Curso de creación de componentes en Delphi

Objetivo del curso.

Este curso pretende ofrecer un acercamiento al diseño de componentes desde un punto de vista eminentemente práctico. En multitud de libros de programación en Delphi se trata el tema del diseño de componentes, pero solamente de una forma rápida y superficial. Cierto es que uno de los manuales de Delphi está dedicado integramente a este tema, pero, a mi modo de ver, adolece de falta de ejemplos prácticos que vayan enseñando paso a paso y de un modo gradual el desarrollo de componentes. Además determinados temas, (p.e el tema de editores de propiedades) se dejan muy en el aire.

Este curso intentará cubrir estas carencias. Este curso pretende ser sobre todo práctico: la teoría de la creación de componentes se irá viendo según se vaya necesitando. Se comenzará viendo la teoría básica de creación de componentes e inmediatamente se aplicará a la creación de componentes totalmente funcionales.

A lo largo del curso aprenderemos a crear componentes de muy diversos tipos. Comenzaremos con componentes simples visuales y no visuales e iremos progresando creando componentes gráficos, editores de propiedades, componentes de base de datos... Además se explicará como crear los archivos de ayuda para hacer que nuestro componente se integre plenamente en el entorno de desarrollo de Delphi.

Conocimientos previos.

Pero este no es un curso de programación en Delphi, ni un curso de programación orientada a objetos. Se supone que el lector tiene ya un bagaje más o menos amplio de estos temas y conceptos tales como herencia, descendencia, etc. no le resultan desconocidos.

Como requisito previo a la creación de componentes resulta altamente recomendable tener claros los siguientes conceptos:

- Qué es la programación orientada a objetos y sus aspectos fundamentales (constructores, destructores, clases, herencia, sobrecarga...)
- Dominio del entorno integrado de desarrollo de Delphi.
- Manejo con soltura de los distintos componentes estandard de Delphi.
- Utilización del ObjectBrowser para determinar relaciones entre objetos.

Si necesitas alguna aclaración sobre estos temas, puedes encontrar toda la información necesaria en los propios manuales de Delphi, así como en la ayuda en línea.

Periodicidad del curso.

En principio la periocidad del curso será semanal o quincenal, pero todo dependerá de la respuesta que encuentre de vosotros, los usuarios del curso así como de mi disponibilidad a lo largo del tiempo (ya sabéis, exámenes y esas cosas).

Consultas, dudas, sugerencias...

Este curso no estaría completo sin tu colaboración, así que espero tus criticas, sugerencias, propuestas de componentes, ejemplos prácticos, etc. en la siguiente direción: revueltaroche@redestb.es

Además, si has desarrollado algún componente que creas que puede ser interesante, estás invitado a compartirlo con todos los lectores del curso.

De modo que no lo olvideis: el que este curso sea interesante y práctico depende de vosotros también...

Unidad 2. Un poco de teoría

¿Qué son componentes? La librería visual de componentes (VCL).

Los componentes son las piedra angular de la programación en Delphi. Aunque la mayoría de los componentes representan partes visibles de la interfaz de usuario, existen también componentes no visuales, como por el ejemplo los objetos Timer y Database.

Un componente, en su definición más simple no es más que un objeto descendiente del tipo TComponent. Todos los componentes descienden en su forma más primitiva de TComponent, ya que TComponent proporciona las características básicas que todo componente debe tener: capacidad de ser mostrado en la paleta de componentes así como de operar a nivel de diseño de formulario.

Los componentes hacen fácil la programación en Delphi. En vez de tener que operar a nivel de unidades, el usuario de un componente simplemente tiene que pinchar en él y situarlo en la posición deseada de su form. Eso es todo: Delphi se encarga de lo demás.

Todos los componentes forman parte de la jerarquía de objetos denominada Visual Component Library (VCL). Cuando se crea un nuevo componente, este se deriva a partir de un componente existente (bien sea TComponent o algún otro más especializado) y se añade a la VCL.

Anatomía de un componente. Propiedades, métodos y eventos.

Como ya se ha mencionado un componente es un objeto, y como tal, consta de código y datos. Pero al referirnos a estos no utilizaremos estos terminos, sino que hablaremos de propiedades y métodos, así como de eventos. A lo largo de este curso iremos estudiando en profundidad todos estos aspectos, pero hagamos ahora una primera aproximación:

- Propiedades

  Las propiedades proporcionan al usuario del componente un fácil acceso al mismo. Al mismo tiempo, permite al programador del componente "esconder" la estructura de datos subyacente. Entre las ventajas de utilizar propiedades para acceder al componente se pueden citar:

  - Las propiedades están disponibles en tiempo de diseño. De este modo el usuario del componente puede inicializar los valores de las propiedades sin necesidad de escribir una sóla línea de código.
  - Las propiedades permiten la validación de los datos al mismo tiempo de ser introducidas. Así se pueden prevenir errores causados por valores inválidos.
  - Nos aseguran que desde el primer momento nuestras propiedades tendrán un valor válido, evitando el error común de hacer referencia a una variable que no ha sido convenientemente inicializada.

- Eventos

  Los eventos son las conexiones existentes entre un determinado suceso y el código escrito por el programador de componentes. Asi por ejemplo, ante el suceso clic del ratón se podría mostrar un mensaje en pantalla. Al código que se ejecuta cuando se produce un determinado evento se le denomina manejador de eventos (event handler) y normalmente es el propio usuario del componente quién lo escribe. Los eventos má comúnes ya forman parte de los propios componentes de Delphi (acciones del ratón, pulsaciones de teclado...), pero es también posible definir nuevos eventos.

- Métodos

  Los métodos son los procedimientos y/o funciones que forman parte del componente. El usuario del componente los utiliza para realizar una determinada acción o para obtener un valor determinado al que no se puede acceder por medio de una propiedad. Ya que requieren ejecución de código, los métodos sólo están disponibles en tiempo de ejecución.

Control de acceso a un componente. Declaraciones privadas, protegidas, públicas y publicadas.

Object Pacal dispone de cuatro niveles de control de acceso para los campos, propiedades y métodos de un componente. Este control de acceso permite especificar al programador de componentes que parte de código puede acceder a que partes del objeto. De este modo se define el interface del componente. Es importante planear bien este interface, ya que así nuestros componentes serán facilmente programables y reutilizables.

A menos que se especifique lo contrario, los campos, propiedades y métodos que se añaden a un objeto son de tipo publicados (published). Todos los niveles de control de acceso operan a nivel de unidades, es decir, si una parte de un objeto es accesible (o inaccesible) en una parte de una unidad, es también accesible (o inaccesible) en cualquier otra parte de la unidad.

A continuación se detallan los tipos de controles de acceso:

Privado: ocultando los detalles de implementación.

Declarando una parte de un componente (bien sea un campo, propiedad o método) privado (**private**) provoca que esa parte del objeto sea invisible al código externo a la unidad en la cuál se declara el objeto. Dentro de la unidad que contiene la declaración, el código puede acceder a esa parte del objeto como si fuera público.

La principal utilidad de las declaraciones privadas es que permiten ocultar los detalles de implementación del componente al usuario final del mismo, ya que estos no pueden acceder a la parte privada de un objeto. De este modo se puede cambiar la implementación interna del objeto sin afectar al código que haya escrito el usuario.

Protegido: definiendo el interface del programador.

Declarar una parte de un componente como protegido (**protected**) provoca, al igual que ocurría al declararlo privado, que el código externo a la unidad no pueda acceder a dicha parte (se hace oculta al código externo a la unidad). La diferencia principal entre declarar una parte de un objeto protegida o hacerla privada es que los descendientes del componente podrán hacer referencia a esa parte.

Este comportamiento es especialmente útil para la cración de componentes que vayan a descender de aquel que hemos creado.

Público: definiendo el interface en tiempo de ejecución.

Todo las partes de un objeto que declaremos públicas (**public**) podrán ser referenciadas por cualquier código ya sea interno o externo a la propia unidad. En este sentido, la parte pública identifica el interface en tiempo de ejecución de nuestro componente. Los métodos que el usuario del componente debe llamar deben ser declarados publicos, así como también las propiedades de sólo lectura, al ser sólo válidas en tiempo de ejecución.

Las propiedades declaradas públicas no aparecerán en el inspector de objetos.

Esta sección es tal vez la más importante a considerar al diseñar un componente. Cuando se diseñan componentes, se debe considerar cuidadosamente que métodos y propiedades deben ser públicas. Si el diseño es correcto, este permitira retocar las estructuras de datos y métodos internos del componente sin tener que tocar el interface público, que seguirá siendo el mismo para el usuario del componente.

Publicado: definiendo el interface en tiempo de diseño.

Al declarar parte de un objeto publicado (**published**) provoca que la parte sea pública y además genera información en tiempo de ejecución para dicha parte.

Las propiedades declaradas publicadas aparecen en el inspector de objetos en tiempo de diseño. Y ya que sólo las partes publicadas aparecen en el inspector de objetos, estas partes definen el interface en tiempo de diseño de nuestro componente. En general sólo se deben declarar publicadas propiedades y no funciones o procedimientos (ya que lo único que logramos con ello es declararlas públicas).

Pasos necesarios para crear un componente. El experto de componentes.

A grandes rasgos, los pasos necesarios para crear un nuevo componente son los siguientes:

1. Crear una unidad para el nuevo componente.
2. Derivar el nuevo componente a partir de otro existente, el cuál servirá de base para añadir las nuevas características deseadas.
3. Añadir las propiedades, eventos y métodos necesarios al nuevo componente.
4. Registrar el componente, incluyendo los bitmaps adicionales, ficheros de ayuda, etc.
5. Instalar el nuevo componente en la paleta de componentes.

De todos los pasos citados, hay uno que es especialmente importente: la elección del ascendiente a partir del cuál derivar el nuevo componente. Este paso es crucial, ya que una buena elección del ascendiente puede hacernos la tarea de añadir nuevas propiedades y métodos realmente fácil, mientras que una mala elección puede hacernos imposible llegar al objetivo propuesto.

Como base para la elección del ascendiente, conviene hacer notar las siguientes normas:

- TComponent - El punto de partida para los componentes no visuales.
- TWinControl - El punto de partida si es necesario que el componente disponga de handles.
- TGraphicControl - Un buen punto de partida para componentes visuales que no sea necesario que dispongan de handles, es decir, que no reciban el foco. Esta clase dispone del método Paint y de Canvas.
- TCustomControl - El punto de partida más común. Esta clase dispone de window handle, eventos y propiedades comúnes y, principalmente, canvas con el método Paint.

Bien, ya sabemos como determinar el punto de partida. Veamos ahora como crear la unidad que albergará el componente. Hay dos opciones, crear la unidad manualmente o dejar que Delphi haga el trabajo "sucio" utilizando el experto de componentes. Si optamos por la primera solución, basta con hacer clic en new unit y ponernos manos a la obra, pero de este modo tendremos que hacer todo a mano: derivar el nuevo componente, registrarlo, etc. Por ello es más recomendable la segunda opción: utilizar el *experto de componentes*.

Para abrir el experto de componentes basta con elegir File|New Component.

Nos aparecerá un cuadro de diálogo en el que debemos cumplimentar los siguientes campos:

- Class Name: Aquí debemos especificar el nombre del nuevo componente.
- Ancestor type: Introduciremos aquí el ascendiente a partir del cuál derivaremos nuestro componente.
- Palette Page: Indicaremos aquí la página de la paleta en la cuál queremos que aparezca el nuevo componente.

Una vez introducidos estos campos, al pulsar sobre OK se nos desplegará el código de nuestra unidad. Si por ejemplo hemos introducido los siguientes datos en el experto de componentes:

Class Name: TMiComponente
Ancestor Type: TComponent
Palette Page: Curso

Al hacer clic en aceptar, Delphi nos generaría la siguiente unidad, lista para introducir las propiedades y métodos de nuestro componente:

```
unit Unit1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TMiComponente = class(TComponent)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Curso', [TMiComponente]);
end;

end.
```

A partir de aquí todo consiste en introducir las propiedades y métodos necesarios para el funcionamiento de nuestro componente. Pero antes de nada conviene hacer notar algunos aspectos:

En la clausula uses, Delphi añade por defecto las unidades estandard. Si nuestro componente no usa alguna de ellas podemos eliminarla de dicha clausula. Del mismo modo, si utilizamos algún procedimiento o función situado en otra unidad, debemos añadir dicha unidad a la clausula uses. Vamos, como siempre ¿no? ;)

Las declaraciones de las propiedades, campos y métodos que vayamos definiendo, las situaremos en la sección apropiada de interfaz según corresponda, es decir las declararemos privadas, protegidas, pública o publicadas.

Delphi declara y define automaticamente el procedimiento register para registrar el componente en la paleta.

Unidad 3. Un componente no visual: TNif

Ha llegado el momento de pasar a la acción. Vamos a crear nuestro primer componente: TNif. Pero antes de ponernos a escribir código conviene analizar el propósito del componente y cómo lo vamos a implementar.

Objetivo del componente

El propósito del componente es muy simple: a partir de un dato de entrada, el número de DNI, nuestro componente debe calcular la letra del NIF correspondiente. El método de calcular esta letra es mediante una operación matemática muy sencilla. Esta operación es: DNI - ((DNI div 23) * 23). El resultado de esta operación será un número comprendido entre 1 y 23. Mediante una tabla asignaremos una letra determinada a cada número, letra que corresponde al NIF pedido. Los detalles completos de este calculo y asignación de letra se encuentran en el código fuente.

Diseño del componente

En base a nuestro objetivo, queda claro que nuestro componente será del tipo no visual y, por lo tanto, lo derivaremos a partir de TComponent, que como ya se ha visto; (unidad 2), es la base para la creación de componentes no visuales.

La forma de introducir el número de DNI será mediante una propiedad (propiedad DNI) de lectura y escritura. El valor de dicha propiedad lo almacenaremos en un campo (FDNI) de tipo longInt. Esta propiedad será publicada (published) para que así aparezca en el inspector de objetos. La lectura y escritura de valores en esta propiedad la haremos directamente sobre el campo FDNI, es decir dejaremos que sea el propio inspector de objetos el que se encargue de verificar que el valor introducido corresponde al tipo longInt.

Tendremos otra propiedad (NIF) en la que se almacenará la letra calculada (de tipo char). Pero esta propiedad será de sólo lectura ya que el usuario del componente no debe

poder introducir la letra manualmente, ya que debe ser el propio componente el que la calcule. Esta propiedad **debe** de ser pública (public) ya que es de sólo lectura. Este aspecto conviene resaltarlo: *las propiedades de sólo lectura deben ir declaradas en la parte pública.* La función que se encarga de calcular la letra del NIF la llamaremos GetNIF.

Los campos que almacenan el valor de propiedades siempre se declararán en la parte privada (private) ya que así nos aseguramos de que el componente que declara la propiedad tiene acceso a ellos, pero el usuario del componente no, ya que él debe acceder a través de la propiedad y no del campo (que representa el almacenamiento interno de la propiedad).

Respecto a los nombres empleados, se siguen las siguientes convenciones:

- Los tipos que definamos comenzarán con la letra T (de tipo). P.e. TNif
- Los campos que almacenan los valores de propiedades comienzan con la letra F seguida del nombre de la propiedad que almacenan. Así el campo FDNI queda claro que almacena el valor de la propiedad DNI.
- Los nombres de los métodos de lectura y escritura de los valores de una propiedad se denominarán mediente el prefijo Get (para lectura) o Set (para escritura) seguidos del nombre de la propiedad. P.e. el método GetNIF.

Una vez acordado el método de diseño empleado es hora de comenzar a teclear.

Código fuente del componente

```
unit Nif;          { (c)1996 by Luis Roche }

interface

uses
  Classes;

type
  TNif = class(TComponent)   {Nuestro propiedad deriva de TComponent}
  private
    FDNI : LongInt;           {Almacenará el número de DNI}
    function GetNIF : char;   {Calcula la letra del NIF}
  protected
  public
    property NIF: char read GetNIF;    {Propiedad NIF: sólo lectura}
  published
     property DNI: LongInt read FDNI write FDNI;  {DNI: lectura y
escritura}
  end;

procedure Register;       {Registra nuestro componente en la paleta}

implementation

function TNIF.GetNIF : char;      {Calcula el NIF a partir del DNI}
Var aux1 : integer;
Const letras : string = 'TRWAGMYFPDXBNJZSQVHLCKE';
begin
  aux1:=FDNI - ((FDNI div 23) * 23);
  result:=letras[aux1+1];
```

```
end;

procedure register;           {registro del componente}
begin
  registercomponents('curso', [tnif]);
end;

end.
```

Comentarios al código fuente

El código fuente se ha creado siguiendo los siguientes pasos:

- Utilizamos el experto de componentes para que nos generere la unidad en la que escribiremos nuestro componente. En el cuadro de diálogo que el experto nos muestra, introducimos TNif como Class Name, TComponent como Ancestor Type y Curso como Palette page. Al pulsar sobre OK, el experto de componentes nos crea el esqueleto básico de nuestro componente, incluyendo el procedimiento Register.
- De todas las unidades que aparecen en la clausula uses dejamos sólo Classes, ya que no utilizamos ningún procedimiento de las demás.
- Declaramos el campo FDNI (longInt) en la sección privada. En este campo, como ya se ha dicho, almacenaremos el DNI. En la sección published escribimos la siguiente línea:
  property DNI: LongInt read FDNI write FDNI
  De este modo declaramos la propiedad DNI y especificamos que la lectura y escritura de valores en la misma se hace directamente con el campo FDNI utilizando el inspector de objetos.
- Declaramos la propiedad NIF de sólo lectura en la parte public. Conviene recordar que las propiedades de sólo lectura deben ir declaradas en la parte pública y no en la publicada. Especificamos que para leer el valor de la propiedad utilizaremos la función GetNIF, la cuál declaramos en la sección privada.
- Escribimos la función que cálcula la letra del NIF en la parte de implementación de la unidad.
- Guardamos la unidad con el nombre nif.pas. Conviene que todos los componentes que vayamos creando durante el curso los almacenemos en un directorio aparte (p.e delphi\componen)

Creando un bitmap para el componente

Cada componente necesita un bitmap para representar al componente en la paleta de componentes. Si no se especifica uno, Delphi utilizará uno por defecto.

El bitmap no se incluye en el código fuente del componente, sino que debe incluirse en un archivo aparte con la extensión .DCR (dynamic component resource). Este fichero puede crearse con el propio editor de imagenes que incorpora Delphi.

El nombre del archivo .DCR debe coincidir con el nombre con que se ha salvado la unidad que contiene el componente. El nombre de la imagen bitmap (que debe estar en

mayúsculas) debe coincidir con el nombre del componente. Los dos ficheros (el de la unidad *.pas y el de el bitmap *.dcr) deben residir en el mismo directorio.

En nuestro componente, si hemos salvado la unidad con el nombre nif.pas nuestro archivo de recursos deberá tener el nombre nif.dcr. Dentro de este archivo se encontrará el bitmap, al que pondremos el nombre TNIF. El bitmap que hemos creado es el siguiente: 

El tamaño del bitmap debe ser de 24x24 pixels.

Como último detalle, si quieres utilizar este mismo bitmap, puedes utilizar un programa de tratamiento de imágenes para cortarlo y pegarlo en el editor de imágenes de Delphi.

Instalando TNif en la paleta de componentes.

Instalar nuestro componente en la paleta de componentes es muy sencillo. Basta con seguir los siguientes pasos:

- Elegir la opción Options|Install Components desde el menú de Delphi.
- Pulsar sobre el botón Add para seleccionar la unidad que contiene el código fuente del componente. En nuestro caso, nif.pas. Una vez seleccionada la unidad, hacer clic sobre OK.
- Delphi compilará la unidad y reconstruirá el archivo COMPLIB.DCL. Si se encuentran errores al compilar, se nos informará de ello. Basta con corregir el error, salvar la unidad y repetir el proceso anterior.
- Si la compilación tiene éxito, nuestro componente entrará a formar parte de la paleta de componentes.

Nota: Es una buena idea antes de comenzar hacer una copia de seguridad del archivo COMPLIB.DCL para evitar posibles problemas derivados de fallos al compilar, caídas de tensión, etc.

Unidad 4. Añadiendo posibilidades a un componente existente: TBlinkLabel


En esta unidad aprenderemos a añadir posibilidades a un componente Delphi ya existente. Conoceremos cómo utilizar objetos (componentes) en nuestro propio componente, y qué son y cómo se declaran los constructores y destructores.

Todo esto, claro está, en base a un ejemplo concreto: TBlinkLabel.

Objetivo del componente

Nuestro propósito es crear un componente idéntico al tipo TLabel (etiqueta), pero con una capacidad añadida: el parpadeo. Es decir, nuestro componente nos debe mostrar el mensaje introducido en la propiedad caption parpadeando con una determinada frecuencia. Además, si la frecuencia de parpadeo introducida es nula, el mensaje debe permanecer fijo en la pantalla (emulando lo que sucede con una etiqueta normal).

Utilizando un componente existente en nuestro propio componente. Timers.

Como acabamos de decir, nuestro componente debe parpadear con una cierta frecuencia. Para ello necesitamos poder medir el tiempo que debe transcurrir entre el encendido y apagado del mensaje. Afortunadamente, Delphi nos proporciona un objeto que se amolda perfectamente a nuestros requisitos de medición de tiempo: TTimer.

Aunque no es este el lugar donde explicar las características del objeto Timer (la información básica de este componente se puede encontrar en la propia ayuda en línea de Delphi), conviene detenernos un instante en la propiedad del timer que vamos a utilizar: la propiedad *interval*. Esta propiedad nos permite especificar el intervalo de tiempo que queremos medir (en milisegundos). Cuando se alcanza el valor establecido se ejecuta el método especificado en el evento OnTimer.

El método que nosotros especificaremos para el evento OnTimer se encargará de visualizar y ocultar alternativamente el mensaje.

La teoría está muy bien, pero... ¿cómo añadimos el objeto timer a nuestro componente? En tiempo de diseño es sencillo. Basta con seleccionar el timer en la paleta de componentes y situarlo en nuestro form.

Pero al escribir un componente no podemos utilizar este sistema, si no que debemos hacerlo a mano. Los pasos generales que se deben seguir son los siguientes:

- Añadir la unidad en que se define el componente a la claúsula uses de nuestra unidad. En nuestro caso, como el objeto timer se declara en la unidad ExtCtrls, es esta unidad la que debemos añadir a la claúsula uses.
- Añadir al método create de nuestro componente el código necesario para construir el objeto (en nuestro caso, el timer).
- Escribir el código necesario para destruir el objeto en el método destroy de nuestro componente.

En la siguiente sección profundizaremos en estos dos últimos pasos.

Constructores y destructores.

Constructores

Los objetos que declaramos en un componente no existen en memoria hasta que el objeto es creado (también se dice que se crea una instancia del objeto) por una llamada al método constructor del objeto.

Un constructor no es más que un método que proporciona memoria para el objeto y apunta (mediante un puntero) hacia él. Llamando al método **create**, el constructor asigna la instancia del objeto a una variable.

Conviene hacer notar que todos los componentes heredan un método denominado create que se encarga de "crear" (en el sentido que acabamos de ver) el componente en memoria. De forma adicional, en el método create se pueden inicializar los valores de determinados campos del componente (comúnmente asignarles un valor por defecto)

En el método create del componente podemos crear objetos adicionales que necesite nuestro componente. En nuestro caso, en el método create de TBlinkLbl crearemos el timer.

Destructores

Cuando terminamos de utilizar un objeto, debemos destruirlo, es decir, liberar la memoria que ocupaba el objeto. Esta operación se realiza mediante el método **destroy** que todos los componentes heredan de TComponent.

Si hemos creado algún objeto adicional, también debemos destruirlo escribiendo el código necesario en el método destroy.

Delphi crea y destruye automaticamente nuestro componente cuando es necesario, ya que como se ha dicho, los métodos create y destroy se heredan de TComponent.

Pero si queremos utilizar algún objeto en nuestro componente, Delphi no lo crea y destruye automaticamente, sino que debemos hacerlo nosotros de forma manual. Este proceso se realiza añadiendo el código necesario a los métodos create y destroy. Es importante notar que nos se escriben los métodos de nuevo, sino que se sobrecargan con el nuevo código.

De una forma general esto se hace de la siguiente forma: (un ejemplo más concreto lo tenemos en el propio código fuente del componente)

- En la parte pública de nuestro componente declaramos los métodos create y destroy seguidos de la palabra clave overrride:

```
public
    constructor create(AOwner : TComponent); override;
    destructor   destroy; override;
```

- Escribir el código adicional del constructor y del destructor en la sección de implementación.

```
constructor TComponent.Create(AOwner : TComponent);
begin
     inherited Create(AOwner);
     ...
end;

destructor TComponent.Destroy;
begin
     ...
     inherited destroy;
end;
```

Lo más importante a tener en cuenta es que al sobrecargar un constructor, lo primero que se debe hacer es hacer una llamada al constructor original (línea inherited Create). A continuación se puede añadir el código necesario para crear el objeto.

De un modo similar, al sobrecargar un destructor, primero se debe liberar el objeto que hayamos creado anteriormente y la última línea debe de ser una llamada al destructor original.

Implememententando el parpadeo.

En nuestro proceso de diseño del componente nos queda aún un paso importante: cómo hacer que el mensaje parpadee. Hemos dicho que cuando se produzca el evento OnTimer se ejecutará el método parpadea que se encargará de visualizar y ocultar el mensaje, es decir, de producir el parpadeo.

La solución es muy simple, ya que nuestro componente, al descender de TLabel incorporá una propiedad que determina si el componente debe estar visible o no. Esta propiedad se denomina **visible**.

De este modo, nuestro método parpadea lo único que hará será alternar el valor de la propiedad booleana visible según se produzca el evento OnTimer. Los detalles concretos de implementación se muestran en el código fuente.

Otros detalles en el diseño del componente. Valores por defecto.

Para que el diseño del componente este finalizado nos quedan por ver dos aspectos:

- Hemos comentado que si el valor introducido como frecuencia de parpadeo es nulo, el mensaje debe permanecer fijo en pantalla. Esto se logra activando y desactivando el componente cuando se introduce el valor de la propiedad velocidad en el inspector de objetos (ver el procedimiento SetVelocidad en el código fuente).
  De este modo, si se introduce 0 como valor de la velocidad, simplemente se deshabilita el timer y se pone el valor de la propiedad visible a True. Si el valor introducido es distinto de cero, se habilita el timer y, por consiguiente, el parpadeo.

- Un último detalle es dar un valor por defecto a la propiedad velocidad de parpadeo. Esto se consigue en dos pasos: primeramente en la declaración de la propiedad velocidad se añade la palabra clave **default** seguida del valor por defecto (400 en nuestro caso). A continuación, en el constructor se inicializa el campo asociado con la propiedad (FVelocidad:=400). Los **dos pasos son necesarios**, no pudiendose obviar ninguno de los dos.
  Este detalle de asignar valores por defecto es más importante de lo que parece a simple vista, ya que de este modo nos evitamos errores de inicialización,

asegurándonos que la propiedad siempre tendrá (incluso desde el principio) un valor válido. Adicionalmente, en los forms en que se utilice el componente, Delphi sólo guardará el valor de la propiedad si difiere de su valor por defecto.

Código fuente del componente

```
unit Blinklbl;      { (c)1996 by Luis Roche }

interface

uses
  Classes, StdCtrls, ExtCtrls;    { TTimer se declara en ExtCtrls }

type
  TBlinkLabel = class(TLabel)    {TBlinkLabel deriva de TLabel}
  private
    FVelocidad : integer;        {Frecuencia de parpadeo}
    FTimer : TTimer;             {Timer para la frecuencia}
    procedure SetVelocidad(valor : integer);  {Almacena la velocidad}
  protected
    procedure parpadea(Sender : TObject);
  public
    constructor Create(AOwner : TComponent); override;
{Constructor}
    destructor Destroy; override;                        {Destructor}
  published
    property Velocidad : integer read FVelocidad write SetVelocidad
default 400;
  end;

procedure Register;

implementation

constructor TBlinkLabel.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);       {Llama al constructor original
(heredado)}
  FTimer := TTimer.Create(Self);  {Creamos el timer}
  FVelocidad := 400;              {Frecuencia (velocidad) por defecto}
  FTimer.Enabled:=True;           {Activamos el timer}
  FTimer.OnTimer:=parpadea;       {Asiganamos el método parpadea}
  FTimer.Interval:=FVelocidad;    {Asignamos el intervalo del timer =
frecuencia parpadeo}
end;

destructor TBlinkLabel.Destroy;
begin
  FTimer.Free;          {Liberamos el timer}
  inherited destroy;    {Llamamos al destructor original (heredado)}
end;

procedure TBlinkLabel.SetVelocidad (valor : integer);
begin
  If FVelocidad <> valor then        {Sólo si el valor introducido es
distinto del almacenado}
  begin
    if valor < 0 then FVelocidad:=0;
    FVelocidad:=Valor;               {Asigna la velocidad}
```

```
    if FVelocidad=0 then FTimer.Enabled:=False else
FTimer.Enabled:=True;
    {Si Fvelocidad=0 el mensaje debe estar siempre visible}
    FTimer.Interval:=FVelocidad;
    Visible:=True;
  end;
end;

procedure TBlinkLabel.parpadea(Sender : TObject);
begin
  if FTimer.Enabled then Visible := not(Visible);  {Alternativamente
muestra y oculta el mensaje si el timer esta activado}
end;

procedure Register;        {Registro del componente}
begin
  RegisterComponents('Curso', [TBlinkLabel]);
end;

end.
```

Unidad 5. El objeto Canvas y el método Paint: TGradiente

 En esta unidad crearemos un componente gráfico del tipo que se utiliza como fondo en la instalación de aplicaciones: TGradiente. Pero nuestro componente irá más allá y añadirá nuevas posibilidades al gráfico estandard. Aprenderemos a dibujar sobre el canvas que disponen todos los componentes gráficos. Estudiaremos el método paint y cómo utilizarlo en nuestros componentes.

Objetivo del componente

Nuestro propósito es crear un componente en el que dibujaremos un gradiente de colores similar al que se muestra en la mayoría de los programas de instalación de aplicaciones (y en otro tipo de programas). Entre las características adicionales que nuestro componente debe tener cabe destacar que el gradiente de colores no finalizará en el color negro, sino que será capaz de comenzar y terminar en dos colores cualesquiera. Además, el gradiente podrá ser horizontal o vertical. Por último, el tamaño de nuestro dibujo (es decir, del canvas del componente) será variable a voluntad, para de este modo poder combinar dos o más componentes para crear efectos de colores espectaculares.

Nuestro componente derivará de TGraphicControl ya que necesitamos que disponga de Canvas, pero no es necesario que disponga de manejador (handler). Esta elección la hacemos en base a lo que se vio en la unidad 2.

La base de la programación gráfica en Delphi: el objeto Canvas.

Manual de Creación de Componentes en Delphi
MBS para el LTIASI

Tradicionalmente, la programación gráfica en windows se ha realizado mediante el uso del interface gráfico de dispositivo (GDI). Este interface es una herramienta poderosa que permite el dibujo de gráficos mediante el uso de pinceles, brochas, rectángulos, elipses, etc.

Pero el GDI tiene un inconveniente: su programación es muy laboriosa. Cuando se va a hacer uso de una fución GDI, se necesita un handle a un contexto dispositivo, asi como crear y destruir las diversas herramientas de dibujo (recursos) que se utilicen. Pr último, al finalizar, se debe restaurar el contexto de dispositivo a su estado original antes de destruirlo.

Delphi encapsula el GDI, haciendo que no nos tengamos que preocupar de contextos de dispositivo ni de si hemos liberado o no los recursos utilizados. De este modo nos podemos centrar en lo que es nuestro objetivo principal: dibujar los gráficos. De todas formas, si lo deseamos, podemos seguir utilizando las funciones GDI si nos interesa. Así tenemos un doble abanico de posibilidades, Delphi o GDI, que utilizaremos según nos convenga.

Como se ha mencionado en el párrafo anterior, Delphi nos proporciona un completo interface gráfico. El objeto principal de este interface es el objeto **Canvas**. El objeto Canvas se encarga de tener un contexto de dispositivo válido, así como de liberarlo cuando no lo utilicemos. Además, el objeto Canvas (o simplemente Canvas) dispone de diversas propiedades que representan el lapiz actual (pen), brocha (brush) y fuente (font).

El canvas maneja todos estos recursos por nosotros, por lo cual nos basta con informarle de que clase de pen queremos manejar y él se encarga del resto. Además, dejando que Delphi se encargue de crear y liberar los recursos gráficos, en muchos casos obtendremos un aumento de velocidad frente a si los manejáramos nosotros mismos.

El objeto canvas encapsula la programación gráfica a tres niveles de profundidad, que son los siguientes:

Nivel	**Operación**	**Herramientas**
Alto	Dibujo de líneas y formas Visualización y modificación de texto Relleno de áreas	Métodos MoveTo, LineTo, Rectangle, Ellipse Métodos TextOut, TextHeight, TextWidth, TextRect Métodos FillRect, FloodFill
Intermedio	Personalizar texto y gráficos Manipulación pixels Copia y unión de imágenes	Propiedades Pen, Brush y Font Propiedad Pixels Draw, StretchDraw, BrushCopy, CopyRect, CopyMode
Bajo	Llamadas a funciones GDI	Propiedad Handle

Ahora no nos vamos a centrar en explicar todos y cada uno de las herramientas disponibles para la creación de gráficos, sino que lo haremos según vayamos

utilizándolas en sucesivas unidades. De todas formas, una descripción detallada de cada una de ellas se encuentra en la ayuda en línea de delphi

Bien, ya conocemos que el objeto canvas nos proporciona un interface gráfico y poderoso de creación gráfica. Pero ahora se nos pueden plantear las siguientes preguntas: ¿Donde reside el objeto canvas?, ¿todos los componentes lo tienen?, y si no es así, ¿qué componentes lo incorporan?

La respuesta a estas preguntas es sencilla: **Todos los objetos derivados de TGraphicComponent poseen Canvas**. Respecto a otros componentes, depende. En caso de duda, lo más sencillo es consultar el Object Browser y verificar si el componente en cuestión posee o no canvas (bien sea declarado de forma protegida o pública). Así por ejemplo, los componentes Tlabel, Timage y TPanel lo poseen, mientras que componentes tales como TButton o TRadioButton no. Este aspecto determinará en gran medida el ancestor que hemos de elegir cuando queramos desarrollar un componente que deba disponer de Canvas.

La forma de acceder al canvas de un objeto es muy sencilla. Supongamos que tenemos un componente (TGradiente) derivado de la clase TGraphicControl. Y supongamos que queremos dibujar en el canvas una línea desde el punto (0,0) hasta el punto (20,20). Para realizar esta tarea deberiamos escribir la siguiente sentencia:

```
TGradiente.Canvas.MoveTo(0,0);
TGradiente.Canvas.LineTo(20,20);
```

Centremonos ahora en las propiedades del canvas que utilizaremos en el desarrollo de nuestro componente. Básicamente TGradiente debe dibujar una serie de rectángulos coloreados con un determinado color. Para ello utilizaremos el método **FillRect**. FillRect recibe como parámetro un objeto del tipo TRect con las coordenadas superior izquierda e inferior derecha del recuadro a pintar. De esta forma, el código que utilizaremos en nuestro componente será similar al siguiente (aún no tenemos en cuenta el color de relleno)

```
Canvas.FillRect(TRect);
```

Para asignar el color de relleno utilizaremos la propiedad **brush**. El objeto brush determina determina el color y patrón que el canvas utiliza para rellenar formas gráficas y fondos. En nuestro caso, utilizaremos la propiedad color del brush para asignar el color de relleno del canvas, de modo que una posterior llamada a FillRect utilice dicho color asignado.

Tambien necesitaremos utilizar la propiedad **pen** del canvas. El objeto pen determina que clase de lapiz utilizará el canvas para dibujar líneas, puntos y recuadros gráficos. En nuestro caso, utilizaremos la propiedad style del pen que determina que tipo de línea dibujar (en nuestro componente psSolid) y la propiedad mode que determina el modo con que el lapiz dibujará sobre el canvas (en nuestro componente pmCopy)

Creando el gradiente de colores.

En Delphi, un color se representa por 4 bytes hexadecimales. El byte más alto se utiliza para determinar el ajuste que Windows hace de la paleta y no vamos a verlo con más detalle. Los otros tres bytes (los tres bytes más bajos) representan la cantidad de rojo, verde y azul que forman el color. Nuestro componente debe calcular un gradiente de colores desde uno inicial (FColorDesde en el código fuente) hasta otro final (FColorHasta). La mejor manera de calcular este gradiente es descomponiendo dichos colores en sus componentes RGB (Rojo, Verde y Azul). De este modo, sabremos la cantidad de cada uno de estos tres colores básicos que forman un color determinado. Por ejemplo el color rojo puro se representa por 255,0,0 (255 de rojo, 0 de verde y azul), y un tono gris tiene los tres valores de rojo, verde y azul iguales (p.e. 150,150,150).

La descomposición de un color en sus tres componentes básicos la realizan tres funciones: *GetRValue, GetGValue y GetBValue* pertenecientes al API de Windows. Estas funciones toman como parámetro el color del que deseamos obtener la descomposición y devuelven respectivamente la "cantidad" de rojo, verde y azul que componen dicho color.

En el código fuente, una vez descompuestos los colores inicial y final en las variables RGBDesde[0..2] y RGBHasta[0..2] (0 para color rojo, 1 para verde y 2 para azul), el proceso de calculo del gradiente es el siguiente:

1. Calcular la diferencia en valor absoluto entre RGBDesde y RGBHasta para cada uno de los colores básicos y guardarla en la variable RGBDif[0..2]. Adicionalmente, en la variable factor guardaremos un +1 si RGBHasta es mayor que RGBDesde (gradiente ascendente) y un -1 en caso contrario (gradiente descendente). Este proceso se realiza para cada uno de los tres colores básicos.
2. Mediante un bucle que varía desde 0 a 255 (nuestro gradiente constará de 256 colores), calculamos el color que corresponde a cada recuadro a dibujar (con el método FillRect) mediante la expresión:

   ```
   Rojo:=RGBDesde[0]+factor[0]*MulDiv(contador,RGBDif[0],255);
   ```

   (de forma análoga para el verde y el azul)
   Nota: MulDiv es una función del Api de Windows que multiplica los dos primeros valores pasados como parámetros y el resultado se divide por el tercer parámetro, devolviendo el valor de esta división en forma de 16 bits redondeado.

3. Asignamos el color calculado a la propiedad color del objeto brush:

   ```
   Canvas.Brush.Color:=RGB(Rojo,Verde,Azul);
   ```

   Como es fácil suponer, la función RGB toma tres valores de rojo, verde y azul, y forma el color correspondiente a dichos colores básicos.

4. Se dibuja el recuadro relleno:

   ```
   Canvas.FillRect(Banda);
   ```

Banda es una variable de tipo TRect que guarda las coordenadas del recuadro a dibujar.

5. Se prosigue con otra iteración del bucle.

Un último detalle: cómo se ha mencionado, nuestro gradiente constará de 256 colores. Sin embargo, dependiendo del modo gráfico en que tengamos configurado Windows y de la existencia de otros objetos con sus propios colores, Windows ajustará los colores disponibles para que el resultado final sea lo más aproximado posible al pedido.

Cuándo dibujar en el Canvas. El mensaje WM_PAINT y el método Paint.

Cuando queremos dibujar sobre el canvas, debemos colocar el código correspondiente dentro del método **paint** perteneciente al objeto TGraphicControl (el objeto TCustomControl tambien dispone de este método). Esto es así porque cuando nuestro componente debe ser dibujado (p.e. el gradiente de colores), bien porque se trate de la primera vez o en respuesta a una solicitud de Windows (p.e. por un solapamiento de ventanas), Windows enviará un mensaje del tipo **WM_PAINT** a nuestro componente. El componente, de forma automática (herencia), hace una llamada al método paint para redibujar el componente (en nuestro caso, el gradiente).

En general no tendremos la necesidad de investigar más en el mensaje WM_PAINT. Cómo acabamos de explicar, lo único que nos interesa de él (al menos para nuestro componente), es que cuando se reciba este mensaje, Delphi ejecutará el método paint asociado al componente. Y es dentro de este método donde debemos añadir el código necesario para dibujar el gradiente.

Por tanto, todo los pasos que hemos visto anteriormente que son necesarios para dibujar el gradiente, estarán situados dentro del método TGradiente.Paint, que será declarado como override. El método paint lo declararemos de tipo protected, ya que no se debe poder acceder a él desde fuera de nuestro componente.

Como último detalle, señalar que cuando cambiemos el valor de algunas de las propiedades que afectan al dibujo del gradiente, deberemos redibujar el gradiente. Esto se consigue mediante el método repaint, que fuerza el redibujado del gradiente. En él código fuente se observa claramente como cada vez que se cambia el valor de la dirección del gradiente o del color inicial o final, se hace una llamada a repaint.

Otros detalles en el diseño del componente. Publicando propiedades heredadas.

Uno de los detalles que nos queda por ver es la alineación del componente. El objeto TGraphicControl posee la propiedad Align (por tanto nuestro componente la heredará), pero está declarada como pública. A nosotros nos interesa declararla como published para que aparezca en el inspector de objetos en tiempo de diseño. Para lograrlo debemos redeclarar la propiedad align en la sección published:

```
published
```

```
  propery align
```

Es importante hacer notar dos aspectos: el primero es que una redeclaración sólo puede hacer menos restrictivo el acceso a una propiedad (p.e paso de protected a public), pero no más restrictivo (p.e paso de public a protected).

El segundo aspecto es que al redeclarar, no es necesario especificar el tipo de la propiedad, basta con indicar su nombre. Lo que si podemos hacer en el momento de la redeclaración es definir un nuevo valor por defecto para dicha propiedad.

- El resto de los detalles del código fuente no creo que merezca la pena comentarlos, ya que la teoría y práctica de los conceptos mostrados ya se ha visto en anteriores unidades. Se declaran las propiedades Direccion, ColorDesde y ColorHasta y se escriben los métodos necesarios para almacenar los valores correspondientes... De todas formas, si alguien tiene dudas, ya sabe, me escribe donde siempre.

Código fuente del componente

```
unit Gradient;

interface

uses
  Classes, Controls, Graphics, WinTypes, WinProcs;

type
  TDireccion = (dHorizontal, dVertical);  {Tipo de dirección del
gradiente}
  TGradiente = class(TGraphicControl)
  private
    FDireccion : TDireccion;   {Dirección del gradiente}
    FColorDesde, FColorHasta : TColor;     {Color del gradiente}
    procedure SetDireccion(valor : TDireccion);
    procedure SetColorDesde(valor : TColor);
    procedure SetColorHasta(valor : TColor);
  protected
    procedure Paint; override;
  public
    constructor Create(AOwner : TComponent); override;
  published
    property Direccion : TDireccion read FDireccion write SetDireccion
default dHorizontal;
    property ColorDesde : TColor read FColorDesde write SetColorDesde
default clBlue;
    property ColorHasta : TColor read FColorHasta write SetColorHasta
default clBlack;
    property Align;     {Redeclaración de la propiedad como publicada}
  end;

procedure Register;

implementation

constructor TGradiente.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);       {Siempre lo primero a hacer}
  FDireccion:=dHorizontal;        {Valores por defecto}
  FColorDesde:=clBlue;
  FColorHasta:=clBlack;
  Width:=100;
  Height:=100;
```

```
end;

procedure TGradiente.SetDireccion(Valor : TDireccion);
begin
  if FDireccion <> valor then
  begin
    FDireccion := Valor;
    Repaint;              {Fuerza redibujado del gradiente}
  end;
end;

procedure TGradiente.SetColorDesde(Valor : TColor);
begin
  if FColorDesde <> Valor then
  begin
    FColorDesde := Valor;
    Repaint;              {Fuerza redibujado del gradiente}
  end;
end;

procedure TGradiente.SetColorHasta(Valor : TColor);
begin
  if FColorHasta <> Valor then
  begin
    FColorHasta := Valor;
    Repaint;              {Fuerza redibujado del gradiente}
  end;
end;

procedure TGradiente.Paint;
var
  RGBDesde, RGBHasta, RGBDif : array[0..2] of byte;  {Colores inicial
y final y diferencia de colores}
  contador, Rojo, Verde, Azul : integer;
  Banda : TRect;                                     {Coordenadas del
recuadro a pintar}
  Factor : array[0..2] of shortint;                  {+1 si gradiente
creciente o -1 en caso decreciente}
begin
  RGBDesde[0]:=GetRValue(ColorToRGB(FColorDesde));
  RGBDesde[1]:=GetGValue(ColorToRGB(FColorDesde));
  RGBDesde[2]:=GetBValue(ColorToRGB(FColorDesde));    {Se descomponen
los colores en rojo, verde y azul}
  RGBHasta[0]:=GetRValue(ColorToRGB(FColorHasta));
  RGBHasta[1]:=GetGValue(ColorToRGB(FColorHasta));
  RGBHasta[2]:=GetBValue(ColorToRGB(FColorHasta));
  for contador:=0 to 2 do         {Calculo de RGBDif[] y factor[]}
  begin
    RGBDif[contador]:=Abs(RGBHasta[contador]-RGBDesde[contador]);
    If RGBHasta[contador]>RGBDesde[contador] then factor[contador]:=1
else factor[contador]:=-1;
  end;
  Canvas.Pen.Style:=psSolid;            {Asignamos el estilo del pen}
  Canvas.Pen.Mode:=pmCopy;              {Idem modo}
  if FDireccion = dHorizontal then      {Si el canvas es
horizontal...}
  begin
    Banda.Left:=0;                      {Coordenadas recuadro}
    Banda.Right:=Width;
    for contador:=0 to 255 do
    begin
```

```
      Banda.Top:=MulDiv(contador,height,256);
      Banda.Bottom:=MulDIv(contador+1,height,256);
      Rojo:=RGBDesde[0]+factor[0]*MulDiv(contador,RGBDif[0],255);
      Verde:=RGBDesde[1]+factor[1]*MulDiv(contador,RGBDif[1],255);
      Azul:=RGBDesde[2]+factor[2]*MulDiv(contador,RGBDif[2],255);
      Canvas.Brush.Color:=RGB(Rojo,Verde,Azul);
      Canvas.FillRect(Banda);           {Pintamos el recuadro}
    end;
  end;
  if FDireccion = dVertical then         {Gradiente vertical}
  begin
    Banda.Top:=0;
    Banda.Bottom:=Height;
    for contador:=0 to 255 do
    begin
      Banda.Left:=MulDiv(contador,width,256);
      Banda.Right:=MulDIv(contador+1,width,256);
      Rojo:=RGBDesde[0]+factor[0]*MulDiv(contador,RGBDif[0],255);
      Verde:=RGBDesde[1]+factor[1]*MulDiv(contador,RGBDif[1],255);
      Azul:=RGBDesde[2]+factor[2]*MulDiv(contador,RGBDif[2],255);
      Canvas.Brush.Color:=RGB(Rojo,Verde,Azul);
      Canvas.FillRect(Banda);
    end;
  end;
end;

procedure Register;
begin
  RegisterComponents('Curso', [TGradiente]);
end;

end.
```

Unidad 6. TMultiGrid: color, alineación y multiples líneas en un TStringGrid.

 En esta unidad crearemos un componente de tipo StringGrid mejorado. Aprenderemos a crear eventos, los cuales nos permitirán dotar de nuevas y potentes posibilidades a nuestros componentes. Además, estudiaremos el método OnDrawCell y profundizaremos en los temas ya tratados en unidades anteriores referentes a los objetos Canvas y Brush.

Objetivo del componente

Como acabamos de mencionar, nuestro propósito es crear un componente de tipo StringGrid pero con nuevas funcionalidades. De modo que lo primero que deberíamos tener presente es que nos permite y que no nos permite hacer el componente TStringGrid estándard. Así que si aún no conoces dicho objeto, haremos una pequeña pausa para darte tiempo a ir Delphi y mirar la ayuda en línea ;) ¿Ya esta? Bien, pues ahora las nuevas características que implementaremos a nuestro componente: el color y la alineación de las celdas al nivel que deseemos (columna, filas e incluso celdas individuales, incluyendo alineación vertical) así como una nueva propiedad denominada

multilinea que nos permitirá mostrar más de una línea de texto en cada celda del grid (rejilla).

La figura que sigue es un ejemplo del componente en funcionamiento:

	Enero	Febrero	Total Año	Notas
Zona A	1.000.000	1.250.000	9.150.000	Incremento sobre año anterior
Zona B	1.450.000	950.000	4.150.000	Decremento
Resto de Zonas	4.000.000	3.250.000	17.250.000	Incremento sobre año anterior
TOTAL	6.450.000	5.450.000	30.550.000	

Implementando la alineación de celdas. Creando nuestros propios eventos.

Vamos a comenzar con la propiedad alineación. Lo primero que debemos decidir es cómo vamos a implantarla. Fundamentalmente, hay tres posibilidades:

- Definiendo la alineación de un modo global. Si nos decidiesemos por esta implementación, bastaría con definir una propiedad (denominada, logicamente, Alignment) que controlaría la alineación de todas las celdas del grid. Esto sería muy sencillo, pero muy poco práctico, ya que lo normal es que queramos tener, por ejemplo, las celdas de cabecera centradas, las de texto alineadas a la izquierda, las númericas a la derecha, etc.
- Una solución un poco mejor: definir la alineación a nivel de columnas. De este modo cada columna puede estar alineada con independencia de las demás, pero persiste el tema de que las filas de una misma columna deben tener la misma alineación (cabecera y datos). Un problema añadido es que para implementar esta elección deberíamos crearnos un editor de propiedades, y aún no sabemos como hacerlo (pero paciencia, que en próximas unidades se verá). Este es el caso del editor de columnas que incorpora Delphi 2.
- La tercera posibilidad nos ofrece un control total: definir la alineación de cada celda a nivel individual. De este modo cada celda tendrá su propia alineación con independencia de las demás. La pega es que, como más adelante veremos, requiere un poco más de esfuerzo por parte del usuario del componente.

Como se ve, cada una de las tres implementaciones tiene sus ventajas e inconvenientes. En nuestro componente vamos a implementar una combinación del primer y tercer método. De este modo, definiremos una propiedad Alignment que especificará la alineación por defecto de las celdas del grid y al mismo tiempo, mediante un nuevo evento, se podrá determinar la alineación de celdas a nivel individual.

La implementación de la alineación horizontal a nivel global no tiene ningún misterio: basta definir la propiedad Alignment de tipo TAlignment (tipo ya incluido en Delphi). El campo que guardará el valor de esta propiedad se denominará FAlignment. Para escribir el valor de la propiedad utilizaremos el método SetAlignment, mientras que la

lectura de la propiedad se hará directamente del campo FAlignment. De este modo tenemos perfectamente definido la interfaz de la propiedad Alignment. Respecto a cómo dibujaremos el contenido de la celda con la alineación apropiada lo dejaremos para más tarde, cuando estudiemos el evento OnDrawCell.

La alineación vertical se desarrolla de una forma similar. El único aspecto que conviene hacer notar es que Delphi no incorpora un tipo TVerticalAlignment, de modo que debemos crearlo nosotros:

```
TVerticalAlignment = (vaTop, vaCenter, vaBottom);
```

Nos queda ver como implementamos la interfaz de la alineación de las celdas a nivel individual. Para ello vamos a crearnos un nuevo **evento**, que se disparará cada vez que necesitemos saber el estado de alineación de una celda en concreto.

Cómo ya se vió en la unidad 2, un evento (también denominado suceso) es un mecanismo que víncula una acción a cierto código. Más concretamente, un evento es un puntero a método, un puntero que apunta a un método específico de una instancia de objeto específica ¡toma ya que impresionante!;)

La forma de implementar un evento se hace mediante propiedades, es decir, los eventos son propiedades. A diferencia de las propiedades estándard, los eventos no utilizan métodos para implementar las partes read y write. En su lugar, las propiedades de eventos utilizan un campo privado del mismo tipo que la propiedad.

Pero basta de teoría y manos a la obra. Como ya se ha mencionado, vamos a crear un nuevo evento que se debe disparar cada vez que necesitemos obtener el valor de la alineación de una celda específica. Lo primero que debemos hacer, por tanto, es definir nuestro tipo de suceso. Esto lo hacemos mediante la siguiente sentencia:

```
  TMultiGridAlignmentEvent=procedure(Sender:TObject;
ARow,ACol:LongInt;
    var HorAlignment: TAlignment; var VerAlignment:
TVerticalAlignment) of object;
```

Los parámetros del evento TMultiGridAlignmentEvent son los siguientes:

- Sender: Que identifica el objeto que hace la llamada.
- ARow y ACol : Que identifican la celda de la que se pide la información de alineación.
- var HorAlignment y var VerAlignment: Parámetros de tipo var en el que el usuario del componente debe devolver la alineación horizontal y vertical correspondiente a la celda. Ver el código de ejemplo al final de la unidad par ver una demostración del uso que un usuario puede hacer de este evento.

Conviene hacer notar las palabras of object al final de la declaración del tipo de evento.

Ya hemos definido el tipo de evento. Ahora debemos crear un campo que guarde el estado de la propiedad OnGetCellAlignment. Esto lo realizamos en la parte private:

```
  private
     ...
     FOnGetCellAlignment : TMultiGridAlignmentEvent;
```

Por último, nos queda definir la propiedad en la parte published:

```
    property OnGetCellAlignment : TMultiGridAlignmentEvent read
FOnGetCellAlignment write   FOnGetCellAlignment;
```

Sólo nos queda disparar el vento cuando lo necesitemos. Aunque un poco más adelante veremos con más detalle cuando queremos hacerlo en nuestro componente, el fragmento que sigue nos permite ver el mecanismo general:

```
  if Assigned(FOnGetCellAlignment) then
    FOnGetCellAlignment(Self,ARow,ACol,HorAlineacion,VerAlineacion);
```

Importante: Antes de activar un evento, conviene mirar primero si dicho evento tiene un manejador de evento asignado, ya que el usuario del componente no tiene porque haber escrito dicho manejador. De ahí la comparación *if Assigned*: si hay un manejedor de evento, se le llama, pero si no lo hay no se hace nada.

Implementando la fuente, estilo y color de celdas.

Si se ha entendido la sección anterior, no habrá ningún problema en entender esta, ya que la forma de implementar los atributos de color y fuente de una celda determinada se hará de una forma similar.

Defininimos un nuevo evento que se activará cuando sea necesario determinar los atributos de la celda. Para ello, crearemos el tipo del evento, el campo privado que lo almacenará y la propiedad asociada a dicho evento:

```
  TMultiGridColorEvent=procedure(Sender: TObject; ARow,Acol: LongInt;
     AState: TGridDrawState; Abrush: TBrush; AFont:TFont) of object;
  ...
  FOnGetCellColor : TMultiGridColorEvent;
  ...
  property OnGetCellColor : TMultiGridColorEvent read FOnGetCellColor
write FOnGetCellColor;
```

La diferencia principal entre este evento y el correspondiente a la alineación viene dada por los parámetros ABruh y AFont. El usuario del componente debe devolver el brush y font correspondientes a la celda en concreto referenciada por ARow y ACol. El parámetro AState nos informa del estado de la celda (seleccionada, enfocada, fija...)

Celdas Multilínea.

Pasemos ahora a la implemantación de celdas multilínea. Esta es una característica que se echa mucho de menos en el StringGrid estándard y que, como vamos a ver, no nos va a costar casi nada implementar.

En primer lugar definiremos la interfaz. Para ello, vamos a crear una nueva propiedad denominada MultiLinea (original, ¿verdad? ;)). Dicha propiedad se almacenará en el campo FMultiLinea que será de tipo boolean. Si FMultiLinea está a false, nuestro componente se comportará como el StringGrid normal, mientras que si está a true se permitirán las celdas multilínea.

```
  private
    FMultiLinea : Boolean;
  ...
```

```
    property MultiLinea : Boolean read FMultiLinea write SetMultiLinea
default False;
```

Conviene hacer notar que en el método SetMultiLinea, si se asigna un nuevo valor a FMultiLinea se provoca el repintado del componente mediante la instrucción Invalidate. Esta misma técnica se utiliza en los métodos SetAlignment, SetVerticalAlignment y SetColor.

Nos queda por ver cómo dibujaremos las celdas multilínea. Este aspecto lo trataremos en la siguiente sección.

El corazón de la bestia: el método DrawCell.

Hasta ahora, nos hemos centrado en la interfaz de nuestro componente; ha llegado el momento de definir la implementación. Todo el proceso de dibujar una celda se realiza en el método DrawCell. Este método es el verdadero corazón de nuestro componente, ya que en él se debe escribir todo el código encargado de calcular y pintar el texto correspondiente a una determinada celda.

El evento OnDrawCell pasa los siguientes parámetros al método DrawCell:

- ACol y ARow: Que identifican a la celda que se debe dibujar en pantalla.
- ARect : Estructura de tipo rectangular que identifica la esquina superior izquierda y la inferior derecha (en pixels) de la celda a dibujar.
- AState: Es el estado actual de la celda (seleccionda, enfocada o fija). En principio no utilizaremos el valor pasado en este parámetro.

Ya sabemos donde. Ahora falta por ver cómo. En principio podría asustarnos todo lo que tenemos que hacer: calcular la alineación horizontal y vertical, activar los eventos de alineación y color, fragmentar el contenido de la celda en varias líneas... Pero no hay motivo: Delphi y el Api de windows vienen en nuestra ayuda y toda esta codificación se reduce a 20 o 30 líneas fácilmente entendibles. A continuación se muestra el código correspondiente al método DrawCell, el cúal paso a comentar:

```
procedure TMultiGrid.DrawCell(ACol,ARow : LongInt; ARect : TRect;
AState : TGridDrawState);
Const
  TextAlignments : Array[TAlignment] of Word = (dt_Left, dt_Right,
dt_Center);
Var
  HorAlineacion : TAlignment;
  VerAlineacion : TVerticalAlignment;
  Texto : string;
  Altura : integer;
  CRect : TRect;
  opciones : integer;
begin
  Texto:=Cells[ARow,Acol];
  HorAlineacion:=FAlignment;
  VerAlineacion:=FVerticalAlignment;
  if Assigned(FOnGetCellAlignment) then
FOnGetCellAlignment(Self,ARow,ACol,HorAlineacion,VerAlineacion);
  if Assigned(FOnGetCellColor) then
    FOnGetCellColor(Self,ARow,ACol,AState,Canvas.Brush,Canvas.Font);
```

```
  Canvas.FillRect(ARect);

  Inc(ARect.Left,2);
  Dec(ARect.Right,2);
  CRect:=Arect;
  opciones:=TextAlignments[HorAlineacion] or dt_VCenter;
  if Multilinea then opciones:=opciones or dt_WordBreak;
  if not DefaultDrawing then
    inherited DrawCell(ACol,ARow,ARect,AState)
  else
    with ARect,Canvas do
    begin
      Altura:=DrawText(Handle,PChar(Texto),-1,CRect,opciones or
dt_CalcRect);
      if FVerticalAlignment = vaCenter then
      begin
        if Altura < Bottom-Top+1 then
        begin
          Top:=(Bottom+Top-Altura) shr 1;
          Bottom:=Top+Altura;
        end;
      end
      else if FVerticalAlignment = vaBottom then
        if Altura < Bottom-Top+1 then Top:=Bottom-Altura;
      DrawText(Handle,PChar(Texto),-1,ARect,opciones)
    end;
end;
```

Lo primero que hacemos es guardar el contenido de la celda a dibujar en la variable Texto (de tipo string). A continuación se dan los valores por defecto a las variables HorAlineacion y VerAlineacion debido a que si el usuario no ha introducido una alineación particular para la celda en cuestión se aplicarán estas alineaciones.

Ahora viene una de las claves: la llamada a los eventos. Si el usuario ha escrito un manejador para el evento Alineación, se llama. Lo mismo ocurre para el evento Color. De este modo ya tenemos el tratamiento específico de la celda.

Lo siguiente es dibujar el fondo de la celda mediante el método FillRect al que pasamos el recuadro de dibujo de dicha celda (ARect).

Necesitamos ahora hacer una pausa para explicar cómo vamos a dibujar el texto.

A primera vista, lo lógico sería utilizar el método TextOut del objeto Canvas. Este método necesita como parámetros las coordenadas (x ,y) donde dibujar el texto y una cadena con el texto a dibujar. Pero para nuestros propósitos se queda corto, ya que tendríamos que calcular a mano la posición de dibujo para que la alineación fuera la correcta. Además tendríamos que calcular las divisiones de palabras necesarias para las celdas multilíneas, etc. En fin, un rollo. Y en mi opinión si podemos evitarnos todo este trabajo, pues mejor, ¿no? ;) Y lo bueno es que... ¡¡¡podemos!!!. El secreto: una función del Api de windows: DrawText

DrawText necesita los siguientes parámetros:

- Un handle al objeto en que queremos dibujar (el canvas del grid).
- Una cadena de tipo terminada en nulo con el texto a dibujar (Pchar(Texto))

- Un número que indica el número de carácteres a dibujar (-1 para todos)
- El rectángulo en el que hay que formatear y dibujar el texto (TRect)
- Una serie de opciones de formateo del texto. Nosotros vamos a utilizar las que controlan la alineación del texto (dt_Left, dt_Center, dt_Right, dt_VCenter), el troceo de palabras (dt_WordBreak) y el cálculo de la altura (dt_CalcRect)

Hay más opciones para DrawText, de modo que si quieres más información sólo tienes que mirar en la ayuda en línea.

En la práctica necesitamos hacer dos llamadas a DrawText, una primera en la que al incluir la opción dt_CalcRect no se dibuja nada en pantallas, sino que sólo se calcula la altura requerida del rectángulo para el troceo de palabras (multilínea). Esta altura la guardamos en la variable Altura. Es importante hacer notar que el parámetro Rect pasado cómo parámetro se modifica, por lo que necesitaremos previamente guardar su estado. Posteriormente, una vez reajustado el rectángulo se hace una segunda llamada sin la opción dt_Calcrect que es la que de verdad dibuja el texto en el canvas.

Volvemos ahora al flujo del programa. Despues de rellenar el fondo de la celda con FillRect, copiamos en la variable CRect el rectángulo original (ARect) y se preparan las opciones con que vamos a llamar a DrawText. Aquí surge un pequeño problema, ya que HorAlineación es del tipo TAlignment (ta_LeftJustify...) y DrawText no entiende este tipo, por lo que es necesario una conversión entre dicho tipo y el que DrawText entiende. Esta conversión la llevamos a cabo mediente una matriz constante denominada TextAlignments. A continuación, si la propiedad Multilinea está a true, se añade dt_WordBreak a las opciones de DrawText.

Lo que se hace a continuación es verificar si el usuario ha tocado el valor de la propiedad DefaultDrawing. Si el valor de esta propiedad es false indica que es el usuario el que se encarga de todo el proceso, en caso contrario, el componente se encarga del dibujo.

Si es el componente el que se encarga de todo (para eso lo queremos, ¿no?) hacemos la primera llamada a DrawText para obtener la altura requerida del rectángulo de celda. Con esta altura y, siempre en cuando sea posible (todo el texto multilínea quepa en la celda), se pasa a centrar el texto en la celda (o a poner a top o bottom según corresponda). Una vez hecho esto, volvemos a llamar a DrawText para que se produzca, ahora si, el dibujado del texto en pantalla. Ojo a que en esta ocación pasamos ARect como rectángulo. Esto es lógico, ya que nos nos podemos salir del rectángulo que tiene la celda. Y ¡¡¡voilà!!! hemos terminado.

Este es a grandes rasgos el funcionamiento del método DrawCell. Conviene que mires y remires el mismo hasta que lo llegues a entender ya que es extremadamente potente y puede servirte para otras ocasiones. Y si tienes dudas, ya sabes, me escribes ;)

Otros detalles.

Por último quiero mencionar algunos pequeños detalles:

- Se redefine la propiedad Options del StringGrid estándard para poner por defecto a true las opciones goRowSizing y goColSizing ya que al ser una rejilla

multilínea lo más normal es que estas opciones esten siempre a True y en el grid estándard están a false.

- Como siempre, en el constructor declaramos los valores por defecto para las distintas propiedades que definimos (alineación, color, multilínea, options).
- Después del código fuente se muestra un ejemplo de los eventos alineación y color que podría escribir un usario del componente. Este ejemplo da lugar al grid que se muestra al principio de la unidad.

Código fuente del componente.

```
unit MultiGrid;       { (c) 1997 by Luis Roche }

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids;

type
  TVerticalAlignment = (vaTop, vaCenter, vaBottom);

TMultiGridAlignmentEvent=procedure(Sender:TObject;ARow,ACol:LongInt;va
r HorAlignment:TAlignment;var VerAlignment:TVerticalAlignment) of
object;

TMultiGridColorEvent=procedure(Sender:TObject;ARow,Acol:LongInt;AState
:TGridDrawState;ABrush:TBrush;AFont:TFont) of object;
  TMultiGrid = class(TStringGrid)
  private
    FAlignment : TAlignment;
    FVerticalAlignment : TVerticalAlignment;
    FMultiLinea : Boolean;
    FOnGetCellAlignment : TMultiGridAlignmentEvent;
    FOnGetCellColor : TMultiGridColorEvent;
    procedure SetAlignment(Valor : TAlignment);
    procedure SetVerticalAlignment(Valor : TVerticalAlignment);
    procedure SetMultiLinea(Valor : Boolean);
  protected
    procedure DrawCell(ACol,ARow : LongInt; ARect : TRect; AState :
TGridDrawState); override;
  public
    constructor Create(AOwner : TComponent); override;
  published
    property Alignment : TAlignment read FAlignment write SetAlignment
default taLeftJustify;
    property VerticalAlignment : TVerticalAlignment read
FVerticalAlignment write SetVerticalAlignment default vaCenter;
    property MultiLinea : Boolean read FMultiLinea write SetMultiLinea
default False;
    property OnGetCellAlignment : TMultiGridAlignmentEvent read
FOnGetCellAlignment write FOnGetCellAlignment;
    property OnGetCellColor : TMultiGridColorEvent read
FOnGetCellColor write FOnGetCellColor;
    property Options default
[goFixedVertLine,goFixedHorzLine,goVertLine,goHorzLine,goRangeSelect,g
oRowSizing,goColSizing];
  end;

procedure Register;
```

```
implementation

constructor TMultiGrid.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);
  FAlignment:=taLeftJustify;
  FVerticalAlignment:=vaCenter;
  {FColor:=clWindowText;}
  FMultiLinea:=False;

Options:=[goFixedVertLine,goFixedHorzLine,goVertLine,goHorzLine,goRang
eSelect,goRowSizing,goColSizing];
end;

procedure TMultiGrid.SetAlignment(Valor : TAlignment);
begin
  if valor <> FAlignment then
  begin
    FAlignment:=Valor;
    Invalidate;
  end;
end;

procedure TMultiGrid.SetVerticalAlignment(Valor : TVerticalAlignment);
begin
  if valor <> FVerticalAlignment then
  begin
    FVerticalAlignment:=Valor;
    Invalidate;
  end;
end;

procedure TMultiGrid.SetMultiLinea(Valor : Boolean);
begin
  if valor <> FMultiLinea then
  begin
    FMultiLinea:=Valor;
    Invalidate;
  end;
end;

procedure TMultiGrid.DrawCell(ACol,ARow : LongInt; ARect : TRect;
AState : TGridDrawState);
Const
  TextAlignments : Array[TAlignment] of Word = (dt_Left, dt_Right,
dt_Center);
Var
  HorAlineacion : TAlignment;
  VerAlineacion : TVerticalAlignment;
  Texto : string;
  Altura : integer;
  CRect : TRect;
  opciones : integer;
begin
  Texto:=Cells[ARow,Acol];
  HorAlineacion:=FAlignment;
  VerAlineacion:=FVerticalAlignment;
  if Assigned(FOnGetCellAlignment) then
FOnGetCellAlignment(Self,ARow,ACol,HorAlineacion,VerAlineacion);
  if Assigned(FOnGetCellColor) then
    FOnGetCellColor(Self,ARow,ACol,AState,Canvas.Brush,Canvas.Font);
```

```
  Canvas.FillRect(ARect);
  Inc(ARect.Left,2);
  Dec(ARect.Right,2);
  CRect:=Arect;
  opciones:=TextAlignments[HorAlineacion] or dt_VCenter;
  if Multilinea then opciones:=opciones or dt_WordBreak;
  if not DefaultDrawing then
    inherited DrawCell(ACol,ARow,ARect,AState)
  else
    with ARect,Canvas do
    begin
      Altura:=DrawText(Handle,PChar(Texto),-1,CRect,opciones or
dt_CalcRect);
      if FVerticalAlignment = vaCenter then
      begin
        if Altura < Bottom-Top+1 then
        begin
          Top:=(Bottom+Top-Altura) shr 1;
          Bottom:=Top+Altura;
        end;
      end
      else if FVerticalAlignment = vaBottom then
        if Altura < Bottom-Top+1 then Top:=Bottom-Altura;
      DrawText(Handle,PChar(Texto),-1,ARect,opciones)
    end;
end;

procedure Register;
begin
  RegisterComponents('Curso', [TMultiGrid]);
end;

end.
```

Ejemplo de utilización.

A continuación se muestra un ejemplo de utilización de nuestro nuevo componente. Por brevedad, sólo se muestra el código correspondiente a los eventos FormCreate y GetCellCOlor y GetCellAlignment. Por supuesto, este no es un jemplo real, ya que por simplicidad, el contenido de las celdas se determina en el FormCreate cuando lo normal es que dicho contenido venga de otra parte (de cálculos, bases de datos, etc.). Pero como muestra de uso es suficiente.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MultiGrid1.Cells[0,1]:='Enero';
  MultiGrid1.Cells[0,2]:='Febrero';
  MultiGrid1.Cells[0,3]:='Total Año';
  MultiGrid1.Cells[0,4]:='Notas';
  MultiGrid1.Cells[1,0]:='Zona A';
  MultiGrid1.Cells[2,0]:='Zona B';
  MultiGrid1.Cells[3,0]:='Resto de Zonas';
  MultiGrid1.Cells[4,0]:='TOTAL';
  MultiGrid1.Cells[1,1]:='1.000.000';
  MultiGrid1.Cells[1,2]:='1.250.000';
  MultiGrid1.Cells[1,3]:='9.150.000';
  MultiGrid1.Cells[1,4]:='Incremento sobre año anterior';
  MultiGrid1.Cells[2,1]:='1.450.000';
  MultiGrid1.Cells[2,2]:='  950.000';
```

```
  MultiGrid1.Cells[2,3]:='4.150.000';
  MultiGrid1.Cells[2,4]:='Decremento';
  MultiGrid1.Cells[3,1]:='4.000.000';
  MultiGrid1.Cells[3,2]:='3.250.000';
  MultiGrid1.Cells[3,3]:='17.250.000';
  MultiGrid1.Cells[3,4]:='Incremento sobre año anterior';
  MultiGrid1.Cells[4,1]:='6.450.000';
  MultiGrid1.Cells[4,2]:='5.450.000';
  MultiGrid1.Cells[4,3]:='30.550.000';
  MultiGrid1.Cells[4,4]:='';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  MultiGrid1.Color:=clRed;
end;

procedure TForm1.MultiGrid1GetCellAlignment(Sender: TObject; ARow,
  ACol: Longint; var HorAlignment: TAlignment;
  var VerAlignment: TVerticalAlignment);
begin
  if (ACol in [1..3]) and (ARow in [1..4]) then
    HorAlignment:=taRightJustify
  else HorAlignment:=taCenter;
end;

procedure TForm1.MultiGrid1GetCellColor(Sender: TObject; ARow,
  Acol: Longint; AState: TGridDrawState; Abrush: TBrush; AFont:
TFont);
begin
  if (ARow=0) then
  begin
    ABrush.Color:=clMaroon;
    AFont.Color:=clWhite;
    AFont.Style:=[fsBold];
  end
  else if (ACol=0) then
  begin
    AFont.Color:=clBlack;
    AFont.Style:=[fsBold];
    ABrush.Color:=clYellow;
  end
  else
  begin
    AFont.Color:=clBlack;
    ABrush.Color:=clYellow;
  end;
end;
```

Unidad 7. TDBViewer: Un visualizador rápido de bases de datos.

 En esta unidad transformaremos un form en un componente. En concreto, crearemos un form para visualizar rápidamente bases de datos y lo integraremos dentro de un componente. De este modo en nuestros proyectos, cuando en una pantalla deseemos una opción de visulizar una base de datos, nos bastará con pinchar el

componente en el form, asignar sus propiedades, llamar al método execute y el resto del trabajo lo hará el propio componente.

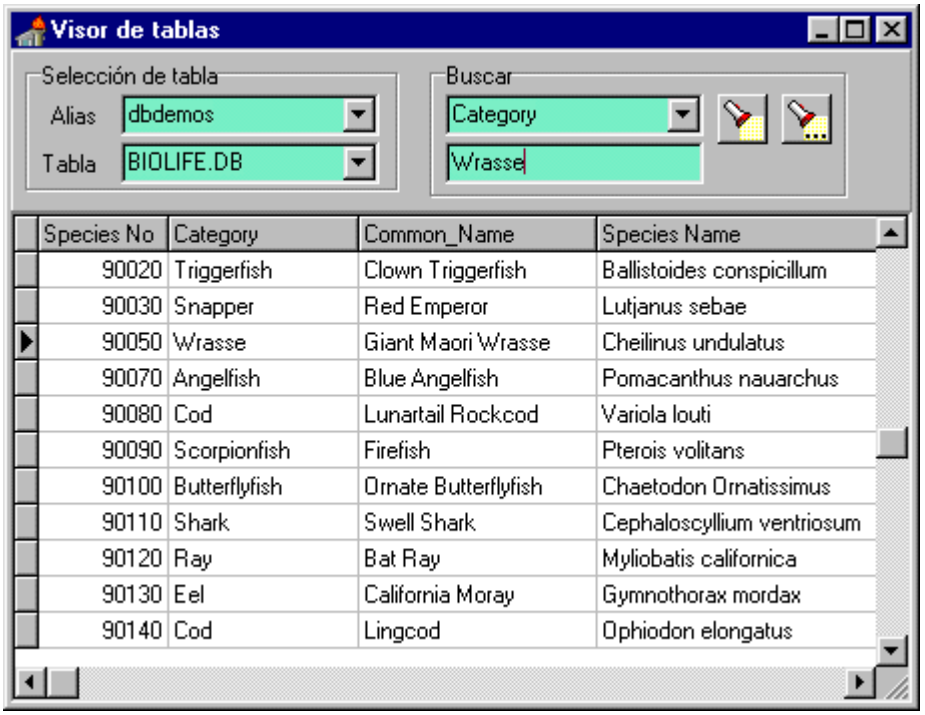
Esta conversión de cuadros de diálogo en componentes es una opción muy potente ya que nos permite reutilizar el código de nuestros diversos proyectos de una forma rápida y sencilla. Seguro que en cuanto conozcamos la técnica de cómo lograrlo se nos ocurrirán infinitas posibilidades para convertir pantallas que utilizamos a menudo en componentes. Baste con decir que yo tengo un componente que encapsula ¡toda una aplicación de 10 o 12 pantallas!.

Hay que hacer notar que en el desarrollo del form utilizo algunas funciones de Delphi 2 que no existen en Delphi 1, con lo cúal el código fuente no es directamente compilable en Delphi 1. Si alguno quiere convertirlo a 16 bits no debe tener demasiados problemas y puede ser un buen ejercicio ;)

Objetivo del componente

Nuestro objetivo es crear un componente que el usuario pueda añadir en un proyecto y para el que pueda definir propiedades en tiempo de diseño. El componente "adaptador" asociado con el form se encargará de crearlo y mostrarlo en tiempo de ejecución. De este modo logramos una alta reutilización del form.

El form que vamos integrar en el componente es un visualizador / editor de tablas de bases de datos. A continuación se muestra el componente en tiempo de ejecución y se comentan sus funciones principales.



- Mediante dos combo box el usuario podrá elegir la tabla a mostrar mediante la selección de un alias y de una tabla de dicho alias.

- Se mostrará un dbgrid de la tabla seleccionada pudiendo este grid ser editable o no en base a una propiedad del componente.
- En un panel adicional se podrán realizar búsquedas sobre cualquier campo de la base de datos mediante la selección del texto a buscar y del campo correspondiente.
- El grid tiene asociado un popup menu que permite o no el filtrado de registros en función de otra propiedad del componente. (Este menu popup no aparece en la imagen enterior).

El proceso de creación de nuestro componente lo vamos a dividir en los siguientes pasos, los cuales iremos viendo en detalle en las siguientes secciones:

- Primero, crearemos de forma visual el form, teniendo en cuenta que debemos dotarle de un mecanismo de conexión con el componente propiamente dicho.
- A continuación crearemos el componente, incluyendo propiedades, métodos, etc.
- Por último realizaremos la conexión entre el componente y el form.

Diseño del form

La creación del form visualizador la haremos de la manera tradicional, es decir, en Delphi, haremos click en File|New Form. Una vez nos aparezca el form en blanco, introduciremos los diversos elementos que lo conforman. No voy a explicar ahora todos los pasos a seguir para diseñar el form, sino que me centrare en los más relevantes. La unidad correspondiente a dicho form y el fichero DFM asociado (frmView.pas y frmView.dfm) puedes bajartelo desde el índice del curso de modo que no tienes que diseñarlo enteramente.

La sección de interface del form es la siguiente:

```
unit frmView;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, DB, DBTables, Grids, DBGrids, ExtCtrls, Buttons, Menus;

type
  TfrmVisor = class(TForm)
    tTable: TTable;
    dsDataSource: TDataSource;
    dbGrid: TDBGrid;
    pTop: TPanel;
    pSeleccionTabla: TPanel;
    pBuscar: TPanel;
    gbTablas: TGroupBox;
    Alias: TLabel;
    Tabla: TLabel;
    cbAlias: TComboBox;
    cbTables: TComboBox;
    gbBuscar: TGroupBox;
    cbFields: TComboBox;
```

```
    eSearch: TEdit;
    Buscar: TSpeedButton;
    BuscarSiguiente: TSpeedButton;
    PopupMenu1: TPopupMenu;
    Igual1: TMenuItem;
    Distinto1: TMenuItem;
    Menor1: TMenuItem;
    Mayor1: TMenuItem;
    N1: TMenuItem;
    Activar1: TMenuItem;
    Eliminar1: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure FillcbFields;
    procedure Inicializar;
    procedure cbAliasChange(Sender: TObject);
    procedure cbTablesChange(Sender: TObject);
    procedure BuscarClick(Sender: TObject);
    procedure BuscarSiguienteClick(Sender: TObject);
    procedure dbGridColEnter(Sender: TObject);
    procedure PopupMenu1Popup(Sender: TObject);
    procedure Activar1Click(Sender: TObject);
    procedure Eliminar1Click(Sender: TObject);
    procedure Igual1Click(Sender: TObject);
    procedure Distinto1Click(Sender: TObject);
    procedure Menor1Click(Sender: TObject);
    procedure Mayor1Click(Sender: TObject);
    private
  public
    FReadOnly, FFiltrar, FBuscar : boolean;
  end;

procedure PascalStr(var s : string);
procedure ComaStr(var s : string);
function Condicion(AField : TField; AOp : string) : string;
procedure AddFilter(AField : TField; AOp : string);

var
  frmVisor: TfrmVisor;
```

Lo primero que conviene hacer notar es la sección **pública**. En ella definimos tres variables de tipo booleano: FReadOnly, FFiltrar y FBuscar. Estas tres variables forman el interface entre el form y el componente. Así, a la propiedad ReadOnly que implementaremos en el componente le corresponde la variable FReadOnly del form y, de forma análoga ocurre con las propiedades PermitirBucar (AllowSearch) y PermitirFiltrar (AllowFilter) del componente. **Será el componente, y no el form** quién en su método execute asignará los valores correspondientes a dichas variables.

Conviene que para seguir la explicación que viene ahora tengas a la vista el código fuente completo del form ya que voy a hacer referencias contínuas al mismo.

El método Inicializar es el que se encarga de ajustar la visualización del form según las variables FReadOnly, FFiltrar y FBuscar. Este método será llamado por el propio componente una vez asignado el valor a dichas variables y su cometido es el siguiente:

- Ajusta la propiedad ReadOnly del grid de acuerdo con el valor de FReadOnly.
- Muestra o no el panel de búsqueda según el valor de FBuscar.
- Asigna o no el menú popup al grid de acuerdo al valor de FFiltrar.

Veamos ahora como se efectúa la visualización de una tabla. En el evento OnCreate del form se comienza por llenar el combo box cbAlias con todos los alias disponibles. A continuación, de todos los existentes, se selecciona el primero y se fuerza el relleno del combo box cbTables con todas las tablas existentes en el alias (método cbAliasChange). Dicho relleno fuerza la llamada al método cbTablesChange, el cúal se encarga de activar la tabla (también se aprovecha para llenar el combo box cbFields con los campos de la tabla seleccionada). De este modo, el grid, que ya está conectado al datasource y a la tabla correspondiente muestra dicha tabla. Posteriormente si el usuario cambia la selección del alias o de la tabla el proceso se repite y se muestra la nueva tabla seleccionada.

Por simplicidad se ha omitido el tratamiento de errores en la apertura de tablas, ya que no es el objetivo de esta lección. Pero deberías añadir como mínimo un bloque try..except cuando se activa la tabla y actuar en función del tipo de error que se puede producir. Por mi parte, como acabo de decir, no he introducido dicho tratamiento y dejo que delphi se las tenga que ver con las excepciones ya que, al fin y al cabo, para eso está, ¿no? ;)

Pasamos ahora a la búsqueda de registros. Para realizarla, el método BuscarClick se basa en el contenido del edit eSearch que contiene la cadena a buscar y en el combo box cbFields que tiene el campo por el que hay que buscar. La búsqueda se realiza mediante los métodos del objeto TTable FindFirst y FindNext, los cuales supongo los conoces de sobra. En todo caso, encontrarás información sobre los mismos en la ayuda en línea de Delphi.

Nos queda por ver como implementar el filtrado de registros. Hay diversas formas de hacerlo, y me he decantado por utilizar un menu popup que permiter activar, desactivar y añadir nuevas condiciones al filtro. Concretamente el menu tiene las opciones siguientes: añadir una condición del tipo Campo = valor, Campo > Valor, Campo < valor y Campo <> Valor, activar el filtro y eliminar el filtro. Las diversas condiciones del filtro se unen entre sí mediante el operador AND.

El método Activar1Click y Eliminar1Click no necesitan demasiada explicación ya que se limitan a poner la propiedad Filtered del objeto TTable a True o False según corresponda.

Los restantes métodos del menú (Igual1Click...) añaden úna nueva condición al filtro. Para ello, se utiliza el procedimiento AddFilter, que recibe como parámetros el campo sobre el que realizar la nueva condición de filtro (que es la columna activa del dbgrid al pulsar el botón derecho del ratón) y el operador (=,<,> o <> según corresponda. Dicho método, después de formatear correctamente la cadena, la añade a la propiedad filter del objeto TTable y efectúa un refresh para que el nuevo filtro se active.

Y con esto acabamos la parte correspondiente al diseño del form. Fácil, ¿verdad? De todos modos, si téneis alguna duda, el código fuente está bastante documentado. Y si aún así no lo veis claro, ya sabeis, le dais al teclado y me mandáis un e-mail.

Creación del componente

Empecemos ahora con el componente, el cúal es realmente sencillo. Consta de tres propiedades y dos métodos, los cuáles, a esta altura del curso no deben tener ningún secreto para vosotros:

- La **propiedad ReadOnly** determinará si el grid del form será editable o no. Por defecto toma el valor False, es decir, editable. Sus métodos read y write se limitan a leer y escribir sobre el campo FReadOnly del componente (no confundir con el FReadOnly del form visto en la sección anterior).
- La **propiedad AllowSearch** determina si debe aparecer el panel de búsqueda en el form. Por defecto su valor es True. Nuevamente sus métodos read y write actúan directamente sobre el campo FAllowSearch.
- La **propiedad AllowFilter** determina si se permite o no el filtrado de registros en el form. Por defecto, su valor es True. De sus métodos read y write ¡que os voy a decir que no os imagineis! ;)
- El **constructor** se limita a llamar al constructor heredado y asignar los valores por defecto para las propiedades.
- El **método Execute** se encargará de crear y mostrar el form y lo veremos en la siguiente sección.

Y eso es todo. ¿Era sencillo o no? Esta simplicidad os la encontrareis en general siempre que estéis transformando un form en componente. Basta con crear unas cuantas propiedades que actuarán sobre el form y definir un método (que por cierto, no tiene porque llamarse execute) para lanzar el form y ¡voila!

¡Ah! que no se os olvide añadir a la clausula uses del componente (en la sección de implementación) la unidad del form. En nuestro caso:

```
...
implementation

uses frmView;
...
```

La conexión entre el componente y el form

Como acabamos de decir, la ejecución del form se lleva a cabo a través del método Execute del componente:

```
procedure TDBViewer.Execute;
begin
  {Crear el form}
  frmVisor:=TFrmVisor.Create(Application);
  try
    {Asignar las propiedades del componente a las variables
     correspondientes del form}
    frmVisor.FReadOnly:=FReadOnly;
    frmVisor.FBuscar:=FAllowSearch;
    frmVisor.FFiltrar:=FAllowFilter;
    {Inicializar el form}
    frmVisor.Inicializar;
    {Mostrar el form de forma modal}
    frmVisor.ShowModal;
  finally
    {Liberar el form}
    frmVisor.Free;
```

```
  end;
end;
```

El método comienza por crear el form cuyo propietario será la aplicación. A continuación se realiza la conexión entre el form y el componente. Para ello se van asignado a las variables FReadOnly, FBuscar y FFiltrar del form las correspondientes del componente. Una vez realizada esta asignación, debemos hacer que el form se visualice correctamente en base a estos valores (recordar que el form se ha creado, pero aún no se ha mostrado). Esto se consigue llamando al método Inicializar del form (este método lo vimos en la sección Diseño del form).

Ya sólo nos queda mostrar el form de forma modal y "esperar" a que el usuario se canse de jugar con él, momento en el cuál aprovecharemos para destruir el form mediante la llamada al método free.

Como último aspecto a destacar conviene hacer notar el bloque try..finally que nos asegura que, pase lo que pase, la memoria ocupada al crear el form, se liberará correctamente.

Instrucciones de uso del componente y conclusiones

Una vez registrado el componente en la paleta de la forma habitual, su uso es muy sencillo. Cuando queramos utilizarlo en un proyecto, basta con pinchar el componente DBViewer donde deseemos, asignar sus propiedades a nuestra elección (bien sea en tiempo de diseño o de ejecución) y ejecutar el método Execute. Dicho método lo llamaremos desde nos interese (bien sea mediante un botón, en el form create, etc).

Como conclusión, si nos olvidamos de lo que hace el form en si (que puede ser desde un simple about box a todo un proyecto encapsulado en él), nos daremos cuenta que el proceso de convertirlo en componente es muy simple. A partir de las dos unidades que tengamos (una para el form y otra para el componente), añadiremos las propiedades necesarias para personalizar la apariencia del form y crearemos un método (execute) que se encargará de crear y mostrar el form.

Por último me gustaría que si añadis nuevas funciones al form, me lo hagáis saber. Para el curso he escogido un diseño simple, pero con dos o tres cosas más se puede convertir en un componente realmente útil. Así que aquí van algunas ideas: añadirle algunos eventos, tales como OnOpenTable, OnTableError, etc; añadirle la posibilidad de imprimir la base de datos; añadirle un visor de campos memo o de imagenes... ¡Todo depende de vuestras necesidades! :)

Código fuente del componente.

```
unit DBView;          { (c) 1997 by Luis Roche }

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs;

type
```

```
  TDBViewer = class(TComponent)
  private
    FReadOnly : boolean;
    FAllowSearch : boolean;
    FAllowFilter : boolean;
  protected
  public
    constructor Create(AOwner : TComponent); override;
    procedure Execute;
  published
    property ReadOnly : boolean read FReadOnly write FReadOnly default
False;
    property AllowSearch : boolean read FAllowSearch write
FAllowSearch default True;
    property AllowFilter : boolean read FAllowFilter write
FAllowFilter default True;
  end;

procedure Register;

implementation

uses frmView;

constructor TDBViewer.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);
  {Asignar la propiedades por defecto}
  FReadOnly:=False;
  FAllowSearch:=True;
  FAllowFilter:=True;
end;

procedure TDBViewer.Execute;
begin
  {Crear el form}
  frmVisor:=TFrmVisor.Create(Application);
  try
    {Asignar las propiedades del componente a las variables
     correspondientes del form}
    frmVisor.FReadOnly:=FReadOnly;
    frmVisor.FBuscar:=FAllowSearch;
    frmVisor.FFiltrar:=FAllowFilter;
    {Inicializar el form}
    frmVisor.Inicializar;
    {Mostrar el form de forma modal}
    frmVisor.ShowModal;
  finally
    {Liberar el form}
    frmVisor.Free;
  end;
end;

procedure Register;
begin
  RegisterComponents('Curso', [TDBViewer]);
end;

end.
```

Código fuente del form.

Se muestra a continuación el código integro de la unidad del form. El fichero dfm junto con el resto del componente lo podeis bajar directamente en formato zip desde el índice del curso.

```
unit frmView;            { (c) 1997 by Luis Roche }

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, DB, DBTables, Grids, DBGrids, ExtCtrls, Buttons, Menus;

type
  TfrmVisor = class(TForm)
    tTable: TTable;
    dsDataSource: TDataSource;
    dbGrid: TDBGrid;
    pTop: TPanel;
    pSeleccionTabla: TPanel;
    pBuscar: TPanel;
    gbTablas: TGroupBox;
    Alias: TLabel;
    Tabla: TLabel;
    cbAlias: TComboBox;
    cbTables: TComboBox;
    gbBuscar: TGroupBox;
    cbFields: TComboBox;
    eSearch: TEdit;
    Buscar: TSpeedButton;
    BuscarSiguiente: TSpeedButton;
    PopupMenu1: TPopupMenu;
    Igual1: TMenuItem;
    Distinto1: TMenuItem;
    Menor1: TMenuItem;
    Mayor1: TMenuItem;
    N1: TMenuItem;
    Activar1: TMenuItem;
    Eliminar1: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure FillcbFields;
    procedure Inicializar;
    procedure cbAliasChange(Sender: TObject);
    procedure cbTablesChange(Sender: TObject);
    procedure BuscarClick(Sender: TObject);
    procedure BuscarSiguienteClick(Sender: TObject);
    procedure dbGridColEnter(Sender: TObject);
    procedure PopupMenu1Popup(Sender: TObject);
    procedure Activar1Click(Sender: TObject);
    procedure Eliminar1Click(Sender: TObject);
    procedure Igual1Click(Sender: TObject);
    procedure Distinto1Click(Sender: TObject);
    procedure Menor1Click(Sender: TObject);
    procedure Mayor1Click(Sender: TObject);
    private
  public
    FReadOnly, FFiltrar, FBuscar : boolean;
  end;

procedure PascalStr(var s : string);
```

```
procedure ComaStr(var s : string);
function Condicion(AField : TField; AOp : string) : string;
procedure AddFilter(AField : TField; AOp : string);

var
  frmVisor: TfrmVisor;

implementation

{$R *.DFM}

procedure TfrmVisor.FormCreate(Sender: TObject);
begin
  {Rellenar combo box con los alias disponibles}
  Session.GetAliasNames(cbAlias.Items);
  {Seleccionar el primer alias}
  cbAlias.ItemIndex:=0;
  {Provocar el relleno del combo box con el nombre de las tablas}
  cbAliasChange(Sender);
  {Deshabilitar el botón de buscar siguiente}
  BuscarSiguiente.Enabled:=False;
end;

procedure TfrmVisor.Inicializar;
{Procedimiento que se encarga de ajustar visualmente el form a las
 propiedades del componente}
begin
  dbGrid.ReadOnly:=FReadOnly;
  pBuscar.Visible:=FBuscar;
  if FFiltrar then
    dbGrid.PopupMenu:=PopupMenu1
  else
    dbGrid.PopupMenu:=nil;
end;

procedure TfrmVisor.cbAliasChange(Sender: TObject);
begin
  {Rellenar combo box con las tablas existentes en el alias
seleccionado}

Session.GetTableNames(cbAlias.Items[cbAlias.ItemIndex],'*.*',True,True
,cbTables.Items);
  {Seleccionar la primera tabla}
  cbTables.ItemIndex:=0;
  {Provocar la apertura de la tabla}
  cbTablesChange(Sender);
end;

procedure TfrmVisor.cbTablesChange(Sender: TObject);
begin
  {Asignar propiedades del objeto tTable}
  tTable.Active:=False;
  tTable.DatabaseName:=cbAlias.Text;
  tTable.TableName:=cbTables.Text;
  {Activar la tabla}
  tTable.Active:=True;
  {Rellenar combo box con los campos existentes en la tabla}
  FillcbFields;
end;

procedure TfrmVisor.FillcbFields;
```

```
{Procedimiento que se encarga de rellenar combo box con los
 campos existentes en la tabla seleccionada}
Var
  i : integer;
begin
  cbFields.Items.Clear;
  cbFields.Text:='';
  for i:=0 to tTable.FieldCount-1 do
    cbFields.Items.Add(tTable.Fields[i].DisplayLabel);
end;

procedure TfrmVisor.BuscarClick(Sender: TObject);
{Procedimiento que busca un texto determinado en el
 campo seleccionado.}
Var
  F : TField;
  s : string;
begin
  {Si no hay seleccionado ningún campo donde buscar, salir}
  if cbFields.ItemIndex=-1 then
    exit;
  with tTable do
  begin
    {Obtener cadena a buscar y campo donde buscar}
    s:=eSearch.Text;
    F:=Fields[cbFields.ItemIndex];
    {Convertir la cadena a buscar al formato adecuado}
    case F.DataType of
      FtString, FtDate, FtTime, FtDateTime : PascalStr(s);
      FtFloat : ComaStr(s);
    end;
    {Realizar la búsqueda propiamente dicha}
    Filter:='['+F.FieldName+']='+s;
    FindFirst;
    {Habilitar el botón buscar siguiente sólo si la búsqueda
     ha tenido éxito}
    BuscarSiguiente.Enabled:=Found;
  end;
end;

procedure TfrmVisor.BuscarSiguienteClick(Sender: TObject);
{Procedimiento que busca la siguiente aparición del
 texto correspondiente}
begin
  tTable.FindNext;
  BuscarSiguiente.Enabled:=tTable.Found;
end;

procedure PascalStr(var s : string);
{Rutina de propósito general que se encarga de formatear adecuadamente
 cadenas que contienen apóstrofes en su interior (p.e. O'Hara)}
Var
  i : integer;
begin
  for i:=Length(s) downto 1 do
    if s[i]='''' then Insert('''',s,i);
  s:=''''+s+'''';
end;

procedure ComaStr(var s : string);
{Rutina que sustituye la coma decimal por el punto decimal}
```

```
Var
  i : integer;
begin
  for i:=Length(s) downto 1 do
    if s[i]=',' then s[i]:='.';
end;

procedure TfrmVisor.dbGridColEnter(Sender: TObject);
begin
  {Al entrar en una columna del grid, actualizar el combo box
   con el campo donde buscar}
  cbFields.ItemIndex:=dbGrid.SelectedField.Index;
end;

procedure TfrmVisor.PopupMenu1Popup(Sender: TObject);
{Mostrar de forma adecuada el menu popup al pulsar el botón
 derecho del ratón sobre el grid}
begin
  with tTable do
  begin
    Activar1.Checked:=Filtered;
    Activar1.Enabled:=Filter<>'';
    Eliminar1.Enabled:=Filter<>'';
  end;
end;

procedure TfrmVisor.Activar1Click(Sender: TObject);
{Activar el filtrado de registros}
begin
  tTable.Filtered:=not tTable.Filtered;
  tTable.Refresh;
end;

procedure TfrmVisor.Eliminar1Click(Sender: TObject);
{Desactivar el filtrado de registros}
begin
  tTable.Filtered:=False;
  tTable.Filter:='';
  tTable.Refresh;
end;

procedure TfrmVisor.Igual1Click(Sender: TObject);
begin
  AddFilter(dbGrid.SelectedField,'=');
end;

procedure TfrmVisor.Distinto1Click(Sender: TObject);
begin
  AddFilter(dbGrid.SelectedField,'<>');
end;

procedure TfrmVisor.Menor1Click(Sender: TObject);
begin
  AddFilter(dbGrid.SelectedField,'<=');
end;

procedure TfrmVisor.Mayor1Click(Sender: TObject);
begin
  AddFilter(dbGrid.SelectedField,'>=');
end;
```

```
procedure AddFilter(AField : TField; AOp : string);
{Añade una nueva condición al filtro}
Var
  s : string;
begin
  {Poner en s la condición a buscar}
  s:=Condicion(AField,AOp);
  with AField.DataSet do
  begin
    if Filter='' then Filter:=s
    else Filter:=Filter+' AND ' + s;
    Filtered:=True;
    Refresh;
  end;
end;

function Condicion(AField : TField; AOp : string) : string;
{Formatea la condición a buscar de forma adecuada}
Var
  Valor : string;
begin
  Valor:=AField.AsString;
  case AField.DataType of
    FtString, FtDate, FtTime, FtDateTime : PascalStr(Valor);
    FtFloat : ComaStr(Valor);
  end;
  Condicion:=Format('([%s] %s %s)',[AField.FieldName,AOp,Valor]);
end;

end.
```

Unidad 8. Editores de Propiedades (I).

En está unidad aprenderemos a crear nuestros propios editores de propiedades. Los editores de propiedades son una poderosa herramienta que pueden marcar la diferencia entre un componente simplemente aceptable y uno realmente formidable. A través de ellos podremos dotar a nuestros componentes de nuevas características que van más allá de la siempre validación del valor de las propiedades. Gracias a ellos haremos la vida más fácil a los sufridos programadores que utilicen nuestros componentes, y eso siempre es de agradecer, ¿no? ;)

Introducción a los editores de propiedades

Hasta ahora nos hemos centrado en la creación de componentes sin preocuparnos de cómo se introducían los valores de las distintas propiedades definidas en ellos. Ha llegado el momento de centrarnos en este aspecto. Como todos sabemos para introducir y leer el valor de las propiedades en tiempo de diseño utilizamos el inspector de objetos. El inspector de objetos nos muestra en una primera columna el nombre de la propiedad y en una segunda el valor de dicha propiedad. Para introducir un nuevo valor lo tecleamos y listo. Pero esto es tan sólo apariencia. Los componentes interactuan con el inspector de objetos a través de los editores de propiedades. Desde el punto de vista del usuario, es el inspector de objetos el responsable de editar las propiedades, pero detrás de la escena hay una serie de objetos, los editores de propiedades, que se encargan de definir las capacidades de edición de las diversas propiedades de un componente.

En tiempo de diseño, cuando se selcciona un componente, el inspector de objetos crea instancias de los editores de propiedades necesarios para editar las propiedades definidas en el componente seleccionado. Cuando termina la edición, el mismo inpector de objetos destruye los editores de propiedades creados.

Delphi incluye una serie de editores de propiedades por defecto que son suficiente para la mayoría de las propiedades con las que trabajamos habitualmente. Pero como siempre, Delphi es tan potente que permite que creemos nuestros propios editores y que incluso reemplacemos los que una determinada propiedad tiene por defecto. ¡Chupate esa VB! ;)

Los deberes de un Editor de Propiedades

Un editor de propiedades debe dar respuesta a dos aspectos principales:

- Definir cómo debe ser editada la propiedad. Aquí hay dos posibilidades: el valor de la propiedad puede ser modificado sobre el propio inspector de objetos (bien sea introduciendo un nuevo valor o seleccionandolo de una lista de valores), o se puede utilizar un cuadro de diálogo para dotar de mayor flexibilidad a la edición (p.e. la propiedad color)

- Convertir el valor de la propiedad a un valor de tipo string. El inspector de objetos **siempre trabaja con strings.** Aunque la propiedad sea de tipo integer o float o sea un método, el inspector de objetos sólo trata con representaciones de tipo string de dichas propiedades. Es el editor de propiedades el que debe suministrar al inspector de objetos de dicha representación en string de la propiedad. Esta "traducción" puede ir desde lo más sencillo (utilizar la función IntToStr) hasta lo más complejo (programar una rutina para la traducción). Todo dependerá del tipo de propiedad con la que estemos tratando.

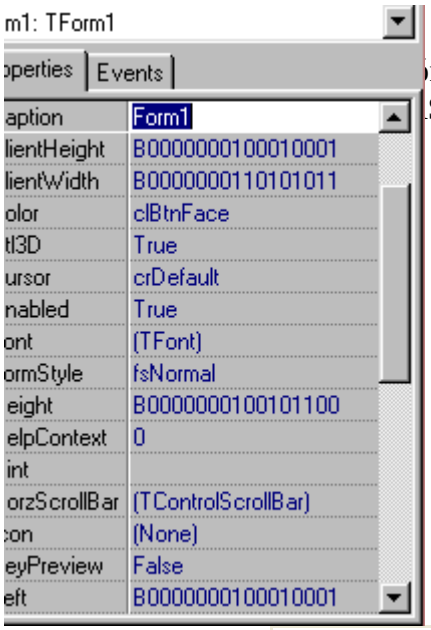
Pasos necesarios para escribir un editor de propiedades

Los pasos que es necesario seguir para escribir un editor de propiedades son los siguientes:

- Crear una nueva unidad en la que definiremos el editor de propiedades. Más adelante hablaremos más extensamente sobre este punto, ya que no es tan trivial como puede parecer en principio ;)

- Añadir la unidad **DsgnIntf** a la clausula uses del editor de propiedades. En esta unidad estan definidos los editores de propiedades por defecto que utiliza Delphi, además de la importantísima clase TPropertyEditor, la cuál es la clase base de todos los editores de propiedades.

- Crear una nueva clase que descienda de TPropertyEditor o de alguno de sus desciendentes. Por convención, el nombre de los editores de propiedades finaliza con la palabra Property. P.e. TIntegerProperty, TStringProperty... A



continuación se muestran los principales editores de propiedades por defecto que incorpora Delphi.

Editor de propiedades	Tipo
TPropertyEditor	Clase base para todos los editores de propiedades
TIntegerProperty	Byte, word, integer, Longint
TCharProperty	Char
TEnumProperty	Tipos enumerados
TSetProperty	Sets
TFloatProperty	Single, Double, Extended, Comp, Currency
TStringProperty	Strings
TClassProperty	Cualquier objeto
TMethodProperty	Cualquier método (eventos)
TComponentProperty	Para propiedades que hacen referencia a componentes

-
- Implementar los métodos necesarios para dotar al editor de propiedades de las funcionalidades deseadas.
- Registrar el editor de propiedades en la VCL

Más adelante profundizaremos en todos aspectos según vayamos desarrollando distintos editores, pero ahora ha llegado del momento de crear nuestro primer editor de propiedades.

Un editor de propiedades para números binarios

Vamos a desarrollar nuestro primer editor de propiedades, que será editable sobre el propio inspector de objetos. Para ello pongámonos en situación: imaginemos que hemos creado un componente con una propiedad de tipo integer que guarda un valor que debe introducirse y visualizarse en base binaria y no en base decimal. Si no crearamos un editor de propiedades y dejaramos que Delphi utilizase el editor por defecto (TIntegerProperty en este caso) nos encontraríamos que tanto la introducción como la visualización del valor de la propiedad se realizaría en base decimal, que no es lo que

queremos. Este es el típico caso en el que desarrollando un simple editor de propiedades ahorramos trabajo al futuro usuario de nuestro componente, evitando que tenga que hacer la conversión entre decimal y binario.

Como ya hemos comentado, tenemos que decidir de que clase vamos a derivar nuestro editor. En nuestro caso es fácil, ya que la propiedad almacenará un valor de tipo integer, así que ¿lo adivinais? ¡Si!, de TIntegerProperty ;)

Muy bien, nuestra propiedad almacena un entero, pero no queremos que el inspector de objetos nos muestre su valor decimal directamente, queremos efectuar una conversión de dicho valor decimal a base binaria y que sólo entonces el inspector de objetos nos muestre el valor. Para lograr esto debemos implementar (override) la función **GetValue**. Esta función, definida en TPropertyEditor, es llamada por el inspector de objetos para obtener la representación del valor de la propiedad en forma de string. Dentro de esta función debemos convertir el valor de la propiedad de decimal a binario y, a continuación, transformar este valor en string, ya que como ya hemos mencionado el inspector de objetos siempre muestra strings. De este modo la función GetValue queda así:

```
unit BinaryPropEd;

interface

uses DsgnIntf;

type
  TBinIntegerProperty = class(TIntegerProperty)
  public
    function GetValue : string; override;
    procedure SetValue(const Value : String); override;
  end;

procedure Register;

implementation

Const
  Bits16 : Array [1..16] of Integer =
(32768,16384,8192,4096,2048,1024,512,256,128,64,32,16,8,4,2,1);

function TBinIntegerProperty.GetValue : string;
Var
  Num, i : integer;
begin
  Num:=GetOrdValue;
  Result := '0000000000000000';
  for i := 1 to 16 Do
    if ( Num >= Bits16[i] ) Then
    begin
      Num := Num - Bits16[i];
      Result[i] := '1';
    end;
  if ( Num > 0 ) Then
    raise EPropertyError.Create('Error converting
'+IntToStr(GetOrdValue) + ' to binary');
  Insert('B',Result,1);
end;
```

```
...

end.
```

De la implementación de esta función la parte más importante es la primera línea:

```
Num:=GetOrdValue
```

Necesitamos obtener el valor que tiene en ese momento la propiedad para trabajar sobre él. Para ello utilizamos el método **GetOrdValue**, definido de nuevo en TPropertyEditor, el cuál se encarga de devolver el valor de la propiedad en forma de ordinal (integer). De forma análoga, existen los métodos GetFloatValue, GetMethodValue, GetVarValue, etc. para utilizar con el tipo de propiedad correspondiente.

Una vez almacenado el valor de la propiedad en la variable Num, comienza la conversión del valor de decimal a binario, la cúal es fácil de entender. Cabe hacer notar que al máximo número de digitos binarios soportados es 16, margen más que suficiente para la mayoría de aplicaciones.

Por último tenemos que devolver un valor de tipo string como resultado de la función. Para ello vamos almacenando en la variable Result el string a devolver. Para finalizar, anteponemos la letra 'B' a la cadena para indicar que se trata de base binaria.

Y eso es todo en lo que a esta función respecta. De este modo, cuando el inspector de objetos deba mostrar el valor de la propiedad, llamará a el método GetValue el cuál le devolverá el string correspondiente. Pero nos queda la otra mitad: estaría bien que pudieramas introducir el valor de la propiedad tanto en decimal como en binario según nos interesase, ¿verdad? pues vamos a ello. ;)

Para conseguir esta funcionalidad debemos implementar (override) el método **SetValue**, definido de nuevo en la clase TPropertyEditor. Cuando el usuario entra un nuevo valor usando el inspector de objetos, este llama al método SetValue, el cuál debe efectuar la traducción inversa a la efectuada por el método GetValue. Es decir, debe convertir el string que contiene el nuevo valor de la propiedad al tipo de datos de dicha propiedad. En nuestro caso, el string vendrá en base decimal o binaria (en este último caso, la primera letra de la cadena será una 'B') y convertirlo a base decimal. Para ello implementaremos el método SetValue de la siguiente forma:

```
...

type
  TBinIntegerProperty = class(TIntegerProperty)
  public
    function GetValue : string; override;
    procedure SetValue(const Value : String); override;
  end;

procedure Register;

implementation

...
```

```
procedure TBinIntegerProperty.SetValue(const Value : String);
Var
  i, Total, Longitud : integer;
  NumText : string;
begin
  if UpperCase(Value[1])='B' then
  begin
    NumText:=Copy(Trim(Value),2,Length(Trim(Value))-1);
    NumText:=Copy('0000000000000000',1,16-Length(NumText)) + NumText;
    Total:=0;
    for i:=1 to Length(NumText) do
    begin
      if not (NumText[i] in ['0','1']) then
        raise EPropertyError.Create(NumText[i] + ' is not a valid
binary digit')
      else if NumText[i]='1' then
        Total:=Total+Bits16[i];
    end;
    SetOrdValue(Total);
  end
  else
    SetOrdValue(StrToInt(Value));
end;

...

end.
```

En la implementación de este método primero comprobamos si el usuario ha introducido el nuevo valor de la propiedad en base decimal o en base binaria. En el primer caso, tan sólo hay que convertir el string a integer mediante la función StrToInt(Value) y, a continuación, utilizar el método **SetOrdValue** para almacenar el valor correspondiente. De forma análoga, según el tipo de la propiedad, existen los métodos SeFloatValue, etc.

En el caso de que la primera letra de la cadena sea una 'B', se convierte la cadena con el valor binario a base decimal y se vuelve a utilizar el método SetOrdValue para almacenar el valor en la propiedad.

Una vez implementados estos dos métodos (GetValue y SetValue) ya tenemos nuestro editor de propiedades terminado; tan sólo nos queda un pequeño, pero indispensable paso, registrarlo en la VCL.

Registro de un editor de propiedades

De igual manera que debemos registrar en la VCL los componentes, con los editores de propiedades debemos hacer lo mismo. Par ello disponemos del método RegisterPropertyEditor (definido en la unidad DsgnIntf) que tiene la siguiente declaración:

```
procedure RegisterPropertyEditor(PropertyType : PTypeInfo;
ComponentClass : TClass;
                                                    const
PropertyName : string; EditorClass : TPropertyEditorClass);
```

PropertyType hace referencia al tipo de la propiedad al que se aplicará el editor de propiedades. Para suministrar un valor a este parámetro normalmente utilizaremos la función TypeInfo, por ejemplo, TypeInfo(integer).

ComponentClass permite restringir el uso del editor de propiedades a la clase específica. Un valor de nil registra el editor para todos los componentes.

PropertyName especifica el nombre de la propiedad. Un valor distinto de nil registra el editor sólo para la propiedad especificada, mientras que un valor " lo registra para todas las propiedades

EditorClass especifica el editor de propiedades que se registra (la clase).

Jugando con estos parámetros, tenemos a nuestra disposición un amplio abánico de posibilidades para registrar nuestro editor de propiedades. Vemos algunos ejemplos con nuestro recién creado editor:

- RegisterPropertyEditor(TypeInfo(integer), nil, '', TBinIntegerProperty) registra el editor de propiedades para todos los componentes que tengan una propiedad de tipo entero. Está es la forma más global de registrar un componente y afecta a **todos** los componentes registrados en la VCL. Si registramos nuestro editor así, veremos que todas las propiedades de tipo integer nos aperecen ¡en binario! Además podemos introducir un nuevo valor en decimal o en binario (anteponiendo la letra B). ¡Estamos sustituyendo el editor de propiedades que Delphi utiliza para enteros por el nuestro! :) Esta es la forma más global de registrar un editor de propiedades.

- RegisterPropertyEditor(TypeInfo(integer),TMiComponente,'PropiedadBinaria',TBinIntegerProperty) registra el editor sólo y exclusivamente para la propiedad 'PropiedadBinaria' del componente 'TMiComponente'. Esta es la formá más restringida de registrar un editor de propiedades.

Os aconsejo que registreis el editor de la forma más global posible para que experimenteis con él. Luego, una vez os canseis de ver **todas** las propiedades enteras en binario, os permito que le desinstaleis ;)

También podéis crearos un componente de "mentirijillas" y registrar sólo el editor para el mismo. Algo así:

```
...

Type
  TMiComponente = class(TComponent)
   ...
     property PropiedadBinaria : integer read FPropBin write FPropBin;
  ...
  end;
...
```

Ubicación de un editor de propiedades

Como ya mencionamos al mostrar los pasos que se deben seguir para crear un editor de propiedades, el primero de ellos es crear una unidad donde situar el editor. En ese momento dijimos que no era una elección tan trivial como pudiera parecer en un pricincipio. ¿En que unidad debemos situar el editor? ¿en la misma unidad donde está el componente que tiene la propiedad que será editada?, ¿en una unidad aparte? Tenemos tres posibles ubicaciones:

- La primera opción es situar el editor de propiedades en la misma unidad en que reside el componente que tiene la propiedad que utiliza el editor de propiedades. Esta es la opción más intuitiva; sin embargo no es la más recomendable, sobre todo si el editor utiliza un cuadro de diálogo.

  El motivo es que Delphi sólo utiliza el editor de propiedades en tiempo de diseño. De hecho, el form que contiene el cuadro de diálogo **no** es enlazado con la aplicación, ni tampoco la clase del editor de propiedades. Sin embargo, **si** se enlazan los recursos asociados al cuadro de diálogo, recursos que en tiempo de ejecución lo único que hacen es ocupar espacio, ya que no se utilizarán para nada. Por tanto, estamos incrementando el tamño del ejecutable a lo tonto :(Por otra parte, si el editor de propiedades no utiliza cuadros de diálogo, el efecto no es tan pernicioso pero sigue sin ser recomendable.

- La segunda opción es situar el editor de propiedades (si utiliza un form cómo cuadro de diálogo) en la misma unidad del cuadro de diálogo. De este modo, la aplicación que usa el componente no enlaza el editor de propiedades ni sus recursos asociados.
- La tercera opción es situar el editor de propiedades en una *unidad de registro*. Una unidad de registro es una unidad normal y corriente que agrupa varias sentencias de registro correspondientes a distintos componentes y editores de propiedades que residen en distintas unidades. Esta es la opción más recomendable si el editor de propiedades es utilizado por varios componentes.

Por tanto, las dos últimas opciones son las más recomendables dependiendo la elección de una u otra del uso que se vaya a dar al editor de propiedades. De todas formas, no olvideis añadir a la clausula uses las unidades correspondientes según donde situeis el editor.

En próximas unidades veremos ejemplos de distintas ubicaciones de los editores de propiedades que iremos construyendo.

Código fuente del editor de propiedades.

```
unit BinaryPropEd;

interface

uses DsgnIntf;

type
  TBinIntegerProperty = class(TIntegerProperty)
  public
    function GetValue : string; override;
    procedure SetValue(const Value : String); override;
  end;
```

```
procedure Register;

implementation

Const
  Bits16 : Array [1..16] of Integer =
(32768,16384,8192,4096,2048,1024,512,256,128,64,32,16,8,4,2,1);

function TBinIntegerProperty.GetValue : string;
Var
  Num, i : integer;
begin
  Num:=GetOrdValue;
  Result := '0000000000000000';
  for i := 1 to 16 Do
    if ( Num >= Bits16[i] ) Then
    begin
      Num := Num - Bits16[i];
      Result[i] := '1';
    end;
  if ( Num > 0 ) Then
    raise EPropertyError.Create('Error converting
'+IntToStr(GetOrdValue) + ' to binary');
  Insert('B',Result,1);
end;

procedure TBinIntegerProperty.SetValue(const Value : String);
Var
  i, Total, Longitud : integer;
  NumText : string;
begin
  if UpperCase(Value[1])='B' then
  begin
    NumText:=Copy(Trim(Value),2,Length(Trim(Value))-1);
    NumText:=Copy('0000000000000000',1,16-Length(NumText)) + NumText;
    Total:=0;
    for i:=1 to Length(NumText) do
    begin
      if not (NumText[i] in ['0','1']) then
        raise EPropertyError.Create(NumText[i] + ' is not a valid
binary digit')
      else if NumText[i]='1' then
        Total:=Total+Bits16[i];
    end;
    SetOrdValue(Total);
  end
  else
    SetOrdValue(StrToInt(Value));
end;

procedure Register;
begin

RegisterPropertyEditor(TypeInfo(Integer),nil,'',TBinIntegerProperty);
end;

end.
```

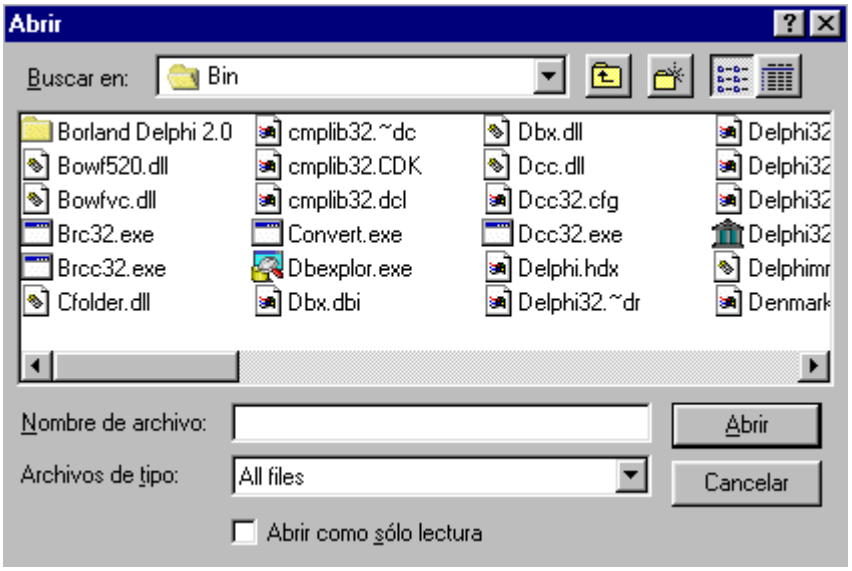
Unidad 9. Editores de Propiedades (II).

Introducción

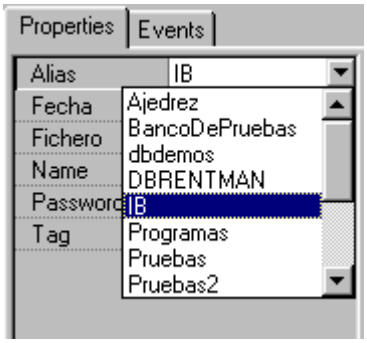
En la anterior unidad aprendimos el funcionamiento básico de un editor de propiedades y desarrollamos un ejemplo de un editor de propiedades que trabajaba sobre el inspector de objetos (BinaryPropEd).

En esta unidad desarrollaremos ¡cuatro! editores de propiedades y un componente que nos servira para probar los editores.

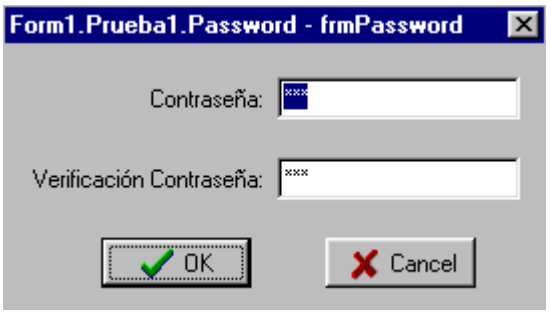
- **TFicheroProperty** es un editor de propiedades de tipo cuadro de diálogo. En alguna ocasión desarrollaremos un componente en el que una de sus propiedades debe almacenar el nombre de un fichero. ¿Obligaremos al sufrido usuario de nuestro componente a tener que escribir la ruta de acceso y el nombre del fichero correspondiente? ¡Claro que no! Mediante el editor de propiedades TFicheroProperty mostraremos al usuario el típico cuadro de diálogo OpenDialog para que le sea más sencilla la asignación de un determinado fichero: No olvidemos que una de las tareas principales de un editor de propiedades es facilitar la vida a los usuarios de nuestros componentes.



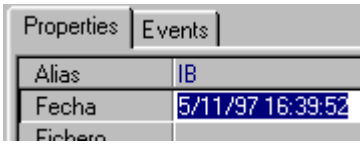
- **TAliasProperty** es un editor de propiedades de tipo lista de valores y su funcionamiento es idéntico al de la propiedad DatabaseName del componente TTable: permite la elección de un alias de base de datos mediante una lista que se despliega en el propio inspector de objetos.



- Profundizando un poco más en el tema de los editores de propiedades tipo cuadro de diálogo crearemos uno partiendo de cero, es decir, sin utilizar un cuadro de diálogo predefinido como en el caso del editor TFicheroProperty. Concretamente, crearemos un editor que nos permita la introducción de contraseñas. Imaginemos que en una propiedad de un componente necesitamos guardar un valor de tipo string (contraseña). Si no creamos ningún editor de propiedades, el usuario del componente escribirá directamente el valor sobre el inspector de objetos, pero lo que escriba será totalmente visible (no aparecerán asteriscos como ocurre con el componente TEdit). Poco elegante, ¿verdad? Así que crearemos un cuadro de diálogo que nos permitirá escribir y verificar la contraseña a asignar a la propiedad. Este será nuestro editor **TPasswordProperty**.



- Por último, y como respuesta a peticiones de algunos de vosotros, crearemos un editor de propiedades, **TDateTimeProperty**,para la introducción de fechas. De este modo podremos introducir cadenas con la fecha en una propiedad de tipo TDateTime (recordemos que las fechas son de tipo Float) en vez de tener que introducir el número equivalente.



Parece interesante, ¿verdad? Pues manos a la obra :)

Un componente de prueba

Antes de comenzar a desarrollar los editores de propiedades vamos a crear un componente que nos permitirá probarlos según los completemos. Este componente lo he denominado TPrueba y su código es el siguiente:

```
unit Unidad9;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  DsgnIntf, DB, PasswordForm;

type
  TPrueba = class(TComponent)
  private
```

```
    FFichero : string;
    FAlias : string;
    FFecha : TDateTime;
    FPassword : string;
  protected
  public
    constructor Create(AOwner : TComponent); override;
  published
    property Fichero : string read FFichero write FFichero;
    property Alias : string read FAlias write FAlias;
    property Fecha : TDateTime read FFecha write FFecha;
    property Password : string read FPassword write FPassword;
  end;

...

implementation

...
constructor TPrueba.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);
  FFecha:=Now;
end;

...
```

Nada del otro mundo. Cuatro propiedades para cada uno de los editores de propiedades a probar y un constructor de los más normal para asignar un valor por defecto a la propiedad fecha.

Lo que si conviene hacer notar es que por simplicidad desarrollaremos el componente y los cuatro editores de propiedades en una única unidad (hay una unidad adicional necesaria para el editor TPasswordProperty). Como digo, lo haremos así por simplicidad, pero no es lo más correcto. En general, cada editor de propiedades debe ir en una unidad independiente del propio componente, cómo ya vimos en la unidad 8.

El editor de propiedades TFicheroProperty

Fijé en la propiedad Fichero tal y como la hemos implementado en el componente TPrueba. Por supuesto, es de tipo string. Además no hemos definido ni métodos Set ni Get, por lo que la escritura/lectura de valores en esta propiedad se hace directamente sobre el campo asociado (FFichero). Tal y como está, cuando un usuario del componente quiera introducir un valor en esta propiedad, tendrá que escribirlo a mano, lo cuál puede llegar a ser una tediosa tarea si el fichero en cuestión consta de un largo path. De modo que, generosos nosotros ;), decidimos echarle una mano: le construiremos un editor de propiedades.

Delphi ya incorporá un componente que maneja muy bien la elección de apertura de archivos: TOpenDialog, de modo que tan sólo nos queda ver cómo podemos utilizarlo en nuestro editor.

El primer paso es decidir de quién descenderá nuestro editor. En nuestro caso, ya que la propiedad fichero es de tipo string, decidimos heredar de TStringProperty. A

continuación debemos decidir si la propiedad será editable en el propio editor de objetos o utilizará un cuadro de diálogo. En este caso parece bastante claro que debemos elegir la segunda opción. Por supuesto esto no quita que el usuario pueda escribir directamente el valor de la propiedad en el inspector de objetos y es que ¡masoquistas los hay en todas partes! ;)

Ahora bien, ¿cómo indicarle a Delphi que se trata de un editor tipo cuadro de diálogo? Para lograrlo, debemos utilizar otro método de la clase base de todos los editores de propiedades (TPropertyEditor): el método **GetAttributes** Este método es una función que determina que características tendrá el editor de propiedades. Las características deseadas debemos devolverlas como resultado de esta función; este resultado es del tipo **TPropertyAttributes** El tipo TPropertyAttributes es un conjunto (set) que puede tomar los siguientes valores:

Atributo	**Descripción**
paValueList	Especifica que el editor debe mostrar una lista con los valores posibles para la propiedad. Para rellenar la lista se utiliza el método GetValues
paSortList	Sólo válida si se selecciona paValueList. Especifica que la lista de valores se mostrará ordenada.
paSubProperties	Indica que el editor define subpropiedades a mostrar identadas a la derecha (p.e. la propiedad font del componente TForm). Para generar la lista de propiedades se utiliza el método GetProperties.
paDialog	Indica que el editor de propiedades debe mostrar un cuadro de diálogo en respuesta al método Edit. (p.e. la propiedad glyph de un TSpeddButton). De este modo al seleccionar la propiedad aparecerá un botón con la cadena '...' para editar la propiedad.
paMultiSelect	Si se selecciona este atributo, la propiedad se mostrará cuando se seleccionen multiples componentes.
paAutoUpdate	Si se selecciona, el inspector de objetos llama al método SetValue cada vez que se cambia el valor de la propiedad. Si no se selecciona, sólo se llama al método SetValue cuando el usuario presiona Enter o sale de la propiedad
paReadOnly	Si se selecciona, el usuario no puede modificar el valor de la propiedad
paRevertable	Especifica si la propiedad puede recuperar su valor original

A la vista de esta tabla, resulta fácil implementar el método GetAtributes en nuestro editor de propiedades:

```
  interface
  ...
  TFicheroProperty = class(TStringProperty)
  public
    function GetAttributes : TPropertyAttributes; override;
    procedure Edit; override;
  end;
  ...

function TFicheroProperty.GetAttributes : TPropertyAttributes;
begin
```

```
  Result:=[paDialog];
end;
```

De nuevo en aras de la simplicidad sólo activamos el atributo paDialog pero, por supuesto, podíamos activar otros atributos tales como paMultiSelect, etc. En este caso, devolveríamos [paDialog, paMultiSelect]

Sólo nos queda por saber cuando activará Delphi el cuadro de diálogo. Al haber seleccionado paDialog, cuando el usuario haga click en el botón '...' o haga doble click sobre la propiedad, Delphi invocará el método Edit de nuestro editor de propiedades. En este método debemos codificar lo necesario para mostrar el cuadro de apertura de ficheros y asignar el fichero seleccionado a la propiedad si el usuario pulsa OK. Dicho método, que debemos reimplementar (override, ver la sección de interface), queda así:

```
...

procedure TFicheroProperty.Edit;
var
  OpenDialog : TOpenDialog;   {TOpenDialog está en la unidad Dialogs,
no ovidar añadir a la clausula uses}
begin
  OpenDialog:=TOpenDialog.Create(Application);   {Creamos el cuadro de
diálogo}
  try
    OpenDialog.Filter:='All files|*.*';    {Asignamos sus propiedades
iniciales}
    if OpenDialog.Execute then   {Si el usuario pulsa OK...}
      SetStrValue(OpenDialog.FileName);   {...asignamos el nuevo valor
a la propiedad}
  finally
    OpenDialog.Free;   {Liberamos el cuadro de diálogo}
  end;
end;

...
```

Más fácil imposible. Tan sólo comentar que nos aseguramos de la liberación de recursos (en este caso del cuadro de diálogo) mediente el uso de la construcción **try..finally**

¡Ya está! Con sólo 10 o 15 líneas de código hemos aliviado el sufrimiento del pobre usuario que se dejaba los dedos escribiendo nombres de ficheros. ¡Si es que somos unos santos! ;) Sólo nos queda registrarlo con la sentencia:

```
procedure Register;
begin
   ...

RegisterPropertyEditor(TypeInfo(string),TPrueba,'Fichero',TFicheroProp
erty);
  ...
end;
```

El editor de propiedades TAliasProperty

Construyamos a continuación un editor de propiedades para la propiedad Alias. Cómo ya sabemos, el componente TTable tiene una propiedad denominada DatabaseName que

especifica el alias de la base de datos al que está conectada la tabla. Para seleccionar un valor para esta propiedad existe una lista desplegable que le muestra al usuario todos los alias disponibles. Este es el comportamiento que queremos para nuestra propiedad Alias. Podríamos buscar por el código fuente de la VCL y tratar de registrar el editor de propiedades de la propiedad DatabaseName para que incluya también nuestra nueva propiedad, pero cómo es muy sencillo la construcción de un editor así, vamos a desarrollarlo nosotros mismos.

La sección de interface de nuestro editor es la asiguiente:

```
...
TAliasProperty = class (TStringProperty)
  public
    function GetAttributes : TPropertyAttributes; override;
    procedure GetValues(Proc : TGetStrProc); override;
  end;
...
```

El método GetAttributes lo hemos conocido en la sección anterior. Nos basta con devolver los valores paValueList para indicar que el editor de propiedad será editable desde el propio inspector de objetos en forma de lista de valores y paSortList para que nos muestre los diversos alias existentes ordenados alfabeticamente.

```
function TAliasProperty.GetAttributes : TPropertyAttributes;
begin
    Result:=[paValueList, paSortList];
end;
```

La tarea que nos queda ahora es llenar la lista con los diversos alias disponibles. Para ello, reimplementaremos (override) el método **GetValues**. Este método recibe un único parámetro: un puntero a método. Realmente este puntero referencia el método interno Add para el string list interno utilizado para llenar la lista desplegable. Los diversos elementos se añaden a la lista en el método GetValues invocando el método referenciado por el puntero al método y convirtiendolo en un valor tipo string. Suena complicado, ¿verdad? No os preocupeis, que no lo es tanto, tan sólo significa que hay que añadir una sentencia del tipo Proc(valor string) por cada elemento a añadir a la lista. En nuestro caso, como queremos añadir los nombres de los alias existentes, haremos un bucle que hará una llamada a Proc(nombre de alias) para ir añadiendo todos los valores. Previamente, habremos obtenido los alias existentes mediante el método **GetAliasList** del objeto TSession:

```
procedure TAliasProperty.GetValues(Proc : TGetStrProc);
Var
  AliasList : TStringList;   {lista con los alias existentes}
  i : integer;
begin
  try
    AliasList := TStringList.Create;   {Creamos la lista}
    Session.GetAliasNames(AliasList);   {Obtenemos los alias
existentes}
    for i:=0 to AliasList.Count - 1 do   {Por cada alias...}
      Proc(AliasList[i]);   {...hacemos la llamada al método Proc}
  finally
    AliasList.Free;   {Liberamos la lista}
  end;
```

```
end;
```

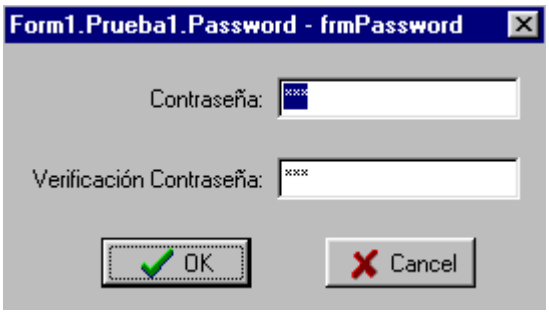
Con esto ya hemos construido nuestro nuevo editor de propiedades. Ya sólo nos falta registrarlo:

```
procedure Register;
begin
   ...

RegisterPropertyEditor(TypeInfo(String),TPrueba,'Alias',TAliasProperty
);
  ...
end;
```

El editor de propiedades TPasswordProperty

Construyamos ahora un nuevo editor de propiedades de tipo cuadro de diálogo. Su funcionamiento será similar al editor TFicheroProperty. Devolveremos paDialog para indicar que se trata de un editor de tipo cuadro de diálogo y en el método Edit codificaremos las sentencias necesarias para el proceso del dicho cuadro de diálogo.

La principal diferencia radica en que no utilizaremos un cuadro de diálogo prediseñado, si no que utilizaremos uno de fabricación casera. Para ello, debemos crear un form y añadirle dos labels, dos cuadros de edición y dos bitbuttons. He aqí una imagen de este form:



Lo más importante es asignar a la propiedad PasswordChar de los dos TEdit, denominados PW1 y PW2, el caracter '*' para que dicho caracter aparezca cuando un usuario teclea una contraseña. Además, en respuesta al evento OnCloseQuery comprobaremos la validez de la contraseña introducida.

El código de la unidad PasswordForm queda así:

```
unit PasswordForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, Buttons;

type
  TfrmPassword = class(TForm)
    lpwd: TLabel;
```

```
    lVpwd: TLabel;
    PW1: TEdit;
    PW2: TEdit;
    bOK: TBitBtn;
    bCancel: TBitBtn;
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
  private
  public
  end;

var
  frmPassword: TfrmPassword;

implementation

{$R *.DFM}

procedure TfrmPassword.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if (ModalResult=mrOK) then
    if (PW1.Text = '') then
    begin
      ShowMessage('Debe introducir una contraseña');
      CanClose:=False;
     end
  else if (ModalResult=mrOK) and (PW1.Text <> PW2.Text) then
   begin
    ShowMessage('Verificación fallida. Por favor reintente');
    CanClose:=False;
   end;
end;

end.
```

Ya hemos construido el cuadro de diálogo, ahora sólo nos resta "engancharlo" al editor de propiedades. Para ello, efectuamos la llamada al form en el método Edit del editor de propiedades. Veamos como queda el código del editor:

```
function TPasswordProperty.GetAttributes : TPropertyAttributes;
begin
  Result:=[paDialog];
end;

function TPasswordProperty.GetValue : string;
begin
  Result:=Format('(%s)',[GetPropType^.Name]);
end;

procedure TPasswordProperty.Edit;
begin
  frmPassword := TfrmPassword.Create(Application);
  try
    frmPassword.Caption:=GetComponent(0).Owner.Name+'.'+
      GetComponent(0).Name+'.'+GetName+' - '+
      frmPassword.Caption;
    frmPassword.PW1.Text:=GetStrValue;
    frmPassword.PW2.Text:=frmPassword.PW1.Text;
    if frmPassword.ShowModal = mrOK then
      SetStrValue(frmPassword.PW1.Text)
```

```
  finally
    frmPassword.Free;
  end;
end;
```

Sólo hay una cosa nueva: en el método GetValue no queremos que se nos muestre el valor de la contraseña, ya que entonces ¿para que nos hemos tomado tantas molestias? De modo que debemos mostrar otra cosa. Podría ser una cadena de asteriscos, pero un convenio utilizado en estos casos es mostrar el tipo de la propiedad, en este caso un string. Dicho esto, sólo nos queda registrar nuestro tercer editor de propiedades:

```
procedure Register;
begin
  ...

RegisterPropertyEditor(TypeInfo(String),TPrueba,'Password',TPasswordPr
operty);
end;
```

El editor de propiedades TDateTimeProperty

Para terminar esta unidad desarrollaremos, a petición de algunos seguidores del curso, un editor de propiedades tipo TDateTime.

Como ya sabemos el tipo TDateTime de Delphi es en realidad un float, de modo que si tenemos en un componente una propiedad de tipo DateTime, y no registramos ningún editor de propiedades para dicha propiedad, el usuario tendrá que escribir la fecha deseada en formato decimal. ¡Qué mal! Sobre todo si pensamos que con seis líneas de código el problema queda solucionado:

```
function TDateTimeProperty.GetValue : string;
begin
  Result:=DateTimeToStr(GetFloatValue);
end;

procedure TDateTimeProperty.SetValue(const Value : string);
begin
  SetFloatValue(StrToDateTime(Value));
end;

procedure Register;
begin
  ...

RegisterPropertyEditor(TypeInfo(TDateTime),TPrueba,'Fecha',TDateTimePr
operty);
end;
```

Nada nuevo aquí. Tan sólo nos limitamos a llamar a los métodos GetFloatValue y SetFloatValue según nos interesa

Conclusiones

Con esto damos por terminado el tema de los editores de propiedades. Hemos aprendido a crear editores editables sobre el inspector de objetos, bien sea directamente o mediante una lista desplegable, así como editores de tipo cuadro de diálogo. La potencia de un editor de propiedades es inmensa: mediante los méetodos SetValue y GetValue podemos efectuar una verificación del valor de la propiedad y actuar en consecuencia. Además, nuestro editor puede hacer mil cosas: consultar un fichero ini, efectuar complejos cálculos o ¡incluso operar con una base de datos!

Pero los editores de propiedades tienen dos limitaciones:

- Operan únicamente en tiempo de diseño, aspecto que debemos tener en cuenta para que el usuario no se quede "colgado" al intentar operar con una propiedad en tiempo de ejecución. Un ejemplo claro son las propiedades de tipo List, p.e. la propiedad items de un TListBox. En tiempo de diseño un editor de propiedades especifico permite al usuario de forma intuitiva añadir y eliminar elementos. A su vez, en tiempo de ejecución, se dispone de los métodos Add y Delete para operar con la lista.
- Un editor de propiedades opera **únicamente** con el valor de **una** propiedad. Es decir, no podemos alterar el valor de distintas propiedades a la vez, ya que los métodos GetValue y SetValue sólo hacen referencia a la propiedad que esta siendo editada, y no al resto de ellas. Por tanto, cuando queramos alterar varias propiedades de un componente al mismo tiempo, deberemos utilizar un **Editor de componentes**, tema este que será objeto de nuestra próxima unidad. Para que se os vaya haciendo la boca agua de lo que se pude lograr con un editor de componentes, os dire que el editor de campos que aparece al hacer doble click en un objeto TTable, o el editor de menús son algunos ejemplos de editores de propiedades... :)

Código fuente de los editores de propiedades.

```
unit Unidad9;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  DsgnIntf, DB, PasswordForm;

type
  TPrueba = class(TComponent)
  private
    FFichero : string;
    FAlias : string;
    FFecha : TDateTime;
    FPassword : string;
  protected
  public
    constructor Create(AOwner : TComponent); override;
  published
    property Fichero : string read FFichero write FFichero;
    property Alias : string read FAlias write FAlias;
    property Fecha : TDateTime read FFecha write FFecha;
    property Password : string read FPassword write FPassword;
  end;
```

```
  TFicheroProperty = class(TStringProperty)
  public
    function GetAttributes : TPropertyAttributes; override;
    procedure Edit; override;
  end;

  TAliasProperty = class (TStringProperty)
  public
    function GetAttributes : TPropertyAttributes; override;
    procedure GetValues(Proc : TGetStrProc); override;
  end;

  TDateTimeProperty = class(TFloatProperty)
    function GetValue : string; override;
    procedure SetValue(const Value  : string); override;
  end;

  TPasswordProperty = class(TPropertyEditor)
    function GetAttributes : TPropertyAttributes; override;
    function GetValue : string; override;
    procedure Edit; override;
  end;

procedure Register;

implementation

constructor TPrueba.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);
  FFecha:=Now;
end;

function TFicheroProperty.GetAttributes : TPropertyAttributes;
begin
  Result:=[paDialog];
end;

procedure TFicheroProperty.Edit;
var
  OpenDialog : TOpenDialog;
begin
  OpenDialog:=TOpenDialog.Create(Application);
  try
    OpenDialog.Filter:='All files|*.*';
    if OpenDialog.Execute then
      SetStrValue(OpenDialog.FileName);
  finally
    OpenDialog.Free;
  end;
end;


function TAliasProperty.GetAttributes : TPropertyAttributes;
begin
    Result:=[paValueList, paSortList];
end;

procedure TAliasProperty.GetValues(Proc : TGetStrProc);
Var
```

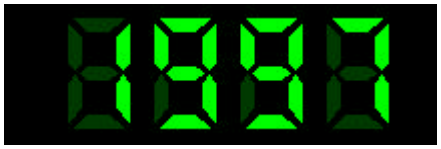
```
  AliasList : TStringList;
  i : integer;
begin
  try
    AliasList := TStringList.Create;
    Session.GetAliasNames(AliasList);
    for i:=0 to AliasList.Count - 1 do
      Proc(AliasList[i]);
  finally
    AliasList.Free;
  end;
end;

function TDateTimeProperty.GetValue : string;
begin
  Result:=DateTimeToStr(GetFloatValue);
end;

procedure TDateTimeProperty.SetValue(const Value : string);
begin
  SetFloatValue(StrToDateTime(Value));
end;

function TPasswordProperty.GetAttributes : TPropertyAttributes;
begin
  Result:=[paDialog];
end;

function TPasswordProperty.GetValue : string;
begin
  Result:=Format('(%s)',[GetPropType^.Name]);
end;

procedure TPasswordProperty.Edit;
begin
  frmPassword := TfrmPassword.Create(Application);
  try
    frmPassword.Caption:=GetComponent(0).Owner.Name+'.'+
      GetComponent(0).Name+'.'+GetName+' - '+
      frmPassword.Caption;
    frmPassword.PW1.Text:=GetStrValue;
    frmPassword.PW2.Text:=frmPassword.PW1.Text;
    if frmPassword.ShowModal = mrOK then
      SetStrValue(frmPassword.PW1.Text)
  finally
    frmPassword.Free;
  end;
end;

procedure Register;
begin
  RegisterComponents('Pruebas', [TPrueba]);

RegisterPropertyEditor(TypeInfo(string),TPrueba,'Fichero',TFicheroProp
erty);

RegisterPropertyEditor(TypeInfo(String),TPrueba,'Alias',TAliasProperty
);

RegisterPropertyEditor(TypeInfo(TDateTime),TPrueba,'Fecha',TDateTimePr
operty);
```

```
RegisterPropertyEditor(TypeInfo(String),TPrueba,'Password',TPasswordPr
operty);
end;

end.



unit PasswordForm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, Buttons;

type
  TfrmPassword = class(TForm)
    lpwd: TLabel;
    lVpwd: TLabel;
    PW1: TEdit;
    PW2: TEdit;
    bOK: TBitBtn;
    bCancel: TBitBtn;
    procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);
  private
  public
  end;

var
  frmPassword: TfrmPassword;

implementation

{$R *.DFM}

procedure TfrmPassword.FormCloseQuery(Sender: TObject;
  var CanClose: Boolean);
begin
  if (ModalResult=mrOK) then
    if (PW1.Text = '') then
    begin
      ShowMessage('Debe introducir una contraseña');
      CanClose:=False;
     end
  else if (ModalResult=mrOK) and (PW1.Text <> PW2.Text) then
   begin
    ShowMessage('Verificación fallida. Por favor reintente');
    CanClose:=False;
   end;
end;

end.
```

Unidad 10. Bitmaps Off-Screen: TDigitalDisplay.

 Si, si ya lo sé. No creais que lo he olvidado. Habíamos quedado que la unidad 10 trataría del tema de los editores de componentes. Lo que pasa es que para explicar este tema... ¡necesitamos un componente apropiado! De modo que esta unidad tratará sobre un nuevo componente sobre el cual explicaremos el tema de los editores de componentes en la unidad 11. Además, así matamos dos pájaros de un sólo tiro. Me explico: últimamente en el grupo de mensajes de Delphi han aparecido varias consultas sobre como evitar el parpadeo que se produce al dibujar imágenes que cambian frecuentemente. Este parpadeo (flicker) se produce al dibujar directamente las imganes en la pantalla. La solución es sencilla: basta con dibujar previamente la imagen en un bitmap oculto (off-screen) y cuando el dibujo este finalizado, volcarlo a la pantalla. De este modo se evita el parpadeo aunque estemos dibujando el gráfico muchas veces por segundo.

Vamos a aplicar esta técnica al componente TDigitalDisplay. Se trata de un display digital tipo calculadora.



Nuestro componente tendrá las siguientes características:

- Será escalable. Es decir, será independiente de la resolución, De este modo tendrá una *buena apariencia* sea cuál sea el tamaño del componente. Este es un aspecto fundamental al hacer un componente gráfico.
- En este componente en concreto es fundamental que este libre de parpadeos, ya que uno de sus usos puede ser de cronómetro, lo cuál implica redibujar el componente una vez por segundo (¡o incluso una vez por centésima de segundo!). Queda claro que no podemos permitirnos ningún tipo de parpadeo, ¿verdad?
- Será fácilmente configurable. Y al decir esto me esto refiriendo a dos aspectos: Por una lado le dotaremos de las suficientes propiedades y eventos para que el usurio pueda configurarlo a su gusto. Por otro lado, este es el típico componente del que luego van a descender otros: el display de una calculadora, el de un lector de discos compactos... Por tanto tenemos que diseñarlo juiciosamente, de forma que un futuro desarrollador pueda elegir lo que le interesa y no le interesa del componente básico.

Antes de ponermos manos a la obra os adelanto que es lo más interesante de este componente y, por tanto, a qué debeis poner más atención ;):

- El manejo de bitmaps ocultos (off-screen)
- La creación de métodos dinámicos que luego un futuro desarrollador pueda reimplementar (override)
- El trabajo con el objeto Canvas.

Consideraciones generales

Comenzemos a diseñar nuestro componente. Nuestro display constará de un número de dígitos configurable por el usuario, por lo que necesitamos una propiedad **Digits** que se encargue de este aspecto. Cada dígito estará formado por siete segmentos (volveremos sobre esto en la siguiente sección). Cada segmento puede estar encendido (propiedad **DigitOnColor**) o apagado (propiedad **DigitOffColor**). Otra propiedad será el color de fondo (propiedad **BckgndColor**).

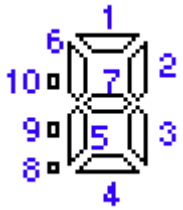
Por supuesto, necesitaremos almacenar el valor a representar (propiedad **Value**) y, además, le dotaremos de alineación (propiedad **Alignment**), de la posibilidad de rellenar con ceros a la izquierda (propiedad **LeadingZeros**) y de estar o no visible (¿adivinais? propiedad **Visible**).

Por último le dotaremos de unos cuantos eventos estándar: OnClick, OnDragOver, OnDragDrop, OnMouseUp... y de un evento personalizado: **OnDigitClick**, que se disparará cuando se haga click sobre un dígito (si se hace click sobre parte del componente que no sea un dígito se disparará el evento OnClick).

Nuestro componente descenderá de **TGraphicControl** y la tárea de dibujado del mismo se llevará a cabo en el método **Paint**, el cuál habrá que redefinir (override). Por cierto ¿es "redefinir" la traducción adecuada de "override"? No estoy seguro, así que, por favor, sacarme de dudas :(

Con esto ya tenemos el marco en el que desarrollaremos el componente, asi que ¡manos a la obra!

Cómo dibujar los digitos



Cómo hemos mencionado en el punto anterior, cada dígito está formado por siete segmentos. En el dibujo se han numerado en color azul. Además de los siete segmentos constituyentes de cada dígito, se han añadido tres segmentos más: uno para representar el punto decimal(.) y otros dos para la separación horaria (:).

El problema principal al dibujar el componente es que nuestro componente tendrá un ancho y alto definidos por el usuario, y estos valores pueden oscilar en un amplio rango de valores. Así el usuario puede querer un display de 100x50 (pequeño) o uno de 600x300 (enorme). Y el componente tiene que tener una buena "apariencia" en ambos casos. ¿Cómo lograrlo? Tenemos dos opciones:

1. Crear un bitmap para cada uno de los dígitos (0..9, . y :) y guardarlo en un fichero de recursos. Después en el método Paint ajustar el tamaño de cada dígito al tamaño real del componente mediante el método StretchDraw. Este es el método más sencillo pero el que da peores resultados, ya que si por ejemplo dibujamos los digitos con un tamaño de 50x100, estos se verán muy mál a tamaños inferiores y a tamaños muy superiores.
2. Dibujarlos a escala. Para cada segmento, guardamos en un array las coordenadas de sus vértices y luego, a la hora de dibujarlo multiplicamos estas coordenadas

por la escala correspondiente. Así, al unir los vértices mediante líneas, la apariencia será perfecta. Este es el método más costoso, pero es el que ofrece mejores resultados y será el que adoptemos.

Si nos fijamos en los segmentos, veremos que los hay de 4,5 y 6 vértices. Por simplicidad haremos que todos tengan 6 vértices para almacenarlos en una matriz de 6 elementos. A los segmentos que les falten vértices, simplemente se los duplicaremos.

Se ha tomado como origen el punto 0,0 y el punto más lejano del origen (el vértice inferior derecho del segmento 3) tiene como coordenadas (26,32), es decir, he dibujado los segmentos sobre una plantilla de 26x34 pixels. Este tamaño es arbitrario, ya que luego cada vértice se le multiplicará por la escala correspondiente. De este modo he definido dos matrices: una para las coordenadass x y otra para las coordenadas y. Por supuesto se podría haber hecho con una sóla matriz, pero como hay que añadirle otra dimensión a la matriz (el indice del segmento) queda un poco más claro así.

```
Const
  {Coordenadas de los puntos}
  {Coordenadas de los 10 (1..10) segmentos posibles: 7 segmentos + 1
punto decimal (.), 2 puntos de hora (:)
   Cada segmento tiene 6 (0..5) vértices}
  {     _   }
  {  . |_|  }
  {  : |_|  }
  Px : Array[1..10,0..5] of integer =
((9,9,24,24,20,13),(26,26,26,24,22,22),(26,26,26,22,22,24),

(24,24,9,9,13,20),(7,7,7,9,11,11),(7,7,7,11,11,9),

(11,13,20,22,20,13),(2,2,5,5,5,2),(2,2,5,5,5,2),(2,2,5,5,5,2)) ;
  Py : Array[1..10,0..5] of integer =
((0,0,0,0,4,4),(2,2,14,16,14,6),(20,20,32,28,20,18),

(34,34,34,34,30,30),(32,32,20,18,20,28),(14,14,2,6,14,16),

(17,15,15,17,19,19),(32,32,32,32,34,34),(22,22,22,22,25,25),(10,10,10,
10,13,13));

  {Para cada digito del 1 al 9, 1 representa segmento encendido, 0
apagado}
  DigitsArray : Array[0..9] of string =
('1111110','0110000','1101101','1111001','0110011','1011011',
    '1011111','1110000','1111111','1111011');

  Seg_dot = 8;     {Segmento correspondiente al .}
  Seg_DotDown = 9; {Segmento correspondiente al punto inferior de :}
  Seg_DotUp = 10;  {Segmento correspondiente al punto superior de :}
```

Hemos definido las dos matrices constantes px y py con las coordenadas de los vértices. Además, se define la matriz DigitsArray de 10 elementos (0..9) que contiene que segmentos están encendidos y apagados para cada dígito. Por ejemplo, el dígito 1 tiene encendidos los segmentos 2y 3 y apagados el resto. Aquí no se tienen en cuenta los segmentos extras (. y :).

Por último a los segmentos correspondientes a el punto decimal y a los puntos de la hora se les asigna un número y una constante para luego poder referirse a ellos con mayor facilidad.

Ya tenemos las bases necesarias para poder pintar el display en el método Paint. Eso sin olvidar que debemos eliminar el parpadeo...

Dibujando sin parpadeos: el método Paint

Cómo en todo componente gráfico el método Paint se encarga de dibujar el componente en pantalla. Recordemos que este método es llamado por Delphi cada vez que se necesita un redibujado del componente, bien sea en respuesta a un mensaje de Windows o a una llamada nuestra, tal y como hacemos en los métodos Set de las diferentes propiedades, al llamar a repaint provocamos el redibujado del componente (Más adelante volveremos sobre este aspecto).

Para evitar el parpadeo no dibujaremos directamente sobre el canvas del componente (esto lo hicimos en el componente TGradiente, unidad 5), sino que lo haremos sobre otro bitmap. el proceso es el siguiente:

1. En el método create creamos nuestro bitmap de trabajo (off-screen):
   FBitmap:=TBitmap.Create;
2. En el método Paint pintamos sobre el Canvas de este bitmap.
3. Una vez terminado el dibujado, volcamos este bitmap sobre el canvas real del componente:
   Canvas.Draw(0,0,FBitmap);
4. Se vuelve al punto 2
5. En el método Destroy liberamos el bitmap de trabajo:
   FBitmap.Free;

Simple ¿verdad? Pues así ¡hemos eliminado el molesto parpadeo de una vez por todas! Ahora bien, quedán un par de aspectos a considerar para que nuestro componente este libre de parpadeo al 100%:

- En el método Create añadimos a la propiedad ControlStyle el atributo csOpaque:
  ControlStyle:=ControlStyle+[csOpaque];
  Esta propiedad esta definida en el objeto TControl e indica los diferentes estilos que se aplican al componente (ver la ayuda en línea para más información).
- Cuando queremos que nuestro componente se repinte, (por ejemplo en los métodos Set de las propiedades) llamamos al método Repaint y no a Invalidate. Repaint **no** borra el area del componente antes de redibujarlo de nuevo (Invalidate si lo hace). Para que repaint nos funcione adecuadamente es por lo que necesitamos que nuestro componente sea opaco (estilo csOpaque).

Ahora si, nuestro componente esta libre de parpadeos al 100% (bueno, quizás exagere un poco y sea realmente al 99%) :) Y si no me creeis, hacer la prueba con el programa mostrado al final de la unidad ¡Incrédulos! ;)

Bien, una vez conocida la teoría, es cuestión de aplicarla. Este es el método **Paint** de nuestro componente:

```
procedure TDigitalDisplay.Paint;
Var
  ex, ey : single;   {Escala a la que dibujar el display}
```

```
  i, Digito : byte;
  Incx, x,y, offsetx, offsety :integer;
  DigitSpace : integer;
  SrcRect : TRect;
  Digitos : byte;
  Caracter : char;
  FillString : string;
begin
  SrcRect:=Rect(0,0,Width,Height);   {Obtenemos el rectángulo del
componente}
  FBitmap.Width:=Width;              {Ajustamos nuestro bitmap}
  FBitmap.Height:=Height;
  {Cálculo del offset a dejar alrededor}
  offsetx:=Trunc(0.2*width);
  offsety:=trunc(0.2*height);
  {Calculamos el espacio disponible para cada dígito}
  DigitSpace:=Trunc((Width-offsetx) / FDigits);
  {Posicionamos la x e y}
  x:=offsetx div 2;
  y:=offsety div 2;
  {Cálculo de la escala}
  ex:=DigitSpace/27;
  ey:=(Height-offsety)/35;
  {Inicializamos el nº de digitos dibujados}
  Digitos:=0;
  {Asignamos colores}
  FBitmap.Canvas.Brush.Color:=FBckgndColor;
  FBitmap.Canvas.FillRect(SrcRect);
  {Asignamos a la variable text el valor a representar}
  Text:=FValue;
  {Creamos una cadena auxiliar de relleno por la izquierda}
  if FLeadingZeros then
    FillString:='00000000000000000000000000000000'
  else
    FillString:='                                ';
  {Rellenar la variable text con ceros o espacios (FillString) según
sea la alineación}
  case FAlignment of
    taRightJustify : Text:=Copy(FillString,1,FDigits-
GetNumberOfDigits)+FValue;
    taCenter : Text:=Copy(FillString,1,(FDigits-GetNumberOfDigits) div
2)+FValue;
  end;
  {Bucle para el dibujo de cada dígito}
  for i:=1 to Length(Text) do
  begin
    Caracter:=Copy(Text,i,1)[1];   {Caracter a dibujar}
    Case caracter of
      '0'..'9': begin
                  Digito:=StrToInt(Caracter);
                  DrawDigit(Digito,x,y,ex,ey);
                  x:=x+DigitSpace;
                  Inc(Digitos);
                  if Digitos = FDigits then break;
                end;
      ':' : begin
              DrawDot(seg_DotUp,x,y,ex,ey);
              DrawDot(seg_DotDown,x,y,ex,ey);
            end;
      '.',',': DrawDot(seg_Dot,x,y,ex,ey);
      ' ':     x:=x+DigitSpace;
```

```
    end;
  end;
  {Una vez dibujado totalmente el display en el bitmap off-screen, lo
volcamos
   al canvas del componente. Resultado ¡No hay parpadeo!}
  Canvas.Draw(0,0,FBitmap);
end;
```

Como se puede ver el método Paint utiliza una serie de rutinas auxiliares para el dibujado del display. Estos métodos son:

- **GetNumberOfDigits**, que se encarga de devolver el número de caracteres numéricos que hay en la propiedad Value. Por ejemplo, si la propiedad contiene el valor 12:25, devolverá 4.

- **DrawDot** es un método que se encarga de dibujar los puntos de separación horaria y decimal.

- **DrawDigit**, que se encarga de dibujar un dígito en la posición correspondiente del Canvas.

Creo que no hace falta explicar cómo se procede al dibujado de cada dígito en el canvas, ya que las funciones utilizadas para ello son de todos bien conocidas (Canvas.Pen, Canvas.Brush y Canvas.Polygon). En cualquier caso si os surgen dudas no teneis más que escribirme :)

Vamos ahora con el último aspecto interesante del componente: hacer que otros desarrolladores puedan descender sus propios componentes de nuestro Display original

Preparando la herencia: métodos dinámicos.

Cuando pretendemos que un componente pueda ser un futuro "padre" de futuros componentes debemos diseñar juiciosamente que aspectos debemos permitir que los hijos reimplementen y que aspectos deben permanecer ocultos a los mismos.

En una primera aproximación queda claro que todos los métodos Set y Get de las propiedades serán privados, ya que los hijos deben acceder a las propiedades directamente, no a través de estos métodos. Por ello declaramos estos métodos en las sección privada de la declaración del componente, lo mismo que los campos "F" asociados a estas propiedades (FBackgndColor, FDigitOnColor...).

Por otra parte, tanto el constructor como el destructor deben ser públicos así que con ellos no hay problema. Pero, ¿qué hacemos con el método Paint? Este método **si** que puede ser redefinido por futuros componentes hijos, por ejemplo, si un desarrollador quiere utilizar otro método para dibujar los digitos. Podríamos dejar el método Paint en la parte privada, pero entonces la única manera que tendría el desarrollador de escribir este nuevo método ¡sería en el componente original! Pero ¿y si no tiene el código fuente, sino tan sólo el fichero .dcu? Pues que no puede hacerlo :(

De modo que no seamos crueles y declaremos el procedimiento Paint como **protected**; así un componente hijo para implementar su propio método de dibujado del display sólo tendrá que escribir:

procedure Paint; override;

Que, dicho sea de paso, es lo mismo que hemos tenido que hacer nosotros :) Y es que no olvidemos que nuestro componente, a su vez, es hijo de TGraphicControl y ¡gracias a Dios! en dicho componente el método Paint se declaro protected y no private ;)

Nos queda por implementar un evento, *OnDigitClick* que se debe disparar cuando el usuario del componente haga click sobre un digito del display. Hay que distinguir entre el click sobre un dígito (que disparará el evento *OnDigitClick*) y el click sobre el resto del display (que disparará el evento *OnClick*).

Lo primero es declarar un campo FOnDigitClick para el nuevo evento. Además declararemos el tipo de procedimiento del evento:

TOnDigitClick = procedure(Sender : TObject; Digit : integer) of object;

Para disparar este evento, redefiniremos el procedimiento **MouseDown**. Para ello declararemos el método en la parte protected (ya sabeis, para que los hijos...) y añadiremos la palabra override. Esta es la implementación de dicho método:

```
procedure TDigitalDisplay.MouseDown(Button : TMouseButton; Shift :
TShiftState; X,Y : Integer);
{Método que se encarga de determinar si se ha pulsado con el ratón
sobre un dígito del display.
 En caso afirmativo se dispara el evento correspondiente}
var
  i, xt, xoff, yoff, DigitSpace : integer;
begin
  inherited MouseDown(Button,Shift,X,Y);
  {Cálculo del offset horizontal}
  xoff:=Trunc(0.2*width);
  xt:=Trunc(0.2*width) div 2;
  {Cálculo del espacio ocupado por un dígito}
  DigitSpace:=Trunc((Width-xoff) / FDigits);
  {Vamos comprobando si el click se ha ha hecho sobre el dígito (i)}
  for i:=1 to FDigits do
  begin
    if (x>=xt) AND (x<=xt+DigitSpace) then
      DigitClick(i);
    xt:=xt+DigitSpace;
  end;
end;
```

Lo primero que hace el método es llamar al antecesor del mismo mediente la llamada inherited. A continuación se calcula si se ha hecho click sobre un dígito. En caso afirmativo se ejecuta el procedimiento *DigitClick* pasándole como parámetro el número de dígito sobre el que se ha hecho click. El método *DigitClick*, por su parte, se encarga de disparar el evento anteriormente definido:

```
procedure TDigitalDisplay.DigitClick(Digit : integer);
{Método que dispara el evento OnDigitClick}
begin
  if Assigned(FOnDigitClick) then
    FOnDigitClick(Self,Digit);
end;
```

Y aquí esta el meollo de la cuestión. ¿Por qué hemos definido el método *DigitClick* y no hemos escrito las sentencias correspondientes dentro del método *MouseDown*? La

respuesta es reusabilidad. Si un usuario del componente quiere que en respuesta a un click sobre un dígito dicho dígito cambie, codificara las sentencias necesarias en respuesta al evento OnDigitClick. Pero si es un nuevo programador el que esta desarrollando un hijo de TDigitalDisplay no puede hacer esto. A este programador el procedimiento MouseDown le sirve perfectamente y tan sólo tiene que reimplementar el procedimiento DigitClick y poner allí lo que quiera. Si lo hubieramos codificado todo en un único procedimiento esto no sería posible. De ahí que mantengamos separados el procedimiento que disparará el evento y el disparo del evento en sí. De este modo aseguramos una gran flexibilidad a futuros desarrolladores: por ejemplo, un hijo reimplementara DigitClick para incrementar el valor del dígito (por ejemplo, para ajustar la hora), mientras que otro lo reimplementara para poner el valor a cero (en un cronómetro). En ambos casos el método MouseDown será el mismo y bastará con trabajar sobre DigitClick directamente.

El último detalle importante es que **el método DigitClick se ha declarado dynamic**, ya que si no se declara así, no se puede reimplementar, ya que los métodos estáticos no se pueden redefinir, sólo los dinámicos (ya sean dynamic o virtual). En otras unidades profundizaremos más sobre este tema. Mientras os refiero a la ayuda en línea de Delphi para más información sobre este tema.

Con esto queda terminado el componente. Quizás haya sido una explicación un poco breve pero considero que con el nivel que ya tenemos no debeis teneis problema para entender el funcionamiento del mismo. Os recomiendo que estudies la forma de trabajar con off-screen bitmaps, ya que es un tema que se utiliza frecuentemente en la programación gráfica bajo Windows. Y Recordar que este componente nos servira en la próxima unidad para desarrollar el tema de los editores de propiedades.

Código fuente del componente.

```
unit DigitalDisplay;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs;

type
  {Tipo para el evento OnDigitClick}
  TOnDigitClick = procedure(Sender : TObject; Digit : integer) of
object;

  TDigitalDisplay = class(TGraphicControl)
  private
    FBitmap : TBitmap;             {Bitmap oculto (off-screen) para el
dibujo del display}
    FDigits : byte;                {Número de digitos a representar}
    FValue : string;               {Valor a arepresentar}
    FBckgndColor, FDigitOnColor, FDigitOffColor : TColor;   {Colores}
    FAlignment : TAlignment;       {Alineación horizontal}
    FLeadingZeros : boolean;       {Rellenar con ceros}
    FOnDigitClick : TOnDigitClick; {Campo para el evento OnDigitClick}

    procedure DrawDot(index : byte;x,y : integer;ex,ey : single);
```

```
    procedure DrawDigit(Digit : byte; x,y : integer;ex,ey : single);
    procedure SetDigits(Value:byte);
    procedure SetValue(Value:String);
    procedure SetBckgndColor(Value : TColor);
    procedure SetDigitOnColor(Value : TColor);
    procedure SetDigitOffColor(Value : TColor);
    procedure SetAlignment(Value : TAlignment);
    procedure SetLeadingZeros(Value : boolean);
    function GetNumberOfDigits : byte;
  protected
    procedure Paint; override;
    procedure MouseDown(Button : TMouseButton; Shift : TShiftState;
X,Y : Integer); override;
    procedure DigitClick(Digit : integer); dynamic;
  public
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
  published
    property LeadingZeros : boolean read FLeadingZeros write
SetLeadingZeros default False;
    property Alignment : TAlignment read FAlignment write SetAlignment
default taRightjustify;
    property Digits : byte read FDigits write SetDigits default 4;
    property Value : string read FValue write SetValue;
    property BckgndColor : TColor read FBckgndColor write
SetBckgndColor;
    property DigitOnColor : TColor read FDigitOnColor write
SetDigitOnColor;
    property DigitOffColor : TColor read FDigitOffColor write
SetDigitOffColor;
    property OnDigitClick : TOnDigitClick read FOnDigitClick write
FOnDigitClick;
    property Visible;
    property OnClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;
    property OnMouseMove;
    property OnMouseUp;
  end;

procedure Register;

implementation

Const
  {Coordenadas de los puntos}
  {Coordenadas de los 10 (1..10) segmentos posibles: 7 segmentos + 1
punto decimal (.), 2 puntos de hora (:)
   Cada segmento tiene 6 (0..5) vértices}
  {     _   }
  {  . |_|  }
  {  : |_|  }
  Px : Array[1..10,0..5] of integer =
((9,9,24,24,20,13),(26,26,26,24,22,22),(26,26,26,22,22,24),

(24,24,9,9,13,20),(7,7,7,9,11,11),(7,7,7,11,11,9),

(11,13,20,22,20,13),(2,2,5,5,5,2),(2,2,5,5,5,2),(2,2,5,5,5,2)) ;
```

```
  Py : Array[1..10,0..5] of integer =
((0,0,0,0,4,4),(2,2,14,16,14,6),(20,20,32,28,20,18),

(34,34,34,34,30,30),(32,32,20,18,20,28),(14,14,2,6,14,16),

(17,15,15,17,19,19),(32,32,32,32,34,34),(22,22,22,22,25,25),(10,10,10,
10,13,13));

  {Para cada digito del 1 al 9, 1 representa segmento encendido, 0
apagado}
  DigitsArray : Array[0..9] of string =
('1111110','0110000','1101101','1111001','0110011','1011011',
    '1011111','1110000','1111111','1111011');

  Seg_dot = 8;     {Segmento correspondiente al .}
  Seg_DotDown = 9; {Segmento correspondiente al punto inferior de :}
  Seg_DotUp = 10;  {Segmento correspondiente al punto superior de :}

constructor TDigitalDisplay.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);
  ControlStyle:=ControlStyle+[csOpaque];  {Necesario para evitar el
parpadeo}
  FBitmap:=TBitmap.Create;                {Creamos el bitmap que no
servirá para dibujar el display off-screen}
  {Valores por defecto}
  FDigits:=4;
  FValue:='1997';
  FBckgndColor:=clBlack;
  FDigitOnColor:=clLime;
  FDigitOffColor:=$4000;
  FAlignment:=taRightJustify;
  Width:=108;
  Height:=35;
end;

destructor TDigitalDisplay.Destroy;
begin
  inherited Destroy;
  FBitmap.Free;        {Liberamos el bitmap off-screen}
end;

procedure TDigitalDisplay.SetAlignment(Value : TAlignment);
begin
  if Value<>FAlignment then
  begin
    FAlignment:=Value;
    repaint;
  end;
end;

procedure TDigitalDisplay.SetDigits(Value : byte);
begin
  if Value<>FDigits then
  begin
    FDigits:=Value;
    repaint;
  end;
end;

procedure TDigitalDisplay.SetLeadingZeros(Value : boolean);
```

```
begin
  if Value<>FLeadingZeros then
  begin
    FLeadingZeros:=Value;
    repaint;
  end;
end;

procedure TDigitalDisplay.SetValue(Value : string);
begin
  if Value<>FValue then
  begin
    FValue:=Value;
    repaint;
  end;
end;

procedure TDigitalDisplay.SetBckgndColor(Value : TColor);
begin
  if FBckgndColor<>Value then
  begin
    FBckgndColor:=Value;
    repaint;
  end;
end;

procedure TDigitalDisplay.SetDigitOnColor(Value : TColor);
begin
  if FDigitOnColor<>Value then
  begin
    FDigitOnColor:=Value;
    repaint;
  end;
end;

procedure TDigitalDisplay.SetDigitOffColor(Value : TColor);
begin
  if FDigitOffColor<>Value then
  begin
    FDigitOffColor:=Value;
    repaint;
  end;
end;

procedure TDigitalDisplay.Paint;
Var
  ex, ey : single;   {Escala a la que dibujar el display}
  i, Digito : byte;
  Incx, x,y, offsetx, offsety :integer;
  DigitSpace : integer;
  SrcRect : TRect;
  Digitos : byte;
  Caracter : char;
  FillString : string;
begin
  SrcRect:=Rect(0,0,Width,Height);   {Obtenemos el rectángulo del
componente}
  FBitmap.Width:=Width;              {Ajustamos nuestro bitmap}
  FBitmap.Height:=Height;
  {Cálculo del offset a dejar alrededor}
  offsetx:=Trunc(0.2*width);
```

```
  offsety:=trunc(0.2*height);
  {Calculamos el espacio disponible para cada dígito}
  DigitSpace:=Trunc((Width-offsetx) / FDigits);
  {Posicionamos la x e y}
  x:=offsetx div 2;
  y:=offsety div 2;
  {Cálculo de la escala}
  ex:=DigitSpace/27;
  ey:=(Height-offsety)/35;
  {Inicializamos el nº de digitos dibujados}
  Digitos:=0;
  {Asignamos colores}
  FBitmap.Canvas.Brush.Color:=FBckgndColor;
  FBitmap.Canvas.FillRect(SrcRect);
  {Asignamos a la variable text el valor a representar}
  Text:=FValue;
  {Creamos una cadena auxiliar de relleno por la izquierda}
  if FLeadingZeros then
    FillString:='00000000000000000000000000000000'
  else
    FillString:='                                ';
  {Rellenar la variable text con ceros o espacios (FillString) según
sea la alineación}
  case FAlignment of
    taRightJustify : Text:=Copy(FillString,1,FDigits-
GetNumberOfDigits)+FValue;
    taCenter : Text:=Copy(FillString,1,(FDigits-GetNumberOfDigits) div
2)+FValue;
  end;
  {Bucle para el dibujo de cada dígito}
  for i:=1 to Length(Text) do
  begin
    Caracter:=Copy(Text,i,1)[1];   {Caracter a dibujar}
    Case caracter of
      '0'..'9': begin
                  Digito:=StrToInt(Caracter);
                  DrawDigit(Digito,x,y,ex,ey);
                  x:=x+DigitSpace;
                  Inc(Digitos);
                  if Digitos = FDigits then break;
                end;
      ':' : begin
              DrawDot(seg_DotUp,x,y,ex,ey);
              DrawDot(seg_DotDown,x,y,ex,ey);
            end;
      '.',',': DrawDot(seg_Dot,x,y,ex,ey);
      ' ':     x:=x+DigitSpace;
    end;
  end;
  {Una vez dibujado totalmente el display en el bitmap off-screen, lo
volcamos
   al canvas del componente. Resultado ¡No hay parpadeo!}
  Canvas.Draw(0,0,FBitmap);
end;

procedure TDigitalDisplay.DrawDot(index : byte;x,y : integer;ex,ey :
single);
{Dibuja un punto en la posición x,y con la escala determinada por ex y
ey}
Var
  j : byte;
```

```
  Puntos : array[0..5] of TPoint;
begin
  FBitmap.Canvas.Pen.Color:=DigitOnColor;
  FBitmap.Canvas.Brush.Color:=DigitOnColor;
  for j:=0 to 5 do
    Puntos[j]:=Point(x+Trunc(Px[index,j]*ex),y+Trunc(Py[index,j]*ey));
  FBitmap.Canvas.Polygon([Puntos[0],Puntos[1],Puntos[2],
                  Puntos[3],Puntos[4],Puntos[5]]);
end;

procedure TDigitalDisplay.DrawDigit(Digit : byte; x,y : integer;ex,ey
: single);
{Dibuja el dígito (0..9) pasado en la posición x,y con la escala
determinada por ex y ey}
Var
  i, j : byte;
  Puntos : array[0..5] of TPoint;
begin
  for i:=1 to 7 do
  begin
    if Copy(DigitsArray[Digit],i,1) = '0' then
    begin
      FBitmap.Canvas.Pen.Color:=FDigitOffColor;
      FBitmap.Canvas.Brush.Color:=FDigitOffColor;
    end
    else
    begin
      FBitmap.Canvas.Pen.Color:=FDigitOnColor;
      FBitmap.Canvas.Brush.Color:=FDigitOnColor;
    end;
    for j:=0 to 5 do
        Puntos[j]:=Point(x+Trunc(Px[i,j]*ex),y+Trunc(Py[i,j]*ey));
    FBitmap.Canvas.Polygon([Puntos[0],Puntos[1],Puntos[2],
                      Puntos[3],Puntos[4],Puntos[5]]);
  end;
end;

function TDigitalDisplay.GetNumberOfDigits : byte;
{Devuelve el número de digitos numéricos de la propiedad Value}
Var
  i : byte;
begin
  Result:=0;
  for i:=1 to Length(FValue) do
    if Copy(FValue,i,1)[1] in ['0'..'9'] then
      Inc(Result);
end;

procedure TDigitalDisplay.MouseDown(Button : TMouseButton; Shift :
TShiftState; X,Y : Integer);
{Método que se encarga de determinar si se ha pulsado con el ratón
sobre un dígito del display.
 En caso afirmativo se dispara el evento correspondiente}
var
  i, xt, xoff, yoff, DigitSpace : integer;
begin
  inherited MouseDown(Button,Shift,X,Y);
  {Cálculo del offset horizontal}
  xoff:=Trunc(0.2*width);
  xt:=Trunc(0.2*width) div 2;
  {Cálculo del espacio ocupado por un dígito}
```

```
  DigitSpace:=Trunc((Width-xoff) / FDigits);
  {Vamos comprobando si el click se ha ha hecho sobre el dígito (i)}
  for i:=1 to FDigits do
  begin
    if (x>=xt) AND (x<=xt+DigitSpace) then
      DigitClick(i);
    xt:=xt+DigitSpace;
  end;
end;

procedure TDigitalDisplay.DigitClick(Digit : integer);
{Método que dispara el evento OnDigitClick}
begin
  if Assigned(FOnDigitClick) then
    FOnDigitClick(Self,Digit);
end;

procedure Register;
begin
  RegisterComponents('Curso', [TDigitalDisplay]);
end;


end.
```

Ejemplo de uso del componente.

Como ejemplo de uso del componente y para demostraros que está realmente libre de flickering, en un form en blanco coloca un componente TDigitalDisplay y un botón. En respuesta al evento OnClick del botón codifica lo siguiente:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  contador : integer;
begin
  for contador:=0 to 1000 do
  begin
    DigitalDisplay1.Value:=IntToStr(Contador);
    Application.ProcessMessages;
  end;
end;
```

¡Ejecutar el programa y disfrutar! ;)

Unidad 11. Editores de Componentes.

Introducción

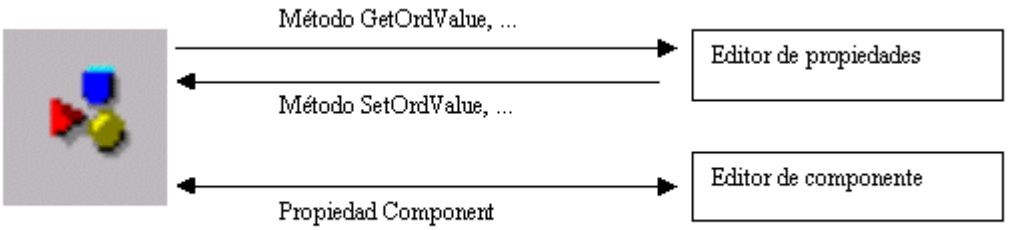
En esta unidad aprenderemos a crear editores de componentes. Comenzaremos viendo las similitudes y diferencias existentes entre editores de propiedades y editores de componentes y, a continuación, desarrollaremos un ejemplo práctico: un editor de componentes para el componente (valga la redundancia) creado en la unidad anterior, TDigitalDisplay.

Diferencias entre editores de propiedades y editores de componentes

Como vimos en las unidades 8 y 9, un editor de propiedades trabaja sobre la representación en forma de string del valor de una propiedad de un componente determinado. El objetivo de esta manipulación es permitir al usuario del componente leer y almacenar el valor de la propiedad que está siendo editada. Adicionalmente un editor de propiedades puede transformar el valor de la propiedad convirtiéndola a un valor más apropiado para el funcionamiento interno del componente. Un ejemplo es el editor que construimos para las propiedades de tipo de fecha, que muestra al usuario la fecha en formato "legible" (dd/mm/aa), pero que opera internamente con un float (TDateTime).

Pero la propia filosofía de diseño de los editores de propiedades trae consigo una ventaja que es a su vez una desventaja: un editor de propiedades sólo conoce del componente la propiedad a editar y nada más. Recordemos que un editor de propiedades leía y almacenaba el valor de la propiedad por medio de los métodos GetOrdValue, SetOrdValue, GetFloatValue, etc. Estos métodos proporcionan el único medio de comunicación que el editor tiene con el componente. Un editor de componentes, en cambio, puede acceder al componente directamente, ya que una de las propiedades que todo editor de componentes tiene es la propiedad Component, que devuelve una referencia al componente que está siendo editado. De este modo, el editor de componentes puede acceder a todos los métodos y propiedades que el componente define como published o public (e incluso a los definidos como private o protected sí el editor reside en la misma unidad que el componente).

Por tanto, si queremos manipular una única propiedad de un componente utilizaremos un editor de propiedades. Si, por el contrario, queremos manipular varias propiedades al mismo tiempo, o ejecutar algún método del componente, deberemos utilizar un editor de componentes. Además, los editores de componentes permiten definir nuevas acciones en forma de menú ítems que se añaden al menú contextual que aparece al seleccionar un componente y hacer click sobre él con el botón derecho del ratón (p.e. el editor de componente TPageControlEditor permite añadir/eliminar nuevas páginas del componente TPageControl). Otra posibilidad que permiten los editores de componentes es ejecutar una acción determinada cuando el usuario hace doble click sobre un componente determinado (p.e. el editor de componentes TMenuEditor ejecuta el editor de menús del componente TMainMenu). A modo de resumen la figura siguiente muestra como se comunican los editores de propiedades y de componentes con el componente a editar:



Estructura de un editor de componentes

Todos los editores de componentes descienden de una clase base denominada TComponentEditor. La declaración de dicha clase es la siguiente:

```
TComponentEditor = class
private
  FComponent: TComponent;
  FDesigner: TFormDesigner;
public
  constructor Create(AComponent: TComponent; ADesigner:
TFormDesigner); virtual;
  procedure Edit; virtual;
  procedure ExecuteVerb(Index: Integer); virtual;
  function GetVerb(Index: Integer): string; virtual;
  function GetVerbCount: Integer; virtual;
  procedure Copy; virtual;
  property Component: TComponent read FComponent;
  property Designer: TFormDesigner read FDesigner;
end;
```

Vamos a ver con más detenimiento las propiedades y métodos declarados en esta clase:

- *property Component*. Propiedad de sólo lectura. Devuelve una referencia al componente que está siendo editado. Esta propiedad es fundamental en todo editor de componentes, ya que gracias a ella podemos asignar/recuperar los valores de las distintas propiedades del componente. Conviene hacer notar que es del tipo TComponent, por lo que tendremos que hacer un casting al tipo de componente específico.

- *property Designer*. Propiedad de sólo lectura. Devuelve una referencia a Diseñador de Forms de Delphi. Por lo general la utilizaremos para comunicar a Delphi que el componente ha cambiado.

- *procedure Create.* Aquí hay poco que decir, ya que este método es un viejo conocido de todos nosotros. ;-)

- *procedure GetVerbCount, GetVerb y ExecuteVerb*. Cuando se selecciona un componente y se hace click sobre él con botón derecho, Delphi llama al método **GetVerbCount** para ver si hay que añadir entradas al menú contextual del componente. Este menú debe devolver el número de entradas (ítems) a añadir al mencionado menú. Si este método devuelve un valor distinto de cero, Delphi llamará al método **GetVerb** tantas veces como entradas se deban añadir. Este método debe devolver el string que se debe visualizar en el menú por cada elemento añadido. Para ello Delphi le pasa el índice del elemento a obtener el string (siendo el primero el que tiene el índice 0). Por último, cuando el usuario hace click sobre una entrada de menú Delphi llama al método **ExecuteVerb**, pasándole como parámetro el índice del menú ítem seleccionado. En este método se debe codificar cualquier acción que deba llevar a cabo el editor del componente (p.e. mostrar un form auxiliar para editar las propiedades del componente).

- *procedure Edit.* Este método se invoca cuando el usuario hace doble click sobre el componente y es equivalente a la acción ExecuteVerb(0). Es decir, hacer

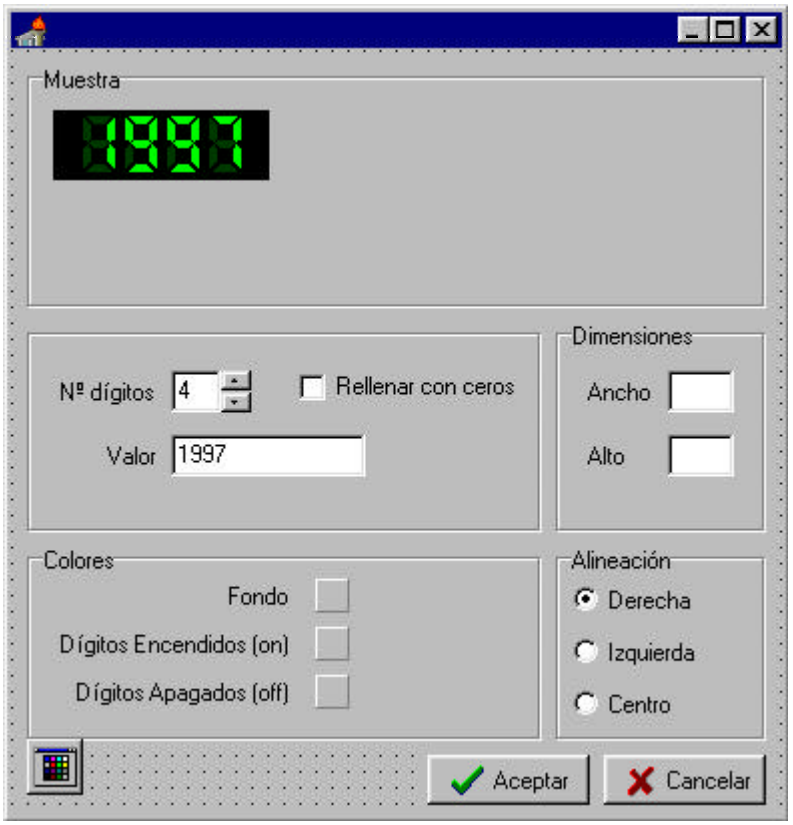
doble click sobre el componente implica ejecutar el primer ítem del menú contextual.

No parece complicado, ¿verdad? De hecho, es más sencillo escribir editores de componentes que editores de propiedades. Y vamos a demostrarlo escribiendo nuestro primer editor de componentes.

Un editor para el componente TDigitalDisplay

En la unidad anterior desarrollamos el componente TDigitalDisplay, al cual dotamos de una serie de propiedades (color de los dígitos, color de fondo, alineación...) que lo hacen ampliamente configurable para el usuario. Esta profusión de posibilidades puede hacer difícil al usuario la elección de valores adecuados para que el aspecto del componente sea "estéticamente bonito" :-) Aquí es donde un editor de componentes puede ayudar: en vez de obligar al usuario a estar continuamente jugando con las distintas propiedades hasta encontrar una combinación satisfactoria, mediante un form auxiliar podemos mostrar al usuario de forma inmediata los cambios que vaya realizando al componente. Una vez que el usuario elija la apariencia que desea darle, pulsa sobre OK y el componente real adoptará la forma deseada. Si al usuario no le convence la apariencia, pulsa Cancel y se olvida de tener que ir propiedad por propiedad asignando los valores antiguos. De este modo es mucho más sencillo el trabajo en tiempo de diseño con el componente y todos contentos ¿verdad? ;-) Manos a la obra entonces.

Comencemos por diseñar el form auxiliar que servirá para que el usuario "juegue" con las propiedades del componente:



Como se puede observar en la figura, el form consta de un componente TDigitalDisplay utilizado para mostrar los diversos cambios efectuados a las propiedades, y una serie de controles que permiten alterar el valor de dichas propiedades.

La construcción de este form es muy sencilla, por lo que muestro directamente el código fuente del mismo:

```
type
  TfrmDigitalDisplay = class(TForm)
    gbMuestra: TGroupBox;
    DigitalDisplay1: TDigitalDisplay;
    rgAlineacion: TRadioGroup;
    gbValor: TGroupBox;
    lValor: TLabel;
    eValor: TEdit;
    cbRellenarCeros: TCheckBox;
    eDigitos: TEdit;
    lDigitos: TLabel;
    upDigitos: TUpDown;
    gbColores: TGroupBox;
    lFondo: TLabel;
    lDigitosOn: TLabel;
    lDigitosOff: TLabel;
    pColorFondo: TPanel;
    pColorOn: TPanel;
    pColorOff: TPanel;
    bbAceptar: TBitBtn;
    bbCancelar: TBitBtn;
    ColorDialog1: TColorDialog;
    gbDimensiones: TGroupBox;
    lAncho: TLabel;
    eAncho: TEdit;
    lAlto: TLabel;
    eAlto: TEdit;
    procedure pColorFondoClick(Sender: TObject);
    procedure pColorOnClick(Sender: TObject);
    procedure pColorOffClick(Sender: TObject);
    procedure rgAlineacionClick(Sender: TObject);
    procedure eDigitosChange(Sender: TObject);
    procedure upDigitosClick(Sender: TObject; Button: TUDBtnType);
    procedure cbRellenarCerosClick(Sender: TObject);
    procedure eValorChange(Sender: TObject);
    procedure eAnchoChange(Sender: TObject);
    procedure eAltoChange(Sender: TObject);
  private
  public
  end;
...
implementation
...
{--------------------------------------}
{ Form del Editor de propiedades       }
{--------------------------------------}

procedure TfrmDigitalDisplay.pColorFondoClick(Sender: TObject);
begin
  ColorDialog1.Color:=pColorFondo.Color;
  if ColorDialog1.Execute then
    pColorFondo.Color:=ColorDialog1.Color;
  DigitalDisplay1.BckgndColor:=pColorFondo.Color;
```

```
end;

procedure TfrmDigitalDisplay.pColorOnClick(Sender: TObject);
begin
  ColorDialog1.Color:=pColorOn.Color;
  if ColorDialog1.Execute then
    pColorOn.Color:=ColorDialog1.Color;
  DigitalDisplay1.DigitOnColor:=pColorOn.Color;
end;

procedure TfrmDigitalDisplay.pColorOffClick(Sender: TObject);
begin
  ColorDialog1.Color:=pColorOff.Color;
  if ColorDialog1.Execute then
    pColorOff.Color:=ColorDialog1.Color;
  DigitalDisplay1.DigitOffColor:=pColorOff.Color;
end;

procedure TfrmDigitalDisplay.rgAlineacionClick(Sender: TObject);
begin
  case rgAlineacion.ItemIndex of
    0 : DigitalDisplay1.Alignment:=taRightJustify;
    1 : DigitalDisplay1.Alignment:=taLeftJustify;
    2 : DigitalDisplay1.Alignment:=taCenter;
  end;
end;

procedure TfrmDigitalDisplay.eDigitosChange(Sender: TObject);
begin
  DigitalDisplay1.Digits:=StrToInt(eDigitos.Text);
end;

procedure TfrmDigitalDisplay.upDigitosClick(Sender: TObject; Button:
TUDBtnType);
begin
  DigitalDisplay1.Digits:=StrToInt(eDigitos.Text);
end;

procedure TfrmDigitalDisplay.cbRellenarCerosClick(Sender: TObject);
begin
  DigitalDisplay1.LeadingZeros:=cbRellenarCeros.Checked;
end;

procedure TfrmDigitalDisplay.eValorChange(Sender: TObject);
begin
  DigitalDisplay1.Value:=eValor.Text;
end;

procedure TfrmDigitalDisplay.eAnchoChange(Sender: TObject);
begin
  DigitalDisplay1.Width:=StrToInt(eAncho.Text);
end;

procedure TfrmDigitalDisplay.eAltoChange(Sender: TObject);
begin
  DigitalDisplay1.Height:=StrToInt(eAlto.Text);
end;
```

Una vez diseñado el form, vamos a crear el editor de componentes propiamente dicho.

Implementado el editor

Nuestro editor de componentes deriva directamente de TComponentEditor. La declaración del mismo es la siguiente:

```
  TDigitalDisplayEditor = class(TComponentEditor)
    function GetVerbCount : integer; override;
    function GetVerb(Index : integer) : string; override;
    procedure ExecuteVerb(Index : integer); override;
    procedure PrepararForm(aForm : TfrmDigitalDisplay);
    procedure ActualizarComponente(aForm : TfrmDigitalDisplay);
  end;
```

Por convención, los editores de componentes finalizan con la palabra Editor (TTableEditor, TPageControlEditor, ...)

Nuestro editor añadirá una única opción al menú contextual del componente. Dicha opción se activará cuando el usuario seleccione la opción o haga doble click sobre el componente, tal y cómo se explico anteriormente. Por tanto definimos el método GetVerbCount de la siguiente manera:

```
function TDigitalDisplayEditor.GetVerbCount : integer;
begin
  Result:=1;
end;
```

Nuestro método se limita a comunicar que sólo se añade un ítem al menú contextual. A continuación indicamos que el ítem a añadir debe tener la leyenda "&Edit"

```
function TDigitalDisplayEditor.GetVerb(Index : integer) : string;
begin
  Result:='E&dit...';
end;
```

Y ahora la parte más interesante. El método ExecuteVerb se debe encargar de las siguientes tareas:

1. Crear el form diseñado en la sección anterior.
2. Asignar a los controles del form los valores de las propiedades que actualmente tiene el componente real (entendiendo por componente real el que el usuario ha seleccionado de la paleta de componente y ha pinchado sobre el form correspondiente).
3. Mostrar, de forma modal, el form de edición.
4. Si el usuario sale del form aceptando los cambios (OK) se debe actualizar el componente real con los valores que el usuario ha elegido en el form.
5. Debemos comunicar a Delphi que el componente ha cambiado. Este paso es **muy importante**. Para ello, ejecutamos la sentencia **Designer.Modified**
6. Liberar el form.

Todos estos pasos son típicos sea cuál sea el editor de componentes que estemos desarrollando, de forma que os recomiendo que os familiaricéis con ellos. Pos supuesto,

si no se necesita el form auxiliar, se obvian los pasos correspondientes ;) La secuencia mostrada queda codificada en nuestro editor en concreto de la siguiente manera:

```
procedure TDigitalDisplayEditor.ExecuteVerb(Index : integer);

  procedure CopyDigitalDisplay(Dest, Source : TDigitalDisplay);
  {procedimiento auxiliar que copia los valores de las propiedades de
un
   componente digital display a otro}
  begin
    Dest.Digits:=Source.Digits;
    Dest.LeadingZeros:=Source.LeadingZeros;
    Dest.Value:=Source.Value;
    Dest.Alignment:=Source.Alignment;
    Dest.BckgndColor:=Source.BckgndColor;
    Dest.DigitOnColor:=Source.DigitOnColor;
    Dest.DigitOffColor:=Source.DigitOffColor;
    Dest.Width:=Source.Width;
    Dest.Height:=Source.Height;
  end;

Var
  aForm : TfrmDigitalDisplay;
begin
  {Creamos el form del editor de propiedades}
  aForm:=TfrmDigitalDisplay.Create(Application);
  try
    {Asignamos el caption correspondiente}
    aForm.Caption:=Component.Owner.Name+'.'+Component.Name+'
'+aForm.Caption;
    {Asignamos a los controles del form los valores correspondientes
del componente
     a editar}
    PrepararForm(aForm);
    {Actualizamos el DigitalDisplay utilizado como muestra para que
presente el mismo
     aspecto que tiene el componente real}
    CopyDigitalDisplay(aForm.DigitalDisplay1, Component As
TDigitalDisplay);
    {Si el usuario acepta los cambios efectuados al form}
    if aForm.ShowModal = mrOK then
    begin
      {Actualizamos el componente en base a los controles del form}
      ActualizarComponente(aForm);
      {Informamos a Delphi que el componente ha cambiado}
      Designer.Modified;
    end;
  finally
    {Liberamos el form}
    aForm.Free;
  end;
end;
```

Cabe destacar lo siguiente:

- El procedimiento PrepararForm se encarga de asignar a los controles del form los valores correspondientes de las propiedades del componente a editar. Para ello utiliza el método CopyDigitalDisplay. De una forma análoga, el método ActualizarComponente se encarga de asignar al componente real los valores del

form auxiliar.
Nota: Nos podríamos haber ahorrado el método ActualizarComponente y haber asignado al componente real los valores de las propiedades de los controles del form mediante el método CopyDigitalDisplay. Si no lo he hecho así, es porque creo que queda más claro tener dos métodos: uno para asignar los valores a los controles del form y otro para asignarlos al componente real, pero es simplemente cuestión de gustos. :-)

- Vuelvo a recalcar aquí que es indispensable informar a Delphi que se han actualizado las propiedades del componente mediante una llamada al método Modified del objeto Designer. En caso contrario Delphi no actualizaría en el inspector de objetos los cambios realizados.

Registro del editor de componentes y consideraciones finales.

Nos quedan por ver un par de detalles. El primero es el registro del editor de componentes. Para ello se debe ejecutar, dentro del procedimiento Register, la sentencia **RegisterComponentEditor**. Este método toma dos parámetros, el primero es la referencia la clase del componente al que se debe asociar el editor, mientras que el segundo especifica el nombre del editor. En nuestro caso lo registramos así:

```
procedure Register;
{Registro del editor de componentes}
begin
  RegisterComponentEditor(TDigitalDisplay,TDigitalDisplayEditor);
end;
```

El segundo aspecto a tratar es dónde situar el editor de componentes. Aquí se aplican los mismos principios que explicamos en la primera unidad en que estudiamos los editores de propiedades. Ya que los editores de componentes sólo se necesitan en tiempo de diseño no se deben situar en la misma unidad en que resida el componente, sino en una unidad independiente. Lo que sí se suele hacer es situarlo en la misma unidad donde reside el form auxiliar, tal y como lo hacemos nosotros con nuestro editor.

Código fuente completo del editor de componente

```
unit DigitalDisplayEditor;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, Buttons, ExtCtrls, ComCtrls, DigitalDisplay, DsgnIntf;

type
  TfrmDigitalDisplay = class(TForm)
    gbMuestra: TGroupBox;
    DigitalDisplay1: TDigitalDisplay;
    rgAlineacion: TRadioGroup;
    gbValor: TGroupBox;
    lValor: TLabel;
```

```
    eValor: TEdit;
    cbRellenarCeros: TCheckBox;
    eDigitos: TEdit;
    lDigitos: TLabel;
    upDigitos: TUpDown;
    gbColores: TGroupBox;
    lFondo: TLabel;
    lDigitosOn: TLabel;
    lDigitosOff: TLabel;
    pColorFondo: TPanel;
    pColorOn: TPanel;
    pColorOff: TPanel;
    bbAceptar: TBitBtn;
    bbCancelar: TBitBtn;
    ColorDialog1: TColorDialog;
    gbDimensiones: TGroupBox;
    lAncho: TLabel;
    eAncho: TEdit;
    lAlto: TLabel;
    eAlto: TEdit;
    procedure pColorFondoClick(Sender: TObject);
    procedure pColorOnClick(Sender: TObject);
    procedure pColorOffClick(Sender: TObject);
    procedure rgAlineacionClick(Sender: TObject);
    procedure eDigitosChange(Sender: TObject);
    procedure upDigitosClick(Sender: TObject; Button: TUDBtnType);
    procedure cbRellenarCerosClick(Sender: TObject);
    procedure eValorChange(Sender: TObject);
    procedure eAnchoChange(Sender: TObject);
    procedure eAltoChange(Sender: TObject);
  private
  public
  end;

  TDigitalDisplayEditor = class(TComponentEditor)
    function GetVerbCount : integer; override;
    function GetVerb(Index : integer) : string; override;
    procedure ExecuteVerb(Index : integer); override;
    procedure PrepararForm(aForm : TfrmDigitalDisplay);
    procedure ActualizarComponente(aForm : TfrmDigitalDisplay);
  end;

var
  frmDigitalDisplay: TfrmDigitalDisplay;

procedure Register;

implementation

{$R *.DFM}

{---------------------------------------}
{ Form del Editor de propiedades        }
{---------------------------------------}

procedure TfrmDigitalDisplay.pColorFondoClick(Sender: TObject);
begin
  ColorDialog1.Color:=pColorFondo.Color;
  if ColorDialog1.Execute then
    pColorFondo.Color:=ColorDialog1.Color;
  DigitalDisplay1.BckgndColor:=pColorFondo.Color;
```

```
end;

procedure TfrmDigitalDisplay.pColorOnClick(Sender: TObject);
begin
  ColorDialog1.Color:=pColorOn.Color;
  if ColorDialog1.Execute then
    pColorOn.Color:=ColorDialog1.Color;
  DigitalDisplay1.DigitOnColor:=pColorOn.Color;
end;

procedure TfrmDigitalDisplay.pColorOffClick(Sender: TObject);
begin
  ColorDialog1.Color:=pColorOff.Color;
  if ColorDialog1.Execute then
    pColorOff.Color:=ColorDialog1.Color;
  DigitalDisplay1.DigitOffColor:=pColorOff.Color;
end;

procedure TfrmDigitalDisplay.rgAlineacionClick(Sender: TObject);
begin
  case rgAlineacion.ItemIndex of
    0 : DigitalDisplay1.Alignment:=taRightJustify;
    1 : DigitalDisplay1.Alignment:=taLeftJustify;
    2 : DigitalDisplay1.Alignment:=taCenter;
  end;
end;

procedure TfrmDigitalDisplay.eDigitosChange(Sender: TObject);
begin
  DigitalDisplay1.Digits:=StrToInt(eDigitos.Text);
end;

procedure TfrmDigitalDisplay.upDigitosClick(Sender: TObject; Button:
TUDBtnType);
begin
  DigitalDisplay1.Digits:=StrToInt(eDigitos.Text);
end;

procedure TfrmDigitalDisplay.cbRellenarCerosClick(Sender: TObject);
begin
  DigitalDisplay1.LeadingZeros:=cbRellenarCeros.Checked;
end;

procedure TfrmDigitalDisplay.eValorChange(Sender: TObject);
begin
  DigitalDisplay1.Value:=eValor.Text;
end;

procedure TfrmDigitalDisplay.eAnchoChange(Sender: TObject);
begin
  DigitalDisplay1.Width:=StrToInt(eAncho.Text);
end;

procedure TfrmDigitalDisplay.eAltoChange(Sender: TObject);
begin
  DigitalDisplay1.Height:=StrToInt(eAlto.Text);
end;

{--------------------------------------}
{ Editor de propiedades                }
{--------------------------------------}
```

```
function TDigitalDisplayEditor.GetVerbCount : integer;
begin
  Result:=1;
end;

function TDigitalDisplayEditor.GetVerb(Index : integer) : string;
begin
  Result:='E&dit...';
end;

procedure TDigitalDisplayEditor.ExecuteVerb(Index : integer);

  procedure CopyDigitalDisplay(Dest, Source : TDigitalDisplay);
  {procedimiento auxiliar que copia los valores de las propiedades de
un
   componente digital display a otro}
  begin
    Dest.Digits:=Source.Digits;
    Dest.LeadingZeros:=Source.LeadingZeros;
    Dest.Value:=Source.Value;
    Dest.Alignment:=Source.Alignment;
    Dest.BckgndColor:=Source.BckgndColor;
    Dest.DigitOnColor:=Source.DigitOnColor;
    Dest.DigitOffColor:=Source.DigitOffColor;
    Dest.Width:=Source.Width;
    Dest.Height:=Source.Height;
  end;

Var
  aForm : TfrmDigitalDisplay;
begin
  {Creamos el form del editor de propiedades}
  aForm:=TfrmDigitalDisplay.Create(Application);
  try
    {Asignamos el caption correspondiente}
    aForm.Caption:=Component.Owner.Name+'.'+Component.Name+'
'+aForm.Caption;
    {Asignamos a los controles del form los valores correspondientes
del componente
     a editar}
    PrepararForm(aForm);
    {Actualizamos el DigitalDisplay utilizado como muestra para que
presente el mismo
     aspecto que tiene el componente real}
    CopyDigitalDisplay(aForm.DigitalDisplay1, Component As
TDigitalDisplay);
    {Si el usuario acepta los cambios efectuados al form}
    if aForm.ShowModal = mrOK then
    begin
      {Actualizamos el componente en base a los controles del form}
      ActualizarComponente(aForm);
      {Informamos a Delphi que el componente ha cambiado}
      Designer.Modified;
    end;
  finally
    {Liberamos el form}
    aForm.Free;
  end;
end;
```

```
procedure TDigitalDisplayEditor.PrepararForm(aForm :
TfrmDigitalDisplay);
{Este método asigna a los controles del form los valores
correspondientes
 del componente a editar}
begin
  with Component As TDigitalDisplay do
  begin
    aForm.eDigitos.Text:=IntToStr(Digits);
    aForm.cbRellenarCeros.Checked:=LeadingZeros;
    aForm.eValor.Text:=Value;
    case Alignment of
      taRightJustify : aForm.rgAlineacion.ItemIndex:=0;
      taLeftJustify : aForm.rgAlineacion.ItemIndex:=1;
      taCenter : aForm.rgAlineacion.ItemIndex:=2;
    end;
    aForm.pColorFondo.Color:=BckgndColor;
    aForm.pColorOn.Color:=DigitOnColor;
    aForm.pColorOff.Color:=DigitOffColor;
    aForm.eAncho.Text:=IntToStr(Width);
    aForm.eAlto.Text:=IntToStr(Height);
  end;
end;

procedure TDigitalDisplayEditor.ActualizarComponente(aForm :
TfrmDigitalDisplay);
{Este método actualiza el componente real en base a los valores de
los controles
 del form }
begin
  with Component As TDigitalDisplay do
  begin
    Digits:=StrToInt(aForm.eDigitos.Text);
    LeadingZeros:=aForm.cbRellenarCeros.Checked;
    Value:=aForm.eValor.Text;
    case aForm.rgAlineacion.ItemIndex of
      0 : Alignment:=taRightJustify;
      1 : Alignment:=taLeftJustify;
      2 : Alignment:=taCenter;
    end;
    BckgndColor:=aForm.pColorFondo.Color;
    DigitOnColor:=aForm.pColorOn.Color;
    DigitOffColor:=aForm.pColorOff.Color;
    Width:=StrToInt(aForm.eAncho.Text);
    Height:=StrToInt(aForm.eAlto.Text);
  end;
end;

procedure Register;
{Registro del editor de componentes}
begin
  RegisterComponentEditor(TDigitalDisplay,TDigitalDisplayEditor);
end;

end.
```

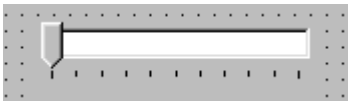
Unidad 12. Componentes de bases de datos: TDBTrackBar.

Antes de nada, una pequeña disculpa por la tardanza de esta nueva unidad. El motivo no es otro que el trabajo. He estado muy ocupado estas últimas semanas desarrollando un componente para la empresa en la que trabajo. Se trata de un conjunto de componentes que dotan a las aplicaciones desarrolladas en Delphi de un potente mecanismo de seguridad. Pero no os preocupeis que nos os voy a soltar el rollo. Tan sólo os invito a bajaros la beta, totalmente funcional hasta el 15-6, de nuestra página web: www.dadvina.com o, si os resulta más comodo, tan sólo debéis pinchar en el banner que aparece al principio de está página.

Introducción

En esta unidad aprendemos a crear componentes enlazados a bases de datos. Los componentes de base de datos (data-aware components) son simplemente componentes estándar, cómo los que hemos estado creando a lo largo del curso, a los que se les añade la característica de poder enlazarse con una base de datos. Este enlace puede ser de sólo lectura (el componente puede reflejar el estado del campo de la base de datos, pero no actualizarlo), o de edición (si es posible realizar esta actualización).

El componente que desarrollaremos será un componente de edición descendiente de TTrackBar, el componente que incorpora Delphi en la pestaña Win95.



He optado por esta elección ya que ilustra el caso más común: a partir de un componente existente, descendemos uno nuevo enlazado a una base de datos (tal y como ocurre por ejemplo con el control TDBEdit, descendiente de TEdit).

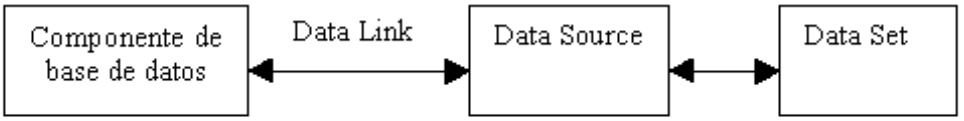
Antes de empezar conviene diferenciar entre dos tipos de componentes enlazados a datos:

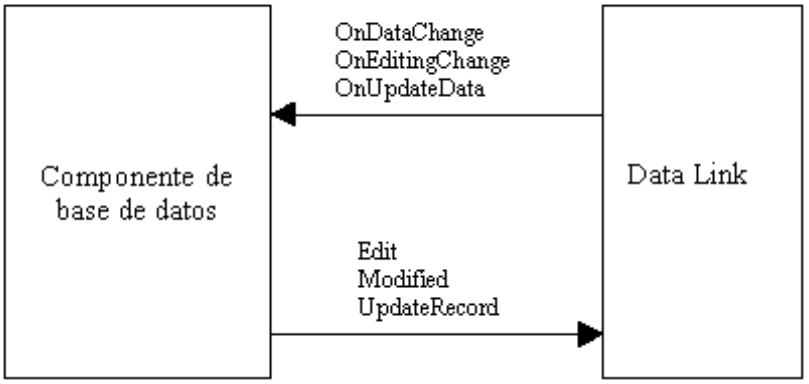
- Los que se enlazan a un sólo campo de la base de datos, cómo por ejemplo los controles TDBEdit, TDBLabel, etc. y el componente que desarrollaremos en esta unidad
- Los que se enlazan a múltiples campos de la base de datos, cómo el control TDBGrid.

De modo que pongámonos manos a la obra estudiando como se realiza la conexión entre el componente y la base de datos.

El enlace entre el componente y la base de datos: TDataLink y TFieldDataLink

Cuando se desarrolla un componente enlazado a una base de datos, la conexión entre ellos no se realiza directanente, sino que se realiza mediante un objeto TDataLink. Este objeto DataLink proporciona una serie de propiedades, métodos y eventos que facilitan la comunicación entre la base de datos y el control asociado:





Todos los componentes de bases de datos utilizan un objeto TDataLink para comunicarse con la base de datos. Pero generalmente el componente no se conecta ciertamente a este objeto, sino que lo hace a uno de sus descendientes: TFieldDataLink o TGridDataLink.

¿Cuándo utilizar uno u otro? Nada más fácil: si el componente que queremos crear debe conectarse a un solo campo de la base de datos utilizaremos un TFieldDataLink (por ejemplo los componentes DBEdit, DBLabel,...). Si, por el contrario, nuestro componente debe conectarse a varios campos de la base de datos utilizaremos el objeto TGridDataLink (o en algún caso, el objeto TDataLink directamente).

Cómo ya hemos mencionado, nuestro componente se conectará a un sólo campo de la base de datos por lo que conviene que estudiemos las propiedades y métodos del objeto TFieldDataLink:

Métodos:

Edit: al llamar a este método, el registro actual del data source se pone en estado de edición (siempre y cuando sea posible). Si tiene éxito, se devuelve True, en caso contrario se devuelve False.

Modified: cuando el componente de base de datos cambia los valores del control debe informar al objeto data source de que dicho campo se ha producido. Para ello se llama a este método. Cabe destacar que la llamada a este método es sólo una notificación al objeto de que se han producido cambios, pero nada más. Será el propio DataLink el que, cuando los necesite, nos pedirá el nuevo valor a almacenar en la base de datos mediante el evento OnUpdateData.

Reset: este método permite descargar los cambios y recuperar los datos del registro desde el Data Source.

Propiedades

CanModify: propiedad de sólo lectura que indica si el control puede modificar el valor del campo. Si el campo o dataset es de sólo lectura, entonces, modified devuelve falso.

Control: esta propiedad devuelve una referencia al control data. Cuando el control crea el objeto data link, debe asignar a esta propiedad el valor self.

Editing: propiedad de sólo lectura que devuelve true si el data source está en estado de edición.

Field: esta propiedad de sólo lectura devuelve una referencia al campo de la base de datos o nil si el data link no está enlazado a un campo.

FieldName: esta propiedad almacena el nombre del campo de la base de datos enlazado.

Eventos

OnActiveChange: como su propio nombre indica este evento es activado cuando la propiedad active del data source cambia.

OnDataChange: este evento es llamado cada que el registro cambia. En este evento el programador debe recuperar los nuevos valores del campo desde el data source.

OnEditingChange: este evento es activado cuando el data source se pone en estado de edición, así como cuando deja de estarlo.

OnUpdateData: este evento es llamado cuando el data source necesita actualizar los datos desde el control. En este evento debemos codificar los pasos necesarios para actualizar el registro de la base de datos. Sólo se llamará a este evento si el campo ha sido modificado (el método modified ha sido llamado). Obviamente, si estamos construyendo un componente de sólo lectura no necesitaremos codificar nada en este método.

Una vez conocido el arsenal que tenemos a nuestra disposición, es hora de utilizarlo: ¡es la guerra! ;-)

Añadiendo la propiedad ReadOnly

Si nos fijamos en los controles enlazados a datos incluidos con Delphi, comprobaremos que todos tienen una propiedad que indica si el control puede modificar los datos del data source asociado o no. Esta es la propiedad ReadOnly. Este es uno de los pasos genéricos que hay que realizar siempre que construyamos un componente de base de datos.

Hasta aquí está claro. Lo que no está tan claro es cómo hacerlo. Me explico. Imáginemos que queremos construir un nuevo componente enlazado a una base de datos, (que curioso, justo lo que queremos ;-)). Podemos encontrarnos en uno de los siguientes casos (partimos siempre de un componente no enlazado a una base de datos):

- Tenemos el código fuente del componente original: perfecto, este es el caso más sencillo. Basta con crear una propiedad ReadOnly y, en el momento de ir actualizar el contenido de la base de datos, comprobar previamente dicha propiedad.

- Partimos de un componente del que no tenemos el código fuente: esto ya es más complicado. Dependiendo de cómo se haya escrito el componente original (existencia de métodos protected, etc), nuestra tarea puede ir de lo medianamente sencillo a lo totalmente imposible. :-(

Nuestro caso, aunque no lo parezca, es el segundo. Es cierto que con Delphi se entrega el código fuente de los componentes, pero esto no quiere decir que lo podamos distribuir libremente. Se puede distribuir libremente el código fuente de los componentes que nosotros desarrollemos, pero no del componente antecesor del nuestro si este forma parte de los controles estandar de Delphi. Así que vamos a suponer que no disponemos del código fuente del componente TTrackBar, lo cuál hará la tarea aún más interesante.

Antes de proseguir un pequeño inciso: ojo si partimos de un componente que ya tenga la propiedad ReadOnly. Esto no indica nada. De hecho, tendríamos que asegurarnos de sincronizar la propiedad ReadOnly del control con el estado real del data source. Un ejemplo de este caso es el control TEdit, que ya tiene la propiedad ReadOnly, la cual no tiene nada que ver con el estado del data source asociado; de hecho, el componente TEdit no está asociado a ninguna base de datos.

Una vez hecha está aclaración, veamos en nuestro caso cómo podemos implementar la propiedad ReadOnly en nuestro componente. La declaración es sencilla:

```
TTrackBar = class(TrackBar)
private
  FReadOnly : boolean;
...
published
  property ReadOnly : boolean read FReadOnly write FReadOnly default
False;
...
end;
```

Ahora bien, un control en estado de sólo lectura, no acepta las pulsaciones de teclas ni los clics del ratón. ¿Cómo conseguir este efecto en nuestro componente? Tenemos dos posibilidades:

- Reimplementar (override) los métodos de mensajes KeyDown, MouseDown, etc. del componente ancestor. En cada uno de estos métodos deberíamos comprobar si el componente está en estado de sólo lectura. En caso afirmativo, obviamos la pulsación, mientras que en caso negativo, llamamos al método heredado (con inherited MouseDown por ejemplo).

Esta es una aproximación que suele funcionar en la mayoría de los casos, pero que en nuestro caso concreto no lo hace. El problema radica en que, aunque el valor de la propiedad ReadOnly sea True, el componente sigue admitiendo las entradas procedentes del teclado y el ratón. Esto es debido a que el código fuente del componente TTrackBar,

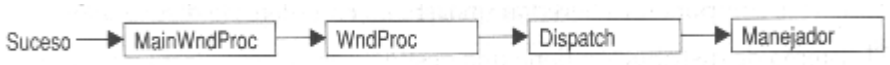
tal y cómo se puede comprobar en el código fuente de la VCL, se límita básicamente a encapsular un control estándar de Windows 95. Por ello, el control ya ha procesado la tecla pulsada antes de ejecutar el código introducido en el evento KeyDown, por lo que no le afecta nada de lo codificado en él. De modo que deshechemos en este caso concreto esta posibilidad, pero teniendo en cuenta que si estuvieramos descendiendo de un ancestor propio, este sería el procedimiento adecuado a seguir.

- La segunda posibilidad consiste en implementar (override) el procedimiento de mensajes del control por medio del del método WndProc.

La explicación detallada de los procedimientos de mensajes de windows queda fuera del alcance de este capítulo, ya que forma parte del propio núcleo de Windows, de modo que me limitare a exponer el funcionamiento básico del mismo. Explicaciones detalladas pueden encontrarse en la ayuda en línea de Delphi y en la ayuda del API de Windows. En cualquier caso, si quereis que dediquemos una unidad del curso a este tema, no teneis más que escribirme.

La explicación del método WndProc, cómo acabo de mencionar, de forma un tanto simplista, es la siguiente:

Toda ventana en Windows tiene mecanismos estándar para gestionar los mensajes denominados manejadores de mensajes. En Delphi, la VCL define un sistema de distribución de mensajes que traduce en llamadas a métodos todos los mensajes Windows dirigidos a un objeto en particular. El diagrama siguiente muestra el sistema de distribución de mensajes:



Cuando queramos procesar de una forma particular un mensaje en concreto, bastará con con redefinir el método manejador del mensaje utilizando la directiva message (veremos un ejemplo de esto un poco más adelante). Pero si esto no nos basta, y lo que queremos es que el mensaje no se llegue a distribuir a su manejador, debemos capturar el mensaje en el método WndProc, ya que este método filtra los mensajes antes de pasarlos a Dispatch, que a su vez los pasa finalmente al manejador de cada mensaje particular.

Esto es lo que realizamos en nuestro componente:

```
TDBTrackBar = class(TTrackBar)
...
protected
  procedure WndProc(var Message: TMessage); override;
...

implementation
...

procedure TDBTrackBar.WndProc(var Message: TMessage);
begin
  if not (csDesigning in ComponentState) then
  begin
    {Si el control está en read only o no está en estado de edición,
     obviamos las pulsaciones de teclas y los mensajes del ratón}
```

```
    if FReadOnly OR (not FReadOnly AND (FDatalink<>nil) AND not
       FDataLink.Edit) then
    begin
       if ((Message.Msg >= WM_MOUSEFIRST) AND (Message.Msg
<=WM_MOUSELAST))
       OR (Message.Msg = WM_KEYDOWN) OR (Message.Msg = WM_KEYUP) then
         exit;
       end;
    end;
  inherited WndProc(Message);
end;
```

Primeramente comprobamos si estamos en tiempo de diseño. Si la respuesta es afirmativa, nos limitamos a llamar al procedimiento WndProc heredado. Esta llamada es fundamental, ya que si no la hacemos el componente nunca recibiría los mensajes que le envía el sistema operativo y el resultado, por lo general, será un cuelgue del programa o incluso del ordenador.

Si estamos en tiempo de ejecución, debemos capturar los mensajes de teclado y ratón y evitar que pasen al componente siempre que el componente esté en estado de sólo lectura. Para ello comprobamos el rango del mensaje pasado: si este es uno de los que queremos eliminar simplemente salimos del procedimiento **sin** llamar a WndProc. En caso contrario, somos buenos y dejamos que el resto de mensajes lleguen al componente :-)

Con esto ya tenemos solucionado el tema de la propiedad ReadOnly. Aunque lo hemos solucionado con poco código, no os engañeis: el tema de la captura de mensajes es uno de los temas más complejos (y potentes) que hemos visto a lo largo del curso. Os recomiendo que antes de empezar a utilizar capturas de este tipo os leais muy detenidamente toda la ayuda disponible en Delphi y en el API sobre este tema. Os evitareis muchos disgustos ;-)

Implementando el DataLink

Ha llegado el momento de añadir a nuestro componente el enlace con la base de datos. Para ello tenemos que construir un objeto TDataLink, concretamente un TFieldDataLink, para conectar el componente a la base de datos. Veamos la declaración completa de nuestro componente:

```
  TDBTrackBar = class(TTrackBar)
  private
    FReadOnly : boolean;
    FDataLink : TFieldDataLink;
    function GetDataField : string;
    procedure SetDataField(const Value : string);
    function GetDataSource : TDataSource;
    procedure SetDataSource(Value : TDataSource);
    procedure UpdateData(Sender: TObject);
    procedure DataChange(Sender : TObject);
    procedure EditingChange(Sender : TObject);
  protected
    procedure CMExit(var Message: TWMNoParams); message CM_EXIT;
    procedure CNHScroll(var Message: TWMHScroll); message CN_HSCROLL;
    procedure CNVScroll(var Message: TWMVScroll); message CN_VSCROLL;
```

```
    procedure WndProc(var Message: TMessage); override;
    procedure Notification(AComponent : TComponent; Operation :
TOperation); override;
  public
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
  published
    property ReadOnly : boolean read FReadOnly write FReadOnly default
False;
    property DataField : string read GetDataField write SetDataField;
    property DataSource : TDataSource read GetDataSource write
SetDataSource;
  end;
```

Primero declaramos el campo FDataLink de tipo TFieldDataLink. Ahora debemos crearlo en el constructor de nuestro componente y enganchar los eventos del mismo que nos interesan:

```
constructor TDBTrackBar.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);
  ControlStyle:=ControlStyle - [csReplicatable]; // El control no se
permite en un DBCtrlGrid
  FReadOnly:=False;
  FDataLink:=TFieldDataLink.Create;
  FDataLink.OnDataChange:=DataChange;
  FDataLink.OnUpdateData:=UpdateData;
  // FDataLink.OnEditingChange:=EditingChange; Descomentar esta línea
para que el control tenga AutoEdit
  FDataLink.Control:=Self;
end;
```

Una vez creado el DataLink, enganchamos a los eventos OnDataChange y OnUpdateData nuestros manejadores para los mismos. Además, asignamos el valor de la propiedad Control del DataLink una referencia a nuestro componente.

Por último, dos cosas: primero: nuestro componente no tiene capacidades de replicación por lo que quitamos dicho flag de la propiedad ControlStyle (de este modo no se podrá utilizar el control en un TDBCtrlGrid). Segundo: Hay controles de base de datos que tiene capacidad de AutoEdición (p.e. el control TEdit). En principio no queremos que nuestro componente tenga esta caracterísca, pero si os interesa probarlo, basta con que descomenteis la línea que asigna el evento OnEditingChange.

Además de construir el DataLink, al final hay que destruirlo, o sea que:

```
destructor TDBTrackBar.Destroy;
begin
  FDataLink.Free;
  FDataLink:=nil;
  inherited Destroy;
end;
```

Una vez creado, el usuario querrá poderlo utilizar ;-) de modo que vamos a proporcionarle las propiedades que todo control enlazado a datos tiene: DataSource y DataField. Comencemos con la propiedad DataSource:

```
property DataSource : TDataSource read GetDataSource write
SetDataSource;

function TDBTrackBar.GetDataSource : TDataSource;
begin
  Result:=FDataLink.DataSource;
end;

procedure TDBTrackBar.SetDataSource(Value : TDataSource);
begin
  if FDataLink.DataSource<>Value then
  begin
    FDataLink.DataSource:=Value;
    if Value<>nil then
      Value.FreeNotification(Self);
  end;
end;
```

El método get de la propiedad no tiene ningún misterio: se limita a devolver el correspondiente DataSource de nuestro objeto DataLink.

El método set si es algo más interesante. Cuando se asigna un nuevo valor a la propiedad, comprobamos que dicho valor sea distinto de nil y, en caso afirmativo, llamamos al método FreeNotification del nuevo data source asignado. Venga, una pausa para que consulteis en la ayuda en línea que hace este método. Espero...

¿Os ha quedado claro? ¿qué no demasiado? Pues para eso estoy yo, para intentar hacerlo un poco mejor que la ayuda en línea. Supongamos que no añadimos la línea Value.FreeNotification(self) e imaginemos que ya hemos terminado el componente y un usuario va a utilizarlo. Para ello, se crea en un módulo de datos su tabla paradox (objeto TTable) y un TDataSource. Luego, en un form de su aplicación coloca nuestro TDBTrackBar y asigna a la propiedad DataSource el Data Source que tiene en el módulo de datos. Lo más normal de mundo ¿no?. Ahora resulta que el usuario cambia de opinión, va al módulo de datos y borra el TDataSource. En este momento, nuestro componente tiene en la propiedad data source una referencia inválida (el objeto del módulo de datos ha sido destruido). Esto provocará un error severo cuando seleccionemos nuestro componente y puede provocar el cuelgue del propio Delphi. Chungo ¿verdad? Pues todo esto se evita añadiendo la famosa línea. De este modo le estamos diciendo al objeto DataSource del módulo de datos: notifícame cuando te vas a destruir para que yo limpie la referencia que apunta a tí. Y esta limpieza se realiza cuando recibimos la notificación:

```
procedure TDBTrackBar.Notification(AComponent : TComponent; Operation
: Toperation);
begin
  inherited Notification(AComponent,Operation);
  if (Operation=opRemove) AND (FDataLink<>nil) AND
(AComponent=DataSource) then
    DataSource:=nil;
end;
```

Veamos ahora la propiedad DataField:

```
property DataField : string read GetDataField write SetDataField;
```

```
function TDBTrackBar.GetDataField : string;
begin
  Result:=FDataLink.FieldName;
end;

procedure TDBTrackBar.SetDataField(const Value : string);
begin
  FDataLink.FieldName:=Value;
end;
```

Está chupada, así que vamos a algo más interesante.

Lectura de los valores de los campos de la base de datos

Una vez que tenemos el objeto DataLink creado, ya podemos utilizarlo para recuperar el valor del campo de la base de datos. El mecanismo es muy sencillo: cada vez que el data source indica un cambio en sus datos (bien sea por un desplazamiento a otro registro de la tabla, modificación del registro actual, etc) el objeto DataLink disparará el evento OnDataChange. En respuesta a este evento nuestro componente debe leer el nuevo valor del campo y reflejarlo de forma visual en el componente.

Recordemos que en el constructor hemos enlazado el método UpdateData como respuesta al evento OnUpdateData. Veamos ahora cómo codificamos este método:

```
procedure TDBTrackBar.DataChange(Sender : TObject);
begin
  if FDataLink.Field=nil then
    Position:=0
  else
    Position:=FDataLink.Field.AsInteger;
end;
```

Primero comprobamos que haya un campo asignado en la propiedad Field del DataLink y, en caso negativo, se devuelve un valor arbitrario como posición del TrackBar (en este caso se devuelve 0). En caso afirmativo, basta con leer el valor del campo de la base de datos mediante FDataLink.Field.AsInteger y asignarlo a la propiedad position del TrackBar.

Así de simple. Pero conviene tener en cuenta dos aspectos:

1. Por conveniencia, leemos el valor del campo como entero, ya que es lo que nos interesa a nuestros propósitos, ya que nuestro componente se enlazará a campos numéricos de la base de datos. Si el usuario intenta enlazarlo con un campo no numérico, obtendrá la excepción 'xxx is not a valid numeric value', cosa perfectamente normal.

2. Una vez leído el valor del campo, hay que actualizar el componente para que refleje el nuevo valor. Esta tarea variará mucho en función del componente , pero básicamente consistirá en asignar el valor del campo a una determinada propiedad y llamar al método Paint para que se muestre el nuevo valor. En nuestro caso, basta con asignar el valor del

campo a la propiedad position del TrackBar, ya que dicha propiedad heredada se encargará de repintar el componente.

Una vez que hemos aprendido a asignar el nuevo valor del campo al componente, nos queda aprender cómo hacer el paso contrario, es decir, cuando el usuario modifica el valor de la propiedad position del trackBar (mediante el teclado, el ratón, o por código), dicho valor debe actualizarse en el campo de la base de datos. En la siguiente sección veremos cómo hacerlo.

Actualizando el valor del campo de la base de datos

El primer paso que debemos ejecutar para actualizar el valor del campo de la base de datos, es informar al objeto DataLink de que se ha producido un cambio. Para ello debemos llamar al método Modified del DataLink. La cuestión de cuando llamarlo, dependerá del tipo de componente que estemos construyendo. En nuestro caso es bastante simple: cada vez que se produce un cambio en la propiedad position, se genera un mensaje CNHScroll (o CCNVScroll si el salto es de más de una unidad). Los mensajes que contienen el prefijo CN son mensajes internos de la VCL de Delphi (Component Notification) y estos son dos claros ejemplos de mensajes que debemos reimplementar, ya que sólo se generan estos mensajes cuando se ha producido un cambio en el estado del TrackBar, lo que los hace ideales para nuestros propósitos:

```
 TDBTrackBar = class(TTrackBar)
  ....
  protected
    procedure CNHScroll(var Message: TWMHScroll); message CN_HSCROLL;
    procedure CNVScroll(var Message: TWMVScroll); message CN_VSCROLL;
  ...

implementation

procedure TDBTrackBar.CNHScroll(var Message: TWMHScroll);
begin
  inherited;
  FDataLink.Modified
end;

procedure TDBTrackBar.CNVScroll(var Message: TWMVScroll);
begin
  inherited;
  FDataLink.Modified
end;
```

El mecanismo de implementar nuestro propio manejador de mensajes es simple: en la sección de interface definimos los métodos que deben responder a los mensajes junto con la directiva message seguida del nombre del mensaje a capturar.

En la sección de implementación, nos limitamos a informar al DataLink de que se han producido cambios en el valor del campo (aunque donde realmente se han producido es en el propio componente y el llamar a Modified es una "declaración de intenciones") y que el objeto DataLink debe tenerlo en cuenta cuando vaya a actualizar el registro de la base de datos.

Con esto, el objeto DataLink sabe que se han producido cambios, por lo que cuando sea necesario actualizar el registro (por una llamada a Post, movimiento del puntero del data set, etc), se activará el evento OnUpdateData. Sólo tenemos que codificar en respuesta a este evento (recordemos que ya lo enganchamos al método UpdateData del constructor), las instrucciones necesarias para actualizar el valor del campo de la base de datos, lo cuál lo conseguimos asignando el nuevo valor a la propiedad DataLink.Field:

```
procedure TDBTrackBar.UpdateData(Sender: TObject);
begin
 FDataLink.Field.AsInteger := Position;
end;
```

Con esto ya casi está. Nos queda un último paso. Una vez que el usuario del componente hace click con el ratón para modificar el valor de la propiedad position, informamos al DataLink de que el campo va a cambiar llamando al método Modified. De este modo el objeto DataLink activará el evento OnUpdateData para asignar el nuevo valor al campo, y este evento se disparará cuando sea requerido por el data source. Pero tambien debemos disparar este evento cuando nuestro componente pierda el foco: así nos aseguramos de que el campo se actualizará correctamente.

Para realizar esta tarea debemos reimplementar el mensaje CMExit, que se genera cuando un componente pierde el foco:

```
procedure CMExit(var Message: TWMNoParams); message CM_EXIT;
...
procedure TDBTrackBar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord; { Actualizamos el data link}
  except
    SetFocus; { Si falla la actualización, mantenemos el foco en el
control}
    raise; { y relanzamos la excepción}
  end;
  inherited;
end;
```

En este método nos limitamos a llamar al método UpdateRecord para forzar la actualización del campo de la base de datos. Si se produce una excepción en este proceso (tipicamente por asignar un valor no válido al campo, por clave duplicada, etc). mantenemos el foco en el control y relanzamos la excepción para que la vea el usuario.

Un detalle a destacar: el método UpdateRecord no lo hemos visto al estudiar los métodos del objeto TFieldDataLink ya que está definido en TDataLink, el antecesor de TFieldDataLink. Su cometido es obvio: forzar la llamada a OnUpdateData para que se asigne un nuevo valor al campo de la base de datos (recordar que previamente se ha debido también llamar al método Modified, ya que en caso contrario el evento OnUpdateData no se dispararía).

Con esto hemos terminado nuestro componente. Tan sólo nos queda registrarlo y disfrutar de él.

Por último estoy recibiendo diversas sugerencias sobre temas para próximas unidades, por lo cuál me gustaría que me enviaseis ideas a mi dirección de correo electrónico. Entre los temas que me han sugerido hasta el momento destacan:

- Ayuda en línea para los componentes
- Desarrollo de controles ActiveX
- Componentes con conexión a múltiples campos de la base de datos

Código fuente del componente

```
unit DBTrackBar;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Comctrls, Controls,
DB, DBTables, DBCtrls;

type
  TDBTrackBar = class(TTrackBar)
  private
    FReadOnly : boolean;
    FDataLink : TFieldDataLink;
    function GetDataField : string;
    procedure SetDataField(const Value : string);
    function GetDataSource : TDataSource;
    procedure SetDataSource(Value : TDataSource);
    procedure UpdateData(Sender: TObject);
    procedure DataChange(Sender : TObject);
    procedure EditingChange(Sender : TObject);
  protected
    procedure CMExit(var Message: TWMNoParams); message CM_EXIT;
    procedure CNHScroll(var Message: TWMHScroll); message CN_HSCROLL;
    procedure CNVScroll(var Message: TWMVScroll); message CN_VSCROLL;
    procedure WndProc(var Message: TMessage); override;
    procedure Notification(AComponent : TComponent; Operation :
TOperation); override;
  public
    constructor Create(AOwner : TComponent); override;
    destructor Destroy; override;
  published
    property ReadOnly : boolean read FReadOnly write FReadOnly default
False;
    property DataField : string read GetDataField write SetDataField;
    property DataSource : TDataSource read GetDataSource write
SetDataSource;
  end;

procedure Register;

implementation

constructor TDBTrackBar.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);
  ControlStyle:=ControlStyle - [csReplicatable];  // El control no se
permite en un DBCtrlGrid
  FReadOnly:=False;
```

```
  FDataLink:=TFieldDataLink.Create;
  FDataLink.OnDataChange:=DataChange;
  FDataLink.OnUpdateData:=UpdateData;
  // FDataLink.OnEditingChange:=EditingChange;  Descomentar esta línea
para que el control tenga AutoEdit
  FDataLink.Control:=Self;
end;

destructor TDBTrackBar.Destroy;
begin
  FDataLink.Free;
  FDataLink:=nil;
  inherited Destroy;
end;

function TDBTrackBar.GetDataField : string;
begin
  Result:=FDataLink.FieldName;
end;

procedure TDBTrackBar.SetDataField(const Value : string);
begin
  FDataLink.FieldName:=Value;
end;

function TDBTrackBar.GetDataSource : TDataSource;
begin
  Result:=FDataLink.DataSource;
end;

procedure TDBTrackBar.SetDataSource(Value : TDataSource);
begin
  if FDataLink.DataSource<>Value then
  begin
    FDataLink.DataSource:=Value;
    if Value<>nil then
      Value.FreeNotification(Self);
  end;
end;

procedure TDBTrackBar.Notification(AComponent : TComponent; Operation
: Toperation);
begin
  inherited Notification(AComponent,Operation);
  if (Operation=opRemove) AND (FDataLink<>nil) AND
(AComponent=DataSource) then
    DataSource:=nil;
end;

procedure TDBTrackBar.DataChange(Sender : TObject);
begin
  if FDataLink.Field=nil then
    Position:=0
  else
    Position:=FDataLink.Field.AsInteger;
end;

procedure TDBTrackBar.CNHScroll(var Message: TWMHScroll);
begin
  inherited;
  FDataLink.Modified
```

```
end;

procedure TDBTrackBar.CNVScroll(var Message: TWMVScroll);
begin
  inherited;
  FDataLink.Modified
end;

procedure TDBTrackBar.UpdateData(Sender: TObject);
begin
 FDataLink.Field.AsInteger := Position;
end;

procedure TDBTrackBar.EditingChange(Sender: TObject);
begin
  FReadOnly := not FDataLink.Editing;
end;

procedure TDBTrackBar.CMExit(var Message: TWMNoParams);
begin
 try
   FDataLink.UpdateRecord; { Actualizamos el data link}
 except
   SetFocus;   { Si falla la actualización, mantenemos el foco en el
control}
   raise;      { y relanzamos la excepción}
 end;
 inherited;
end;

procedure TDBTrackBar.WndProc(var Message: TMessage);
begin
  if not (csDesigning in ComponentState) then
  begin
    {Si el control está en read only o no está en estado de edición,
     obviamos las pulsaciones de teclas y los mensajes del ratón}
    if FReadOnly OR (not FReadOnly AND (FDatalink<>nil) AND not
FDataLink.Edit) then
    begin
    if ((Message.Msg >= WM_MOUSEFIRST) AND (Message.Msg <=
WM_MOUSELAST))
    OR (Message.Msg = WM_KEYDOWN) OR (Message.Msg = WM_KEYUP) then
      exit;
    end;
  end;
  inherited WndProc(Message);
end;

procedure Register;
begin
  RegisterComponents('Curso', [TDBTrackBar]);
end;

end.
```