APÉNDICE

D Interfaces gráficas de usuario[1]

Una *interfaz gráfica de usuario* (GUI[2]) es la alternativa moderna a la E/S por el terminal que permite a un programa comunicarse con su usuario. En una GUI se crea una aplicación de ventanas. Tenemos diversas alternativas para realizar la entrada: seleccionando una opción entre una lista de ellas, pulsando botones, completando formularios y empleando el ratón. La salida puede efectuarse mediante la escritura de texto o la realización de gráficos. En Java, la programación de GUI se realiza a través del *Abstract Window Toolkit* (AWT), que es un paquete estándar incluido en todos los sistemas Java. Un elemento relacionado es el applet de Java, que es un programa que puede descargarse de Internet y ejecutarse en un terminal. Los applets emplean, invariablemente, el AWT.

En este apéndice veremos:

- Los elementos básicos de las GUI en el AWT.
- Cómo estos elementos comunican información.
- Cómo pueden integrarse dichos elementos en una ventana.
- Cómo se dibujan gráficos.
- Cómo se diseñan applets en Java.

Una *interfaz gráfica de usuario (GUI)* es la alternativa moderna a la E/S por el terminal, que permite a un programa comunicarse con su usuario.

D.1 El Abstract Window Toolkit

El *Abstract Window Toolkit* (AWT) es un conjunto de utilidades GUI incluido en todos los sistemas Java. Está compuesto por las clases básicas que permiten diseñar interfaces de usuario. Dichas clases se encuentran en el paquete `java.awt`. El AWT es portable y funciona en multitud de plataformas. Su uso es sencillo si se pretende diseñar interfaces simples. Gracias a él, las GUI pueden implementarse sin recurrir a ayudas de desarrollo visual, y supone una mejora importante sobre las interfaces de terminal básicas.

El *Abstract Window Toolkit (AWT)* es un conjunto de utilidades GUI incluido en todos los sistemas Java.

Normalmente, en un programa en el que se realiza E/S por el terminal, se pregunta al usuario acerca de la entrada y después se ejecuta una instrucción que lee una línea del terminal. Cuando dicha línea es leída, se procesa. En este contexto el flujo de control puede seguirse fácilmente. Sin embargo, la programación de GUI es diferente. En ella, las componentes de la entrada se organizan en una ventana. Después de mostrar la ventana, el programa espera un evento, como que se pulse un botón, y en ese momento se invoca un manejador de eventos. Esto implica que

La programación de GUI es programación dirigida por eventos.

[1] *N. del T.:* En este apéndice los programas se muestran en su versión original para asegurar la coherencia con el resto de figuras.
[2] *N. del T.:* La abreviatura procede de su nombre en inglés: *Graphical User Interface*.

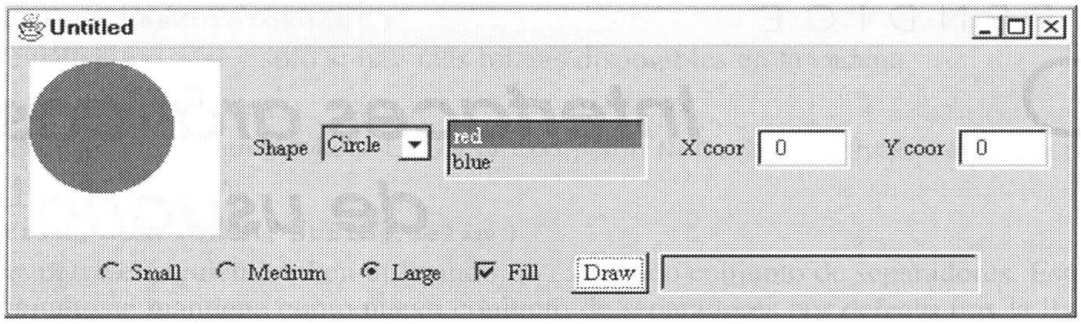


Figura D.1 Una GUI que muestra algunas de las componentes básicas.

en un programa GUI, el flujo de control es menos evidente. El programador debe suministrar el manejador de eventos empleado en la ejecución de algunos fragmentos de código.

El modelo de eventos se ha transformado de forma no compatible desde Java 1.0 a Java 1.1. En este apéndice sólo se describe la última versión.

Java 1.0 incluía un modelo de eventos que era bastante complejo de manejar. En Java 1.1 ha sido sustituido por un modelo de eventos más robusto y sencillo. Como era de esperar, los dos modelos no son completamente compatibles. Más concretamente, un compilador de Java 1.0 no puede compilar código que haga uso del nuevo modelo de eventos. Por su parte, los compiladores de Java 1.1 muestran mensajes ante el uso de las construcciones de Java 1.0. Sin embargo, el código de Java 1.0 ya compilado sí puede ejecutarse sin problemas en un intérprete de Java 1.1. En este apéndice sólo se describe el nuevo modelo de eventos.

La Figura D.1 muestra algunos de los elementos básicos incluidos en el AWT. Entre ellos están `Choice` (se ha seleccionado *Circle*), un elemento de tipo `List` (se ha seleccionado *red*), campos de texto de tipo `TextField`, campos de confirmación `Checkbox` y un botón de tipo `Button` (cuyo nombre es *Draw*). Al lado del botón puede verse un campo de texto empleado únicamente para la salida (por esta razón es más oscuro que los campos de texto que hay sobre él). En la esquina superior izquierda hay un objeto de la clase `Canvas`, empleado para dibujar y aceptar la entrada a través del ratón.

Este apéndice describe la organización básica del AWT. En primer lugar, se explican diferentes clases de objetos, cómo pueden emplearse para realizar la entrada y la salida, cómo organizarlos y cómo se manejan los distintos eventos. Tras ello, se describe el concepto relacionado de applet, por medio del cual se descarga un programa de Internet y se ejecuta en un browser como Netscape Navigator o Internet Explorer.

D.2 Elementos básicos del AWT

El AWT se organiza empleando una jerarquía de herencia de clases. En la Figura D.2 se muestra una versión reducida de esta jerarquía, ya que no se muestran algunas clases intermedias. Por ejemplo, en la jerarquía completa, `TextField` y `TextArea` extienden la clase `TextComponent`, mientras que no se muestran muchas clases de las relacionadas con fuentes de letras, colores, y otros objetos de la jerarquía de `Component`. Las clases `Font` y `Color`, definidas en el paquete `java.awt`, extienden la clase `Object`.

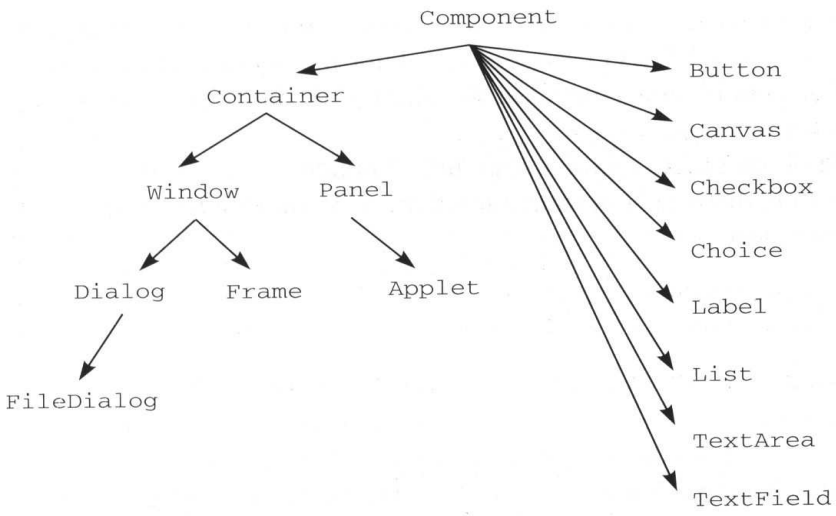


Figura D.2 Jerarquía reducida del AWT.

D.2.1 Component

La clase *Component* es una clase abstracta, superclase de muchos objetos del AWT. Debido a que es abstracta, no puede instanciarse. Un objeto de la clase Component representa un elemento que tiene una posición y un tamaño, y puede dibujarse en la pantalla, además de aceptar eventos de entrada. En la Figura D.2 se muestran algunos ejemplos de objetos de esta clase.

La clase *Component* es una clase abstracta, superclase de muchos objetos del AWT. Representa los elementos que tienen una posición, un tamaño y pueden dibujarse en la pantalla, además de aceptar eventos de entrada.

La clase Component contiene multitud de métodos. Algunos de ellos pueden emplearse para especificar el color o el tipo de fuente; otras se usan para manipular eventos. Algunos de los métodos más importantes son

```
void paint ( Graphics g );
void setSize( int width, int height );
void setBackground( Color c );
void setFont( Font f );
void show( );
```

Normalmente, el método paint se asocia a objetos de tipo Canvas. Se describe en la Sección D.3.2. El método setSize[3] se emplea para cambiar el tamaño de un objeto. Funciona sobre objetos de la clase Canvas, pero no debe invocarse sobre objetos que tengan configuración automática, como los de tipo Button. Los métodos setBackground y setFont se usan para modificar el color del fondo y el tipo de letra asociados a un objeto de la clase Component. Necesitan, respectivamente, un objeto de las clases Color y Font. Por último, el método show convierte una componente en visible. Su uso típico se realiza sobre un Frame.

D.2.2 Container

La clase *Container* es la superclase abstracta que representa todas las componentes que pueden contener a otras. Un ejemplo de subclase es la clase Window,

La clase *Container* es la superclase abstracta que representa todas las componentes que pueden contener a otras.

[3] En Java 1.1, el método setSize sustituye a resize.

del objeto `Dialog` se realizaría antes de cualquier otra cosa, a menos que se necesite un esfuerzo adicional de programación. Además, como el `Frame f` no contiene ningún otro objeto, no debemos preocuparnos por la configuración de los objetos.

D.2.4 Panel

Los objetos de tipo *`Panel`* se emplean para almacenar una colección de objetos, pero no generan ningún borde. Por ello, es la más simple de las clases `Container`.

Otra subclase de `Container` es *`Panel`*. Los objetos de tipo `Panel` se emplean para almacenar una colección de objetos, pero no generan ningún borde. Por ello, es la más simple de las clases `Container`.

El uso principal de esta clase es la organización de objetos en unidades. Por ejemplo, considere un registro que requiere un nombre, dirección, número de la seguridad social y los números de teléfono del domicilio y del trabajo. Todos estos elementos podrían componer un `PersonalPanel`. Entonces, el formulario puede contener varias entidades de tipo `PersonalPanel` para permitir la posibilidad de realizar varios registros.

Como ejemplo, la Figura D.5 indica cómo se agrupan las componentes de la Figura D.1 en una clase `Panel` y muestra la técnica general para crear una subclase de `Panel`. Nos queda construir los objetos, inicializarlos de la forma adecuada y tratar el evento de pulsar el botón.

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class GUI extends Panel implements ActionListener
5  {
6      public GUI( )
7      {
8          makeTheObjects( );
9          doTheLayout( );
10         theDrawButton.addActionListener( this );
11     }
12         // Construye todos los objetos
13     private void makeTheObjects( )
14       { /* Figura D.6 */ }
15
16         // Configura todas las componentes
17     private void doTheLayout( )
18       { /* Figura D.9 */ }
19
20         // Procesa la pulsación del botón
21     private void ActionPerformed( ActionEvent evt )
22       { /* Figura D.12 */ }
23
24     private GUICanvas theCanvas;
25     private Choice    theShape;
26     private List      theColor;
27     private TextField theXCoor;
28     private TextField theYCoor;
29     private Checkbox  smallPic;
30     private Checkbox  mediumPic;
31     private Checkbox  largePic;
32     private Checkbox  theFillBox;
33     private Button    theDrawButton;
34     private TextField theMessage;
35 }
```

Figura D.5 Clase GUI básica mostrada en la Figura D.1.

Observe que `GUI` implementa la interfaz `ActionListener`. Esto significa que sabe cómo manejar un evento (en este caso, el pulsar un botón). Para que una clase implemente la interfaz `ActionListener`, debe incluir un método `actionPerformed`. Además, cuando el botón genera el evento, debe saberse qué componente va a recibir dicho evento. En este caso, gracias a la llamada de la línea 10 (en la Figura D.5), el objeto `GUI` que contiene el objeto de tipo `Button` le comunica a este último que debe producir el evento. Los detalles del procesamiento de eventos se explican en la Sección D.3.3.

Un segundo uso de los objetos `Panel` es la agrupación de objetos en unidades con el objetivo de simplificar las configuraciones. Esto se discute en la Sección D.3.4. El tercer uso de la clase `Panel` es el `Applet`, que es una subclase suya.

D.2.5 Componentes importantes de la E/S

El AWT incluye un conjunto de componentes que pueden emplearse para realizar la entrada y salida. Estas componentes son sencillas de generar y usar. El código de la Figura D.6 muestra cómo se construye cada una de las componentes de la Figura D.1. Generalmente, esto incluye la llamada a un constructor y la aplicación de un método para adaptar la componente. El código no especifica cómo se orga-

```
1       // Genera todos los objetos
2   private void makeTheObjects( )
3   {
4       theCanvas = new GUICanvas( );
5       theCanvas.setBackground( Color.white );
6       theCanvas.setSize( 100, 100 );
7
8       theShape = new Choice( );
9       theShape.addItem( "Circle" );
10      theShape.addItem( "Square" );
11
12      theColor = new List( 2, false );
13      theColor.addItem( "red" );
14      theColor.addItem( "blue" );
15      theColor.select( 0 ); // La elección por defecto es "red"
16
17      theXCoor = new TextField( 5 );
18      theYCoor = new TextField( 5 );
19
20      CheckboxGroup theSize = new CheckboxGroup( );
21      smallPic = new Checkbox( "Small", theSize, false );
22      mediumPic = new Checkbox( "Medium", theSize, true );
23      largePic = new Checkbox( "Large", theSize, false );
24
25      theFillBox = new Checkbox( "Fill" );
26      theFillBox.setState( false );
27
28      theDrawButton = new Button( "Draw" );
29      theMessage = new TextField( 25 );
30      theMessage.setEditable( false );
31  }
```

Figura D.6 Código que genera los objetos de la Figura D.1.

nizan los elementos en el panel o cómo se examina el estado de las mismas. Recordemos que la programación de las GUI consiste en mostrar la interfaz y esperar a que se produzcan eventos. La configuración de las componentes y el tratamiento de los eventos se discuten en la Sección D.3.

Label

Label es una componente utilizada para colocar texto en un contenedor. Su principal uso es etiquetar otras componentes.

Label es una componente utilizada para colocar texto en un contenedor. Su uso principal es etiquetar otras componentes como `Choice`, `List`, `TextField` o `Panel` (muchas otras componentes ya muestran su nombre de algún modo). En la Figura D.1. las frases *Shape*, *X Coor* e *Y Coor* son etiquetas. Los objetos de la clase `Label` se construyen con una cadena de caracteres opcional que puede modificarse con el método `setText`. Los métodos son

```
Label( );
Label( String theLabel );
void setText( String theLabel );
```

Button

La clase *Button* se emplea para crear botones etiquetados. Cuando se pulsa uno de ellos se genera un *evento*.

La clase *Button* se emplea para crear botones con etiqueta. La Figura D.1 contiene uno de ellos, cuya etiqueta es *Draw*. Cuando se pulsa un objeto de la clase `Button`, se genera un *evento*. La Sección D.3.3 describe cómo se procesan estos eventos. La interfaz `Button` es similar a `Label`. Más concretamente, un objeto de la clase `Button` se construye con una cadena de caracteres opcional. Esta etiqueta puede modificarse con el método `setText`. Los métodos son

```
Button( );
Button( String theLabel );
void setText( String theLabel );
```

Choice

La clase *Choice* se emplea para seleccionar una sola cadena de caracteres de entre una lista desplegable de ellas.

La clase *Choice* se emplea para seleccionar una sola cadena de caracteres de entre una lista desplegable de ellas. Sólo podemos seleccionar cadenas en la lista y en cada momento sólo se puede realizar una única elección. En la Figura D.1 el tipo de `theShape` es un objeto `Choice`; en este momento la opción elegida es *Circle*. Algunos de los métodos de esta clase son

```
Choice( );
void addItem( String item );
String getSelectedItem( );
int getSelectedIndex( );
void select( int index );
```

Un objeto `Choice` se construye sin parámetros. Se pueden añadir cadenas de caracteres a la lista de alternativas `Choice`. Cuando se invoca al método `getSelectedItem`, se devuelve una cadena que representa el elemento elegido (o `null` si no se realiza ninguna elección). Gracias al método `getSelectedIndex`,

puede devolverse el índice de la opción (calculado por el orden de llamadas a `addItem`) en lugar de devolver la cadena correspondiente. El primer elemento añadido tiene índice 0, el siguiente índice 1 y así sucesivamente. Esto puede ser muy útil ya que si un vector guarda información relativa a cada una de las elecciones, `getSelectedIndex` puede emplearse para indexar dicho vector. El método `select` se utiliza para especificar cuál es la elección por defecto.

List

La componente *`List`* permite la selección de una cadena de caracteres de entre una lista desplegada de ellas. En la Figura D.1 la elección de los colores se presenta en forma de lista. Los objetos de la clase `List` se distinguen de los objetos de la clase `Choice` por tres detalles:

La componente *`List`* permite la selección de una cadena de caracteres de entre una lista desplegada de ellas. Puede configurarse para permitir selecciones de uno o varios elementos.

1. Los objetos de la clase `List` pueden configurarse de modo que permitan seleccionar uno o varios elementos.
2. Los objetos de la clase `List` permiten al usuario ver varias opciones al mismo tiempo.
3. Los objetos de la clase `List` ocupan mayor espacio en la pantalla que los de la clase `Choice`.

Los métodos básicos de esta clase son

```
List( );
List( int rows, boolean multipleSelections );
void addItem( String item );
String getSelectedItem( );
String [ ] getSelectedItems( );
void select( int index );
```

Un objeto de `List` se construye sin parámetros o con dos de ellos. El constructor con dos parámetros especifica el número visible de filas (en otras palabras, el número de filas que se mostrarán) y un booleano que determina si se permiten selecciones múltiples. Los métodos `addItem`, `getSelectedItem` y `select` tienen el mismo comportamiento que los métodos correspondientes de la clase `Choice`. `getSelectedItem` devuelve `null` si no se selecciona ningún elemento o bien se selecciona más de uno. `getSelectedItems` se emplea para tratar selecciones múltiples; devuelve un vector de cadenas de caracteres correspondiente a los elementos seleccionados. Al igual que en el caso de `Choice`, empleando otros métodos públicos se pueden obtener índices en lugar de cadenas de caracteres.

Checkbox

Una *`Checkbox`* es una componente GUI que tiene dos estados posibles: *activado* y *desactivado*. El estado *activado* es `true` y el estado *desactivado* es `false`. La Figura D.1 contiene cuatro objetos de tipo `Checkbox`. En esta figura, la caja *Fill* es `true` y es diferente a las otras tres. Esto es debido a que los otros objetos `Checkbox` forman parte de un *`CheckboxGroup`*: sólo una de las tres cajas de este grupo podría ser `true`. Cuando se selecciona un objeto `Checkbox` de un grupo, todos los demás objetos del grupo dejan de estar seleccionados. Un objeto de

Una *`Checkbox`* es una componente GUI que tiene dos estados posibles: *activado* y *desactivado*. Un *`CheckboxGroup`* puede contener un conjunto de cajas `Checkbox`, de entre las cuales sólo una puede ser cierta en cada momento.

la clase `CheckboxGroup` se construye sin parámetros. Observe que no se trata de un elemento de tipo `Component`; es sólo un objeto lógico.

Los métodos más empleados en esta clase son

```
Checkbox( );
Checkbox( String theLabel );
Checkbox( String theLabel, CheckboxGroup group, boolean state );
boolean getState( );
void setLabel( );
void setState( );
```

Una caja `Checkbox` individual se construye con una etiqueta opcional. Si no se especifica ninguna, puede añadirse posteriormente una con `setLabel`. `setLabel` también puede emplearse para cambiar la etiqueta existente. Un objeto de la clase `Checkbox` que forma parte de un `CheckboxGroup` se construye indicando una etiqueta, el grupo y su estado inicial. Nótese que si `state` es `true`, el estado de todas las cajas de ese grupo construidas anteriormente se actualiza a `false`. `setState` es empleado con mayor asiduidad para establecer el estado por defecto de una caja `Checkbox` individual. `Checkbox.getState` devuelve el estado de un objeto de tipo `Checkbox`.

Canvas

Una componente *`Canvas`* representa un área de la pantalla, rectangular y en blanco, sobre la que la aplicación puede dibujar o recibir eventos de entrada.

Una componente *`Canvas`* representa un área de la pantalla, rectangular y en blanco, en la que la aplicación puede dibujar. Los gráficos más sencillos se describen en la Sección D.3.2. Un objeto de la clase `Canvas` también puede recibir datos de entrada del usuario en forma de eventos producidos por el ratón o el teclado. Los elementos de esta clase nunca se emplean directamente: el programador define una subclase de `Canvas` con la funcionalidad adecuada. La subclase diseñada debe sobrescribir el método

```
void paint( Graphics g );
```

TextField

Un objeto de la clase *`TextField`* es una componente que se muestra al usuario como una única línea de texto. Un objeto de la clase *`TextArea`* se muestra con varias líneas y su funcionalidad es similar.

Un objeto de la clase *`TextField`* es una componente que se muestra al usuario como una única línea de texto. Un objeto de la clase *`TextArea`* permite mostrar varias líneas y su funcionalidad es similar. Por este motivo, en esta sección sólo consideramos la clase `TextField`. Por defecto, el texto puede ser editado por el usuario, pero es posible convertirlo en no editable. En la Figura D.1 aparecen tres objetos de tipo `TextField`: dos para las coordenadas y una tercera, no editable por el usuario, que se emplea para los mensajes de error. El color de fondo de un campo de texto no editable es distinto del de un campo de texto editable. Algunos de los métodos más usuales de esta clase son

```
TextField( );
TextField( int cols );
TextField( String text, int cols );
String getText( );
void setEditable( boolean editable );
void setText( String text );
```

Un objeto de la clase `TextField` se construye sin parámetros o especificando un texto opcional inicial y el número de columnas. Una advertencia: en muchos sistemas es necesario especificar más columnas de las que se espera emplear, debido a un aparente fallo en el AWT. El método `setEditable` se emplea para impedir la escritura en el campo de texto. `setText` puede emplearse para escribir mensajes de texto en el objeto `TextField`. `getText` se utiliza para leer el texto del objeto `TextField`.

En muchos sistemas es necesario especificar más columnas de las que se espera utilizar, debido a un aparente fallo en el AWT.

D.3 Principios básicos del AWT

Esta sección estudia tres facetas importantes de la programación AWT. En primer lugar, se estudia cómo se organizan los objetos dentro de un contenedor. Seguidamente se explica cómo se tratan los eventos tales como la pulsación de botones. Por último, describimos cómo se realizan gráficos en el interior de los objetos de tipo `Canvas`.

D.3.1 Configuradores

Un *configurador* organiza de forma automática las componentes dentro de un contenedor. Se asocia a un contenedor mediante el comando `setLayout`. Un ejemplo de su uso es la llamada

```
setLayout( new FlowLayout( ) );
```

El *configurador* organiza de forma automática las componentes dentro de un contenedor. Un configurador se asocia a un contenedor mediante el método `setLayout`.

Observe que no se necesita guardar una referencia a un configurador. El contenedor sobre el que se aplica el comando `setLayout` lo hace, almacenándolo como un atributo privado. Cuando se emplea un configurador, no funcionan muchas peticiones para cambiar el tamaño de algunas componentes, como los botones, ya que el configurador escoge sus propias dimensiones. La idea es que el configurador determinará las dimensiones ideales para satisfacer las especificaciones.

`FlowLayout`

El más sencillo de los configuradores es *`FlowLayout`*. Cuando un contenedor se configura empleando `FlowLayout`, sus componentes se añaden en una fila de izquierda a derecha. Cuando no queda espacio libre en una fila, se forma una nueva. Por defecto, cada fila está centrada. Esto puede modificarse especificando un parámetro adicional en el constructor con el valor `FlowLayout.LEFT` o `FlowLayout.RIGHT`.

El más sencillo de los configuradores es *`FlowLayout`*, que añade componentes en una fila de izquierda a derecha.

El problema al emplear este configurador aparece cuando una fila se rompe en un lugar inadecuado. Por ejemplo, si una fila es demasiado corta, puede producirse una separación entre un objeto de tipo `Label` y otro de tipo `TextField`, aunque ambos deban aparecer juntos. Una forma de evitar estas complicaciones es crear un `Panel` con esos dos elementos y añadir después dicho `Panel` al contenedor. Otro problema que presenta `FlowLayout` es alinear verticalmente los objetos.

El configurador `FlowLayout` es el empleado por defecto en un objeto de tipo `Panel`.

BorderLayout

El configurador *BorderLayout* es el que se emplea por defecto con objetos de la jerarquía Window, como Frame o Dialog. Configura un contenedor situando sus componentes en cinco posiciones predeterminadas.

El configurador *BorderLayout* es el que se emplea por defecto con objetos de la jerarquía `Window`, como `Frame` o `Dialog`. Configura un contenedor situando sus componentes en cinco posiciones predeterminadas. Para que esto suceda, el método `add` debe emplearse con un segundo parámetro cuyo valor debe ser `"North"`, `"South"`, `"East"`, `"West"` o `"Center"`. La Figura D.7 muestra cinco botones añadidos en un `Frame` empleando `BorderLayout`. El código para generar esta configuración se muestra en la Figura D.8. Normalmente, alguna de las cinco posiciones queda vacante. Además, las componentes empleadas suelen ser objetos de tipo `Panel` que contienen otras componentes siguiendo configuraciones diferentes.

Como ejemplo, el código de la Figura D.9 muestra cómo se organizan los objetos de la Figura D.1. Aquí se tienen dos filas, pero deseamos asegurar que las cajas de confirmación, los botones, y las cajas de salida de texto se encuentran debajo del resto de la GUI. Para ello, la idea es crear dos objetos de la clase `Panel`, uno que contiene los elementos de la parte superior y otro que contiene los de la parte inferior. Los paneles se colocan uno encima de otro, configurándolos mediante `BorderLayout`.

Las líneas 4 y 5 crean dos objetos de tipo `Panel`, `topHalf` y `bottomHalf`. Cada uno de estos objetos se configuran independientemente usando `FlowLayout`. Observe que los métodos `setLayout` y `add` se aplican al `Panel` apropiado. Debido a que estos paneles se configuran con `FlowLayout`, pueden consumir más de una fila y no tener suficiente espacio libre disponible. Esto podría causar una sepa-

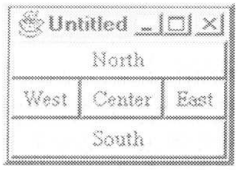


Figura D.7 Cinco botones organizados por `BorderLayout`.

```
1  import java.awt.*;
2
3      // Genera la Figura D.7
4  public class BorderTest extends Frame
5  {
6      public static void main( String [ ] args )
7      {
8          Frame f = new BorderTest( );
9
10         f.setLayout( new BorderLayout( ) );
11         f.add( new Button( "North" ), "North" );
12         f.add( new Button( "East" ), "East" );
13         f.add( new Button( "South" ), "South" );
14         f.add( new Button( "West" ), "West" );
15         f.add( new Button( "Center" ), "Center" );
16
17         f.pack( );      // Reduce el marco al tamaño mínimo
18         f.show( );      // Muestra el marco
19     }
20 }
```

Figura D.8 Código que ilustra el uso de `BorderLayout`.

```
1       // Sitúa todos los objetos
2  private void doTheLayout( )
3  {
4       Panel topHalf = new Panel( );
5       Panel bottomHalf = new Panel( );
6
7            // Coloca la primera mitad
8       topHalf.setLayout( new FlowLayout( ) );
9       topHalf.add( theCanvas );
10      topHalf.add( new Label( "Shape" ) );
11      topHalf.add( theShape );
12      topHalf.add( theColor );
13      topHalf.add( new Label( "X coor" ) );
14      topHalf.add( theXCoor );
15      topHalf.add( new Label( "Y coor" ) x);
16      topHalf.add( theYCoor );
17
18           // Coloca la segunda mitad
19      bottomHalf.setLayout( new FlowLayout( ) );
20      bottomHalf.add( smallPic );
21      bottomHalf.add( mediumPic );
22      bottomHalf.add( largePic );
23      bottomHalf.add( theFillBox );
24      bottomHalf.add( theDrawButton );
25      bottomHalf.add( theMessage );
26
27           // Configuración de la GUI
28      setLayout( new BorderLayout( ) );
29      add( topHalf, "North" );
30      add( bottomHalf, "South" );
31 }
```

Figura D.9 Código que configura los objetos de la Figura D.1.

ración entre los objetos de tipo `Label` y los de tipo `TextField`. Se deja como ejercicio al lector la creación de objeto de tipo `Panel` adicionales para asegurar que no se producen estas separaciones entre una etiqueta y su componente correspondiente. Una vez que los paneles han sido generados se emplea `BorderLayout` para alinearlos. Esto se hace en las líneas 28 a 30. Observe que las componentes de los dos objetos están centradas. Éste es el resultado de `FlowLayout`. Para alinear a la izquierda el contenido de los paneles, las líneas 8 y 19 podrían construir el objeto `FlowLayout` con el parámetro adicional `FlowLayout.LEFT`.

Cuando se emplea `BorderLayout` y los métodos `add` se utilizan sin cadenas de caracteres, la posición por defecto es `"Center"`. Si se especifica un `String`, pero no es ninguno de los cinco mencionados, se lanza una excepción[6].

Cuando se emplea `BorderLayout` y los métodos `add` se utilizan sin cadenas de caracteres, la posición por defecto es `"Center"`.

El configurador `null`

El *configurador* `null` se emplea para situar los elementos en una posición exacta. En este configurador, cada objeto se incluye en el contenedor empleando `add`. Su tamaño y posición se actualizan mediante una llamada al método `setBounds`:

El configurador `null` se emplea para situar los elementos en una posición exacta.

```
void setBounds( int x, int y, int width, int height )
```

[6] Tenga en cuenta que en Java 1.0, los argumentos de `add` están invertidos, y las cadenas de caracteres incorrectas u omitidas son ignoradas, lo que dificulta la depuración. Este estilo aún se permite, pero se desaconseja.

Aquí `x` e `y` representan la posición del extremo superior izquierdo del objeto con respecto al extremo superior izquierdo de su contenedor. `width` y `height` representan el tamaño del objeto. Las unidades empleadas son pixels.

El configurador `null` depende de la plataforma empleada, y esto normalmente es un gran inconveniente.

Configuradores estéticos

Otros configuradores simulan hojas de índices tabulados y permiten organizar los elementos sobre una malla arbitraria.

Java también pone a nuestra disposición `CardLayout`, `GridLayout` y `GridBagLayout`. El configurador `CardLayout` simula la estética de las aplicaciones de Windows, pero tiene una apariencia terrible en el AWT. El configurador `GridLayout` añade componentes en una malla, pero hace que todas las entradas a dicha malla tengan exactamente el mismo tamaño. Esto hace que en algunas ocasiones las componentes se agranden de manera extraña. Es muy útil cuando esto no puede provocar problemas, por ejemplo en el teclado de una calculadora que consiste en una malla bidimensional de botones. El configurador `GridBagLayout` incorpora las componentes en una malla, pero permite que un mismo elemento ocupe varias celdas. Su uso es más complicado que el del resto de configuradores.

Herramientas visuales

Algunos productos comerciales, como Symantec Café y Microsoft J++, incluyen herramientas que permiten al programador diseñar la configuración usando un sistema de tipo CAD. La herramienta produce el código Java necesario para construir los objetos y colocarlos de la forma indicada. Normalmente, genera la interfaz empleando el configurador `null`. Aun utilizando estos sistemas el programador debe escribir la mayoría del código, incluyendo el tratamiento de los eventos; pero, es cierto que evitan el sucio trabajo de calcular la posición precisa de los objetos.

D.3.2 Gráficos

Los gráficos se dibujan definiendo una clase que extienda a la clase `Canvas`. La nueva clase debe sobrescribir el método `paint` y tener un método público que pueda invocarse desde el contenedor del objeto `Canvas`.

Como ya hemos comentado en la Sección D.2.5, los gráficos se dibujan empleando un objeto de tipo `Canvas`. Más concretamente, para generar gráficos, el programador debe definir una nueva clase que extienda a `Canvas`. Esta nueva clase debe contener un constructor (el generado por defecto es inaceptable), sobrescribir el método `paint` y tener un método que pueda invocarse desde el contenedor del objeto `Canvas`. El método `paint` es

```
void paint( Graphics g );
```

`Graphics` es una clase abstracta que define una gran cantidad de métodos de dibujo.

`Graphics` es una clase abstracta que define una gran cantidad de métodos. Algunos de ellos son

```
void drawOval( int x, int y, int width, int height );
void drawRect( int x, int y, int width, int height );
void fillOval( int x, int y, int width, int height );
void fillRect( int x, int y, int width, int height );
void drawLine( int x1, int x2, int y1, int y2 );
void drawString( String str, int x, int y );
void setColor( Color c );
```

En Java, las coordenadas son relativas al extremo superior izquierdo de la componente. Los métodos `drawOval`, `drawRect`, `fillOval` y `fillRect` dibujan un objeto con los atributos `height` y `width` especificados, cuyo extremo superior izquierdo está indicado por los parámetros `x` e `y`. `drawLine` y `drawString` dibujan líneas y texto, respectivamente. El método `setColor` se emplea para cambiar el color actual; el nuevo color se emplea en todas las rutinas de dibujo, hasta que es cambiado de nuevo.

En Java, las coordenadas son relativas al extremo superior izquierdo de la componente.

La Figura D.10 muestra cómo se implementa el objeto de tipo `Canvas` de la Figura D.1. La nueva clase `GUICanvas` extiende a `Canvas`. Se definen varios atributos privados que describen el estado actual del objeto `Canvas`. El constructor por defecto de `GUICanvas` es razonable, luego lo aceptamos.

```
1  import java.awt.*;
2
3  public class GUICanvas extends Canvas
4  {
5      public void setParams( String theShape, String theColor,
6                             int x, int y, int size, boolean fill )
7      {
8          this.theShape = theShape;
9          this.theColor = theColor;
10         xcoor = x; ycoor = y;
11         theSize = size;
12         fillOn = fill;
13         repaint( );
14     }
15
16     public void paint( Graphics g )
17     {
18         if( theColor.equals( "red" ) )
19             g.setColor( Color.red );
20         else if( theColor.equals( "blue" ) )
21             g.setColor( Color.blue );
22         width = 25 * ( theSize + 1 );
23
24         if( theShape.equals( "Square" ) )
25             if( fillOn )
26                 g.fillRect( xcoor, ycoor, width, width );
27             else
28                 g.drawRect( xcoor, ycoor, width, width );
29         else if( theShape.equals( "Circle" ) )
30             if( fillOn )
31                 g.fillOval( xcoor, ycoor, width, width );
32             else
33                 g.drawOval( xcoor, ycoor, width, width );
34     }
35
36     private String theShape = "";
37     private String theColor = "";
38     private int xcoor;
39     private int ycoor;
40     private int theSize; // 0 = pequeño, 1 = mediano, 2 = grande
41     private boolean fillOn;
42     private int width;
43 }
```

Figura D.10 Objeto `Canvas` básico mostrado en la esquina superior izquierda de la Figura D.1.

```
1  public void update( Graphics g )
2  {
3  paint( g );
4  }
```

Figura D.11 Sobrescritura de `update` para evitar el borrado en un `repaint`.

Los atributos se actualizan con el método `setParams`, definido de modo que el contenedor (esto es, la clase `GUI` que almacena el `GUICanvas`) puede comunicar al `GUICanvas` el estado de sus componentes de entrada. `setParams` se muestra en las líneas 5 a 14. Su última línea es una llamada al método `repaint`.

El método `repaint` invoca al método `update`. Por defecto, `update` vacía la componente y después llama a `paint`. Todo lo que necesitamos entonces es escribir un método `paint` que realice los dibujos tal y como se especifica en los atributos. Como puede verse en las líneas 16 a 34 de su implementación, el método `paint` se limita a llamar a los métodos de la clase `Graphics` descritos anteriormente en este mismo apéndice.

El método `repaint` invoca al método `update`. Por defecto, `update` vacía la componente y después llama a `paint`. Este comportamiento puede modificarse sobrescribiendo `update`.

Recuerde que, por defecto, una llamada a `repaint` vacía la componente. En algunas ocasiones, preferiríamos escribir en ella en lugar de borrarla. Para hacerlo, sobrescribimos el método `update` en la nueva clase, tal y como se muestra en la Figura D.11.

Si no hay hebras, `repaint` puede aparecer como la última instrucción.

Es importante observar que, a menos que se empleen hebras (Sección D.4), la llamada al método `update` no es inmediata y suele retrasarse hasta que se produzca un evento. En tal caso, el código que sigue a `repaint` se ejecuta antes de volver a pintar.

D.3.3 Eventos

El sistema inicial de tratamiento de eventos de Java era bastante complejo y ha sido completamente rehecho.

Cuando el usuario utiliza el ratón o el teclado, el sistema operativo produce un evento. El sistema inicial de tratamiento de eventos de Java era bastante complejo y ha sido completamente rehecho. El nuevo modelo es mucho más sencillo de programar que el antiguo. Observe que ambos modelos son incompatibles: los eventos de Java 1.1 no son comprensibles para los compiladores de Java 1.0 y viceversa. Las reglas básicas que deben seguirse son las siguientes:

1. Cualquier clase que incluya código para el tratamiento de eventos debe implementar una interfaz *de escucha*. Ejemplos de estas interfaces son `ActionListener`, `WindowListener` y `MouseListener`. Como es habitual, implementar una interfaz supone que todos los métodos de la interfaz deben definirse en la clase.
2. Un objeto que quiere tratar un evento generado por una componente, debe declarar ese posible tratamiento mediante un mensaje de *añadir escucha* enviado a la componente que genera dicho evento. Cuando una componente genera un evento, éste se envía al objeto que ha declarado que lo recibirá. Si ningún objeto ha declarado que lo hará, entonces es ignorado.

Cuando el usuario presiona un objeto de tipo `Button` se genera un evento, que se trata mediante un `ActionListener`.

Por ejemplo, considere el evento que se genera cuando el usuario pulsa un objeto de la clase `Button`, pulsa la tecla *Return* cuando se encuentra en una `TextField`, o realiza una selección en un objeto de la clase `List` o `MenuItem`. El modo más

sencillo de tratar la pulsación de un objeto `Button` es que su contenedor implemente la interfaz `ActionListener`, codificando un método `actionPerformed` y declarando que él es el manejador del evento mencionado.

Esto se hace sobre nuestro ejemplo en la Figura D.1, como sigue. Recuerde que en la Figura D.5 ya hemos hecho dos cosas. En la línea 4, `GUI` declara que implementa la interfaz `ActionListener`, y en la línea 10 una instancia de `GUI` se declara como el manejador del evento de pulsar su objeto `Button`. En la Figura D.12 implementamos la interfaz de escucha invocando a `setParam` desde `actionPerformed` en la clase `GUICanvas`. Este ejemplo se simplifica por el hecho de que existe un único objeto `Button`, así que cuando `actionPerformed` es invocado, sabemos lo que hacer. Si `GUI` contiene varios objetos de tipo `Button` y declara ser el manejador de los eventos producidos por todos ellos, entonces `actionPerformed` debería examinar el parámetro `evt` para determinar qué evento debe ser procesado; esto probablemente suponga ejecutar una secuencia de tests `if/else`[7]. El parámetro `evt`, que en nuestro caso es una referencia de tipo `ActionEvent`, siempre se pasa a un manejador de eventos. El tipo del evento está en función del tipo de su manejador (`ActionEvent`, `WindowEvent` y así sucesivamente), pero será siempre una subclase de `AWTEvent`.

El evento de cerrar una ventana se genera cuando se cierra una aplicación.

Un evento importante que necesita ser procesado es el evento de cierre de una ventana. Este evento se genera cuando se cierra una aplicación presionando ☒, situado en la esquina superior derecha de la ventana de la aplicación. Desafortunadamente, este evento se ignora por defecto, de modo que si no se indica ningún manejador para él, el mecanismo usual para cerrar una aplicación no funcionará.

El evento de cierre de una ventana se trata implementando la interfaz `WindowListener`.

El cierre de una ventana es uno de los muchos eventos asociado a `WindowListener`. Debido a que dicho interfaz requiere que implementemos muchos métodos (cuyos cuerpos serán probablemente vacíos), el modo más razonable de proceder es definir una clase que extienda a `Frame` e implemente la interfaz `WindowListener`. Esta clase, `CloseableFrame`, se muestra en la Figura D.13.

```
1       // Trata la pulsación del botón en el dibujo
2   public void actionPerformed( ActionEvent evt )
3   {
4       try
5       {
6           theCanvas.setParams(
7               theShape.getSelectedItem( ),
8               theColor.getSelectedItem( ),
9               Integer.parseInt( theXCoor.getText( ) ),
10              Integer.parseInt( theYCoor.getText( ) ),
11              smallPic.getState( ) ? 0 :
12                      mediumPic.getState( ) ? 1 : 2,
13              theFillBox.getState( ) );
14          theMessage.setText( "" );
15      }
16      catch( Exception e )
17          { theMessage.setText( "Incomplete input" ); }
18  }
```

Figura D.12 Código para tratar la pulsación del botón en el dibujo de la Figura D.1.

[7] Una forma de hacerlo es emplear `evt.getSource( )`, que devuelve una referencia al objeto que ha generado el evento.

```
1  import java.awt.*; import java.awt.event.*;
2  // Marco que se cierra cuando se genera un evento de cierre de ventana
3  public class CloseableFrame extends Frame implements WindowListener
4  {
5      public CloseableFrame( )
6        { addWindowListener( this ); }
7
8      public void windowClosing( WindowEvent event )
9        { System.exit( 0 ); }
10     public void windowClosed( WindowEvent event )
11       { }
12     public void windowDeiconified( WindowEvent event )
13       { }
14     public void windowIconified( WindowEvent event )
15       { }
16     public void windowActivated( WindowEvent event )
17       { }
18     public void windowDeactivated( WindowEvent event )
19       { }
20     public void windowOpened( WindowEvent event )
21       { }
22 }
```

Figura D.13 Clase `CloseableFrame`; es similar a `Frame` pero trata el evento de cierre de una ventana.

`CloseableFrame` extiende a `Frame` e implementa `WindowListener`.

El manejador del evento de cierre de ventana es sencillo; sólo realiza la llamada `System.exit`. Los métodos restantes no tienen ninguna implementación especial. El constructor declara que puede recibir el evento del cierre de una ventana. En lo sucesivo podemos emplear la clase `CloseableFrame` en lugar de la clase `Frame`.

El método `pack` comprime el `Frame` tanto como sea posible, dadas las componentes que lo forman. El método `show` muestra el `Frame`.

La Figura D.14 muestra un método `main` que puede emplearse para iniciar la aplicación de la Figura D.1. Lo colocamos en una clase aparte, a la que llamamos `BasicGUI`. `BasicGUI` extiende la clase `CloseableFrame`. El método `main` simplemente crea un objeto `Frame`, en el que colocamos un objeto `GUI`. Como sólo existe un objeto, puede emplearse el configurador `FlowLayout` para el `Frame f`. Después añadimos un objeto de tipo `GUI` sin nombre y aplicamos el método `pack` sobre el objeto `Frame`. La rutina `pack` comprime el `Frame` tanto como sea posible, dadas las componentes que lo forman. El método `show` muestra el `Frame`.

```
1  import java.awt.*;
2
3  public class BasicGUI extends CloseableFrame
4  {
5      public static void main( String [ ] args )
6      {
7          Frame f = new BasicGUI( );
8
9          f.setLayout( new FlowLayout( ) );
10         f.add( new GUI( ) );
11         f.pack( );
12         f.show( );
13     }
14 }
```

Figura D.14 Rutina `main` para la Figura D.1.

D.3.4 Resumen: encajando las piezas

He aquí un resumen de cómo crear una aplicación GUI. La funcionalidad de la GUI debe situarse en una clase que extienda a `Panel`. Para esta clase debe hacerse lo siguiente:

- Decidir cuáles son los elementos de entrada y cuáles los de salida. Si los mismos elementos se emplean varias veces, debe construirse una clase adicional para agrupar la funcionalidad común y aplicar estos principios a dicha clase.
- Si se emplean gráficos, debemos construir una nueva clase que extienda a `Canvas`. Esta clase debe implementar un método `paint` e incluir otro método público que pueda ser empleado por el contenedor para comunicarse con ella. También se necesita codificar un método constructor.
- Escoger un configurador y emplear el comando `setLayout`.
- Añadir las componentes a la GUI usando `add`.
- Tratar los eventos. La manera más sencilla de hacerlo es utilizar un objeto de tipo `Button` y capturar su pulsación con `actionPerformed`.

Una vez que la GUI ha sido implementada, una aplicación define una clase que extiende `CloseableFrame` con una rutina `main`. La rutina `main` simplemente crea una instancia de esta clase de marco extendida, coloca el panel de la GUI dentro del marco y emplea sobre él los métodos `pack` y `show`.

D.4 Animaciones y hebras

Debido a que el AWT nos permite dibujar objetos, es natural intentar realizar animaciones. Probablemente, la habilidad de Java a la hora de animar páginas Web ha sido la responsable de su apogeo entre los lenguajes de programación.

La animación en Java puede parecer trivial; todo lo que tenemos que hacer es llamar repetidamente a `repaint` con distintas peticiones. Si separamos las llamadas a `repaint` utilizando `Thread.sleep`, podemos retardar la animación hasta que alcance una velocidad adecuada. Un ejemplo de estas ideas se muestra en la Figura D.15. Aquí, intentamos mostrar cómo un círculo de radio 50 se desplaza a lo largo de la diagonal desde (0,0) hasta (199,199), donde, como es habitual, estas coordenadas representan el extremo superior izquierdo de la caja en la que está inscrito el círculo (las posiciones son relativas al extremo superior izquierdo del marco). Invocamos reiteradamente a `repaint`, separando las llamadas 25 milisegundos.

Desgraciadamente, cuando se ejecuta este programa, se observa una espera de unos 5 segundos, tras lo cual simplemente se dibuja el último círculo. Esto no es lo que queríamos.

Las animaciones requieren el uso de *hebras*.

La solución a nuestro problema es emplear *hebras* (en inglés *threads*). Cómo emplear en general las hebras es algo complicado, pero los elementos concretos necesarios para resolver nuestro problema son simples. Comenzamos creando una hebra de ejecución independiente. Para hacerlo, realizamos los siguientes pasos:

1. Hacemos que nuestra clase implemente la interfaz `Runnable`. Dicha interfaz tiene un solo método llamado `run`.

```
1  import java.awt.*;
2
3      // Intenta dibujar una secuencia animada de círculos
4      // sobre una diagonal; no funciona
5  public class BadCircles extends CloseableFrame
6  {
7      int extremity = 0;
8
9      public BadCircles( )
10     {
11         drawCircles( );
12     }
13
14     public void drawCircles( )
15     {
16         for( extremity = 0; extremity < 200; extremity++ )
17         {
18             repaint( );
19             try
20               { Thread.sleep( 25 ); }
21             catch( InterruptedException e ) { }
22         }
23     }
24
25     public void paint( Graphics g )
26     {
27         g.fillOval( extremity, extremity, 50, 50 );
28     }
29
30     public static void main( String [ ] args )
31     {
32         Frame f = new BadCircles( );
33         f.setSize( 300, 300 );
34         f.show( );
35     }
36 }
```

Figura D.15 Programa que intenta realizar una animación del movimiento de un círculo sobre una diagonal.

2. Declaramos una hebra, u objeto de la clase `Thread`, como un atributo y lo inicializamos antes de la construcción del objeto.
3. Iniciamos la hebra o `Thread`.
4. Implementamos el método `run` de modo que él haga todo el trabajo.

La Figura D.16 muestra cómo se hace esto. Los cambios hechos en la Figura D.15 se muestran en negrita (el nombre de la clase ha sido cambiado, de modo que ambas versiones están disponibles en Internet como `BadCircles` y `GoodCircles`, respectivamente). En primer lugar, en la línea 3 declaramos que la clase implementa el interfaz `Runnable`. En segundo lugar, en la línea 6 declaramos un atributo de clase `Thread` y lo construimos en la línea 10. En tercer lugar, iniciamos la ejecución de la hebra en la línea 11. Finalmente, en las líneas 14 a 17 implementamos el método `run`, que se limita a llamar al método `drawCircles`.

Como hemos mencionado anteriormente, el tema de las hebras está relacionado con las animaciones. Son muy útiles en la implementación de programas concurrentes, en los cuales existen varias hebras de ejecución. La principal dificultad

```
1  import java.awt.*;
2  public class GoodCircles extends CloseableFrame
3                           implements Runnable
4  {
5      int extremity = 0;
6      private Thread animator = null;
7
8      public GoodCircles( )
9      {
10         animator = new Thread( this );
11         animator.start( );
12     }
13
14     public void run( )
15     {
16         drawCircles( );
17     }
18
19     public void drawCircles( )
20     {
21         for( extremity = 0; extremity < 200; extremity++ )
22         {
23             repaint( );
24             try
25               { Thread.sleep( 25 ); }
26             catch( InterruptedException e ) { }
27         }
28     }
29
30     public void paint( Graphics g )
31     {
32         g.fillOval( extremity, extremity, 50, 50 );
33     }
34
35     public static void main( String [ ] args )
36     {
37         Frame f = new BadCircles( );
38         f.setSize( 300, 300 );
39         f.show( );
40     }
41 }
```

Figura D.16 Implementación correcta de la animación del círculo.

que presentan es la comunicación de las hebras entre sí y el asegurarnos de que comparten de forma consistente los recursos comunes. Por ejemplo, en muchas ocasiones es deseable que una hebra bloquee a las restantes en lo que se refiere a la modificación o incluso al acceso a un recurso compartido, como por ejemplo una variable. Java incluye mecanismos para todo ello a través de la palabra reservada `synchronized`. Puede encontrarse más información acerca de las hebras en la mayoría de las referencias de Java.

D.5 Applets

Un *applet* es un pequeño programa inmerso en otra aplicación. El sistema Java incluye una aplicación denominada *Applet Viewer* que puede utilizarse para ejecutar un applet. Sin embargo, la situación más común es que un applet se descargue vía

Un *applet* es un pequeño programa inmerso en otra aplicación. Típicamente un applet se descarga vía Internet y se ejecuta localmente por un navegador que soporte Java.

Internet y se ejecute localmente por un navegador que soporte Java, tal como *Netscape Navigator* o *Microsoft Internet Explorer*.

Cuando se incluye un applet en una página Web, su código Java se descarga desde el computador servidor siendo ejecutado por el navegador. Esto tiene la ventaja de rebajar la carga en el servidor. Esto se debe a que una vez que se descarga el código Java, todo el trabajo se hace localmente.

Java se considera una opción para los futuros «computadores en la red».

Una de las fascinantes direcciones futuras que Java está empezando a explorar tiene que ver con el almacenamiento de software en un sitio Web. Actualmente, los usuarios de un computador personal deben instalar el software en su propia máquina. Esto requiere un cierto espacio en disco, y limita por tanto la cantidad de software diferente que puede estar simultáneamente instalado. Además, existe el problema de mantener la última versión de cada software, que habitualmente incluye la corrección de algunos errores detectados. Con los applets, todo el software se almacenaría en un sitio Web y los usuarios pagarían por cada uso del software. Esto ahorraría espacio en disco, permitiría que hubiese más software disponible y haría más fácil asegurar que siempre se utiliza la última versión disponible. El resultado sería el denominado «computador en la red» y podría ser significativamente más barato que un PC actual. En el momento en que se escribió este libro, el uso más común de Java consista en el diseño de animaciones que adornaban páginas Web que en otro caso serían tristes y estáticas.

Para usar un applet, el usuario debe conocer algo acerca del lenguaje de hipertexto HTML, el lenguaje que entienden los navegadores Web, a fin de que el navegador cargue correctamente el código del applet. Un applet también se implementa de una forma algo diferente a un programa de aplicación. En particular, no se llama a `main`. También, los applets se ejecutan con restricciones adicionales de seguridad. Sin embargo, dejando aparte los temas de la seguridad, las diferencias entre la programación de un applet y un programa de aplicación son relativamente pequeñas, lo que permite escribir programas que sirvan como aplicación y como applet. El resto de esta sección estudia estos temas.

D.5.1 Lenguaje de hipertexto

El *lenguaje de hipertexto (HTML)* es el lenguaje que entienden los navegadores Web. Un fichero HTML está formado por una secuencia de mandatos de formato y texto.

El *lenguaje de hipertexto HTML* es el lenguaje que entienden los navegadores Web. Un fichero HTML está formado por una secuencia de mandatos de formato y texto, que circula por Internet hasta el navegador. HTML es un híbrido entre los lenguajes de procesamiento de textos *troff* y *LaTeX*. Ya desde el principio, permitía el cambio de colores y fuentes, la importación de imágenes, la creación de enlaces a otras páginas Web y la generación automática de listas numeradas y con puntos. Recientemente, se ha hecho más complejo, incorporando, entre otras cosas, tablas y múltiples marcos.

La Figura D.17 muestra una breve página que contiene enlaces a las transparencias que acompañan este libro. Su código HTML se muestra en la Figura D.18. Los comandos de formato se incluyen dentro de corchetes <>: típicamente, un mandato como `<ul>`, que indica el comienzo de una lista no ordenada (o con puntos), se termina con la marca `</ul>` (observe cómo se ha añadido la barra). También se muestra el mandato `<li>`, que representa una entrada de la lista. Los ficheros HTML deberían empezar con las etiquetas `<HTML>` y `<BODY>` y terminar

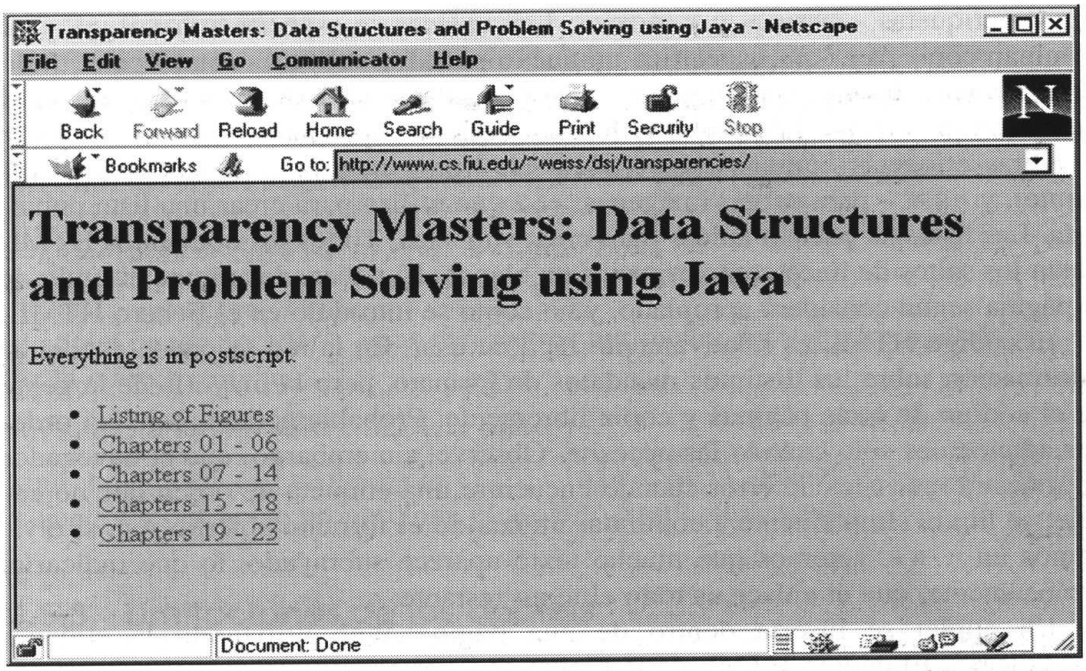


Figura D.17 Página Web, vista a través de Netscape Navigator, correspondiente al código HTML de la Figura D.18.

```
<HTML>
<BODY>

<TITLE>
Transparency Masters:
Data Structures and Problem Solving using Java
</TITLE>

<H1>
Transparency Masters:
Data Structures and Problem Solving using Java
</H1>

<p>
Everything is in postscript.

<ul>
<li><A HREF="TransparencysTOC.ps">Listing of Figures</A></li>
<li><A HREF="Transparencys01-06.ps">Chapters 01 - 06</A></li>
<li><A HREF="Transparencys07-14.ps">Chapters 07 - 14</A></li>
<li><A HREF="Transparencys15-18.ps">Chapters 15 - 18</A></li>
<li><A HREF="Transparencys19-23.ps">Chapters 19 - 23</A></li>
</ul>

</BODY>
</HTML>
```

Figura D.18 Código HTML para generar la página Web de la Figura D.17, que contiene las transparencias de este libro.

con las etiquetas </HTML> y </BODY>. Los enlaces se especifican con <A> y se terminan con </A>. <p> especifica un nuevo párrafo, <TITLE> establece el título de la ventana del navegador, y <H1> especifica que el texto se mostrará con la fuente del primer nivel de título, utilizando la fuente apropiada.

Los comandos de formato se incluyen dentro de <>.

Otras etiquetas comunes son <B> e <I> (para negrita e itálica, respectivamente), y <SRC> para cargar imágenes. se utiliza para crear una lista numerada. Las listas se pueden anidar para crear sublistas. Todos los comandos HTML tratan los saltos de línea como espacios en blanco. El navegador rompe el título de la página según considera apropiado, y no como se introdujo en el fichero HTML.

El código HTML es relativamente fácil de usar. En la red se puede encontrar información sobre los distintos mandatos de formato, pero la mayoría de la gente ve el código de otras páginas y copia libremente. Probablemente aprenderá ordenes adicionales sólo cuando las necesite. Observe, sin embargo, que el navegador no muestra mensajes de error cuando encuentre una etiqueta de orden que no encaje; se limita simplemente a continuar utilizando el formato abierto. Así, si olvidamos un </A>, veremos que mucho texto aparece subrayado, lo que indicaría, erróneamente, que el enlace es todo el texto restante.

El tamaño del applet, en pixels, debe incluirse en la etiqueta <APPLET>.

Un applet se inserta dentro de una página Web utilizando la etiqueta <APPLET>[8]. El código HTML de la Figura D.19 es el ejemplo más simple de página Web que contiene un applet. De hecho, esta página no contiene nada más. Observemos que una página Web puede contener otras cosas junto a un applet, así como varios applets. Dentro de la etiqueta <APPLET> se encuentra el nombre del fichero que contiene el código Java y las dimensiones (en pixels) del applet en el navegador.

Si el applet no se encuentra en el mismo lugar que la página Web, se debe proporcionar el atributo CODEBASE (preferentemente antes del atributo CODE). Esto permite referenciar directamente applets escritos por otros.

Algunos navegadores no permiten que los applets utilicen paquetes definidos por el usuario. Si tiene código que usa paquetes, necesitará extraer las clases utilizadas de sus paquetes.

Observe que la mayoría de los applets estarán formados por varias clases: sólo se especifica la clase con la que comienza el applet. Sin embargo, todas las clases deben estar disponibles y deben poder ser leídas por cualquier usuario. En otro caso, se obtendrá un error de ClassLoader[9]. Si se utilizan paquetes, se debe establecer una jerarquía de directorios. Sin embargo, algunos navegadores no permiten que los applets utilicen paquetes definidos por el usuario. Esto quiere decir que si tiene un código que usa paquetes, necesitará extraer las clases utilizadas de sus paquetes.

```
1  <HTML>
2  <BODY>
3
4  <APPLET code = "BasicGUIApplet.class" width = "600" height = "150">
5  </APPLET>
6
7  </BODY>
8  </HTML>
```

Figura D.19 Fichero HTML para el applet BasicGUIApplet.

[8] Hay una propuesta para reemplazar esta etiqueta por <EMBED>.

[9] Los mensajes de error, además de escribirse en System.out, se colocan en la consola de Java. La mayoría de los navegadores tienen una opción que permite ver esta consola.

D.5.2 Parámetros

Los applets pueden invocarse con parámetros, utilizando la etiqueta `PARAM` entre `<APPLET>` y `</APPLET>`. Por ejemplo, puede tener

Los applets pueden invocarse con parámetros, utilizando la etiqueta `PARAM`.

```
<APPLET CODE="Programa.class" WIDTH="150" HEIGHT="150">
<PARAM NAME="InitialColor" VALUE="Blue">
<PARAM NAME="InitialShape" VALUE="Oval">
</APPLET>
```

El applet puede acceder a estos parámetros por medio del método `getParameter` (de la clase `Program`):

Un applet puede acceder a sus parámetros mediante el método `getParameter`.

```
String getParameter( String name );
```

Aquí, la llamada `getParameter("InitialColor")` devuelve `"Blue"`. Si en la etiqueta `PARAM` no se proporciona el atributo `NAME`, se devuelve `null`.

D.5.3 Limitaciones de los applets

Un applet representa código que se descarga vía Internet y se ejecuta en nuestro computador. El proceso de descarga y ejecución empieza tan pronto como se visita la página Web que contiene el applet. Como resultado, es esencial que exista la garantía de que el applet no intente hacer algo malicioso en el computador local. Algo malicioso puede ser introducir un virus o acceder a información confidencial. Por ejemplo, si un applet tiene la capacidad de escribir en ficheros, sería muy fácil para un hacker elaborar un applet destructivo y borrar discos duros de visitantes confiados.

Como resultado, los applets se ejecutan con severas restricciones y pueden hacer un número significativo de cosas menos que una aplicación. Enumeramos a continuación algunas operaciones que un applet no puede realizar, explicando por qué es necesaria cada restricción. Tenga en cuenta que ésta no es una lista exhaustiva. Estas restricciones no se comprueban en tiempo de compilación. Por el contrario, si un applet intenta realizar cualquiera de estas operaciones, se producirá una excepción en tiempo de ejecución. Algunos navegadores eliminan estas restricciones cuando el applet se carga del sistema local. En consecuencia, al probar su applet, puede encontrar que funciona cuando se ejecuta localmente, pero no cuando se ejecuta en Internet.

Los applets se ejecutan con restricciones severas y hacen bastantes menos cosas que las aplicaciones.

1. *Los applets no pueden borrar ficheros del sistema local.* La razón de esta restricción se explicó anteriormente en esta sección.
2. *Los applets no pueden escribir ficheros en el sistema local.* En otro caso, todos los ficheros podrían ser reducidos a tamaño 0 o sobrescritos por un applet maligno.
3. *Los applets no pueden leer ficheros del sistema local.* En otro caso, el applet podría acceder a información potencialmente confidencial y transferirla al servidor.
4. *Los applets no pueden renombrar ficheros del sistema local.* El renombramiento permite la eliminación de ficheros importantes del sistema y el ocultamiento de ficheros por un applet maligno.
5. *Los applets no pueden crear o listar directorios, comprobar la existencia de un fichero, u obtener el tipo, tamaño o fecha de modificación de un*

fichero. Todas estas restricciones se requieren para evitar que un applet maligno obtenga información del sistema.

6. *Los applets no pueden crear una conexión por red con cualquier computador distinto de aquél del cual se obtuvo el applet. En particular, el método* `getURL` *está limitado*. En otro caso, un applet cargado por Internet podría acceder a computadores de una intranet que estén protegidos, obteniendo potencialmente información confidencial.
7. *Los applets no pueden escuchar o aceptar conexiones de otros puertos en el sistema local*. De otra forma el applet podría interpretar información local.
8. *Cuando el applet crea una ventana al nivel más alto, como una caja de diálogo, debe aparecer un mensaje de que la ventana no es fiable*. Esto previene a los programas locales confiados.
9. *Los applets no pueden invocar a programas locales (llamando a* `Runtime.exec`*)*. Esto generaría absurdos agujeros en la seguridad.
10. *Un applet no puede averiguar las propiedades del usuario, tales como el nombre o el directorio principal*. Esto permitiría la transmisión de información privada.
11. *Los applets no pueden llamar a* `System.exit`. Esto provocaría la terminación del navegador.

Todavía son posibles los ataques de negación de servicio, en los cuales un applet utiliza la mayoría de los recursos del sistema.

Aun con estas restricciones, es imposible asegurar el sistema completamente. *Ataques de negación de servicio* son todavía posibles; es decir, un applet Java que simplemente monopolice la mayoría de los recursos del sistema, tales como la memoria o los ciclos de CPU. Además los investigadores encuentran casi todos los meses nuevos agujeros en la seguridad.

Se pueden evitar algunas de las restricciones si se ha probado que el applet proviene de una fuente segura.

Estas restricciones pueden resultar problemáticas. Por ejemplo, ¿cómo se puede conseguir el objetivo de distribuir software por Internet (tales como procesadores de textos) si estos programas no pueden acceder a los recursos fundamentales del sistema local? La respuesta es que ello es ciertamente imposible. En Java 1.1, algunas de estas restricciones pueden evitarse en aquellos applets que han probado ser parte de un código seguro. En otras palabras, el usuario local puede permitir que las applets de Microsoft se ejecuten con libertad. Cuando Microsoft diseña un applet, codifica información que prueba que el applet es de hecho de Microsoft y por tanto se puede confiar en él. Esto se denomina *firmar* un applet, y requiere la tecnología de encriptación de clave pública. Algunas de las ideas de la encriptación de clave pública se estudiaron en la Sección 7.4.4.

D.5.4 Conversión de una aplicación en un applet

En general, es bastante simple convertir una aplicación en un applet, siempre que la aplicación no intente ejecutar ninguna de las acciones prohibidas a los applets. Las modificaciones básicas a realizar son las siguientes:

1. El applet debe importar `java.applet.*`.
2. La clase que define el applet debe extender a la clase `Applet`.
3. No se utiliza la rutina `main`.
4. Se reemplaza el constructor por la rutina `public void init( )`.

Hay otras cosas a tener en cuenta si el applet utiliza hebras para lanzar varios procesos concurrentes. Cuando un usuario abandona una página Web que contiene un applet, se detiene la hebra principal del mismo hasta que el usuario regrese a la página (en cuyo momento se reinicia el applet). Sin embargo, las hebras adicionales no se detienen, por lo que se consumirán ciclos de CPU aunque el applet ya no sea visible. El usuario debe sobrescribir el método `stop` de los applets de forma que llamen al método `stop` de cualquier hebra que se haya lanzado. Se puede usar el método `start` de los applets para crear una nueva hebra y comenzar a ejecutarla, llamando al método `start` de la hebra. En la Sección D.5.5 se habla de los applets animados.

La Figura D.20 muestra un applet que reutiliza la componente GUI que se ha visto a lo largo de este capítulo. Observemos que el applet es muy similar, pero no idéntico, a la aplicación de la Figura D.1. Esto es así, porque los tamaños de las componentes y el aspecto general de cada una de ellas varían de un sistema Java a otro. En la Figura D.21 se muestra el código que genera este applet, el cual se puede usar también para generar la aplicación de la Figura D.1.

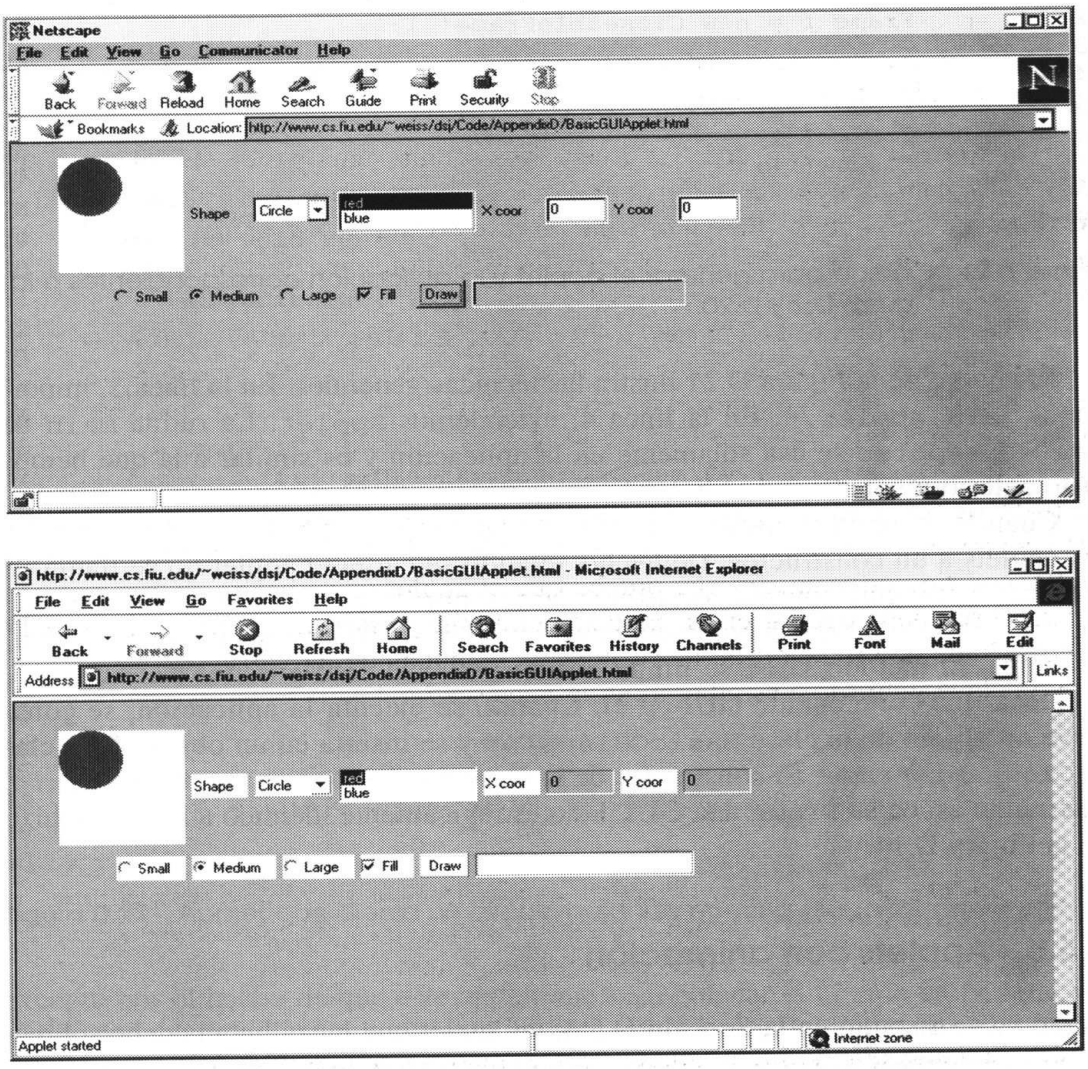


Figura D.20 Visualización del applet de la Figura D.21 en Netscape Navigator y Microsoft Internet Explorer.

```
1  import java.awt.*;
2  import java.applet.*;
3
4  public class BasicGUIApplet extends Applet
5  {
6          // El applet comienza aquí
7      public void init( )
8      {
9          setLayout( new FlowLayout( ) );
10         add( new GUI( ) );
11     }
12
13         // Se llama al constructor desde la aplicación
14     public BasicGUIApplet( )
15     {
16         init( );
17     }
18
19         // método main para una aplicación
20     public static void main( String [ ] args )
21     {
22         Frame f = new CloseableFrame( );
23
24         f.setLayout( new FlowLayout( ) );
25         f.add( new BasicGUIApplet( ) );
26         f.pack( );
27         f.show( );
28     }
29 }
```

Figura D.21 Código para generar el applet y la aplicación correspondientes a las Figuras D.1 y D.20.

Una clase applet extiende a `Applet`. Se debe importar `java.applet`.

La rutina `init` es el equivalente en los applets a un constructor. Se llama a dicha rutina cuando se carga el applet.

El código de la Figura D.21 ilustra las técnicas generales. En la línea 2, importamos `java.applet.*`. En la línea 4, extendemos `Applet`. La rutina `main` de `BasicGUIApplet` se usa solamente en la aplicación y es similar a la que hemos visto anteriormente.

Cuando se carga el applet se llama a la rutina `init`, que es el equivalente en los applets a un constructor. De hecho, cuando se carga el applet no se llama en absoluto a ningún constructor. Recordemos que `Applet` es una subclase de `Panel`, por lo que tras decidir una configuración se pueden añadir nuevas componentes. El método `init` se limita a seleccionar el configurador `FlowLayout` y coloca una componente GUI en él. Cuando se ejecuta la aplicación, se construye un objeto de la clase `BasicGUIApplet` y se inserta en un objeto de la clase `CloseableFrame`. El constructor de la clase `BasicGUIApplet` llama a `init`, insertando así en su `Frame` una GUI. Esto es lógicamente idéntico a lo que se hizo en la Figura D.14.

D.5.5 Applets con animación

Como ya se discutió en la Sección D.5.4, un applet que lanza hebras debe sobrescribir los métodos `start` y `stop` con implementaciones que llamen a los métodos `start` y `stop` de las hebras que hayan podido crearse. Ilustramos esta técnica con el applet de la Figura D.22, que es similar a la animación de la Figura D.16,

```
1  import java.awt.*;
2  import java.applet.*;
3
4  // Este applet anima un círculo moviéndolo hacia abajo a lo
5  // largo de una diagonal. Cuando llega al extremo inferior,
6  // comienza de nuevo en el extremo superior.
7
8  public class CircleApplet extends Applet implements Runnable
9  {
10     int extremity = 0;
11     private Thread animator = null;
12
13     public void stop( )
14     {
15         if( animator != null )
16             animator.stop( );
17         animator = null;
18     }
19
20     public void start( )
21     {
22         if( animator == null )
23         {
24             animator = new Thread( this );
25             animator.start( );
26         }
27     }
28
29     public void run( )
30     {
31         drawCircles( );
32     }
33
34     public void drawCircles( )
35     {
36         for( ; ; )
37             for( extremity = 0; extremity < 200; extremity++ )
38             {
39                 repaint( );
40                 try
41                   { Thread.sleep( 25 ); }
42                 catch( InterruptedException e ) { }
43             }
44     }
45
46     public void paint( Graphics g )
47     {
48         g.fillOval( extremity, extremity, 50, 50 );
49     }
50 }
```

Figura D.22 Applet que dibuja un círculo bajando por una diagonal.

excepto por el hecho de que este applet vuelve a colocar el círculo en la parte superior de la diagonal cuando alcanza la parte inferior, con lo que el programa (teóricamente) se ejecuta indefinidamente. Por ello es importante que el applet no consuma recursos una vez que el navegador abandona la página web que contiene dicho applet (recordemos que los applets no pueden llamar a `exit`).

En este applet, tenemos (a lo sumo) una hebra activa referenciada por `animator`. El método `stop` del applet termina esta hebra y asigna `null` a `animator` de forma que se puedan recolectar sus recursos. El método `start` crea una nueva hebra y la inicializa (esto tiene el efecto de llamar a `run`). No hemos sobrescrito `init`, por lo que se usa el método por defecto (que no hace nada). Esto parece correcto, ya que siempre se llama a `start` inmediatamente después de `init`.

Observe que, en este código, cuando abandonamos la página Web del applet y después volvemos, iniciamos una hebra diferente de la que se estaba ejecutando antes. Si no deseamos esto, podemos rescribir `stop` y `start` para suspender la hebra, mediante una llamada a `suspend`, y después continuar con ella, mediante una llamada a `resume` (algunos navegadores tienen errores relacionados con `suspend`).

Resumen

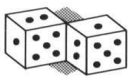
En este apéndice se han examinado las bases del Abstract Window Toolkit (AWT). El AWT es un paquete que permite programar GUI, lo que hace que los programas tengan una apariencia más profesional que si se utilizara una E/S simple por pantalla.

Las aplicaciones GUI difieren de las aplicaciones con E/S por pantalla en que están dirigidas por eventos. Para diseñar una GUI, escribimos una clase. Debemos decidir cuáles van a ser los elementos básicos de entrada y de salida, elegir una configuración de los mismos ejecutando un mandato `setLayout`, añadir componentes a la GUI usando `add`, y tratar los eventos, lo que en Java 1.1 se hace por medio de las escuchas de eventos. Todo ello forma parte de la citada clase.

Una vez hemos escrito esta clase, se define una clase que extiende a la clase `Frame`, que contiene una rutina `main` y un manejador de eventos que se encarga de procesar el evento de cierre de la ventana. La forma más simple de hacer esto es usar la clase `CloseableFrame` de la Figura D.13. La rutina `main` crea una instancia de esta clase `Frame` extendida, coloca una instancia de la clase GUI (cuyo constructor creará con seguridad un panel) dentro del objeto de la clase `Frame`, y ejecuta un mandato `pack` y otro `show` sobre dicho objeto.

Los applets son similares a las aplicaciones, excepto por el hecho de que son ejecutados por otros programas, como los navegadores. Estos últimos suelen ejecutarse con fuertes restricciones de seguridad. Difieren ligeramente en detalles como que no llaman a `main`, sino que usan un método `init` equivalente al constructor.

Aquí se han estudiado solamente las características básicas del AWT, pero hay libros enteros dedicados al tema.



Elementos del juego

Abstract Window Toolkit (AWT) Herramienta GUI que proporcionan todos los sistemas Java. Facilita las clases básicas utilizadas en la creación de interfaces de usuario.

`ActionEvent` Evento generado cuando un usuario presiona una componente `Button`, pulsa la tecla de retorno en una componente `TextField`, o seleccio-

na un elemento de una componente `MenuItem` o `List`. Debería ser tratado por método `actionPerformed` en la clase que implementa la interfaz `ActionListener`.

`actionPerformed` Método usado para tratar los eventos de acción.

`Applet` Clase que debemos extender cuando queremos implementar un applet.

applet Pequeño programa dentro de otra aplicación. Normalmente se carga desde Internet y se ejecuta localmente en un navegador compatible con Java.

ataque de negación de servicio Mecanismo de un applet de Java con malas intenciones que hace fallar a un computador consumiendo la mayor parte de los recursos del sistema, como la memoria o los ciclos de CPU.

`AWTEvent` Objeto que almacena información sobre un evento.

`BorderLayout` Es el configurador gráfico por defecto para los objetos en la jerarquía `Window`. Se usa para configurar un contenedor mediante la colocación de componentes en cinco posiciones distintas (`"North"`, `"South"`, `"East"`, `"West"`, `"Center"`).

`Button` Componente usada para crear un botón etiquetado. Cuando se pulsa el botón, se genera un evento de acción.

`Canvas` Área rectangular de la pantalla sobre la que una aplicación puede dibujar y recibir entradas del usuario en forma de eventos producidos por el teclado y el ratón.

`Component` Clase abstracta que es la superclase de muchos objetos AWT. Representa algo que tiene un tamaño y una posición y que se puede dibujar en la pantalla al igual que puede recibir eventos de entrada.

configurador Objeto que coloca de forma automática las componentes en un contenedor.

configurador `null` Configurador usado para colocar las componentes de forma precisa. Permite trabajar al método `setBounds`.

`Container` Superclase abstracta que representa todas las componentes que pueden contener a otras componentes. Normalmente tiene un configurador asociado.

`Checkbox` Componente que tiene un estado que indica si está activada o no.

`CheckboxGroup` Objeto usado para agrupar una colección de objetos `Checkbox` que garantiza que solamente uno está activado a la vez.

`Choice` Componente usada para seleccionar una única cadena de caracteres de una lista de opciones.

`Dialog` Ventana de alto nivel utilizada para facilitar diálogos.

error `ClassLoader` Error generado si no está disponible una clase necesaria como fichero `.class`.

eventos Son producidos por el sistema operativo en varias situaciones, como en las operaciones de entrada, y se pasan a Java.

`FileDialog` Ventana al más alto nivel utilizada para ofrecer una elección entre todos los ficheros en un directorio.

`FlowLayout` Configurador por defecto de `Panel`. Se usa para configurar un contenedor añadiendo componentes en fila de izquierda a derecha. Cuando no queda suficiente espacio en una fila, se genera una nueva.

`Frame` Ventana de alto nivel con borde y que puede llevar asociado un objeto `MenuBar`.

`getParameter` Método usado para acceder a los parámetros de un applet especificados en la etiqueta `<PARAM>`.

`Graphics` Clase abstracta que define varios métodos que se pueden utilizar para dibujar formas.

hebra Flujo de ejecución separado. Se pueden ejecutar varias hebras simultáneamente, generando así un programa concurrente. Las hebras son necesarias para ejecutar animaciones.

`init` Equivalente en los applets a un constructor. Se le llama cuando se carga un applet.

interfaz `ActionListener` Interfaz usada para tratar eventos de acción. Contiene el método abstracto `actionPerformed`.

interfaz gráfica de usuario (GUI) Alternativa moderna a la E/S por pantalla que permite a un programa comunicarse con el usuario mediante botones, cuadros de selección, líneas de edición, listas de selección, menús y el ratón.

interfaz `WindowListener` Interfaz usada para especificar el tratamiento de los eventos de ventana, como el cierre de una ventana.

`Label` Componente utilizada para etiquetar otras componentes como `Choice`, `List` , `TextField` o `Panel`.

lenguaje de hipertexto (HTML) Lenguaje que comprenden los navegadores Web. Consta de mandatos de texto y formato.

limitaciones de los applets Restricciones que evitan la ejecución de applets con malas intenciones. Entre otras, los applets no pueden acceder a los ficheros locales o a otros computadores distintos de aquél desde el que se ha cargado el applet.

`List` Componente que permite la selección de una lista desplegada de cadenas de caracteres. Puede permitir la selección de uno o varios elementos, pero utiliza más espacio en la pantalla que `Choice`.

`pack` Método usado para compactar un objeto `Frame` hasta conseguir que su tamaño sea lo menor posible, en función de sus componentes.

`paint` Método usado para dibujar en una componente. Normalmente lo sobrescriben las clases que extienden a `Canvas`.

`Panel` Contenedor usado para almacenar una colección de objetos, pero sin bordes.

`repaint` Método usado para volver a dibujar una componente. Por defecto llama a `update`.

`setLayout` Método que asocia un configurador a un contenedor.

`show` Método que hace visible una componente.

`TextArea` Componente que presenta al usuario varias líneas de texto.

`TextField` Componente que presenta al usuario una sola línea de texto.

`update` Método llamado por `repaint`. Por defecto, borra la componente y llama después a `paint`.

`Window` Ventana de alto nivel sin borde.

Errores comunes

1. Para mostrar un objeto `String` de *N* caracteres, necesitamos una componente `TextField` de más de *N* columnas. Esto es consecuencia de un error del AWT.
2. Es un error habitual olvidarse de definir un configurador. Si es así, se utilizará uno por defecto, aunque puede que éste no sea el que deseábamos.

3. El configurador debe aparecer antes de las llamadas a `add`.
4. Es un error habitual aplicar `add` o asociar un configurador al contenedor erróneo. Por ejemplo, en un contenedor que contiene paneles, aplicar el método `add` sin especificar el panel, hará que se aplique al contenedor principal.
5. Si le falta un argumento de tipo `String` al método `add` para el configurador `BorderLayout`, se usa `"Center"` por defecto. Un error habitual es especificarlo de forma incorrecta, como por ejemplo escribiendo `"north"`. Los cinco argumentos válidos son `"North"`, `"South"`, `"East"`, `"West"` y `"Center"`. En Java 1.1, si el `String` es el segundo parámetro, una excepción en tiempo de ejecución capturará el error. Si usamos el estilo antiguo, donde el `String` aparece primero, puede que no se detecte el error.
6. La animación requiere el uso de hebras. Sin ellas, es importante que observemos que la llamada de `repaint` a `update` no es inmediata y que a veces se retrasa hasta que se produce un evento. El código a continuación de la llamada a `repaint` podría ejecutarse antes de la ejecución real de la acción de volver a pintar. Por ello, es una buena idea que `repaint` sea la última instrucción.
7. Se necesita código especial para procesar el evento de cierre de una ventana.
8. Si falta un fichero `.class` se generará un error `ClassLoader`. Este error también se puede producir si el fichero no es accesible debido a un modo de protección incorrecto.
9. Los paquetes no funcionan en algunos navegadores.
10. Los applets se ejecutan con muchas restricciones. Es importante darse cuenta de que algunas de estas restricciones se eliminan cuando se trata de applets que pertenecen al sistema local, por lo que si un applet trabaja localmente, pero falla en Internet, puede ser porque se ha topado con una restricción.
11. Los applets que se ejecutan en el Applet Viewer a veces fallan bajo Netscape, y viceversa, debido a fallos en ambos productos.
12. No se puede cambiar el tamaño de los applets; el tamaño viene determinado por la etiqueta `<APPLET>` del fichero HTML. Es obligatoria la presencia de la especificación del tamaño.
13. Un error típico en HTML es olvidar la etiqueta de cierre correspondiente a la de apertura.

En Internet

Todo el código de este apéndice está disponible en versión original en el directorio **AppendixD**. Los nombres de los ficheros son los siguientes:

BadCircles.java	Muestra la forma incorrecta de hacer animaciones; se describe en la Figura D.15.
BasicGUI.java	La clase principal, mostrada en la Figura D.14, para la aplicación GUI usada en este capítulo.

BasicGUIApplet.java	La clase, mostrada en la Figura D.21, que ejecuta tanto el applet como la aplicación vistos en la Sección D.5.4.
BasicGUIApplet.html	Código HTML, mostrado en la Figura D.19, para el applet de la Sección D.5.4.
BorderTest.java	Ilustración simple del uso de `BorderLayout`, mostrado en la Figura D.8. Usa un `CloseableFrame`.
CircleApplet.java	Applet animado de la Figura D.22. También se proporciona un fichero HTML llamado**CircleApplet.html**.
CloseableFrame.java	Implementa la interfaz `WindowListener`, como se muestra en la Figura D.13.
FileDialogTest.java	Código para mostrar el uso de `FileDialog`, mostrado en la Figura D.3. (El fichero `ListFiles.java` de la Figura 2.11 está duplicado en este directorio.)
GoodCircles.java	Muestra el uso correcto de hacer animaciones, como se describe en la Figura D.16.
GUI.java	La clase GUI usada a lo largo de todo el capítulo y mostrada en las Figuras D.5, D.6, D.9 y D.12.
GUICanvas.java	Extensión de `Canvas` usada en el ejemplo de la GUI y mostrada en la Figura D.10.

Ejercicios

Cuestiones breves

D.1. ¿Qué es una GUI?

D.2. Escriba una lista de varias componentes que puedan utilizarse para introducir datos en una GUI.

D.3. Describa las cuatro ventanas básicas de alto nivel.

D.4. ¿Cuáles son las diferencias entre las componentes `List` y `Choice`?

D.5. ¿Para qué se usa una componente `CheckBox`?

D.6. Explique los pasos necesarios para diseñar una GUI.

D.7. Explique cómo colocan las componentes los configuradores `FlowLayout`, `BorderLayout` y `null`.

D.8. Describa los pasos necesarios para incluir una componente gráfica dentro de un `Panel`.

D.9. ¿Cuál es el comportamiento por defecto cuando se produce un evento? ¿Cómo se puede cambiar dicho comportamiento?

D.10. ¿Qué eventos genera un `ActionEvent`?

D.11. ¿Cómo se trata el evento de cierre de una ventana?

D.12. Explique las diferencias entre escribir un applet y una aplicación.

D.13. Escriba una lista de algunas de las restricciones que se aplican a los applets.

Problemas prácticos

D.14. Escriba un programa de copia de ficheros (Ejercicio 2.10) que use dos objetos `FileDialog` para obtener los nombres de los ficheros.

D.15. Añada una componente `CheckBox` a la clase `GUI` para evitar borrar los dibujos entre operaciones. Necesitará modificar `GUICanvas` en concordancia e implementar un método `update`.

D.16. Se puede escribir el método `paint` para cada componente. Muestre lo que le sucede al applet cuando se pinta un círculo en la clase `GUI`.

D.17. Trate el evento producido por la pulsación de la tecla Enter en el campo de texto correspondiente a la coordenada *y* de la clase `GUI`. Necesitará modificar `actionPerformed` y registrar la existencia de un segundo manejador de eventos.

D.18. Añada (0,0) como valor por defecto para las coordenadas de una figura en la clase `GUI`.

D.19. Añada parámetros a `BasicGUIApplet` para especificar el tamaño del dibujo. Actualmente, es de 100 × 100 pixels.

D.20. La animación del círculo ilustra un problema conocido como *parpadeo*: cuando ejecuta el programa observará un retraso resultante de la llamada a `repaint`, y la animación perderá fluidez. Una forma de reducir el parpadeo consiste en redibujar solamente la parte del dibujo que ha cambiado. (Alternativamente, `repaint` aceptaría cuatro parámetros acotando la zona en que hay que volver a dibujar.) Modifique una de las animaciones del círculo para conseguir esto.

Prácticas de programación

D.21. Escriba un programa que se pueda usar para introducir dos fechas y obtener el número de días entre ellas. Use la clase `Fecha` del Ejercicio 3.12. Su programa debe funcionar como una aplicación y como un applet.

D.22. Escriba un programa que le permita dibujar líneas en un lienzo (`canvas`) usando un ratón. Una pulsación comienza la línea; una segunda pulsación la termina. Se pueden dibujar varias líneas en el lienzo. Para ello debemos extender la clase `Canvas` y tratar los eventos del ratón implementando `MouseListener`. También necesitaremos sobrescribir `update` para evitar borrar el lienzo entre los dibujos de las líneas. Añada finalmente un botón para borrar el lienzo.

D.23. Escriba una aplicación que contenga dos objetos `GUI`. Cuando se produce una acción en uno de los objetos `GUI`, el otro guarda su estado. Al efecto, necesitamos añadir un método `copiarEstado` a la clase `GUI` que copiará los estados de todos los campos de la `GUI` y volverá a dibujar el lienzo.

D.24. Escriba un programa que tenga un único contenedor y un conjunto de diez componentes de entrada cada una de las cuales especifique una forma, un color, unas coordenadas, un tamaño y un objeto `CheckBox` que indique si la componente está activa. Tras ello dibuje la unión de las componentes de entrada en un contenedor. Represente la componente `GUI` de entrada usando una clase con funciones de acceso. El programa principal debería disponer de un vector con esas componentes de entrada más el contenedor.

Bibliografía

Una referencia rápida que lista los paquetes del AWT proporciona diversos ejemplos completos es [1].

1. D. Flanagan, *Java in a Nutshell*, 2.ª ed., O'Reilly and Associates, Sebastopol, Calif. (1997).