"},
    };
    // Imprime los datos cuando la tabla cambia:
    class TML implements TableModelListener {
      public void tableChanged(TableModelEvent e){
        txt.setText(""); // Limpiarlo
        for(int i = 0; i < datos.length; i++) {
          for(int j = 0; j < datos[0].length; j++)
            txt.append(datos[i][j] + " ");
          txt.append("\n");
        }
      }
    }
    public ModeloDatos() {
      addTableModelListener(new TML());
    }
    public int getColumnCount() {
      return datos[0].length;
    }
    public int getRowCount() {
      return datos.length;
    }
    public Object getValueAt(int row, int col) {
      return datos[row][col];
    }
    public void
    setValueAt(Object val, int row, int col) {
      datos[row][col] = val;
      // Indica que se produjo el cambio:
      fireTableDataChanged();
    }
    public boolean
    isCellEditable(int row, int col) {
      return true;
    }
  }
  public void init() {
    Container cp = getContentPane();
    JTable tabla = new JTable(new ModeloDatos());
    cp.add(new JScrollPane(tabla));
    cp.add(BorderLayout.SOUTH, txt);
  }
  public static void main(String[] args) {
```

```
    Console.run(new Tabla(), 350, 200);
  }
} ///:~
```

ModeloDatos contiene un array de datos, pero también se podrían obtener los datos a partir de otra fuente, como una base de datos. El constructor añade un **TableModelListener** que imprime el array cada vez que se cambia la tabla. El resto de métodos siguen la convención de nombres de los Beans, y **JTable** los usa cuando quiere presentar la información en **ModeloDatos**. **AbstractTable Model** proporciona métodos por defecto para **setValueAt( )** e **isCellEditable( )** que evitan cambios a los datos, por lo que si se desea poder editar los datos, hay que superponer estos métodos.

Una vez que se tiene un **TableModel**, simplemente hay que pasárselo al constructor **JTable**. Se encargará de todos los detalles de presentación, edición y actualización. Este ejemplo también pone la **JTable** en un **JScrollPane**.

Seleccionar la Apariencia

Uno de los aspectos más interesantes de Swing es la "Apariencia conectable". Ésta permite al programa emular la apariencia y comportamiento de varios entornos operativos. Incluso se puede hacer todo tipo de cosas elegantes como cambiar dinámicamente la apariencia y el comportamiento mientras se está ejecutando el programa. Sin embargo, por lo general simplemente se desea una de las dos cosas, o seleccionar un aspecto y comportamiento "multiplataformas" (el "metal" del Swing), o seleccionar el aspecto y comportamiento del sistema en el que se está, de forma que el programa Java parece haber sido creado específicamente para ese sistema. El código para seleccionar entre estos comportamientos es bastante simple —pero hay que ejecutarlo *antes* de crear cualquier componente visual, puesto que los componentes se construirán basados en la apariencia actual y no se cambiarán simplemente porque se cambie la apariencia y el comportamiento a mitad de camino durante el programa (ese proceso es más complicado y fuera de lo común, y está relegado a libros específicos de Swing).

De hecho, si se desea usar el aspecto y comportamiento multiplataformas ("metal") característico de los programas Swing, no hay que hacer nada —es la opción por defecto. Pero si en vez de ello se desea usar el aspecto y comportamiento del entorno operativo actual, simplemente se inserta el código siguiente, generalmente el principio del **main( )** pero de cualquier forma antes de añadir componentes:

```
try {
  UIManager.setLookAndFeel(UIManager.
    getSystemLookAndFeelClassName());
} catch(Exception e) {}
```

No es necesario poner nada en la cláusula **catch** porque el **UIManager** redireccionará por defecto al aspecto y comportamiento multiplataforma si fallan los intentos de optar por otra alternativa. Sin embargo, durante la depuración, puede ser útil la excepción, pues al menos se puede desear poner una sentencia de impresión en la cláusula *catch*.

He aquí un programa que toma un parámetro de línea de comandos para seleccionar un comportamiento y una imagen, y que muestra la apariencia de varios de estos comportamientos:

```
//: c13:Apariencia.java
// Seleccionando apariencias diferentes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class Apariencia extends JFrame {
  String[] opciones = {
    "eeny", "meeny", "minie", "moe", "toe", "you"
  };
  Component[] pruebas = {
    new JButton("JButton"),
    new JTextField("JTextField"),
    new JLabel("JLabel"),
    new JCheckBox("JCheckBox"),
    new JRadioButton("Radio"),
    new JComboBox(opciones),
    new JList(opciones),
  };
  public Apariencia() {
    super("Apariencia");
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    for(int i = 0; i < pruebas.length; i++)
      cp.add(pruebas[i]);
  }
  private static void errorUso() {
    System.out.println(
      "Uso:Apariencia [cross|system|motif]");
    System.exit(1);
  }
  public static void main(String[] args) {
    if(args.length == 0) errorUso();
    if(args[0].equals("cross")) {
      try {
        UIManager.setLookAndFeel(UIManager.
          getCrossPlatformLookAndFeelClassName());
      } catch(Exception e) {
          e.printStackTrace(System.err);
      }
```

```
    } else if(args[0].equals("system")) {
      try {
        UIManager.setLookAndFeel(UIManager.
          getSystemLookAndFeelClassName());
      } catch(Exception e) {
          e.printStackTrace(System.err);
      }
    } else if(args[0].equals("motif")) {
      try {
        UIManager.setLookAndFeel("com.sun.java."+
          "swing.plaf.motif.MotifLookAndFeel");
      } catch(Exception e) {
          e.printStackTrace(System.err);
      }
    } else errorUso();
    // Nótese que la apariencia debe establecerse
    // antes de crear cualquier componente.
    Console.run(new Apariencia(), 300, 200);
  }
} ///:~
```

Se puede ver que una de las opciones es especificar explícitamente una cadena de caracteres para un aspecto y un comportamiento, como se ha visto con **MotifLookAndFeel**. Sin embargo, ése y el "metal" por defecto son los únicos que se pueden usar legalmente en cualquier plataforma; incluso aunque hay cadenas de caracteres para los aspectos y comportamientos de Windows y Macintosh, éstos sólo pueden usarse en sus plataformas respectivas (se producen al invocar a **getSystemLook AndFeelClassName( )** estando en esa plataforma particular).

También es posible crear un paquete de "apariencia" personalizado, por ejemplo, si se está construyendo un marco de trabajo para una compañía que quiere una apariencia distinta. Éste es un gran trabajo y se escapa con mucho del alcance de este libro (¡de hecho, se descubrirá que se escapa del alcance de casi todos los libros dedicados a Swing!).

El portapapeles

Las JFC soportan algunas operaciones con el portapapeles del sistema (gracias al paquete **java.awt.datatransfer**). Se pueden copiar objetos **String** al portapapeles como si de texto se tratara, y se puede pegar texto de un portapapeles dentro de objetos **String**. Por supuesto, el portapapeles está diseñado para guardar cualquier tipo de datos, pero cómo represente el portapapeles la información que en él se deposite depende de cómo haga el programa las copias y los pegados. El API de portapapeles de Java proporciona extensibilidad mediante el concepto de versión. Todos los datos provenientes del portapapeles tienen asociadas varias versiones en los que se puede convertir (por ejemplo, un gráfico podría representarse como un string de números o como una imagen) y es posible consultar si ese portapapeles en particular soporta la versión en la que uno está interesado.

El programa siguiente es una mera demostración de cortar, copiar y pegar con datos **String** dentro de un **JTextArea**. Fíjese que las secuencias de teclado que suelen usarse para cortar, copiar y pegar funcionan igualmente. Incluso si se echa un vistazo a cualquier **JTextArea** o **JTextField** de cualquier otro programa, se descubre que también soportan las secuencias de teclas correspondientes al portapapeles. Este ejemplo simplemente añade control programático del portapapeles, ofreciendo una serie de técnicas que pueden usarse siempre que se desee capturar texto en cualquier cosa que no sea un **JTextComponent**.

```
//: c13:CortarYPegar.java
// Usando el portapapeles.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;
import com.bruceeckel.swing.*;

public class CortarYPegar extends JFrame  {
  JMenuBar mb = new JMenuBar();
  JMenu editar = new JMenu("Editar");
  JMenuItem
    cortar = new JMenuItem("Cortar"),
    copiar = new JMenuItem("Copiar"),
    pegar = new JMenuItem("Pegar");
  JTextArea texto = new JTextArea(20, 20);
  Clipboard clipbd =
    getToolkit().getSystemClipboard();
  public CortarYPegar()  {
    cortar.addActionListener(new CortarL());
    copiar.addActionListener(new Copiar());
    pegar.addActionListener(new PegarL());
    editar.add(cortar);
    editar.add(copiar);
    editar.add(pegar);
    mb.add(editar);
    setJMenuBar(mb);
    getContentPane().add(texto);
  }
  class CopiarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String seleccion = texto.getSelectedText();
      if (seleccion == null)
        return;
      StringSelection trozoCadena =
        new StringSelection(seleccion);
      clipbd.setContents(trozoCadena,trozoCadena);
```

```
    }
  }
  class CortarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      String seleccion = texto.getSelectedText();
      if (seleccion == null)
        return;
      StringSelection trozoCadena =
        new StringSelection(seleccion);
      clipbd.setContents(trozoCadena, trozoCadena);
      texto.replaceRange("",
        texto.getSelectionStart(),
        texto.getSelectionEnd());
    }
  }
  class PegarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      Transferable trozoDatos =
        clipbd.getContents(CortarYPegar.this);
      try {
        String trozoCadena =
          (String)trozoDatos.
            getTransferData(
              DataFlavor.stringFlavor);
        texto.replaceRange(trozoCadena,
          texto.getSelectionStart(),
          texto.getSelectionEnd());
      } catch(Exception ex) {
        System.err.println("No hay versión String");
      }
    }
  }
  public static void main(String[] args) {
    Console.run(new CortarYPegar(), 300, 200);
  }
} ///:~
```

La creación y adición del menú y del **JTextArea** deberían parecer ya una actividad de andar por casa. Lo diferente es la creación del campo **clipbd** de **Clipboard** que se lleva a cabo a través del **Toolkit**.

Toda la acción se da en los oyentes. Tanto el oyente **Copiar** como el **CortarL** son iguales excepto por la última línea de **CortarL**, que borra la línea que se está copiando. Las dos líneas especiales son la creación de un objeto **StringSelection** a partir del **String** y la llamada a **setContents( )** con esta **StringSelection**. El propósito del resto del código es poner un **String** en el portapapeles.

En **PegarL**, se extraen datos del portapapeles utilizando **getContents( )**. Lo que se obtiene es un objeto **Transferable**, bastante anónimo, y se desconoce lo que contiene. Una forma de averiguarlo es llamar a **getTransferDataFlavours( )**, que devuelve un array de objetos **DataFlavor** que indica las versiones soportadas por ese objeto en particular. También se le puede preguntar directamente con **isDataFlavorSupported( )** al que se le pasa la versión en la que uno está interesado. Aquí, sin embargo, se sigue el enfoque de invocar a **getTransferData( )** asumiendo que el contenido soporta la versión **String**, y si no lo hace se soluciona el problema en el gestor de excepciones.

Más adelante es posible que cada vez se de soporte a más versiones.

Empaquetando un applet en un archivo JAR

Un uso importante de la utilidad JAR es optimizar la carga de *applets*. En Java 1.0, la gente tendía a intentar concentrar todo su código en una única clase *applet* de forma que el cliente sólo necesitaría un acceso al servidor para descargar el código del *applet*. Esto no sólo conducía a programas complicados y difíciles de leer, sino que el archivo **.class** seguía sin estar comprimido, por lo que su descarga no era tan rápida como podría haberlo sido.

Los archivos JAR solucionan el problema comprimiendo todos los archivos **.class** en un único archivo que descarga el navegador. Ahora se puede crear el diseño correcto sin preocuparse de cuántos archivos **.class** conllevará, y el usuario obtendrá un tiempo de descarga mucho menor.

Considérese **TicTacToe.java**. Parece contener una única clase, pero, de hecho, contiene cinco clases internas, con lo que el total son seis. Una vez compilado el programa, se empaqueta en un archivo JAR con la línea:

```
jar cf TicTacToe.jar *.class
```

Ahora se puede crear una página HTML con la nueva etiqueta **archive** para indicar el nombre del archivo JAR. He aquí la etiqueta usando la forma vieja de la etiqueta HTML, a modo ilustrativo:

```
<head><tittle>TicTacToe Ejemplo Applet
</tittle></head>
<vody>
<applet code=TicTacToe.class
          archive=TicTacToe.jar
          width=200 height=100>
</applet>
</body>
```

Habrá que ponerlo en la forma nueva (más complicada y menos clara) vista anteriormente en este capítulo si queremos que funcione.

Técnicas de programación

Dado que la programación IGU en Java ha sido una tecnología en evolución con algunos cambios muy importantes entre Java 1.0/1.1 y la biblioteca Swing de Java 2, ha habido algunas estructuras y formas de obrar de programación antiguas que se han convertido precisamente en los ejemplos que vienen con Swing. Además, Swing permite programar de más y mejores formas que con los modelos viejos. En esta sección se demostrarán algunos de estos aspectos presentando y examinando algunos casos de programación.

Correspondencia dinámica de objetos

Uno de los beneficios del modelo de eventos de Swing es la flexibilidad. Se puede añadir o quitar comportamiento de eventos simplemente con llamadas a métodos. Esto se demuestra en el ejemplo siguiente:

```
//: c13:EventosDinamicos.java
// Se puede cambiar el comportamiento de los eventos dinámicamente.
// También muestra acciones múltiples para un evento.
// <applet code=EventosDinamicos
//  width=250 height=400></applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class EventosDinamicos extends JApplet {
  ArrayList v = new ArrayList();
  int i = 0;
  JButton
    b1 = new JButton("Boton1"),
    b2 = new JButton("Boton2");
  JTextArea txt = new JTextArea();
  class B implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      txt.append("se presiona un boton\n");
    }
  }
  class OyenteConteo implements ActionListener {
    int indice;
    public OyenteConteo(int i) { indice = i; }
    public void actionPerformed(ActionEvent e) {
      txt.append("Oyente contado "+indice+"\n");
    }
```

```
  }
  class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      txt.append("Boton 1 pulsado\n");
      ActionListener a = new OyenteConteo(i++);
      v.add(a);
      b2.addActionListener(a);
    }
  }
  class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      txt.append("Boton2 pulsado\n");
      int fin = v.size() - 1;
      if(fin >= 0) {
        b2.removeActionListener(
          (ActionListener)v.get(fin));
        v.remove(fin);
      }
    }
  }
  public void init() {
    Container cp = getContentPane();
    b1.addActionListener(new B());
    b1.addActionListener(new B1());
    b2.addActionListener(new B());
    b2.addActionListener(new B2());
    JPanel p = new JPanel();
    p.add(b1);
    p.add(b2);
    cp.add(BorderLayout.NORTH, p);
    cp.add(new JScrollPane(txt));
  }
  public static void main(String[] args) {
    Console.run(new EventosDinamicos(), 250, 400);
  }
} ///:~
```

El nuevo enfoque de este ejemplo radica en:

1. Hay más de un oyente para cada **Button**. Generalmente, los componentes manejan eventos de forma *multidifusión*, lo que quiere decir que se pueden registrar muchos oyentes para un único evento. En los componentes especiales en los que un evento se maneje de forma *unidifusion*, se obtiene una **TooManyListenersException**.

2. Durante la ejecución del programa, se añaden y eliminan dinámicamente los oyentes del **Button b2**. La adición se lleva a cabo de la forma vista anteriormente, pero cada componente también tiene un método **removeXXXListener( )** para eliminar cualquier tipo de oyente.

Este tipo de flexibilidad permite una potencia de programación muchísimo mayor.

El lector se habrá dado cuenta de que no se garantiza que se invoque a los oyentes en el orden en el que se añaden (aunque de hecho, muchas implementaciones sí que funcionan así).

Separar la lógica de negocio de la lógica IU

En general, se deseará diseñar clases de forma que cada una "sólo haga una cosa". Esto es especialmente importante cuando se ve involucrado el código de interfaz de usuario, puesto que es fácil vincular "lo que se está haciendo" con "cómo mostrarlo". Este tipo de emparejamiento evita la reutilización de código. Es mucho más deseable separar la "lógica de negocio" del IGU. De esta forma, no sólo se puede volver a usar más fácilmente la lógica de negocio, sino que también se facilita la reutilización del IGU.

Otro aspecto son los sistemas *multihilo* en los que los "objetos de negocio" suelen residir en una máquina completamente separada. Esta localización central de las reglas de negocio permite que los cambios sean efectivos al instante para toda nueva transacción, y es, por tanto, un método adecuado para actualizar un sistema. Sin embargo, se pueden usar estos objetos de negocio en muchas aplicaciones de forma que no estén vinculados a una única forma de presentación en pantalla. Es decir, su propósito debería restringirse a llevar a cabo operaciones de negocio y nada más.

El ejemplo siguiente muestra lo sencillo que resulta separar la lógica de negocio del código IGU:

```
//: c13:Separacion.java
// Separando lógica IGU de los objetos de negocio.
// <applet code=Separacion
// width=250 height=150> </applet>
import javax.swing.*;
import java.awt.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.applet.*;
import com.bruceeckel.swing.*;

class LogicaNegocio {
  private int modificador;
  public LogicaNegocio(int mod) {
    modificador = mod;
  }
  public void establecerModificador(int mod) {
    modificador = mod;
  }
```

```
  public int obtenerModificador() {
    return modificador;
  }
  // Algunas operaciones de negocio:
  public int calculo1(int arg) {
    return arg * modificador;
  }
  public int calculo2(int arg) {
    return arg + modificador;
  }
}

public class Separacion extends JApplet {
  JTextField
    t = new JTextField(15),
    mod = new JTextField(15);
  LogicaNegocio bl = new LogicaNegocio(2);
  JButton
    calc1 = new JButton("Calculo 1"),
    calc2 = new JButton("Calculo 2");
  static int obtenerValor(JTextField tf) {
    try {
      return Integer.parseInt(tf.getText());
    } catch(NumberFormatException e) {
      return 0;
    }
  }
  class Calc1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText(Integer.toString(
        bl.calculo1(obtenerValor(t))));
    }
  }
  class Calc2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      t.setText(Integer.toString(
        bl.calculo2(getValue(t))));
    }
  }
  // Si se desea que ocurra algo siempre que
  // cambia un JTextField, añade este oyente:
  class ModL implements DocumentListener {
    public void changedUpdate(DocumentEvent e) {}
    public void insertUpdate(DocumentEvent e) {
      bl.establecerModificador(getValue(mod));
```

```
    }
    public void removeUpdate(DocumentEvent e) {
      bl.establecerModificador(obtenerValor(mod));
    }
  }
  public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    calc1.addActionListener(new Calc1L());
    calc2.addActionListener(new Calc2L());
    JPanel p1 = new JPanel();
    p1.add(calc1);
    p1.add(calc2);
    cp.add(p1);
    mod.getDocument().
      addDocumentListener(new ModL());
    JPanel p2 = new JPanel();
    p2.add(new JLabel("Modificador:"));
    p2.add(mod);
    cp.add(p2);
  }
  public static void main(String[] args) {
    Console.run(new Separacion(), 250, 100);
  }
} ///:~
```

Se puede ver que **LogicaNegocio** es una clase directa que lleva a cabo sus operaciones sin ningún tipo de línea de código relacionada con un entorno IGU. Simplemente hace su trabajo.

Separación mantiene un seguimiento de todos los detalles de IU, y se comunica con **LogicaNegocio** sólo a través de su interfaz **public**. Todas las operaciones se centran en conseguir información de ida y vuelta a través del IU y el objeto **LogicaNegocio**. Así, a cambio **Separación** simplemente hace su trabajo. Dado que **Separación** sólo sabe que está hablando a un objeto **LogicaNegocio** (es decir, no está altamente acoplado), se podría hacer que se comunique con otros tipos de objetos sin grandes problemas.

Pensar en términos de separar IU de la lógica de negocio también facilita las cosas cuando se está adaptando código antiguo para que funcione con Java.

Una forma canónica

Las clases internas, el modelo de eventos de Swing, y el hecho de que el viejo modelo de eventos siga siendo soportado junto con las nuevas facetas de biblioteca basadas en estilos de programación antiguos, vienen a añadir elementos de confusión al proceso de diseño de código. Ahora hay incluso más medios para que la gente escriba código desagradable.

Excepto en circunstancias de extenuación, siempre se puede usar el enfoque más simple y claro: clases oyente (escritas generalmente como clases internas) para solucionar necesidades de manejo de eventos. Así se ha venido haciendo en la mayoría de ejemplos de este capítulo.

Siguiendo este modelo uno debería ser capaz de reducir las sentencias de su programa que digan: "Me pregunto qué causó este evento". Cada fragmento de código está involucrado con *hacer algo*, y no con la comprobación de tipos. Ésta es la mejor forma de escribir código; no sólo es fácil de conceptuar, sino que es aún más fácil de leer y mantener.

Programación visual y Beans

Hasta este momento, en este libro hemos visto lo valioso que es Java para crear fragmentos reusables de código. La unidad "más reusable" de código ha sido la clase, puesto que engloba una unidad cohesiva de características (campos) y comportamientos (métodos) que pueden reutilizarse directamente vía composición o mediante la herencia.

La herencia y el polimorfismo resultan partes esenciales de la programación orientada a objetos, pero en la mayoría de ocasiones, cuando se juntan en una aplicación, lo que se desea son componentes que hagan exactamente lo que uno necesita. Se desearía juntar estos fragmentos en un diseño exactamente igual que un ingeniero electrónico junta chips en una placa de circuitos. Parece también que debería existir alguna forma de acelerar este estilo de programación basado en el "ensamblaje modular".

La "programación visual" tuvo éxito por primera vez —tuvo *mucho* éxito— con el Visual Basic (VB) de Microsoft, seguido de un diseño de segunda generación en el Delphi de Borland (la inspiración primaria del diseño de los JavaBeans). Con estas herramientas de programación los componentes se representan visualmente, lo que tiene sentido puesto que suelen mostrar algún tipo de componente visual como botones o campos de texto. La representación visual, de hecho, suele ser la apariencia exacta del componente dentro del programa en ejecución. Por tanto, parte del proceso de programación visual implica arrastrar un componente desde una paleta para depositarlo en un formulario. La herramienta constructora de aplicaciones escribe el código correspondiente, y ese código será el encargado de crear todos los componentes en el programa en ejecución.

Para completar un programa, sin embargo, no basta con simplemente depositar componentes en un formulario. A menudo hay que cambiar las características de un componente, como su color, el texto que tiene, la base de datos a la que se conecta, etc. A las características que pueden modificarse en tiempo de ejecución se les denomina *propiedades*. Las propiedades de un componente pueden manipularse dentro de la herramienta constructora de aplicaciones, y al crear el programa se almacena esta información de configuración, de forma que pueda ser regenerada al comenzar el programa.

Hasta este momento, uno ya se ha acostumbrado a la idea de que un objeto es más que un conjunto de características; también es un conjunto de comportamientos. En tiempo de diseño, los comportamientos de un componente visual se representan mediante *eventos*, que indican que "aquí hay algo que le puede ocurrir al componente". Habitualmente, se decide qué es lo que se desea que ocurra cuando se da el evento, vinculando código a ese evento.

Aquí está la parte crítica: la herramienta constructora de aplicaciones utiliza la reflectividad para interrogar dinámicamente al componente y averiguar qué propiedades y eventos soporta. Una vez que sabe cuáles son, puede mostrar las propiedades y permitir cambiarlas (salvando el estado al construir el programa), y también mostrar los eventos. En general, se hace algo como doble clic en el evento y la herramienta constructora de aplicaciones crea un cuerpo de código que vincula a ese evento en particular. Todo lo que hay que hacer en ese momento es escribir el código a ejecutar cuando se dé el evento.

Todo esto conlleva a que mucho código lo haga la herramienta generadora de aplicaciones. Como resultado, uno puede centrarse en qué apariencia tiene el programa y en qué es lo que se supone que debe hacer, y confiar en la herramienta constructora de aplicaciones para que gestione los detalles de conexión. La razón del gran éxito de las herramientas constructoras de aplicaciones es que aceleran dramáticamente el proceso de escritura de una aplicación —sobretodo la interfaz de usuario, pero a menudo también otras porciones de la aplicación.

¿Qué es un Bean?

Como puede desprenderse, un Bean es simplemente un bloque de código generalmente embebido en una clase. El aspecto clave es la habilidad de la herramienta constructora de aplicaciones para descubrir las propiedades y eventos de ese componente. Para crear un componente VB, el programador tenía que escribir un fragmento de código bastante complicado siguiendo determinadas convenciones para exponer las propiedades y eventos. Delphi era una herramienta de programación visual de segunda generación y el lenguaje se diseñó activamente en torno a la programación visual, por lo que es mucho más fácil crear componentes visuales. Sin embargo, Java ha desarrollado la creación de componentes visuales hasta conducirla a su estado más avanzado con los JavaBeans, porque un Bean no es más que una clase. No hay que escribir ningún código extra o usar extensiones especiales del lenguaje para convertir algo en un Bean. Todo lo que hay que hacer, de hecho, es modificar ligeramente la forma de nombrar los métodos. Es el nombre del método el que dice a la herramienta constructora de aplicaciones si algo es una propiedad, un evento o simplemente un método ordinario.

En la documentación de Java, a esta convención de nombres se le denomina erróneamente "un patrón de diseño". Este nombre es desafortunado, puesto que los patrones de diseño (ver *Thinking in Patterns with Java*, descargable de *http://www.BruceEckel.com*) ya conllevan bastantes retos sin necesidad de este tipo de confusiones. No es un patrón de diseño, es simplemente una convención de nombres, y es bastante simple:

1. En el caso de una propiedad de nombre **xxx**, se suelen crear dos métodos: **getXxx( )** y **setXxx( )**. Nótese que la primera letra tras "get" y "set" se pone automáticamente en minúscula para generar el nombre de la propiedad. El tipo producido por el método "get" es el mismo que constituye el argumento para el método "set". El nombre de la propiedad y el tipo del "set" y "get" no tienen por qué guardar relación.

2. Para una propiedad **boolean**, se puede usar el enfoque "get" y "set" de arriba, pero también se puede usar "is" en vez de "get".

3. Los métodos generales del Bean no siguen la convención de nombres de arriba, siendo **public**.

4. En el caso de eventos, se usa el enfoque del "oyente" de Swing. Es exactamente lo que se ha estado viendo: **addFooBarListener(FooBarListener)** y **removeFooBarListerner(FooBarListener)** para manejar un **FooBarEvent**. La mayoría de las veces los eventos y oyentes ya definidos serán suficientes para satisfacer todas las necesidades, pero también se pueden crear interfaces oyentes y eventos propios.

El Punto 1 de los citados responde a la pregunta sobre algo que el lector podría haber visto al mirar al código viejo frente al código nuevo: muchos nombres de método han sufrido pequeños cambios de nombre, aparentemente no significativos. Ahora puede verse que la mayoría de esos cambios están relacionados con adaptar las convenciones de nombrado del "get" y "set" para convertir ese componente particular en un Bean.

Para crear un Bean simple pueden seguirse estas directrices:

```
//: beanrana:Rana.java
// Un JavaBean trivial.
package beanrana;
import java.awt.*;
import java.awt.event.*;

class Lugares {}

public class Rana {
  private int saltos;
  private Color color;
  private Lugares lugares;
  private boolean saltador;
  public int getSaltos() { return saltos; }
  public void setSaltos(int nuevosSaltos) {
    saltos = nuevosSaltos;
  }
  public Color getColor() { return color; }
  public void setColor(Color nuevoColor) {
    color = nuevoColor;
  }
  public Lugares getLugares() { return lugares; }
  public void setLugares(Lugares nuevosLugares) {
    lugares = nuevosLugares;
  }
  public boolean isSaltador() { return saltador; }
  public void setSaltador(boolean s) { saltador = s; }
  public void addActionListener(
      ActionListener l) {
```

```
    //...
  }
  public void removeActionListener(
      ActionListener l) {
    // ...
  }
  public void addKeyListener(KeyListener l) {
    // ...
  }
  public void removeKeyListener(KeyListener l) {
    // ...
  }
  // Un método público "ordinario":
  public void croar() {
    System.out.println("¡Croac!");
  }
} ///:~
```

En primer lugar, puede verse que es simplemente una clase. Generalmente todos los campos serán **private** y accesibles sólo a través de métodos. Siguiendo esta convención de nombres, las propiedades son **saltos**, **color**, **lugares** y **saltador** (nótese el cambio de mayúscula a minúscula en el caso de la primera letra). Aunque el nombre del identificador interno es el mismo que el nombre de la propiedad en los tres primeros casos, en **saltador** se puede ver que el nombre de la propiedad no obliga a usar ningún identificador particular para variables internas (ni de hecho, a *tener* ninguna variable interna para esa propiedad).

Los eventos que maneja este Bean son **ActionEvent** y **KeyEvent**, basados en el nombrado de los métodos "añadir" y "remover" del oyente asociado. Finalmente, podemos ver que el método ordinario **croar( )** sigue siendo parte del Bean simplemente por ser un método **public**, y no porque se someta a ningún esquema de nombres.

Extraer **BeanInfo** con el **Introspector**

Una de las partes más críticas del esquema de los Beans se dan al sacar un Bean de una paleta y colocarlo en un formulario. La herramienta constructora de aplicaciones debe ser capaz de crear el Bean (lo que puede hacer siempre que haya algún constructor por defecto) y después, sin acceder al código fuente del Bean, extraer toda la información necesaria para crear la hoja de propiedades y los manejadores de eventos.

Parte de la solución ya es evidente desde la parte final del Capítulo 12: la *reflectividad* de Java permite descubrir todos los métodos de una clase anónima. Esto es perfecto para solucionar el problema de los Beans sin que sea necesario usar ninguna palabra clave extra del lenguaje, como las que hay que usar en otros lenguajes de programación visual. De hecho, una de las razones primarias por las que se añadió reflectividad a Java era dar soporte a los Beans (aunque la reflectividad también soporta la serialización de objetos y la invocación remota de métodos). Por tanto, podría esperarse

que el creador de la herramienta constructora de aplicaciones hubiera aplicado reflectividad a cada Bean para recorrer sus métodos y localizar las propiedades y eventos del Bean.

Esto es verdaderamente posible, pero los diseñadores de Java querían proporcionar una herramienta estándar, no sólo para que los Beans fueran más fáciles de usar, sino también para proporcionar una pasarela estándar de cara a la creación de Beans más complejos. Esta herramienta es la clase **Introspector**, y su método más importante es el **static getBeanInfo( )**. A este método se le pasa una referencia a una **Class**, e interroga concienzudamente a la clase, devolviendo un objeto **BeanInfo** que puede ser después diseccionado para buscar en él propiedades, métodos y eventos.

Generalmente no hay que preocuparse por esto —probablemente se irán obteniendo los Beans ya diseñados por un tercero, y no habrá que conocer toda la magia subyacente en ellos. Simplemente se arrastrarán los Beans al formulario, se configurarán sus propiedades y se escribirán manipuladores para los eventos en los que se esté interesado. Sin embargo, es interesante y educativo ejercitar el uso del **Instrospector** para mostrar la información relativa a un Bean, así que he aquí una herramienta que lo hace:

```
//: c13:VolcadorBean.java
// Haciendo Introspección de un Bean.
// <applet code=VolcadorBean width=600 height=500>
// </applet>
import java.beans.*;
import java.lang.reflect.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import com.bruceeckel.swing.*;

public class VolcadorBean extends JApplet {
  JTextField consulta =
    new JTextField(20);
  JTextArea resultados = new JTextArea();
  public void prt(String s) {
    resultados.append(s + "\n");
  }
  public void volcar(Class bean){
    resultados.setText("");
    BeanInfo bi = null;
    try {
      bi = Introspector.getBeanInfo(
        bean, java.lang.Object.class);
    } catch(IntrospectionException e) {
      prt("No se pudo hacer instrospeccion a " +
        bean.getName());
      return;
    }
    PropertyDescriptor[] propiedades =
```

```
    bi.getPropertyDescriptors();
for(int i = 0; i < propiedades.length; i++) {
    Class p = propiedades[i].getPropertyType();
    prt("Tipo de la propiedad:\n   " + p.getName() +
      "Nombre de la propiedad:\n   " +
      propiedades[i].getName());
    Method leerMetodo =
      propiedades[i].getReadMethod();
    if(readMethod != null)
      prt("Leer metodo:\n   " + leerMetodo);
    Method escribirMetodo =
      propiedades[i].getWriteMethod();
    if(escribirMetodo != null)
      prt("Escribir metodo:\n   " + escribirMetodo);
    prt("====================");
}
prt("Metodos public:");
MethodDescriptor[] metodos =
    bi.getMethodDescriptors();
for(int i = 0; i < metodos.length; i++)
    prt(metodos[i].getMethod().toString());
prt("======================");
prt("Soporte a eventos:");
EventSetDescriptor[] eventos =
    bi.getEventSetDescriptors();
for(int i = 0; i < eventos.length; i++) {
    prt("Tipo de Oyente:\n   " +
      eventos[i].getListenerType().getName());
    Method[] lm =
      eventos[i].getListenerMethods();
    for(int j = 0; j < lm.length; j++)
      prt("Metodo Oyente:\n   " +
        lm[j].getName());
    MethodDescriptor[] lmd =
      eventos[i].getListenerMethodDescriptors();
    for(int j = 0; j < lmd.length; j++)
      prt("Descriptor de metodo:\n   " +
        lmd[j].getMethod());
    Method aniadirOyente =
      eventos[i].getAddListenerMethod();
    prt("Aniadir metodo Oyente:\n   " +
        aniadirOyente);
    Method quitarOyente =
      eventos[i].getRemoveListenerMethod();
    prt("Eliminar metodo Oyente:\n   " +
```

```
            quitarOyente);
          prt("====================");
        }
      }
      class Volcador
    implements ActionListener {
        public void actionPerformed(ActionEvent e) {
          String nombre = consulta.getText();
          Class c = null;
          try {
            c = Class.forName(nombre);
          } catch(ClassNotFoundException ex) {
            resultados.setText("No se pudo encontrar " + nombre);
            return;
          }
          volcar(c);
        }
      }
      public void init() {
        Container cp = getContentPane();
        JPanel p = new JPanel();
        p.setLayout(new FlowLayout());
        p.add(new JLabel("Nombre de bean cualificado:"));
        p.add(consulta);
        cp.add(BorderLayout.NORTH, p);
        cp.add(new JScrollPane(resultados));
        Volcador vlcdr = new Volcador();
        consulta.addActionListener(vlcdr);
        consulta.setText("beanRana.Rana");
        // Forzar evaluación
        vlcdr.actionPerformed(
          new ActionEvent(vlcdr, 0, ""));
      }
      public static void main(String[] args) {
        Console.run(new VolcadorBean(), 600, 500);
      }
} ///:~
```

VolcadorBean.volcar( ) es el método que hace todo el trabajo. Primero intenta crear un objeto **BeanInfo**, y si tiene éxito llama a los métodos de **BeanInfo** que producen información sobre las propiedades, métodos y eventos. En **Instrospector.getBeanInfo( )**, se verá que hay un segundo parámetro. Éste dice a **Introspector** dónde parar en la jerarquía de herencia. Aquí se para antes de analizar todos los métodos de **Object**, puesto que no nos interesan.

En el caso de las propiedades, **getPropertyDescriptors( )** devuelve un array de **PropertyDescriptors**. Por cada **PropertyDescriptor** se puede invocar a **getPropertyType( )** para averiguar la clase

del objeto pasado hacia dentro y hacia fuera vía los métodos de propiedad. Después, por cada propiedad se puede acceder a sus pseudónimos (extraídos de los métodos de nombre) con **getName( )**, el método para leer con **getReadMethod( )**, y el método para escribir con **getWriteMethod( )**. Estos dos últimos métodos devuelven un objeto **Method**, que puede, de hecho, ser usado para invocar al método correspondiente en el objeto (esto es parte de la reflectividad).

En el caso de los métodos **public** (incluyendo los métodos de propiedad), **getMethodDescriptors( )** devuelve un array de **MethodDescriptor**s. Por cada uno se puede obtener el objeto **Method** asociado e imprimir su nombre.

En el caso de los eventos, **getEventSetDescriptors( )** devuelve un array de (¿qué otra cosa podría ser?) **EventSetDescriptor**s. Cada uno de éstos puede ser interrogado para averiguar la clase del oyente, los métodos de esa clase oyentes, y los métodos de adición y eliminación de oyentes. El programa **VolcadorBean** imprime toda esta información.

Al empezar, el programa fuerza la evaluación de **beanrana.Rana**. La salida, una vez eliminados los detalles extra innecesarios por ahora, es:

```
class name: Rana
Tipo de la propiedad:
     Color
Nombre de la propiedad:
     color
Leer metodo:
     public Color getColor ()
Escribir metodo:
     public void setColor (Color)
====================
Tipo de la propiedad:
     lugares
Nombre de la propiedad:
     lugares
Leer metodo
     public Lugares getLugares ()
Escribir metodo:
     public void setLugares (Lugares)
====================
Tipo de la propiedad:
     boolean
Nombre de la propiedad:
     saltador
Leer metodo:
     public boolean isSaltador ()
Escribir metodo:
     public void setSaltador (boolean)
====================
```

```
Tipo de la propiedad:
     int
Nombre de la propiedad:
     saltos
Leer metodo:
     public int getSaltos ()
Escribir metodo:
     public void setSaltos (int)
====================
Metodos public:
public void setSaltos (int)
public void croar ()
public void removeActionListener (ActionListener)
public void addActionListener (ActionListener)
public int getSaltos ()
public void setColor (Color)
public void setLugares (Lugares)
public void setSaltador (boolean)
public boolean isSaltador ()
public void addKeyListener (KeyListener)
public Color getColor ()
public void removeKeyListener (KeyListener)
public Lugares getLugares ()
====================
Soporte a Eventos:
Tipo de Oyente
     KeyOyente
Metodo Oyente:
     keyTyped
Metodo Oyente:
     keyPressed
Metodo Oyente:
     keyReleased
Descriptor de metodo:
     public void keyTyped (KeyEvent)
Descriptor de metodo:
     public void keyPressed (KeyEvent)
Descriptor de metodo:
     public void keyReleased (KeyEvent)
Añadir metodo Oyente:
     public void addKeyListener (KeyListener)
Eliminar metodo Oyente:
     public  void removeKeyListener (KeyListener)
====================
Tipo de Oyente:
```

```
     ActionListener
Tipo de Oyente:
     ActionPerformed
Descriptor de metodo:
     public void actionPerformed (ActionEvent)
Añadir metodo Oyente:
     public void addActionListener (ActionListener)
Eliminar metodo Oyente:
     public void removeActionListener (ActionListener)
====================
```

Esto revela la mayoría de lo que **Introspector** ve a medida que produce un objeto **BeanInfo** a partir del Bean. Se puede ver que el tipo de propiedad es independiente de su nombre. Nótese que los nombres de propiedad van en minúsculas. (La única ocasión en que esto no ocurre es cuando el nombre de propiedad empieza con más de una letra mayúscula en una fila.) Y recuerde que los nombres de método que se ven en este caso (como los métodos de lectura y escritura) son de hecho producidos a partir de un objeto **Method** que puede usarse para invocar al método asociado del objeto.

La lista de métodos **public** incluye la lista de métodos no asociados a una propiedad o evento, como **croar( )**, además de los que sí lo están. Éstos son todos los métodos a los que programando se puede invocar en un Bean, y la herramienta constructora de aplicaciones puede elegir listarlos todos al hacer llamadas a métodos, para facilitarte la tarea.

Finalmente, se puede ver que se analizan completamente los eventos en el oyente, sus métodos y los métodos de adición y eliminación de oyentes. Básicamente, una vez que se dispone del **BeanInfo**, se puede averiguar todo lo que sea relevante para el Bean. También se puede invocar a los métodos para ese Bean, incluso aunque no se tenga otra información excepto el objeto (otra faceta, de nuevo, de la reflectividad).

Un Bean más sofisticado

El siguiente ejemplo es ligeramente más sofisticado, a la vez que frívolo. Es un **JPanel** que dibuja un pequeño círculo en torno al ratón cada vez que se mueve el ratón. Cuando se presiona el ratón, aparece la palabra "¡Bang!" en medio de la pantalla, y se dispara un oyente de acción.

Las propiedades que pueden cambiarse son el tamaño del círculo, además del color, tamaño, y texto de la palabra que se muestra al presionar el ratón. Un **BeanExplosion** también tiene su propio **addActionListener( )** y **removeActionListener( )**, de forma que uno puede adjuntar su propio oyente a disparar cuando el usuario haga clic en el **BeanExplosion**. Habría que ser capaz de reconocer la propiedad y el soporte al evento:

```
//: beanexplosion:BeanExplosion.java
// Un Bean gráfico.
package beanexplosion;
import javax.swing.*;
```

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class BeanExplosion extends JPanel
      implements Serializable {
  protected int xm, ym;
  protected int tamanioC = 20; // Tamaño del circulo
  protected String texto = "¡Bang!";
  protected int tamanioFuente = 48;
  protected Color colorT = Color.red;
  protected ActionListener oyenteAccion;
  public BeanExplosion() {
    addMouseListener(new ML());
    addMouseMotionListener(new MML());
  }
  public int getTamanioCirculo() { return tamanioC; }
  public void setTamanioCirculo(int nuevoTamanio) {
    tamanioC = nuevotamanio;
  }
  public String getTextExplosion() { return texto; }
  public void setTextExplosion(String nuevoTexto) {
    texto = nuevoTexto;
  }
  public int getTamanioFuente() { return tamanioFuente; }
  public void setTamanioFuente(int nuevoTamanio) {
    tamanioFuente = nuevoTamanio;
  }
  public Color getColorTexto() { return colorT; }
  public void setColorTexto(Color nuevoColor) {
    colorT = nuevoColor;
  }
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.black);
    g.drawOval(xm - tamanioC/2, ym - tamanioC/2,
      tamanioC, tamanioC);
  }
  // Éste es un oyente unidifusión, que es la forma mas simple
  // de gestión de oyente:
  public void addActionListener (
      ActionListener l)
        throws TooManyListenersException {
```

```
    if(oyenteAccion != null)
      throw new TooManyListenersException();
    oyenteAccion = l;
  }
  public void removeActionListener(
      ActionListener l) {
    oyenteAccion = null;
  }
  class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
      Graphics g = getGraphics();
      g.setColor(colorT);
      g.setFont(
        new Font(
          "TimesRoman", Font.BOLD, tamanioFuente));
      int ancho =
        g.getFontMetrics().stringWidth(texto);
      g.drawString(texto,
        (getSize().width - ancho) /2,
        getSize().alto/2);
      g.dispose();
      // Llamar al método del listener:
      if(oyenteAccion != null)
        oyenteAccion.actionPerformed(
          new ActionEvent(BeanExplosion.this,
            ActionEvent.ACTION_PERFORMED, null));
    }
  }
  class MML extends MouseMotionAdapter {
    public void mouseMoved(MouseEvent e) {
      xm = e.getX();
      ym = e.getY();
      repaint();
    }
  }
  public Dimension getPreferredSize() {
    return new Dimension(200, 200);
  }
} ///:~
```

Lo primero que se verá es que **BeanExplosion** implementa la interfaz **Serializable**. Esto significa que la herramienta constructora de aplicaciones puede "acceder" a toda la información del **BeanExplosion** usando la serialización una vez que el diseñador del programa haya ajustado los valores de las propiedades. Cuando se crea el Bean como parte de la aplicación en ejecución se restauran estas propiedades y se realmacenan, obteniendo así exactamente lo que se diseñó.

Se puede ver que todos los campos son **private**, que es lo que generalmente se hará con un Bean —permitir el acceso sólo a través de métodos, generalmente usando el esquema "de propiedades".

Cuando se echa un vistazo a la signatura de **addActionListener( )**, se ve que puede lanzar una **TooManyListenersException**. Esto indica que es *unidifusión*, lo que significa que sólo notifica el evento a un oyente. Generalmente, se usarán eventos *multidifusión* de forma que puedan notificarse a varios oyentes. Sin embargo, eso conlleva aspectos para los que uno no estará preparado hasta acceder al siguiente capítulo, por lo que se volverá a retomar este tema en él (bajo el título de "Revisar los JavaBeans"). Un evento unidifusión "circunvala" este problema.

Cuando se hace clic con el ratón, se pone el texto en el medio del **BeanExplosion** y si el campo **actionListener** no es **null**, se invoca su **actionPerformed( )**, creando un nuevo objeto **ActionEvent** en el proceso. Cada vez que se mueva el ratón, se capturan sus nuevas coordenadas y se vuelve a dibujar el lienzo (borrando cualquier texto que esté en él, como se verá).

He aquí la clase **PruebaBeanExplosion** que permite probar el Bean como otro *applet* o aplicación:

```
//: c13:PruebaBeanExplosion.java
// <applet code=PruebaBeanExplosion
// width=400 height=500></applet>
import bangbean.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class PruebaBeanExplosion extends JApplet {
  JTextField txt = new JTextField(20);
  // Durante la prueba, informar de las acciones:
  class BBL implements ActionListener {
    int conteo = 0;
    public void actionPerformed(ActionEvent e){
      txt.setText("Accion BeanExplosion "+ conteo++);
    }
  }
  public void init() {
    BeanExplosion bb = new BeanExplosion();
    try {
      bb.addActionListener(new BBL());
    } catch(TooManyListenersException e) {
      txt.setText("Demasiados oyentes");
    }
    Container cp = getContentPane();
    cp.add(bb);
    cp.add(BorderLayout.SOUTH, txt);
```

```
  }
  public static void main(String[] args) {
    Console.run(new PruebaBeanExplosion(), 400, 500);
  }
} ///:~
```

Cuando un Bean está en un entorno de desarrollo, no se usará su clase, pero es útil proporcionar un método de prueba rápida para cada uno de los Beans. **PruebaBeanExplosion** coloca un **BeanExplosion** dentro del *applet*, adjuntando un **ActionListener** simple al **BeanExplosion** para imprimir una cuenta de eventos al **JTextField** cada vez que se da un **ActionEvent**. Generalmente, por supuesto, la herramienta constructora de aplicaciones crearía la mayoría de código que usa el Bean.

Cuando se ejecuta el **BeanExplosion** a través de **VolcadoBean** o se pone **BeanExplosion** dentro de un entorno de desarrollo que soporta Beans, se verá que hay muchas más propiedades y acciones de lo que parece en el código de arriba. Esto es porque **BeanExplosion** se hereda de **JPanel**, y **JPanel** es también un Bean, por lo que se están viendo también sus propiedades y eventos.

Empaquetar un Bean

Antes de poder incorporar un Bean en una herramienta de construcción visual habilitada para Beans, hay que ponerlo en un contenedor estándar de Beans, que es un archivo JAR que incluye todas las clases Bean además de un archivo de *manifiesto* que dice: "Esto es un Bean". Un archivo *manifiesto* no es más que un archivo de texto que sigue un formato particular. En el caso del **BeanExplosion**, el archivo *manifiesto* tiene la apariencia siguiente (sin la primera y última líneas):

```
//:! :BeanExplosion.mb
Manifest-Version: 1.0

Name: BeanExplosion/BeanExplosion.class
Java-Bean: True
///:~
```

La primera línea indica la versión del esquema de *manifiesto*, que mientras Sun no indique lo contrario es la 1.0. La segunda línea (las líneas en blanco se ignoran) nombra el archivo **BeanExplosion.class**, y la tercera dice, "Es un Bean". Sin la tercera línea, la herramienta constructora de programas no reconocería la clase como un Bean.

El único punto del que hay que asegurarse es de que se pone el nombre correcto en el campo "Name". Si se vuelve atrás a **BeanExplosion.java**, se verá que está en el **package beanexplosion** (y por consiguiente en un subdirectorio denominado "beanexplosion" que está fuera del *classpath*), y el nombre del archivo de *manifiesto* debe incluir esta información de paquete. Además, hay que colocar el archivo de *manifiesto* en el directorio *superior* a la raíz de la trayectoria de los paquetes, lo que en este caso significa ubicar el archivo en el directorio padre al subdirectorio "beanexplosion". Posteriormente hay que invocar a **jar** desde el directorio que contiene al archivo de *manifiesto*, así:

```
jar cfm BeanExplosion.jar BeanExplosion.mf beanExplosion
```

Esta línea asume que se desea que el archivo JAR resultante se denomine **BeanExplosion.jar**, y que se ha puesto el *manifiesto* en un archivo denominado **BeanExplosion.mf**.

Uno podría preguntarse "¿Qué ocurre con todas las demás clases que se generaron al compilar **BeanExplosion.java**?" Bien, todas acabaron dentro del subdirectorio **beanexplosion**, y se verá que el último parámetro para la línea de comandos de **jar** de arriba es el subdirectorio **beanexplosión**. Cuando se proporciona a **jar** el nombre de un subdirectorio, empaqueta ese subdirectorio al completo en el archivo jar (incluyendo, en este caso, el archivo de código fuente **BeanExplosion.java** original —uno podría elegir no incluir el código fuente con sus Beans). Además, si se deshace esta operación y se desempaqueta el archivo JAR creado, se descubrirá que el archivo de *manifiesto* no está dentro, pero que **jar** ha creado su propio archivo de *manifiesto* (basado parcialmente en el nuestro) de nombre **MANIFEST.MF** y lo ha ubicado en el subdirectorio **META-INF** (es decir, "metainformación"). Si se abre este archivo de *manifiesto* también se verá que **jar** ha añadido información de firma digital por cada archivo, de la forma:

```
Digest-Algorithm: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/O0=
MD5-Digest: O4NcS1hE3Smnzlp2hj6qeg==
```

En general, no hay que preocuparse por nada de esto, y si se hacen cambios se puede modificar simplemente el archivo de *manifiesto* original y volver a invocar a **jar** para que cree un archivo JAR nuevo para el Bean. También se pueden añadir otros Beans al archivo JAR simplemente añadiendo su información al de *manifiesto*.

Una cosa en la que hay que fijarse es que probablemente se querrá poner cada Bean en su propio subdirectorio, puesto que cuando se crea un archivo JAR se pasa a la utilidad **jar** el nombre de un subdirectorio y éste introduce todo lo que hay en ese subdirectorio en el archivo JAR. Se puede ver que tanto **Rana** como **BeanExplosion** están en sus propios subdirectorios.

Una vez que se tiene el Bean adecuadamente insertado en el archivo JAR, se puede incorporar a un entorno constructor de programas habilitado para Beans. La manera de hacer eso cambia de una herramienta a otra, pero Sun proporciona un banco de pruebas para JavaBeans disponible gratuitamente en su *Beans Development Kit* (BDK) denominado el "beanbox". (El BDK puede descargarse de *http://java.sun.com/beans*.) Para ubicar un Bean en la beanbox, se copia el archivo JAR en el directorio "jars" del BDK antes de iniciar el beanbox.

Soporte a Beans más complejo

Se puede ver lo remarcadamente simple que resulta construir un Bean. Pero uno no está limitado a lo que ha visto aquí. La arquitectura de JavaBeans proporciona un punto de entrada simple, pero también se puede escalar a situaciones más complejas. Estas situaciones van más allá del alcance de este libro, pero se presentarán de forma breve en este momento. Hay más detalles en *http://java.sun.com/beans*.

Un lugar en el que se puede añadir sofisticación es con las propiedades. Los ejemplos de arriba sólo mostraban propiedades simples, pero también es posible representar múltiples propiedades en un array. A esto se le llama una *propiedad indexada*. Simplemente se proporcionan los métodos ade-

cuados (siguiendo de nuevo una convención para los nombres de los métodos) y el **Introspector** reconoce la propiedad indexada de forma que la herramienta constructora de aplicaciones pueda responder de forma apropiada.

Las propiedades pueden *vincularse*, lo que significa que se notificarán a otros objetos vía un **PropertyChangeEvent**. Los otros objetos pueden después elegir cambiarse a sí mismos basándose en el cambio sufrido por el Bean.

Las propiedades pueden *limitarse*, lo que significa que los otros objetos pueden vetar cambios en esa propiedad si no los aceptan. A los otros objetos se les avisa usando un **PropertyChangeEvent**, y puede lanzar una **PropertyVetoException** para evitar que ocurra el cambio y restaurar los valores viejos.

También se puede cambiar la forma de representar el Bean en tiempo de diseño:

1. Se puede proporcionar una hoja general de propiedades del Bean en particular. Esta hoja de propiedades se usará para todos los otros Beans, invocándose el nuestro automáticamente al seleccionar el Bean.

2. Se puede crear un editor personalizado para una propiedad particular, de forma que se use la hoja de propiedades ordinaria, pero cuando se edite la propiedad especial, se invocará automáticamente a este editor.

3. Se puede proporcionar una clase **BeanInfo** personalizada para el Bean, que produzca información diferente de la información por defecto que crea en **Introspector**.

4. También es posible pasar a modo "experto" o no por cada **FeatureDescriptor**s para distinguir entre facetas básicas y aquéllas más complicadas.

Más sobre Beans

Hay otro aspecto que no se pudo describir aquí. Siempre que se cree un Bean, cabría esperar que se ejecute en un entorno multihilo. Esto significa que hay que entender los conceptos relacionados con los hilos, y que se presentarán en el Capítulo 14. En éste se verá una sección denominada "Volver a visitar los Beans" que describirá este problema y su solución.

Hay muchos libros sobre JavaBeans; por ejemplo, *JavaBeans* de Elliotte Rusty Harold (IDG, 1998).

Resumen

De todas las bibliotecas de Java, la biblioteca IGU ha visto los cambios más dramáticos de Java 1.0 a Java 2. La AWT de Java 1.0 se criticó como uno de los peores diseños jamás vistos, y aunque permitía crear programas portables, el IGU resultante era "igualmente mediocre en todas las plataformas". También imponía limitaciones y era extraño y desagradable de usar en comparación con las herramientas de desarrollo de aplicaciones nativas disponibles en cada plataforma particular.

Cuando Java 1.1 presentó el nuevo modelo de eventos y los JavaBeans, se dispuso el escenario —de forma que era posible crear componentes IGU que podían ser arrastrados y depositados fácilmente dentro de herramientas de construcción de aplicaciones visuales. Además, el diseño del modelo de eventos y de los Beans muestra claramente un alto grado de consideración por la facilidad de programación y la mantenibilidad del código (algo que no era evidente en la AWT de Java 1.0). Pero hasta que no aparecieron las clases JFC/Swing, el trabajo no se dio por concluido. Con los componentes Swing, la programación de IGU multiplataforma se convirtió en una experiencia civilizada.

De hecho, lo único que se echa de menos es la herramienta constructora de aplicaciones, que es donde radica la verdadera revolución. El Visual Basic y Visual C++ de Microsoft requieren de herramientas de construcción de aplicaciones propias de Microsoft, al igual que ocurre con Borland en el caso de Delphi y C++. Si se desea que la herramienta de construcción de aplicaciones sea mejor, hay que cruzar los dedos y esperar a que el productor haga lo que uno espera. Pero Java es un entorno abierto, y por tanto, no sólo permite entornos de construcción de aplicaciones de otros fabricantes, sino que trata de hacer énfasis en que se construyan. Y para que estas herramientas sean tomadas en serio, deben soportar JavaBeans. Esto significa un campo de juego por niveles: si sale una herramienta constructora de aplicaciones mejor, uno no está obligado a usar la que venía usando —puede decidir cambiarse a la nueva e incrementar su productividad. Este tipo de entorno competitivo para herramientas constructoras de aplicaciones IGU no se había visto nunca anteriormente, y el mercado resultante no puede sino ser beneficioso para la productividad del programador.

Este capítulo solamente pretendía dar una introducción a la potencia de Swing al lector de forma que pudiera ver lo relativamente simple que es introducirse en las bibliotecas. Lo visto hasta la fecha será probablemente suficiente para una porción significativa de necesidades de diseño de IU. Sin embargo, Swing tiene muchas más cosas —se pretende que sea un conjunto de herramientas de extremada potencia para el diseño de IU. Probablemente proporciona alguna forma de lograr absolutamente todo lo que se nos pudiera ocurrir.

Si uno no ve aquí todo lo que necesita, siempre puede optar por la documentación en línea de Sun y buscar en la Web, y si sigue siendo poco, buscar un libro dedicado a Swing —un buen punto de partida es *The JFC Swing Tutorial*, de Walrath & Campione (Addison Welsey, 1999).

Ejercicios

Las soluciones a determinados ejercicios se encuentran en el documento *The Thinking in Java Annotated Solution Guide*, disponible a bajo coste en *http://www.BruceEckel.com*.

1. Crear un *applet*/aplicación usando la clase **Console** mostrada en este capítulo. Incluir un campo de texto y tres botones. Al presionar cada botón, hacer que aparezca algún texto distinto en el campo de texto.

2. Añadir una casilla de verificación al *applet* creado en el Ejercicio 1, capturar el evento e insertar un texto distinto en el campo de texto.

3. Crear un *applet*/aplicación utilizando **Console**. En la documentación HTML de *java.sun.com*, encontrar el **JPasswordField** y añadirlo al programa. Si el usuario teclea la contraseña correcta, usar **JOptionPane** para proporcionar un mensaje de éxito al usuario.

4. Crear un *applet*/aplicación usando **Console**, y añadir todos los componentes que tengan un método **addActionListener( )**. (Consultarlos todos en la documentación HTML de *http://java.sun.com*. Truco: usar el índice.) Capturar sus eventos y mostrar un mensaje apropiado para cada uno dentro de un campo de texto.

5. Crear un *applet*/aplicación usando **Console**, con un **JButton**, y un **JTextField**. Escribir y adjuntar el oyente apropiado de forma que si el foco reside en el botón, los caracteres del mismo aparezcan en el **JTextField**.

6. Crear un *applet*/aplicación usando **Console**. Añadir al marco principal todos los componentes descritos en este capítulo, incluyendo menús y una caja de diálogo.

7. Modificar **CamposTexto.java** de forma que los caracteres de **t2** retengan el uso de mayúsculas y minúsculas original, en vez de forzarlas automáticamente a mayúsculas.

8. Localizar y descargar uno o más entornos de desarrollo constructores de IGU gratuitos disponibles en Internet, o comprar un producto comercial. Descubrir qué hay que hacer para añadir **BeanExplosion** al entorno, y hacerlo.

9. Añadir **Rana.class** al archivo de *manifiesto* como se muestra en este capítulo y ejecutar **jar** para crear un archivo JAR que contenga tanto a **Rana** como a **BeanExplosion**. Ahora descargar e instalar el BDK de Sun, o bien usar otra herramienta constructora de programas habilitada para Beans, y añadir el archivo JAR al entorno, de forma que se puedan probar estos dos Beans.

10. Crear su propio JavaBean denominado **Valvula**, que contenga dos propiedades: una **boolean** denominada "on" y otra **int** denominada "nivel". Crear un archivo de *manifiesto*, usar **jar** para empaquetar el Bean, y después cargarlo en la beanbox o en una herramienta constructora de programas habilitada para Beans, y así poder probarlo.

11. Modificar **CajasMensajes.java** de forma que tenga un **ActionListener** individual por cada botón (en vez de casar el texto del botón).

12. Monitorizar un nuevo tipo de evento en **RastrearEvento.java**, añadiendo el nuevo código de manejo del evento. Primero habrá que decidir cuál será el nuevo tipo de evento a monitorizar.

13. Heredar un nuevo tipo de botón de **JButton**. Cada vez que se presione este botón, debería cambiar de color a un valor seleccionado al azar. Ver **CajasColores.java** del Capítulo 14 para tener un ejemplo de cómo generar un valor de código al azar.

14. Modificar **PanelTexto.java** para que use un **JTextArea** en vez de un **JTextPane**.

15. Modificar **Menus.java** de forma que use botones de opción en vez de casillas de verificación en los menús.

16. Simplificar **Lista.java** pasándole el array al constructor y eliminando la adición dinámica de elementos a la lisa.

17. Modificar **OndaSeno.java** para que **DibujarSeno** sea un JavaBean añadiendo métodos "getter" y "setter".

18. ¿Recuerdas el juguete "Telesketch", con dos controles, uno encargado del movimiento vertical del punto del dibujo, y otro que controla el movimiento horizontal? Crear uno de ésos usando **OndaSeno.java** como comienzo. En vez de controles rotatorios, usar barras de desplazamiento. Añadir un botón que borre toda la pantalla.

19. Crear un "indicador de progresos asintótico" que vaya cada vez más despacio a medida que se acerca a su meta. Añadir comportamiento errático al azar de forma que periódicamente parezca que comienza a acelerar.

20. Modificar **Progreso.java** de forma que no comparta modelos sino que use un oyente para conectar el deslizador y la barra de progreso.

21. Seguir las instrucciones de la sección "Empaquetando un *applet* en un archivo JAR" para ubicar **TicTacToe.java** en un archivo JAR. Crear una página HTML con la versión (complicada y difícil) de la etiqueta *applet*, y modificarla para que use la etiqueta de archivo para usar el archivo JAR. (Truco: empezar con la página HTML de **TicTacToe.java** que viene con la distribución de código fuente de este libro.)

22. Crear un *applet*/aplicación usando **Console**. Este debería tener tres deslizadores, para los valores del rojo, verde y azul de **java.awt.Color**. El resto del formulario debería ser un **JPanel** que muestre el color determinado por los tres deslizadores. Incluir también campos de texto no editables que muestren los valores RGB actuales.

23. En la documentación HTML de **javax.swing**, buscar **JColorChooser**. Escribir un programa con un botón que presente este selector de color como un diálogo.

24. Casi todo componente Swing se deriva de **Component**, que tiene un método **setCursor( )**. Buscarlo en la documentación HTML. Crear un *applet* y cambiar el cursor a uno de los almacenados en la clase **Cursor**.

25. A partir de **MostrarAddListeners.java**, crear un programa con la funcionalidad completa de **LimpiarMostrarMetodos.java** del Capítulo 12.