DELPHI AL LIMITE

(http://delphiallimite.blogspot.com/)

CONTENIDO

Explorar unidades y directorios

Si importante es controlar el manejo de archivos no menos importante es el saber moverse por las unidades de disco y los directorios.

Veamos que tenemos Delphi para estos menesteres:

function CreateDir(const Dir: string): Boolean;

Esta función crea un nuevo directorio en la ruta indicada por Dir. Devuelve True o False dependiendo si ha podido crearlo o no. El único inconveniente que tiene esta función es que deben existir los directorios padres. Por ejemplo:

```
CreateDir( 'C:\prueba' )                devuelve True
CreateDir( 'C:\prueba\documentos' )     devuelve True
CreateDir( 'C:\otraprueba\documentos' ) devuelve False (y no lo crea)
```

function ForceDirectories(Dir: string): Boolean;

Esta función es similar a CreateDir salvo que también crea toda la ruta de directorios padres.

```
ForceDirectories( 'C:\prueba' )                devuelve True
ForceDirectories( 'C:\prueba\documentos' )     devuelve True
ForceDirectories( 'C:\otraprueba\documentos' ) devuelve True
```

procedure ChDir(const S: string); overload;

Este procedimiento cambia el directorio actual al indicado por el parámetro S. Por ejemplo:

```
ChDir( 'C:\Windows\Fonts' );
```

function GetCurrentDir: string;

Nos devuelve el nombre del directorio actual donde estamos posicionados. Por ejemplo:

```
GetCurrentDir devuelve C:\Windows\Fonts
```

function SetCurrentDir(const Dir: string): Boolean;

Establece el directorio actual devolviendo True si lo ha conseguido. Por ejemplo:

```
SetCurrentDir( 'C:\Windows\Java' );
```

procedure GetDir(D: Byte; var S: string);

Devuelve el directorio actual de una unidad y lo mete en la variable S. El parámetro D es el número de la unidad siendo:

```
D    Unidad
---  ------------------
0    Unidad por defecto
1    A:
2    B:
3    C:
...
```

Por ejemplo para leer el directorio actual de la unidad C:

```
var
  sDirectorio: String;
begin
  GetDir( 3, sDirectorio );
  ShowMessage( 'El directorio actual de la unidad C: es ' +
sDirectorio );
end;
```

function RemoveDir(const Dir: string): Boolean;

Elimina un directorio en el caso de que este vacío, devolviendo False si no ha podido hacerlo.

```
RemoveDir( 'C:\prueba\documentos' ) devuelve True
RemoveDir( 'C:\prueba' )            devuelve True
RemoveDir( 'C:\otraprueba' )        devuelve False porque no esta
vacío
```

function DirectoryExists(const Directory: string): Boolean;

Comprueba si existe el directorio indicado por el parámetro Directory. Por ejemplo:

```
DirectoryExists( 'C:\Windows\System32\' )      devuelve True
DirectoryExists( 'C:\Windows\MisDocumentos\' ) devuelve False
```

function DiskFree(Drive: Byte): Int64;

Devuelve el número de bytes libres de una unidad de dico indicada por la letra Drive:

```
Drive          Unidad
-----------    ------
0              Unidad por defecto
1              A:
2              B:
3              C:
...
```

Por ejemplo vamos a ver el número de bytes libres de la unidad C:

```
DiskFree( 3 ) devuelve 5579714560
```

function DiskSize(Drive: Byte): Int64;

Nos dice el tamaño total en bytes de una unidad de disco. Por ejemplo:

```
DiskSize( 3 ) devuelve 20974428160
```

BUSCANDO ARCHIVOS DENTRO DE UN DIRECTORIO

Para buscar archivos dentro de un directorio disponemos de las funciones:

function FindFirst(const Path: string; Attr: Integer; var F: TSearchRec): Integer;

Busca el primer archivo, directorio o unidad que se encuentre dentro de una ruta en concreto. Devuelve un cero si ha encontrado algo. El parámetro TSearchRec es una estructura de datos donde se almacena lo encontrado:

```
type
  TSearchRec = record
    Time: Integer;
    Size: Integer;
    Attr: Integer;
    Name: TFileName;
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData;
  end;
```

function FindNext(var F: TSearchRec): Integer;

Busca el siguiente archivo, directorio o unidad especificado anteriormente por la función FindFirst. Devuelve un cero si ha encontrado algo.

procedure FindClose(var F: TSearchRec);

Este procedimiento cierra la búsqueda comenzada por FindFirst y FindNext.

Veamos un ejemplo donde se utilizan estas funciones. Vamos a hacer un procedimiento que lista sólo los archivos de un directorio que le pasemos y vuelca su contenido en un StringList:

```
procedure TFPrincipal.Listar( sDirectorio: string; var Resultado:
TStringList );
var
  Busqueda: TSearchRec;
  iResultado: Integer;
begin
  // Nos aseguramos que termine en contrabarra
  sDirectorio := IncludeTrailingBackslash( sDirectorio );
```

```
  iResultado :=  FindFirst( sDirectorio + '*.*', faAnyFile, Busqueda
);

  while iResultado = 0 do
  begin
    // ¿Ha encontrado un archivo y no es un directorio?
    if ( Busqueda.Attr and faArchive = faArchive ) and
       ( Busqueda.Attr and faDirectory <> faDirectory ) then
      Resultado.Add( Busqueda.Name );

    iResultado := FindNext( Busqueda );
  end;

  FindClose( Busqueda );
end;
```

Si listamos el raiz de la unidad C:

```
var
  Directorio: TStringList;
begin
  Directorio := TStringList.Create;
  Listar( 'C:', Directorio );
  ShowMessage( Directorio.Text );
  Directorio.Free;
end;
```

El resultado sería:

```
AUTOEXEC.BAT
Bootfont.bin
CONFIG.SYS
INSTALL.LOG
IO.SYS
MSDOS.SYS
NTDETECT.COM
```

Con estas tres funciones se pueden hacer cosas tan importantes como eliminar directorios, realizar búsquedas de archivos, calcular lo que ocupa un directorio en bytes, etc.

Pruebas realizadas en Delphi 7.

Mostrando datos en el componente StringGrid

Anteriormente vimos como mostrar información en un componente ListView llegando incluso a cambiar el color de filas y columnas a nuestro antojo. El único inconveniente estaba en que no se podían cambiar los títulos de las columnas, ya que venían predeterminadas por los colores de Windows.

Pues bien, el componente de la clase TStringGrid es algo más cutre que el ListView, pero permite cambiar al 100% el formato de todas las celdas. Veamos primero como meter información en el mismo. Al igual que ocurría con el ListView, todos las celdas de un componente StringGrid son de tipo string, siendo nosotros los que le tenemos que dar formato a mano.

AÑADIENDO DATOS A LA REJILLA

Vamos a crear una rejilla de datos con las siguiente columnas:

```
NOMBRE, APELLIDO1, APELLIDO2, NIF, IMPORTE PTE.
```

Cuando insertamos un componente StringGrid en el formulario nos va a poner por defecto la primera columna con celdas fijas (fixed). Vamos a fijar las siguientes propiedades:

```
Propiedad          Valor     Descripción
---------          -----     -----------
ColCount           5         5 columnas
RowCount           4         4 filas
FixedCols          0         0 columnas fijas
FixedRows          1         1 fila fija
DefaultRowHeight   20        altura de las filas a 20 pixels
```

Ahora creamos un procedimiento para completar de datos la rejilla:

```
procedure TFormulario.RellenarTabla;
begin
  with StringGrid do
  begin
    // Título de las columnas
    Cells[0, 0] := 'NOMBRE';
    Cells[1, 0] := 'APELLIDO1';
    Cells[2, 0] := 'APELLIDO2';
    Cells[3, 0] := 'NIF';
    Cells[4, 0] := 'IMPORTE PTE.';

    // Datos
    Cells[0, 1] := 'PABLO';
    Cells[1, 1] := 'GARCIA';
    Cells[2, 1] := 'MARTINEZ';
    Cells[3, 1] := '67348321D';
    Cells[4, 1] := '1500,36';

    // Datos
    Cells[0, 2] := 'MARIA';
    Cells[1, 2] := 'SANCHEZ';
    Cells[2, 2] := 'PALAZON';
    Cells[3, 2] := '44878234A';
    Cells[4, 2] := '635,21';

    // Datos
    Cells[0, 3] := 'CARMEN';
    Cells[1, 3] := 'PEREZ';
    Cells[2, 3] := 'GUILLEN';
    Cells[3, 3] := '76892693L';
    Cells[4, 3] := '211,66';
  end;
end;
```

Al ejecutar el programa puede apreciarse lo mal que quedan los datos en

pantalla, sobre todo la columna del importe pendiente:

DANDO FORMATO A LAS CELDAS DE UN COMPONENTE STRINGGRIND

Lo que vamos a hacer a continuación es lo siguiente:

- La primera fila fija va a ser de color de fondo azul oscuro con fuente blanca y además el texto va a ir centrado.

- La columna del importe pendiente va a tener la fuente de color verde y va a ir alineada a la derecha.

- El resto de columnas tendrán el color de fondo blanco y el texto en negro.

Todo esto hay que hacerlo en el evento OnDrawCell del componente StringGrid:

```
procedure TFormulario.StringGridDrawCell( Sender: TObject; ACol,
  ARow: Integer; Rect: TRect; State: TGridDrawState );
var
  sTexto: String;           // Texto que va a imprimir en la celda
actual
  Alineacion: TAlignment;   // Alineación que le vamos a dar al texto
  iAnchoTexto: Integer;     // Ancho del texto a imprimir en pixels
begin
  with StringGrid.Canvas do
  begin
    // Lo primero es coger la fuente por defecto que le hemos asignado
al componente
    Font.Name := StringGrid.Font.Name;
    Font.Size := StringGrid.Font.Size;

    if ARow = 0 then
      Alineacion := taCenter
    else
      // Si es la columna del importe pendiente alineamos el texto a
la derecha
      if ACol = 4 then
        Alineacion := taRightJustify
      else
        Alineacion := taLeftJustify;

    // ¿Es una celda fija de sólo lectura?
    if gdFixed in State then
    begin
      Brush.Color := clNavy;      // le ponemos azul de fondo
      Font.Color := clWhite;      // fuente blanca
      Font.Style := [fsBold];     // y negrita
    end
    else
```

```
    begin
      // ¿Esta enfocada la celda?
      if gdFocused in State then
      begin
        Brush.Color := clRed;     // fondo rojo
        Font.Color := clWhite;    // fuente blanca
        Font.Style := [fsBold];   // y negrita
      end
      else
      begin
        // Para el resto de celdas el fondo lo ponemos blanco
        Brush.Color := clWindow;

        // ¿Es la columna del importe pendiente?
        if ACol = 4 then
        begin
          Font.Color := clGreen;   // la pintamos de azul
          Font.Style := [fsBold];  // y negrita
          Alineacion := taRightJustify;
        end
        else
        begin
          Font.Color := clBlack;
          Font.Style := [];
        end;
      end;
    end;

    sTexto := StringGrid.Cells[ACol,ARow];
    FillRect( Rect );
    iAnchoTexto := TextWidth( sTexto );

    case Alineacion of
      taLeftJustify: TextOut( Rect.Left + 5, Rect.Top + 2, sTexto );
      taCenter: TextOut( Rect.Left + ( ( Rect.Right - Rect.Left ) -
iAnchoTexto ) div 2, Rect.Top + 2, sTexto );
      taRightJustify: TextOut( Rect.Right - iAnchoTexto - 2, Rect.Top
+ 2, sTexto );
    end;
  end;
end;
```

Así quedaría al ejecutarlo:

Sólo hay un pequeño inconveniente y es que la rejilla primero se pinta de manera normal y luego nosotros volvemos a pintarla encima con el evento OnDrawCell con lo cual hace el proceso dos veces. Si queremos que sólo se haga una vez hay que poner a False la propiedad DefaultDrawing. Quedaría de la siguiente manera:

Por lo demás creo que este componente que puede sernos muy útil para mostrar datos por pantalla en formato de sólo lectura. En formato de escritura es algo flojo porque habría que controlar que tipos de datos puede escribir el usuario según en que columnas esté.

Pruebas realizadas en Delphi 7.

La barra de estado

Es raro encontrar una aplicación que no lleve en alguno de sus formularios la barra de estado. Hasta ahora he utilizado ejemplos sencillos de meter texto en la barra de estado de manera normal, pero el componente StatusBar permite meter múltiples paneles dentro de la barra de estado e incluso podemos cambiar el formato de la fuente en cada uno de ellos.

ESCRIBIENDO TEXTO SIMPLE

El componente de la clase TStatusBar tiene dos estados: uno para escribir texto simple en un sólo panel y otro para escribir en múltiples paneles. Supongamos que el componente de la barra de estado dentro de nuestro formulario de llama BarraEstado. Para escribir en un sólo panel se hace de la siguiente manera:

```
BarraEstado.SimplePanel := True;
BarraEstado.SimpleText := 'Texto de prueba';
```

Se puede escribir tanto texto como longitud tenga la barra de estado, o mejor dicho, tanto como sea la longitud del formulario.

ESCRIBIENDO TEXTO EN MULTIPLES PANELES

Para escribir en múltiples paneles dentro de la misma barra de estado hay que crear un panel por cada apartado. En este ejemplo voy a crear en la barra de estado tres paneles y en cada uno de ellos voy a poner un formato diferente.

```
begin
  BarraEstado.SimplePanel := False;
  BarraEstado.Panels.Clear;

  with BarraEstado.Panels.Add do
  begin
    Text := 'x=10';
    Width := 50;
    Style := psOwnerDraw;
    Alignment := taRightJustify;
  end;
```

```
  with BarraEstado.Panels.Add do
  begin
    Text := 'y=50';
    Width := 50;
    Style := psOwnerDraw;
    Alignment := taRightJustify;
  end;

  with BarraEstado.Panels.Add do
  begin
    Text := 'Texto seleccionado';
    Style := psText;
    Width := 50;
  end;
end;
```

La propiedad Style de cada panel determina si es psText o psOwnerDraw. Por defecto todos los paneles que se crean tiene el estilo psText (texto normal). Si elegimos el estilo psOwnerDraw significa que vamos a ser nosotros los encargados de dibujar el contenido del mismo. Ello se hace en el evento OnDrawPanel de la barra de estado:

```
procedure TFormulario.BarraEstadoDrawPanel( StatusBar: TStatusBar;
Panel: TStatusPanel; const Rect: TRect );
begin
  case Panel.ID of
    0: with BarraEstado.Canvas do
       begin
         Font.Name := 'Tahoma';
         Font.Size := 10;
         Font.Style := [fsBold];
         Font.Color := clNavy;
         TextOut( Rect.Left + 2, Rect.Top, Panel.Text );
       end;

    1: with BarraEstado.Canvas do
       begin
         Font.Name := 'Tahoma';
         Font.Size := 10;
         Font.Style := [fsBold];
         Font.Color := clRed;
         TextOut( Rect.Left + 2, Rect.Top, Panel.Text );
       end;
  end;
end;
```

Cuando se van creando paneles dentro de una barra de estado, a cada uno de ellos se le va asignado la propiedad ID a 0, 1, 2, etc, la cual es de sólo lectura. Como puede verse en el evento OnDrawPanel si el ID es 0 lo pinto de azul y si es 1 de rojo. Pero sólo funcionará en aquellos paneles cuyo estilo sea psOwnerDraw. Quedaría de la siguiente manera:

También puede cambiarse en cada panel propiedades tales como el marco (bevel), la alineación del texto (alignment) y el ancho (width).

Esto nos permitirá informar mejor al usuario sobre el comportamiento de nuestra aplicación en tiempo real y de una manera elegante que no interfiere con el contenido del resto del formulario.

Pruebas realizadas en Delphi 7.

Creando un navegador con el componente WebBrowser

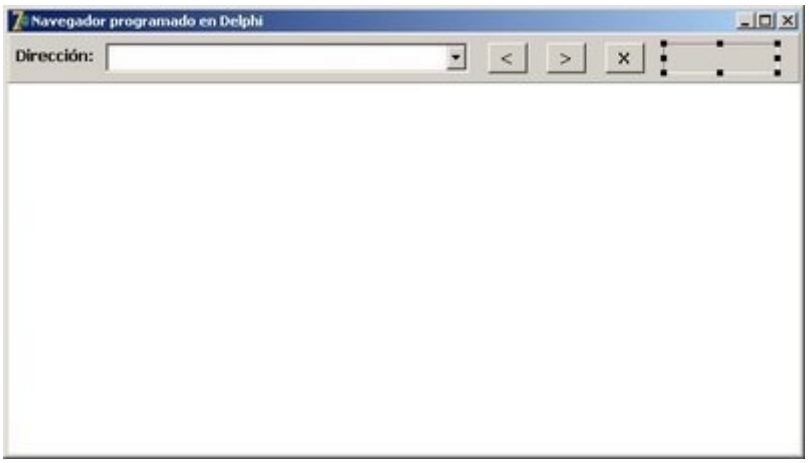
Delphi incorpora dentro de la pestaña Internet el componente de la clase TWebBrowser el cual utiliza el motor de Internet Explorer para añadir un navegador en nuestras aplicaciones.

Seguro que os preguntareis, ¿que utilidad puede tener esto si ya tengo Internet Explorer, Firefox, Opera, etc.? Pues hay ocasiones en que un cliente nos pide que ciertos usuarios sólo puedan entrar a ciertas páginas web. Por ejemplo, si un usuario está en el departamento de compras, es lógico que a donde sólo tiene que entrar es a las páginas de sus proveedores y no a otras para bajarse música MP3 a todo trapo (no os podeis imaginar la pasta que pierden las empresas por este motivo).

Entonces vamos a ver como crear un pequeño navegador con las funcionalidades mínimas.

CREANDO LA VENTANA DE NAVEGACION

Vamos a crear la siguiente ventana:



Incorpora los siguientes componentes:

- Un panel en la parte superior con su propiedad Align fijada a alTop (pegado arriba).

- Dentro del panel tenemos una etiqueta para la dirección.

- Un componente ComboBox llamado URL para la barra de direcciones.

- Tres botones para ir atrás, adelante y para detener.

- Una barra de progreso para mostrar la carga de la página web.

- Un componente WebBrower ocupando el resto del formulario mediante su propiedad Align en alClient, de tal manera que si maximizamos la ventana se respete el posicionamiento de todos los componentes.

CREANDO LAS FUNCIONES DE NAVEGACION

Una vez hecha la ventana pasemos a crear cada uno de los eventos relacionados con la navegación. Comencemos con el evento de pulsar Intro dentro de la barra de direcciones llamada URL:

```
procedure TFormulario.URLKeyDown( Sender: TObject; var Key: Word;
Shift: TShiftState );
begin
  if key = VK_RETURN then
  begin
    WebBrowser.Navigate( URL.Text );
    URL.Items.Add( URL.Text );
  end;
end;
```

Si se pulsa la tecla Intro hacemos el objeto WebBrowser navegue a esa

dirección. Además añadimos esa dirección a la lista de direcciones URL por la que hemos navegado para guardar un historial de las mismas.

Cuando se pulse el botón atrás en la barra de navegación hacemos lo siguiente:

```
procedure TFormulario.BAtrasClick( Sender: TObject );
begin
  WebBrowser.GoBack;
end;
```

Lo mismo cuando se pulse el botón adelante:

```
procedure TFormulario.BAdelanteClick( Sender: TObject );
begin
  WebBrowser.GoForward;
end;
```

Y también si queremos detener la navegación:

```
procedure TFormulario.BDetenerClick( Sender: TObject );
begin
  WebBrowser.Stop;
end;
```

Hasta aquí tenemos la parte básica de la navegación. Pasemos ahora a controlar el progreso de la navegación así como que el usuario sólo pueda entrar a una página en concreto.

A la barra de progreso situada a la derecha del botón BDetener la llamaremos Progreso y por defecto estará invisible. Sólo la vamos a hacer aparecer cuando comience la navegación. Eso se hace en el evento OnBeforeNavigate2:

```
procedure TFormulario.WebBrowserBeforeNavigate2( Sender: TObject;
  const pDisp: IDispatch; var URL, Flags, TargetFrameName, PostData,
  Headers: OleVariant; var Cancel: WordBool );
begin
  if Pos( 'terra', URL ) = 0 then
    Cancel := True;

  Progreso.Show;
end;
```

Aquí le hemos dicho al evento que antes de navegar si la URL de la página web no contiene la palabra terra que cancele la navegación. Así evitamos que el usuario se distraiga en otras páginas web.

En el caso de que si pueda navegar entonces mostramos la barra de progreso de la carga de la página web. Para controlar el progreso de la navegación se utiliza el evento OnProgressChange:

```
procedure TFormulario.WebBrowserProgressChange( Sender: TObject;
Progress, ProgressMax: Integer );
begin
```

```
  Progreso.Max := ProgressMax;
  Progreso.Position := Progress;
end;
```

Cuando termine la navegación debemos ocultar de nuevo la barra de progreso. Eso lo hará en el evento OnDocumentComplete:

```
procedure TFormulario.WebBrowserDocumentComplete( Sender: TObject;
  const pDisp: IDispatch; var URL: OleVariant );
begin
  Progreso.Hide;
end;
```

Con esto ya tenemos un sencillo navegador que sólo accede a donde nosotros le digamos. A partir de aquí podemos ampliarle muchas más características tales como memorizar las URL favoritas, permitir visualizar a pantalla completa (FullScreen) e incluso permitir múltiples ventanas de navegación a través de pestañas (con el componente PageControl).

También se pueden bloquear ventanas emergentes utilizando el evento OnNewWindow2 evitando así las asquerosas ventanas de publicidad:

```
procedure TFormulario.WebBrowserNewWindow2(Sender: TObject;
  var ppDisp: IDispatch; var Cancel: WordBool);
begin
  Cancel := True;
end;
```

Aunque últimamente los publicistas muestran la publicidad en capas mediante hojas de estilo en cascada, teniendo que utilizar otros programas más avanzados para eliminar publicidad.

Pruebas realizadas en Delphi 7.

Creando consultas SQL con parámetros

En el artículo anterior vimos como realizar consultas SQL para INSERT, DELETE, UPDATE y SELECT utilizando el componente IBSQL que forma parte de la paleta de componentes IBExpress.

También quedó muy claro que la velocidad de ejecución de consultas con este componente respecto a otros como IBQuery es muy superior. Todo lo que hemos visto esta bien para hacer consultas esporádicas sobre alguna tabla que otra, pero ¿que ocurre si tenemos que realizar miles de consultas SQL de una sola vez?

UTILIZANDO UNA TRANSACCION POR CONSULTA

Supongamos que tenemos que modificar el nombre de 1000 registros de la tabla CLIENTES:

```
var
  i: Integer;
  dwTiempo: DWord;
begin
  with Consulta do
  begin
    //////////////  METODO LENTO  ///////////////

    dwTiempo := TimeGetTime;

    for i := 1 to 1000 do
    begin
      SQL.Clear;
      SQL.Add( 'UPDATE CLIENTES' );
      SQL.Add( 'SET NOMBRE = ' + QuotedStr( 'NOMBRE CLIENTE Nº ' +
IntToStr( i ) ) );
      SQL.Add( 'WHERE ID = ' + IntToStr( i ) );

      Transaction.StartTransaction;

      try
        ExecQuery;
        Transaction.Commit;
      except
        on E: Exception do
        begin
          Application.MessageBox( PChar( E.Message ), 'Error de SQL',
MB_ICONSTOP );
          Transaccion.Rollback;
        end;
      end;
    end;

    ShowMessage( 'Tiempo: ' + IntToStr( TimeGetTime - dwTiempo ) + '
milisegundos' );
  end;
end;
```

Como puede verse arriba, por cada cliente actualizado he generado una SQL distinta abriendo y cerrando una transacción para cada registro. He utilizado la función TimeGetTime que se encuentra en la unidad MMSystem para calcular el tiempo que tarda en actualizarme el nombre de los 1000 clientes. En un PC con Pentium 4 a 3 Ghz, 1 GB de RAM y utilizando el motor de bases de datos Firebird 2.0 me ha tardado 4167 milisegundos.

Aunque las consultas SQL van muy rápidas con los componentes IBSQL aquí el fallo que cometemos es que por cada registro actualizado se abre y se cierra una transacción. En una base de datos local no se nota mucho pero en una red local con muchos usuarios trabajando a la vez le puede pegar fuego al concentrador.

Lo ideal sería poder modificar la SQL pero sin tener que cerrar la transacción.

Como eso no se puede hacer en una consulta que esta abierta entonces hay que utilizar los parámetros. Los parámetros (Params) nos permiten enviar y recoger información de una consulta SQL que se esta ejecutando sin tener que cerrarla y abrila.

UTILIZANDO PARAMETROS EN LA CONSULTA

Para introducir parámetros en una consulta SQL hay que añadir dos puntos delante del parámetro. Por ejemplo:

```
UPDATE CLIENTES
SET NOMBRE = :NOMBRE
WHERE ID = :ID
```

Esta consulta tiene dos parámetros: ID y NOMBRE. Los nombres de los parámetros no tienen porque coincidir con el nombre del campo. Bien podrían ser así:

```
UPDATE CLIENTES
SET NOMBRE = :NUEVONOMBRE
WHERE ID = :IDACTUAL
```

De este modo se pueden modificar las condiciones de la consulta SQL sin tener que cerrar la transacción. Después de crear la consulta SQL hay que llamar al método Prepare para que prepare la consulta con los futuros parámetros que se le van a suministrar (no es obligatorio pero si recomendable). Veamos el ejemplo anterior utilizando parámetros y una sóla transacción para los 1000 registros:

```
var
  i: Integer;
  dwTiempo: DWord;
begin
  with Consulta do
  begin
    //////////////  METODO RÁPIDO  ////////////////

    dwTiempo := TimeGetTime;

    Transaction.StartTransaction;

    SQL.Clear;
    SQL.Add( 'UPDATE CLIENTES' );
    SQL.Add( 'SET NOMBRE = :NOMBRE' );
    SQL.Add( 'WHERE ID = :ID' );
    Prepare;

    for i := 1 to 1000 do
    begin
      Params.ByName( 'NOMBRE' ).AsString := 'NOMBRE CLIENTE Nº '+
IntToStr( i );
      Params.ByName( 'ID' ).AsInteger := i;
      ExecQuery;
    end;

    try
      Transaction.Commit;
```

```
    except
      on E: Exception do
      begin
        Application.MessageBox( PChar( E.Message ), 'Error de SQL',
MB_ICONSTOP );
        Transaccion.Rollback;
      end;
    end;

    ShowMessage( 'Tiempo: ' + IntToStr( TimeGetTime - dwTiempo ) + '
milisegundos' );
  end;
end;
```

En esta ocasión me ha tardado sólo 214 milisegundos, es decir, se ha reducido al 5% del tiempo anterior sin saturar al motor de bases de datos abriendo y cerrando transacciones sin parar.

Este método puede aplicarse también para consultas con INSERT, SELECT y DELETE. En lo único en lo que hay que tener precaución es en no acumular muchos datos en la transacción, ya que podría ser peor el remedio que la enfermedad.

Si teneis que actualizar cientos de miles de registros de una sola vez, recomiendo realizar un Commit cada 1000 registros para no saturar la memoria caché de la transacción. Todo depende del número de campos que tengan las tablas así como el número de registros a modificar. Utilizad la función TimeGetTime para medir tiempos y sacar conclusiones.

Y si el proceso a realizar va a tardar más de 2 o 3 segundos utilizar barras de progreso e hilos de ejecución, ya que algunos usuarios neuróticos podrían creer que nuestro programa se ha colgado y empezarían ha hacer clic como posesos (os aseguro que existe gente así, antes de que termine la consulta SQL ya te están llamando por teléfono echándote los perros).

Pruebas realizadas con Firebird 2.0 y Dephi 7

Creando consultas SQL rápidas con IBSQL

Cuando se crea un programa para el mantenimiento de tablas de bases de datos (clientes, artículos, etc.) el método ideal es utilizar componentes ClientDataSet como vimos anteriormente.

Pero hay ocasiones en las que es necesario realizar consultas rápidas en el servidor tales como incrementar existencias en almacén, generar recibos automáticamente o incluso incrementar nuestros contadores de facturas sin utilizar generadores.

En ese caso el componente más rápido para bases de datos Interbase/Firebird es IBSQL el cual permite realizar consultas SQL sin que estén vinculadas a ningún componente visual. Vamos a ver unos ejemplos de inserción,

modificación y eliminación de registros utilizando este componente.

INSERTANDO REGISTROS EN UNA TABLA

Aquí tenemos un ejemplo de insertar un registro en una tabla llamada CLIENTES utilizando un objeto IBSQL llamado Consulta:

```
  with Consulta do
  begin
    SQL.Clear;
    SQL.Add( 'INSERT INTO CLIENTES' );
    SQL.Add( '( NOMBRE, NIF, IMPORTEPTE )' );
    SQL.Add( 'VALUES' );
    SQL.Add( '( ''ANTONIO GARCIA LOPEZ'', ''46876283D'', 140.23 )' );

    Transaction.StartTransaction;

    try
      ExecQuery;
      Transaction.Commit;
    except
      on E: Exception do
      begin
        Application.MessageBox( PChar( E.Message ), 'Error de SQL',
MB_ICONSTOP );
        Transaccion.Rollback;
      end;
    end;
  end;
```

Como puede apreciarse hemos tenido que abrir nosotros a mano la transacción antes de ejecutar la consulta ya que el objeto IBSQL no la abre automáticamente tal como ocurre en los componentes IBQuery.

Una vez ejecutada la consulta, si todo ha funcionado correctamente enviamos la transacción al servidor mediante el método Commit. En el caso de que falle mostramos el error y cancelamos la transacción utilizando el método RollBack.

MODIFICANDO LOS REGISTROS DE UNA TABLA

El método para modificar los registros es el mismo que para insertarlos:

```
  with Consulta do
  begin
    SQL.Clear;
    SQL.Add( 'UPDATE CLIENTES' );
    SQL.Add( 'SET NOMBRE = ''MARIA GUILLEN ROJO'',' );
    SQL.Add( 'NIF = ''69236724W'', ' );
    SQL.Add( 'IMPORTEPTE = 80.65' );
    SQL.Add( 'WHERE ID=21963' );

    Transaction.StartTransaction;

    try
```

```
      ExecQuery;
      Transaction.Commit;
    except
      on E: Exception do
      begin
        Application.MessageBox( PChar( E.Message ), 'Error de SQL',
MB_ICONSTOP );
        Transaccion.Rollback;
      end;
    end;
  end;
```

ELIMINANDO REGISTROS DE LA TABLA

Al igual que para las SQL para INSERT y UPDATE el procedimiento es el mismo:

```
  with Consulta do
  begin
    SQL.Clear;
    SQL.Add( 'DELETE FROM CLIENTES' );
    SQL.Add( 'WHERE ID=21964' );

    Transaction.StartTransaction;

    try
      ExecQuery;
      Transaction.Commit;
    except
      on E: Exception do
      begin
        Application.MessageBox( PChar( E.Message ), 'Error de SQL',
MB_ICONSTOP );
        Transaccion.Rollback;
      end;
    end;
  end;
```

CONSULTANDO LOS REGISTROS DE UNA TABLA

El método de consultar los registros de una tabla mediante SELECT difiere de los que hemos utilizado anteriormente ya que tenemos que dejar la transacción abierta hasta que terminemos de recorrer todos los registros:

```
  with Consulta do
  begin
    SQL.Clear;
    SQL.Add( 'SELECT * FROM CLIENTES' );
    SQL.Add( 'ORDER BY ID' );

    Transaction.StartTransaction;

    // Ejecutarmos consulta
    try
      ExecQuery;
    except
      on E: Exception do
      begin
        Application.MessageBox( PChar( E.Message ), 'Error de SQL',
MB_ICONSTOP );
```

```
        Transaccion.Rollback;
      end;
    end;

    // Recorremos los registros
    while not Eof do
    begin
      Memo.Lines.Add( FieldByName( 'NOMBRE' ).AsString );
      Next;
    end;

    // Cerramos la consulta y la transacción
    Close;
    Transaction.Active := False;
  end;
```

Aunque pueda parecer un coñazo el componente de la clase TIBSQL respesto a los componentes TIBTable o TIBQuery, su velocidad es muy superior a ambos componentes, sobre todo cuando se ejecutan las consultas sucesivamente.

En el próximo artículo veremos cómo utilizar parámetros en las consultas.

Pruebas realizadas en Firebird 2.0 y Delphi 7.

Como poner una imagen de fondo en una aplicación MDI

En un artículo anterior vimos como crear aplicaciones MDI gestionando múltiples ventanas hijas dentro de la ventana padre.

Una de las cosas que más dan vistosidad a una aplicación comercial es tener un fondo con nuestra marca de fondo de la aplicación (al estilo Contaplus o Facturaplus).

Para introducir una imagen de fondo en la ventana padre MDI hay que hacer lo siguiente:

- Introducir en la ventana padre (la que tiene la propiedad FormStyle a MDIForm) un componente de la clase TImage situado en la pestaña Additional. Al componenente lo vamos a llamar Fondo.

- En dicha imagen vamos a cambiar la propidedad Align a alClient para que ocupe todo el fondo del formulario padre.

- Ahora sólo falta cargar la imagen directamente:

```
Fondo.Picture.LoadFromFile( 'c:\imagenes\fondo.bmp' );
```

El único inconveniente que tiene esto es que no podemos utilizar los eventos del formulario al estar la imagen encima (Drag and Drop, etc).

UTILIZANDO EL CANVAS

Otra forma de hacerlo sería poniendo el objeto TImage en medio del formulario pero de manera invisible (sin alClient). Después en el evento OnPaint del formulario copiamos el contenido de la imagen TImage al fondo del formulario:

```
procedure TFormulario.FormPaint( Sender: TObject );
var R: TRect;
begin
  R.Left := 0;
  R.Top := 0;
  R.Right := Fondo.Width;
  R.Bottom := Fondo.Height;
  Canvas.CopyRect( R, Fondo.Canvas, R );
end;
```

Así podemos tener igualmente una imagen de fondo sin renunciar a los eventos del formulario (OnMouseMove, OnClick, etc.).

Pruebas realizadas en Dephi 7.

Leyendo los metadatos de una tabla de Interbase/Firebird

No hay nada que cause más pereza a un programador que la actualización de campos en las bases de datos de nuestros clientes. Hay muchas maneras de hacerlo: desde archivos de texto con metadatos, archivos SQL o incluso actualizaciones por Internet con FTP o correo electrónico.

Yo lo que suelo hacer es tener un archivo GDB o FDB (según sea Interbase o Firebird respectivamente) que contiene todas las bases de datos vacías. Si tengo que ampliar algún campo lo hago sobre esta base de datos.

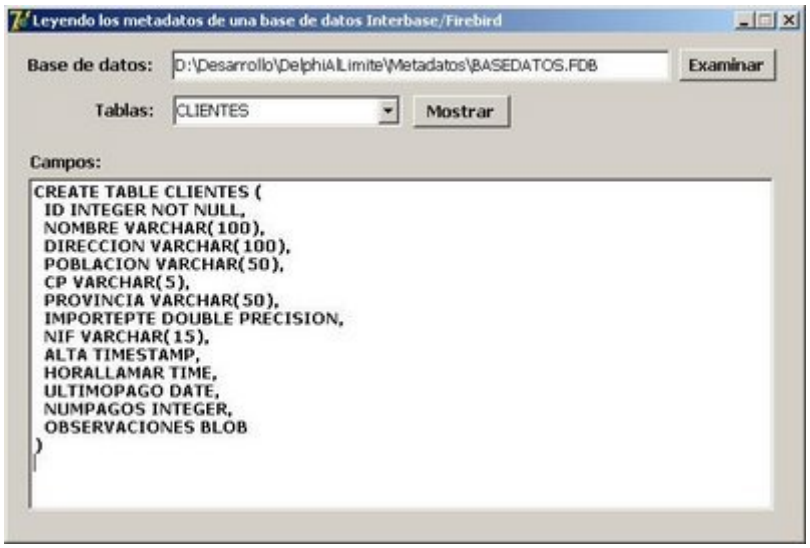
Después cuando tengo que actualizar al cliente lo que hago es mandarle mi GDB vacío y mediante un pequeño programa que tengo hecho compara las estructuras de la base de datos de mi GDB vacío y las del cliente, creando tablas y campos nuevos según las diferencias sobre ambas bases de datos.

Ahora bien, ¿Cómo podemos leer el tipo de campos de una tabla? Pues en principio tenemos que el componente IBDatabase tiene la función GetTableNames que devuelve el nombre de las tablas de una base de datos y la función GetFieldNames que nos dice los campos pertenecientes a una tabla en concreto. El problema radica en que no me dice que tipo de campo es (float, string, blob, etc).

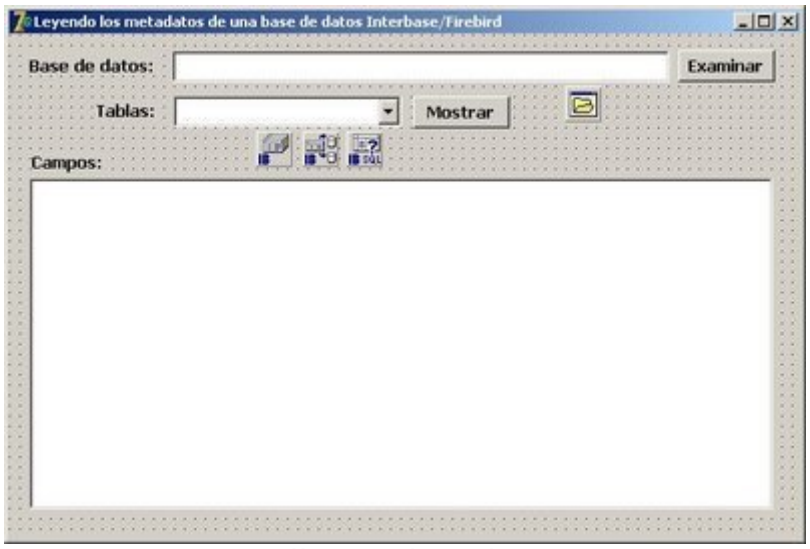
LEYENDO LOS CAMPOS DE UNA TABLA

Para leer los campos de una tabla utilizo el componente de la clase TIBQuery situado en la pestaña Interbase. Cuando este componente abre una tabla carga el nombre de los campos y su tipo en su propiedad FieldDefs. Voy a

realizar una aplicación sencilla en la cual seleccionamos una base de datos Interbase o Firebird y cuando se elija una tabla mostrará sus metadatos de esta manera:



Va a contener los siguientes componentes:



- Un componente Edit con el nombre RUTADB que va a guardar la ruta de la base de datos.

- Un componente de la clase TOpenDialog llamado AbrirBaseDatos para buscar la base de datos Firebird/Interbase.

- Un botón llamado BExaminar.

- Un componente ComboBox llamado TABLAS que guardará el nombre de las tablas de la base de datos.

- Un componente IBDatabase llamado BaseDatos que utilizaremos para conectar.

- Un componente IBTransaction llamado Transaccion para asociarlo a la base

de datos.

- Un componente IBQuery llamado IBQuery que utilizaremos para abrir una tabla.

- Y por último un campo Memo llamado Memo para mostrar la información de los metadatos.

Comenzemos a asignar lo que tiene que hacer cada componente:

- Hacemos doble clic sobre el componente BaseDatos y le damos de usuario SYSDBA y password masterkey. Pulsamos Ok y desactivamos su propiedad LoginPrompt.

- Asignamos el componente Transaccion a los componentes BaseDatos y IBQuery.

- Asignamos el componente BaseDatos a los componentes Transaccion y IBQuery.

- En la propiedad Filter del componente AbrirBaseDatos ponemos:

Interbase|*.gdb|Firebird|*.fdb

- Al pulsar el botón Examinar hacemos lo siguiente:

```
procedure TFormulario.BExaminarClick( Sender: TObject );
begin
  if AbrirBaseDatos.Execute then
  begin
    RUTADB.Text := AbrirBaseDatos.FileName;
    BaseDatos.DatabaseName := '127.0.0.1:' + RUTADB.Text;

    try
      BaseDatos.Open;
    except
      on E: Exception do
        Application.MessageBox( PChar( E.Message ), 'Error al abrir
base de datos',
          MB_ICONSTOP );
    end;

    BaseDatos.GetTableNames( TABLAS.Items, False );
  end;
end;
```

Lo que hemos hecho es abrir la base de datos y leer el nombre de las tablas que guardaremos dentro del ComboBox llamado TABLAS.

Cuando el usuario seleccione una tabla y pulse el botón Mostrar hacemos lo siguiente:

```
procedure TFormulario.BMostrarClick( Sender: TObject );
var
  i: Integer;
  sTipo: String;
begin
  with IBQuery do
  begin
    Memo.Lines.Add( 'CREATE TABLE ' + TABLAS.Text + ' (' );

    SQL.Clear;
    SQL.Add( 'SELECT * FROM ' + TABLAS.Text );
    Open;

    for i := 0 to FieldDefs.Count - 1 do
    begin
      sTipo := '';

      if FieldDefs.Items[i].FieldClass.ClassName = 'TIBStringField'
then
        sTipo := 'VARCHAR(' + IntToStr( FieldByName(
FieldDefs.Items[i].Name ).Size ) + ')';

      if FieldDefs.Items[i].FieldClass.ClassName = 'TFloatField' then
        sTipo := 'DOUBLE PRECISION'; // También podría ser FLOAT (32
bits) aunque prefiero DOUBLE (64 bits)

      if FieldDefs.Items[i].FieldClass.ClassName = 'TIntegerField'
then
        sTipo := 'INTEGER';

      if FieldDefs.Items[i].FieldClass.ClassName = 'TDateField' then
        sTipo := 'DATE';

      if FieldDefs.Items[i].FieldClass.ClassName = 'TTimeField' then
        sTipo := 'TIME';

      if FieldDefs.Items[i].FieldClass.ClassName = 'TDateTimeField'
then
        sTipo := 'TIMESTAMP';

      if FieldDefs.Items[i].FieldClass.ClassName = 'TBlobField' then
        sTipo := 'BLOB';

      // ¿Es un campo obligatorio?
      if FieldByName( FieldDefs.Items[i].Name ).Required then
        sTipo := sTipo + ' NOT NULL';

      Memo.Lines.Add( '  ' + FieldDefs.Items[i].Name + ' ' + sTipo );

      // Si no es el último campo añadimos una coma al final
      if i < FieldDefs.Count - 1 then
        Memo.Lines[Memo.Lines.Count-1] := Memo.Lines[Memo.Lines.Count-
1] + ',';
    end;

    Memo.Lines.Add( ')' );

    Close;
    Transaction.Active := False;
```

```
  end;
end;
```

Lo que hace este procedimiento es abrir con el componente IBQuery la tabla seleccionada y según los tipos de campos creamos la SQL de creación de la tabla.

Este método también nos podría ser útil para hacer un programa que copie datos entre tablas Interbase/Firebird.

Pruebas realizadas con Firebird 2.0 y Delphi 7.

Generando números aleatorios

Las funciones de las que disponen los lenguajes de programación para generar números aleatorios se basan en una pequeña semilla según la fecha del sistema y a partir de ahí se van aplicando una serie de fórmulas se van generando números al azar según los milisegundos que lleva el PC arrancado.

Dephi dispone de la función Random para generar números aleatorios entre 0 y el parámetro que se le pase. Pero para que la semilla no sea siempre la misma es conveniente inicializarla utilizando el procedimiento Randomize.

Por ejemplo, si yo quisiera inventarme 10 números del 1 y 100 haría lo siguiente:

```
procedure TFormulario.Inventar10Numeros;
var
  i: Integer;
begin
  Randomize;

  for i := 1 to 10 do
    Memo.Lines.Add( IntToStr( Random( 100 ) + 1 ) );
end;
```

El resultado lo he volcado a un campo Memo.

INVENTANDO LOS NUMEROS DE LA LOTO

Supongamos que quiero hacer el típico programa que genera automáticamente las combinaciones de la loto. El método es tan simple como hemos visto anteriormente:

```
procedure TFormulario.InventarLoto;
var
  i: Integer;
begin
  Randomize;

  for i := 1 to 6 do
    Memo.Lines.Add( IntToStr( Random( 49 ) + 1 ) );
end;
```

Pero así como lo genera es algo chapucero. Primero tenemos el problema de que los números inventados no los ordena y luego podría darse el caso de el ordenador se invente un número dos veces.

Para hacerlo como Dios manda vamos a crear la clase TSorteo encargada de inventarse los 6 números de la loto y el complementario. Además lo vamos a hacer como si fuera de verdad, es decir, vamos a crear un bombo, le vamos a introducir las 49 bolas y el programa las va a agitar y sacará una al azar. Y por último también daremos la posibilidad de excluir ciertos números del bombo (por ejemplo el 1 y 49 son los que menos salen por estadística).

Comencemos creando la clase TSorteo en la sección type:

```
type
  TSorteo = class
  public
    Fecha: TDate;
    Numeros: TStringList;
    Complementario: String;
    Excluidos: TStringList;

    constructor Create;
    destructor Destroy; override;
    procedure Inventar;
    procedure Excluir( sNumero: String );
  end;
```

Como podemos ver en la clase los números inventados y los excluidos los voy a meter en un StringList. El constructor de la clase TSorteo va a crear ambos StringList:

```
constructor TSorteo.Create;
begin
  Numeros := TStringList.Create;
  Excluidos := TStringList.Create;
end;
```

Y el destructor los liberará de memoria:

```
destructor TSorteo.Destroy;
begin
  Excluidos.Free;
  Numeros.Free;
end;
```

Nuestra clase TSorteo también va a incluir un método para excluir del sorteo el número que queramos:

```
procedure TSorteo.Excluir( sNumero: String );
begin
  // Antes de excluirlo comprobamos si ya lo esta
  if Excluidos.IndexOf( sNumero ) = -1 then
    Excluidos.Add( sNumero );
end;
```

Y aquí tenemos la función que se inventa el sorteo evitando los excluidos:

```
procedure TSorteo.Inventar;
var
  Bombo: TStringList;
  i, iPos1, iPos2: Integer;
  sNumero, sBola: String;
begin
  // Metemos las 49 bolas en el bombo
  Bombo := TStringList.Create;
  Numeros.Clear;

  for i := 1 to 49 do
  begin
    sNumero := CompletarCodigo( IntToStr( i ), 2 );

    if Excluidos.IndexOf( sNumero ) = -1 then
      Bombo.Add( sNumero );
  end;

  // Agitamos las bolas con el método de la burbuja
  if Bombo.Count > 0 then
    for i := 1 to 10000 + Random( 10000 ) do
    begin
      // Nos inventamos dos posiciones distintas en el bombo
      iPos1 := Random( Bombo.Count );
      iPos2 := Random( Bombo.Count );

      if ( iPos1 >= 0 ) and ( iPos1 <= 49 ) and ( iPos2 >= 0 ) and (
iPos2 <= 49 ) then
      begin
        // Intercambiamos las bolas en esas dos posiciones inventadas
        sBola := Bombo[iPos1];
        Bombo[iPos1] := Bombo[iPos2];
        Bombo[iPos2] := sBola;
      end;
    end;

  // Vamos sacando las 6 bolas al azar + complementario
  for i := 0 to 6 do
  begin
    if Bombo.Count > 0 then
      iPos1 := Random( Bombo.Count )
    else
      iPos1 := -1;

    if ( iPos1 >= 0 ) and ( iPos1 <= 49 ) then
      sBola := Bombo[iPos1]
    else
      sBola := '';

    // ¿Es el complementario?
    if i = 6 then
      // Lo sacamos aparte
      Complementario := sBola
    else
      // Lo metemos en la lista de números
      Numeros.Add( sBola );

    // Sacamos la bola extraida del bombo
```

```
    if ( iPos1 >= 0 ) and ( iPos1 <= 49 ) and ( Bombo.Count > 0 ) then
      Bombo.Delete( iPos1 );
  end;

  // Ordenamos los 6 números
  Numeros.Sort;

  Bombo.Free;
end;
```

El procedimiento Inventar hace lo siguiente:

1º Crea un bombo dentro de un StringList y le mete las 49 bolas.

2º Elimina los número excluidos si los hay (los excluidos hay que meterlos en dos cifras, 01, 07, etc.)

3º Agita los números dentro del bombo utilizando del método de la burbuja para que queden todas las bolas desordenadas.

4º Extrae las 6 bolas y el complementario eligiendo a azar dos posiciones del StringList para hacerlo todavía mas rebuscado. Al eliminar la bola extraida evitamos así números repetidos, tal como si fuera el sorteo real.

5º Una vez inventados los números los deposita en el StringList llamado Numeros y elimina de memoria el Bombo.

Ahora vamos a utilizar nuestra clase TSorteo para generar una combinación:

```
procedure TFormulario.InventarSorteo;
var
  S: TSorteo;
begin
  Randomize;
  S := TSorteo.Create;
  S.Inventar;
  Memo.Lines.Add( S.Numeros.Text );
  S.Free;
end;
```

Si quisiera excluir del sorteo los número 1 y 49 haría lo siguiente:

```
var
  S: TSorteo;
begin
  Randomize;
  S := TSorteo.Create;
  S.Excluir( '01' );
  S.Excluir( '49' );
  S.Inventar;
  Memo.Lines.Add( S.Numeros.Text );
  S.Free;
end;
```

Este es un método simple para generar los números de la loto pero las variantes que se pueden hacer del mismo son infinitas. Ya depende de la

imaginación de cada cual y del uso que le vaya a dar al mismo.

Igualmente sería sencillo realizar algunas modificaciones para inventar otros sorteos tales como el gordo de la primitiva, el sorteo de los euromillones o la quiniela de fútbol.

Moviendo sprites con el teclado y el ratón

Basándome en el ejemplo del artículo anterior que mostraba como realizar el movimiento de sprites con fondo vamos a ver como el usuario puede mover los sprites usando el teclado y el ratón.

CAPTURANDO LOS EVENTOS DEL TECLADO

La clase TForm dispone de dos eventos para controlar las pulsaciones de teclado: OnKeyDown y OnKeyUp. Necesitamos ambos eventos porque no sólo me interesa saber cuando un usuario ha pulsado una tecla sino también cuando la ha soltado (para controlar las diagonales).

Para hacer esto voy a crear cuatro variables booleanas en la sección private el formulario que me van a informar de cuando están pulsadas las teclas del cursor:

```
type
  private
    { Private declarations }
    Sprite: TSprite;
    Buffer, Fondo: TImage;
    bDerecha, bIzquierda, bArriba, bAbajo: Boolean;
```

Estas variables las voy a actualizar en el evento OnKeyDown:

```
procedure TFormulario.FormKeyDown( Sender: TObject; var Key: Word;
Shift: TShiftState );
begin
  case key of
    VK_LEFT: bIzquierda := True;
    VK_DOWN: bAbajo := True;
    VK_UP: bArriba := True;
    VK_RIGHT: bDerecha := True;
  end;
end;
```

y en el evento OnKeyUp:

```
procedure TFormulario.FormKeyUp( Sender: TObject; var Key: Word;
Shift: TShiftState );
begin
  case key of
    VK_LEFT: bIzquierda := False;
    VK_DOWN: bAbajo := False;
    VK_UP: bArriba := False;
    VK_RIGHT: bDerecha := False;
```

```
  end;
end;
```

Al igual que hice con el ejemplo anterior voy a utilizar un temporizador (TTimer) llamado TmpTeclado con un intervalo que va a ser también de 10 milisegundos y cuyo evento OnTimer va a encargarse de dibujar el sprite en pantalla:

```
procedure TFormulario.TmpTecladoTimer( Sender: TObject );
begin
  // ¿Ha pulsado la tecla izquierda?
  if bIzquierda then
    if Sprite.x > 0 then
      Dec( Sprite.x );

  // ¿Ha pulsado la tecla arriba?
  if bArriba then
    if Sprite.y > 0 then
      Dec( Sprite.y );

  // ¿Ha pulsado la tecla derecha?
  if bDerecha then
    if Sprite.x + Sprite.Imagen.Width < ClientWidth then
      Inc( Sprite.x );

  // ¿Ha pulsado la tecla abajo?
  if bAbajo then
    if Sprite.y + Sprite.Imagen.Height < ClientHeight then
      Inc( Sprite.y );

  DibujarSprite;
end;
```

Este evento comprueba la pulsación de todas las teclas controlando que el sprite no se salga del formulario. El procedimiento de DibujarSprite sería el siguiente:

```
procedure TFormulario.DibujarSprite;
var
  Origen, Destino: TRect;
begin
  // Copiamos el fondo de pantalla al buffer
  Origen.Left := Sprite.x;
  Origen.Top := Sprite.y;
  Origen.Right := Sprite.x + Sprite.Imagen.Width;
  Origen.Bottom := Sprite.y + Sprite.Imagen.Height;
  Destino.Left := 0;
  Destino.Top := 0;
  Destino.Right := Sprite.Imagen.Width;
  Destino.Bottom := Sprite.Imagen.Height;
  Buffer.Canvas.CopyMode := cmSrcCopy;
  Buffer.Canvas.CopyRect( Destino, Fondo.Canvas, Origen );

  // Dibujamos el sprite en el buffer encima del fondo copiado
  Sprite.Dibujar( 0, 0, Buffer.Canvas );

  // Dibujamos el contenido del buffer a la pantalla
  Canvas.Draw( Sprite.x, Sprite.y, Buffer.Picture.Graphic );
end;
```

Prácticamente es el mismo visto en el artículo anterior. Ya sólo hace falta poner el marcha el mecanismo que bien podría ser en el evento OnCreate del formulario:

```
begin
  TmpTeclado.Enabled := True;
  Sprite.x := 250;
  Sprite.y := 150;
end;
```

CAPTURANDO LOS EVENTOS DEL RATON

Para capturar las coordenadas del ratón vamos a utilizar el evento OnMouseMove del formulario:

```
procedure TFormulario.FormMouseMove( Sender: TObject; Shift:
TShiftState; X, Y: Integer );
begin
  Sprite.x := X;
  Sprite.y := Y;
end;
```

Para controlar los eventos del ratón voy a utilizar un temporizador distinto al del teclado llamado TmpRaton con un intervalo de 10 milisegundos. Su evento OnTimer sería sencillo:

```
procedure TFormulario.TmpRatonTimer( Sender: TObject );
begin
  DibujarSprite;
end;
```

Aquí nos surge un problema importante: como los movimientos del ratón son más bruscos que los del teclado volvemos a tener el problema de que el sprite nos va dejando manchas en pantalla. Para solucionar el problema tenemos que restaurar el fondo de la posición anterior del sprite antes de dibujarlo en la nueva posición..

Para ello voy a guardar en la clase TSprite las coordenadas anteriores:

```
type
  TSprite = class
  public
    x, y, xAnterior, yAnterior: Integer;
    ColorTransparente: TColor;
    Imagen, Mascara: TImage;
    constructor Create;
    destructor Destroy; override;
    procedure Cargar( sImagen: string );
    procedure Dibujar( x, y: Integer; Canvas: TCanvas );
  end;
```

Al procedimiento DibujarSprite le vamos a añadir que restaure el fondo del sprite de la posición anterior:

```
procedure TFormulario.DibujarSprite;
```

```
var
  Origen, Destino: TRect;
begin
  // Restauramos el fondo de la posición anterior del sprite
  if ( Sprite.xAnterior <> Sprite.x ) or ( Sprite.yAnterior <>
Sprite.y ) then
  begin
    Origen.Left := Sprite.xAnterior;
    Origen.Top := Sprite.yAnterior;
    Origen.Right := Sprite.xAnterior + Sprite.Imagen.Width;
    Origen.Bottom := Sprite.yAnterior + Sprite.Imagen.Height;
    Destino := Origen;
    Canvas.CopyMode := cmSrcCopy;
    Canvas.CopyRect( Destino, Fondo.Canvas, Origen );
  end;

  // Copiamos el fondo de pantalla al buffer
  Origen.Left := Sprite.x;
  Origen.Top := Sprite.y;
  Origen.Right := Sprite.x + Sprite.Imagen.Width;
  Origen.Bottom := Sprite.y + Sprite.Imagen.Height;
  Destino.Left := 0;
  Destino.Top := 0;
  Destino.Right := Sprite.Imagen.Width;
  Destino.Bottom := Sprite.Imagen.Height;
  Buffer.Canvas.CopyMode := cmSrcCopy;
  Buffer.Canvas.CopyRect( Destino, Fondo.Canvas, Origen );

  // Dibujamos el sprite en el buffer encima del fondo copiado
  Sprite.Dibujar( 0, 0, Buffer.Canvas );

  // Dibujamos el contenido del buffer a la pantalla
  Canvas.Draw( Sprite.x, Sprite.y, Buffer.Picture.Graphic );

  Sprite.xAnterior := Sprite.x;
  Sprite.yAnterior := Sprite.y;
end;
```

Y finalmente activamos el temporizador que controla el ratón y ocultamos el cursor del ratón para que no se superponga encima de nuestro sprite:

```
begin
  TmpRaton.Enabled := True;
  Sprite.x := 250;
  Sprite.y := 150;
  ShowCursor( False );
end;
```

Al ejecutar el programa podeis ver como se mueve el sprite como si fuera el cursor del ratón.

Aunque se pueden hacer cosas bonitas utilizando el Canvas no os hagais muchas ilusiones ya que si por algo destaca la librería GDI de Windows (el Canvas) es por su lentitud y por la diferencia de velocidad entre ordenadores.

Para hacer cosas serías habría que irse a la librerías SDL (mi favorita), OpenGL o DirectX (aunque hay decenas de motores gráficos 2D y 3D para Delphi en Internet que simplifican el trabajo).

Mover sprites con doble buffer

En el artículo anterior creamos la clase TSprite encargada de dibujar figuras en pantalla. Hoy vamos a reutilizarla para mover sprites, pero antes vamos a hacer una pequeña modificación:

```
type
  TSprite = class
  public
    x, y: Integer;
    ColorTransparente: TColor;
    Imagen, Mascara: TImage;
    constructor Create;
    destructor Destroy; override;
    procedure Cargar( sImagen: string );
    procedure Dibujar( x, y: Integer; Canvas: TCanvas );
  end;
```

Sólo hemos modificado el evento Dibujar añadiendo las coordenadas de donde se va a dibujar (independientemente de las que tenga el sprite). La implementación de toda la clase TSprite quedaría de esta manera:

```
{ TSprite }

constructor TSprite.Create;
begin
  inherited;
  Imagen := TImage.Create( nil );
  Imagen.AutoSize := True;
  Mascara := TImage.Create( nil );
  ColorTransparente := RGB( 255, 0, 255 );
end;

destructor TSprite.Destroy;
begin
  Mascara.Free;
  Imagen.Free;
  inherited;
end;

procedure TSprite.Cargar( sImagen: string );
var
  i, j: Integer;
begin
  Imagen.Picture.LoadFromFile( sImagen );
  Mascara.Width := Imagen.Width;
  Mascara.Height := Imagen.Height;

  for j := 0 to Imagen.Height - 1 do
    for i := 0 to Imagen.Width - 1 do
      if Imagen.Canvas.Pixels[i, j] = ColorTransparente then
      begin
```

```
        Imagen.Canvas.Pixels[i, j] := 0;
        Mascara.Canvas.Pixels[i, j] := RGB( 255, 255, 255 );
      end
      else
        Mascara.Canvas.Pixels[i, j] := RGB( 0, 0, 0 );
end;

procedure TSprite.Dibujar( x, y: Integer; Canvas: TCanvas );
begin
  Canvas.CopyMode := cmSrcAnd;
  Canvas.Draw( x, y, Mascara.Picture.Graphic );
  Canvas.CopyMode := cmSrcPaint;
  Canvas.Draw( x, y, Imagen.Picture.Graphic );
end;
```

CREANDO EL DOBLE BUFFER

Cuando se mueven figuras gráficas en un formulario aparte de producirse parpadeos en el sprite se van dejando rastros de las posiciones anteriores. Sucede algo como esto:

Para evitarlo hay muchísimas técnicas tales como el doble o triple buffer. Aquí vamos a ver como realizar un doble buffer para mover sprites. El formulario va a tener el siguiente fondo:

El fondo tiene unas dimensiones de 500x300 pixels. Para ajustar el fondo al formulario configuramos las siguientes propiedades en el inspector de objetos:

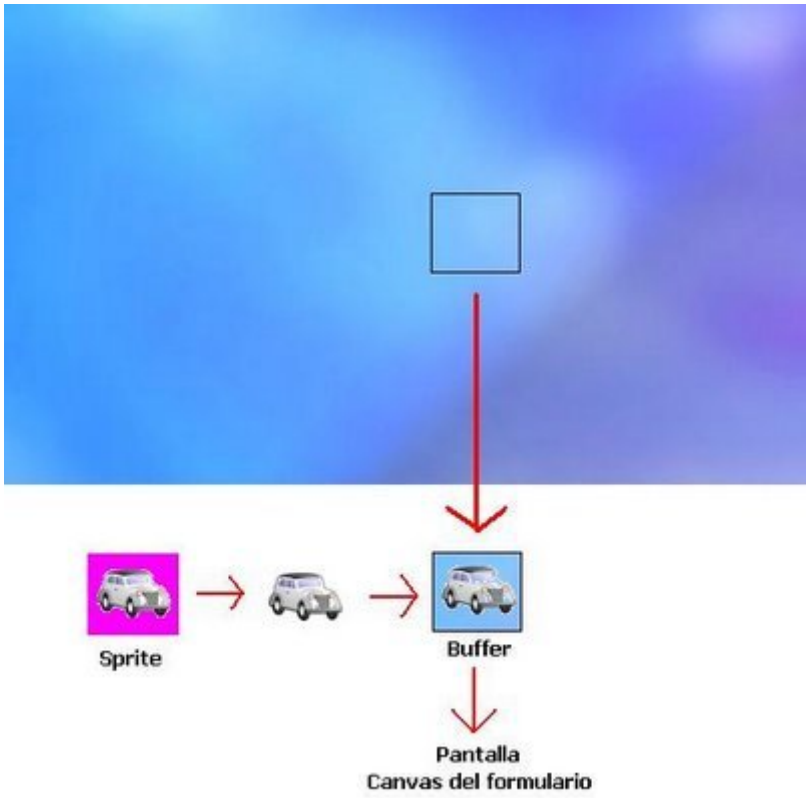
```
Formulario.ClientWidth = 500
Formulario.ClientHeight = 300
```

Se llama doble buffer porque vamos a crear dos imágenes:

```
Fondo, Buffer: TImage;
```

El Fondo guarda la imagen de fondo mostrada anteriormente y el Buffer va a encargarse de mezclar el sprite con el fondo antes de llevarlo a la pantalla.

Los pasos para dibujar el sprite serían los siguientes:



1º Se copia un trozo del fondo al buffer.

2º Se copia el sprite sin fondo encima del buffer.

3º Se lleva el contenido del buffer a pantalla.

Lo primero que vamos a hacer es declarar en la sección private del formulario los objetos:

```
  private
    { Private declarations }
    Sprite: TSprite;
    Buffer, Fondo: TImage;
```

Después los creamos en el evento OnCreate del formulario:

```
procedure TFormulario.FormCreate( Sender: TObject );
begin
  Sprite := TSprite.Create;
  Sprite.Cargar( ExtractFilePath( Application.ExeName ) + 'sprite.bmp'
);
  Buffer := TImage.Create( nil );
  Buffer.Width := Sprite.Imagen.Width;
  Buffer.Height := Sprite.Imagen.Height;
  Fondo := TImage.Create( nil );
  Fondo.Picture.LoadFromFile( ExtractFilePath( Application.ExeName ) +
'fondo.bmp' );
end;
```

El fondo también lo he creado como imagen BMP en vez de JPG para poder utilizar la función CopyRect del Canvas. Nos aseguramos de que en el evento OnDestroy del formulario se liberen de memoria:

```
procedure TFormulario.FormDestroy( Sender: TObject );
begin
  Sprite.Free;
  Buffer.Free;
  Fondo.Free;
end;
```

Aunque podemos mover el sprite utilizando un bucle for esto podría dejar nuestro programa algo pillado. Lo mejor es moverlo utilizando un objeto de la clase TTimer. Lo introducimos en nuestro formulario con el nombre Temporizador. Por defecto hay que dejarlo desactivado (Enabled = False) y vamos a hacer que se mueva el sprite cada 10 milisegundos (Invertal = 10).

En el evento OnTimer hacemos que se mueva el sprite utilizando los pasos mencionados:

```
procedure TFSprites.TemporizadorTimer( Sender: TObject );
var
  Origen, Destino: TRect;
begin
  if Sprite.x < 400 then
  begin
    Inc( Sprite.x );

    // Copiamos el fondo de pantalla al buffer
    Origen.Left := Sprite.x;
    Origen.Top := Sprite.y;
    Origen.Right := Sprite.x + Sprite.Imagen.Width;
    Origen.Bottom := Sprite.y + Sprite.Imagen.Height;
    Destino.Left := 0;
    Destino.Top := 0;
    Destino.Right := Sprite.Imagen.Width;
    Destino.Bottom := Sprite.Imagen.Height;
    Buffer.Canvas.CopyMode := cmSrcCopy;
    Buffer.Canvas.CopyRect( Destino, Fondo.Canvas, Origen );

    // Dibujamos el sprite en el buffer encima del fondo copiado
    Sprite.Dibujar( 0, 0, Buffer.Canvas );

    // Dibujamos el contenido del buffer a la pantalla
    Canvas.Draw( Sprite.x, Sprite.y, Buffer.Picture.Graphic );
  end
  else
    Temporizador.Enabled := False;
end;
```

En el evento OnPaint del formulario tenemos que hacer que se dibuje el fondo:

```
procedure TFormulario.FormPaint( Sender: TObject );
begin
  Canvas.Draw( 0, 0, Fondo.Picture.Graphic );
end;
```

Esto es necesario por si el usuario minimiza y vuelve a mostrar la aplicación, ya que sólo se refresca la zona por donde está moviéndose el sprite.

Por fin hemos conseguido el efecto deseado:

Pruebas realizadas en Delphi 7.

Como dibujar sprites transparentes

Un Sprite es una figura gráfica móvil utilizada en los videojuegos de dos dimensiones. Por ejemplo, un videojuego de naves espaciales consta de los sprites de la nave, los meteoritos, los enemigos, etc., es decir, todo lo que sea móvil en pantalla y que no tenga que ver con los paisajes de fondo.

Anteriormente vimos como copiar imágenes de una superficie a otra utilizando los métodos Draw o CopyRect que se encuentran en la clase TCanvas. También se vió como modificar el modo de copiar mediante la propiedad CopyMode la cual permitia los valores cmSrcCopy, smMergePaint, etc.

El problema radica en que por mucho que nos empeñemos en dibujar una imagen transparente ningún tipo de copia funciona: o la hace muy transparente o se estropea el fondo.

DIBUJAR SPRITES MEDIANTE MASCARAS

Para dibujar sprites transparentes hay que tener dos imágenes: la original cuyo color de fondo le debemos dar alguno como común (como el negro) y la imagen de la máscara que es igual que la original pero como si fuera un negativo.

Supongamos que quiero dibujar este sprite (archivo BMP):

Es conveniente utilizar de color transparente un color de uso como común. En este caso he elegido el color rosa cuyos componentes RGB son (255,0,255). La máscara de esta imagen sería la siguiente:

Esta máscara no es necesario crearla en ningún programa de dibujo ya que la vamos a crear nosotros internamente. Vamos a encapsular la creación del sprite en la siguiente clase:

```
type
  TSprite = class
  public
    x, y: Integer;
    ColorTransparente: TColor;
    Imagen, Mascara: TImage;
    constructor Create;
    destructor Destroy; override;
    procedure Cargar( sImagen: string );
    procedure Dibujar( Canvas: TCanvas );
  end;
```

Esta clase consta de las coordenadas del sprite, el color que definimos como transparente y las imágenes a dibujar incluyendo su máscara. En el constructor de la clase creo dos objetos de la clase TImage en memoria:

```
constructor TSprite.Create;
begin
  inherited;
  Imagen := TImage.Create( nil );
  Imagen.AutoSize := True;
  Mascara := TImage.Create( nil );
  ColorTransparente := RGB( 255, 0, 255 );
end;
```

También he definido el color rosa como color transparente. Como puede apreciarse he utilizado el procedimiento RGB que convierte los tres componentes del color al formato de TColor que los guarda al revés BGR. Así podemos darle a Delphi cualquier color utilizando estos tres componentes copiados de cualquier programa de dibujo.

En el destructor de la clase nos aseguramos de que se liberen de memoria ambos objetos:

```
destructor TSprite.Destroy;
begin
  Mascara.Free;
  Imagen.Free;
  inherited;
end;
```

Ahora implementamos la función que carga de un archivo la imagen BMP y a continuación crea su máscara:

```
procedure TSprite.Cargar( sImagen: string );
var
  i, j: Integer;
begin
```

```
  Imagen.Picture.LoadFromFile( sImagen );
  Mascara.Width := Imagen.Width;
  Mascara.Height := Imagen.Height;

  for j := 0 to Imagen.Height - 1 do
    for i := 0 to Imagen.Width - 1 do
      if Imagen.Canvas.Pixels[i,j] = ColorTransparente then
      begin
        Imagen.Canvas.Pixels[i,j] := 0;
        Mascara.Canvas.Pixels[i,j] := RGB( 255, 255, 255 );
      end
      else
        Mascara.Canvas.Pixels[i,j] := RGB( 0, 0, 0 );
end;
```

Aquí nos encargamos de dejar la máscara en negativo a partir de la imagen original.

Para dibujar sprites recomiendo utilizar archivos BMP en lugar de JPG debido a que este último tipo de imágenes pierden calidad y podrían afectar al resultado de la máscara, dando la sensación de que el sprite tiene manchas. Hay otras librerías para Delphi que permiten cargar imágenes PNG que son ideales para la creación de videojuegos, pero esto lo veremos en otra ocasión.

Una vez que ya tenemos nuestro sprite y nuestra máscara asociada al mismo podemos crear el procedimiento encargado de dibujarlo:

```
procedure TSprite.Dibujar( Canvas: TCanvas );
begin
  Canvas.CopyMode := cmSrcAnd;
  Canvas.Draw( x, y, Mascara.Picture.Graphic )
  Canvas.CopyMode := cmSrcPaint;
  Canvas.Draw( x, y, Imagen.Picture.Graphic );}
end;
```

El único parámetro que tiene es el Canvas sobre el que se va a dibujar el sprite. Primero utilizamos la máscara para limpiar el terreno y después dibujamos el sprite sin el fondo. Vamos a ver como utilizar nuestra clase para dibujar un sprite en el formulario:

```
procedure TFormulario.DibujarCoche;
var
  Sprite: TSprite;
begin
  Sprite := TSprite.Create;
  Sprite.x := 100;
  Sprite.y := 100;
  Sprite.Cargar( ExtractFilePath( Application.ExeName ) + 'sprite.bmp'
);
  Sprite.Dibujar( Canvas );
  Sprite.Free;
end;
```

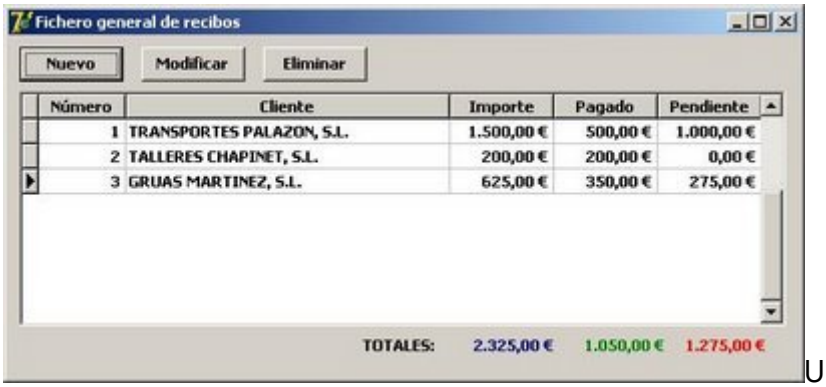
Este sería el resultado en el formulario:

Si el sprite lo vamos a dibujar muchas veces no es necesario crearlo y destruirlo cada vez. Deberíamos crearlo en el evento OnCreate del formulario y en su evento OnDestroy liberarlo (Sprite.Free).

En el próximo artículo veremos como mover el sprite en pantalla utilizando una técnica de doble buffer para evitar parpadeos en el movimiento.

Pruebas realizadas en Delphi 7.

Creando tablas de memoria con ClientDataSet



Una de las cosas que más se necesitan en un programa de gestión es la posibilidad crear tablas de memoria para procesar datos temporalmente, sobre todo cuando los datos origen vienen de tablas distintas.

Es muy común utilizar componentes de tablas de memoria tales como los que llevan los componentes RX (TRxMemoryData) o el componente kbmMemTable. Pues veamos como hacer tablas de memoria utilizando el componente de la clase TClientDataSet sin tener que utilizar ningún componente externo a Delphi.

DEFINIENDO LA TABLA

Lo primero es añadir a nuestro proyecto un componente ClientDataSet ya sea en un formulario o en un módulo de datos. Como vamos a crear una tabla de recibos lo vamos a llamar TRecibos.

Ahora vamos a definir los campos de los que se compone la tabla. Para ello pulsamos el botón [...] en la propiedad FieldDefs. En la ventana que se abre pulsamos el botón Add New y vamos creando los campos:

```
Name        DataTpe         Size
-----------------------------------
```

```
NUMERO     ftInteger      0
CLIENTE    ftString       80
IMPORTE    ftFloat        0
PAGADO     ftFloat        0
PENDIENTE  ftFloat        0
```

Para crear la tabla de memoria pulsamos el componente TClientDataSet con el botón derecho del ratón y seleccionamos Create DataSet. Una vez creado sólo nos falta hacer que los campos sean persistentes. Eso se consigue haciendo doble clic sobre el componente y pulsando la combinación de teclas CTRL + A.

Con estos sencillos pasos ya hemos creado una tabla de memoria y ya podemos abrirla para introducir datos. No es necesario abrir la tabla ya que estas tablas de memoria hay que dejarlas activas por defecto.

DANDO FORMATO A LOS CAMPOS

Como tenemos tres campos de tipo real vamos a dar formato a los mismos del siguiente modo:

1. Hacemos doble clic sobre el componente ClientDataSet.

2. Seleccionamos los campos IMPORTE, PAGADO y PENDIENTE.

3. Activamos en el inspector de objetos su propiedad Currency.

Con esto ya tenemos los campos en formato moneda y con decimales.

REALIZANDO CALCULOS AUTOMATICAMENTE

A nuestra tabla de recibos le vamos a hacer que calcule automáticamente el importe pendiente. Esto lo vamos a hacer antes de que se ejecute el Post, en el evento BeforePost:

```
procedure TFormulario.TRecibosBeforePost( DataSet: TDataSet );
begin
  TRecibosPENDIENTE.AsFloat := TRecibosIMPORTE.AsFloat -
TRecibosPAGADO.AsFloat;
end;
```

De este modo, tanto si insertamos un nuevo registro como si lo modificamos realizará el cálculo del importe pendiente antes de guardar el registro.

AÑADIENDO, EDITANDO Y ELIMINANDO REGISTROS DE LA TABLA

Insertamos tres registros:

```
begin
  TRecibos.Append;
  TRecibosNUMERO.AsInteger := 1;
  TRecibosCLIENTE.AsString := 'TRANSPORTES PALAZON, S.L.';
  TRecibosIMPORTE.AsFloat := 1500;
```

```
  TRecibosPAGADO.AsFloat := 500;
  TRecibos.Post;

  TRecibos.Append;
  TRecibosNUMERO.AsInteger := 2;
  TRecibosCLIENTE.AsString := 'TALLERES CHAPINET, S.L.';
  TRecibosIMPORTE.AsFloat := 200;
  TRecibosPAGADO.AsFloat := 200;
  TRecibos.Post;

  TRecibos.Append;
  TRecibosNUMERO.AsInteger := 3;
  TRecibosCLIENTE.AsString := 'GRUAS MARTINEZ, S.L.';
  TRecibosIMPORTE.AsFloat := 625;
  TRecibosPAGADO.AsFloat := 350;
  TRecibos.Post;
end;
```

Si queremos modificar el primer registro:

```
begin
  TRecibos.First;
  TRecibos.Edit;
  TRecibosCLIENTE.AsString := 'ANTONIO PEREZ BERNAL';
  TRecibosIMPORTE.AsFloat := 100;
  TRecibosPAGADO.AsFloat := 55;
  TRecibos.Post;
end;
```

Y para eliminarlo:

```
begin
  TRecibos.First;
  TRecibos.Delete;
end;
```

MODIFICANDO LOS CAMPOS DE LA TABLA

Si intentamos añadir un nuevo campo a la tabla en FieldDefs y luego pulsamos CTRL + A para hacer el campo persistente veremos que desaparece de la definición de campos. Para hacerlo correctamente hay que hacer lo siguiente:

1. Pulsamos el componente ClientDataSet con el botón derecho del ratón.

2. Seleccionamos Clear Data.

3. Añadimos el nuevo campo en FieldDefs.

4. Volvemos a pulsar el el botón derecho del ratón el componente y seleccionamos Create DataSet.

5. Pulsamos CTRL + A para hacer persistente el nuevo campo.

Estos son los pasos que hay que seguir si se crean, modifican o eliminan campos en la tabla.

CREANDO CAMPOS VIRTUALES PARA SUMAR COLUMNAS

Vamos a crear tres campos virtuales que sumen automáticamente el valor de las columnas IMPORTE, PAGADO y PENDIENTE para totalizarlos. Comencemos con el cálculo del importe total:

1. Pulsamos el componente ClientDataSet con el botón derecho del ratón.

2. Seleccionamos Clear Data.

3. Hacemos doble clic en el componente ClientDataSet.

4. Pulsamos la combinación de teclas CTRL + N para añadir un nuevo campo:

Name: TOTALIMPORTE
FieldType: Agregate

5. Pulsamos Ok. Seleccionamos el campo creado escribimos en su propiedad Expression:

```
SUM(IMPORTE)
```

y activamos su propiedad Active. También activamos su propiedad Currency.

6. Creamos en el formulario un campo de tipo DBText y asociamos en nuevo campo creado:

DataSource: TRecibos
DataField: TOTALIMPORTE

7. Volvemos a pulsar el el botón derecho del ratón el componente y seleccionamos Create DataSet.

8. Activamos en el componente TClientDataSet la propiedad AggregatesActive.

Igualmente habría que crear dos campos más para sumar las columnas del importe pagado y el importe pendiente.

Utilizar ClientDataSet para crear tablas de memoria es ideal para procesar listados en tablas temporales sin tener que volcar el resultado en ninguna base de datos. Además podemos importar y exportar datos a XML usando el menú contextual de este componente.

Pruebas realizadas en Delphi 7.

Cómo crear un hilo de ejecución

Hay ocasiones en que necesitamos que nuestro programa realize paralelamente algún proceso secundario que no interfiera en la aplicación principal, ya que si nos metemos en bucles cerrados o procesos pesados

(traspaso de ficheros, datos, etc.) nuestra aplicación se queda medio muerta (no se puede ni mover la ventana, minimizarla y menos cerrarla).

Para ello lo que hacemos es crear un hilo de ejecución heredando de la clase TThread del siguiente modo:

```
THilo = class( TThread )
  Ejecutar: procedure of object;
  procedure Execute; override;
end;
```

La definición anterior hay que colocarla dentro del apartado Type de nuestra unidad (en la sección interface). Le he añadido el procedimiento Ejecutar para poder mandarle que procedimiento queremos que se ejecute paralelamente.

En el apartado implementation de nuestra unidad redifinimos el procedimiento de la clase TThread para que llame a nuestro procedimiento Ejecutar:

```
procedure THilo.Execute;
begin
  Ejecutar;
  Terminate;
end;
```

Con esto ya tenemos nuestra clase THilo para crear todos los hilos de ejecución que nos de la gana. Ahora vamos a ver como se crea un hilo y se pone en marcha:

```
var
Hilo: THilo; // variable global o pública

procedure CrearHilo;
begin
  Hilo.Ejecutar := ProcesarDatos;
  Hilo.Priority := tpNormal;
  Hilo.Resume;
end;

procedure ProcesarDatos;
begin
  // Este es el procedimiento que ejecutará nuestro hilo
  // Cuidado con hacer procesos críticos aquí
  // El procesamiento paralelo de XP no es el de Linux
  // Se puede ir por las patas abajo...
end;
```

Si en cualquier momento queremos detener la ejecución del hilo:

```
Hilo.Terminate;
FreeAndNil( Hilo );
```

Los hilos de ejecución sólo conviene utilizarlos en procesos críticos e

importantes. No es conveniente utilizarlos así como así ya que se pueden comer al procesador por los piés.

Pruebas realizadas en Delphi 7

Conectando a pelo con INTERBASE o FIREBIRD

Aunque Delphi contiene componentes para mostrar directamente datos de una tabla, en ocasiones nos obligan a mostrar el contenido de una base de datos en una página web o en una presentación multimedia con SDL, OPENGL ó DIRECTX. En este caso, los componentes de la pestaña DATA CONTROLS no nos sirven de nada. Nos los tenemos que currar a mano.

Voy a mostraros un ejemplo de conexión con una base de datos de INTERBASE o FIREBIRD mostrando el resultado directamente dentro de un componente ListView, aunque con unas modificaciones se puede lanzar el resultado a un archivo de texto, página web, XML o lo que sea.

Lo primero es conectar con la base de datos:

```
function ConectarBaseDatos( sBaseDatos: String ): TIBDatabase;
var DB: TIBDatabase;
begin  DB := TIBDatabase.Create( nil );
  DB.Name := 'IB';
  DB.DatabaseName := '127.0.0.1:' + sBaseDatos;
  DB.Params.Add( 'user_name=SYSDBA' );
  DB.Params.Add( 'password=masterkey' );
  DB.SQLDialect := 3;
  DB.LoginPrompt := False;
  try
   DB.Open;
  except
   raise Exception.Create( 'No puedo conectar con INTERBASE/FIREBIRD.'
+ #13 + #13 + 'Consulte con el administrador del programa.' );
  end;

  Result := DB;
end;
```

Si nos fijamos en el procedimiento, primero se crea en tiempo real un componente de conexión a bases de datos TIBDatabase. Después le decimos con que IP va a conectar (en principio en nuestra misma máquina) y la ruta de la base de datos que es la que se le pasa al procedimiento.

Más adelante le damos el usuario y password por defecto y desactivamos en Login. Finalmente contectamos con la base de datos controlando la excepción si casca.

Un ejemplo de conexión sería:

```
var DB: TIBDatabase;
```

```
DB := ConectarBaseDatos( 'c:\bases\bases.gdb' ); // PARA INTERBASE Ó
DB := ConectarBaseDatos( 'c:\bases\bases.fdb' ); // PARA FIREBIRD

if DB = nil then
  Exit;
```

Una vez conectados a la base de datos vamos a ver como listar los registros de una tabla dentro de un ListView:

```
procedure ListarTabla( DB: TIBDatabase; sTabla: String; Listado:
TListView );
var Campos: TStringList;
  i: Integer;
  Consulta: TIBSQL;
  Transaccion: TIBTransaction;
begin
  if DB = nil then Exit;

  // Creamos un stringlist para meter los campos de la tabla
  Campos := TStringList.Create;
  DB.GetFieldNames( sTabla, Campos );

  // Creamos una transacción para la consulta
  Transaccion := TIBTransaction.Create( nil );
  Transaccion.DefaultDatabase := DB;

  // Creamos una consulta
  Consulta := TIBSQL.Create( nil );
  Consulta.Transaction := Transaccion;
  Consulta.SQL.Add( 'SELECT * FROM ' + sTabla );
  Transaccion.StartTransaction;
  try
   Consulta.ExecQuery;
  except
   Transaccion.Rollback;
   raise;
  end;

  // Creamos en el listview una columna por cada campo
  Listado.Columns.Clear;
  Listado.Columns.Add;
  Listado.Columns[0].Width := 0;
  for i := 0 to Campos.Count - 1 do
  begin
   Listado.Columns.Add;
   Listado.Columns[i+1].Caption := Campos[i];
   Listado.Columns[i+1].Width := 100;
  end;

  // Listamos los registros
  Listado.Clear;
  while not Consulta.Eof do
  begin
   Listado.Items.Add;

   for i := 0 to Campos.Count - 1 do
    Listado.Items[Listado.Items.Count-1].SubItems.Add(
Consulta.FieldByName(
Campos[i] ).AsString );

   Consulta.Next;
```

```
  end;

  // Una vez hemos terminado liberamos los objetos creados
  FreeAndNil( Campos );
  FreeAndNil( Consulta );
  FreeAndNil( Transaccion );
end;
```

Por supuesto, todo este proceso se puede mejorar refactorizando código y dividiendo las partes más importantes en clases más pequeñas. Haciendo muchas pruebas con objetos *TIBSQL* y TIBQuery me he dado cuenta que para operaciones donde se requiere velocidad los objetos TIBSQL con mucho más rápidos que los TIBQuery, aunque estos últimos son mucho más completos.

Pruebas realizadas en Delphi 7

Efectos de animación en las ventanas

En este artículo explicaré de forma detallada cómo crear animaciones para las ventanas de delphi con los mismos efectos que disponen los sistemas operativos Windows, y aclararé cuándo aplicarlos y los problemas que tienen.

La función encargada de animar ventanas es la siguiente (api de windows):

AnimateWindow

Y los parámetros que la definen son los siguientes:

hWnd - Manejador o *Handle* de la ventana, a la cuál se aplica el efecto.
dwTime - Velocidad para reproducir el efecto. A más tiempo, más suave y con más lentitud es el efecto.
dwFlags - Parámetros que definen el tipo de efecto, la orientación y la activación de la ventana.
Se pueden combinar varios parámetros para conseguir efectos personalizados.

Dentro del parámetro dwFlags, se pueden realizar los efectos de animación que detallo:

Tipos de efectos

AW_SLIDE

Esta es una animación de deslizamiento. Este parámetro es ignorado si se utiliza la bandera AW_CENTER. De forma predeterminada, y si no se indica este parámetro, todas las ventanas utilizan el efecto de persiana, o enrollamiento.

AW_BLEND

Aplica un efecto de aparición gradual. Recuerde utilizar este parámetro si la ventana tiene prioridad sobre las demás. Este efecto sólo funciona con

Windows 2000 y Windows XP.

AW_HIDE

Oculta la ventana, sin animación. Hay que combinar con otro parámetro para que la ocultación muestre animación. Por ejemplo con *AW_SLIDE* o *AW_BLEND*.

AW_CENTER

Este efecto provoca que la ventana aparezca desde el centro de la pantalla o escritorio. Para que funcione, debe ser combinado con el parámetro *AW_HIDE* para mostrar la ventana, o no utilizar *AW_HIDE* para ocultarla.

Orientación al mostrar u ocultar

AW_HOR_POSITIVE

Animar la ventana de izquierda a derecha. Este parámetro puede ser combinado con las animaciones de deslizamiento o persiana. Si utiliza *AW_CENTER* o *AW_BLEND*, no tendrá efecto.

AW_HOR_NEGATIVE

Animar la ventana de derecha a izquierda. Este parámetro puede ser combinado con las animaciones de deslizamiento o persiana. Si utiliza *AW_CENTER* o *AW_BLEND*, no tendrá efecto.

AW_VER_POSITIVE

Animar la ventana de arriba hacia abajo. Este parámetro puede ser combinado con las animaciones de deslizamiento o persiana. Si utiliza *AW_CENTER* o *AW_BLEND*, no tendrá efecto.

AW_VER_NEGATIVE

Animar la ventana de abajo hacia arriba. Este parámetro puede ser combinado con las animaciones de deslizamiento o persiana. Si utiliza *AW_CENTER* o *AW_BLEND*, no tendrá efecto.

Otros parámetros

AW_ACTIVATE

Este parámetro traspasa el foco de activación a la ventana antes de aplicar el efecto. Recomiendo utilizarlo, sobre todo cuando las ventanas contiene algún tema de Windows XP. No utilizar con la bandera *AW_HIDE*.

Utilizando la función en Delphi

¿En qué evento utilizar esta función?

Normalmente, y a nivel personal y por experiencias negativas, siempre la utilizo en el evento FormShow de la ventana a la cuál aplicar el efecto. Un ejemplo sería el siguiente:

```
procedure TFForm1.FormShow(Sender: TObject);
begin
  AnimateWindow( Handle, 400, AW_ACTIVATE or AW_SLIDE or
AW_VER_POSITIVE );
end;
```

(Este efecto va mostrando la ventana de arriba hacia abajo con deslizamiento).

Problemas con los temas de Windows XP y las ventanas de Delphi

Naturalmente, no todo es una maravilla, y entre los problemas que pueden surgir al crear estos efectos, están los siguientes:

- Temas visuales de Windows XP:

Cuando un efecto de animación es mostrado, a veces ciertos controles de la ventana, cómo los TEdit, ComboBox, etc, no terminan de actualizarse, quedando con el aspecto antiguo de Windows 98. Para solucionar este problema, hay que escribir la función "RedrawWindow" a continuación de AnimateWindow:

```
procedure TFForm1.FormShow(Sender: TObject);
begin
  AnimateWindow( Handle, 400, AW_ACTIVATE or AW_SLIDE or
AW_VER_POSITIVE );
  RedrawWindow( Handle, nil, 0, RDW_ERASE or RDW_FRAME or
RDW_INVALIDATE or RDW_ALLCHILDREN );
end;
```

- Ocultando ventanas entre ventanas de Delphi:

Por un problema desconocido de delphi (por lo menos desconozco si Delphi

2006 lo hace), ocultar una ventana con animación, teniendo otras ventanas de delphi (de tu aplicación) detrás, produce un efecto de redibujado fatal, que desmerece totalmente el efecto realizado. Esto no pasa si las ventanas que aparecen detrás no son de Delphi, o de tu propia aplicación. Por ese motivo, personalmente nunca utilizo este efecto de ocultación de ventanas.

Últimos consejos

Por último, permitidme daros un consejo. Estos efectos también son válidos en Windows 2000, pero los efectos pueden no ser tan fluidos como en Windows XP. Por ello, no estaría mal que estos efectos sean una opción de configuración de vuestra aplicación, es decir, permitir al usuario activarlos o desactivarlos.

También recomiendo no abusar de estos efectos, al final terminan siendo un poco molestos. Realizarlos en las ventanas principales es mejor que en todas las ventanas.

Espero que el artículo sea de utilidad, y dé un toque de elegancia a vuestras aplicaciones.

Pruebas realizadas en Delphi 7.

Guardando y cargando opciones

Hay ocasiones en que nos interesa que las opciones de nuestro programa permanezcan en el mismo después de terminar su ejecución. Principalmente se suelen utilizar cuatro maneras de guardar las opciones:

1º En un archivo de texto plano.

2º En un archivo binario.

3º En un archivo INI.

4º En el registro del sistema de Windows.

Vamos a suponer que tenemos un formulario de opciones con campos de tipo string, integer, boolean, date, time y real.

Los archivos de opciones se crearán en el mismo directorio en el que se ejecuta nuestra aplicación.

GUARDANDO OPCIONES EN TEXTO PLANO

Para ello utilizamos un archivo de tipo TextFile para guardar la información:

```
procedure TFPrincipal.GuardarTexto;
var F: TextFile;
begin
  // Asignamos el archivo de opciones al puntero F
  AssignFile( F, ExtractFilePath( Application.ExeName ) +
'opciones.txt' );

  // Abrimos el archivo en modo creación/escritura
  Rewrite( F );

  // Guardamos las opciones
  WriteLn( F, IMPRESORA.Text );
  WriteLn( F, IntToStr( COPIAS.Value ) );

  if VISTAPREVIA.Checked then
    WriteLn( F, 'CON VISTA PREVIA' )
  else
    WriteLn( F, 'SIN VISTA PREVIA' );

  WriteLn( F, DateToStr( FECHA.Date ) );
  WriteLn( F, HORA.Text );
  WriteLn( F, FormatFloat( '###0.00', MARGEN.Value ) );

  CloseFile( F );
end;
```

CARGANDO OPCIONES DESDE TEXTO PLANO

Antes de abrir el archivo comprobamos si existe:

```
procedure TFPrincipal.CargarTexto;
var F: TextFile;
    sLinea: String;
begin
  // Si no existe el archivo de opciones no hacemos nada
  if not FileExists( ExtractFilePath( Application.ExeName ) +
'opciones.txt' ) then
    Exit;

  // Asignamos el archivo de opciones al puntero F
  AssignFile( F, ExtractFilePath( Application.ExeName ) +
'opciones.txt' );

  // Abrimos el archivo en modo lectura
  Reset( F );

  // Cargamos las opciones
  ReadLn( F, sLinea );
  IMPRESORA.Text := sLinea;

  ReadLn( F, sLinea );
  COPIAS.Value := StrToInt( sLinea );

  ReadLn( F, sLinea );
  VISTAPREVIA.Checked := sLinea = 'CON VISTA PREVIA';
```

```
  ReadLn( F, sLinea );
  FECHA.Date := StrToDate( sLinea );

  ReadLn( F, sLinea );
  HORA.Text := sLinea;

  ReadLn( F, sLinea );
  MARGEN.Value := StrToFloat( sLinea );

  CloseFile( F );
end;
```

GUARDANDO OPCIONES EN UN ARCHIVO BINARIO

Lo que hacemos en esta ocación es crear un registro (record) que contenga las opciones de nuestro programa. Después volcamos todo el contenido del registro en un archivo binario del mismo tipo.

En la interfaz de nuestra unidad definimos:

```
type
  TOpciones = record
    sImpresora: String[100];
    iCopias: Integer;
    bVistaPrevia: Boolean;
    dFecha: TDate;
    tHora: TTime;
    rMargen: Real;
  end;
```

Y creamos el procemimiento que lo graba:

```
procedure TFPrincipal.GuardarBinario;
var
  // Creamos un registro y un fichero para el mismo
  Opciones: TOpciones;
  F: file of TOpciones;
begin
  // Metemos las opciones del formulario en el registro
  Opciones.sImpresora := IMPRESORA.Text;
  Opciones.iCopias := COPIAS.Value;
  Opciones.bVistaPrevia := VISTAPREVIA.Checked;
  Opciones.dFecha := FECHA.Date;
  Opciones.tHora := StrToTime( HORA.Text );
  Opciones.rMargen := MARGEN.Value;

  // Asignamos el archivo de opciones al puntero F
  AssignFile( F, ExtractFilePath( Application.ExeName ) +
'opciones.dat' );

  // Abrimos el archivo en modo creación/escritura
  Rewrite( F );

  // Guardamos de golpe todas las opciones
  Write( F, Opciones );

  // Cerramos el fichero
```

```
  CloseFile( F );
end;
```

CARGANDO OPCIONES DESDE UN ARCHIVO BINARIO

Utilizamos el registro creado anteriormente:

```
procedure TFPrincipal.CargarBinario;
var
  // Creamos un registro y un fichero para el mismo
  Opciones: TOpciones;
  F: file of TOpciones;
begin
  // Asignamos el archivo de opciones al puntero F
  AssignFile( F, ExtractFilePath( Application.ExeName ) +
'opciones.dat' );

  // Abrimos el archivo en modo creación/escritura
  Reset( F );

  // Guardamos de golpe todas las opciones
  Read( F, Opciones );

  // Cerramos el fichero
  CloseFile( F );

  // Copiamos las opciones del registro en el formulario de opciones
  IMPRESORA.Text := Opciones.sImpresora;
  COPIAS.Value := Opciones.iCopias;
  VISTAPREVIA.Checked := Opciones.bVistaPrevia;
  FECHA.Date := Opciones.dFecha;
  HORA.Text := TimeToStr( Opciones.tHora );
  MARGEN.Value := Opciones.rMargen;
end;
```

GUARDANDO OPCIONES EN UN ARCHIVO INI

Los dos casos anteriores tienen un defecto: si ampliamos el número de opciones e intentamos cargar las opciones con el formato antiguo se puede provocar un error de E/S debido a que los formatos de texto o binario han cambiado.

Lo más flexible en este claso es utilizar un archivo INI, el cual permite agrupar opciones y asignar un nombre a cada una:

```
procedure TFPrincipal.GuardarINI;
var INI: TIniFile;
begin
  // Creamos el archivo INI
  INI := TINIFile.Create( ExtractFilePath( Application.ExeName ) +
'opciones.ini' );

  // Guardamos las opciones
  INI.WriteString( 'OPCIONES', 'IMPRESORA', IMPRESORA.Text );
  INI.WriteInteger( 'OPCIONES', 'COPIAS', COPIAS.Value );
  INI.WriteBool( 'OPCIONES', 'VISTAPREVIA', VISTAPREVIA.Checked );
  INI.WriteDate( 'OPCIONES', 'FECHA', FECHA.Date );
  INI.WriteTime( 'OPCIONES', 'HORA', StrToTime( HORA.Text ) );
```

```
  INI.WriteFloat( 'OPCIONES', 'MARGEN', MARGEN.Value );

  // Al liberar el archivo INI se cierra el archivo opciones.ini
  INI.Free;
end;
```

CARGANDO OPCIONES DE UN ARCHIVO INI

Aunque aquí comprobamos si existe el archivo, no es necesario, ya que cargaría las opciones por defecto:

```
procedure TFPrincipal.CargarINI;
var INI: TIniFile;
begin
  // Si no existe el archivo no hacemos nada
  if not FileExists( ExtractFilePath( Application.ExeName ) +
'opciones.ini' ) then
    Exit;

  // Creamos el archivo INI
  INI := TINIFile.Create( ExtractFilePath( Application.ExeName ) +
'opciones.ini' );

  // Guardamos las opciones
  IMPRESORA.Text := INI.ReadString( 'OPCIONES', 'IMPRESORA', '' );
  COPIAS.Value := INI.ReadInteger( 'OPCIONES', 'COPIAS', 0 );
  VISTAPREVIA.Checked := INI.ReadBool( 'OPCIONES', 'VISTAPREVIA',
False );
  FECHA.Date := INI.ReadDate( 'OPCIONES', 'FECHA', Date );
  HORA.Text := TimeToStr( INI.ReadTime( 'OPCIONES', 'HORA', Time ) );
  MARGEN.Value := INI.ReadFloat( 'OPCIONES', 'MARGEN', 0.00 );

  // Al liberar el archivo INI se cierra el archivo opciones.ini
  INI.Free;
end;
```

GUARDANDO OPCIONES EN EL REGISTRO DEL SISTEMA

Si queremos que nadie nos toque las opciones del programa podemos guardarlo todo en el registro de Windows:

```
procedure TFPrincipal.GuardarRegistroSistema;
var Reg: TRegistry;
begin
  // Creamos un objeto para manejar el registro
  Reg := TRegistry.Create;

  // Guardamos las opciones
  try
    Reg.RootKey := HKEY_LOCAL_MACHINE;
    if Reg.OpenKey( '\Software\MiPrograma', True ) then
    begin
      Reg.WriteString( 'IMPRESORA', IMPRESORA.Text );
      Reg.WriteInteger( 'COPIAS', COPIAS.Value );
      Reg.WriteBool( 'VISTAPREVIA', VISTAPREVIA.Checked );
      Reg.WriteDate( 'FECHA', FECHA.Date );
      Reg.WriteTime( 'HORA', StrToTime( HORA.Text ) );
      Reg.WriteFloat( 'MARGEN', MARGEN.Value );
      Reg.CloseKey;
```

```
    end;
  finally
    Reg.Free;
  end;
end;
```

Para probar si se ha guardado la información pulsa el botón INICIO y opción EJECUTAR: REGEDIT. Las opciones se habrán guardado en la carpeta:

\HKEY_LOCAL_MACHINE\SOFTWARE\MiPrograma

CARGANDO OPCIONES DESDE EL REGISTRO DEL SISTEMA

Antes de cargar las opciones comprueba si existe la clave:

```
procedure TFPrincipal.CargarRegistroSistema;
var Reg: TRegistry;
begin
  // Creamos un objeto para manejar el registro
  Reg := TRegistry.Create;

  // Guardamos las opciones
  try
    Reg.RootKey := HKEY_LOCAL_MACHINE;
    if Reg.OpenKey( '\Software\MiPrograma', True ) then
    begin
      IMPRESORA.Text := Reg.ReadString( 'IMPRESORA' );
      COPIAS.Value := Reg.ReadInteger( 'COPIAS' );
      VISTAPREVIA.Checked := Reg.ReadBool( 'VISTAPREVIA' );
      FECHA.Date := Reg.ReadDate( 'FECHA' );
      HORA.Text := TimeToStr( Reg.ReadTime( 'HORA' ) );
      MARGEN.Value := Reg.ReadFloat( 'MARGEN' );
      Reg.CloseKey;
    end;
  finally
    Reg.Free;
  end;
end;
```

Pruebas realizadas en Delphi 7

Minimizar en la bandeja del sistema

Una de las características mas utilizadas en los programas P2P es la de minimizar nuestra aplicación en la bandeja del sistema (al lado del reloj de Windows en la barra de tareas).

Voy a mostraros como modificar el formulario principal de vuestra aplicación para que se minimize en la bandeja del sistema y una vez minimizado cuando se pulse sobre el icono se restaure. También vamos a añadir la posibilidad de pulsar dicho icono con el botón derecho del ratón y que muestre un menu contextual (popup) con la opción Mostrar.

Lo primero de todo es añadir un menu contextual a nuestro formulario principal (PopupMenu) con el nombre MenuBandeja. Añadimos una sola

opción llamada Mostrar. A continuación añadimos en la sección uses del formulario principal la unidad ShellAPI:

```
uses
  Windows, Messages, ...., ShellAPI;
```

Después en la sección private insertamos la variable:

```
  IconData: TNotifyIconData;
```

En la misma sección private añadimos los procedimientos:

```
  procedure WMSysCommand( var Msg: TWMSysCommand ); message
WM_SYSCOMMAND;
  procedure Restaurar( var Msg: TMessage ); message WM_USER+1;
```

Cuya implementación sería la siguiente:

```
procedure TFPrincipal.WMSysCommand( var Msg: TWMSysCommand );
begin
  if Msg.CmdType = SC_MINIMIZE then
    Minimizar
  else
    DefaultHandler( Msg );
end;

procedure TFPrincipal.Restaurar( var Msg: TMessage );
var p: TPoint;
begin
  // ¿Ha pulsado el botón izquierdo del ratón?
  if Msg.lParam = WM_LBUTTONDOWN then
    MostrarClick( Self );

  // ¿Ha pulsado en la bandeja del sistema con el botón derecho del
ratón?
  if Msg.lParam = WM_RBUTTONDOWN then
  begin
    SetForegroundWindow( Handle );
    GetCursorPos( p );
    MenuBandeja.Popup( p.x, p.y );
    PostMessage( Handle, WM_NULL, 0, 0 );
  end;
end;
```

El procedimiento WMSysCommand es el encargado de interceptar los mensajes del sistema que manda Windows a nuestra aplicación. En el caso de que el mensaje enviado sea SC_MINIMIZE minimizamos la ventana en la bandeja del sistema. Si es otro mensaje dejamos que Windows lo maneje (DefaultHandler).

El procedimiento Restaurar comprueba si ha pulsado el botón izquierdo del ratón sobre el icono de la bandeja del sistema para volver a mostrar nuestra ventana. Si pulsa el botón derecho llamará a nuestro menu contextual MenuBandeja.

Ahora creamos el procedimiento encargado de minimizar la ventana:

```
procedure TFPrincipal.Minimizar;
begin
  with IconData do
  begin
    cbSize := sizeof( IconData );
    Wnd := Handle;
    uID := 100;
    uFlags := NIF_MESSAGE + NIF_ICON + NIF_TIP;
    uCallbackMessage := WM_USER + 1;

    // Usamos de icono el mismo de la aplicación
    hIcon := Application.Icon.Handle;

    // Como Hint del icono, el nombre de la aplicación
    StrPCopy( szTip, Application.Title );
  end;

  // Ponemos el icono al lado del reloj
  Shell_NotifyIcon( NIM_ADD, @IconData );

  // Ocultamos el formulario
  Hide;
end;
```

Y por último el evento al pulsar la opción Mostrar en el menú contextual:

```
procedure TFPrincipal.MostrarClick( Sender: TObject );
begin
  // Volvemos a mostrar de nuevo el formulario
  FPrincipal.Show;
  ShowWindow( Application.Handle, SW_SHOW );

  // Eliminamos el icono de la bandeja del sistema
  Shell_NotifyIcon( NIM_DELETE, @IconData );
  IconData.Wnd := 0;
end;
```

Aunque pueda parecer algo engorroso creo que es mas limpio que tener que instalar componentes para que realicen esto. Al fin y al cabo sólo hay que hacerlo sólo en el formulario principal.

Pruebas realizadas en Delphi 7.

Cómo ocultar una aplicación

Vamos a ver como hacer que una aplicación cualquiera hecha en Delphi quede oculta de la barra de tareas de Windows y del escritorio. Sólo podrá verse ejecutando el administrador de tareas en la pestaña Procesos.

Para ello vamos a añadir en el evento OnCreate del formulario principal de nuestra aplicación lo siguiente:

```
procedure TFPrincipal.FormCreate(Sender: TObject);
```

```
begin
  // Hacemos que el formulario sea invisible poniendolo en la
  // esquina superior izquierda, tamaño cero y aplicación invisible
  BorderStyle := bsNone;
  Left := 0;
  Top := 0;
  Width := 0;
  Height := 0;
  Visible := False;
  Application.Title := '';
  Application.ShowMainForm := False;

  // Lo ocultamos de la barra de tareas
  ShowWindow( Application.Handle, SW_HIDE );
  SetWindowLong( Application.Handle, GWL_EXSTYLE,
                 GetWindowLong(Application.Handle, GWL_EXSTYLE) or
                 WS_EX_TOOLWINDOW and not WS_EX_APPWINDOW);
end;
```

Esto nos puede ser util para crear programas residentes ocultos al usuario para administración de copias de seguridad, reparación automática de bases de datos y envío de mailing automátizado.

Capturar el teclado en Windows

Hay ocasiones en las cuales nos interesa saber si una tecla de Windows ha sido pulsada aunque estemos en otra aplicación que no sea la nuestra.

Por ejemplo en el artículo anterior mostré como capturar la pantalla. Sería interesante que si pulsamos F8 estando en cualquier aplicación nos capture la pantalla (incluso si nuestra aplicación esta minimizada).

Para ello vamos a utilizar la función de la API de Windows GetAsyncKeyState la cual acepta como parámetro la tecla pulsada (VK_RETURN, VK_ESCAPE, VK_F8, etc) y nos devuelve -32767 si la tecla ha sido pulsada.

Como el teclado hay que leerlo constantemente y no conviene dejar un bucle cerrado consumiendo mucho procesador, lo que vamos a hacer es meter a nuestro formulario un temporizador TTimer activado cada 10 milisegundos (Inverval) y con el evento OnTimer definido de la siguiente manera:

```
procedure TFormulario.TemporizadorTimer( Sender: TObject );
begin
  // ¿Ha pulsado una tecla?
  if GetAsyncKeyState( VK_F8 ) = -32767 then
    CapturarPantalla;
end;
```

Para capturar números o letras se hace con la función ord:

```
if GetAsyncKeyState( Ord( 'A' ) ) then ...
if GetAsyncKeyState( Ord( '5' ) ) then ...
```

Si es una letra hay que pasarla a mayúsculas.

Sólo con esto podemos interceptar cualquier tecla del buffer de Windows. Por ejemplo se podría hacer una aplicación que al pulsar F10 minimize todas las ventanas de Windows.

Pruebas realizadas en Delphi 7.

Capturar la pantalla de Windows

Vamos a crear un procedimiento que captura un trozo de la pantalla de Windows y la guarda en un bitmap:

```
procedure CapturarPantalla( x, y, iAncho, iAlto: Integer; Imagen:
TBitmap );
var
  DC: HDC;
  lpPal : PLOGPALETTE;
begin
  if ( iAncho = 0 ) OR ( iAlto = 0 ) then
    Exit;

  Imagen.Width := iAncho;
  Imagen.Height := iAlto;
  DC := GetDc( 0 );

  if ( DC = 0 ) then
    Exit;

  if ( GetDeviceCaps( dc, RASTERCAPS) and  RC_PALETTE = RC_PALETTE )
then
  begin
    GetMem( lpPal, SizeOf( TLOGPALETTE ) + ( 255 * SizeOf(
TPALETTEENTRY ) ) );
    FillChar( lpPal^, SizeOf( TLOGPALETTE ) + ( 255 * SizeOf(
TPALETTEENTRY ) ), #0 );
    lpPal^.palVersion := $300;
    lpPal^.palNumEntries := GetSystemPaletteEntries( DC, 0, 256,
lpPal^.palPalEntry );

    if (lpPal^.PalNumEntries <> 0) then
      Imagen.Palette := CreatePalette( lpPal^ );

    FreeMem( lpPal, SizeOf( TLOGPALETTE ) + ( 255 * SizeOf(
TPALETTEENTRY ) ) );
  end;

  BitBlt( Imagen.Canvas.Handle, 0, 0, iAncho, iAlto, DC, x, y, SRCCOPY
);
  ReleaseDc( 0, DC );
end;
```

Resumiendo a grandes rasgos lo que hace el procedimiento es crear un dispositivo de contexto donde según el número de bits por pixel reserva una

zona de memoria para capturar el escritorio. Después mediante la función BitBlt vuelca la imagen capturada al Canvas de la imagen que le pasamos.

Para capturar toda la pantalla de Windows utilizando este procedimiento hacemos lo siguiente:

```
var Imagen: TBitmap;
begin
  Imagen := TBitmap.Create;
  CapturarPantalla( 0, 0, Screen.Width, Screen.Height, Imagen );
  Imagen.SaveToFile( ExtractFilePath( Application.ExeName ) +
'captura.bmp' );
  Imagen.Free;
end;
```

La pantalla capturada la guarda en el archivo captura.bmp al lado de nuestro ejecutable. Sólo faltaría el poder capturar una tecla de Windows desde cualquier aplicación para activar nuestro capturador de pantalla (para que no se capture a si mismo).

Pruebas realizadas en Delphi 7.

Obtener los favoritos de Internet Explorer

El siguiente procedimiento recursivo obtiene la lista de los enlaces favoritos de Internet Explorer. Puede ser de mucha utilidad en el caso de formatear el equipo y salvar la lista de favoritos a un archivo de texto o a una base de datos.

Lo primero es añadir en uses:

```
uses
  Windows, Dialogs, ..., ShlObj;
```

Y este sería el procedimiento:

```
function ObtenerFavoritosIE( sRutaFavoritos: String ): TStrings;
var
  Busqueda: TSearchrec;
  ListaFavoritos: TStrings;
  sRuta, sDirectorio, sArchivo: String;
  Buffer: array[0..2047] of Char;
  iEncontrado: Integer;
begin
  ListaFavoritos := TStringList.Create;
```

```
  try
    sRuta := sRutaFavoritos + '\*.url';
    sDirectorio := ExtractFilepath( sRuta );
    iEncontrado := FindFirst( sRuta, faAnyFile, Busqueda );

    while iEncontrado = 0 do
    begin
      SetString( sArchivo, Buffer,
                 GetPrivateProfileString( 'InternetShortcut',
                 PChar( 'URL' ), nil, Buffer, SizeOf( Buffer ),
                 PChar( sDirectorio + Busqueda.Name ) ) );
      ListaFavoritos.Add( sArchivo );
      iEncontrado := FindNext( Busqueda );
    end;

    iEncontrado := FindFirst( sDirectorio + '\*.*', faAnyFile,
Busqueda );

    while iEncontrado = 0 do
    begin
      if ( ( Busqueda.Attr and faDirectory ) > 0 ) and (
Busqueda.Name[1] <> '.' ) then
        ListaFavoritos.AddStrings( ObtenerFavoritosIE( sDirectorio +
'\' + Busqueda.name ) );

      iEncontrado := FindNext( Busqueda );
    end;

    FindClose( Busqueda );
  finally
    Result := ListaFavoritos;
  end;
end;
```

Para utilizar el procedimiento supongamos que en el formulario tenemos un componente ListBox (FAVORITOS) y un botón (BFavoritos) que al pulsarlo nos trae todos los favoritos a dicha lista:

```
procedure TFPrincipal.BFavoritosClick( Sender: TObject );
var
  pidl: PItemIDList;
  sRutaFavoritos: array[0..MAX_PATH] of Char;
begin
  SHGetSpecialFolderLocation( Handle, CSIDL_FAVORITES, pidl );
  SHGetPathFromIDList( pidl, sRutaFavoritos );
  FAVORITOS.Items := ObtenerFavoritosIE( StrPas( sRutaFavoritos ) );
end;
```

Pruebas realizadas en Delphi 7.

Crear un acceso directo

Aquí tenemos un pequeño pero interesante procedimiento para crear accesos directos en Windows. Antes de implementarlo hay que añadir en uses una serie de unidades externas:

```
uses
  Windows, Dialogs, ...,  ShlObj, ActiveX, StdCtrls, Registry, ComObj;
```

Este sería el procedimiento:

```
procedure CrearAccesoDirecto( sExe, sArgumentos, sDirTrabajo,
sNombreLnk, sDirDestino: string );
var
  Objeto: IUnknown;
  UnSlink: IShellLink;
  FicheroP: IPersistFile;
  WFichero: WideString;
begin
  Objeto := CreateComObject( CLSID_ShellLink );
  UnSlink := Objeto as IShellLink;
  FicheroP := Objeto as IPersistFile;

  with UnSlink do
  begin
    SetArguments( PChar( sArgumentos ) );
    SetPath( PChar( sExe ) );
    SetWorkingDirectory( PChar( sDirTrabajo ) );
  end;

  WFichero := sDirDestino + '\' + sNombreLnk;
  FicheroP.Save( PWChar( WFichero ), False );
end;
```

Y estos son sus parámetros:

```
sExe        -> Ruta que apunta al ejecutable o archivo a crear el
acceso directo
sArgumentos -> Parámetros que le mandamos al EXE
sDirTrabajo -> Ruta al directorio de trabajo del ejecutable
sNombreLnk  -> Nombre del acceso directo
sDirDestino -> Ruta destino donde se creará el acceso directo
```

Aquí os muestro un ejemplo de cómo crear un acceso directo de la calculadora de Windows al escritorio:

```
procedure CrearAccesoCalculadora;
var
  sEscritorio: String;
  Registro: TRegistry;
begin
  Registro := TRegistry.Create;

  // Leemos la ruta del escritorio
  try
    Registro.RootKey := HKEY_CURRENT_USER;

    if Registro.OpenKey(
'\Software\Microsoft\Windows\CurrentVersion\explorer\Shell Folders',
True ) then
      sEscritorio := Registro.ReadString( 'Desktop' );
  finally
    Registro.CloseKey;
    Registro.Free;
    inherited;
```

```
  end;

  CrearAccesoDirecto( 'C:\Windows\System32\calc.exe', '',
                      'C:\Windows\System32\', 'Calculadora.lnk',
sEscritorio );
end;
```

Pruebas realizadas en Delphi 7.

Averiguar la versión de Windows

La siguiente función nos devuelve la versión de Windows donde se está ejecutando nuestro programa:

```
function ObtenerVersion: String;
var
   osVerInfo: TOSVersionInfo;
   VersionMayor, VersionMenor: Integer;
begin
   Result := 'Desconocida';
   osVerInfo.dwOSVersionInfoSize := SizeOf( TOSVersionInfo );

   if GetVersionEx( osVerInfo ) then
   begin
     VersionMenor := osVerInfo.dwMinorVersion;
     VersionMayor := osVerInfo.dwMajorVersion;

     case osVerInfo.dwPlatformId of
       VER_PLATFORM_WIN32_NT:
       begin
         if VersionMayor <= 4 then
           Result := 'Windows NT'
         else
           if ( VersionMayor = 5 ) and ( VersionMenor = 0 ) then
             Result := 'Windows 2000'
           else
             if ( VersionMayor = 5 ) and ( VersionMenor = 1 ) then
               Result := 'Windows XP'
             else
               if ( VersionMayor = 6 ) then
                 Result := 'Windows Vista';
       end;

       VER_PLATFORM_WIN32_WINDOWS:
       begin
         if ( VersionMayor = 4 ) and ( VersionMenor = 0 ) then
           Result := 'Windows 95'
         else
           if ( VersionMayor = 4 ) and ( VersionMenor = 10 ) then
           begin
             if osVerInfo.szCSDVersion[1] = 'A' then
               Result := 'Windows 98 Second Edition'
             else
               Result := 'Windows 98';
           end
           else
             if ( VersionMayor = 4 ) and ( VersionMenor = 90 ) then
               Result := 'Windows Millenium'
```

```
            else
              Result := 'Desconocida';
          end;
      end;
   end;
end;
```

Primero averigua de que plataforma se trata. Si es Win32 los sistemas operativos pueden ser: Windows 95, Windows 98, Windows 98 Second Edition y Windows Millenium. Por otro lado, si se trata de la plataforma NT entonces las versiones son: Windows NT, Windows 2000, Windows XP y Windows Vista.

Esta función puede ser de utilidad para saber que librerias DLL tiene instaladas, que comandos del sistema podemos ejecutar o para saber si tenemos que habilitar o deshabilitar ciertas funciones de nuestra aplicación.

Pruebas realizadas en Delphi 7.

Recorrer un árbol de directorios

El procedimiento que voy a mostrar a continuación recorre el contenido de directorios, subdirectorios y archivos volcando la información en un campo memo (TMemo).

Modificando su comportamiento puede ser utilizado para realizar copias de seguridad, calcular el tamaño de un directorio o borrar el contenido de los mismos.

El procedimiento RecorrerDirectorio toma como primer parámetro la ruta que desea recorrer y como segundo parámetro si deseamos que busque también en subdirectorios:

```
procedure TFBuscar.RecorrerDirectorio( sRuta: String;
bIncluirSubdirectorios: Boolean );
var
  Directorio: TSearchRec;
  iResultado: Integer;
begin
  // Si la ruta no termina en contrabarra se la ponemos
  if sRuta[Length(sRuta)] <> '\' then
    sRuta := sRuta + '\';

  // ¿No existe el directorio que vamos a recorrer?
  if not DirectoryExists( sRuta ) then
  begin
    Application.MessageBox( PChar( 'No existe el directorio:' + #13 +
#13 + sRuta ), 'Error', MB_ICONSTOP );
    Exit;
  end;

  iResultado := FindFirst( sRuta + '*.*', FaAnyfile, Directorio );
  while iResultado = 0 do
  begin
```

```
    // ¿Es un directorio y hay que entrar en él?
    if ( Directorio.Attr and faDirectory = faDirectory ) and
bIncluirSubdirectorios then
    begin
      if ( Directorio.Name <> '.' ) and ( Directorio.Name <> '..' )
then
        RecorrerDirectorio( sRuta + Directorio.Name, True );
    end
    else
      // ¿No es el nombre de una unidad ni un directorio?
      if ( Directorio.Attr and faVolumeId <> faVolumeID ) then
        Archivos.Lines.Add( sRuta + Directorio.Name );

    iResultado := FindNext( Directorio );
  end;

  SysUtils.FindClose( Directorio );
end;
```

Antes de comenzar a buscar directorios se asegura de que la ruta que le pasemos termine en contrabarra y en el caso de que no sea así se la pone al final.

Para recorrer un directorio utiliza la estructura de datos TSearchRec la cual se utiliza para depositar en ella la información del contenido de un directorio mediante las funciones FindFirst y FindNext.

TSearchRec no contiene la información de todo el directorio sino que es un puntero al directorio o archivo actual. Sólo mirando los atributos mediante la propiedad Attr podemos saber si lo que estamos leyendo es un directorio, archivo o unidad.

También se cuida de saltarse los directorios '.' y '..' ya que sino el procedimiento recursivo RecorrerDirectorio se volvería loco hasta reventar la pila.

Realizar modificaciones para cambiar su comportamiento puede ser peligroso si no lleváis cuidado ya que la recursividad puede de dejar sin memoria la aplicación. Al realizar tareas como borrar subdirectorios mucho cuidado no darle la ruta C:\. Mejor hacer ensayos volcando el contenido en un Memo hasta tener el resultado deseado.

Pruebas realizadas en Delphi 7.

Ejecutar un programa al arrancar Windows

Para ejecutar automáticamente nuestra aplicación al arrancar Windows vamos a utilizar la siguiente clave del registro del sistema:

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run\

Todos los programas que se introduzcan dentro de esa clave (antivirus, monitores del sistema, etc.) arrancarán al iniciar Windows.

Para ello vamos a utilizar el objeto TRegistry. Para ello hay que añadir su unidad correspondiente en uses:

```
uses Windows, Messages, ..., Registry;
```

Ahora vamos con el procedimiento encargado de poner nuestro programa al inicio de Windows:

```
procedure TFPrincipal.PonerProgramaInicio;
var Registro: TRegistry;
begin
  Registro := TRegistry.Create;
  Registro.RootKey := HKEY_LOCAL_MACHINE;

  if Registro.OpenKey(
'Software\Microsoft\Windows\CurrentVersion\Run', FALSE ) then
  begin
    Registro.WriteString( ExtractFileName( Application.ExeName ),
Application.ExeName );
    Registro.CloseKey;
  end;

  Registro.Free;
end;
```

El método WriteString toma como primer parámetro la el nombre del valor en el registro y como segúndo parámetro la ruta donde se encuentra el programa a ejecutar. En nuestro caso como nombre del valor le he dado el nombre de nuestro ejecutable y como segundo la ruta desde donde estamos ejecutando el programa en este mismo instante.

Si en un futuro deseamos quitar el programa entonces sólo hay que eliminar la clave:

```
procedure TFPrincipal.QuitarProgramaInicio;
var Registro: TRegistry;
begin
  Registro := TRegistry.Create;
  Registro.RootKey := HKEY_LOCAL_MACHINE;

  if Registro.OpenKey(
'Software\Microsoft\Windows\CurrentVersion\Run', FALSE ) then
  begin
    // ¿Existe el valor que vamos a borrar?
    if Registro.ValueExists( ExtractFileName( Application.ExeName ) )
then
      Registro.DeleteValue( ExtractFileName( Application.ExeName ) );

    Registro.CloseKey;
  end;

  Registro.Free;
end;
```

Hay ciertos antivirus como el NOD32 que saltan nada más compilar nuestro

programa por el simple hecho de tocar la clave Run. Habrá que decirle a nuestro antivirus que nuestro programa no es maligno.

Pruebas realizadas en Delphi 7.

Listar los programas instalados en Windows

Windows almacena la lista de programas instalados (Agregar/Quitar programas) en la clave de registro:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall
\
```

En esa clave hay tantas subclaves como programas instalados. Pero lo que nos interesa a nosotros no es el nombre de la clave del programa instalado sino el nombre del programa que muestra Windows en Agregar/Quitar programas. Para ello entramos en cada clave y leemos el valor DisplayName.

Lo primero añadimos la unidad:

```
uses
  Windows, Messages, ..., Registry;
```

Y aquí tenemos un procedimiento al cual le pasamos un ListBox y nos lo rellena con la lista de programas instalados en Windows:

```
procedure ListarAplicaciones( Lista: TListBox );
const
  INSTALADOS = '\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall';
var
  Registro: TRegistry;
  Lista1 : TStringList;
  Lista2 : TStringList;
  j, n : integer;
begin
  Registro := TRegistry.Create;
  Lista1 := TStringList.Create;
  Lista2 := TStringList.Create;

  // Guardamos todas las claves en la lista 1
  with Registro do
  begin
    RootKey := HKEY_LOCAL_MACHINE;
    OpenKey( INSTALADOS, False );
    GetKeyNames( Lista1 );
  end;

  // Recorremos la lista 1 y leemos el nombre del programa instalado
  for j := 0 to Lista1.Count-1 do
  begin
    Registro.OpenKey( INSTALADOS + '\' + Lista1.Strings[j], False );
    Registro.GetValueNames( Lista2 );

    // Mostramos el programa instalado sólo si tiene DisplayName
    n := Lista2.IndexOf( 'DisplayName' );
    if ( n <> -1 ) and ( Lista2.IndexOf('UninstallString') <> -1 )
```

```
then
      Lista.Items.Add( ( Registro.ReadString( Lista2.Strings[n] ) ) );
  end;

  Lista.Sorted := True; // Ordenamos la lista alfabéticamente
  Lista1.Free;
  Lista2.Free;
  Registro.CloseKey;
  Registro.Destroy;
end;
```

Con esto se podría hacer un programa que eliminara de Agregar/Quitar programas aquellas claves de programas mal desinstalados.

Pruebas realizadas en Delphi 7.

Ejecutar un programa y esperar a que termine

Uno de los problemas habituales con los que se enfrenta un programador es que su cliente le pida algo que o bien no sabe como programarlo o no dispone del componente o librería necesaria para llevar tu tarea a cabo.

Un ejemplo puede ser realizar una copia de seguridad en formatos ZIP, RAR, 7Z, etc., convertir de un formato de video o sonido a otro e incluso llamar a comandos del sistema para realizar procesos criticos en un servidor. Entonces sólo se nos ocurre llamar a un programa externo que realice la tarea por nosostros (y que soporte parámetros).

Sé lo que estáis pensando (la función WinExec), pero en este caso no me vale ya que el programa tiene que esperar a que termine de ejecutarse antes de pasar al siguiente proceso.

Aquí os muestro un procedimiento que ejecuta un programa y se queda esperando a que termine:

```
function EjecutarYEsperar( sPrograma: String; Visibilidad: Integer ):
Integer;
var
  sAplicacion: array[0..512] of char;
  DirectorioActual: array[0..255] of char;
  DirectorioTrabajo: String;
  InformacionInicial: TStartupInfo;
  InformacionProceso: TProcessInformation;
  iResultado, iCodigoSalida: DWord;
begin
  StrPCopy( sAplicacion, sPrograma );
  GetDir( 0, DirectorioTrabajo );
  StrPCopy( DirectorioActual, DirectorioTrabajo );
  FillChar( InformacionInicial, Sizeof( InformacionInicial ), #0 );
```

```
  InformacionInicial.cb := Sizeof( InformacionInicial );

  InformacionInicial.dwFlags := STARTF_USESHOWWINDOW;
  InformacionInicial.wShowWindow := Visibilidad;
  CreateProcess( nil, sAplicacion, nil, nil, False,
                 CREATE_NEW_CONSOLE or NORMAL_PRIORITY_CLASS,
                 nil, nil, InformacionInicial, InformacionProceso );

  // Espera hasta que termina la ejecución
  repeat
    iCodigoSalida := WaitForSingleObject( InformacionProceso.hProcess,
1000 );
    Application.ProcessMessages;
  until ( iCodigoSalida <> WAIT_TIMEOUT );

  GetExitCodeProcess( InformacionProceso.hProcess, iResultado );
  MessageBeep( 0 );
  CloseHandle( InformacionProceso.hProcess );
  Result := iResultado;
end;
```

El parámetro iVisibilidad puede ser:

```
SW_SHOWNORMAL     -> Lo normal
SW_SHOWMINIMIZED  -> Minimizado (ventanas MS-DOS o ventanas no
modales)
SW_HIDE           -> Oculto     (ventanas MS-DOS o ventanas no
modales)
```

La función devuelve un cero si la ejecución terminó correctamente.

Por ejemplo para ejecutar la calculadora de Windows y esperar a que termine:

```
procedure EjecutarCalculadora;
begin
  if EjecutarYEsperar( 'C:\Windows\System32\Calc.exe', SW_SHOWNORMAL )
= 0 then
    ShowMessage( 'Ejecución terminada con éxito.' )
  else
    ShowMessage( 'Ejecución no terminada correctamente.' );
end;
```

Pruebas realizadas en Delphi 7.

Obtener modos de video

A la hora de realizar un programa hay que tener muy presente la resolución de la pantalla y el área de trabajo donde se pueden colocar las ventanas.

Para ello tenemos el objeto TScreen que nos devuelve no sólo la resolución actual de video sino que además nos da el tamaño del escritorio y el área de trabajo del mismo (si esta fija la barra de tareas hay que respetar su espacio y procurar que nuestra ventana no se superponga a la misma).

Si utilizamos las propiedades Position o WindowsState del formulario no hay que preocuparse por esto, pero si hemos creado nuestra propia piel y pasamos

de las ventanas normales de Windows hay que andarse con ojo y no dejar que el usuario se crea que ha desaparecido la barra de tareas.

El siguiente procedimiento vuelca la información de la pantalla actual en un objeto Memo llamado PANTALLA:

```
procedure TFInformacion.InfoPantalla;
begin
  PANTALLA.Lines.Clear;
  PANTALLA.Lines.Add( Format( 'Resolución: %dx%d ', [Screen.Width,
Screen.Height] ) );
  PANTALLA.Lines.Add( Format( 'Escritorio: x: %d y: %d Ancho: %d Alto:
%d',
                      [Screen.DesktopLeft, Screen.DesktopTop,
Screen.DesktopWidth, Screen.DesktopHeight] ) );
  PANTALLA.Lines.Add( Format( 'Area de trabajo: x: %d y: %d Ancho: %d
Alto: %d',
                      [Screen.WorkAreaLeft, Screen.WorkAreaTop,
Screen.WorkAreaWidth, Screen.WorkAreaHeight] ) );
end;
```

Otra información interesante sería saber que resoluciones posibles tiene nuestra tarjeta de video. Vamos a mostrar todos los modos de video posible en un ListBox llamado MODOS:

```
procedure TFInformacion.InfoModosVideo;
var i: Integer;
    ModoVideo: TDevMode;
begin
  i := 0;
  MODOS.Clear;
  while EnumDisplaySettings( nil, i, ModoVideo ) do
  begin
    with ModoVideo do
      MODOS.Items.Add(Format( '%dx%d %d Colores', [dmPelsWidth,
dmPelsHeight, Int64(1) shl dmBitsperPel] ) );

    Inc( i );
  end;
end;
```

Esta es la típica información que suelen mostrar los programas tipo TuneUp.

Pruebas realizadas en Delphi 7.

Utilizar una fuente TTF sin instalarla

Uno de los primeros inconvenientes al distribuir nuestras aplicaciones es ver que en otros Windows aparecen nuestras etiquetas y campos desplazados debido a que la fuente que utilizan no es la que tenemos nosotros en nuestro equipo.

O bien utilizamos fuentes estandar tales como Tahoma, Arial, etc. o podemos utilizar el siguiente truco que consiste en añadir la fuente utilizada al lado de nuestro ejecutable y cargarla al arrancar nuestra aplicación.

El procedimiento para cargar una fuente es:

```
procedure CargarFuente( sFuente: String );
begin
  AddFontResource( PChar( ExtractFilePath( Application.ExeName ) +
sFuente ) );
  SendMessage( HWND_BROADCAST, WM_FONTCHANGE, 0, 0 );
end;
```

Y en el procedimiento OnCreate de nuestro formulario cargamos la fuente:

```
procedure TFPrincipal.FormCreate( Sender: TObject );
begin
  CargarFuente( 'Diner.ttf' );
  Etiqueta.Font.Name := 'Diner';
end;
```

Donde se supone que el archivo Diner.ttf está al lado de nuestro ejecutable.

Antes de cerrar nuestra aplicación debemos liberar de memoria la fuente utilizada con el procedimiento:

```
procedure EliminarFuente( sFuente: String );
begin
  RemoveFontResource( PChar( ExtractFilePath( Application.ExeName ) +
sFuente ) );
  SendMessage( HWND_BROADCAST, WM_FONTCHANGE, 0, 0 );
end;
```

Este prodecimiento sería llamado en el evento OnDestroy del formulario:

```
procedure TFPrincipal.FormDestroy( Sender: TObject );
begin
  EliminarFuente( 'Diner.ttf' );
end;
```

Es recomendable hacer esto una sola vez en el formulario principal de la aplicación y no en cada formulario del programa, a menos que tengamos un formulario que utiliza exclusivamente una fuente en concreto.

Pruebas realizadas en Delphi 7.

Convertir un icono en imagen BMP

Aunque hay cientos de librerías de iconos por la red que suele utilizar todo el mundo para sus programas, lo ideal sería diseñar nuestros propios iconos utilizando como plantilla los que hay por Internet.

Se que hay decenas de programas de diseño que convierten entre distintos formatos, pero lo ideal sería tener nuestro propio conversor. Para ello tenemos el siguiente procedimiento que convierte un archivo ICO en una imagen BMP para que luego podamos utilizarla en nuestras aplicaciones:

```
procedure ConvertirImagen( sIcono, sBMP: String );
var
  Bitmap: TBitmap;
  Imagen: TImage;
begin
  Imagen := TImage.Create( nil );
  Imagen.Picture.LoadFromFile( sIcono );
  Bitmap := TBitMap.Create;

  with Bitmap do
  begin
    PixelFormat := pf24bit;
    Height := Application.Icon.Height;
    Width := Application.Icon.Width;
    Canvas.Draw( 0, 0, Imagen.Picture.Graphic );
  end;

  Bitmap.SavetoFile( sBMP );

  Imagen.Free;
end;
```

El primer parámetro es el icono y el segundo la imagen BMP resultante. Lo que hace el procedimiento es cargar el icono en un objeto TImage, para después copiar su contenido en un bitmap antes de guardarlo en un archivo.

Pruebas realizadas en Delphi 7.

Borrar archivos temporales de Internet

Uno de los componentes más útiles de Delphi hoy en día es WebBrowser el cual nos permite crear dentro de nuestros programas un navedador web utilizando el motor de Internet Explorer.

El único inconveniente es que al finalizar nuestro programa tenemos que ir a Internet Explorer y vaciar la caché, evitando que se llene el disco duro de basura.

Pues vamos a ver un procedimiento que elimina los archivos temporales de Internet Explorer, no sin antes añadir la unidad WinINet:

```
uses
  Windows, Messages, ..., WinInet;

procedure BorrarCacheIE;
var
  lpEntryInfo: PInternetCacheEntryInfo;
  hCacheDir: LongWord;
  dwEntrySize: LongWord;
begin
  dwEntrySize := 0;
  FindFirstUrlCacheEntry( nil, TInternetCacheEntryInfo( nil^ ),
dwEntrySize );
  GetMem( lpEntryInfo, dwEntrySize );
```

```
  if dwEntrySize > 0 then
    lpEntryInfo^.dwStructSize := dwEntrySize;

  hCacheDir := FindFirstUrlCacheEntry( nil, lpEntryInfo^, dwEntrySize
);

  if hCacheDir <> 0 then
  begin
    repeat
      DeleteUrlCacheEntry( lpEntryInfo^.lpszSourceUrlName );
      FreeMem( lpEntryInfo, dwEntrySize );
      dwEntrySize := 0;
      FindNextUrlCacheEntry( hCacheDir, TInternetCacheEntryInfo( nil^
), dwEntrySize );
      GetMem( lpEntryInfo, dwEntrySize );

      if dwEntrySize > 0 then
        lpEntryInfo^.dwStructSize := dwEntrySize;

    until not FindNextUrlCacheEntry( hCacheDir, lpEntryInfo^,
dwEntrySize );
  end;

  FreeMem( lpEntryInfo, dwEntrySize );
  FindCloseUrlCache( hCacheDir );
end;
```

Este procedimiento habría que ejecutarlo al cerrar nuestro programa dejando el sistema limpio.

Pruebas realizadas en Delphi 7.

Deshabilitar el cortafuegos de Windows XP

Una de las tareas más frecuentes a las que se enfrenta un programador es la de crear aplicaciones que automaticen procesos de nuestra aplicación tales como subir datos por FTP, conectar con otro motor de bases de datos para enviar, etc.

Y si hay algún programa que pueda interrumpir el proceso de cara a las comunicaciones TCP/IP es el cortafuegos de Windows. Primero añadimos a uses:

```
uses
  Windows, Messages, ..., WinSvc, ShellApi;
```

Este sería un el procedimiento que detiene el servicio:

```
procedure DeshabilitarCortafuegosXP;
var
  SCM, hService: LongWord;
  sStatus: TServiceStatus;
begin
  SCM := OpenSCManager( nil, nil, SC_MANAGER_ALL_ACCESS );
  hService := OpenService( SCM, PChar( 'SharedAccess' ),
```

```
SERVICE_ALL_ACCESS );
  ControlService( hService, SERVICE_CONTROL_STOP, sStatus );
  CloseServiceHandle( hService );
end;
```

Para volver a activarlo sólo hay que ir al panel de control y ponerlo en marcha de nuevo. Esto no vale para otros cortafuegos (Panda, Norton, etc.)

Pruebas realizadas en Delphi 7.

Leer el número de serie de una unidad

Cuando se vende un programa generalmente se suele poner el precio según el número de equipos donde se va a instalar (licencia). Proteger nuestra aplicación contra copias implica leer algo característico en el PC que lo haga único.

Pues bien, cuando se formatea una unidad de disco Windows le asigna un número de serie que no cambiará hasta que vuelva a ser formateada. Lo que vamos a hacer es una función que toma como parámetro la unidad de disco que le pasemos (C:, D:, ...) y nos devolverá su número de serie:

```
function LeerSerieDisco( cUnidad: Char ): String;
var
  dwLongitudMaxima, VolFlags, dwSerie: DWord;
begin
  if GetVolumeInformation( PChar( cUnidad + ':\' ), nil, 0,
                           @dwSerie, dwLongitudMaxima, VolFlags, nil,
0) then
  begin
    // devolvemos el número de serie en hexadecimal
    Result := IntToHex( dwSerie, 8 );
    Insert( '-', Result, 5 );
  end
  else
    Result := '';
end;
```

Nos devolverá algo como esto:

D4BD-0EC7

Con ese número ya podemos crear nuestro propio keygen alterando las letras, el orden o utilizando el algoritmo de encriptación que nos apetezca.

El único inconveniente es que si el usuario vuelve a formatear esa unidad

entonces nos tiene que volver a pedir el número de serie. Hay otros programadores que prefieren leer el número de la BIOS o de la tarjeta de video, ya depende del nivel de protección que se desee.

Pruebas realizadas en Delphi 7.

Leer los archivos del portapapeles

El siguiente procedimiento lee el nombre de archivos o directorios del portapapales capturados por el usuario con CTRL + C ó CTRL + X y los muestra en un ListBox que le pasamos como parámetro:

```
procedure LeerArchivosPortapapeles( Lista: TListBox );
var
  HPortapapeles: THandle;  // Handle del portapapeles
  iNumArc, i: Integer;     // Nº de archivos
  Archivo: array [0..MAX_PATH - 1] of char;
begin
  if ClipBoard.HasFormat( CF_HDROP ) then
  begin
    HPortapapeles := ClipBoard.GetAsHandle( CF_HDROP );
    iNumArc := DragQueryFile( HPortapapeles, $FFFFFFFF, nil, 0);

    for i := 0 to iNumArc - 1 do
    begin
      DragQueryFile( HPortapapeles, i, @Archivo, MAX_PATH );
      Lista.Items.Add( Archivo );
    end;
  end;
end;
```

Para poder compilarlo hay que añadir las unidades externas:

```
uses
  Windows, Messages, ..., ClipBrd, ShellAPI;
```

Sólo mostrará archivos o directorios y no imágenes o cualquier otro archivo capturado dentro de un programa. Puede sernos de utilidad para realizar programas de copia de seguridad, conversiones de archivo, etc.

Pruebas realizadas en Delphi 7.

Averiguar los datos del usuario de Windows

Una de las mejores cosas que se pueden hacer en un programa cuando da un error es que nos envíe automáticamente los datos por correo electrónico. Pero es importante saber que usuario ha enviado el error dentro de la red local.

A continuación vamos a ver cuatro procedimientos que nos van a dar el nombre del usuario de Windows, el nombre de su PC en la red, su IP local y su IP pública.

Lo primero como siempre es añadir las unidades:

```
uses
  Windows, Messages, ..., WinSock, IdHttp, WinInet;
```

Esta función nos devuelve el nombre del usuario:

```
function LeerUsuarioWindows: string;
var
  sNombreUsuario: String;
  dwLongitudNombre: DWord;
begin
  dwLongitudNombre := 255;
  SetLength( sNombreUsuario, dwLongitudNombre );

  if GetUserName( PChar( sNombreUsuario ), dwLongitudNombre ) Then
    Result := Copy( sNombreUsuario, 1, dwLongitudNombre - 1 )
  else
    Result := 'Desconocido';
end;
```

Y esta otra nos da el nombre del PC en la red:

```
function LeerNombrePC: string;
var
  Buffer: array[0..255] of char;
  dwLongitud: DWord;
begin
  dwLongitud := 256;

  if GetComputerName( Buffer, dwLongitud ) then
    Result := Buffer
  else
    Result := ''
end;
```

La siguiente nos da la IP Local en la red:

```
function IPLocal: String;
var
  p: PHostEnt;
  s: array[0..128] of char;
  p2: pchar;
  wVersionRequested: WORD;
  wsaData: TWSAData;
begin
  // Arranca la librería WinSock
  wVersionRequested := MAKEWORD( 1, 1 );
  WSAStartup( wVersionRequested, wsaData );

  // Obtiene el nombre del PC
  GetHostName( @s, 128 );
  p := GetHostByName( @s );

  // Obtiene la dirección IP y libera la librería WinSock
  p2 := iNet_ntoa( PInAddr( p^.h_addr_list^ )^ );
```

```
  Result := Result + p2;
  WSACleanup;
end;
```

Y esta última lo que hace es decirnos nuestra IP pública conectando con el servidor dyndns.org y utiliza el componente Indy HTTP el cual leer el contenido del HTML:

```
function IP_Publica: string;

  function EsNumerico( S: string ): Boolean;
  begin
    Result := false;
    if ( Length( S ) > 0 ) then
      case S[1] of
        '0'..'9': Result := True;
      end;
  end;
var
  HTMLBody: string;
  i: Integer;
  IdHTTP: TIdHTTP;
begin
  Result := '';

  // ¿Estamos conectados a Internet?
  if WinInet.InternetGetConnectedState( nil, 0 ) then
  begin
    IdHTTP := TIdHTTP.Create( Application );

    try
      HTMLBody := IdHTTP.Get( 'http://checkip.dyndns.org/' );

      for i := 0 to Length( HTMLBody ) - 1 do
      begin
        if EsNumerico( HTMLBody[i] ) or ( HTMLBody[i] = '.' ) then
          Result := Result + HTMLBody[i];
      end;

     finally
       IdHTTP.Free;
     end;
  end;
end;
```

Pruebas realizadas en Delphi 7.

Leer la cabecera PE de un programa

¿Queréis verle las tripas a un archivo EXE? El siguiente procedimiento que voy a mostrar lee la cabecera PE de los archivos ejecutables y nos informa del punto de entrada del programa, el estado de los registros, la pila, etc.

Un archivo ejecutable se compone de distintas cabeceras dentro del mismo, ya sea si se va a ejecutar dentro del antiguo sistema operativo MS-DOS o en cualquier versión de Windows.

El siguiente procedimiento toma como parámetro un archivo ejecutable y lo

guarda en un supuesto campo memo llamado INFORMACION que se encuentra en el formulario FPrincipal:

```
procedure TFPrincipal.ExaminarEXE( sArchivo: String );
var
  FS: TFilestream;
  Firma: DWORD;
  Cabecera_dos: IMAGE_DOS_HEADER;
  Cabecera_pe: IMAGE_FILE_HEADER;
  Cabecera_opc: IMAGE_OPTIONAL_HEADER;
begin
  INFORMACION.Clear;

  FS := TFilestream.Create( sArchivo, fmOpenread or fmShareDenyNone );

  try
    FS.Read( Cabecera_dos, SizeOf( Cabecera_dos ) );

    if Cabecera_dos.e_magic <> IMAGE_DOS_SIGNATURE then
    begin
      INFORMACION.Lines.Add( 'Cabecera DOS inválida' );
      Exit;
    end;

    LeerCabeceraDOS( Cabecera_dos, INFORMACION.Lines );

    FS.Seek( Cabecera_dos._lfanew, soFromBeginning );
    FS.Read( Firma, SizeOf( Firma ) );

    if Firma <> IMAGE_NT_SIGNATURE then
    begin
      INFORMACION.Lines.Add( 'Cabecera PE inválida' );
      Exit;
    end;

    FS.Read( Cabecera_pe, SizeOf( Cabecera_pe ) );
    LeerCabeceraPE( Cabecera_pe, INFORMACION.Lines );

    if Cabecera_pe.SizeOfOptionalHeader > 0 then
    begin
      FS.Read( Cabecera_opc, SizeOf( Cabecera_opc ) );
      LeerCabeceraOpcional( Cabecera_opc, INFORMACION.Lines );
    end;
  finally
    FS.Free;
  end;
end;
```

Éste a su vez llama a cada uno de los procedimientos que leen las cabeceras DOS, PE y opcional dentro del mismo EXE:

```
procedure LeerCabeceraDOS( const h: IMAGE_DOS_HEADER; Memo: TStrings
);
begin
  Memo.Add( 'Cabecera DOS del archivo' );
  Memo.Add( Format( 'Número mágico: %d', [h.e_magic] ) );
  Memo.Add( Format( 'Byes de la última página del archivo: %d',
[h.e_cblp] ) );
```

```
  Memo.Add( Format( 'Páginas en archivo: %d', [h.e_cp] ) );
  Memo.Add( Format( 'Relocalizaciones: %d', [h.e_crlc] ) );
  Memo.Add( Format( 'Tamaño de la cabecera en párrafos: %d',
[h.e_cparhdr] ) );
  Memo.Add( Format( 'Mínimo número de párrafos que necesita: %d',
[h.e_minalloc] ) );
  Memo.Add( Format( 'Máximo número de párrafos que necesita: %d',
[h.e_maxalloc] ) );
  Memo.Add( Format( 'Valor inicial (relativo) SS: %d', [h.e_ss] ) );
  Memo.Add( Format( 'Valor inicial SP: %d', [h.e_sp] ) );
  Memo.Add( Format( 'Checksum: %d', [h.e_csum]));
  Memo.Add( Format( 'Valor inicial IP: %d', [h.e_ip] ) );
  Memo.Add( Format( 'Valor inicial (relativo) CS: %d', [h.e_cs] ) );
  Memo.Add( Format( 'Dirección del archivo de la tabla de
relocalización: %d', [h.e_lfarlc] ) );
  Memo.Add( Format( 'Número overlay: %d', [h.e_ovno]));
  Memo.Add( Format( 'Identificador OEM (para e_oeminfo): %d',
[h.e_oemid] ) );
  Memo.Add( Format( 'Información OEM; específico e_oemid: %d',
[h.e_oeminfo] ) );
  Memo.Add( Format( 'Dirección de la nueva cabecera exe: %d',
[h._lfanew] ) );
  Memo.Add( '' );
end;

procedure LeerCabeceraPE( const h: IMAGE_FILE_HEADER; Memo: TStrings
);
var
  Fecha: TDateTime;
begin
  Memo.Add( 'Cabecera PE del archivo' );
  Memo.Add( Format( 'Máquina: %4x', [h.Machine]));

  case h.Machine of
    IMAGE_FILE_MACHINE_UNKNOWN : Memo.Add(' Máquina desconocida ' );
    IMAGE_FILE_MACHINE_I386: Memo.Add( ' Intel 386. ' );
    IMAGE_FILE_MACHINE_R3000: Memo.Add( ' MIPS little-endian, 0x160
big-endian ' );
    IMAGE_FILE_MACHINE_R4000: Memo.Add( ' MIPS little-endian ' );
    IMAGE_FILE_MACHINE_R10000: Memo.Add( ' MIPS little-endian ' );
    IMAGE_FILE_MACHINE_ALPHA: Memo.Add( ' Alpha_AXP ' );
    IMAGE_FILE_MACHINE_POWERPC: Memo.Add( ' IBM PowerPC Little-Endian
' );
    $14D: Memo.Add( ' Intel i860' );
    $268: Memo.Add( ' Motorola 68000' );
    $290: Memo.Add( ' PA RISC' );
    else
      Memo.Add( ' tipo de máquina desconocida' );
  end;

  Memo.Add( Format( 'Número de secciones: %d', [h.NumberOfSections] )
);
  Memo.Add( Format( 'Fecha y hora: %d', [h.TimeDateStamp] ) );
  Fecha := EncodeDate( 1970, 1, 1 ) + h.Timedatestamp / SecsPerDay;
  Memo.Add( FormatDateTime( ' c', Fecha ) );

  Memo.Add( Format( 'Puntero a la tabla de símbolos: %d',
[h.PointerToSymbolTable] ) );
  Memo.Add( Format( 'Número de símbolos: %d', [h.NumberOfSymbols] ) );
  Memo.Add( Format( 'Tamaño de la cabecera opcional: %d',
[h.SizeOfOptionalHeader] ) );
```

```
  Memo.Add( Format( 'Características: %d', [h.Characteristics] ) );

  if ( IMAGE_FILE_DLL and h.Characteristics ) <> 0 then
    Memo.Add(' el archivo es una' )
  else
    if (IMAGE_FILE_EXECUTABLE_IMAGE and h.Characteristics) <> 0 then
      Memo.Add(' el archivo es un programa' );

  Memo.Add('');
end;

procedure LeerCabeceraOpcional( const h: IMAGE_OPTIONAL_HEADER; Memo:
TStrings );
begin
  Memo.Add( 'Información sobre la cabecera PE de un archivo ejecutable
EXE' );
  Memo.Add( Format( 'Magic: %d', [h.Magic] ) );

  case h.Magic of
    $107: Memo.Add( ' Imagen de ROM' );
    $10b: Memo.Add( ' Imagen de ejecutable' );
    else
      Memo.Add( ' Tipo de imagen desconocido' );
  end;

  Memo.Add( Format( 'Versión mayor del enlazador: %d',
[h.MajorLinkerVersion] ) );
  Memo.Add( Format( 'Versión menor del enlazador: %d',
[h.MinorLinkerVersion]));
  Memo.Add( Format( 'Tamaño del código: %d', [h.SizeOfCode]));
  Memo.Add( Format( 'Tamaño de los datos inicializados: %d',
[h.SizeOfInitializedData]));
  Memo.Add( Format( 'Tamaño de los datos sin inicializar: %d',
[h.SizeOfUninitializedData]));
  Memo.Add( Format( 'Dirección del punto de entrada: %d',
[h.AddressOfEntryPoint]));
  Memo.Add( Format( 'Base de código: %d', [h.BaseOfCode]));
  Memo.Add( Format( 'Base de datos: %d', [h.BaseOfData]));
  Memo.Add( Format( 'Imagen base: %d', [h.ImageBase]));
  Memo.Add( Format( 'Alineamiento de la sección: %d',
[h.SectionAlignment]));
  Memo.Add( Format( 'Alineamiento del archivo: %d',
[h.FileAlignment]));
  Memo.Add( Format( 'Versión mayor del sistema operativo: %d',
[h.MajorOperatingSystemVersion]));
  Memo.Add( Format( 'Versión mayor del sistema operativo: %d',
[h.MinorOperatingSystemVersion]));
  Memo.Add( Format( 'Versión mayor de la imagen: %d',
[h.MajorImageVersion]));
  Memo.Add( Format( 'Versión menor de la imagen: %d',
[h.MinorImageVersion]));
  Memo.Add( Format( 'Versión mayor del subsistema: %d',
[h.MajorSubsystemVersion]));
  Memo.Add( Format( 'Versión menor del subsistema: %d',
[h.MinorSubsystemVersion]));
  Memo.Add( Format( 'Valor de la versión Win32: %d',
[h.Win32VersionValue]));
  Memo.Add( Format( 'Tamaño de la imagen: %d', [h.SizeOfImage]));
  Memo.Add( Format( 'Tamaño de las cabeceras: %d',
[h.SizeOfHeaders]));
  Memo.Add( Format( 'CheckSum: %d', [h.CheckSum]));
```

```
  Memo.Add( Format( 'Subsistema: %d', [h.Subsystem]));

  case h.Subsystem of
    IMAGE_SUBSYSTEM_NATIVE:
      Memo.Add( ' La imagen no requiere un subsistema. ' );

    IMAGE_SUBSYSTEM_WINDOWS_GUI:
      Memo.Add( ' La imagen se corre en un subsistema GUI de Windows.
' );

    IMAGE_SUBSYSTEM_WINDOWS_CUI:
      Memo.Add( ' La imagen corre en un subsistema terminal de
Windows. ' );

    IMAGE_SUBSYSTEM_OS2_CUI:
      Memo.Add( ' La imagen corre sobre un subsistema terminal de
OS/2. ' );

    IMAGE_SUBSYSTEM_POSIX_CUI:
      Memo.Add( ' La imagen corre sobre un subsistema terminal Posix.
' );
    else
      Memo.Add( ' Subsistema desconocido.' )
  end;

  Memo.Add( Format( 'Características DLL: %d', [h.DllCharacteristics])
);
  Memo.Add( Format( 'Tamaño de reserva de la pila: %d',
[h.SizeOfStackReserve]) );
  Memo.Add( Format( 'Tamaño de trabajo de la pila: %d',
[h.SizeOfStackCommit]) );
  Memo.Add( Format( 'Tamaño del Heap de reserva: %d',
[h.SizeOfHeapReserve]) );
  Memo.Add( Format( 'Tamaño de trabajo del Heap: %d',
[h.SizeOfHeapCommit]) );
  Memo.Add( Format( 'Banderas de carga: %d', [h.LoaderFlags] ) );
  Memo.Add( Format( 'Numeros RVA y tamaño: %d',
[h.NumberOfRvaAndSizes] ) );
end;
```

Espero que os sea de utilidad si os gusta programar herramientas de administración de sistemas operativos Windows.

Pruebas realizadas en Delphi 7.

Leer las dimensiones de imágenes JPG, PNG y GIF

Si estáis pensando en crear un visor de fotografías aquí os traigo tres procedimientos que leen al ancho y alto de imagenes con extensión JPG, PNG y GIF leyendo los bytes de su cabecera. No hay para BMP ya que se puede hacer con un componente TImage.

Antes de nada hay que incluir una función que lee los enteros almacenados en formato del procesador motorola, que guarda los formatos enteros en memoria al contrario de los procesadores Intel/AMD:

```
function LeerPalabraMotorola( F: TFileStream ): Word;
type
  TPalabraMotorola = record
    case Byte of
      0: ( Value: Word );
      1: ( Byte1, Byte2: Byte );
  end;
var
  MW: TPalabraMotorola;
begin
  F.Read( MW.Byte2, SizeOf( Byte ) );
  F.Read( MW.Byte1, SizeOf( Byte ) );
  Result := MW.Value;
end;
```

El siguiente procedimiento toma como parámetros la ruta y nombre de una imagen JPG, y dos variales enteras donde se almacenará el ancho y alto de la imagen:

```
procedure DimensionJPG( sArchivo: string; var wAncho, wAlto: Word );
const
  ValidSig: array[0..1] of Byte = ($FF, $D8);
  Parameterless = [$01, $D0, $D1, $D2, $D3, $D4, $D5, $D6, $D7];
var
  Sig: array[0..1] of byte;
  F: TFileStream;
  x: integer;
  Seg: byte;
  Dummy: array[0..15] of byte;
  Len: word;
  iLongitudLinea: LongInt;
begin
  FillChar( Sig, SizeOf( Sig ), #0 );

  F := TFileStream.Create( sArchivo, fmOpenRead );
  try
    iLongitudLinea := F.Read( Sig[0], SizeOf( Sig ) );

    for x := Low( Sig ) to High( Sig ) do
      if Sig[x] <> ValidSig[x] then
        iLongitudLinea := 0;

    if iLongitudLinea > 0 then
    begin
      iLongitudLinea := F.Read( Seg, 1 );

      while ( Seg = $FF ) and ( iLongitudLinea > 0 ) do
      begin
        iLongitudLinea := F.Read( Seg, 1 );

        if Seg <> $FF then
        begin
          if ( Seg = $C0 ) or ( Seg = $C1 ) then
          begin
            iLongitudLinea := F.Read( Dummy[0], 3 ); // Nos saltamos
estos bytes
            wAlto := LeerPalabraMotorola( F );
            wAncho  := LeerPalabraMotorola( F );
          end
```

```
          else
          begin
            if not ( Seg in Parameterless ) then
            begin
              Len := LeerPalabraMotorola( F );
              F.Seek( Len - 2, 1 );
              F.read( Seg, 1 );
            end
            else
              Seg := $FF; { Fake it to keep looping. }
          end;
        end;
      end;
    end;
  finally
    F.Free;
  end;
end;
```

Lo mismo para una imagen PNG:

```
procedure DimensionPNG( sArchivo: string; var wAncho, wAlto: Word );
type
  TPNGSig = array[0..7] of Byte;
const
  ValidSig: TPNGSig = (137,80,78,71,13,10,26,10);
var
  Sig: TPNGSig;
  F: TFileStream;
  x: Integer;
begin
  FillChar( Sig, SizeOf( Sig ), #0 );
  F := TFileStream.Create( sArchivo, fmOpenRead );
  try
    F.read( Sig[0], SizeOf( Sig ) );

    for x := Low( Sig ) to High( Sig ) do
      if Sig[x] <> ValidSig[x] then
        Exit;

    F.Seek( 18, 0 );
    wAncho := LeerPalabraMotorola( F );
    F.Seek( 22, 0 );
    wAlto := LeerPalabraMotorola( F );
  finally
    F.Free;
  end;
end;
```

Y para una imagen GIF:

```
procedure DimensionGIF( sArchivo: string; var wAncho, wAlto: Word );
type
  TCabeceraGIF = record
    Sig: array[0..5] of char;
    ScreenWidth, ScreenHeight: Word;
    Flags, Background, Aspect: Byte;
  end;

  TBloqueImagenGIF = record
```

```
    Left, Top, Width, Height: Word;
    Flags: Byte;
  end;
var
  F: file;
  Cabecera: TCabeceraGIF;
  BloqueImagen: TBloqueImagenGIF;
  iResultado: Integer;
  x: Integer;
  c: char;
  bEncontradasDimensiones: Boolean;
begin
  wAncho  := 0;
  wAlto := 0;

  if sArchivo = '' then
    Exit;

  {$I-}
  FileMode := 0;   // Sólo lectura
  AssignFile( F, sArchivo );
  Reset( F, 1);
  if IOResult <> 0 then
    Exit;

  // Lee la cabecera y se asegura de que sea un archivo válido
  BlockRead( F, Cabecera, SizeOf( TCabeceraGIF ), iResultado );

  if ( iResultado <> SizeOf( TCabeceraGIF ) ) or ( IOResult <> 0 ) or
     ( StrLComp( 'GIF', Cabecera.Sig, 3 ) <> 0 ) then
  begin
    Close( F );
    Exit;
  end;

  { Skip color map, if there is one }
  if ( Cabecera.Flags and $80 ) > 0 then
  begin
    x := 3 * ( 1 shl ( ( Cabecera.Flags and 7 ) + 1 ) );
    Seek( F, x );
    if IOResult <> 0 then
    begin
      Close( F );
      Exit;
    end;
  end;

  bEncontradasDimensiones := False;
  FillChar( BloqueImagen, SizeOf( TBloqueImagenGIF ), #0 );

  BlockRead( F, c, 1, iResultado );
  while ( not EOF( F ) ) and ( not bEncontradasDimensiones ) do
  begin
    case c of
      ',': // Encontrada imagen
        begin
          BlockRead( F, BloqueImagen, SizeOf( TBloqueImagenGIF ),
iResultado );
          if iResultado <> SizeOf( TBloqueImagenGIF ) then
          begin
            Close( F );
```

```
            Exit;
          end;

          wAncho := BloqueImagen.Width;
          wAlto := BloqueImagen.Height;
          bEncontradasDimensiones := True;
        end;

      'ÿ': // esquivar esto
        begin
          // Nada
        end;

      // No hacer nada, ignorar
    end;

    BlockRead( F, c, 1, iResultado );
  end;

  Close( F );
  {$I+}
end;
```

Así no será necesario cargar toda la imagen para averiguar sus dimensiones.

Pruebas realizadas en Delphi 7.

Descargar un archivo de Internet sin utilizar componentes

Añadiendo a nuestro formulario la librería WinINet se pueden descargar archivos por HTTP con la siguiente función:

```
function DescargarArchivo( sURL, sArchivoLocal: String ): boolean;
const BufferSize = 1024;
var
  hSession, hURL: HInternet;
  Buffer: array[1..BufferSize] of Byte;
  LongitudBuffer: DWORD;
  F: File;
  sMiPrograma: String;
begin
  sMiPrograma := ExtractFileName( Application.ExeName );
  hSession := InternetOpen( PChar( sMiPrograma ),
INTERNET_OPEN_TYPE_PRECONFIG, nil, nil, 0 );

  try
    hURL := InternetOpenURL( hSession, PChar( sURL ), nil, 0, 0, 0 );

    try
      AssignFile( F, sArchivoLocal );
```

```
      Rewrite( F, 1 );

      repeat
        InternetReadFile( hURL, @Buffer, SizeOf( Buffer ),
LongitudBuffer );
        BlockWrite( F, Buffer, LongitudBuffer );
      until LongitudBuffer = 0;

      CloseFile( F );
      Result := True;
    finally
      InternetCloseHandle( hURL );
    end
  finally
    InternetCloseHandle( hSession );
  end
end;
```

El primer parámetro es la URL completa del archivo a descargar y el segundo la ruta y nombre del archivo donde se va a guardar en nuestro disco duro. Un ejemplo de llamada a la función sería:

```
DescargarArchivo( 'http:\\miweb.com\imagen.jpg', 'C:\Mis
documentos\imagen.jpg' );
```

Averiguar el nombre del procesador y su velocidad

El registro de Windows suele almacenar gran cantidad de información no sólo de la configuración de los programas instalados, sino también el estado real del hardware de nuestro PC.

En esta ocasión vamos a leer el nombre del procesador y su velocidad desde nuestro programa. Antes de nada añadimos a uses:

```
uses
  Windows, Messages, ..., Registry;
```

La siguiente función nos devuelve el nombre del procesador:

```
function NombreProcesador: string;
var
  Registro: TRegistry;
begin
  Result := '';
  Registro := TRegistry.Create;

  try
```

```
    Registro.RootKey := HKEY_LOCAL_MACHINE;

    if Registro.OpenKey(
'\Hardware\Description\System\CentralProcessor\0', False ) then
      Result := Registro.ReadString( 'Identifier' );
  finally
    Registro.Free;
  end;
end;
```

Y esta otra nos da su velocidad (según la BIOS y el fabricante):

```
function VelocidadProcesador: string;
var
  Registro: TRegistry;
begin
  Registro := TRegistry.Create;
  try
    Registro.RootKey := HKEY_LOCAL_MACHINE;

    if Registro.OpenKey(
'Hardware\Description\System\CentralProcessor\0', False ) then
    begin
      Result := IntToStr( Registro.ReadInteger( '~MHz' ) ) + ' MHz';
      Registro.CloseKey;
    end;
  finally
    Registro.Free;
  end;
end;
```

Hay veces que dependiendo del procesador y del multiplicador de la BIOS casi nunca coincide la velocidad real que nos da Windows con la de verdad (sobre todo en procesadores AMD). Aquí tenemos otra función que calcula en un segundo la velocidad real del procesador con una pequeña rutina en ensamblador:

```
function CalcularVelocidadProcesador: Double;
const
  Retardo = 500;
var
  TimerHi, TimerLo: DWORD;
  ClasePrioridad, Prioridad: Integer;
begin
  ClasePrioridad := GetPriorityClass( GetCurrentProcess );
  Prioridad := GetThreadPriority( GetCurrentThread );

  SetPriorityClass( GetCurrentProcess, REALTIME_PRIORITY_CLASS );
  SetThreadPriority( GetCurrentThread, THREAD_PRIORITY_TIME_CRITICAL
);

  Sleep( 10 );

  asm
    dw 310Fh
    mov TimerLo, eax
    mov TimerHi, edx
  end;
```

```
  Sleep( Retardo );

  asm
    dw 310Fh
    sub eax, TimerLo
    sbb edx, TimerHi
    mov TimerLo, eax
    mov TimerHi, edx
  end;

  SetThreadPriority( GetCurrentThread, Prioridad );
  SetPriorityClass( GetCurrentProcess, ClasePrioridad );

  Result := TimerLo / ( 1000 * Retardo );
end;
```

Nos devuelve el resultado en una variable double, donde que podríamos sacar la información en pantalla de la siguiente manera:

```
ShowMessage( Format( 'Velocidad calculada: %f MHz',
[CalcularVelocidadProcesador] ) );
```

Pruebas realizadas en Delphi 7.

Generar claves aleatorias

El siguiente procedimiento que voy a mostrar genera una clave aleatoria según el número de sílabas y números que le indiquemos. Por ejemplo:

```
GenerarClave( 3, 2 )
```

puede devolver:

```
catamo56
dilema12
catoye97
...
```

Aquí tenemos el generador de claves:

```
function GenerarClave( iSilabas, iNumeros: Byte ): string;
const
  Consonante: array [0..19] of Char = ( 'b', 'c', 'd', 'f', 'g', 'h',
'j',
                                        'k', 'l', 'm', 'n', 'p', 'r',
's',
                                        't', 'v', 'w', 'x', 'y', 'z'
);
```

```
  Vocal: array [0..4] of Char = ( 'a', 'e', 'i', 'o', 'u' );

  function Repetir( sCaracter: String; iVeces: Integer ): string;
  var
    i: Integer;
  begin
    Result := '';
    for i := 1 to iVeces do
      Result := Result + sCaracter;
  end;

var
  i: Integer;
  si, sf: Longint;
  n: string;
begin
  Result := '';
  Randomize;

  if iSilabas <> 0 then
    for i := 1 to iSilabas do
    begin
      Result := Result + Consonante[Random(19)];
      Result := Result + Vocal[Random(4)];
    end;

  if iNumeros = 1 then
    Result := Result + IntToStr(Random(9))
  else
    if iNumeros >= 2 then
    begin
      if iNumeros > 9 then
        iNumeros := 9;

      si     := StrToInt('1' + Repetir( '0', iNumeros - 1 ) );
      sf     := StrToInt( Repetir( '9', iNumeros ) );
      n      := FloatToStr( si + Random( sf ) );
      Result := Result + Copy( n, 0, iNumeros );
    end;
end;
```

Suele utilizarse en la creación de cuentas de usuario y departamentos en programas de gestión, dejando que posteriormente el usuario pueda cambiarse la clave.

Pruebas realizadas en Delphi 7.

Meter recursos dentro de un ejecutable

Una de las cosas que mas hacen engordar el tamaño de un ejecutable es meter los mismos botones con imagenes en distintos formularios. Si diseñamos nuestros propios botones Aceptar y Cancelar y los vamos replicando en cada formulario el tamaño de nuestro programa puede crecer considerablemente.

Para evitar esto lo que vamos a hacer es meter la imagen dentro de un recurso compilado y posteriormente accederemos a la misma dentro del programa. La ventaja de utilizar este método es que la imagen sólo esta una vez en el programa independientemente del número de formularios donde

vaya a aparecer.

Supongamos que vamos a meter la imagen MiBoton.jpg dentro de nuestro programa. Creamos al lado de la misma el archivo imagenes.rc el cual contiene:

```
1 RCDATA MiBoton.jpg
```

Se pueden meter en un archivo de recursos tantas imagenes como se desee, así como otros tipos de archivo (sonidos, animaciones flash, etc.). Las siguientes líneas serían:

```
2 RCDATA imagen2.jpg
3 RCDATA imagen3.jpg
...
```

Ahora abrimos una ventana de símbolo del sistema dentro del mismo directorio donde este la imagen y compilamos el recurso:

```
c:\imagenes\brc32 -r -v imagenes.rc
```

Lo cual nos creará el archivo imagenes.res.

Ahora para utilizar el recurso dentro de nuestro programa hay que añadir debajo de implementation (del formulario principal de la aplicación) la directiva {$R imagenes.res}:

```
implementation

{$R *.dfm}
{$R imagenes.res}
```

Esto lo que hace es unir todos los recursos de imagenes.res con nuestro ejecutable. Para cargar la imagen dentro de un objeto TImage hacemos lo siguiente:

```
procedure TFPrincipal.FormCreate( Sender: TObject );
var
  Recursos: TResourceStream;
  Imagen: TJPegImage;
begin
  Imagen := TJPegImage.Create;
  Recursos := TResourceStream.Create( hInstance, '#1', RT_RCDATA );
  Recursos.Seek( 0, soFromBeginning );
  Imagen.LoadFromStream( Recursos );
  Imagen1.Canvas.Draw( 0, 0, Imagen );
  Recursos.Free;
  Imagen.Free;
end;
```

Donde Imagen1 es un objeto TImage. Aunque pueda parecer algo molesto tiene las siguientes ventajas:

- Evita que un usuario cualquiera utilice nuestras imagenes (a menos claro que sepa utilizar un editor de recursos).

- Facilita la distribución de nuestro programa (va todo en el exe).

- Se puede añadir cualquier tipo de archivo o dato dentro del ejecutable.

- Ahorramos mucha memoria al cargar una sola vez el recurso.

Pruebas realizadas en Delphi 7.

Dibujar un gradiente en un formulario

Con un sencillo procedimiento podemos hacer que el fondo de nuestros formularios quede con un aspecto profesional con un degradado de color. Para ello escribimos en el evento OnPaint del formulario:

```
procedure TFormulario.FormPaint( Sender: TObject );
var
  wFila, wVertical: Word;
  iRojo: Integer;
begin
  iRojo := 200;
  wVertical := ( ClientHeight + 512 ) div 256;

  for wFila := 0 to 512 do
  begin
    with Canvas do
    begin
      Brush.Color := RGB( iRojo, 0, wFila );
      FillRect( Rect( 0, wFila * wVertical, ClientWidth, ( wFila + 1 )
* wVertical ) );
      Dec( iRojo );
    end;
  end;
end;
```

Lo que hace es crear un barrido vertical según el ancho y alto de nuestro formulario y va restando de la paleta RGB el componente rojo. Si queremos cambiar el color sólo hay que jugar con los componentes RGB hasta conseguir el efecto deseado.

Al maximizar o cambiar el ancho y alto de la ventana se quedará el degradado cortado. Para evitar esto le decimos en el evento OnResize que vuelva a pintar la ventana:

```
procedure TFormulario.FormResize( Sender: TObject );
begin
  Repaint;
end;
```

Este efecto suele utilizarse mucho instalaciones o en CD-ROM interactivos, catálogos, etc.

Pruebas realizadas en Delphi 7.

Trocear y unir archivos

Una de las utilidades más famosas que se han asociado a la descarga de archivos es el programa Hacha, el cual trocea archivos a un cierto tamaño para luego poder unirlos de nuevo.

El siguiente procedimiento parte un archivo a la longitud en bytes que le pasemos:

```
procedure TrocearArchivo( sArchivo: TFileName; iLongitudTrozo: Integer
);
var
  i: Word;
  FS, Stream: TFileStream;
  sArchivoPartido: String;
begin
  FS := TFileStream.Create( sArchivo, fmOpenRead or fmShareDenyWrite
);

  try
    for i := 1 to Trunc( FS.Size / iLongitudTrozo ) + 1 do
    begin
      sArchivoPartido := ChangeFileExt( sArchivo, '.' + FormatFloat(
'000', i ) );
      Stream := TFileStream.Create( sArchivoPartido, fmCreate or
fmShareExclusive );

      try
        if fs.Size - fs.Position < iLongitudTrozo then
          iLongitudTrozo := FS.Size - FS.Position;

        Stream.CopyFrom( FS, iLongitudTrozo );
      finally
        Stream.Free;
      end;
    end;
  finally
    FS.Free;
  end;
end;
```

Si por ejemplo le pasamos el archivo Documentos.zip creará los archivos:

Documentos.001
Documentos.002
....

Para volver a unirlo tenemos otro procedimiento donde le pasamos el primer trozo y el nombre del archivo original:

```
procedure UnirArchivo( sTrozo, sArchivoOriginal: TFileName );
var
  i: integer;
```

```
  FS, Stream: TFileStream;
begin
  i := 1;
  FS := TFileStream.Create( sArchivoOriginal, fmCreate or
fmShareExclusive );

  try
    while FileExists( sTrozo ) do
    begin
      Stream := TFileStream.Create( sTrozo, fmOpenRead or
fmShareDenyWrite );

      try
        FS.CopyFrom( Stream, 0 );
      finally
        Stream.Free;
      end;

      Inc(i);
      sTrozo := ChangeFileExt( sTrozo, '.' + FormatFloat( '000', i )
);
    end;
  finally
    FS.Free;
  end;
end;
```

Una ampliación interesante a estos procedimientos sería meter el nombre del archivo original en el primer o último trozo, así como un hash (MD4, MD5, SHA, etc.) para saber si algún trozo está defectuoso.

Pruebas realizadas en Delphi 7.

Mover componentes en tiempo de ejecución

Para darle un toque profesional a un programa no esta mal añadir un editor que permita al usuario personalizar sus formularios, informes o listados. Por ejemplo en un programa de facturación sería interesante que el usuario pudiera personalizar el formato de su factura.

Antes de nada hay que crear en la sección private del formulario tres variables encargadas de guardar las coordenadas del componente que se esta moviendo así como una variable booleana que nos dice si en este momento se esta moviendo un componente:

```
  private
    { Private declarations }
    iComponenteX, iComponenteY: Integer;
    bMoviendo: Boolean;
```

Creamos dentro de type la definición de un control movible:

```
type
  TMovible = class( TControl );
```

Los siguientes procedimientos hay que asignarlos a un componente para que puedan ser movidos por todo el formulario al ejecutar el programa:

```
procedure TFPrincipal.ControlMouseDown( Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer );
begin
  iComponenteX := X;
  iComponenteY := Y;
  bMoviendo := True;
  TMovible( Sender ).MouseCapture := True;
end;

procedure TFPrincipal.ControlMouseMove( Sender: TObject; Shift:
TShiftState; X, Y: Integer );
begin
  if bMoviendo then
    with Sender as TControl do
    begin
      Left := X - iComponenteX + Left;
      Top  := Y - iComponenteY + Top;
    end;
end;

procedure TFPrincipal.ControlMouseUp( Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer );
begin
  if bMoviendo then
  begin
    bMoviendo := False;
    TMovible( Sender ).MouseCapture := False;
  end;
end;
```

Por ejemplo, si queremos mover una etiqueta por el formulario habría que asignar los eventos:

```
Label1.OnMouseDown := ControlMouseDown;
Label1.OnMouseUp := ControlMouseUp;
Label1.OnMouseMove := ControlMouseMove;
```

Con esto ya podemos crear nuestro propio editor de formularios sin tener que utilizar las propiedades DragKing y DragMode de los formularios que resultan algo engorrosas.

Pruebas realizadas en Delphi 7.

Trabajando con arrays dinámicos

Una de las ventajas de los arrays dinámicos respecto a los estáticos es que pueden modificar su tamaño en tiempo de ejecución y no es necesaria la gestión de memoria para los mismos, ya que se liberan automáticamente al terminar el procedimiento, función o clase donde están alojados. También son ideales para enviar un número indeterminado de parámetros a funciones o procedimientos.

Para crear una array dinámico lo que hacemos es declarar el array pero sin especificar su tamaño:

```
  public
    Clientes: array of string;
```

Antes de meter un elemento al array hay que especificar su tamaño con la función SetLength. En este ejemplo creamos tres elementos:

```
  SetLength( Clientes, 3 );
```

Y ahora introducimos los datos en el mismo:

```
  Clientes[0] := 'JUAN';
  Clientes[1] := 'CARLOS';
  Clientes[2] := 'MARIA';
```

A diferencia de los arrays estáticos el primer elemento es el cero y no el uno. Como he dicho antes no es necesario liberarlos de memoria, pero si aún así deseamos hacerlo sólo es necesario hacer lo siguiente:

```
  Clientes := nil;
```

con lo cual el array queda inicializado para que pueda volver a ser utilizado.

Pruebas realizadas en Delphi 7.

Clonar las propiedades de un control

Cuantas veces hemos deseado en tiempo de ejecución copiar las características de un control a otro según las acciones del usuario. Por ejemplo si tenemos nuestro propio editor de informes se podrían copiar las carácterísticas de las etiquetas seleccionas por el usuario al resto del formulario (font, color, width, etc.)

Para ello tenemos que añadir la unidad TypInfo:

```
uses
  Windows, Dialogs, ..., TypInfo;
```

A esta función que voy a mostrar hay que pasarle el control origen y el destino (la copia) así como que propiedades deseamos copiar:

```
function ClonarPropiedades( Origen, Destino: TObject;
                            Propiedades: array of string ): Boolean;
var
```

```
  i: Integer;
begin
  Result := True;
  try
    for i := Low( Propiedades ) to High( Propiedades ) do
    begin
      // ¿Existe la propiedad en el control origen?
      if not IsPublishedProp( Origen, Propiedades[I] ) then
        Continue;

      // ¿Existe la propiedad en el control destino?
      if not IsPublishedProp( Destino, Propiedades[I] ) then
        Continue;

      // ¿Son del mismo tipo las dos propiedades?
      if PropType( Origen, Propiedades[I]) <>
        PropType( Destino, Propiedades[I] ) then
        Continue;

      // Copiamos la propiedad según si es variable o método
      case PropType(Origen, Propiedades[i]) of
        tkClass:
          SetObjectProp( Destino, Propiedades[i],
            GetObjectProp( Origen, Propiedades[i] ) );

        tkMethod:
          SetMethodProp( Destino, Propiedades[I],
                         GetMethodProp( Origen, Propiedades[I] ) );
        else
          SetPropValue( Destino, Propiedades[i],
                        GetPropValue( Origen, Propiedades[i] ) );
      end;
    end;
  except
    Result := False;
  end;
end;
```

Para copiar las características principales de una etiqueta habría que llamar a la función de la siguiente manera:

```
ClonarPropiedades( Label1, Label2, ['Font', 'Color', 'Alignment',
                   'Width', 'Height', 'Layout'] );
```

También se pueden copiar eventos tales como OnClick, OnMouseDown, etc. permitiendo así abarcar muchos controles con un solo evento.

Pruebas realizadas en Delphi 7.

Aplicar antialiasing a una imagen

El algoritmo antialiasing se suele aplicar a una imagen para evitar los bordes dentados y los degradados bruscos de color. Lo que hace es suavizar toda la imagen.

En este caso el siguiente procedimiento toma un objeto TImage como primer parámetro y como segundo el porcentaje de antialiasing deseado, siendo normal no aplicar más del 10 o 20%:

```
procedure Antialiasing( Imagen: TImage; iPorcentaje: Integer );
type
  TRGBTripleArray = array[0..32767] of TRGBTriple;
  PRGBTripleArray = ^TRGBTripleArray;
var
  SL, SL2: PRGBTripleArray;
  l, m, p: Integer;
  R, G, B: TColor;
  R1, R2, G1, G2, B1, B2: Byte;
begin
  with Imagen.Canvas do
  begin
    Brush.Style  := bsClear;
    Pixels[1, 1] := Pixels[1, 1];

    for l := 0 to Imagen.Height - 1 do
    begin
      SL := Imagen.Picture.Bitmap.ScanLine[l];

      for p := 1 to Imagen.Width - 1 do
      begin
        R1 := SL[p].rgbtRed;
        G1 := SL[p].rgbtGreen;
        B1 := SL[p].rgbtBlue;

        if (p < 1) then
          m := Imagen.Width
        else
          m := p - 1;

        R2 := SL[m].rgbtRed;
        G2 := SL[m].rgbtGreen;
        B2 := SL[m].rgbtBlue;

        if ( R1 <> R2 ) or ( G1 <> G2 ) or ( B1 <> B2 ) then
        begin
          R := Round( R1 + ( R2 - R1 ) * 50 / ( iPorcentaje + 50 ) );
          G := Round( G1 + ( G2 - G1 ) * 50 / ( iPorcentaje + 50 ) );
          B := Round( B1 + ( B2 - B1 ) * 50 / ( iPorcentaje + 50 ) );
          SL[m].rgbtRed := R;
          SL[m].rgbtGreen := G;
          SL[m].rgbtBlue := B;
        end;

        if ( p > Imagen.Width - 2 ) then
          m := 0
        else
          m := p + 1;

        R2 := SL[m].rgbtRed;
        G2 := SL[m].rgbtGreen;
        B2 := SL[m].rgbtBlue;

        if ( R1 <> R2 ) or ( G1 <> G2 ) or ( B1 <> B2 ) then
        begin
          R := Round( R1 + ( R2 - R1 ) * 50 / ( iPorcentaje + 50 ) );
          G := Round( G1 + ( G2 - G1 ) * 50 / ( iPorcentaje + 50 ) );
          B := Round( B1 + ( B2 - B1 ) * 50 / ( iPorcentaje + 50 ) );
          SL[m].rgbtRed := R;
          SL[m].rgbtGreen := G;
```

```
          SL[m].rgbtBlue := B;
        end;

        if ( l < 1 ) then
          m := Imagen.Height - 1
        else
          m := l - 1;

        SL2 := Imagen.Picture.Bitmap.ScanLine[m];
        R2  := SL2[p].rgbtRed;
        G2  := SL2[p].rgbtGreen;
        B2  := SL2[p].rgbtBlue;

        if ( R1 <> R2 ) or ( G1 <> G2 ) or ( B1 <> B2 ) then
        begin
          R := Round( R1 + ( R2 - R1 ) * 50 / ( iPorcentaje + 50 ) );
          G := Round( G1 + ( G2 - G1 ) * 50 / ( iPorcentaje + 50 ) );
          B := Round( B1 + ( B2 - B1 ) * 50 / ( iPorcentaje + 50 ) );
          SL2[p].rgbtRed := R;
          SL2[p].rgbtGreen := G;
          SL2[p].rgbtBlue := B;
        end;

        if ( l > Imagen.Height - 2 ) then
          m := 0
        else
          m := l + 1;

        SL2 := Imagen.Picture.Bitmap.ScanLine[m];
        R2  := SL2[p].rgbtRed;
        G2  := SL2[p].rgbtGreen;
        B2  := SL2[p].rgbtBlue;

        if ( R1 <> R2 ) or ( G1 <> G2 ) or ( B1 <> B2 ) then
        begin
          R := Round( R1 + ( R2 - R1 ) * 50 / ( iPorcentaje + 50 ) );
          G := Round( G1 + ( G2 - G1 ) * 50 / ( iPorcentaje + 50 ) );
          B := Round( B1 + ( B2 - B1 ) * 50 / ( iPorcentaje + 50 ) );
          SL2[p].rgbtRed := R;
          SL2[p].rgbtGreen := G;
          SL2[p].rgbtBlue := B;
        end;
      end;
    end;
  end;
end;
```

Suponiendo que tengamos en un formulario un objeto TImage llamado Image1 llamaríamos a este procedimiento de la siguiente manera:

```
  Antialiasing( Image1, 10 );
```

Dibujar varias columnas en un ComboBox

Vamos a ver un ejemplo de dibujar tres columnas al desplegar un ComboBox. El truco está en guardar el valor de las tres columnas en el mismo item pero separado por un punto y coma.

Después implementamos nuestra propia función de dibujado de columnas para que muestre las tres columnas. Para ello metemos en el evento OnDrawItem del ComboBox:

```
procedure TFormulario.ComboBoxDrawItem( Control: TWinControl; Index:
Integer;
                                        Rect: TRect; State:
TOwnerDrawState );
var
  sValor, sTodo: string;
  i, iPos: Integer;
  rc: TRect;
  AnchoColumna: array[0..3] of Integer;
begin
  ComboBox.Canvas.Brush.Style := bsSolid;
  ComboBox.Canvas.FillRect( Rect );

  // Las columnas deben ir separadas por un ;
  sTodo := ComboBox.Items[Index];

  // Establecemos el ancho de las columnas
  AnchoColumna[0] := 0;
  AnchoColumna[1] := 100;  // Ancho de la columna 1
  AnchoColumna[2] := 200;  // Ancho de la columna 2
  AnchoColumna[3] := 300;  // Ancho de la columna 3

  // Leemos el texto de la primera columna
  iPos := Pos( ';', sTodo );
  sValor := Copy( sTodo, 1, iPos - 1 );

  for i := 0 to 3 do
  begin
    // Dibujamos la primera columna
    rc.Left   := Rect.Left + AnchoColumna[i] + 2;
    rc.Right  := Rect.Left + AnchoColumna[i+1] - 2;
    rc.Top    := Rect.Top;
    rc.Bottom := Rect.Bottom;

    // Escribimos el texto
    Combobox.Canvas.TextRect( rc, rc.Left, rc.Top, sValor );

    // Dibujamos las líneas que separan las columnas
    if i < 3 then
    begin
      Combobox.Canvas.MoveTo( rc.Right, rc.Top );
      Combobox.Canvas.LineTo( rc.Right, rc.Bottom );
    end;

    // Leemos el texto de la segunda columna
    sTodo := Copy( sTodo, iPos + 1, Length( sTodo ) - iPos );
    iPos := Pos( ';', sTodo );
    sValor := Copy( sTodo, 1, iPos - 1 );
  end;
end;
```

Modificando el bucle y el array de enteros AnchoColumna podemos crear el número de columnas que queramos. Ahora sólo hay que meter los items en el ComboBox separados por punto y coma:

```
 with Combobox.Items do
 begin
   Add( 'JOSE;SANCHEZ;GARCIA;' );
   Add( 'MARIA;PEREZ;GOMEZ;' );
   Add( 'ANDRES;MARTINEZ;RUIZ;' );
 end;
```

Por último hay que decirle al ComboBox que la rutina de pintar los items corre por nuestra cuenta:

```
procedure TFormulario.FormCreate(Sender: TObject);
begin
  // Le decimos al ComboBox que lo vamos a pintar nosotros
  Combobox.Style := csOwnerDrawFixed;
end;
```

Pruebas realizadas en Delphi 7.

Conversiones entre unidades de medida

Delphi incorpora una librería interesante encargada de realizar conversiones entre unidades de tiempo, volumen, distancia, etc. Para ello hay que añadir las unidades ConvUtils y StdConvs:

```
uses
  Windows, Messages, ..., ConvUtils, StdConvs;
```

La función encargada de realizar conversiones es la siguiente:

Convert(ValorAConvertir: Double; DesdeUnidad, HastaUnidad: TConvType): Double;

Veamos algunos ejemplos mostrando el resultado en un campo Memo.
Para convertir de millas a kilómetros:

```
var dMillas, dKilometros: Double;
begin
  dMillas := 15;
  dKilometros := Convert( dMillas, duMiles, duKilometers );
  Memo.Lines.Add( Format( '%8.4f Millas = %8.4f Kilometros', [dMillas,
dKilometros] ) );
```

Esta otra convierte de pulgadas de área a centímetros de area:

```
var dPulgadas, dCentimetros: Double;
begin
  dPulgadas := 21;
  dCentimetros := Convert( dPulgadas, auSquareInches,
auSquareCentimeters );
  Memo.Lines.Add( Format( '%8.4f Pulgadas de área = %8.4f Centímetros
de área', [dPulgadas, dCentimetros] ) );
```

Y si queremos convertir libras en kilogramos:

```
var dLibras, dKilos: Double;
begin
  dLibras := 60;
  dKilos := Convert( dLibras, muPounds, muKilograms );
  Memo.Lines.Add( Format( '%8.4f Libras = %8.4f Kilos', [dLibras,
dKilos] ) );
```

También podemos convertir unidades de temperatura:

```
var dFahrenheit, dCelsius: Double;
begin
  dFahrenheit := 84;
  dCelsius := Convert( dFahrenheit, tuFahrenheit, tuCelsius );
  Memo.Lines.Add( Format( '%8.4fº Fahrenheit = %8.4fº Celsius',
[dFahrenheit, dCelsius] ) );
```

Así como conversión entre unidades de volumen:

```
var dMetrosCubicos, dLitros: Double;
begin
  dMetrosCubicos := 43;
  dLitros := Convert( dMetrosCubicos, vuCubicMeters, vuLiters );
  Memo.Lines.Add( Format( '%8.4f Metros cúbicos = %8.4fº Litros',
[dMetrosCubicos, dLitros] ) );
```

Ahora vamos a ver todos los tipos de conversión según las unidades de medida.
Para convertir entre unidades de área tenemos:

```
  auSquareMillimeters
  auSquareCentimeters
  auSquareDecimeters
  auSquareMeters
  auSquareDecameters
  auSquareHectometers
  auSquareKilometers
  auSquareInches
  auSquareFeet
  auSquareYards
  auSquareMiles
  auAcres
  auCentares
  auAres
  auHectares
  auSquareRods
```

Convertir entre unidades de distancia:

```
  duMicromicrons
  duAngstroms
  duMillimicrons
  duMicrons
  duMillimeters
  duCentimeters
  duDecimeters
  duMeters
  duDecameters
  duHectometers
```

```
duKilometers
duMegameters
duGigameters
duInches
duFeet
duYards
duMiles
duNauticalMiles
duAstronomicalUnits
duLightYears
duParsecs
duCubits
duFathoms
duFurlongs
duHands
duPaces
duRods
duChains
duLinks
duPicas
duPoints
```

Convertir entre unidades de masa:

```
muNanograms
muMicrograms
muMilligrams
muCentigrams
muDecigrams
muGrams
muDecagrams
muHectograms
muKilograms
muMetricTons
muDrams
muGrains
muLongTons
muTons
muOunces
muPounds
muStones
```

Convertir entre unidades de temperatura:

```
tuCelsius
tuKelvin
tuFahrenheit
tuRankine
tuReamur
```

Convertir entre unidades de tiempo:

```
tuMilliSeconds
tuSeconds
tuMinutes
tuHours
tuDays
tuWeeks
tuFortnights
tuMonths
```

```
tuYears
tuDecades
tuCenturies
tuMillennia
tuDateTime
tuJulianDate
tuModifiedJulianDate
```

Convertir entre unidades de volumen:

```
vuCubicMillimeters
vuCubicCentimeters
vuCubicDecimeters
vuCubicMeters
vuCubicDecameters
vuCubicHectometers
vuCubicKilometers
vuCubicInches
vuCubicFeet
vuCubicYards
vuCubicMiles
vuMilliLiters
vuCentiLiters
vuDeciLiters
vuLiters
vuDecaLiters
vuHectoLiters
vuKiloLiters
vuAcreFeet
vuAcreInches
vuCords
vuCordFeet
vuDecisteres
vuSteres
vuDecasteres
vuFluidGallons
vuFluidQuarts
vuFluidPints
vuFluidCups
vuFluidGills
vuFluidOunces
vuFluidTablespoons
vuFluidTeaspoons
vuDryGallons
vuDryQuarts
vuDryPints
vuDryPecks
vuDryBuckets
vuDryBushels
vuUKGallons
vuUKPottles
vuUKQuarts
vuUKPints
vuUKGills
vuUKOunces
vuUKPecks
vuUKBuckets
vuUKBushels
```

Pruebas realizadas en Delphi 7.

Tipos de puntero

Pointer:

- Es un tipo general de puntero hacia cualquier objeto o variable en memoria.
- Al no ser de ningún tipo suele ser bastante peligroso si provoca desbordamientos de memoria.

PAnsiChar:

- Es un tipo de puntero hacia un valor AnsiChar.
- También puede ser utilizado para apuntar a caracteres dentro de una cadena AnsiString.
- Al igual que otros punteros permite la aritmética, es decir, los procedimientos Inc y Dec pueden utilizarse para mover el puntero en memoria.

PAnsiString:

- Apunta hacia una cadena AnsiString.
- Debido a que AnsiString ya es un puntero hacia si misma, el puntero PAnsiString no suele utilizarse mucho.

PChar:

- Es un tipo de puntero hacia un valor Char.
- Puede ser utilizado para apuntar a caracteres dentro de una cadena string.
- Permite aritmética de punteros mediante los procedimientos Inc y Dec.
- Suele utilizarse mucho para procesar cadenas de caracteres terminadas en cero, tal como las utilizadas en el lenguaje C/C++.
- Los caracteres Char son idénticos a los de las variables AnsiChar, siendo de 8 bits de tamaño.

PCurrency:

- Apunta hacia un valor Currency.
- Permite aritmética de punteros mediante los procedimientos Inc y Dec.

PDateTime:

- Apunta hacia un valor TDateTime.
- Permite aritmética de punteros mediante los procedimientos Inc y Dec.

PExtended:

- Apunta hacia un valor Extended.
- Permite aritmética de punteros mediante los procedimientos Inc y Dec.

PInt64:

- Apunta hacia un valor Int64.
- Permite aritmética de punteros mediante los procedimientos Inc y Dec.

PShortString:

- Apunta hacia una cadena ShortString.
- Debido a que las variables ShortString difieren de las variables string, para apuntar a una variable ShortString es necesario utilizar la función Addr.

PString:

- Apunta hacia una cadena String.
- Al ser la cadena String un puntero en si misma no suele utilizarse mucho este puntero.

PVariant:

- Apunta hacia un valor Variant.
- Al ser Variant un tipo genérico y variable hay que extremar la precaución en el manejo de este puntero.

PWideChar:

- Apunta hacia un valor WideChar.
- Puede ser utilizado para apuntar a caracteres dentro de una cadena WideString.
- Permite aritmética de punteros mediante los procedimientos Inc y Dec.

PWideString:

- Apunta hacia una cadena WideString.
- Al ser ya cadena WideString un puntero en si misma no suele utilizarse mucho.

Pruebas realizadas en Delphi 7.

Dando formato a los números reales

La unidad SysUtils dispone del tipo TFloatFormat siguiente:

```
type TFloatFormat = (ffGeneral, ffExponent, ffFixed, ffNumber,
ffCurrency);
```

Este tipo es utilizado por las funciones CurrToStrF, FloatToStrF y FloatToText para dar formato a los números reales que pasen a formato texto. Como hemos visto anteriormente, los posibles valores de TFloatFormat son:

ffGeneral

Define el formato general de un número real acercando el valor resultante tanto como sea posible. Quita los ceros que se arrastran y la coma cuando sea necesario. No muestra ningún separador de miles y utiliza el formato exponencial cuando la mantisa es demasiado grande para el valor especificado según el formato. El formato de la coma es determinado por la variable DecimalSeparator.

Veamos un ejemplo mostrando el resultado en un campo Memo:

```
var rCantidad: Extended; // Número real para hacer pruebas
begin
  rCantidad := 1234.56;

  Memo.Lines.Add( 'General 4,0 = ' + FloatToStrF( rCantidad,
ffGeneral, 4, 0 ) );
  Memo.Lines.Add( 'General 6,0 = ' + FloatToStrF( rCantidad,
ffGeneral, 6, 0 ) );
  Memo.Lines.Add( 'General 6,2 = ' + FloatToStrF( rCantidad,
ffGeneral, 6, 2 ) );
  Memo.Lines.Add( 'General 3,2 = ' + FloatToStrF( rCantidad,
ffGeneral, 3, 2 ) );
```

El resultado que nos muestra es el siguiente:

```
  General 4,0 = 1235
  General 6,0 = 1234,56
  General 6,2 = 1234,56
  General 3,2 = 1,23E03
```

Como vemos la función FloatToStrF toma los siguientes parámetros:

function FloatToStrF(Value: Extended; Format: TFloatFormat; Precision, Digits: Integer): string; overload;

Value es el número real que vamos a pasar a texto.
Format es el tipo de formato en coma flotante que vamos a utilizar (en este caso ffGeneral).
Precision es el número máximo de dígitos enteros que soporta el formato.
Digits es el número máximo de decimales que soporta el formato.

En el caso anterior cuando forzamos a mostrar menos dígitos de precisión de lo que el número real tiene lo que hace la función es recortar o bien decimales o si la cantidad entera es superior al formato lo pasa al formato exponencial.

Veamos el resto de formatos:

ffExponent

Muestra el número real en formato científico cuyo exponente viene

representado con la letra E con base 10. Por ejemplo E+15 significa 1015. El carácter de la coma es representado por la variable DecimalSeparator.

Aquí vemos como queda el mismo número en distintos formatos exponenciales:

```
  Memo.Lines.Add( 'Exponencial 4,0 = ' + FloatToStrF( rCantidad,
ffExponent, 4, 0 ) );
  Memo.Lines.Add( 'Exponencial 6,0 = ' + FloatToStrF( rCantidad,
ffExponent, 6, 0 ) );
  Memo.Lines.Add( 'Exponencial 6,2 = ' + FloatToStrF( rCantidad,
ffExponent, 6, 2 ) );
  Memo.Lines.Add( 'Exponencial 3,2 = ' + FloatToStrF( rCantidad,
ffExponent, 3, 2 ) );
```

cuyo resultado es:

```
  Exponencial 4,0 = 1,235E+3
  Exponencial 6,0 = 1,23456E+3
  Exponencial 6,2 = 1,23456E+03
  Exponencial 3,2 = 1,23E+03
```

El formato ffFixed

Este formato no utiliza ningún separador de unidades de millar. Al igual que los formatos anteriores si la precisión del número real es superior al formato entonces muestra el resultado en notación científica. Quedaría de la siguiente manera:

```
  Memo.Lines.Add( 'Fijo 4,0 = ' + FloatToStrF( rCantidad, ffFixed, 4,
0 ) );
  Memo.Lines.Add( 'Fijo 6,0 = ' + FloatToStrF( rCantidad, ffFixed, 6,
0 ) );
  Memo.Lines.Add( 'Fijo 6,2 = ' + FloatToStrF( rCantidad, ffFixed, 6,
2 ) );
  Memo.Lines.Add( 'Fijo 3,2 = ' + FloatToStrF( rCantidad, ffFixed, 3,
2 ) );
```

dando los valores:

```
  Fijo 4,0 = 1235
  Fijo 6,0 = 1235
  Fijo 6,2 = 1234,56
  Fijo 3,2 = 1,23E03
```

El formato ffNumber

Es igual al formato ffFixed salvo que también incluye el separador de unidades de millar, el cual viene representado por la variable ThousandSeparator. En nuestro ejemplo:

```
  Memo.Lines.Add( 'Número 4,0 = ' + FloatToStrF( rCantidad, ffNumber,
4, 0 ) );
  Memo.Lines.Add( 'Número 6,0 = ' + FloatToStrF( rCantidad, ffNumber,
6, 0 ) );
  Memo.Lines.Add( 'Número 6,2 = ' + FloatToStrF( rCantidad, ffNumber,
```

```
6, 2 ) );
  Memo.Lines.Add( 'Número 3,2 = ' + FloatToStrF( rCantidad, ffNumber,
3, 2 ) );
```

mostraría:

```
  Número 4,0 = 1.235
  Número 6,0 = 1.235
  Número 6,2 = 1.234,56
  Número 3,2 = 1,23E03
```

El formato ffCurrency

Es similar al formato ffNumber pero con un símbolo de secuencia agregado, según se haya definido en la variable CurrencyString. Este formato también esta influenciado por las variables CurrencyFloat y NegCurrFloat. Sigamos el ejemplo:

```
  Memo.Lines.Add( 'Moneda 4,0 = ' + FloatToStrF( rCantidad,
ffCurrency, 4, 0 ) );
  Memo.Lines.Add( 'Moneda 6,0 = ' + FloatToStrF( rCantidad,
ffCurrency, 6, 0 ) );
  Memo.Lines.Add( 'Moneda 6,2 = ' + FloatToStrF( rCantidad,
ffCurrency, 6, 2 ) );
  Memo.Lines.Add( 'Moneda 3,2 = ' + FloatToStrF( rCantidad,
ffCurrency, 3, 2 ) );
```

Da como resultado:

```
  Moneda 4,0 = 1.235 €
  Moneda 6,0 = 1.235 €
  Moneda 6,2 = 1.234,56 €
  Moneda 3,2 = 1,23E03
```

según las variables mencionadas anteriormente que recogen el formato moneda por defecto configurado en Windows.

Pruebas realizadas en Delphi 7.

Conversiones entre tipos numéricos

Hasta ahora hemos visto como pasar cualquier tipo de variable a string o al revés. Ahora vamos a ver funciones para convertir valores de un tipo numérico a otro.

PASAR NÚMEROS REALES A NÚMEROS ENTEROS

function Trunc(X: Extended): Int64;

Esta función convierte un tipo Extended al tipo entero Int64 truncando los decimales (sin redondeos). Por ejemplo:

```
Trunc( 1234.5678 ) devuelve 1234
```

function Round(X: Extended): Int64;

Esta función convierte un tipo Extended al tipo entero Int64 pero redondeando la parte entera según la parte decimal. Por ejemplo:

```
Round( 1234.5678 ) devuelve 1235
Round( 1234.4678 ) devuelve 1234
```

function Ceil(const X: Extended): Integer;

Convierte un valor Extended en Integer redondeando al número entero que este mas próximo hacia arriba. Por ejemplo:

```
Ceil( 1234.5678 ) devuelve 1235
Ceil( 1234.4678 ) devuelve 1235
Ceil( 1235.5678 ) devuelve 1236
Ceil( 1235.4678 ) devuelve 1236
```

function Floor(const X: Extended): Integer;

Convierte un valor Extended en Integer redondeando al número entero que este más próximo hacia abajo. Por ejemplo:

```
Floor( 1234.5678 ) devuelve 1234
Floor( 1234.4678 ) devuelve 1234
Floor( 1235.5678 ) devuelve 1235
Floor( 1235.4678 ) devuelve 1235
```

También tenemos una función un poco rara para extraer el exponente y la mantisa de un número real Extended:

procedure FloatToDecimal(var DecVal: TFloatRec; const Value; ValueType: TFloatValue; Precision, Decimals: Integer);

Convierte un número Extented o Currency a formato decimal guardándolo en la siguiente estructura de datos:

```
type TFloatRec = record
  Exponent: Smallint;
  Negative: Boolean;
  Digits: array[0..20] of Char;
end;
```

Por ejemplo:

```
var
  e: Extended;
  RegistroFloat: TFloatRec;
begin
  e := 1234.5678;
  FloatToDecimal( RegistroFloat, e, fvExtended, 10, 4 );
end;
```

Al ejecutarlo da los siguientes valores:

```
RegistroFloat.Digits = 12345678
RegistroFloat.Negative = False
RegistroFloat.Exponent = 4
```

TRUNCANDO NÚMEROS REALES

function Int(X: Extended): Extended;

Aunque esta función devuelve sólo la parte entera de un valor Extended, el valor devuelto sigue siendo Extended. El resultado es similar a la función Trunc (sin redondeos). Por ejemplo:

```
Int( 1234.5678 ) devuelve 1234
Int( 1234.4678 ) devuelve 1234
```

function Frac(X: Extended): Extended;

Devuelve la parte fraccionaria de un valor Extended. Por ejemplo:

```
Frac( 1234.4678 ) devuelve 0.5678
```

CONVIRTIENDO VALORES EXTENDED Y CURRENCY

Como vimos anteriormente, el tipo Currency no es un auténtico formato en punto flotante sino un entero de punto fijo con 4 decimales como máximo, es decir, los valores Currency se almacenan como un entero multiplicado por 10000. Veamos de que funciones disponemos para convertir de un tipo a otro:

function FloatToCurr(const Value: Extended): Currency;

Esta función convierte de tipo Extended a Currency. Por ejemplo:

```
FloatToCurr( 1234.5678 ) devuelve 1234.5678 (de tipo Currency)
FloatToCurr( 123.456789 ) devuelve 123.4568 (de tipo Currency y
perdemos 2 decimales)
```

function TryFloatToCurr(const Value: Extended; out AResult: Currency): Boolean;

Esta función es similar a FloatToCurr pero sólo comprueba si se puede convertir a Currency. El procedimiento sería:

```
var
  c: Currency;
begin
  if TryFloatToCurr( 123.456789, c ) then
    ShowMessage( CurrToStr( c ) );
end;
```

CONVIRTIENDO VALORES ENTEROS A VALORES REALES

Se puede convertir un valor entero a real convirtiendo la variable directamente:

```
var
  i: Integer;
  r: Real;
begin
  i := 1234;
  r := i;                          // realiza el casting
automáticamente
  ShowMessage( FloatToStr( r ) ); // muestra 1234
end;
```

Esto no ocurre en todos los casos pero es cuestión de probar (aunque es recomendable utilizar los tipos Variant para esa labor).

Pruebas realizadas en Delphi 7.

Creando un cliente de chat IRC con Indy (I)

Aunque MSN es el rey de la comunicación instantánea aún siguen muy vivos los servidores de chat IRC tales como irc-hispano.

IRC define un protocolo de comunicaciones entre clientes y servidores permitiendo incluso el envío de archivos por conexión directa. De todos es conocido el potente programa para Windows llamado MIRC creado hace años y que aún sigue siendo el cliente más utilizado para el IRC debido a sus múltiples extensiones.

En nuestro caso vamos a ver como utilizar el componente de la clase TIdIRC el cual esta situado en la pestaña de componentes Indy.

CONECTANDO CON EL SERVIDOR

Antes de comenzar a chatear con el servidor hay que establecer una conexión con el mismo utilizando un apodo (nick) y una contraseña en el caso de que sea necesaria (la mayoría de los servidores IRC son públicos).

Supongamos que tenemos en el formulario principal de nuestra aplicación un componente IdIRC que vamos a llamar IRC. Para conectar con el servidor hay que hacer lo siguiente:

```
IRC.Nick := 'juanito33487';
IRC.AltNick := 'juanito33488';
IRC.Username := 'juanito33487';
IRC.RealName := 'juan';
IRC.Password := '';
IRC.Host := 'irc.irc-hispano.org';

try
  IRC.Connect;
except
```

```
  Application.MessageBox( 'No se ha podido conectar con el servidor.',
                          'Error de conexión', MB_ICONSTOP );
end;
```

Estos son los parámetros para conectar:

```
Nick     -> apodo por el que nos conocerán los demás usuarios
AltNick  -> si el nick que hemos utilizado está ocupado por otro
usuario cogerá este otro nick
Username -> Nombre del usuario para el servidor (da lo mismo si no
estamos registrados)
Realname -> Nombre real del usuario
Password -> Clave de acceso para usuarios registrados
Host     -> Dirección IP del servidor
```

Cualquier persona puede conectarse a un servidor de chat dando sólo el nombre del usuario sin contraseña. Pero si queremos que nadie utilice nuestro nick nos podemos registrar gratuitamente (en la mayoría de los casos) en el servidor con un usuario y password obteniendo además algunas características adicionales como entrar a salas de chat restringidas o tener nuestro propio servidor de mensajes y correo online 24 horas al día.

Una vez conectados al servidor toca entrar en una sala de chat. Pero, ¿de que salas de chat dispone el servidor? Para ello utilizamos el comando:

```
IRC.Raw( 'LIST' );
```

El método Raw permite mandar cualquier tipo de comando al servidor. Se suele utilizar cuando el componente IRC no contiene métodos para realizar tareas específicas. Los servidores IRC trabajan enviando y recibiendo mensajes de texto continuamente. Para los que les interese saber como funciona por dentro el protocolo IRC disponen de este documento RFC:

```
http://www.rfc-es.org/rfc/rfc1459-es.txt
```

Pese a los comandos del IRC estándar algunos servidores disponen de comandos propios para tareas específicas. Para esos casos utilizaremos el método Raw.

Hay que tener mucho cuidado cuando se llama al comando LIST ya que hay servidores como los irc-hispano que disponen de miles de canales lo cual puede hacer que el tiempo en sacar el listado llegue a ser desesperante. Por ello sería deseable que tanto la función de conectar a un servidor IRC como la de listar los canales estuviera dentro de un hilo de ejecución, ya que hasta que no conecta da la sensación de que nuestro programa esta colgado.

Después de ejecutar dicho comando tenemos que programar el evento OnList para recoger la información del listado de canales. En este caso para volcar los canales en un componente ListBox llamado Canales hacemos lo siguiente:

```
procedure TFormulario.IRCList( Sender: TObject; AChans: TStringList;
                               APosition: Integer; ALast: Boolean );
begin
```

```
  Canales.Items := AChans;
end;
```

Con esto ya tenemos la lista de canales que dispone el servidor así como el número de usuarios que hay conectados por canal y su descripción. Por ejemplo:

```
#amistades 20 Busca nuevos amig@s
#amor 50 Conoce a tu media naranja
#sexo 80 No te cortes un pelo
#musica 34 Comparte la pasión por tus artistas preferidos
....
```

En un servidor de IRC los canales comienzan con el carácter # seguido de su nombre sin espacios.

En el próximo artículo veremos como entrar a los canales de chat y hablar con otros usuarios.

Pruebas realizadas en Delphi 7.

Creando un cliente de chat IRC con Indy (II)

Una vez establecida la conexión con el servidor y sabiendo de que salas de chat dispone, vamos a entrar a un canal a chatear.

ENTRANDO A UN CANAL

Para entrar a un canal se utiliza el método Join:

```
IRC.Join( '#amistades' );
```

Esto viene a ser lo mismo que hacer:

```
IRC.Raw( 'join #amistades' );
```

Podemos entrar a tantos canales como deseemos pero para hacer un buen cliente de chat hay que crear un formulario por canal, ya que el protocolo IRC nos permite chatear en múltiples canales simultaneamente.

Una vez se ha enviado el comando JOIN el servidor nos devuelve la lista de usuarios que hay en ese canal. Para recoger dicha lista hay que utilizar el

evento OnNames del componente IdIRC, donde volcaremos el contenido de la misma en un ListBox que vamos a llamar Usuarios:

```
procedure TFormulario.IRCNames( Sender: TObject; AUsers: TIdIRCUsers;
AChannel: TIdIRCChannel );
var i: Integer;
begin
  if Assigned( AUsers ) then
  begin
    Usuarios.Clear;

    for i := 0 to AUsers.Count - 1 do
      Usuarios.Items.Add( AUsers.Items[i].Nick );
  end;
end;
```

Es conveniente que el ListBox este ordenado alfabéticamente activando la propiedad Sorted para no volvernos locos buscando los usuarios por su nombre.

LEYENDO LOS MENSAJES DEL SERVIDOR

Una vez conectados a un canal, el servidor nos mandará dos tipos de mensajes: lo que hablan todos los usuarios en el canal y lo que un usuario en concreto nos esta diciendo. ¿Cómo distinguimos cada cual? Pues para ello el evento OnMessaje nos dice el usuario y canal que nos manda el mensaje:

```
procedure TFormulario.IRCMessage( Sender: TObject; AUser: TIdIRCUser;
                                  AChannel: TIdIRCChannel; Content:
string );
begin
  if Assigned( AUser ) then
  begin
    if Assigned( AChannel ) then
      Canal.Lines.Add( AUser.Nick + '> ' + Content )
    else
      Privado.Lines.Add( AUser.Nick + '> ' + Content );
  end;
end;
```

Como se puede apreciar primero comprobamos si existe el usuario (por si acaso es un mensaje del sistema) y luego si tiene canal metemos el mensaje por el mismo (poniendo el nick delante como MIRC) y si no es asi lo mandamos a una conversación privada. En este caso he enviado el texto de la conversación a un componente RichEdit.

Sería interesante que nuestro cliente de chat dispusiera de un formulario por cada conversación privada. Por ello, sería recomendable que nuestro programa de chat utilizara ventanas MDI para permitir el control absoluto cada una de las salas en las que estamos así como cada una de las conversaciones privadas. En otro artículo escribiré como crear aplicaciones MDI.

Otra característica que lo haría más profesional sería dibujar el nick y el mensaje del usuario con colores distintos como vimos anteriormente en el

artículo dedicado al componente RichEdit. Y si además damos al usuario la posibilidad de seleccionar los colores de fondo, texto, nicks y tipo de fuente, el programa se aproximaría al famoso cliente MIRC.

ENVIANDO MENSAJES AL SERVIDOR

Se pueden enviar dos tipos de mensajes: a un canal donde estamos (para el público en general) o a un usuario en concreto. El método say del componente IdIRC cumple dicho cometido.

Si quiero enviar un mensaje al canal amistades:

```
IRC.Say( '#amistades', 'hola a todos' );
```

Y si es para un usuario con el que estamos chateando:

```
IRC.Say( '#maria', 'hola guapa, que tal?' );
```

Al fin y al cabo para el servidor todo son canales, tanto canales públicos como usuarios particulares. Tenemos que ser nosotros los que debemos procesar de donde viene el mensaje y a donde deseamos enviarlo. De ahí la necesidad de crear un formulario por canal y otro por conversación privada.

ABANDONANDO UN CANAL Y UN SERVIDOR

Para abandonar un canal se utiliza el método Part:

```
IRC.Part( '#amistades' );
```

Si se nos olvida hacer esto y seguimos abriendo canales al final gastaremos mucho ancho de banda, ya que el servidor nos manda todos los mensajes publicos de cada uno de los canales abiertos.

Si queremos desconectar del servidor se utiliza el comando:

```
IRC.Disconnect;
```

Esto cancelará la conexión con el servidor cerrando todos los mensajes públicos y privados del mismo. En nuestro programa deberemos cerrar todas las ventanas de conversación abiertas.

Todavía quedan algunos puntos importantes para terminar de crear un cliente de chat, tales como saber si un usuario entra o abandona el canal donde estamos.

Pruebas realizadas en Delphi 7.

Creando un cliente de chat IRC con Indy (III)

Después de haber visto la parte básica del componente IdIRC pasemos a ver algunas cosas que hay que controlar en un cliente de chat.

QUIEN SALE Y QUIEN ENTRA A UN CANAL

Mientras permanezcamos en un canal entrarán y saldrán usuarios constantemente, lo cual nos obliga a refrescar la lista de usuarios y notificar los cambios en la misma ventana del canal:

```
Ha entrado el usuario lola33
Ha salido el usuario carlos21
```

Para controlar los usuarios que entran a un canal utilizaremos el evento OnJoin:

```
procedure TFormulario.IRCJoin( Sender: TObject; AUser: TIdIRCUser;
AChannel: TIdIRCChannel );
begin
  if Assigned( AChannel ) and Assigned( AUser ) then
    if AChannel.Name = CanalActual.Text then
    begin
      Canal.Lines.Add( 'Ha entrado el usuario ' + AUser.Nick );

      if Usuarios.Items.IndexOf( AUser.Nick ) = -1 then
        Usuarios.Items.Add( AUser.Nick );
    end;
end;
```

Después de informar en el canal actual que ha aparecido un nuevo usuario también lo hemos dado de alta en la lista de usuarios, controlando que no se repitan, ya que a veces suele coincidir que se ejecuta el comando NAMES del servidor y a la vez entra un usuario nuevo.

Y cuando un usuario abandona el canal entonces utilizamos el evento OnPart:

```
procedure TFormulario.IRCPart( Sender: TObject; AUser: TIdIRCUser;
AChannel: TIdIRCChannel );
var iUsuario: Integer;
begin
  if Assigned( AChannel ) and Assigned( AUser ) then
    if AChannel.Name = CanalActual.Text then
    begin
      Canal.Lines.Add( 'Ha salido el usuario ' + AUser.Nick );
      iUsuario := Usuarios.Items.IndexOf( AUser.Nick );

      if iUsuario > -1 then
        Usuarios.Items.Delete( iUsuario );
    end;
end;
```

La única dificultad que hay que controlar es que si tenemos varias ventanas (una por canal) hay que llevar las notificaciones a la ventana correspondiente.

Los canales de chat tienen dos clases de usuarios: operadores y usuarios

normales. Los operadores son usuarios que tienen permisos especiales: pueden hacer operadores a usuarios normales o echar a un usuario de un canal si esta incumpliendo las normas del mismo.

Cuando un operador echa fuera a un usuario entonces nuestro componente IdIRC provoca el evento OnKick:

```
procedure TFormulario.IRCKick( Sender: TObject; AUser, AVictim:
TIdIRCUser; AChannel: TIdIRCChannel );
var iUsuario: Integer;
begin
  if Assigned( AChannel ) and Assigned( AUser ) then
    if AChannel.Name = CanalActual.Text then
    begin
      Canal.Lines.Add( 'El usuario ' + Auser.Nick + ' ha expulsado a
usuario ' + AVictim.Nick );
      iUsuario := Usuarios.Items.IndexOf( AUser.Nick );

      if iUsuario > -1 then
        Usuarios.Items.Delete( iUsuario );
    end;
end;
```

Este evento nos informa del nombre del operador, el de la victima y del canal de donde ha sido expulsado.

Y por último tenemos el problema de que un usuario puede cambiarse el apodo (nick) en cualquier momento. Para ello utilizaremos el evento OnNickChange para notificarlo:

```
procedure TFormulario.IRCNickChange( Sender: TObject; AUser:
TIdIRCUser; ANewNick: String );
var iUsuario: Integer;
begin
  Canal.Lines.Add( 'El usuario ' + AUser.Nick + ' ahora se llama ' +
ANewNick );
  iUsuario := Usuarios.Items.IndexOf( AUser.Nick );

  if iUsuario > -1 then
    Usuarios.Items[iUsuario] := ANewNick;
end;
```

Aparte de notificarlo le hemos cambiado el nombre en nuestra lista de usuarios.

INTERCEPTANDO LOS MENSAJES DEL SISTEMA

A pesar de todos los eventos de los que dispone el componente de la clase TIdIRC también podemos interceptar a pelo los mensajes del sistema a través del evento OnSystem, el cual nos dice el usuario, el código de comando y el contenido del mensaje del sistema. Por ejemplo estos son algunos códigos de comando del servidor:

```
353 -> Comienzo del comando NAMES
366 -> Fin del comando NAMES
```

```
376 -> Mensaje del día
etc.
```

Para más información hay que ver el documento RFC perteneciente al protocolo IRC que mencione en el artículo anterior. Por ejemplo, para averiguar los datos sobre un usuario en concreto hay que mandarle al servidor el comando:

WHOIS maria

Y el servidor devolverá lo siguiente:

```
319 - WHOIS - maria is on #amistades
312 - WHOIS - maria is using jupiter2.irc-hispano.org Servidor IRC de
LLeida Networks
318 - WHOIS - maria :End of /WHOIS list
```

Nos esta diciendo que maria está en el canal #amistades y que está conectada utilizando el servidor jupiter2. Si estuviera conectada a más canales también lo diría (así sabemos también sus otras aficiones).

¿Cómo nos comemos todo eso? Pues supongamos que si yo hago doble clic sobre un usario quiero que me devuelva información sobre el mismo en un campo Memo cuyo nombre es DatosUsuario. Lo primero sería procesar el evento OnDblClick en la lista de usuarios:

```
procedure TFormulario.UsuariosDblClick( Sender: TObject );
begin
  if Usuarios.ItemIndex > -1 then
    IRC.Raw( 'WHOIS ' + Usuarios.Items[Usuarios.ItemIndex] );
end;
```

Y ahora en el evento OnSystem del componente IdIRC esperamos a que nos llegue la información:

```
procedure TFormulario.IRCSystem( Sender: TObject; AUser: TIdIRCUser;
ACmdCode:  Integer; ACommand, AContent: string );
begin
  case ACmdCode of
    312, 318, 319: DatosUsuario.Lines.Add( AContent );
  end;
end;
```

Así, estudiando un poco el protocolo IRC y sabiendo algunos comandos podemos hacer más cosas de las que permite el componente IdIRC.

ENVIO Y RECEPCION DE ARCHIVOS POR ACCESO DIRECTO

El protocolo IRC permite establecer el envío y recepción de archivos con conexión punto a punto entre la IP del remitente y la IP del receptor. Esto ya no se suele utilizar hoy en día por las siguiente razones:

- Debido a que casi todas las redes locales están detrás de un router se hace necesario abrir y redireccionar puertos en el mismo, cosa que no todo el

mundo sabe hacer.

- En cuanto intentemos establecer conexión por puertos que no sean los estandar los cortafuegos saltarán como liebres pidiendo permiso para conectar. Los novatos siempre dirán NO por si acaso.

- Hay robots automáticos programados en MIRC mediante scripts que no paran de realizar ataques continuamente a conexiones DCC.

Por ello no voy a comentar aquí como utilizarlo (para ello tenemos rapidshare, megaupload y demás hierbas que superan con creces este tipo de conexiones y además son más seguros).

Con esto finalizamos lo más importante del componente IdIRC de la paleta de componentes Indy.

Pruebas realizadas en Delphi 7.

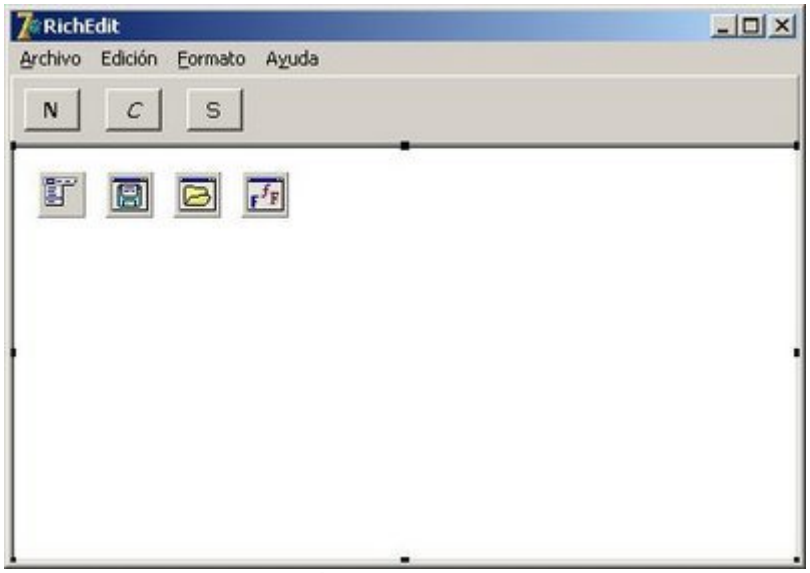
Creando un procesador de textos con RichEdit (I)

Para crear un procesador de textos vamos a utilizar el componente RichEdit que se encuentra en la pestaña Win32. Este componente tiene la particularidad de poder definir distintos estilos de texto al contrario de un componente Memo cuya fuente es estática para todo el documento.

El componente de la clase TRichEdit hereda de TCustomMemo añadiendo características tan interesantes como la de modificar el estilo de la fuente, colorear palabras o frases, etc.

Veamos como se podría crear un mini procesador de textos utilizando este componente. Como procesador de textos lo primero que debemos definir son las funciones para la creación, carga y grabación de documentos en disco.

La programación la voy a realizar sobre este formulario:



Las opciones del menú son:

```
Archivo -> Nuevo, Abrir, Guardar, Guardar como y Salir.
Edición -> Cortar, Copiar y Pegar.
Formato -> Fuente
Ayuda   -> Acerca de...
```

GUARDANDO EL TEXTO EN UN ARCHIVO

Lo primero que vamos a contemplar es la grabación en un archivo de texto. Para ello vamos a insertar en el formulario un componente de clase TSaveDialog que se encuentra en la pestaña Dialogs y lo vamos a llamar GuadarTexto.

Entonces al pulsar la opción Archivo -> Guardar como del ménu ejecutamos lo siguiente:

```
procedure TFormulario.GuardarComoClick( Sender: TObject );
begin
  if GuardarTexto.Execute then
  begin
    RichEdit.Lines.SaveToFile( GuardarTexto.FileName );
    sArchivo := GuardarTexto.FileName;
  end;
end;
```

La variable sArchivo se va a encargar de guardar la ruta y el nombre archivo que se guardó por primera vez, para que cuando seleccionemos la opción guardar no haya que volver a decirle de nuevo el nombre. Esta variable la vamos a declarar en la sección privada de nuestro formulario:

```
  private
    { Private declarations }
    sArchivo: String;
```

Ahora tenemos que hacer que si el usuario pulsa la opción Guardar se guarde el archivo sin preguntarnos el nombre si ya lo tiene:

```
procedure TFormulario.GuardarClick( Sender: TObject );
begin
  // ¿No tiene nombre?
  if sArchivo = '' then
    GuardarComoClick( Self )
  else
    RichEdit.Lines.SaveToFile( sArchivo );
end;
```

Si no tuviera nombre es que es un archivo nuevo, con lo cual lo desviamos a la opción Guardar como.

CARGADO EL TEXTO DESDE UN ARCHIVO

Para cargar el texto tenemos que añadir al formulario el componente TOpenDialog y lo llamamos CargarTexto. Y al pulsar en el menú la opción Archivo -> Abrir se ejecutaría:

```
procedure TFormulario.AbrirClick( Sender: TObject );
begin
  if CargarTexto.Execute then
  begin
    RichEdit.Lines.LoadFromFile( CargarTexto.FileName );
    sArchivo := CargarTexto.FileName;
  end;
end;
```

También guardamos en la variable sArchivo el nombre del archivo cargado para su posterior utilización en la opción Guardar.

CREANDO UN NUEVO TEXTO

En nuestro programa vamos a hacer que si el usuario selecciona la opción del menú Archivo -> Nuevo se elimine el texto del componente RichEdit. Lo que si hay que controlar es que si había un texto anterior le pregunte al usuario si desea guardarlo.

```
procedure TFormulario.NuevoClick( Sender: TObject );
begin
  // ¿Hay algo introducido?
  if RichEdit.Text <> '' then
    if Application.MessageBox( '¿Desea guardar el texto actual?',
'Atención',
                               MB_ICONQUESTION OR MB_YESNO ) = ID_YES
then
      GuardarClick( Self );

  RichEdit.Clear;
end;
```

CONTROLANDO LA EDICION DEL TEXTO

Otra de las cosas básicas que debe llevar todo editor de texto son las funciones de cortar, copiar y pegar. Para ello vamos a implemantar primero la opción del menú Edición -> Cortar:

```
procedure TFormulario.CortarClick( Sender: TObject );
begin
  RichEdit.CutToClipboard;
end;
```

Después hacemos la opción de Edición -> Copiar:

```
procedure TFormulario.CopiarClick( Sender: TObject );
begin
  RichEdit.CopyToClipboard;
end;
```

Y por último la opción de Edición -> Pegar:

```
procedure TFormulario.PegarClick( Sender: TObject );
begin
  RichEdit.PasteFromClipboard;
end;
```

CAMBIANDO EL ESTILO DEL TEXTO

Una vez tenemos implementada la parte básica del editor de texto vamos a darle la posibilidad al usuario de que pueda cambiar la fuente, el estilo, el color, etc.

Para que el usuario pueda elegir la fuente tenemos que introducir en el formulario el componente de la clase TFontDialog situado en la pestaña Dialogs. Le vamos a dar el nombre de ElegirFuente.

Ahora en la opción del menú Formato -> Fuente ejecutamos:

```
procedure TFormulario.FuenteClick( Sender: TObject );
begin
  if ElegirFuente.Execute then
    with RichEdit, ElegirFuente do
    begin
      SelAttributes.Name := Font.Name;
      SelAttributes.Size := Font.Size;
      SelAttributes.Color := Font.Color;
      SelAttributes.Pitch := Font.Pitch;
      SelAttributes.Style := Font.Style;
      SelAttributes.Height := Font.Height;
    end;
end;
```

¿Por qué no hemos hecho lo siguiente?

```
RichEdit.Font := ElegirFuente.Font;
```

Si hago esto me cambia la fuente de todo el texto, incluso la que he escrito anteriormente y a mi lo que me interesa es modificar la fuente de lo que se vaya a escribir a partir de ahora. Para ello se utilizan las propiedades de

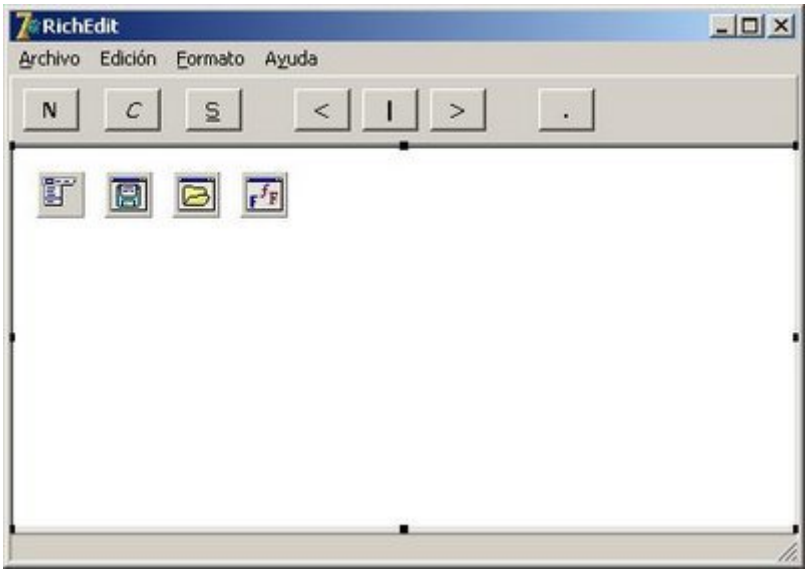
SelAttributes las cuales se encargan de establecer el estilo del texto seleccionado, y en el caso de que no haya texto seleccionado se aplica a donde esté el cursor.

En el próximo artículo seguiremos ampliando nuestro pequeño procesador de textos.

Pruebas realizadas en Delphi 7

Creando un procesador de textos con RichEdit (II)

Sigamos añadiendo características a nuestro pequeño editor de textos:



NEGRITA, CURSIVA Y SUBRAYADO

Vamos a definir la función de los típicos botones de negrita, cursiva y subrayado que llevan los procesadores de texto. Comencemos con el botón de negrita:

```
procedure TFormulario.BNegritaClick( Sender: TObject );
begin
 with RichEdit.SelAttributes do
   if not ( fsBold in Style ) then
     Style := Style + [fsBold]
   else
     Style := Style - [fsBold];

 RichEdit.SetFocus;
end;
```

Hay que tener en cuenta que si el usuario pulsa una vez negrita y el texto no estaba en negrita entonces se aplica dicho estilo, pero si ya lo estaba entonces hay que quitarlo.

Lo mismo sería para el botón de cursiva:

```
procedure TFormulario.BCursivaClick( Sender: TObject );
begin
 with RichEdit.SelAttributes do
   if not ( fsItalic in Style ) then
     Style := Style + [fsItalic]
   else
     Style := Style - [fsItalic];

 RichEdit.SetFocus;
end;
```

Y para el botón de subrayado:

```
procedure TFormulario.BSubrayadoClick( Sender: TObject );
begin
 with RichEdit.SelAttributes do
   if not ( fsUnderline in Style ) then
     Style := Style + [fsUnderline]
   else
     Style := Style - [fsUnderline];

 RichEdit.SetFocus;
end;
```

MOSTRANDO LA POSICION ACTUAL DEL CURSOR

Una de las cosas que se agradece en todo procesador de textos es que muestre en que fila y columna estoy situado. Para ello vamos a insertar en la parte inferior del formulario una barra de estado. El componente se llama StatusBar y se encuentra en la pestaña Win32. Le vamos a poner el nombre de Estado y ponemos a True su propiedad SimplePanel.

Lo que haremos a continuación es un procedimiento que muestre la posición actual del cursor dentro del RichEdit en la barra de estado:

```
procedure TFormulario.MostrarPosicion;
begin
 Estado.SimpleText := Format( 'Fila: %d Columna %d',
   [RichEdit.CaretPos.y+1, RichEdit.CaretPos.x+1] );
end;
```

Este procedimiento hay que llamarlo cuando creamos el formulario:

```
procedure TFormulario.FormCreate( Sender: TObject );
begin
 MostrarPosicion;
end;
```

y en el evento OnSelectionChange del RichEdit:

```
procedure TFormulario.RichEditSelectionChange( Sender: TObject );
begin
 MostrarPosicion;
end;
```

Al ejecutar el programa veremos que el resultado queda más profesional (ya podían los de Microsoft ponerle esto al cutre Bloc de Notas).

DANDO FORMATO A LOS PARRAFOS DEL DOCUMENTO

Otra característica que vamos a implementar es la de alinear el párrafo actual a la izquierda, al centro y a la derecha. Para ello vamos a poner arriba tres botones llamados BIzquierda, BDerecha y BCentro, cuyos eventos serían:

```
procedure TFormulario.BIzquierdaClick( Sender: TObject );
begin
 RichEdit.Paragraph.Alignment := taLeftJustify;
end;

procedure TFormulario.BCentroClick( Sender: TObject );
begin
 RichEdit.Paragraph.Alignment := taCenter;
end;

procedure TFormulario.BDerechaClick( Sender: TObject );
begin
 RichEdit.Paragraph.Alignment := taRightJustify;
end;
```

Otra característica que se le puede añadir a un párrafo es la de añadir un punto por la izquierda como hace Microsoft Word. Para ello vamos a añadir el botón de nombre BPunto cuyo procedimiento es:

```
procedure TFormulario.BPuntoClick( Sender: TObject );
begin
 with RichEdit.Paragraph do
   if Numbering = nsNone then
     Numbering := nsBullet
   else
     Numbering := nsNone;

 RichEdit.SetFocus;
end;
```

Si se pulsa una vez este botón añade un punto al párrafo y si se pulsa de nuevo lo quita.

Al párrafo actual se le pueden modificar también las propiedades:

```
FirstIndent -> Es el espacio en pixels por la izquierda que se le da a
la primera línea
LeftIndent  -> Es el espacio en pixels por la izquierda que se le da a
todas las líneas
RightIndent -> Es el espacio en pixels por la derecha que se le da a
todas las líneas
```

LAS OTRAS PROPIEDADES DEL COMPONENTE RICHEDIT

Hay otras propiedades que nos permiten personalizar nuestro editor de texto como son:

PlainTex: Si se activa esta propiedad al guardar el archivo de texto a disco no almacena las propiedades de color, fuente, etc. Se comporta como el Bloc de Notas.

SelLength: Es el número de caracteres seleccionados por el usuario.

SelStart: Es la posición en el texto donde comienza la selección.

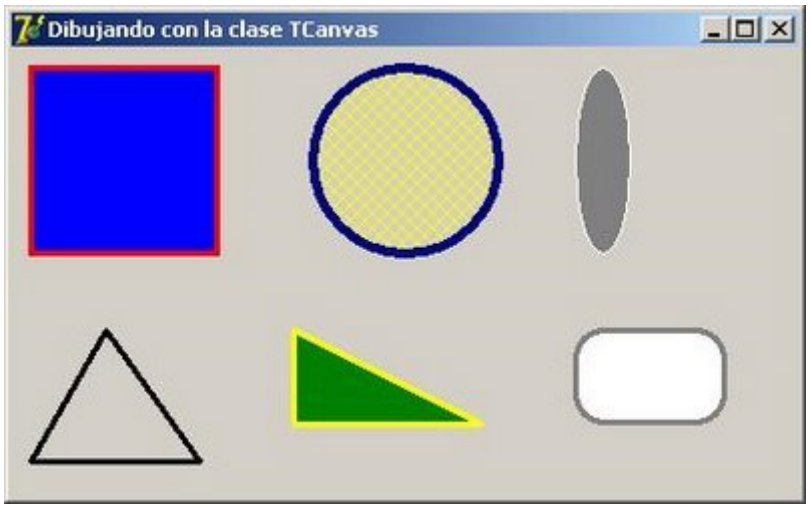
SelText: Es el texto seleccionado.

Con esto finalizamos las características más importantes del componente RichEdit.

Pruebas realizadas en Delphi 7.

Dibujando con la clase TCanvas (I)

La unidad Graphics dispone de la clase TCanvas dedicada al dibujo de objetos sobre la superficie de un control visual. Los controles estándar de Windows tales como Edit o Listbox no requieren canvas, ya que son dibujados por el sistema operativo.



Un objeto Canvas proporciona propiedades, métodos y eventos para realizar tareas como pueden ser:

- Escribir texto especificando fuente, color y estilo.

- Copiar imágenes de una superficie a otra.

- Dibujar líneas, rectángulos y circulos con distintos patrones de color y estilo.

- Leer y modificar cada pixel de la imagen dibujada.

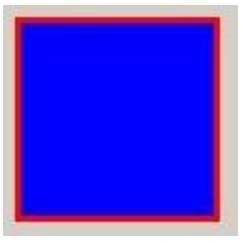
COMO DIBUJAR UN RECTANGULO

Antes de dibujar una figura se pueden seleccionar los colores del margen y de

fondo. Supongamos que quiero hacer un rectángulo con un marco de 3 pixels de ancho y de color rojo. El fondo va a ser de color azul. Para ello nos vamos al evento OnPaint del formulario y escribimos lo siguiente:

```
procedure TFormulario.FormPaint( Sender: TObject );
begin
  with Canvas do
  begin
    Pen.Color := clRed;
    Pen.Width := 3;
    Brush.Color := clBlue;
    Rectangle( 10, 10, 110, 110 );
  end;
end;
```

El resultado es el siguiente:



Hemos utilizado la propiedad Pen para seleccionar el color del marco y su ancho. Con la propiedad Brush establecemos el color de fondo del rectángulo. Y mediante el procedimiento Rectangle hemos dibujado un rectángulo en la posición x = 10 e y = 10 teniendo un ancho y alto de 100x100. El procedimiento Rectangle tiene los siguientes parámetros:

```
procedure Rectangle( X1, Y1, X2, Y2: Integer );
procedure Rectangle( const Rect: TRect );
```

X1, Y1 -> Son las coordenadas de la esquina superior izquierda del rectángulo
X2, Y2 -> Son las coordenadas de la esquina inferior derecha del rectángulo

También se podría haber utilizado la estructura de datos TRect (rectángulo) para dibujar el rectángulo del siguiente modo:

```
var
  R: TRect;
begin
  with Canvas do
  begin
    Pen.Color := clRed;
    Pen.Width := 3;
    Brush.Color := clBlue;
    R.Left := 10;
    R.Top := 10;
    R.Right := 110;
    R.Bottom := 110;
    Rectangle( R );
  end;
end;
```

La estructura TRect esta definida dentro de la unidad Type del siguiente modo:

```
TRect = packed record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
  end;
```

donde los valores Left y Top determinan la posición superior izquierda del rectángulo y los valores Right y Bottom la esquina inferior derecha.

También existe otro procedimiento para hacer rectángulos sólo con fondo (sin borde) utilizando el procedimiento:

```
procedure FillRect( const Rect: TRect );
```

Si lo hubiésemos utilizado este procedimiento sólo tendríamos un rectángulo con fondo azul. Lo único que tiene en cuenta es la propiedad Brush ignorando el valor de Pen.

Para el caso anterior no es necesario utilizar la estructura TRect, pero si se van a dibujar muchos rectángulos utilizando las mismas rutinas si es interesante utilizarla, evitándonos el tener que declarar las variables x1, y1, x2, y2 para las coordenadas. La estructura TRect es de uso general y no tiene porque estar asociada sólo al dibujo de rectángulos como veremos más adelante.

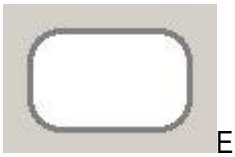
Las propiedades Pen y Brush son permantentes, es decir, que mientras no se modifiquen todo lo que se dibuje posteriormente tendrá los colores y estilo especificado por ambas. Si las figuras que se van a dibujar tienen mismo color de borde y fondo no es necesario especificar de nuevo el valor de Pen y Brush.

DIBUJANDO UN RECTANGULO CON ESQUINAS REDONDEADAS

Mediante el método RoundRect se pueden crear rectángulos con esquinas suavizadas. Veamos un ejemplo:

```
with Canvas do
begin
  Pen.Color := clGray;
  Pen.Width := 3;
  Brush.Color := clWhite;
  RoundRect( 300, 150, 380, 200, 30, 30 );
end;
```

Cuyo resultado sería:

 El procedimiento RoundRect tiene los siguientes parámetros:

```
procedure RoundRect( X1, Y1, X2, Y2, X3, Y3: Integer );
```

donde:

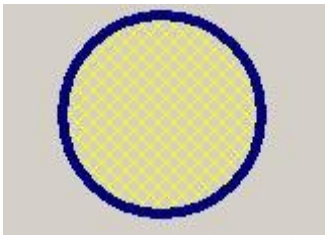
X1, Y1 -> Son las coordenadas de la esquina superior izquierda
X2, Y2 -> Son las coordenadas de la esquina inferior derecha
X3, Y3 -> Es grado de redondeo de las esquinas (cuanto más grande más redondeado)

COMO DIBUJAR UN CIRCULO

Para dibujar un círculo se utiliza el procedimiento Ellipse. Vamos a dibujar un círculo con un borde de color azul oscuro y un fondo de color amarillo:

```
with Canvas do
begin
  Pen.Color := clNavy;
  Pen.Width := 5;
  Brush.Color := clYellow;
  Brush.Style := bsDiagCross;
  Ellipse( 160, 10, 260, 110 );
end;
```

Quedaría así:



A la propiedad Brush le he especificado que me dibuje el fondo amarillo utilizando un patrón de líneas cruzadas diagonales. Al igual que el procedimiento Rectangle, el procedimiento Ellipse utiliza los mismos parámetros:

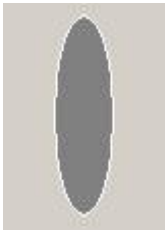
```
procedure Ellipse( X1, Y1, X2, Y2: Integer );
procedure Ellipse( const Rect: TRect );
```

En este otro ejemplo voy a dibujar un círculo chafado horizontalmente (una elipse) con un borde de 1 pixel de color blanco y un fondo gris:

```
with Canvas do
begin
  Pen.Color := clWhite;
  Pen.Width := 1;
  Brush.Color := clGray;
```

```
  Brush.Style := bsSolid;
  Ellipse( 300, 10, 330, 110 );
end;
```

Este sería el resultado:



He tenido que volver a dejar la propiedad Style del Brush con el valor bsSolid para que vuelva a dibujarme el fondo sólido porque sino me lo hubiera hecho con líneas cruzadas como lo dejé anteriormente.

COMO DIBUJAR LINEAS

La clase TCanvas dispone de un puntero invisible donde dibujará las líneas si no se especifica posición. Para modificar la posición de dicho puntero existe el método MoveTo:

```
procedure MoveTo( X, Y: Integer );
```

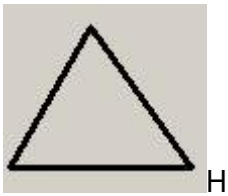
Si queremos averiguar donde se ha quedado el puntero disponemos de la propiedad PenPos que es de tipo TPoint:

```
type TPoint = packed record
  X: Longint;
  Y: Longint;
end;
```

Al igual que la estructura de datos TRect está definida en la unidad Types. Veamos como realizar un triángulo de color negro y sin fondo:

```
with Canvas do
begin
  Pen.Color := clBlack;
  Pen.Width := 3;
  MoveTo( 50, 150 );
  LineTo( 100, 220 );
  LineTo( 10, 220 );
  LineTo( 50, 150 );
end;
```

El triángulo quedaría así:



Hemos situado el puntero en un punto y a partir de ahí hemos ido tranzando líneas hasta cerrar el triángulo. En este caso la propiedad Brush

no tiene ningún valor para el procedimiento LineTo.

COMO DIBUJAR POLIGONOS

El triángulo que hemos hecho anteriormente también podría haberse realizado mediante un polígono. Además los polígonos permiten especificar el color de fondo mediante la propiedad Brush. Por ejemplo vamos a crear un triángulo con un borde amarillo y fondo de color verde:

```
var
  Puntos: array of TPoint;
begin
  with Canvas do
  begin
    Pen.Color := clYellow;
    Pen.Width := 3;
    Brush.Color := clGreen;
    SetLength( Puntos, 3 );
    Puntos[0].x := 150;
    Puntos[0].y := 150;
    Puntos[1].x := 250;
    Puntos[1].y := 200;
    Puntos[2].x := 150;
    Puntos[2].y := 200;
    Polygon( Puntos );
  end;
end;
```

Quedaría así:

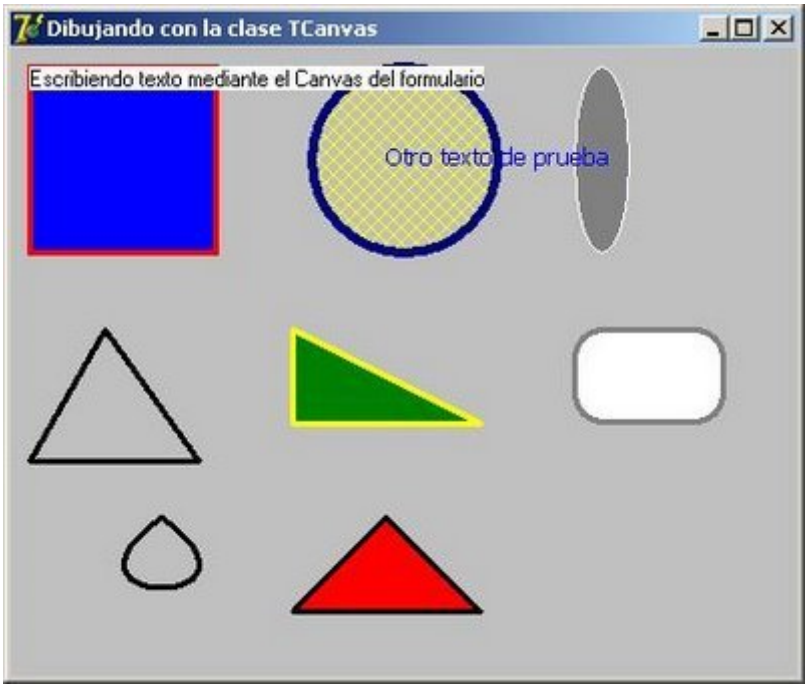
He creado un array dinámico del tipo TPoint con una longitud de 3. Después he ido especificando cada punto del polígono (en este caso un triángulo) y por último le paso dicho array al procedimiento Polygon.

En el próximo artículo seguiremos exprimiendo las características del Canvas.

Pruebas realizadas en Delphi 7.

Dibujando con la clase TCanvas (II)

Una vez que ya sabemos como dibujar las figuras básicas con el objeto Canvas pasemos ahora a ver algunas más complicadas.



DIBUJAR UN POLIGONO SOLO CON LINEAS

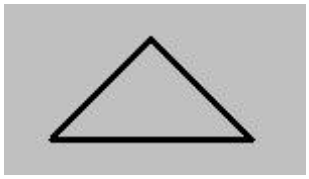
Para dibujar un polígono transparente utilizando líneas tenemos el procedimiento:

```
procedure Polyline( Points: array of TPoint );
```

Funciona exactamente igual que el procedimiento Polygon salvo que no cierra el polígono automáticamente, es decir, el último punto y el primero tiene que ser el mismo (para crear el polígono, aunque puede realizarse cualquier otra figura). Este procedimiento es equivalente a crear un trazado de líneas utilizando el procedimiento LineTo. Veamos un ejemplo:

```
with Canvas do
begin
  Pen.Color := clBlack;
  SetLength( Puntos, 4 );
  Puntos[0].x := 200;
  Puntos[0].y := 250;
  Puntos[1].x := 250;
  Puntos[1].y := 300;
  Puntos[2].x := 150;
  Puntos[2].y := 300;
  Puntos[3].x := 200;
  Puntos[3].y := 250;
  PolyLine( Puntos );
end;
```

Este sería el resultado:



DIBUJAR POLIGONOS BEZIER

Los polígonos de Bezier se dibujan trazando curvas entre todos los puntos del polígono, es decir, son polígonos con las esquinas muy redondeadas. El procedimiento para dibujarlos es el siguiente:

```
with Canvas do
begin
  Pen.Color := clBlack;
  SetLength( Puntos, 4 );
  Puntos[0].x := 80;
  Puntos[0].y := 250;
  Puntos[1].x := 150;
  Puntos[1].y := 300;
  Puntos[2].x := 10;
  Puntos[2].y := 300;
  Puntos[3].x := 80;
  Puntos[3].y := 250;
  PolyBezier( Puntos );
end;
```

Este sería el resultado:

Aunque he dado las mismas coordenadas que un triángulo, tiene las esquinas inferiores redondeadas.

ESCRIBIENDO TEXTO

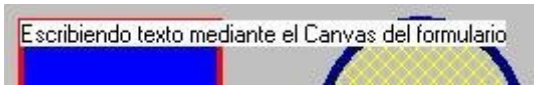
Para escribir texto encima de una imagen con la clase TCanvas tenemos el método:

```
procedure TextOut( X, Y: Integer; const Text: string );
```

Este método toma como parámetros las coordenadas donde va a comenzar a escribirse el texto y el texto a escribir. Por ejemplo:

```
with Canvas do
begin
  TextOut( 10, 10, 'Escribiendo texto mediante el Canvas del
formulario' );
end;
```

Al no especificar fuente ni color quedaría del siguiente modo:



Cuando se escribe texto en un formulario la fuente, el color del texto y el color del fondo vienen predeterminados por las propiedades Brush y Font del Canvas. Supongamos que quiero escribir texto con fuente Tahoma, negrita, 10 y de color azul con fondo transparente:

```
with Canvas do
begin
  Brush.Style := bsClear;
  Font.Color := clBlue;
  Font.Name := 'Tahoma';
  Font.Size := 10;
  TextOut( 200, 50, 'Otro texto de prueba' );
end;
```

Aquí tenemos nuestro texto personalizado:

Pero nos puede surgir un pequeño problema. ¿Como podemos averiguar lo que va a ocupar en pixels el texto que vamos a escribir? Pues para ello tenemos la siguiente función:

```
function TextWidth( const Text: string ): Integer;
```

Nos devuelve el ancho en pixels del texto que le pasamos como parámetro según como esté la propiedad Font antes de llamar a esta función. Y si queremos saber su altura entonces tenemos esta otra función:

```
function TextHeight( const Text: string ): Integer;
```

Lo que afecta a esta función principalmente es la propiedad Font.Size del Canvas. También tenemos esta otra función para obtener simultáneamente el ancho y alto del texto a escribir:

```
function TextExtent( const Text: string ): TSize;
```

Donde TSize es una estructura definida en la unidad Types del siguiente modo:

```
type
  tagSIZE = packed record
    cx: Longint;
    cy: Longint;
  end;
TSize = tagSIZE;
```

COMO RELLENAR LAS FIGURAS DE UN COLOR

Todos los programas de dibujo incorporan la función de rellenar con pintura una imagen hasta que encuentre bordes. Es como volcar un bote de pintura sobre las superficie hasta que choque con algo. El Canvas del formulario dispone del procedimiento:

```
procedure FloodFill( X, Y: Integer; Color: TColor; FillStyle:
TFillStyle );
```

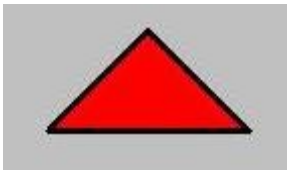
Los parámetros X, Y especifican donde se va a comenzar a pintar, el parámetro Color establece el color del borde donde va a chocar la pintura y FillStyle define el modo de pintar. Veamos un ejemplo de cómo pintar un triángulo de rojo con bordes negros:

```
with Canvas do
begin
  Brush.Color := clRed;
  FloodFill( 200, 270, clBlack, fsBorder );
end;
```

El último parámetro (FillStyle) se utiliza para decirle a la función si queremos que pinte hasta el borde de un color en concreto (clBlack y fsBorder) o si deseamos que dibuje toda la superficie de un color hasta que encuentre un color distinto. Por ejemplo:

```
with Canvas do
begin
  Brush.Color := clRed;
  FloodFill( 200, 270, clSilver, fsSurface );
end;
```

En este caso le he dicho que me pinte de rojo toda la superficie cuyo fondo es de color clSilver. Si encuentra un color que no sea clSilver entonces se para. Esa es la diferencia entre los valores fsBorder (un sólo borde) y fsSurface (cualquier borde pero la misma superficie). En ambos ejemplos he rellenado de color rojo el polígono creado al principio de este artículo:



En el próximo artículo terminaremos de ver las propiedades de la clase TCanvas.

Pruebas realizadas en Delphi 7.

Dibujando con la clase TCanvas (III)

Vamos a terminar de ver lo más importante de las operaciones que se pueden realizar con el objeto Canvas.

DIBUJAR UN RECTANGULO SIN FONDO

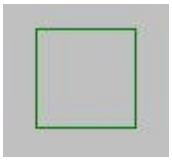
El procedimiento FrameRect permite crear rectángulos sin fondo teniendo en cuenta el valor de la propiedad Brush:

```
procedure FrameRect( const Rect: TRect );
```

Veamos un ejemplo:

```
with Canvas do
begin
  R.Left := 300;
  R.Top := 250;
  R.Right := 350;
  R.Bottom := 300;
  Brush.Color := clGreen;
  FrameRect( R );
end;
```

Este sería el dibujo resultante:



COMO COPIAR IMAGENES DE UN BITMAP A OTRO

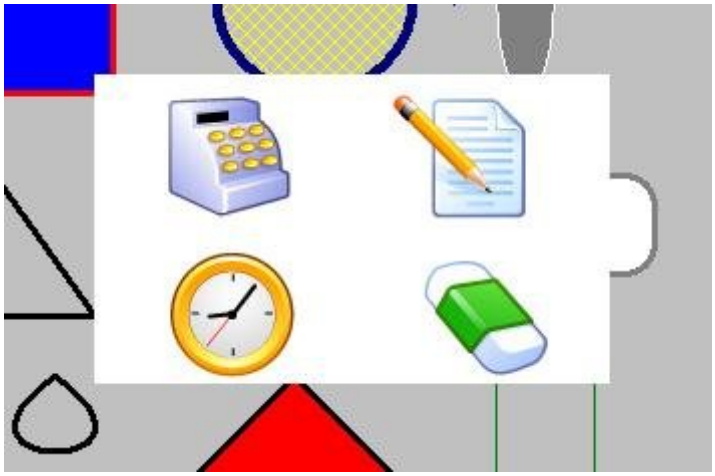
La manera más utilizada de copiar imágenes es mediante el método Draw:

```
procedure Draw( X, Y: Integer; Graphic: TGraphic );
```

Vamos a ver un ejemplo de como copiar la imagen de un objeto de la clase TImage a nuestro formulario utilizando el procedimiento Draw:

```
var
R: TRect;
begin
  with Canvas do
  begin
    R.Left := 300;
    R.Top := 250;
    R.Right := 350;
    R.Bottom := 300;
    Brush.Color := clGreen;
    FrameRect( R );
    Draw( 100, 100, Imagen.Picture.Graphic );
  end;
end;
```

Hemos supuesto que el objeto de la clase TImage se llama Imagen. Este sería el resultado:



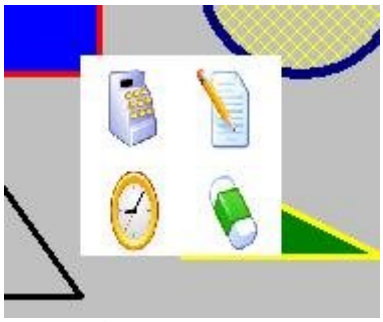
La imagen copiada consta de cuatro iconos y un fondo de color blanco. Otra cosa que podemos hacer es modificar el tamaño de la imagen a copiar utilizando el procedimiento:

```
procedure StretchDraw( const Rect: TRect; Graphic: TGraphic );
```

El parámetro Rect determina las nuevas coordenadas así como en ancho y alto de la imagen a copiar. En el siguiente ejemplo voy a copiar la misma imagen pero voy a reducirla un tamaño de 100x100:

```
var
  R: TRect;
begin
  with Canvas do
  begin
    R.Left := 100;
    R.Top := 100;
    R.Right := 200;
    R.Bottom := 200;
    StretchDraw( R, Imagen.Picture.Graphic );
  end;
end;
```

Quedaría de la siguiente manera:



También se pueden copiar imágenes utilizando el procedimiento:

```
procedure CopyRect( const Dest: TRect; Canvas: TCanvas; const Source:
TRect );
```

cuyos parámetros son:

Dest: Coordenadas del rectángulo destino.
Canvas: Referencia al Canvas del origen a copiar.
Source: Coordenadas del rectángulo origen.

Para el ejemplo anterior voy a copiar la imagen en su tamaño original al formulario:

```
var
  Origen, Destino: TRect;
begin
  with Canvas do
  begin
    Origen.Left := 0;
    Origen.Top := 0;
    Origen.Right := Imagen.Width;
    Origen.Bottom := Imagen.Height;
    Destino.Left := 100;
    Destino.Top := 100;
    Destino.Right := 100 + Imagen.Width;
    Destino.Bottom := 100 + Imagen.Height;
    CopyRect( Destino, Imagen.Canvas, Origen );
  end;
end;
```

Entonces, ¿que diferencia hay entre Draw o StretchDraw y CopyRect? La diferencia es que CopyRect sólo puede utilizarse cuando el objeto TImage ha cargado un bitmap (*.BMP) porque si es un JPG provoca una excepción. En cambio las funciones Draw o StretchDraw siempre funcionan independientemente del tipo de imagen que sea.

LOS MODOS DE COPIA DE UNA IMAGEN

En principio cuando se realiza la copia de una imagen a otra, la copia de los pixels es exacta. Pero si queremos modificar el modo de copiar entonces la clase TCanvas tiene de la propiedad CopyMode que establece que tipo de operación que se va a realizar. Estos son sus posibles valores:

cmBlackness: pinta la imagen destino de color negro, independientemente del origen.

cmDstInvert: Invierte los colores de la imagen según la imagen destino.

cmMergeCopy: mezcla la imagen origen y destino utilizando el operador binario AND.

cmMergePaint: mezcla la imagen origen y destino utilizando el operador binario OR.

cmNotSrcCopy: copia la imagen origen invertida a la imagen destino.

cmNotSrcErase: mezcla las imagenes origen y destino y después las invierte con el operador binario OR.

cmPatCopy: copia la imagen según el valor de la propiedad Brush.Style del Canvas.

cmPatInvert: copia la imagen invertida según el valor de la propiedad Brush.Style del Canvas.

cmPatPaint: combina las imágenes origen y destino utilizando la operación binaria OR para luego invertirlas.

cmSrcAnd: combina la imagen origen con la imagen destino utilizando el operador binario AND.

cmSrcCopy: realiza una copia exacta de la imagen origen a la imagen destino (por defecto).

cmSrcErase: invierte la imagen destino y copia el origen utilizando el operador binario AND.

cmSrcInvert: invierte las imágenes origen y destino utilizando el operador binario XOR.

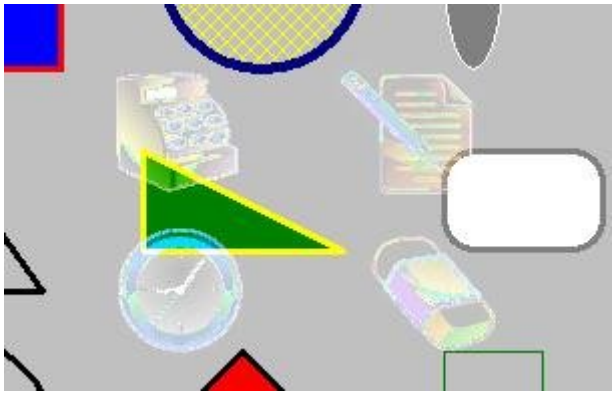
cmSrcPaint: combina las imágenes destino y origen utilizando el operador binario AND.

cmWhiteness: pinta toda la imagen destino de color blanco ignorando la imagen origen.

Por ejemplo voy a crear un efecto fantasma con la imagen origen:

```
with Canvas do
begin
  CopyMode := cmMergePaint;
  Draw( 100, 100, Imagen.Picture.Graphic );
end;
```

El efecto sería el siguiente:



COMO ACCEDER A LOS PIXELS DE LA IMAGEN

Si no son suficientes las operaciones que podemos realizar en una imagen

también podemos acceder directamente a los pixels de una imagen a traves de su propiedad Pixels:

```
property Pixels[ X, Y: Integer ]: TColor;
```

Esta propiedad nos sirve igualmente para leer y para escribir pixels en la imagen. El componente TColor es un tipo entero de 32 bits definido en la unidad Graphics del siguiente modo:

```
type
 TColor = -$7FFFFFFF-1..$7FFFFFFF;
```

Dentro del mismo número entero están definidos los colores básicos RGB los cuales son:

R = Red (Rojo)
G = Green (Verde)
B = Blue (Azul)

Combinando estos tres colores se puede crear cualquier otro color. Estos colores estan dentro del valor TColor del siguiento modo:

$00GGBBRR

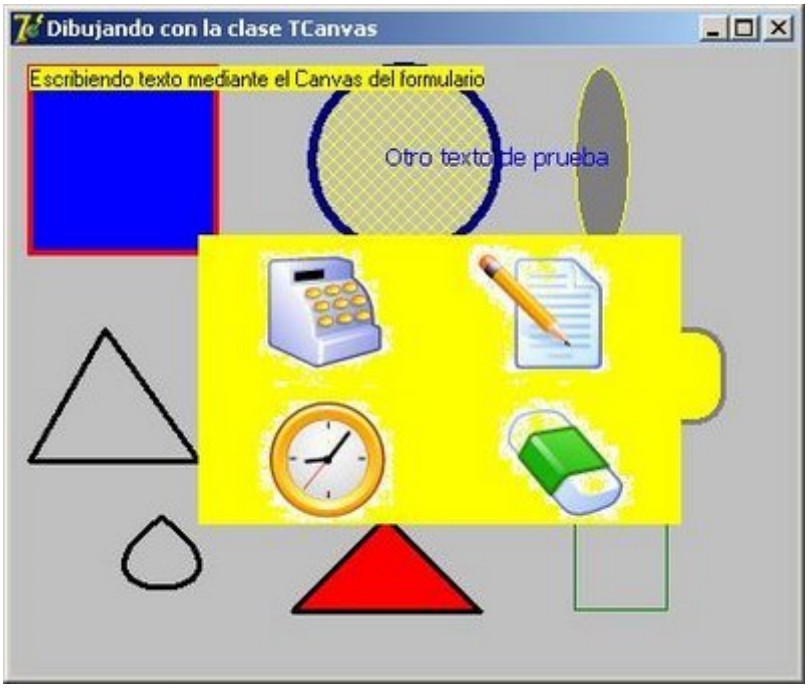
Donde GG es el byte en hexadecimal que representa el color verde, BB es el color azul y RR es el color rojo. Aquí tenemos unos ejemplos de los números en hexadecimal:

```
Rojo   := $000000FF;
Azul   := $0000FF00;
Verde  := $00FF0000;
Blanco := $00FFFFFF;
Negro  := $00000000;
```

Para no complicarnos mucho la vida Delphi ya dispone de colores predeterminados tales como clRed (Rojo), clBlue (Azul), etc. Sabiendo esto imaginaos que deseo hacer un programa que convierta todos los colores blancos de la imagen en amarillo:

```
var
  i, j: Integer;
begin
  with Canvas do
  for j := 0 to ClientHeight - 1 do
    for i := 0 to ClientWidth - 1 do
      if Pixels[i,j] = clWhite then
        Pixels[i,j] := clYellow;
end;
```

Mediante un doble bucle recorro toda la imagen y compruebo si el pixel donde estoy es blanco para sustituirlo por el amarillo. Este sería el resultado:



Con este método se podemos realizar nuestros propios filtros o crear cualquier tipo de efecto.

Con esto finalizamos el apartado dedicado al objeto Canvas.

Pruebas realizadas en Delphi 7.

El componente TTreeView (I)

Cuando queremos representar una información en forma de árbol incluyendo nodos padres e hijos el componente ideal para ello es el TreeView, el cual funciona exactamente igual que la parte izquierda del Explorador de Windows.

Veamos las peculiaridades que aporta este componente. Todas las funciones las voy a aplicar sobre esta pantalla:

AÑADIENDO ELEMENTOS AL ARBOL

Cuando se añaden elementos a un árbol pueden pertenecer al elemento raiz del mismo (sin padre) o pertenecer a un elemento ya creado. Los elementos del árbol se llaman nodos (Node) pudiendose crear tantos niveles de nodos como se deseen.

Vamos a crear un procedimiento asociado al botón Nuevo que va a añadir un nuevo nodo al árbol. El elemento insertado será un nodo raiz, a menos que el usuario seleccione un nodo antes de ello, que hará que sea su hijo:

```
procedure TFTreeView.BNuevoClick( Sender: TObject );
var
  sNombre: string;
begin
  sNombre := InputBox( 'Crear un nodo', 'Nombre:', '' );

  if sNombre <> '' then
  begin
    // ¿Hay un nodo seleccionado?
    if TreeView.Selected <> nil then
    begin
      TreeView.Items.AddChild( TreeView.Selected, sNombre );
      TreeView.Selected.Expanded := True;
    end
    else
      TreeView.Items.Add( nil, sNombre );
  end;
end;
```

Lo que hace este procedimiento es preguntarnos el nombre del nodo y si el usuario ha seleccionado uno anteriormente lo mete como hijo (AddChild) expandiendo además el nodo padre (Expanded) como si el usuario hubiese pulsado el botón + del elemento padre.

MODIFICANDO LOS ELEMENTOS DEL ARBOL

Una vez creado un árbol de elementos se puede modificar el texto de cada uno de ellos mediante la propiedad Text. Vamos a hacer que si el usuario pulsa el botón Modificar y hay un nodo seleccionado el programa nos pregunte el nuevo nombre del nodo y lo cambie:

```
procedure TFTreeView.BModificarClick( Sender: TObject );
var
  sNombre: string;
begin
  if TreeView.Selected <> nil then
  begin
    sNombre := InputBox( 'Crear un nodo', 'Nombre:',
TreeView.Selected.Text );

    if sNombre <> '' then
      TreeView.Selected.Text := sNombre;
  end;
end;
```

Cuando se van a hacer operaciones con los nodos hay que asegurarse siempre que el nodo al que vayamos a acceder no sea nil, para evitar Access Violations.

ELIMINANDO LOS ELEMENTOS DEL ARBOL

Para eliminar un nodo del árbol se utiliza el método Delete:

```
procedure TFTreeView.BEliminarClick( Sender: TObject );
begin
  if TreeView.Selected <> nil then
    TreeView.Selected.Delete;
end;
```

Y si queremos eliminar todos los nodos se utiliza el método Clear de la propiedad Items:

```
procedure TFTreeView.BBorrarTodosClick( Sender: TObject );
begin
  TreeView.Items.Clear;
end;
```

ORDENANDO LOS NODOS ALFABETICAMENTE

El modo de ordenar los elementos de un componente TreeView es igual al de ordenar los elementos de un componente ListView. Hay dos métodos: CustomSort y AlphaSort. El método CustomSort no voy a explicarlo ya que lo mencioné anteriormente en el artículo dedicado al componente ListView y es un poco más primitivo que utilizar AlphaSort.

Para ordenar todos los elementos de un árbol TreeView se utiliza el método:

```
procedure TFTreeView.BOrdenarClick( Sender: TObject );
begin
  TreeView.AlphaSort( True );
end;
```

El parámetro booleano especifica si queremos que ordene tanto los elementos padres como los hijos (True) o sólo los elementos padres. Si no se especifica nada se asume que ordene todos los elementos.

Ahora hay que programar el evento OnCompare del TreeView para que ordene alfabéticamente los elementos:

```
procedure TFTreeView.TreeViewCompare( Sender: TObject; Node1,
  Node2: TTreeNode; Data: Integer; var Compare: Integer );
begin
  Compare := CompareText( Node1.Text, Node2.Text );
end;
```

Si queremos que la ordenación sea descendente entonces sólo hay que cambiar el signo:

```
Compare := -CompareText( Node1.Text, Node2.Text );
```

GUARDANDO LA INFORMACION DEL ARBOL EN UN ARCHIVO DE TEXTO

La clase TTreeView dispone del método SaveToFile para volcar todos los nodos en un archivo de texto para su posterior utilización. Voy a hacer un procedimiento que pregunte al usuario que nombre deseamos dar al archivo y lo guardará en el mismo directorio del programa con la extensión .TXT:

```
procedure TFTreeView.BGuardarClick( Sender: TObject );
var sNombre: string;
begin
  sNombre := InputBox( 'Crear un nodo', 'Nombre:', '' );

  if sNombre <> '' then
    TreeView.SaveToFile( ExtractFilePath( Application.ExeName ) +
sNombre + '.txt' );
end;
```

La información la guarda por líneas donde cada elemento hijo va separado por tabuladores:

```
documentos
    ventas
contactos
    Pablo
    Ana
    Maria
claves
    Terra
    Hotmail
    GMail
```

CARGANDO LA INFORMACION DEL ARBOL GUARDADA ANTERIORMENTE

Para cargar el archivo de texto vamos a crear en tiempo real el componente TOpenDialog para que le pregunte al usuario el archivo de texto a cargar:

```
procedure TFTreeView.BCargarClick( Sender: TObject );
var
  Abrir: TOpenDialog;
begin
  Abrir := TOpenDialog.Create( Self );
  Abrir.InitialDir := ExtractFilePath( Application.ExeName );
  Abrir.Filter := 'TreeView|*.txt';

  if Abrir.Execute then
    TreeView.LoadFromFile( Abrir.FileName );

  Abrir.Free;
end;
```

Hemos modificado el filtro de la carga para que sólo se vean archivos TXT.

En el próximo artículo seguiremos viendo más propiedades interesantes del componente TreeView.

Pruebas realizadas en Delphi 7.

El componente TTreeView (II)

Después de ver el manejo básico de un árbol TreeView pasemos a ver otras características menos conocidas pero también de suma importancia.

OBTENIENDO EL NIVEL DE CADA NODO

Cuando se van insertando elementos dentro de un árbol tenemos la dificultad de saber en que nivel de profundidad se encuentra cada uno de los nodos. Cada uno de los Items del componente TTreeView es de la clase TTreeNode.

Esta clase dispone de la propiedad Level la cual nos da el nivel de profundidad dentro del árbol. Veamos como obtener el nivel de cada uno de los nodos del árbol. Este procedimiento recorre todos los nodos del árbol y les añade a su nombre el nivel:

```
procedure TFTreeView.BNivelClick( Sender: TObject );
var i: Integer;
begin
  // Averiguamos el nivel de cada nodo
  for i := 0 to TreeView.Items.Count - 1 do
    TreeView.Items[i].Text := TreeView.Items[i].Text + '_' +
      IntToStr( ( TreeView.Items[i] as TTreeNode ).Level );
end.
```

Los elementos situados en la raiz tienen un nivel 0, los hijos un nivel 1, los nietos un 2...

LEYENDO LOS NODOS SELECCIONADOS

El nodo actualmente seleccionado se obtiene mediante:

```
TreeView.Selected
```

donde valdrá nil si no hay ninguno seleccionado. Pero si activamos la propiedad MultiSelect la forma de leer aquellos nodos seleccionados sería la siguiente:

```
procedure TFTreeView.BSeleccionadosClick( Sender: TObject );
var
  i: Integer;
  Seleccionados: TStringList;
begin
  Seleccionados := TStringList.Create;

  for i := 0 to TreeView.Items.Count - 1 do
    if TreeView.Items[i].Selected then
      Seleccionados.Add( TreeView.Items[i].Text );

  ShowMessage( Seleccionados.Text );
```

```
  Seleccionados.Free;
end;
```

Al igual que hicimos con el componente ListView volcamos el nombre de los nodos seleccionados en un StringList y lo sacamos por pantalla.

ASOCIANDO UNA IMAGEN A CADA NODO

Si añadimos a nuestro formulario el componente TImageList se pueden asocionar iconos distintos a cada uno de los elementos del árbol TTreeView. Para ello asociamos este componente a la propiedad Images del TTreeView. Una vez hecho esto todos los nodos tendrán a su izquierda la primera imagen de la lista de imágenes TImageList.

Ya será cuestión de cada cual asociar la imagen correspondiente cada nodo. Vamos a ver un ejemplo que recorre los elementos del árbol y va a poner la segunda imagen de la lista de imágenes a aquellos nodos hijos (de nivel 1):

```
procedure TFTreeView.CambiarNodosHijos;
var
  i: Integer;
begin
  for i := 0 to TreeView.Items.Count - 1 do
    if ( TreeView.Items[i] as TTreeNode ).Level = 1 then
      ( TreeView.Items[i] as TTreeNode ).ImageIndex := 1;
end;
```

CAMBIANDO LA FORMA EN QUE SE MUESTRAN LOS NODOS

El componente TreeView dispone la propiedad Ident la cual determina la identación de entre los nodos padres y sus hijos que por defecto es de 19 pixels. Se puede cambiar en cualquier momento, pero afectará a todos los nodos del árbol.

Otra cosa que se puede hacer al cargar un árbol desde un archivo de texto es expandir todos sus nodos, ya que cuando se carga el árbol esta compactado. Para solucionarlo se hace:

```
TreeView.FullExpand;
```

Equivale a pulsar el botón + de cada uno de los nodos padres.

CAMBIANDO LA FORMA DE DIBUJAR LOS NODOS

Al igual que vimos con el componente ListView también se puede modificar en tiempo real la forma de dibujar cada nodo. Para ello lo que hacemos es reprogramar el evento OnCustomDrawItem. Veamos un ejemplo de como hacer que los nodos hijos aparenzan de color de fuente azul y negrita:

```
procedure TFTreeView.TreeViewCustomDrawItem( Sender: TCustomTreeView;
  Node: TTreeNode; State: TCustomDrawState; var DefaultDraw: Boolean
);
```

```
begin
  if Node.Level = 1 then
  begin
    Sender.Canvas.Font.Color := clBlue;
    Sender.Canvas.Font.Style := [fsBold];
  end
  else
  begin
    Sender.Canvas.Font.Color := clBlack;
    Sender.Canvas.Font.Style := [];
  end;

  if cdsFocused in State then
    Sender.Canvas.Font.Color := clWhite;
end;
```

En las dos últimas líneas del procedimiento también nos hemos asegurado de que si un elemento esta seleccionado la fuente salga de color blanca. Hemos utilizado para ello el parámetro State del evento. Los posibles valores de esta variable son:

```
cdsSelected       -> La columna o fila ha sido seleccionada
cdsGrayed         -> La columna o fila esta grisacea
cdsDisabled       -> La columna o fila esta deshabilitada
cdsChecked        -> La fila aparece con el CheckBox activado
cdsFocused        -> La columna o fila esta enfocada
cdsDefault        -> Por defecto
cdsHot            -> Se ha activado el HotTrack y esta enfocado
cdsMarked         -> La fila esta marcada
cdsIndeterminate  -> La fila no esta seleccionada ni deseleccionada
```

Con esto se resumen las propiedades más importantes del componente TreeView.

Pruebas realizadas en Delphi 7.

Explorar unidades y directorios

Si importante es controlar el manejo de archivos no menos importante es el saber moverse por las unidades de disco y los directorios.

Veamos que tenemos Delphi para estos menesteres:

function CreateDir(const Dir: string): Boolean;

Esta función crea un nuevo directorio en la ruta indicada por Dir. Devuelve True o False dependiendo si ha podido crearlo o no. El único inconveniente que tiene esta función es que deben existir los directorios padres. Por ejemplo:

```
CreateDir( 'C:\prueba' )                devuelve True
CreateDir( 'C:\prueba\documentos' )     devuelve True
CreateDir( 'C:\otraprueba\documentos' ) devuelve False (y no lo crea)
```

function ForceDirectories(Dir: string): Boolean;

Esta función es similar a CreateDir salvo que también crea toda la ruta de directorios padres.

```
ForceDirectories( 'C:\prueba' )                devuelve True
ForceDirectories( 'C:\prueba\documentos' )     devuelve True
ForceDirectories( 'C:\otraprueba\documentos' ) devuelve True
```

procedure ChDir(const S: string); overload;

Este procedimiento cambia el directorio actual al indicado por el parámetro S. Por ejemplo:

```
ChDir( 'C:\Windows\Fonts' );
```

function GetCurrentDir: string;

Nos devuelve el nombre del directorio actual donde estamos posicionados. Por ejemplo:

```
GetCurrentDir devuelve C:\Windows\Fonts
```

function SetCurrentDir(const Dir: string): Boolean;

Establece el directorio actual devolviendo True si lo ha conseguido. Por ejemplo:

```
SetCurrentDir( 'C:\Windows\Java' );
```

procedure GetDir(D: Byte; var S: string);

Devuelve el directorio actual de una unidad y lo mete en la variable S. El parámetro D es el número de la unidad siendo:

```
D    Unidad
---  ------------------
0    Unidad por defecto
1    A:
2    B:
3    C:
...
```

Por ejemplo para leer el directorio actual de la unidad C:

```
var
  sDirectorio: String;
begin
  GetDir( 3, sDirectorio );
  ShowMessage( 'El directorio actual de la unidad C: es ' +
sDirectorio );
end;
```

function RemoveDir(const Dir: string): Boolean;

Elimina un directorio en el caso de que este vacío, devolviendo False si no ha podido hacerlo.

```
RemoveDir( 'C:\prueba\documentos' ) devuelve True
RemoveDir( 'C:\prueba' )            devuelve True
RemoveDir( 'C:\otraprueba' )        devuelve False porque no esta
vacío
```

function DirectoryExists(const Directory: string): Boolean;

Comprueba si existe el directorio indicado por el parámetro Directory. Por ejemplo:

```
DirectoryExists( 'C:\Windows\System32\' )      devuelve True
DirectoryExists( 'C:\Windows\MisDocumentos\' ) devuelve False
```

function DiskFree(Drive: Byte): Int64;

Devuelve el número de bytes libres de una unidad de dico indicada por la letra Drive:

```
Drive          Unidad
-----------    ------
0              Unidad por defecto
1              A:
2              B:
3              C:
...
```

Por ejemplo vamos a ver el número de bytes libres de la unidad C:

```
DiskFree( 3 ) devuelve 5579714560
```

function DiskSize(Drive: Byte): Int64;

Nos dice el tamaño total en bytes de una unidad de disco. Por ejemplo:

```
DiskSize( 3 ) devuelve 20974428160
```

BUSCANDO ARCHIVOS DENTRO DE UN DIRECTORIO

Para buscar archivos dentro de un directorio disponemos de las funciones:

function FindFirst(const Path: string; Attr: Integer; var F: TSearchRec): Integer;

Busca el primer archivo, directorio o unidad que se encuentre dentro de una

ruta en concreto. Devuelve un cero si ha encontrado algo. El parámetro TSearchRec es una estructura de datos donde se almacena lo encontrado:

```
type
  TSearchRec = record
    Time: Integer;
    Size: Integer;
    Attr: Integer;
    Name: TFileName;
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData;
  end;
```

function FindNext(var F: TSearchRec): Integer;

Busca el siguiente archivo, directorio o unidad especificado anteriormente por la función FindFirst. Devuelve un cero si ha encontrado algo.

procedure FindClose(var F: TSearchRec);

Este procedimiento cierra la búsqueda comenzada por FindFirst y FindNext.

Veamos un ejemplo donde se utilizan estas funciones. Vamos a hacer un procedimiento que lista sólo los archivos de un directorio que le pasemos y vuelca su contenido en un StringList:

```
procedure TFPrincipal.Listar( sDirectorio: string; var Resultado:
TStringList );
var
  Busqueda: TSearchRec;
  iResultado: Integer;
begin
  // Nos aseguramos que termine en contrabarra
  sDirectorio := IncludeTrailingBackslash( sDirectorio );

  iResultado :=  FindFirst( sDirectorio + '*.*', faAnyFile, Busqueda
);

  while iResultado = 0 do
  begin
    // ¿Ha encontrado un archivo y no es un directorio?
    if ( Busqueda.Attr and faArchive = faArchive ) and
       ( Busqueda.Attr and faDirectory <> faDirectory ) then
      Resultado.Add( Busqueda.Name );

    iResultado := FindNext( Busqueda );
  end;

  FindClose( Busqueda );
end;
```

Si listamos el raiz de la unidad C:

```
var
  Directorio: TStringList;
```

```
begin
  Directorio := TStringList.Create;
  Listar( 'C:', Directorio );
  ShowMessage( Directorio.Text );
  Directorio.Free;
end;
```

El resultado sería:

```
AUTOEXEC.BAT
Bootfont.bin
CONFIG.SYS
INSTALL.LOG
IO.SYS
MSDOS.SYS
NTDETECT.COM
```

Con estas tres funciones se pueden hacer cosas tan importantes como eliminar directorios, realizar búsquedas de archivos, calcular lo que ocupa un directorio en bytes, etc.

Pruebas realizadas en Delphi 7.

Mostrando datos en el componente StringGrid

Anteriormente vimos como mostrar información en un componente ListView llegando incluso a cambiar el color de filas y columnas a nuestro antojo. El único inconveniente estaba en que no se podían cambiar los títulos de las columnas, ya que venían predeterminadas por los colores de Windows.

Pues bien, el componente de la clase TStringGrid es algo más cutre que el ListView, pero permite cambiar al 100% el formato de todas las celdas. Veamos primero como meter información en el mismo. Al igual que ocurría con el ListView, todos las celdas de un componente StringGrid son de tipo string, siendo nosotros los que le tenemos que dar formato a mano.

AÑADIENDO DATOS A LA REJILLA

Vamos a crear una rejilla de datos con las siguiente columnas:

```
NOMBRE, APELLIDO1, APELLIDO2, NIF, IMPORTE PTE.
```

Cuando insertamos un componente StringGrid en el formulario nos va a poner por defecto la primera columna con celdas fijas (fixed). Vamos a fijar las siguientes propiedades:

```
Propiedad          Valor     Descripción
---------          -----     -----------
ColCount           5         5 columnas
RowCount           4         4 filas
FixedCols          0         0 columnas fijas
```

```
FixedRows          1         1 fila fija
DefaultRowHeight   20        altura de las filas a 20 pixels
```

Ahora creamos un procedimiento para completar de datos la rejilla:

```
procedure TFormulario.RellenarTabla;
begin
  with StringGrid do
  begin
    // Título de las columnas
    Cells[0, 0] := 'NOMBRE';
    Cells[1, 0] := 'APELLIDO1';
    Cells[2, 0] := 'APELLIDO2';
    Cells[3, 0] := 'NIF';
    Cells[4, 0] := 'IMPORTE PTE.';

    // Datos
    Cells[0, 1] := 'PABLO';
    Cells[1, 1] := 'GARCIA';
    Cells[2, 1] := 'MARTINEZ';
    Cells[3, 1] := '67348321D';
    Cells[4, 1] := '1500,36';

    // Datos
    Cells[0, 2] := 'MARIA';
    Cells[1, 2] := 'SANCHEZ';
    Cells[2, 2] := 'PALAZON';
    Cells[3, 2] := '44878234A';
    Cells[4, 2] := '635,21';

    // Datos
    Cells[0, 3] := 'CARMEN';
    Cells[1, 3] := 'PEREZ';
    Cells[2, 3] := 'GUILLEN';
    Cells[3, 3] := '76892693L';
    Cells[4, 3] := '211,66';
  end;
end;
```

Al ejecutar el programa puede apreciarse lo mal que quedan los datos en pantalla, sobre todo la columna del importe pendiente:

DANDO FORMATO A LAS CELDAS DE UN COMPONENTE STRINGGRIND

Lo que vamos a hacer a continuación es lo siguiente:

- La primera fila fija va a ser de color de fondo azul oscuro con fuente blanca y además el texto va a ir centrado.

- La columna del importe pendiente va a tener la fuente de color verde y va a ir alineada a la derecha.

- El resto de columnas tendrán el color de fondo blanco y el texto en negro.

Todo esto hay que hacerlo en el evento OnDrawCell del componente StringGrid:

```
procedure TFormulario.StringGridDrawCell( Sender: TObject; ACol,
  ARow: Integer; Rect: TRect; State: TGridDrawState );
var
  sTexto: String;           // Texto que va a imprimir en la celda
actual
  Alineacion: TAlignment;   // Alineación que le vamos a dar al texto
  iAnchoTexto: Integer;     // Ancho del texto a imprimir en pixels
begin
  with StringGrid.Canvas do
  begin
    // Lo primero es coger la fuente por defecto que le hemos asignado
al componente
    Font.Name := StringGrid.Font.Name;
    Font.Size := StringGrid.Font.Size;

    if ARow = 0 then
      Alineacion := taCenter
    else
      // Si es la columna del importe pendiente alineamos el texto a
la derecha
      if ACol = 4 then
        Alineacion := taRightJustify
      else
        Alineacion := taLeftJustify;

    // ¿Es una celda fija de sólo lectura?
    if gdFixed in State then
    begin
      Brush.Color := clNavy;      // le ponemos azul de fondo
      Font.Color := clWhite;      // fuente blanca
      Font.Style := [fsBold];     // y negrita
    end
    else
    begin
      // ¿Esta enfocada la celda?
      if gdFocused in State then
      begin
        Brush.Color := clRed;     // fondo rojo
        Font.Color := clWhite;    // fuente blanca
        Font.Style := [fsBold];   // y negrita
      end
      else
      begin
        // Para el resto de celdas el fondo lo ponemos blanco
        Brush.Color := clWindow;

        // ¿Es la columna del importe pendiente?
        if ACol = 4 then
        begin
          Font.Color := clGreen;   // la pintamos de azul
          Font.Style := [fsBold];  // y negrita
          Alineacion := taRightJustify;
        end
        else
```

```
        begin
          Font.Color := clBlack;
          Font.Style := [];
        end;
      end;
    end;

    sTexto := StringGrid.Cells[ACol,ARow];
    FillRect( Rect );
    iAnchoTexto := TextWidth( sTexto );

    case Alineacion of
      taLeftJustify: TextOut( Rect.Left + 5, Rect.Top + 2, sTexto );
      taCenter: TextOut( Rect.Left + ( ( Rect.Right - Rect.Left ) -
iAnchoTexto ) div 2, Rect.Top + 2, sTexto );
      taRightJustify: TextOut( Rect.Right - iAnchoTexto - 2, Rect.Top
+ 2, sTexto );
    end;
  end;
end;
```

Así quedaría al ejecutarlo:

Sólo hay un pequeño inconveniente y es que la rejilla primero se pinta de manera normal y luego nosotros volvemos a pintarla encima con el evento OnDrawCell con lo cual hace el proceso dos veces. Si queremos que sólo se haga una vez hay que poner a False la propiedad DefaultDrawing. Quedaría de la siguiente manera:

Por lo demás creo que este componente que puede sernos muy útil para mostrar datos por pantalla en formato de sólo lectura. En formato de escritura es algo flojo porque habría que controlar que tipos de datos puede escribir el usuario según en que columnas esté.

Pruebas realizadas en Delphi 7.

Funciones y procedimientos para fecha y hora (I)

Si hay algo de lo que no podemos escapar los programadores de gestión es el tener que lidiar con campos de fecha y hora tales como cálculo de días entre fechas, averiguar en que fecha caen los días festivos, cronometrar el tiempo que tardan nuestras rutinas, etc.

Para la mayoría de funciones tenemos que añadir la unidad DateUtils:

```
uses
  Windows, Messages, ..., DateUtils;
```

Veamos entonces que funciones nos pueden ser útiles para ello:

function DayOfTheMonth(const AValue: TDateTime): Word;

Esta función extrae el número de día de una fecha en concreto sin tener que utilizar la función DecodeDate. Por ejemplo:

```
var Fecha: TDate;
begin
  Fecha := StrToDate( '23/08/2007' );
  ShowMessage( IntToStr( DayOfTheMonth( Fecha ) ); // muestra 23
end;
```

function DayOfTheWeek(const AValue: TDateTime): Word;

Nos díce en que día de la semana (1-7) cae una fecha en concreto (el primer día es el Lunes, al contrario de otras funciones de fecha y hora donde el primer día es el Domingo). Por ejemplo:

```
Fecha := StrToDate( '5/08/2007' );
ShowMessage( IntToStr( DayOfTheWeek( Fecha ) ) );  // devuelve 7
(Domingo)
Fecha := StrToDate( '10/08/2007' );
ShowMessage( IntToStr( DayOfTheWeek( Fecha ) ) );  // devuelve 5
(Viernes)
Fecha := StrToDate( '23/08/2007' );
ShowMessage( IntToStr( DayOfTheWeek( Fecha ) ) );  // devuelve 4
(Jueves)
```

function DayOfWeek(Date: TDateTime): Integer;

Esta función es igual a la función DayOfTheWeek con la diferencia de que el primer día de la semena es el Domingo. Por ejemplo:

```
Fecha := StrToDate( '5/08/2007' );
ShowMessage( IntToStr( DayOfWeek( Fecha ) ) ); // devuelve 1 (Domingo)
Fecha := StrToDate( '10/08/2007' );
ShowMessage( IntToStr( DayOfWeek( Fecha ) ) ); // devuelve 6 (Viernes)
Fecha := StrToDate( '23/08/2007' );
ShowMessage( IntToStr( DayOfWeek( Fecha ) ) ); // devuelve 5 (Jueves)
```

function DayOfTheYear(const AValue: TDateTime): Word;

Al pasarle una fecha nos devuelve el número de día que corresponde a lo largo del año. Por ejemplo:

```
Fecha := StrToDate( '5/08/2007' );
ShowMessage( IntToStr( DayOfTheYear( Fecha ) ) ); // devuelve 217
```

```
Fecha := StrToDate( '10/08/2007' );
ShowMessage( IntToStr( DayOfTheYear( Fecha ) ) ); // devuelve 222
Fecha := StrToDate( '23/08/2007' );
ShowMessage( IntToStr( DayOfTheYear( Fecha ) ) ); // devuelve 235
```

function DaysBetween(const ANow, AThen: TDateTime): Integer;

Devuelve el número de días que hay entre las fechas ANow y AThen. Por ejemplo:

```
var
  Fecha1, Fecha2: TDate;
begin
  Fecha1 := StrToDate( '10/08/2007' );
  Fecha2 := StrToDate( '23/08/2007' );
  ShowMessage( IntToStr( DaysBetween( Fecha1, Fecha2 ) ) ) ; //
devuelve 13
end;
```

function DaysInMonth(const AValue: TDateTime): Word;

Nos dice cuantos días tiene el mes que le pasamos como fecha. Por ejemplo:

```
DaysInMonth( StrToDate( '01/01/2007' ) ) devuelve 31
DaysInMonth( StrToDate( '01/02/2007' ) ) devuelve 28
DaysInMonth( StrToDate( '01/04/2007' ) ) devuelve 30
```

function DaysInYear(const AValue: TDateTime): Word;

Nos dice el número de días que tiene un año según la fecha que le pasamos. Por ejemplo:

```
DaysInYear( StrToDate( '01/01/2007' ) ) devuelve 365
```

function DaySpan(const ANow, AThen: TDateTime): Double;

Devuelve el número de días de diferencia entre dos fechas y horas incluyendo la parte fraccional. Por ejemplo:

```
var
  FechaHora1, FechaHora2: TDateTime;
begin
  FechaHora1 := StrToDateTime( '10/08/2007 13:00' );
  FechaHora2 := StrToDateTime( '23/08/2007 19:00' );
  ShowMessage( FloatToStr( DaySpan( FechaHora1, FechaHora2 ) ) ); //
devuelve 13,25
end;
```

procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);

Este procedimiento convierte un valor de fecha TDate en valores enteros para el año, mes y día. Por ejemplo:

```
var
  wAnyo, wMes, wDia: Word;
begin
```

```
  DecodeDate( StrToDate( '23/08/2007' ), wAnyo, wMes, wDia );
  Memo.Lines.Add( 'Día: ' + IntToStr( wDia ) );
  Memo.Lines.Add( 'Mes: ' + IntToStr( wMes ) );
  Memo.Lines.Add( 'Año: ' + IntToStr( wAnyo ) );
end;
```

Al ejecutarlo mostraría en el campo memo:

```
Día: 23
Mes: 8
Año: 2007
```

procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);

Este procedimiento convierte un valor de hora TTime en valores enteros para la hora, minutos, segundos y milisegundos. Por ejemplo:

```
var
  wHora, wMinutos, wSegundos, wMilisegundos: Word;
begin
  DecodeTime( StrToTime( '14:25:37' ), wHora, wMinutos, wSegundos,
wMilisegundos );
  Memo.Lines.Add( IntToStr( wHora ) + ' horas' );
  Memo.Lines.Add( IntToStr( wMinutos ) + ' minutos' );
  Memo.Lines.Add( IntToStr( wSegundos ) + ' segundos' );
  Memo.Lines.Add( IntToStr( wMilisegundos ) + ' milisegundos' );
end;
```

Al ejecutar este código mostraría:

```
14 horas
25 minutos
37 segundos
0 milisegundos
```

En el siguiente artículo seguiremos con más funciones para fecha y hora.

Pruebas realizadas en Delphi 7.

Funciones y procedimientos para fecha y hora (II)

Vamos a seguir viendo funciones para fecha y hora:

procedure DecodeDateTime(const AValue: TDateTime; out AYear, AMonth, ADay, AHour, AMinute, ASecond, AMilliSecond: Word);

Este procedimiento decodifica un valor TDateTime en valores enteros para el año, mes, día, hora, minutos, segundos y milisegundos. Por ejemplo:

```
var
  wAnyo, wMes, wDia: Word;
  wHora, wMinutos, wSegundos, wMilisegundos: Word;
begin
  DecodeDateTime( StrToDateTime( '24/08/2007 12:34:53' ), wAnyo, wMes,
wDia, wHora, wMinutos, wSegundos, wMilisegundos );
  Memo.Lines.Add( 'Día: ' + IntToStr( wDia ) );
  Memo.Lines.Add( 'Mes: ' + IntToStr( wMes ) );
  Memo.Lines.Add( 'Año: ' + IntToStr( wAnyo ) );
  Memo.Lines.Add( IntToStr( wHora ) + ' horas' );
  Memo.Lines.Add( IntToStr( wMinutos ) + ' minutos' );
  Memo.Lines.Add( IntToStr( wSegundos ) + ' segundos' );
  Memo.Lines.Add( IntToStr( wMilisegundos ) + ' milisegundos' );
end;
```

Al ejecutarlo mostraría:

```
Día: 24
Mes: 8
Año: 2007
12 horas
34 minutos
53 segundos
0 milisegundos
```

function EncodeDate(Year, Month, Day: Word): TDateTime;

Convierte los valores enteros de año, mes y día a un valor TDateTime o TDate. Por ejemplo:

```
EncodeDate( 2007, 8, 24 ) devuelve 24/08/2007
```

function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;

Convierte los valores enteros de hora, minutos, segundos y milisegundos en un valor TDateTime o TTime. Por ejemplo:

```
EncodeTime( 12, 34, 47, 0 ) devuelve 12:34:47
```

function EncodeDateTime(const AYear, AMonth, ADay, AHour, AMinute, ASecond, AMilliSecond: Word):TDateTime;

Convierte los valores enteros de año, mes, día, hora, minutos, segundos y milisegundos en un valor TDateTime. Por ejemplo:

```
EncodeDateTime( 2007, 8, 24, 12, 34, 47, 0 ) devuelve 24/08/2007
12:34:47
```

function EndOfADay(const AYear, ADayOfYear: Word): TDateTime; overload;
function EndOfADay(const AYear, AMonth, ADay: Word): TDateTime; overload;

Devuelve un valor TDateTime que representa la última fecha y hora de un día en concreto. Por ejemplo:

```
EndOfADay( 2007, 8, 13 ) devuelve 13/08/2007 23:59:59
EndOfADay( 2007, 1, 1 )  devuelve 01/01/2007 23:59:59
EndOfADay( 2007, 2, 1 )  devuelve 01/02/2007 23:59:59
EndOfADay( 2007, 8, 23 ) devuelve 08/23/2007 23:59:59
EndOfADay( 2007, 1 )     devuelve 01/01/2007 23:59:59
EndOfADay( 2007, 32 )    devuelve 01/02/2007 23:59:59
```

NOTA IMPORTANTE: La función con tres parámetros es erronea en la versión de Delphi 7 Build 4.453. Da los siguientes valores:

```
EndOfADay( 2007, 8, 13 )  devuelve 12/09/2007 23:59:59 (mal)
EndOfADay( 2007, 1, 1 )   devuelve 31/01/2007 23:59:59 (mal)
EndOfADay( 2007, 2, 1 )   devuelve 28/02/2007 23:59:59 (mal)
EndOfADay( 2007, 8, 23 )  devuelve 12/09/2007 23:59:59 (mal)
EndOfADay( 2007, 1 )      devuelve 01/01/2007 23:59:59 (bien)
EndOfADay( 2007, 32 )     devuelve 01/02/2007 23:59:59 (bien)
```

Con lo cual es recomendable actualizarse a una versión superior (yo me he actualizado a la versión Delphi 7.0 Build 8.1 y sigue sin hacerme ni puto caso).

function EndOfAMonth(const AYear, AMonth: Word): TDateTime;

Devuelve un valor TDateTime que representa la última fecha y hora de un mes. Por ejemplo:

```
EndOfAMonth( 2007, 1 ) devuelve 31/01/2007 23:59:59
EndOfAMonth( 2007, 2 ) devuelve 28/02/2007 23:59:59
EndOfAMonth( 2007, 8 ) devuelve 31/08/2007 23:59:59
```

function EndOfAWeek(const AYear, AWeekOfYear: Word; const ADayOfWeek: Word = 7): TDateTime;

Devuelve un valor TDateTime que representa la última fecha y hora de una semana. El parámetro AWeekOfYear representa el número de semana a lo largo del año. Por ejemplo:

```
EndOfAWeek( 2007, 1 ) devuelve 07/01/2007 23:59:59
EndOfAWeek( 2007, 2 ) devuelve 14/01/2007 23:59:59
EndOfAWeek( 2007, 3 ) devuelve 21/01/2007 23:59:59
```

function EndOfAYear(const AYear): TDateTime;

Devuelve un valor TDateTime que representa la última fecha y hora del año que le pasemos. Por ejemplo:

```
EndOfAYear( 2007 ) devuelve 31/12/2007 23:59:59
EndOfAYear( 2008 ) devuelve 31/12/2008 23:59:59
EndOfAYear( 2009 ) devuelve 31/12/2009 23:59:59
```

function IncDay(const AValue: TDateTime; const ANumberOfDays: Integer = 1): TDateTime;

Devuelve el valor fecha y hora AValue incrementando el número de días especificado en ANumberOfDays (por defecto 1). Por ejemplo:

```
IncDay( StrToDate( '24/08/2007' ) )     devuelve 25/08/2007
IncDay( StrToDate( '24/08/2007' ), 10 ) devuelve 03/09/2007
```

function IncMonth(const Date: TDateTime; NumberOfMonths: Integer = 1): TDateTime;

Devuelve el valor fecha y hora AValue incrementando el número de meses especificado en NumberOfMonths (por defecto 1). Por ejemplo:

```
IncMonth( StrToDate( '24/08/2007' ) )    devuelve 24/09/2007
IncMonth( StrToDate( '24/08/2007' ), 3 ) devuelve 24/11/2007
```

procedure IncAMonth(var Year, Month, Day: Word; NumberOfMonths: Integer = 1);

Este procedimiento es similar a la función IncMonth pero en vez de pasarle la fecha en formato TDateTime se le pasa en formato año, mes y día. Por ejemplo:

```
var
  wAnyo, wMes, wDia: Word;
begin
  wDia := 24;
  wMes := 8;
  wAnyo := 2007;
  IncAMonth( wAnyo, wMes, wDia );
  Memo.Lines.Add( 'Día: ' + IntToStr( wDia ) );
  Memo.Lines.Add( 'Mes: ' + IntToStr( wMes ) );
  Memo.Lines.Add( 'Año: ' + IntToStr( wAnyo ) );
end;
```

Al ejecutarlo obtenemos:

```
Día: 24
Mes: 9
Año: 2007
Día: 1
Mes: 4
Año: 2007
```

function IncWeek(const AValue: TDateTime; const ANumberOfWeeks: Integer = 1): TDateTime;

Devuelve el valor fecha y hora AValue incrementando el número de semanas especificado en ANumberOfWeeks (por defecto 1). Por ejemplo:

```
IncWeek( StrToDate( '24/08/2007' ) )    devuelve 31/08/2007
IncWeek( StrToDate( '24/08/2007' ), 2 ) devuelve 07/09/2007
```

function IncYear(const AValue: TDateTime; const ANumberOfYears: Integer = 1): TDateTime;

Devuelve el valor fecha y hora AValue incrementando el número de años especificado en ANumberOfYears (por defecto 1). Por ejemplo:

```
IncYear( StrToDate( '24/08/2007' ) )    devuelve 24/08/2008
IncYear( StrToDate( '24/08/2007' ), 2 ) devuelve 24/08/2009
```

En el próximo artículo veremos más funciones interesantes.

Pruebas realizadas en Delphi 7.

Funciones y procedimientos para fecha y hora (III)

Sigamos con las funciones para el tratamiento de TTime, TDate y TDateTime:

function IsLeapYear(Year: Word): Boolean;

Esta función nos dice si un año es bisiesto. Por ejemplo:

```
IsLeapYear( 2004 ) devuelve True
IsLeapYear( 2006 ) devuelve False
IsLeapYear( 2007 ) devuelve False
IsLeapYear( 2008 ) devuelve True
IsLeapYear( 2009 ) devuelve False
```

function MonthOfTheYear(const AValue: TDateTime): Word;

Devuelve el número de mes de un año a partir de un valor TDateTime (evita el tener que utilizar DecodeTime). Por ejemplo:

```
MonthOfTheYear( StrToDate( '20/01/2007' ) ) devuelve 1
MonthOfTheYear( StrToDate( '27/08/2007' ) ) devuelve 8
MonthOfTheYear( StrToDate( '31/12/2007' ) ) devuelve 12
```

function Now: TDateTime;

Esta función dos devuelve la fecha y hora actual del sistema (reloj de Windows). Por ejemplo:

```
Now devuelve 27/08/2007 10:05:01
```

function RecodeDate(const AValue: TDateTime; const AYear, AMonth, ADay: Word): TDateTime;

Esta función modifica la fecha de un valor TDateTime sin afectar a la hora. Por ejemplo:

```
var
  dt: TDateTime;
```

```
begin
  dt := StrToDateTime( '27/08/2007 15:21:43' );
  ShowMessage( 'Fecha y hora antes de modificar:     ' +
DateTimeToStr( dt ) );
  dt := RecodeDate( dt, 2005, 7, 26 );
  ShowMessage( 'Fecha y hora después de modificadar: ' +
DateTimeToStr( dt ) );
end;
```

Al ejecutarlo muestra:

```
Fecha y hora antes de modificar:   27/08/2007 15:21:43
Fecha y hora después de modificar: 26/07/2005 15:21:43
```

function RecodeTime(const AValue: TDateTime; const AHour, AMinute, ASecond, AMilliSecond: Word): TDateTime;

Modifica la hora de un valor TDateTime sin afectar a la fecha. Por ejemplo:

```
var
  dt: TDateTime;
begin
  dt := StrToDateTime( '27/08/2007 15:21:43' );
  ShowMessage( 'Fecha y hora antes de modificar:   ' +  DateTimeToStr(
dt ) );
  dt := RecodeTime( dt, 16, 33, 14, 0 );
  ShowMessage( 'Fecha y hora después de modificar: ' +  DateTimeToStr(
dt ) );
end;
```

El resultado sería:

```
Fecha y hora antes de modificar:   27/08/2007 15:21:43
Fecha y hora después de modificar: 27/08/2007 16:33:14
```

procedure ReplaceDate(var DateTime: TDateTime; const NewDate: TDateTime);

Este procedimiento modifica la fecha de una variable TDateTime sin afectar a la hora. Por ejemplo:

```
var
  dt: TDateTime;
begin
  dt := StrToDateTime( '27/08/2007 18:15:31' );
  ShowMessage( 'Fecha y hora antes de modificar:   ' +  DateTimeToStr(
dt ) );
  ReplaceDate( dt, StrToDate( '01/09/2007' ) );
  ShowMessage( 'Fecha y hora después de modificar: ' +  DateTimeToStr(
dt ) );
end;
```

El resultado sería:

```
Fecha y hora antes de modificar:   27/08/2007 18:15:31
Fecha y hora después de modificar: 01/09/2007 18:15:31
```

procedure ReplaceTime(var DateTime: TDateTime; const NewTime: TDateTime);

Modifica la hora de una variable TDateTime sin afectar a la fecha. Por ejemplo:

```
var
  dt: TDateTime;
begin
  dt := StrToDateTime( '27/08/2007 19:33:22' );
  ShowMessage( 'Fecha y hora antes de modificar:   ' +  DateTimeToStr(
dt ) );
  ReplaceTime( dt, StrToTime( '14:21:05' ) );
  ShowMessage( 'Fecha y hora después de modificar: ' +  DateTimeToStr(
dt ) );
end;
```

El resultado sería:

```
Fecha y hora antes de modificar:   27/08/2007 19:33:22
Fecha y hora después de modificar: 27/08/2007 14:21:05
```

function Time: TDateTime;
function GetTime: TDateTime;

Estas dos funciones devuelven la hora actual en formato TDateTime. Por ejemplo:

```
ShowMessage( 'Time devuelve ' +  TimeToStr( Time ) );
ShowMessage( 'GetTime devuelve ' +  TimeToStr( GetTime ) );
ShowMessage( 'Time devuelve ' +  DateTimeToStr( Time ) );
ShowMessage( 'GetTime devuelve ' +  DateTimeToStr( GetTime ) );
```

Al ejecutarlo muestra:

```
Time devuelve 10:36:10
GetTime devuelve 10:36:10
Time devuelve 30/12/1899 10:36:10
GetTime devuelve 30/12/1899 10:36:10
```

Si os fijais bien cuando mostramos la hora en formato TDateTime vemos que la fecha esta nula (30/12/1899). Mucho cuidado con eso si no quereis que el usuario se quede con cara de flipado, sobre todo al guardar o cargar el valor de un campo de la base de datos.

function Tomorrow: TDateTime;

Nos devuelve la fecha de mañana. Por ejemplo:

```
ShowMessage( 'DateToStr( Tomorrow ) devuelve ' + DateToStr( Tomorrow )
);
ShowMessage( 'DateToStr( Tomorrow ) devuelve ' + DateTimeToStr(
Tomorrow ) );
ShowMessage( 'DateToStr( Tomorrow ) devuelve ' + TimeToStr( Tomorrow )
);
```

Nos mostraría:

```
DateToStr( Tomorrow ) devuelve 28/08/2007
DateToStr( Tomorrow ) devuelve 28/08/2007
DateToStr( Tomorrow ) devuelve 0:00:00
```

Como vemos en el ejemplo sólo nos da el día de mañana, pero no la hora (0:00:00).

function Yesterday: TDateTime;

Nos devuelve el día de ayer en formato TDateTime. Por ejemplo:

```
DateToStr( Yesterday ) devuelve 26/08/2007
```

Las siguientes funciones ya las hemos visto anteriormente:

function DateToStr(Date: TDateTime): string; overload;
function TimeToStr(Time: TDateTime): string; overload;
function DateTimeToStr(DateTime: TDateTime): string; overload;
function StrToDate(const S: string): TDateTime; overload;
function StrToTime(const S: string): TDateTime; overload;
function StrToDateTime(const S: string): TDateTime; overload;

Pruebas realizadas en Delphi 7.

Implementando interfaces en Delphi (I)

Delphi es un lenguaje que utiliza la herencia simple al contrario de C++ que permite herencia múltiple. Esto significa que cualquier clase sólo puede heredar de una clase padre. Por lo tanto, si queremos que una clase herede métodos de más de una clase entonces hay que utilizar interfaces (interface).

Una interfaz es como una clase que contiene sólo métodos abstractos (métodos sin implentación) definiendo limpiamente su funcionalidad. Por convención, los nombres de las interfaces comienzan con la letra mayúscula I.

Veamos un ejemplo de definición de interfaz para un artículo:

```
type
  IArticulo = interface
    procedure IncrementarExistencias( rCantidad: Real );
    procedure DecrementarExistencias( rCantidad: Real );
```

```
    function Facturar: Integer;
  end;
```

Las interfaces nunca pueden ser instanciadas. Es decir, no se puede hacer:

```
var
  Articulo: IArticulo;
begin
  Articulo := IArticulo.Create;
end;
```

Al compilar nos daría el error:

Object or class typed required (se requiere una clase u objeto)

Para utilizar una interfaz necesitamos implementarla a través de una clase. Se haría de la siguiente manera:

```
type
  TArticulo = class( TInterfacedObject, IArticulo )
    procedure IncrementarExistencias( rCantidad: Real );
    procedure DecrementarExistencias( rCantidad: Real );
    function Facturar: Integer;
```

¿Que ventajas aporta esto respecto a una clase abstracta? Pues en este caso la definición es más limpia, nada más. Ahora veremos que ventaja de utilizar interfaces.

UTILIZANDO EL POLIMORFIRMO PARA LA CREACION DE CLASES

Las clases polimórficas son aquellas clases que teniendo una definición común se pueden utilizar para distintos tipos de objeto. Supongamos que tenemos la siguiente interfaz:

```
IVehiculo = interface
  procedure Matricular;
end;
```

Utilizando la misma interfaz vamos a definir una clase para cada tipo de vehículo:

```
type
  TCoche = class( TInterfacedObject, IVehiculo )
    procedure Matricular;
  end;

  TCamion = class( TInterfacedObject, IVehiculo )
    procedure Matricular;
  end;
```

Ahora instanciamos ambas clases utilizando la misma interfaz:

```
var
  Vehiculo: IVehiculo;
begin
  Vehiculo := TCoche.Create;
```

```
  Vehiculo.Matricular;
  Vehiculo := TCamion.Create;
  Vehiculo.Matricular;
end;
```

En este caso todas las clases que implementan la interfaz IVehiculo tienen un método común pero podría darse el caso de que tuvieran métodos distintos, lo cual significa que hay que utilizar la herencia múltiple.

UTILIZANDO HERENCIA MULTIPLE MEDIANTE INTERFACES

Supongamos que tenemos estas dos interfaces:

```
type
  IVehiculo = interface
    procedure Matricular;
  end;

  IRemolque = interface
    procedure Cargar;
  end;
```

El vehículo tiene el método Matricular y el remolque tiene el método Cargar. A mi me interesa que un coche utilice el método Matricular, pero sólo el camión tiene que poder Cargar. Entonces la implementación de las clases de haría así:

```
  TCoche = class( TInterfacedObject, IVehiculo )
    procedure Matricular;
  end;

  TCamion = class( TInterfacedObject, IVehiculo, IRemolque )
    procedure Matricular;
    procedure Cargar;
  end;
```

Como puede apreciarse la clase TCoche sólo implementa la interfaz IVehiculo pero la clase TCamion hereda implementa simultáneamente las interfaces IVehiculo y IRemolque pudiendo utilizar sus métodos indistintamente. Esto es lo que se conoce como herencia múltiple.

En el próximo artículo seguiremos viendo más características sobre las interfaces.

Pruebas realizadas en Delphi 7.

Implementando interfaces en Delphi (II)

Cuando creamos clases para nuestros programas de gestión uno se pregunta que ventajas pueden aportar las interfaces dentro de nuestro programa. Pues una de las ventajas es que encapsula aún más nuestra clase dejando sólo accesible los métodos y no las variables internas (ya sean private, protected o public). Por ejemplo, supongamos que deseo implementar una interfaz y una clase para el control de mis clientes:

```
type
  ICliente = interface
    function Get_ID: Integer;
    function Get_Nombre: string;
    function Get_NIF: string;
    function Get_Saldo: Real;
    procedure Set_ID( ID: Integer );
    procedure Set_Nombre( sNombre: string );
    procedure Set_NIF( sNIF: string );
    procedure Set_Saldo( rSaldo: Real );
  end;

  TCliente = class( TInterfacedObject, ICliente )
  private
    ID: Integer;
    sNombre, sNIF: string;
    rSaldo: Real;
  public
    function Get_ID: Integer;
    function Get_Nombre: string;
    function Get_NIF: string;
    function Get_Saldo: Real;
    procedure Set_ID( ID: Integer );
    procedure Set_Nombre( sNombre: string );
    procedure Set_NIF( sNIF: string );
    procedure Set_Saldo( rSaldo: Real );
  end;
```

Esta clase se puede instanciar de dos formas. Si se hace en una variable de clase:

```
var
  Cliente: TCliente;
begin
  Cliente := TCliente.Create;
  Cliente.Set_ID( 1 );
  Cliente.Set_Nombre( 'CARLOS MARTINEZ RUIZ' );
  Cliente.Set_NIF( '67876453F' );
  Cliente.Set_Saldo( 145.87 );
end;
```

entonces se comporta como una clase normal, permitiendo acceder incluso a las variables privadas cuando en el editor de Delphi ponemos:

```
Cliente.
```

y esperamos a que el asistente nos muestre las propiedades y métodos de la clase. Y no sólo eso, sino que además nos muestra todas propiedades y métodos que hemos heredado de TObject haciendo más engorrosa la búsqueda de variables y métodos.

Pero si implementamos la clase a partir de la interfaz:

```
var
  Cliente: ICliente;
begin
  Cliente := TCliente.Create;
```

```
  Cliente.Set_ID( 1 );
  Cliente.Set_Nombre( 'CARLOS MARTINEZ RUIZ' );
  Cliente.Set_NIF( '67876453F' );
  Cliente.Set_Saldo( 145.87 );
end;
```

al teclear en el editor de Delphi:

```
Cliente.
```

no sólo hemos eliminado las variables privadas de la lista, sino que además sólo muestra nuestros métodos y los de las interfaz (QueryInterface, _AddRef y _Release). Esto aporta mucha mayor claridad al programador a la hora de instanciar sus clases, sobre todo cuando tenemos que distribuir nuestra librería a otros programadores (ellos sólo tienen que fijarse como se utiliza la interfaz, ni siquiera tienen porque saber como está implementada la clase).

La única desventaja es que hay que crear tantas funciones y procedimientos Set y Get como propiedades tenga nuestra clase. Pero es bueno acostumbrarse a hacerlo así como ocurre con los Beans de Java.

USANDO INTERFACES COMO PARAMETROS EN PROCEDIMIENTOS

Utilizando el polimorfismo mediante interfaces ahora podemos crear procedimientos genéricos que manejen objetos de distinta clase de una manera simple. Usando las interfaces IArticulo e IVehiculo definidas en artículo anterior podemos escribir los siguientes procedimientos:

```
procedure FacturarArticulos( Articulos: array of IArticulo );
var
  i: Integer;
begin
  for i := Low( Articulos ) to High( Articulos ) do
    Articulos[i].Facturar;
end;

procedure MatricularVehiculos( Vehiculos: array of IVehiculo );
var
  i: Integer;
begin
  for i := Low( Vehiculos ) to High( Vehiculos ) do
    Vehiculos[i].Matricular;
end;
```

El procedimiento FacturarArticulos no tiene porque saber como se factura internamente cada artículo. Lo mismo ocurre con el procedimiento MatricularVehiculos, ya sea de la clase TCoche o TCamion ya se encargará internamente el objeto de llamar a su método correspondiente, abstrayéndonos a nosotros de como funciona internamente cada clase.

LA INTERFAZ IINTERFACE

Al igual que todos los objetos heredan directa o indirectamente de TObject, todas las interfaces heredan de la interfaz IInterface. Esta interfaz incorpora

el método QueryInterface el cual es muy útil para descubrir y usar otras interfaces que implementan el mismo objeto.

Para contar el número de referencias introduce también los métodos _AddRef y _Release. El compilador de Delphi automáticamente proporciona llamadas a estos métodos cuando las interfaces son utilizadas. Para no tener que implementar nosotros a mano estos métodos (ya que una interfaz nos obliga a implementar todos sus métodos), para ello heredamos de la clase TInterfaceObject que proporciona una implementación base para interfaces. Heredar de TInterfaceObject no es obligatorio pero muy útil.

La clase TInterfacedObject está declarada en la unidad System de la siguiente manera:

```
type
  TInterfacedObject = class ( TObject, IInterface )
  protected
    FRefCount: Integer;
    function QueryInterface( const IID: TGUID; out Obj ): HResult;
stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer; read FRefCount;
  end;
```

Por ello en los ejemplos que he mostrado heredo de esa clase aparte de la interfaz:

```
type
  TCliente = class( TInterfacedObject, ICliente )
  ...
```

Como puede apreciarse la clase TInterfacedObject hereda de la clase TObject y de la interfaz IUnknown. La interfaz IUnknown es utilizada para crear objetos COM.

En el próximo artículo terminaremos de ver las características más importantes de las interfaces.

Pruebas realizadas en Delphi 7.

Implementando interfaces en Delphi (III)

Una de las propiedades más interesantes que incorporan las interfaces es la de añadir un identificador único que la diferencie del resto.

IDENTIFICACION DE INTERFACES

Una interfaz puede tener un identificador unico a nivel global llamado GUID. Este identificador tiene la siguiente forma:

['{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}']

donde cada X es un dígito hexadecimal. Cada GUID es un valor binario de 16 bytes que hace que cada interfaz sea única. Los tipos TGUID y PGUID están declarados dentro de la unidad System del siguiente modo:

```
type
  PGUID = ^TGUID;
  TGUID = record
    D1: Integer;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
```

Este sería un ejemplo de declaración de interfaz con GUID:

```
type
  IFactura = interface
    ['{78EAC667-ED60-443B-B84B-5D7AF161B28B}']
    procedure Calcular;
  end;
```

¿De donde sacamos ese código? Pues cuando en el editor de Delphi se pulsa la combinación de teclas CTRL + MAYÚSCULAS + G entonces nos genera una GUID aleatoria.

CREANDO PROPIEDADES EN LA INTERFAZ

Pese que la declarión de interfaces es similar a la de las clases hay que tener varias cosas presentes:

- No permiten declarar variables internas.

- No pueden ser instanciadas.

- No se permite especificar la visibilidad de los métodos (private, protected o public).

- Las propiedades no pueden apuntar a variables sino sólo a métodos.

Entonces, ¿como podemos definir propiedades? Hay que definirlas utilizando métodos en vez de variables cuando se definen los parámetros read y write de la propiedad. Sería de este modo:

```
type
  IAlbaran = interface
    function Get_Numero: Integer;
    function Get_Importe: Real;
    procedure Set_Numero( iNumero: Integer );
    procedure Set_Importe( rImporte: Real );
    procedure Imprimir;
```

```
    property Numero: Integer read Get_Numero write Set_Numero;
    property Importe: Real read Get_Importe write Set_Importe;
  end;
```

y su implementación a través de la clase:

```
  TAlbaran = class( TInterfacedObject, IAlbaran )
  private
    iNumero: Integer;
    rImporte: Real;

  public
    function Get_Numero: Integer;
    function Get_Importe: Real;
    procedure Set_Numero( iNumero: Integer );
    procedure Set_Importe( rImporte: Real );
    procedure Imprimir;

    property Numero: Integer read Get_Numero write Set_Numero;
    property Importe: Real read Get_Importe write Set_Importe;
  end;

implementation

{ TAlbaran }

function TAlbaran.Get_Importe: Real;
begin
  Result := rImporte;
end;

function TAlbaran.Get_Numero: Integer;
begin
  Result := iNumero;
end;

procedure TAlbaran.Imprimir;
begin
  ShowMessage( 'Imprimiendo...' );
end;

procedure TAlbaran.Set_Importe( rImporte: Real );
begin
  Self.rImporte := rImporte;
end;

procedure TAlbaran.Set_Numero( iNumero: Integer );
begin
  Self.iNumero := iNumero;
end;
```

Esto nos permite utilizar nuestra interfaz sin llamar a los procedimientos Set y Get de cada campo:

```
var
  Albaran: IAlbaran;
begin
  Albaran := TAlbaran.Create;
  Albaran.Numero := 1;
```

```
  Albaran.Importe := 68.21;

  ShowMessage( 'Numero=' + IntToStr( Albaran.Numero ) );
  ShowMessage( 'Importe=' + FloatToStr( Albaran.Importe ) );
end;
```

Con esta comodidad se pueden instanciar clases a partir de interfaces manteniendo la claridad de código tanto dentro como fuera de la implementación.

Con esto finalizamos la parte básica de implementación de interfaces en Delphi.

Pruebas realizadas en Delphi 7.

La potencia de los ClientDataSet (I)

El mundo de la programación de bases de datos cambia tan rápidamente que casi no nos da tiempo a asimilar los nuevos protocolos. Comenzamos con DBase, Clipper, FoxPro, .., hasta hoy en día que tenemos Microsoft SQL Server, Interbase, Firebird, etc.

Luego tenemos el maremagnum de lenguajes de programación donde cada cual se come el acceso a datos a su manera: Java con sus infinitos frameswork tales como hibernate, struts, etc., Ruby con el archiconocido Ruby On Rails que utiliza el paradigma MVC (Modelo, Vista, Controlador), Microsoft a su rollo con ADO y su plataforma Microsoft.NET. Todo eso sin contar con los potentes lenguajes script que no tienen la atención que merecen como son PHP, Ruby, Python, TCL/TK, Groovy, etc. Hasta la mismísima CodeGear nos ha sorprendido con su IDE 3rdRails para programación en Ruby On Rails.

Pero si hay algo que hace que Delphi destaque sobre el resto de entornos de programación es su acceso a múltiples motores de bases de datos utilizando una misma lógica de negocio que abstrae al programador de las rutinas a bajo nivel. En la conocida tecnología de acceso a datos llamada MIDAS.

Las primeras versiones de Delphi contaban con el veterano controlador de bases de datos BDE (Borland Database Engine) que permitía acceder a las clásicas bases de datos DBASE, PARADOX, etc. En la versión 5 de Delphi se incluyeron los componentes IBExpres (IBX) que permitían tener un acceso directo a bases de datos Interbase y Firebird. Fue a partir de Delphi 6 cuando Borland apostó por la tecnología DBExpress, un sistema rápido mediante drivers que utilizando un mismo protocolo permitía conectividad con múltiples motores de bases de datos tales como Interbase, Firebird, Oracle, MySQL, Informix, Microsoft SQL Server, etc. Incluso en su última versión (DBX4) permite acceder a su nuevo motor de bases de datos multiplataforma llamado BlackFish programado integramente en .NET y Java (anteriormente se llamaba JDataStore y estaba programado en Java).

Luego tenemos también otros componentes muy buenos de acceso a datos tales como Zeos, Interbase Objects (IBO), FIBPlus, etc. Y por supuesto el

conocido protocolo de acceso a datos de Microsoft llamado ADO, siendo su última versión ADO.NET la que tiene bastantes posibilidades de convertirse en un estandar para todos los entornos Windows.

Pero si hay un componente de acceso a bases de datos en Delphi que destaque sobre todos los demás ese es el ClientDataSet. Combina la facilidad de acceso a datos a través de la clase TDataSet y la potencia de controlar automáticamente las transacciones al motor de bases de datos, las SQL de consulta, actualización y eliminación así como la conexión y desconexión de las tablas con el servidor haciendo que el programador no tenga que preocuparse de las particularidades del motor de bases de datos.

El componente de la clase TClientDataSet no conecta directamente sobre una base de datos en concreto, si no que utiliza el componente DataSetProvider que actua de intermediario haciendo de puente entre los componentes de bases de datos (IBX,IBO,etc) y nuestra tabla ClientDataSet. El componente ClientDataSet es algo así como una tabla de memoria (como la que tienen los componentes RX) que se trae y lleva datos a las tablas de la base de datos encargándose automáticamente de las transacciones.

LA ESTRUCTURA CORRECTA DE UN PROGRAMA

Para crear una buena aplicación con acceso a bases de datos hay que dividir nuestro programa en tres partes principales:

Capa de acceso a datos: se encarga de conectar con un motor de bases de datos en concreto ya sea con componentes IBX, ADO, BDE, etc.

Lógica de negocio: aquí se definen como son nuestras tablas (CLIENTES, ARTICULOS,etc), los campos que contienen así como el comportamiento al dar de alta registros, modificarlos, realización de cálculos internos, etc. Todo esto lo haremos con componentes de la clase TClientDataSet y TDataSetProvider.

Interfaz de usuario: se compone de los formularios, informes y menús de opciones que va a visualizar el usuario y que estarán internamente conectados con los ClientDataSet.

La interfaz de usuario sólo podrá acceder a la lógica de negocio y esta última sólo a la capa de acceso a datos. Así, si en un futuro queremos cambiar la interfaz de usuario (por ejemplo para Windows Vista) o el motor de base de datos no afectaría al resto de las capas del programa.

Para ello vamos a utilizar los componentes contenedores de la clase TDataModule para alojar la lógica de negocio y el acceso a datos. En nuestro ejemplo crearemos una base de datos de clientes definiendo las tres capas.

CREANDO LA BASE DE DATOS

En este ejemplo voy a crear una base de datos de clientes utilizando el motor

de bases de datos Firebird 2.0 y con los componentes IBX. Las tablas las voy a crear con el programa IBExpert (http://www.ibexpert.com) cuya versión personal es gratis y muy potente. La base de datos se va a llamar BASEDATOS.FDB, pero si haceis las pruebas con Interbase entonces sería BASEDATOS.GDB. No voy a explicar aquí como funciona el programa IBExpert o IBConsole ya que hay en la red información abundate sobre ambos programas.

Creamos la tabla de clientes:

```
CREATE TABLE CLIENTES (
    ID          INTEGER NOT NULL,
    NOMBRE      VARCHAR(100),
    NIF         VARCHAR(15),
    DIRECCION   VARCHAR(100),
    POBLACION   VARCHAR(50),
    CP          VARCHAR(5),
    PROVINCIA   VARCHAR(50),
    IMPORTEPTE  DOUBLE PRECISION,
    PRIMARY KEY (ID)
)
```

Como quiero que el ID sea autonumérico voy a crear un generador:

```
CREATE GENERATOR IDCLIENTE
```

Y un disparador para que cuando demos de alta el registro rellene automáticamente el ID y autoincremente el contador del generador:

```
CREATE TRIGGER CONTADOR_CLIENTES FOR CLIENTES
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  NEW.ID = GEN_ID( IDCLIENTE, 1 );
END
```

NOTA IMPORTANTE: Hay algunas versiones de Delphi 7 que tienen un error en los componentes IBExpress (IBX) que hacen que los componentes ClientDataSet no funcionen correctamente. Recomiendo actualizar los componentes IBX a la versión 7.04 que podeis encontrarla en:

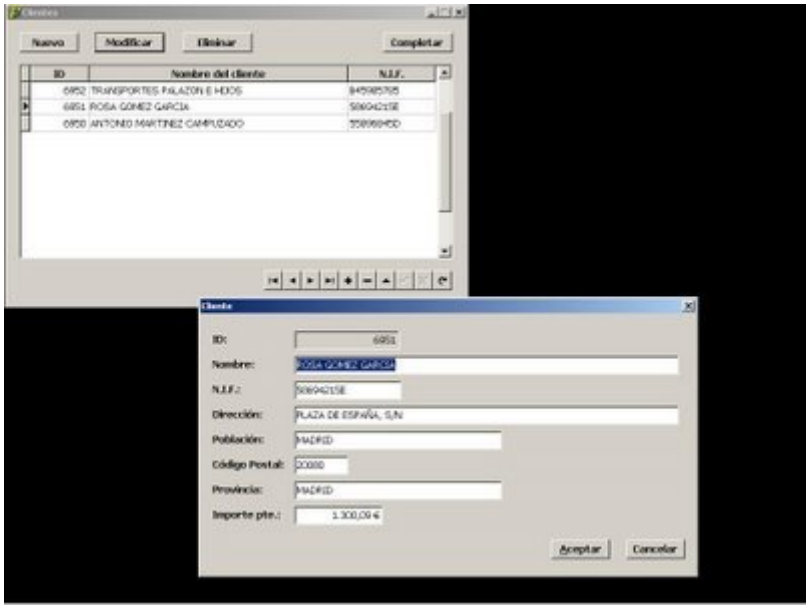
http://codecentral.borland.com/Item.aspx?id=18893

En el siguiente artículo comenzaremos a realizar el programa utilizando esta base de datos.

Pruebas realizadas en Firebird 2.0 y Delphi 7.

La potencia de los ClientDataSet (II)

Después de haber creado la base de datos en Firebird 2.0 ya podemos comenzar a crear un nuevo proyecto que maneje dicha información. El objetivo del proyecto es hacer el siguiente mantenimiento de clientes:



CREANDO LA CAPA DE ACCESO A DATOS

La capa de acceso a datos encargada de conectar con Firebird va a incorporar los siguientes componentes:

- Un contenedor DataModule llamado AccesoDatos.

- Un compomente IBDatabase (pestaña Interbase) llamado BaseDatos.

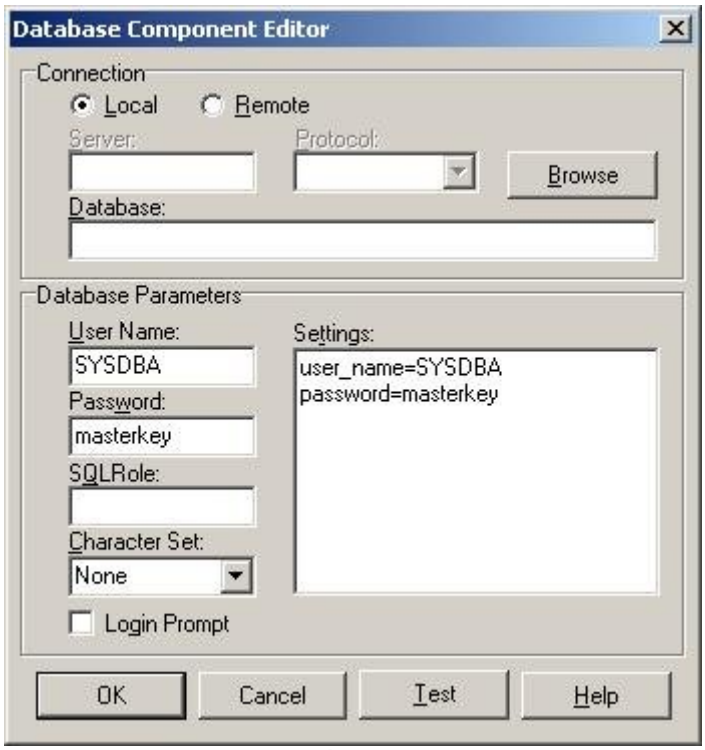
- Un componente IBTransaction (pestaña Interbase) llamado Transaccion.

- Dos componentes IBQuery (pestaña Interbase) llamados LstClientes y Clientes.

Se sopone que la base de datos BASEDATOS.FDB esta al lado de nuestro ejecutable. Vamos a comenzar a configurar cada uno de estos componentes.

Hacemos doble clic sobre el componente IBDatabase y en el campo User

Name le ponemos SYSDBA. En en password le ponemos masterkey:



Pulsamos OK, y después en la propiedad DefaultTransaction del componente IBDatabase seleccionamos Transaccion, desactivamos la propiedad LoginPrompt y nos aseguramos que en SQLDialect ponga un 3.

Para el componente IBTransaction llamado Transaccion seleccionaremos en su propiedad DefaultDatabase la base de datos BaseDatos.

Como nuestra intención es tener una rejilla que muestre el listado general de clientes y un formulario para dar de alta clientes, lo que vamos a hacer es crear una tabla IBQuery para la rejilla llamada LstClientes y otra tabla para el formulario del cliente llamada Clientes. Sería absurdo utilizar un mismo mantenimiento para ambos casos ya que si tenemos miles de clientes en la rejilla, el tener cargados en memoria todos los campos del cliente podría relentizar el programa.

En los componentes LstClientes y Clientes vamos a configurar también:

Database: BaseDatos
Transaction: Transaccion
UniDirectional: True

El motivo de activar el campo UniDirectional es para que el cursor SQL trabaje más rápido en el servidor ya que los componentes ClientDataSet gestionan en memoria los registros leidos anteriormente.

Como en la rejilla sólo quiero mostrar los campos ID, NOMBRE y NIF entonces en el componente LstClientes en su propiedad SQL vamos a definir:

```
SELECT ID,NOMBRE,NIF FROM CLIENTES
ORDER BY ID DESC
```

y para el componente Clientes:

```
SELECT * FROM CLIENTES
WHERE ID=:ID
```

En la condición WHERE de la SQL hemos añadido el parámetro :ID para que se pueda más adelante acceder directamente a un registro en concreto a partir de su campo ID.

Una vez definida nuestra capa de acceso a datos en el siguiente artículo nos encargaremos de definir nuestra lógica de negocio.

Pruebas realizadas en Firebird 2.0 y Delphi 7.

La potencia de los ClientDataSet (III)

Después de crear la base de datos y la capa de acceso a datos dentro del DataModule vamos a crear la lógica de negocio.

Antes de seguir tenemos que hacer doble clic en los componente IBQuery de la capa de acceso a datos y pulsar la combinación de teclas CTRL + A para introducir todos los campos en el módulo de datos. Y en cada una de ellas hay que seleccionar el campo ID y quitar la propiedad Required ya que el propio motor de bases de datos va a meter el campo ID con un disparador.

Pero tenemos un problema, y es que no hemos seleccionado donde esta la base de datos. Para ello hacemos doble clic en el objeto BaseDatos situado módulo de datos AccesoDatos. Seleccionamos una base de datos Remota (Remote) con IP 127.0.0.1. Y la base de datos donde la tengamos, por ejemplo:

D:\Desarrollo\Delphi7\ClientDataSet\BaseDatos.fdb

De este modo, al hacer CTRL + A sobre las tablas conectará automáticamente sobre la base de datos para traerse los campos. No se os olvide luego desconectarla.

CREANDO LA LOGICA DE NEGOCIO

La capa de lógica de negocio también la vamos a implementar dentro de un objeto DataModule y va a constar de los siguientes componentes:

- Un DataModule llamado LogicaNegocio.

- Dos componentes DataSetProvider llamados DSPLstClientes y DSPClientes.

- Dos componentes ClientDataSet llamados TLstClientes y TClientes.

Ahora vamos a configurar cada uno de estos componentes:

- Enlazamos el DataModule LogicaNegocio con el DataModule AccesoDatos en la sección uses.

- Al componente DSPLstClientes le asignamos en su propiedad DataSet el componente AccesoDatos.LstClientes.

- Al componente DSPClientes le asignamos en su propiedad DataSet el componente AccesoDatos.Clientes.

- El componente TLstClientes lo vamos a vincular con DSPLstClientes mediante su propiedad ProviderName.

- El componente TClientes lo vamos a vincular con DSPClientes mediante su propiedad ProviderName.

- Debemos hacer doble clic en ambos ClientDataSets y pulsar la combinación de teclas CTRL + A para meter todos los campos.

CONTROLANDO EL NUMERO DE REGISTROS CARGADOS EN MEMORIA

Los componentes ClientDataSet tienen una propiedad llamada PacketRecord la cual determina cuandos registros se van a almacenar en memoria. Por defecto tiene configurado -1 lo que significa que se van a cargar todos los registros en la tabla. Como eso no me interesa en el listado general del formulario principal lo que vamos a hacer es poner esta propiedad a 100.

Por eso he ordenado la lista por el campo ID descendentemente para que se vean sólo los últimos 100 registros insertados. Una de las cosas que más me gustan de los componentes ClientDataSet es que se trae los 100 últimos registros y desconecta la tabla y la transacción quitándole trabajo al motor de bases de datos. Si el usuario que maneja el programa llega hasta el registro número 100 el propio componente conecta automáticamente con el servidor, se trae otros 100 registros y vuelve desconectar.

Lo único en lo que hay que tener cuidado es no acumular demasiados registros en memoria ya que puede relentizar el programa e incluso el sistema operativo si el PC no tiene mucha potencia.

El componente ClientDataSet llamado TClientes lo dejamos como está en -1 ya que sólo lo vamos a utilizar para dar de alta un registro o modificarlo.

ENVIANDO LAS TRANSACCIONES AL SERVIDOR

Cuando se utilizan los clásicos métodos Insert, Append, Post y Delete con los objetos de la clase TClientDataSet el resultado de las operaciones con registros no tiene lugar en la base de datos hasta que envíamos la transacción al servidor con el método ApplyUpdates.

Por ello vamos a utilizar el evento OnAfterPost para enviar la transacción al servidor en el caso que haya sucedido alguna modificación en el registro:

```
procedure TLogicaNegocio.TClientesAfterPost( DataSet: TDataSet );
begin
  if TClientes.ChangeCount > 0 then
  begin
    TClientes.ApplyUpdates( 0 );
    TClientes.Refresh;

    if TLstClientes.Active then
      TLstClientes.Refresh;
  end;
end;
```

Después de enviar la transacción hemos refrescado la tabla TClientes y también la tabla TLstClientes para que se actualicen los cambios en la rejilla. Por último, cuando en el listado de clientes se elimine un registro también hay que enviar la trasacción al servidor en su evento OnAfterDelete:

```
procedure TLogicaNegocio.TLstClientesAfterDelete( DataSet: TDataSet );
begin
  TLstClientes.ApplyUpdates( 0 );
end;
```

Con esto ya tenemos controlada la inserción, modificación y eliminación de registros hacia el motor de bases de datos.

ESTABLECIENDO REGLAS DE NEGOCIO EN NUESTRAS TABLAS

Las reglas de negocio definidas en el módulo de datos le quitan mucho trabajo al formulario que esté vinculado con la tabla. En un primer ejemplo vamos a hacer que cuando se demos de alta un cliente su importe pendiente sea cero. Esto se hace en el evento OnNewRecord del componente TClientes:

```
procedure TLogicaNegocio.TClientesNewRecord( DataSet: TDataSet );
begin
```

```
  TClientesIMPORTEPTE.AsFloat := 0;
end;
```

Otra de las malas costumbres que solemos cometer en los programas es controlar los datos que introduce o no el usuario en el registro, cuya lógica la hacemos en el formulario. Esto tiene el inconveniente en que si en otro formulario hay que acceder a la misma tabla hay que volver a controlar las acciones del usuario.

Para evitar esto, tenemos que definir también en la capa de lógica de negocio las reglas sobre las tablas y los campos, lo que se llama comunmente validación de campos. El componente ClientDataSet dispone de la propiedad Constraints donde pueden definirse tantas reglas como queramos. Para verlo con un ejemplo, vamos a hacer que el usuario no pueda guardar el cliente si no ha introducido su nombre.

Para definir una regla en un ClientDataSet hay que hacer lo siguiente (con TClientes):

- Pulsamos el botón [...] en la propiedad Constraints.

- Pulsamos el botón Add New.

- En la propiedad CustomConstraint definimos la condición de error mediante SQL:

```
NOMBRE IS NOT NULL
```

- En el campo ErrorMessage del mismo Constraint ponemos el mensaje de error:

```
No ha introducido el nombre del cliente
```

Con esta regla definida, si en cualquier parte de nuestro programa hacemos un Post de la tabla clientes y esta vacío el campo NOMBRE el programa lanzará un mensaje de error sin tener que programar nada. Antes teníamos que hacer esto en el botón Aceptar del formulario para validar los campos. Con los Constraints las validadiones las hacemos en sin tener que programar.

Ahora vamos a definir otra regla la cual establece que un cliene no puede tener un importe pendiente superior a 2000 €. Creamos un nuevo Constraint con la propiedad CustomConstrait definida con:

```
IMPORTEPTE <= 2000
```

y con la propiedad ErrorMessage que tenga:

```
El importe pendiente no puede ser superior a 2000 €
```

Se pueden definir tantas reglas como deseemos. En el próximo artículo vamos a hacer los formularios de mantenimiento de clientes.

La potencia de los ClientDataSet (IV)

Ya tenemos todo lo necesario para realizar el programa:

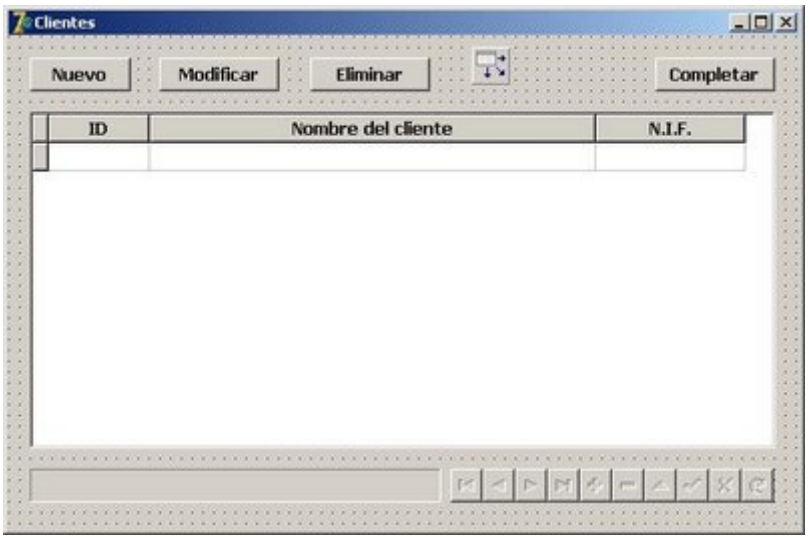
- La base de datos firebird: BASEDATOS.FDB

- La capa de acceso a datos en el módulo de datos: AccesoDatos.

- La lógica de negocio en el módulo de datos: LogicaNegocio.

CREANDO EL FICHERO GENERAL DE CLIENTES

El primer formulario que vamos a crear se va a llamar FClientes y va a contener el listado general de clientes:



Va a contener los siguientes componentes:

- 4 botones de la clase TButton para dar de alta clientes, modificarlos y eliminarlos. Habrá otro llamado BCompletar que utilizaremos más adelante para dar de alta clientes de forma masiva.

- Una rejilla de datos de la clase TDBGrid que va a contener 3 columnas para el listado del cliente: ID, NOMBRE y NIF. Su nombre va a ser ListadoClientes.

- Un componente DataSource (pestaña Data Access) llamado DSLstClientes que va a encargarse de suministrar datos a la rejilla.

- Un componente DBNavitator (pestaña Data Controls) para hacer pruebas con los registros. Lo llamaremos Navegador.

- Una barra de progreso a la izquierda del DBNavigator de la clase TProgressBar que mostrará el progreso de los clientes dados de alta automáticamente a través del botón BCompletar.

Ahora hay que vincular los componentes como corresponde:

- Lo primero es añadir en la sección uses el módulo de datos LogicaNegocio para poder vincular el listado de clientes a la rejilla.

- Vinculamos el componente DSLstClientes con el ClientDataSet llamado TLstClientes a través de su propiedad DataSet.

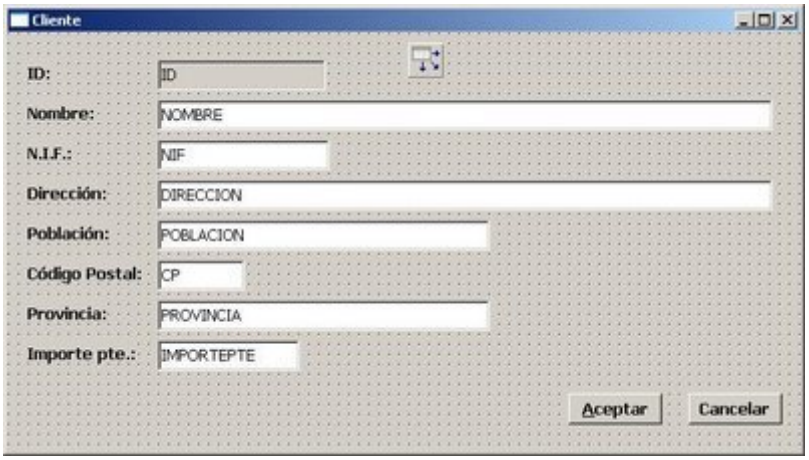
- En la propiedad DataSource de la rejilla de datos ListadoClientes asignamos el componente DSLstClientes.

- Vinculamos el componente Navegador al componente DSLstClientes mediante su propiedad DataSource.

Más adelante escribiremos código para cada uno de los botones.

CREANDO EL FORMULARIO DEL CLIENTE

Vamos a crear el siguiente formulario llamado FCliente:



Va a constar de tantos componentes de la clase TLabel y TDBEdit como campos tenga la tabla clientes. Todos los campos son modificables a excepción del ID que lo he puesto con su propiedad Enabled a False. También lo he oscurecido podiendo de Color el valor clBtnFace.

Para vincular los campos a la tabla real de clientes introducimos un componente DataSource llamado DSClientes, pero antes hay que añadir en la sección uses el módulo de datos LogicaNegocio.

Ahora se vincula el componente DSClientes con el ClientDataSet llamado TClientes situado en el módulo de datos LogicaNegocio a través de su propiedad Dataset. Con esto ya podemos vincular cada campo TDBEdit con el DataSource DSClientes y con su campo correspondiente especificado en la propiedad DataField.

Cuando se pulse el botón Aceptar ejecutamos el código:

```
procedure TFCliente.BAceptarClick( Sender: TObject );
begin
```

```
  LogicaNegocio.TClientes.Post;
  ModalResult := mrOk;
end;
```

Al botón BAceptar le ponemos en su propiedad ModalResult el valor mrNone. Esto tiene su explicación. Cuando pulsemos Aceptar, si el usuario no ha rellenado correctamente algún dato saltará un error definido en los Constraint y no hará nada, evitamos que se cierre el formulario. Así obligamos al usuario a introducir los datos correctamente. Sólo cuando el Post se realice correctamente le diremos al formulario que el resultado es mrOk para que pueda cerrarse.

En el botón BCancelar con sólo asignar en su propiedad ModalResult el valor mrCancel no será necesario hacer nada más.

Sólo una cosa más: si utilizais el teclado numérico en los campos de tipo Float vereis que la tecla decimal no funciona (aquí en España). Nos obliga a utilizar la coma que está encima de la barra de espacio. Para transformar el punto en coma yo lo que suelo hacer es lo siguiente (para el campo IMPORTEPTE):

```
procedure TFCliente.IMPORTEPTEKeyPress( Sender: TObject; var Key: Char
);
begin
  if key = '.' then
    key := ',';
end;
```

Es decir, en el evento OnKeyPress cuando el usuario pulse el punto lo transformo en una coma. Si tuvieramos más campos float sólo habría que reasignar el mismo eventos al resto de campos.

Con esto ya tenemos terminado el formulario del cliente.

TERMINANDO EL FICHERO GENERAL DE CLIENTES

Una vez que tenemos la ficha del cliente vamos a terminar el mantenimiento de clientes asignado el código a cada botón. Comencemos con el botón Nuevo:

```
procedure TFClientes.BNuevoClick( Sender: TObject );
begin
  LogicaNegocio.TClientes.Open;
  LogicaNegocio.TClientes.Insert;
  Application.CreateForm( TFCliente, FCliente );
  FCliente.ShowModal;
  LogicaNegocio.TClientes.Close;
end;
```

Como puede apreciarse abrimos la tabla TClientes sólo cuando hay que dar de alta un cliente o modificarlo. Después la cerramos ya que para ver el listado de clientes tenemos nuestra tabla TLstClientes.

Para el botón Modificar es muy parecido:

```
procedure TFClientes.BModificarClick( Sender: TObject );
begin
  LogicaNegocio.TClientes.Params.ParamByName( 'ID' ).AsString :=
LogicaNegocio.TLstClientesID.AsString;
  LogicaNegocio.TClientes.Open;
  LogicaNegocio.TClientes.Edit;
  Application.CreateForm( TFCliente, FCliente );
  FCliente.ShowModal;
  LogicaNegocio.TClientes.Close;
  ListadoClientes.SetFocus;
end;
```

Aquí hay que detenerse para hablar de algo importante. Cuando abrimos una tabla por primera vez el cursor SQL dentro del motor de bases de datos se va al primer registro. ¿Cómo hacemos para ir al registro seleccionado por TLstClientes? Pues le tenemos que pasar el ID del cliente que queremos editar.

Esto se hace utilizando parámetros, los cuales hay que definirlos dentro del componente ClientDataSet llamado TClientes que está en el módulo de datos LogicaNegocio. Generalmente cuando se pulsa CTRL + A para traernos los campos de la tabla al ClientDataSet se dan de alta los parámetros. Como el componente TClientes está vinculado al IBQuery Clientes que tiene la SQL:

```
SELECT * FROM CLIENTES
WHERE ID=:ID
```

entonces al pulsar CTRL + A en el ClientDataSet nos da de alta automáticamente el parámetro ID. Si no fuera así, tendríamos que ir al componente TClientes, pulsar el botón [...] en su propiedad Params y dar de alta un parámetro con las propiedades:

DataType: ftInteger
Name: ID
ParamType: ptInput

Los parámetros dan mucha velocidad a un programa porque permiten modificar las opciones de la SQL sin tener que cerrar y abrir de nuevo la consulta, permitiendo saltar de un registro a otro dentro de una misma tabla a una velocidad impresionante.

Y por último introducimos el código correspondiente al botón Eliminar:

```
procedure TFClientes.BEliminarClick( Sender: TObject );
begin
  with LogicaNegocio.TLstClientes do
    if RecordCount > 0 then
      if Application.MessageBox( '¿Desea eliminar este cliente?',
'Atención',
        MB_ICONQUESTION or MB_YESNO ) = ID_YES then
        Delete;

  ListadoClientes.SetFocus;
end;
```

Antes de eliminar el registro nos aseguramos que de tenga algún dato y preguntamos al usuario si esta seguro de eliminarlo.

Con esto finalizamos nuestro mantenimiento de clientes a falta de utilizar el botón Completar donde crearemos un bucle que dará de alta tantos clientes como deseemos para probar la velocidad de la base de datos. Esto lo haremos en el próximo artículo.

Pruebas realizadas en Firebird 2.0 y Delphi 7.

La potencia de los ClientDataSet (V)

Antes de proceder a crear el código para el botón Completar vamos a hacer que la conexión del programa con la base de datos sea algo más flexible.

Hasta ahora nuestro objeto IBDatabase está conectado directamente a una ruta fija donde tenemos el archivo BASEDATOS.FDB. Pero si cambiamos de ruta el programa dejará de funcionar provocando un fallo de conexión.

Para evitar esto vamos a hacer que el programa conecte con la base de datos al mostrar el formulario principal FClientes. Para ello en el evento OnShow de dicho formulario ponemos lo siguiente:

```
procedure TFClientes.FormShow( Sender: TObject );
begin
  AccesoDatos.BaseDatos.DatabaseName := '127.0.0.1:' +
ExtractFilePath( Application.ExeName ) + 'BASEDATOS.FDB';

  try
    AccesoDatos.BaseDatos.Open;
  except
    raise;
  end;

  LogicaNegocio.TLstClientes.Active := True;
end;
```

Esto hace que conecte con la base de datos que está al lado de nuestro ejecutable. Así hacemos que nuestro programa sea portable (a falta de instalarle el motor de bases de datos correspondiente). Si la conexión con la base de datos es correcta entonces abrimos la tabla del listado de clientes (TLstClientes).

Cuando se trata de una base de datos Interbase la conexión puede ser local o remota. Si es local no es necesario poner la IP:

```
AccesoDatos.BaseDatos.DatabaseName := 'C:\MiPrograma\BaseDatos.gdb';
```

Aunque viene a ser lo mismo que hacer esto:

```
AccesoDatos.BaseDatos.DatabaseName :=
'127.0.0.1:C:\MiPrograma\BaseDatos.gdb';
```

Se trata de una conexión remota aunque estemos accediendo a nuestro mismo equipo. Si se tratara de otro equipo de la red habría que hacer lo mismo:

```
AccesoDatos.BaseDatos.DatabaseName :=
'192.168.0.1:C:\MiPrograma\BaseDatos.gdb';
```

Para bases de datos Firebird no existe la conexión local, siempre es remota. Así que si vamos a conectar con nuestro equipo en local habría que hacerlo así:

```
AccesoDatos.BaseDatos.DatabaseName :=
'127.0.0.1:C:\MiPrograma\BaseDatos.fdb';
```

Yo recomiendo utilizar siempre la conexión remota utilizando para ello un archivo INI al lado de nuestro programa que contenga la IP del servidor. Por ejemplo:

```
[CONEXION]
IP=127.0.0.1
```

Así podemos hacer que nuestro programa se conecte en local o remoto sin tener que volver a compilarlo. Pero que no se os olvide dejar desconectado el componente IBDatabase en el módulo de acceso a datos AccesoDatos porque si no lo primero que ha a hacer es conectarse al arrancar el programa provocando un error.

DANDO DE ALTA CLIENTES DE FORMA MASIVA

Para probar el rendimiento de un programa no hay nada mejor que darle caña metiendo miles y miles de registros a la base de datos. Para ello voy a crear un procedimiento dentro del botón Completar que le preguntará al usuario cuandos clientes desea crear. Al pulsar Aceptar dará de alta todos esos registros mostrando el progreso en la barra de progreso que pusimos anteriormente:

```
procedure TFClientes.BCompletarClick( Sender: TObject );
var
  sNumero: string;
  i, iInventado: Integer;
begin
  sNumero := InputBox( 'Nº de clientes', 'Rellenando clientes', '' );
  Randomize; // Inicializamos el generador de números aleatorios

  if sNumero <> '' then
  begin
    with LogicaNegocio do
    begin
      TClientes.Open;
      Progreso.Max := StrToInt( sNumero );
      Progreso.Visible := True;
      TLstClientes.DisableControls;
```

```
      for i := 1 to StrToInt( sNumero ) do
      begin
        iInventado := Random( 99999999 ) + 1000000; // Nos invertamos
un número identificador de cliente
        TClientes.Insert;
        TClientesNOMBRE.AsString := 'CLIENTE Nº ' + IntToStr(
iInventado );
        TClientesNIF.AsString := IntToStr( iInventado );
        TClientesDIRECCION.AsString := 'CALLE Nº ' + IntToStr(
iInventado );
        TClientesPOBLACION.AsString := 'POBLACION Nº ' + IntToStr(
iInventado );
        TClientesPROVINCIA.AsString := 'PROVINCIA Nº ' + IntToStr(
iInventado );
        iInventado := Random( 79999 ) + 10000; // Nos inventamos el
código postal
        TClientesCP.AsString := IntToStr( iInventado );
        TClientesIMPORTEPTE.AsFloat := 0;
        TClientes.Post;
        Progreso.Position := i;
        Application.ProcessMessages;
      end;

      TClientes.Close;
      Progreso.Visible := False;
      TLstClientes.EnableControls;
    end;
  end;
end;
```

Lo que hemos hecho es inventar el nombre de usuario, dirección, NIF, etc. También he desconectado y he vuelto a conectar la tabla TLstClientes de la rejilla utilizando los métodos DisableControls y EnableControls para ganar más velocidad. La barra de progreso llamada Progreso mostrará la evolución del alta de registros.

Realmente es una burrada dar de alta masivamente registros utilizando objetos ClientDataSet ya que están pensados para utilizarlos para altas, modificaciones y eliminación de registros de uno a uno y sin mucha velocidad. Si quereis hacer una inserción masiva de registros hay que realizar consultas SQL con INSERT utilizando el objetos de la clase TIBSQL que es mucho más rápido que los ClientDataSet. Ya explicaré en otro momento la ventaja de utilizar dichos componentes.

Con esto finalizamos la introducción a los componentes ClientDataSet.

Pruebas realizadas en Firebird 2.0 y Delphi 7.

Mostrando información en un ListView (I)

Cuando nos conectamos a una base de datos estamos acostumbrados a mostrar la información en un componente TDBGrid, pero hay ocasiones en las que hay que mostrar información temporal que no va conectada a la base de datos. Un ejemplo sería el listar ciertos campos de los albaranes antes de facturar o procesar la información de varias tablas en una sola.

DEFINIENDO COMO SE MUESTRA LA INFORMACIÓN EN UN LISTVIEW

El componente TListView permite mostrar la información de varias maneras diferentes lo cual se especifica en su propiedad ViewStyle cuyas posibilidades son:

```
vsIcon      -> Muestra sus elementos como iconos de tamaño normal
vsList      -> Muestra sus elementos como una lista a varias columnas
vsReport    -> Muestra sus elementos según las filas y columnas al
estilo DBGrid.
vsSmallIcon -> Muestra sus elementos como iconos pequeños
```

En esta ocasión vamos a poner la propiedad ViewStyle a vsReport para que tenga un comportamiento similar a una rejilla de datos al estilo TDBGrid.

A continuación vamos a ver como definir las columnas a mostrar dentro de un ListView. Para ello lo que hacemos es hacer doble clic sobre el ListView y se abrirá una ventana para añadir columnas. Pulsando el botón Add New vamos a añadir las siguientes columnas:

Nº columna	Caption	Width	Alignment
0	Nada	0	taLeft
1	ID	50	taRightJustify
2	Artículo	150	taLeftJustify
3	Unidades	50	taRightJustify
4	Precio	50	taRightJustify
5	Total	50	taRightJustify

La primera columna de un ListView es especialmente utilizada para guardar el título de una fila o una imagen asociada como veremos más adelante. Como la primera columna que quiero mostrar es el ID y la quiero alinear a la derecha entonces no me sirve. Lo que hago es llamar a la primera columna Nada y le doy un ancho de 0 para que no se vea. Mas adelante le daremos utilidad a la misma.

Una vez definidas las columnas vamos a ver como añadir filas.

AÑADIENDO ELEMENTOS AL LISTADO

Todas las columnas que se dan de alta en un objeto ListView son de tipo String, con lo cual si deseamos mostrar otro tipo de información debemos utilizar las típicas rutinas de conversión FloatToStr, BoolToStr, etc.

Veamos un ejemplo de como dar de alta tres artículos:

```
procedure TFormulario.NuevoElemento;
begin
  with ListView.Items.Add do
  begin
    SubItems.Add( '1' );
    SubItems.Add( 'MONITOR LG' );
    SubItems.Add( '3' );
    SubItems.Add( '230,45' );
    SubItems.Add( '691,35' );
  end;

  with ListView.Items.Add do
  begin
    SubItems.Add( '2' );
    SubItems.Add( 'TECLADO LOGITECH' );
    SubItems.Add( '2' );
    SubItems.Add( '49,99' );
    SubItems.Add( '99,98' );
  end;

  with ListView.Items.Add do
  begin
    SubItems.Add( '3' );
    SubItems.Add( 'RATÓN OPTICO DELL' );
    SubItems.Add( '5' );
    SubItems.Add( '15,99' );
    SubItems.Add( '79,95' );
  end;
end;
```

Como se puede apreciar primero se crea un Item por cada fila y un SubItem para cada columna, ignorando la columna 0 donde interviene la propiedad Caption como veremos más adelante. De hecho, SubItem[0] es la segunda columna y no la primera.

MODIFICANDO LAS FILAS DEL LISTADO

El primer elemento introducido en un ListView tiene un índice de 0. Lo que vamos a hacer es cambiar el TECLADO LOGITECH por un GENIUS:

```
procedure TFormulario.ModificarElemento;
begin
  // Modificamos el segundo elemento de la lista
  with ListView.Items[1] do
  begin
    SubItems[1] := 'TECLADO GENIUS';
    SubItems[2] := '7';
    SubItems[3] := '31,99';
    SubItems[4] := '223,93';
  end;
end;
```

Hemos cambiado todos las columnas menos la primera (el ID). Aquí hay que andarse con ojo y no acceder a un elemento de que no exista porque si no provocaría el error que tanto nos gusta:

Access violation at address ....

ELIMINANDO FILAS DE LA LISTA

Eliminar un registro de la lista es algo tan secillo de hacer como:

```
ListView.Items[0].Delete;
```

Eso elimina la primera fila. Y si deseamos eliminar todos los elementos de la lista:

```
ListView.Items.Clear;
```

CAMBIANDO EL ASPECTO Y EL COMPORTAMIENTO DEL LISTADO

En la lista que hemos creado no podemos seleccionar ninguna fila, esta algo así como bloqueado. Para poder hacerlo hay que activar la propiedad RowSelect a True.

Otra característica interesante es la de mostrar un separador de filas y columnas al estilo de la rejilla DBGrid. Para ello activa la propiedad GridLines a True.

También ocurre que si tenemos una fila seleccionada y pasamos el foco a otro control parece que se pierde la selección de dicha fila, aunque realmente sigue seleccionada si volvemos al mismo. Para evitar esto lo que debemos hacer es desactivar la propiedad HideSelection de tal manera que cuando pasemos a otro control en vez de estar la fila seleccionada en azul lo hace en color gris pero sigue estando seleccionada para el usuario.

Y si queremos seleccionar más de una fila activamos la propiedad Multiselect, de tal manera que si hay una fila seleccionada puede saberse de la siguiente manera:

```
if ListView.Selected <> nil then
  ...
```

Siendo Selected el Item de la lista que ha sido seleccionado.

Si hay muchas seleccionadas la propiedad SelCount nos lo dirá el número de filas seleccionadas en todo momento. Para averiguar que filas están seleccionadas lo que hacemos es recorrer la lista y ver aquellos elementos que tienen a True la propiedad Selected:

```
procedure TFormulario.VerSeleccionados;
var
  Seleccionados: TStringList;
  i: Integer;
begin
  Seleccionados := TStringList.Create;
```

```
  for i := 0 to ListView.Items.Count - 1 do
    if ListView.Items[i].Selected then
      Seleccionados.Add( ListView.Items[i].SubItems[1] );

  ShowMessage( Seleccionados.Text );

  Seleccionados.Free;
end;
```

Lo que hace este procedimiento es crear un objeto StringList y volcar dentro del mismo el nombre de los artículos seleccionados para después sacarlo por pantalla.

En el próximo artículo seguiremos viendo que más cosas se pueden hacer con un ListView.

Pruebas realizadas en Delphi 7.

Mostrando información en un ListView (II)

Veamos que más le podemos hacer con el componente TListView.

ORDENANDO LOS ELEMENTOS DE LA LISTA

El objeto ListView dispone de dos tipos de ordenación parecidos. Por un lado tenemos la ordenación mediante la función CustomSort:

function CustomSort(SortProc: TLVCompare; lParam: Longint): Boolean;

Esta función toma como primer parámetro la dirección de una función CALLBACK encargada de establecer los parámetros de la ordenación. Veamos un ejemplo de ordenación ascendente por la columna ARTÍCULO (la segunda):

```
ListView.CustomSort( @Ordenacion, 0 );
```

Donde la función de ordenación sería la siguiente:

```
function Ordenacion( Item1, Item2: TListItem; ParamSort: Integer ):
integer; stdcall;
begin
  Result := CompareText( Item1.SubItems[1], Item2.SubItems[1] );
end;
```

Esta función compara los items 1 y 2 y le devuelve el resultado a CustomSort. Si queremos ordenarla descendentemente sería así:

```
function Ordenacion( Item1, Item2: TListItem; ParamSort: Integer ):
integer; stdcall;
begin
  Result := -CompareText( Item1.SubItems[1], Item2.SubItems[1] );
end;
```

Por otro lado tenemos la función AlphaSort la cual no necesita una función CALLBACK porque para eso tenemos el evento OnCompare. Se haría de la siguiente manera:

```
ListView.AlphaSort;
```

y en el evento OnCompare del ListView:

```
procedure TFormulario.ListViewCompare( Sender: TObject; Item1, Item2:
TListItem;
                                       Data: Integer; var Compare:
Integer );
begin
  Compare := CompareText( Item1.SubItems[1], Item2.SubItems[1] );
end;
```

Prácticamente es parecido a CustomSort (ya que un ListView desciende del componente CustomListView).

UTILIZANDO LA PRIMERA COLUMNA PARA CASOS ESPECIALES

Una de las cosas que se pueden meter en primera columna es un CheckBox para que el usuario marque filas. Antes de eso vamos a volver a mostrar la primera columna (Nada) a 70 de ancho y le cambiamos el nombre a Seleccionado. Por último activamos en las propiedades del ListView el campo CheckBoxes.

Al ejecutar el programa y dar de alta filas en el listado veremos que por cada elemento aparece un CheckBox a la izquierda para poder marcarlo. ¿Cómo sabemos metiante código los elementos que han sido marcados? Con la propiedad Checked de cada Item.

Veamos un ejemplo que comprueba los artículos marcados con CheckBox, los vuelca a un StringList y los saca a pantalla:

```
var
  Seleccionados: TStringList;
  i: Integer;
begin
  Seleccionados := TStringList.Create;

  for i := 0 to ListView.Items.Count - 1 do
    if ListView.Items[i].Checked then
      Seleccionados.Add( ListView.Items[i].SubItems[1] );

  ShowMessage( Seleccionados.Text );

  Seleccionados.Free;
end;
```

Otra utilidad muy interesante que se le puede dar a la primera columna es asociarle una imagen a cada fila. Para ello hay que añadir al formulario el componente ImageList que encuentra en la pestaña Win32.

Después añadimos imágenes a la lista y asociamos las propiedades LargeImages, StateImages y SmallImages con el componente ImageList. Con sólo hacer eso todas las filas tendrán a la izquierda la primera imagen de la lista de imágenes. Para cambiar la imagen en una fila determinada se hace:

```
ListView.Items[2].ImageIndex := 2;
```

o bien cambiamos sólo la fila seleccionada por el usuario:

```
if ListView.Selected <> nil then
  ListView.Selected.ImageIndex := 1;
```

ACTIVANDO LA SELECCIÓN AUTOMÁTICA

Si activamos en el componente ListView la propiedad HotTrack y pasamos el puntero del ratón por las filas veremos como se iluminan en otro color, quedando seleccionadas fijamente si permanecemos con el ratón sobre las mismas.

Esto puede ser de utilidad para crear programas visualizadores de imágenes que muestren fotografías con sólo posicionarse con el ratón encima del nombre de la foto JPG. O para crear un listado de URL para nuestras páginas web preferidas.

También se pueden activar las propiedades:

```
htHandPoint     -> Cambia el puntero del ratón a una mano cuando se
para por encima de las filas.
htUnderlineCold -> Subraya la fila que se va a seleccionar
htUnderlineHold -> Subraya la fila que hay seleccionada
```

En el próximo artículo vamos a ver como reprogramar el dibujado de las filas en tiempo real.

Pruebas realizadas en Delphi 7.

Mostrando información en un ListView (III)

En esta última parte referida al componente ListView vamos a ver como modificar los colores de las filas y las columnas según nuestro propio criterio. Para ello vamos a utilizar el evento OnCustomDrawSubItem.

En el primer ejemplo voy a mostrar como cambiar la columna 2 (la del nombre del artículo) poniendo la fuente azul oscuro y negrita:

```
procedure TFormulario.ListViewCustomDrawSubItem( Sender:
TCustomListView;
  Item: TListItem; SubItem: Integer; State: TCustomDrawState;
  var DefaultDraw: Boolean );
begin
```

```
  // ¿Va pintar la segunda columna?
  if SubItem = 2 then
  begin
    Sender.Canvas.Font.Color := clNavy;
    Sender.Canvas.Font.Style := [fsBold];
  end
  else
    Sender.Canvas.Font.Color := clBlack;
end;
```

Por el contrario, si en vez de modificar las características de una columna queremos modificar las de una fila entonces sería de la siguiente manera:

```
procedure TFormulario.ListViewCustomDrawSubItem( Sender:
TCustomListView;
  Item: TListItem; SubItem: Integer; State: TCustomDrawState;
  var DefaultDraw: Boolean );
begin
  // ¿Es la primera fila?
  if Item.Index = 1 then
    Sender.Canvas.Font.Color := clRed
  else
    Sender.Canvas.Font.Color := clBlack;
end;
```

En este ejemplo mostrado hemos puesto la segunda fila con la fuente de color rojo. También se podría dibujar una fila según los datos contenidos en ella. Supongamos que deseo que se ponga la fuente de color rojo con un fondo amarillo en aquellos artículos cuyas unidades sean superiores a 3:

```
procedure TFormulario.ListViewCustomDrawSubItem( Sender:
TCustomListView;
  Item: TListItem; SubItem: Integer; State: TCustomDrawState;
  var DefaultDraw: Boolean );
begin
  // Si el número de unidades es superior a tres iluminamos la fila de
rojo con fondo amarillo
  if StrToInt( Item.SubItems[2] ) > 3 then
  begin
    Sender.Canvas.Brush.Color := clYellow;
    Sender.Canvas.Font.Color := clRed;
  end
  else
  begin
    Sender.Canvas.Brush.Color := clWhite;
    Sender.Canvas.Font.Color := clBlack;
  end;
end;
```

Las posibilidades para realizar combinaciones de este tipo son enormes tanto para columnas como para filas.

PINTANDO FILAS Y COLUMNAS SEGÚN SU ESTADO

Hasta ahora lo que hemos dibujado ha sido para todas las filas y columnas pero se podría modificar sólo la fila que tiene el foco en azul:

```
procedure TFormulario.ListViewCustomDrawSubItem( Sender:
```

```
TCustomListView;
  Item: TListItem; SubItem: Integer; State: TCustomDrawState;
  var DefaultDraw: Boolean );
begin
  // La fuente de la fila seleccionada en negrita
  if cdsFocused in State then
    Sender.Canvas.Font.Style := [fsBold]
  else
    Sender.Canvas.Font.Style := [];
end;
```

Las posibilidades de la variable State son:

```
cdsSelected       -> La columna o fila ha sido seleccionada
cdsGrayed         -> La columna o fila esta grisacea
cdsDisabled       -> La columna o fila esta deshabilitada
cdsChecked        -> La fila aparece con el CheckBox activado
cdsFocused        -> La columna o fila esta enfocada
cdsDefault        -> Por defecto
cdsHot            -> Se ha activado el HotTrack y esta enfocado
cdsMarked         -> La fila esta marcada
cdsIndeterminate  -> La fila no esta seleccionada ni deseleccionada
```

Con lo que hemos visto ya podemos utilizar el componente ListView como una rejilla de datos sin tener que utilizar tablas temporales ni un DBGrid.

Pruebas realizadas en Delphi 7.

Trabajando con archivos de texto y binarios (I)

Voy a mostrar a continuación las distintas maneras que tenemos para crear, editar o eliminar archivos de texto o binarios. También aprenderemos a movermos por los directorios y por dentro de los archivos obteniendo la información del sistema cuando sea necesario.

CREANDO UN ARCHIVO DE TEXTO

Los pasos para crear un archivo son los siguientes:

1º Se crea una variable de tipo TextFile, la cual es un puntero a un archivo de texto.

2º Se utiliza la función AssignFile para asignar el puntero F al archivo de texto.

3º A continuación abrimos el archivo en modo escritura mediante el procedimiento Rewrite.

4º Escrimimos en el archivo mediante la función WriteLn.

5º Cerramos el archivo creado con el procedimiento CloseFile.

Vamos a crear un procedimiento para crear el archivo prueba.txt al lado de nuestro programa:

```
procedure TFormulario.CrearArchivoTexto;
var
  F: TextFile;
begin
  AssignFile( F, ExtractFilePath( Application.ExeName ) + 'prueba.txt'
);
  Rewrite( F );
  WriteLn( F, 'Esto es el contenido del archivo de texto.' );
  CloseFile( F );
end;
```

MODIFICANDO UN ARCHIVO DE TEXTO

Cuando se utiliza la función Rewrite no comprueba si el archivo ya existía anteriormente, lo que puede provocar que elimine la información anterior. Para prevenir esto lo que hacemos es preguntar si ya existe el archivo y si es así entonces añadimos al contenido del archivo mediante el procedimiento Append:

```
procedure TFormulario.AnadirArchivoTexto;
var
  F: TextFile;
  sArchivo: string;
begin
  sArchivo := ExtractFilePath( Application.ExeName ) + 'prueba.txt';
  AssignFile( F, sArchivo );

  if FileExists( sArchivo ) then
    Append( F )
  else
    Rewrite( F );

  WriteLn( F, 'Añadiendo información al archivo de texto.' );
  CloseFile( F );
end;
```

ABRIENDO UN ARCHIVO DE TEXTO

Vamos a abrir el archivo de texto en modo lectura para volcar su contenido en un campo memo del formulario. Para ello abrimos el archivo con el procedimiento Reset y leemos cada línea de texto mediante ReadLn:

```
procedure TFormulario.CargarArchivoTexto;
var F: TextFile;
    sLinea: String;
begin
  AssignFile( F, ExtractFilePath( Application.ExeName ) + 'prueba.txt'
);
  Reset( F );

  while not Eof( F ) do
  begin
    ReadLn( F, sLinea );
```

```
    Memo.Lines.Add( sLinea );
  end;

  CloseFile( F );
end;
```

El procedimiento ReadLn obliga a leer la línea de texto dentro de una variable. Después incrementa automáticamente el puntero F hasta la siguiente línea de texto, siendo la función Eof la que nos dice si ha llegado al final del archivo. Aunque en este caso lo más fácil sería utilizar la propiedad LoadFromFile del objeto memo:

```
Memo.Lines.LoadFromFile( ExtractFilePath( Application.ExeName ) +
'prueba.txt' );
```

ELIMINANDO UN ARCHIVO

La eliminación de un archivo se hace con la función DeleteFile la cual está dentro de la unidad SysUtils:

```
procedure TFormulario.EliminarArchivoTexto;
var sArchivo: String;
begin
  sArchivo := ExtractFilePath( Application.ExeName ) + 'prueba.txt';
  if FileExists( sArchivo ) then
    DeleteFile( sArchivo );
end;
```

Todas estas funciones (Rewrite, Append, etc.) vienen del lenguaje estándar de Pascal, ya que en Delphi hay maneras mucho más sencillas de realizar estas tareas, como veremos más adelante.

Pruebas ralizadas en Delphi 7.

Trabajando con archivos de texto y binarios (II)

Vamos a ver otra manera de manejar ficheros de texto utilizando algunas clases que lleva Delphi para hacer nuestra labor mucho más fácil.

Para ello utilizaremos la clase TFileStream que hereda de la clase TStream para manejar flujos de datos, en este caso archivos. Una de las ventanas de utilizar FileStream en vez de los clásicos métodos de pascal tales como Rewrite, WriteLn, etc. es que controla automáticamente los buffer en disco según el tamaño de los mismos en Windows.

CREANDO UN ARCHIVO DE TEXTO USANDO LA CLASE TFILESTREAM

La clase TFileStream proporciona un método rápido y flexible para el manejo de archivos. Veamos como crear un archivo detexto:

```
procedure TFormulario.CrearArchivoStream;
var F: TFileStream;
    s: String;
begin
  F := TFileStream.Create( ExtractFilePath( Application.ExeName ) +
'prueba.txt', fmCreate );
  s := 'Añadiendo información al archivo de texto.' + #13 + #10;
  F.Write( s[1], Length( s ) );
  F.Free;
end;
```

El constructor Create de la clase TFileStream toma como primer parámetro la ruta del archivo y como segundo parámetro el tipo de acceso. Los tipos de acceso son:

```
fmCreate         -> Crea un nuevo archivo. Si el archivo ya existe lo
sobrescribe.
fmOpenRead       -> Abre el archivo en modo de solo lectura.
fmOpenWrite      -> Abre el archivo en modo de escritura.
fmOpenReadWrite  -> Abre el archivo en modo lectura/escritura.
```

Después se graba la información mediante el método Write el cual lee el contenido de un buffer (puede ser texto o binario) pasando como segundo parámetro su longitud (Length). Finalmente liberamos la clase con Free y el mismo objeto cierra automáticamente el archivo.

Le hemos pasado como primer parámetro s[1] porque es la dirección de memoria donde comienza la variable s (los string empiezan por 1). Al final de la cadena le hemos metido cambién los caracteres de retorno de carro para que pase a la siguiente línea, ya que un objeto FileStream lo trata todo como texto continuo.

Cuando la cantidad de información a mover es muy grande, éste metodo es mucho más rápido que utilizar el clasico Rewrite, WriteLn, etc. (a menos claro que creemos nuestro buffer).

AÑADIENDO TEXTO A UN ARCHIVO CON FILESTREAM

Para añadir líneas a nuestro archivo creado anteriormente abrimos el archivo en modo fmOpenWrite, nos vamos al final del archivo utilizando la propiedad Position y añadimos el texto:

```
procedure TFPrincipal.AnadirArchivoStream;
var F: TFileStream;
    s: String;
begin
  F := TFileStream.Create( ExtractFilePath( Application.ExeName ) +
'prueba.txt', fmOpenWrite );
  F.Position := F.Size;
  s := 'Añadiendo información al archivo de texto.' + #13 + #10;
  F.Write( s[1], Length( s ) );
  F.Free;
end;
```

Si no se hubiese utilizado la propiedad Position habría machadado la información introducida anteriormente. Podemos movernos a nuestro antojo con la propiedad Position para guardar información en un fichero en cualquier sitio. Para irse al principio de un archivo sólo hay que hacer:

```
F.Position := 0;
```

LEYENDO LOS DATOS MEDIANTE FILESTREAM

El siguiente procedimiento lee el contenido del archivo en modo fmOpenRead en la variable s que hace de buffer y posteriormente lo manda al memo:

```
procedure TFPrincipal.CargarArchivoStream;
var F: TFileStream;
    s: String;
begin
  F := TFileStream.Create( ExtractFilePath( Application.ExeName ) +
'prueba.txt', fmOpenRead );
  F.Read( s[1], F.Size );
  Memo.Text := s;
  F.Free;
end;
```

Aunque en este caso no tenemos la flexibilidad que nos aportaban las funciones ReadLn y WriteLn para archivos de texto.

BLOQUEANDO EL ARCHIVO A OTROS USUARIOS

Una de las cualidades de las que goza el objeto FileStream es la de bloquear el acceso a otros usuarios mientras estamos trabajando con un archivo. La protección se realiza cuando se abre el archivo. Por ejemplo, si queremos abrir el archivo en modo lectura exclusivamente para nosotros hacemos:

```
F := TFileStream.Create( ExtractFilePath( Application.ExeName ) +
'prueba.txt', fmReadOnly or fmShareExclusive );
```

Mediante el operador OR mezclamos las opciones de apertura con las opciones de bloqueo. Las posibles opciones de bloqueo son:

```
fmShareCompat        ->  Permite compatibilizar la apertura con otro
usuario o programa
fmShareExclusive     ->  Sólo nosotros tenemos derecho acceso de
lectura/escritura mientras este abierto
fmShareDenyWrite     ->  Bloqueamos el archivo para que nadie pueda
escribir salvo nosotros
fmShareDenyRead      ->  Bloqueamos el archivo para que nadie pueda
leerlo salvo nosotros
fmShareDenyNone      ->  Permite la lectura/escritura por parte de
otros usuarios.
```

También se podrían manipular archivos de texto facilmente mediante objetos TStringList como vimos con anterioridad. Yo personalmente utilizo TStringList, objetos Memo o RitchEdit ya que permiten una manipulación del texto en memoria bastante flexible antes de guardarla en disco.

En el próximo artículo veremos el tratamiento de archivos binarios.

Pruebas realizadas en Delphi 7.

Trabajando con archivos de texto y binarios (III)

Vamos a ver las distintas maneras que tenemos de manejar archivos binarios.

CREANDO UN ARCHIVO BINARIO

El crear un archivo binario utilizando AssignFile no es muy diferente de crear un archivo de texto:

```
procedure TFPrincipal.CrearArchivoBinario;
var
  F: File of byte;
  i: Integer;
begin
  AssignFile( F, ExtractFilePath( Application.ExeName ) + 'prueba.dat'
);
  Rewrite( F );

  for i := 1 to 10 do
    Write( F, i );

  CloseFile( F );
end;
```

Como se puede apreciar, la variable F representa un puntero a un tipo de archivo de bytes. Después utilizamos la función Write para ir guardando byte a byte dentro del archivo.

Cuando los archivos binarios son pequeños no hay problema pero cuando son grandes la lentitud puede ser insoportable (parece como si se hubiera colgado el programa). Cuando ocurre esto entonces hay que crear un buffer temporal para ir guardando los bytes en bloques de 1 Kb, 10 Kb, 20 Kb, etc. Hoy en día todo lo que entra y sale de un disco duro para por la memoria caché del mismo (a parte de la memoria virtual de Windows).

Vamos a ver un ejemplo de como crear un archivo binario grande (100 Kb) utilizando un buffer. Vamos a guardar en el archivo bytes consecutivos (0, 1, 2, .. 255, 0, 1, 2, ...):

```
procedure TFPrincipal.CrearArchivoBinarioConBuffer;
var
  F: File of byte;
  i, j: Integer;
  b: Byte;
  Buffer: array[1..1024] of byte;
```

```
begin
  AssignFile( F, ExtractFilePath( Application.ExeName ) + 'prueba.dat'
);
  Rewrite( F );

  b := 0;
  // Guardamos 100 veces el buffer de 1 KB (100 KB)
  for j := 1 to 100 do
  begin
    for i := 1 to 1024 do
    begin
      Buffer[i] := b;
      Inc( b );
    end;

    BlockWrite( F, Buffer, 1024 );
  end;

  CloseFile( F );
end;
```

Hemos creado un buffer de 1024 bytes para guardar la información mediante el procedimiento BlockWrite que toma como primer parámetro el puntero F, como segundo parámetro el buffer del cual va a guardar la información y como tercer parámetro la longitud del buffer. Cuando más grande sea nuestro buffer menos accesos a disco se necesita y más suelto va nuestro programa.

Ahora veremos como hacer lo mismo utilizando la clase TFileStream.

CREANDO UN ARCHIVO BINARIO CON LA CLASE TFILESTREAM

El parecido con la rutina anterior es casi idéntico salvo que hemos sustituido un fichero de tipo File of byte por la clase TFileStream. El método Write toma como parámetros el buffer y su longitud:

```
procedure TFPrincipal.CrearStreamBinario;
var F: TFileStream;
    Buffer: array[0..1023] of byte;
    i, j: Integer;
    b: Byte;
begin
  F := TFileStream.Create( ExtractFilePath( Application.ExeName ) +
'prueba.dat', fmCreate );

  b := 0;
  // Guardamos 100 veces el buffer de 1 KB (100 KB)
  for j := 1 to 100 do
  begin
    for i := 0 to 1023 do
    begin
      Buffer[i] := b;
      Inc( b );
    end;

    F.Write( Buffer, 1024 );
  end;
```

```
  F.Free;
end;
```

MODIFICANDO UN ARCHIVO BINARIO

En un archivo que no sea de tipo TextFile no se puede utilizar el procedimiento Append para añadir datos al final del mismo. Usando AssignFile se pueden utilizar dos métodos:

- Leer la información de todo el archivo en un buffer, añadir información al final del mismo y posteriormente guardarlo todo sobrescribiendo el archivo con Rewrite.

- Otro método sería crear una copia del archivo y añadirle datos al final del mismo. Luego habría que borrar el original y sustituirlo por este nuevo.

Ambos métodos no los voy a mostrar ya que sería algo primitivo en los tiempos que estamos. Para ello nada mejor que la clase TFileStream para tratamiento de archivos binarios.

MODIFICANDO UN ARCHIVO BINARIO UTILIZANDO LA CLASE TFILESTREAM

Añadir datos a un archivo binario utilizando la clase TFileStream es tan fácil como irse al final del archivo y ponerse a escribir. De hecho sólo hemos añadido una línea de código al archivo anterior y hemos cambiado el método de apertura:

```
procedure TFPrincipal.AnadirStreamBinario;
var F: TFileStream;
    Buffer: array[0..1023] of byte;
    i, j: Integer;
    b: Byte;
begin
  F := TFileStream.Create( ExtractFilePath( Application.ExeName ) +
'prueba.dat', fmOpenWrite );
  F.Position := F.Size;

  b := 0;
  // Guardamos 100 veces el buffer de 1 KB (100 KB)
  for j := 1 to 100 do
  begin
    for i := 0 to 1023 do
    begin
      Buffer[i] := b;
      Inc( b );
    end;

    F.Write( Buffer, 1024 );
  end;

  F.Free;
end;
```

LEYENDO UN ARCHIVO BINARIO

Para la lectura de un archivo binario también utilizaremos un buffer para acelerar el proceso:

```
procedure TFPrincipal.CargarStreamBinario;
var F: TFileStream;
    Buffer: array[0..1023] of byte;
begin
  F := TFileStream.Create( ExtractFilePath( Application.ExeName ) +
'prueba.dat', fmOpenRead );

  // ¿No ha llegado al final de archivo?
  while F.Position < F.Size do
  begin
    // Leemos un bloque de 1024 bytes
    F.Read( Buffer, 1024 );
    // ya tenemos un bloque de información en el buffer
    // podemos hacer lo que queramos con el antes de cargar el
siguiente bloque
  end;

  F.Free;
end;
```

¿Cómo sabemos cuando se termina el archivo? Lo sabemos por la variable Position que se va moviendo automáticamente a través del método Read. Cuando llegue al final (F.Size) cerramos el archivo.

Con esto llegamos a la conclusión de que para el manejo de archivos de texto lo ideal es utilizar AssignFile, StringList o campos Memo, pero para archivos binarios TFileStream cumple mejor su función.

En el próximo artículo seguiremos viendo más cosas sobre el tratamiento de archivos.

Pruebas realizadas en Delphi 7.

Trabajando con archivos de texto y binarios (IV)

En el artículo anterior vimos como movernos por un archivo binario utilizando la propiedad Position de la clase TFileStream. Ahora vamos a ver lo mismo utilizando AssignFile.

RECORRIENDO UN ARCHIVO BINARIO EN MODO LECTURA

Si utilizamos AssignFile para leer un archivo en vez de TFileStream podemos mover el puntero en cualquier dirección utilizando el procedimiento Seek:

procedure Seek(var F; N: Longint);

Este procedimiento toma como primer parámetro el puntero al archivo (F:

File of Byte) y como segundo parámetro la posición donde queremos movernos, siendo cero la primera posición. Para irse al final del archivo se hace:

```
Seek( F, FileSize(F) )
```

Vamos a abrir el archivo prueba.dat de 100 KB creado anteriormente y nos vamos a ir a la posición 50 para leer 10 bytes volcando la información en un campo memo en formato hexadecimal:

```
var F: file of byte;
  i: Integer;
  Buffer: array[0..9] of Byte;
begin
  AssignFile( F, ExtractFilePath( Application.ExeName ) + 'prueba.dat'
);
  Reset( F );
  Seek( F, 50 );
  BlockRead( F, Buffer, 10 );

  for i := 0 to 9 do
    Memo.Text := Memo.Text + IntToHex( Buffer[i], 2 ) + ' ';

  CloseFile( F );
end;
```

El resultado sería:

```
32 33 34 35 36 37 38 39 3A 3B
```

El único inconveniente que tiene la función Seek es que sólo funciona en modo lectura (Reset). No se puede utilizar en modo escritura (Rewrite) para irse al final del archivo y seguir añadiendo datos.

Si no sabemos donde estamos se puede utilizar la función FilePos para averiguarlo:

```
ShowMessage( 'posición: ' + IntToStr( FilePos( F ) ) );
```

LEYENDO LAS PROPIEDADES DE UN ARCHIVO

Veamos de que funciones dispone Delphi para leer los atributos de un archivo (fecha, modo de acceso. etc.):

function FileAge(const FileName: string): Integer;

Esta función devuelve la fecha y hora de última modificación de un archivo en formato TTimeStamp. Para pasar de formato TTimeStamp a formato TDateTime utilizamos la función FileDateToDateTime. Por ejemplo:

```
FileDateToDateTime( FileAge( 'prueba.txt' ) ) -> devuelve la fecha y
hora de modificación del archivo prueba.txt
```

También disponemos de una función para obtener los atributos de un archivo:

function FileGetAttr(const FileName: string): Integer;

Devuelve un valor entero conteniendo los posibles atributos de un archivo (puede tener varios a la vez). Por ejemplo, para averiguar si el archivo esta oculto se haría lo siguiente:

```
var
  iAtributos: Integer;
begin
  if ( iAtributos and faHidden <> 0 ) and ( iAtributos and faArchive
<> 0  ) then
    ...
end;
```

Esta función no sólo comprueba archivos, sino también directorios y unidades de disco. Todos los posibles valores se encuentran en binario activando el bit correspondiente. Los valores posibles son:

```
Constante       Valor   Tipo de archivo
-------------------------------------------------------------------
--------------------
faReadOnly       1       Sólo lectura
faHidden         2       Oculto
faSysFile        4       Archivo del sistema
faVolumeID       8       Unidad de disco
faDirectory     16       Directorio
faArchive       32       Archivo
faSymLink       64       Enlace simbólico
faAnyFile       71       Cualquier archivo
```

Otra función interesante es FileSize la cual devuelve en bytes el tamaño de un archivo. El único inconveniente es que hay que abrir el archivo para averiguarlo:

```
var
  F: File of byte;
begin
  AssignFile( F, 'c:\prueba.txt' );
  Reset( F );
  ShowMessage( IntToStr( FileSize( F ) ) + ' bytes' );
  CloseFile( F );
end;
```

MODIFICANDO LAS PROPIEDADES DE UN ARCHIVO

Las función para modificar las propiedad de un archivo es:

function FileSetAttr(const FileName: string; Attr: Integer): Integer;

Si se pudo modificar el atributo del archivo devuelve un 0 o en caso de error el código del mismo. Por ejemplo para hacer un archivo de sólo lectura y oculto:

```
FileSetAttr( 'c:\prueba.txt', faReadOnly or faHidden );
```

En el próximo artículo terminaremos de ver las funciones más importates para el manejo de archivos.

Pruebas realizadas en Delphi 7.

Trabajando con archivos de texto y binarios (V)

Vamos a terminar de ver los recursos de los que dispone Delphi para el tratamiento de archivos.

PARTIENDO UN ARCHIVO EN DOS

En este ejemplo vamos a coger el archivo prueba.dat de 100 Kb y dejamos sólo las primeras 30 Kb eliminando el resto:

```
procedure TFormulario.PartirArchivo;
var F: File of byte;
    Buffer: array[0..1023] of Byte;
    i: Integer;
begin
  AssignFile( F, ExtractFilePath( Application.ExeName ) + 'prueba.dat'
);
  Reset( F );

  // Leemos 30 Kb utilizando un buffer de 1 Kb
  for i := 1 to 30 do
    BlockRead( F, Buffer, 1024 );

  Truncate( F );
  CloseFile( F );
end;
```

Hemos utilizado para ello la función Truncate, la cual parte el archivo que estamos leyendo según donde este el puntero F.

ASEGURANDO QUE SE GUARDE LA INFORMACIÓN EN ARCHIVOS DE TEXTO

Cuando abrimos un archivo de texto para escribir en él no se guarda completamente toda la información hasta que se cierra con el procedimiento CloseFile. Esto puede dar problemas si en algún momento el procedimiento WriteLn provoca un error dejando el archivo abierto. Se perdería la mayor parte de la información que supuestamente debería haberse guardado en el mismo.

Para evitar esto disponemos de la función Flush:

function Flush(var t: TextFile): Integer;

Esta función lo que hace es vaciar el buffer del archivo de texto en el disco duro. Es algo así como ejecutar CloseFile pero sin cerrar el archivo. Por ejemplo:

```
var
  F: TextFile;
  sArchivo: String;
begin
  sArchivo := ExtractFilePath( Application.ExeName ) + 'prueba.txt';
  AssignFile( F, sArchivo );
  Rewrite( F );
  WriteLn( F, 'Esto es el contenido del archivo de texto.' );
  Flush( F );
  // aquí podemos seguir escribiendo en el mismo
  CloseFile( F );
end;
```

GUARDANDO OTROS TIPOS DE DATOS EN LOS ARCHIVOS

Hasta ahora sólo hemos guardado información de tipo texto y binaria en archivos pero se puede guardar cualquier tipo de información utilizando cualquier tipo de dato. Por ejemplo para guardar una serie de números reales se haría de la siguiente manera:

```
procedure TFormulario.GuardarRecibos;
var
  F: File of Real;
  r: Real;
begin
  AssignFile( F, ExtractFilePath( Application.ExeName ) +
'ImporteRecibos.dat' );
  Rewrite( F );
  r := 120.45;
  Write( F, r );
  r := 1800.05;
  Write( F, r );
  r := 66.31;
  Write( F, r );
  CloseFile( F );
end;
```

Si os fijais bien en el archivo resultante vemos que ocupa 24 bytes. Esto es así porque el tipo real ocupa 8 bytes en memoria y como son 3 números reales lo que hemos guardado entonces hace un total de 24 bytes. Hay que asegurarse de que tanto al leer como al guardar información en este tipo de archivos se haga con el mismo tipo de variable. Si guardamos información con File of Real y la leemos con File of Single los resultados podrían ser catastróficos (a parte de quedarse el archivo a medio leer).

También se pueden guardar estructuras de datos almacenadas en registros. En

este ejemplo que voy a mostrar vamos a ver como guardar los datos de un cliente en un archivo:

```
type
  TCliente = record
    ID: Integer;
    sNombre: String[50];
    rSaldo: Real;
    bPagado: Boolean;
  end;

procedure TFPrincipal.CrearArchivoRegistro;
var
  Cliente: TCliente;
  F: File of TCliente;
begin
  AssignFile( F, ExtractFilePath( Application.ExeName ) +
'clientes.dat' );
  Rewrite( F );

  with Cliente do
  begin
    ID := 1;
    sNombre := 'FRANCISCO MARTINEZ LÓPEZ';
    rSaldo := 1200.54;
    bPagado := False;
  end;

  Write( F, Cliente );

  with Cliente do
  begin
    ID := 2;
    sNombre := 'MARIA ROJO PALAZÓN';
    rSaldo := 622.32;
    bPagado := True;
  end;

  Write( F, Cliente );

  CloseFile( F );
end;
```

Unas de las cosas a tener en cuenta es que cuando se define una estructura de datos no se puede definir una variable de tipo String sin dar su tamaño, ya que cada registro debe tener una longitud fija. Si ponemos String nos da un error al compilar, por eso hemos puesto en el nombre un String[50].

Con esto finalizamos la parte básica de tratamiento de archivos en Delphi.

Pruebas realizadas en Delphi 7.

Trabajando con documentos XML (I)

Un documento XML (Extensible Markup Language) contiene un lenguaje de etiquetas para almacenar información de forma estructurada. Es similar a las páginas web HTML, exceptuando que sólo guarda la información y no la forma de visualizarla. Los documentos XML proporcionan una forma simple para almacenar la información que permite ser comprendida fácilmente.

Además se ha convertido de hecho en un estándar, siendo utilizado en aplicaciones web, comunicaciones, etc. Se trata de un lenguaje de intercambio de datos que ya es soportado por la mayoría de lenguajes de programación.

Los documentos XML manejan la información de forma jerárquica, siendo las etiquetas las que describen la información de cada elemento así como los elementos hijos. Veamos un ejemplo de un documento XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Clientes SYSTEM "cli.dtd">
<Clientes>
  <Cliente id="1">
    <nombre>JUAN SANCHEZ LOPEZ</nombre>
    <nif>78562876E</nif>
    <saldopte>156.78</saldopte>
    <diaspago>120</diaspago>
  </Cliente>
  <Cliente id="2">
    <nombre>ROSA MARTINEZ GUILLEN</nombre>
    <nif>43897653W</nif>
    <saldopte>93.81</saldopte>
    <diaspago>30</diaspago>
  </Cliente>
</Clientes>
```

Este ejemplo muestra como está almacenada la información en un documento XML. La primera línea contiene la declaración de un documento XML. Aunque declaración es opcional es recomendable introducirla en todos los documentos XML. Es este caso nos dice que es un documento XML con versión 1.0 y que utiliza la codificación de caracteres UTF-8.

La segunda línea nos informa que el documento se llama Clientes y que va a estar enlazado con un documento externo llamado cli.dtd. Los documentos DTD se utilizan para definir las reglas de cómo tienen que estar estructurados los documentos XML. En nuestro caso define la regla de cómo deben estar definida la estructura de un cliente. Sería algo así como la definición de un esquema para la jerarquía de los datos del cliente. No es obligatorio definir un DTD por cada documento XML.

La información que se guarda en los documentos XML comienza con un nodo raiz Clientes que a su vez tiene nodos hijos: Cliente, nombre, etc. Las etiquetas a su vez pueden contener dentro valores predeterminados, por ejemplo Cliente tiene el valor id="1".

Aunque es posible trabajar directamente con un documento XML utilizando cualquier procesador de textos (incluso el Bloc de Notas de Windows) existen

aplicaciones para editar este tipo de documentos. El consorcio internacional W3C define un conjunto de reglas estándar que definen como debe crearse un documento XML cuyo nombre es DOM (Document Object Model).

USANDO DOM

Las interfaces denifidas por el estandar DOM contienen la implementación de diferentes documentos XML suministrados por terceras partes llamados vendedores (vendors). Si no deseamos utilizar la implementación DOM de estos vendedores, Delphi permite utilizar un mecanismo de registro adicional para crearnos nuestra propia implementación de documentos XML.

La unidad XMLDOM incluye declaraciones para todas las interfaces DOM definidas por las especificaciones de nivel 2 para documentos XML según el consorcio W3C. Cada vendedor DOM proporciona sus propias interfaces para manejar documentos XML. Una interfaz DOM viene a ser algo así como un driver para manejar documentos XML.

Para que Delphi pueda utilizar un documento DOM de algun vendedor en concreto hay que añadir la unidad correspondiente a dicha implementación DOM. Estas unidades finalizan con la cadena 'xmldom'. Por ejemplo, la unidad para la implementación de Microsoft es MSXMLDOM, la unidad para documentos de IBM es IBMXMLDOM y la unidad para implementaciones Open XML es OXMLDOM.

Si quisieramos crear nuestra propia implementación DOM tendríamos que crear una unidad que defina una clase descendiente de la clase TDOMVendor. Se haría de la siguiente manera:

- En nuestra clase descendiente, debemos sobrescribir dos métodos: el método Description, el cual devuelve una cadena con el identificador del venvedor y el método DOMImplementation donde define la interfaz IDOMImplementation.

- Después registramos nuestra nueva unidad llamando al procedimiento RegistrerDOMVendor y ya estaría lista para poder ser utilizada.

- Y si queremos eliminarla de las implementaciones estándar DOM debemos anular el registro llamando al procedimiento UnRegisterDOMVendor, el cual debería estar en la sección finalization.

Algunos vendedores suministran extensiones a las interfaces estandar DOM. Para que puedas usar esas extensiones la unidad XMLDOM define la interfaz IDOMNodeEx, la cual es descendiente de la interfaz estandar IDOMNode.

Podemos trabajar directamente con interfaces DOM para leer y editar documentos XML. Simplemente llamamos la la función GetDOM para obtener el intefaz IDOMImplementation, el cual proporciona un punto de partida.

TRABAJANDO CON COMPONENTES XML

El punto de partida para comenzar a trabajar con documentos XML es utilizando el componente de la clase TXMLDocument. Para poder crear un documento XML hay que seguir los siguientes pasos:

1. Añadimos el componente XMLDocument a nuestro formulario o módulo de datos. Este componente se encuentra en la pestaña Internet.

2. Seleccionamos en la propiedad DOMVendor la implementación del documento DOM que queremos usar. Por defecto aparece seleccionada la de Microsoft (MSXML).

3. Dependiendo de la implementación seleccionada, podemos modificar sus opciones en propiedad ParseOptions para configurar como va a procesarse el documento XML. Un parser viene a ser algo así como un analizador sintáctico.

4. Para trabajar con un documento XML que ya existe hay que especificar el documento de la siguiente manera:

- Si el documento esta almacenado en un archivo, ponemos en FileName el nombre del archivo.

- O bien podemos meter el texto XML dentro del mismo componente XMLDocument utilizando su propiedad XML.

5. Activamos el documento XML poniendo la propiedad Active a True.

Una vez que tenemos activo el documento, podremos navegar por su jerarquía de nodos leyendo o modificando sus valores. El nodo raiz está disponible en la propiedad DocumentElement.

En el próximo artículo veremos ejemplos de código fuente para leer y modificar documentos XML con Delphi.

Pruebas realizadas en Delphi 7.

Trabajando con documentos XML (II)

Anteriormente vimos que cada documento XML podía llevar asociado un documento DTD. Los documentos DTD (Document Type Definition) definen la estructura y la sintaxis de un documento XML. Su misión es mantener la consistencia entre los elementos del documento XML para que todos los documentos de la misma clase mantengan un mismo estilo.

Nuestro documento cli.dtd va a ser el siguiente:

```
<!ELEMENT Clientes (Cliente*)>
<!ELEMENT Cliente (nombre, nif, saldopte?, diaspago?)>
<!ELEMENT nombre (#PCDATA) >
<!ELEMENT nif (#PCDATA) >
<!ELEMENT saldopte (#PCDATA) >
<!ELEMENT diaspago (#PCDATA) >
```

Veamos que hace cada línea:

!ELEMENT Clientes (Cliente*)

Aquí establecemos que el documento XML se va a llamar Clientes y que cada elemento se llamará Cliente. El asterístico significa que va a tener cero o más elementos de tipo Cliente. Si quisieramos que el nodo Cliente apareciese una o más veces utilizaríamos el caracter +.

!ELEMENT Cliente (nombre, nif, saldopte?, diaspago?)

La segunda línea nos dice de que campos está compuesto el nodo Cliente: nombre, nif, saldopte y diaspago. Todos los campos que no lleven el signo de interrogación son obligatorios (en este caso nombre y nif).

!ELEMENT nombre (#PCDATA)
!ELEMENT nif (#PCDATA)
!ELEMENT saldopte (#PCDATA)
!ELEMENT diaspago (#PCDATA)

Por último se describe el tipo de dato que van a contener los nodos finales. El atributo #PCDATA significa que puede contener cualquier valor.

Con este archivo al lado de nuestros documentos XML podemos hacer doble clic sobre los mismos para que se vean en cualquier navegador.

NAVEGANDO POR LOS NODOS DEL DOCUMENTO XML

Una vez que el documento ha sido analizado sintácticamente por la implementación DOM, los datos representados estarán disponibles en la jerarquía de nodos. Cada nodo corresponde a un elemento etiqueta del documento. Por ejemplo, para el documento de clientes.xml que hemos visto anteriormente tendríamos la siguiente jerarquía: El nodo raiz es Clientes que a su vez tiene nodos hijos con el nombre Cliente. Cada uno de esos nodos hijos tienen a su vez otros nodos hijos tales como (nombre, nif, importepte y diaspago). Esos nodos hijos actuan como los nodos finales de un arból ya no tienen más nodos hijos.

Para acceder a cada nodo se utiliza la interfaz IXMLNode, comenzando por el nodo raiz a través de la propiedad DocumentElement del componente XMLDocument.

LEYENDO EL VALOR DE LOS NODOS

Vamos a ver un ejemplo de cómo leer el saldo pentiende (saldopte) del primer cliente y mostrarlo por pantalla:

```
var
  Cliente: IXMLNode;
begin
  XML.LoadFromFile( ExtractFilePath( Application.ExeName ) +
'clientes.xml' );
  XML.Active := True;
  Cliente := XML.DocumentElement.ChildNodes[0];
  ShowMessage( Cliente.ChildNodes['saldopte'].Text );
end;
```

Suponemos que el documento clientes.xml está en el mismo directorio que nuestro programa. Como se puede apreciar en el código hemos abierto el documento XML, lo hemos activado y a continuación hemos recogido el nodo del primer cliente dentro de la variable Cliente que es de tipo IXMLNode. Después hemos accedido a su nodo hijo llamado saldopte para leer su valor.

MODIFICANDO EL VALOR DE LOS NODOS

Utilizando el mismo sistema podemos modificar el valor de cada nodo. En el siguiente ejemplo voy a modificar el nombre del segundo cliente (ROSA MARTINEZ GUILLEN) por el de MARIA PEREZ ROJO:

```
var
  Cliente: IXMLNode;
begin
  XML.LoadFromFile( ExtractFilePath( Application.ExeName ) +
'clientes.xml' );
  XML.Active := True;
  Cliente := XML.DocumentElement.ChildNodes[1];
  Cliente.ChildNodes['nombre'].Text := 'MARIA PEREZ ROJO';
  XML.SaveToFile( ExtractFilePath( Application.ExeName ) +
'clientes2.xml' );
end;
```

Una vez modificado el nombre grabo todo el documento XML en otro archivo llamado clientes2.xml (aunque podía haber utilizado el mismo).

AÑADIENDO Y BORRANDO NODOS

Mediante el método AddChild podemos crear nuevos nodos hijos dentro de un documento XML. Veamos como dar de alta un nuevo cliente y guardar el archivo como clientes3.xml:

```
var
  Cliente, Nodo: IXMLNode;
begin
  XML.LoadFromFile( ExtractFilePath( Application.ExeName ) +
'clientes.xml' );
  XML.Active := True;
  Cliente := XML.DocumentElement.AddChild( 'Cliente' );
  Cliente.Attributes['id'] := '3';
```

```
  Nodo := Cliente.AddChild( 'nombre' );
  Nodo.Text := 'PABLO PALAZON ALCOLEA';
  Nodo := Cliente.AddChild( 'nif' );
  Nodo.Text := '79469163E';
  Nodo := Cliente.AddChild( 'saldopte' );
  Nodo.Text := '0.00';
  Nodo := Cliente.AddChild( 'diaspago' );
  Nodo.Text := '15';
  XML.SaveToFile( ExtractFilePath( Application.ExeName ) +
'clientes3.xml' );
end;
```

En esta ocasión utilizamos dos nodos: Cliente para crear el nodo padre y Nodo para crear cada uno de los nodos hijos.

Por último podemos eliminar elementos utilizando el método Delete cuyo parámetro es el número de nodo hijo que deseamos eliminar (siendo cero el primer elemento). Para eliminar el primer cliente de la lista hacemos los siguiente:

```
begin
  XML.LoadFromFile( ExtractFilePath( Application.ExeName ) +
'clientes.xml' );
  XML.Active := True;
  XML.DocumentElement.ChildNodes.Delete( 0 );
  XML.SaveToFile( ExtractFilePath( Application.ExeName ) +
'clientes4.xml' );
end;
```

Este código coge la lista de los dos clientes iniciales, elimina el primero y guarda el documento como clientes4.xml. De esta manera podemos crear una pequeña base de datos utilizando documentos XML.

En el próximo artículo veremos como simplificar aún más el manejo de archivos XML utilizando el asistente XML Data Binding.

Pruebas realizadas en Delphi 7.

Trabajando con documentos XML (III)

Aunque es posible trabajar con un documento XML usando solamente el componente XMLDocument y la interfaz IXMLNode es mucho más fácil utilizar el asistente XML Data Binding.

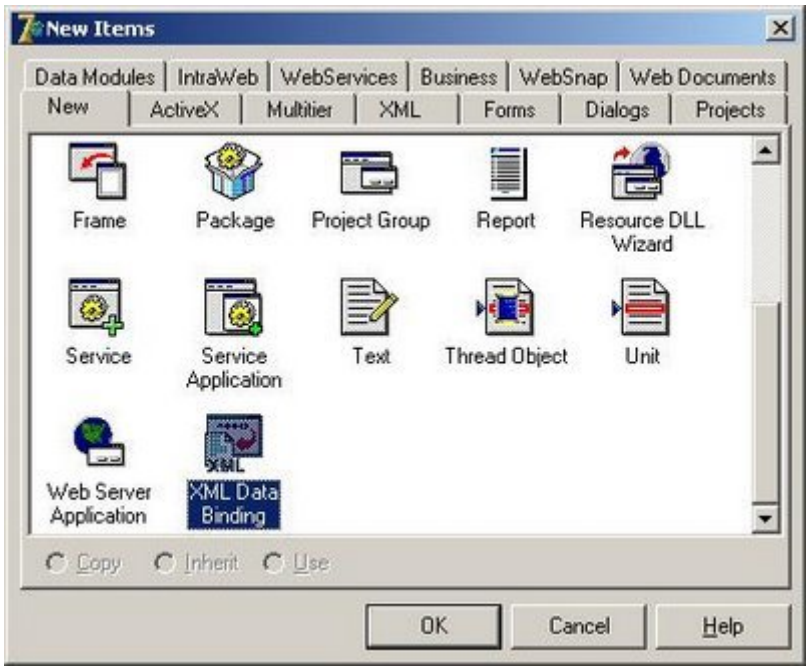
Dicho asistente toma los datos del esquema de un archivo XML a través de su archivo asociado DTD y realiza una nueva unidad con una interfaz que maneja nuestro documento XML sin tener que navegar por los nodos.

CREANDO UNA INTERFAZ PARA CLIENTES.XML

Para crear una unidad nueva para nuestro documento clientes.xml hay que hacer lo siguiente:

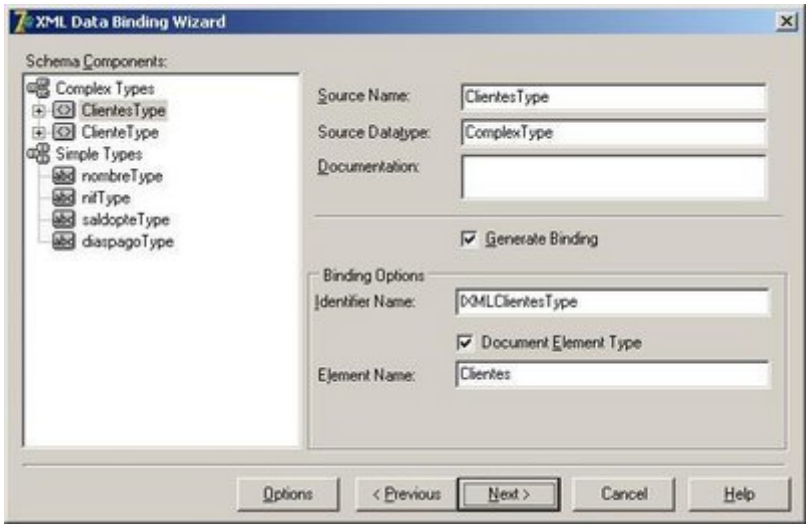
1. Seleccionamos en Delphi la opción File -> New -> Other...

2. Estando situados en la pestaña New seleccionamos el icono XML Data Binding y pulsamos OK.

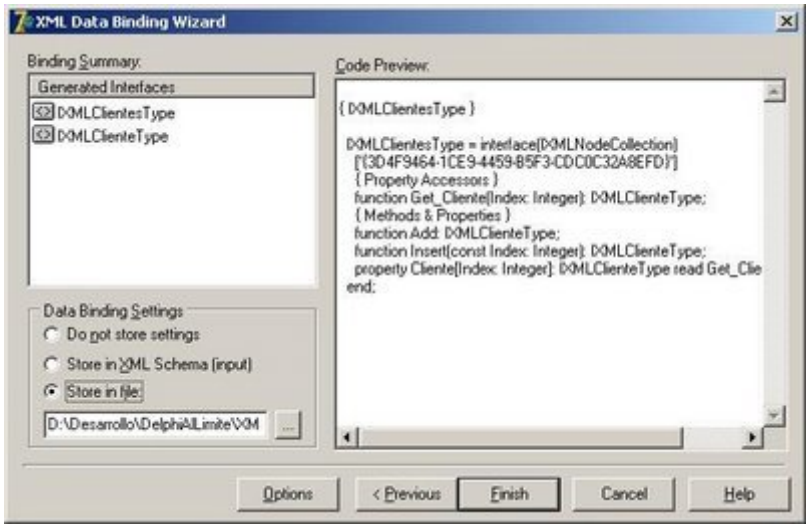


3. En la ventana siguiente seleccionamos el archivo cli.dtd y pulsamos Next.

4. Después nos aparece otra ventana donde se nos permite modificar el nombre de la clase a crear así como los nombres que van tener las propiedades y métodos de la nueva clase. Lo dejamos como está y pulsamos Next.



5. La última pantalla del asistente nos muestra como va a ser definitivamente la interfaz que va a crear. Sólo nos queda pulsar Finish.



Va a generar la siguiente unidad:

```
{*******************************************************************
*****}
{
}
{                              XML Data Binding
}
{
}
{         Generated on: 11/10/2007 11:15:57
}
{       Generated from: D:\Desarrollo\DelphiAlLimite\XML\cli.dtd
}
{   Settings stored in: D:\Desarrollo\DelphiAlLimite\XML\cli.xdb
}
{
}
{*******************************************************************
```

```
*****}

unit UInterfazClientes;

interface

uses xmldom, XMLDoc, XMLIntf;

type

{ Forward Decls }

  IXMLClientesType = interface;
  IXMLClienteType = interface;

{ IXMLClientesType }

  IXMLClientesType = interface(IXMLNodeCollection)
    ['{A21F5A80-EBA7-48F5-BFE1-EB9F390DFBBD}']
    { Property Accessors }
    function Get_Cliente(Index: Integer): IXMLClienteType;
    { Methods & Properties }
    function Add: IXMLClienteType;
    function Insert(const Index: Integer): IXMLClienteType;
    property Cliente[Index: Integer]: IXMLClienteType read
Get_Cliente; default;
  end;

{ IXMLClienteType }

  IXMLClienteType = interface(IXMLNode)
    ['{624B6C2E-4E94-4CE0-A8D4-FE719AA7D975}']
    { Property Accessors }
    function Get_Nombre: WideString;
    function Get_Nif: WideString;
    function Get_Saldopte: WideString;
    function Get_Diaspago: WideString;
    procedure Set_Nombre(Value: WideString);
    procedure Set_Nif(Value: WideString);
    procedure Set_Saldopte(Value: WideString);
    procedure Set_Diaspago(Value: WideString);
    { Methods & Properties }
    property Nombre: WideString read Get_Nombre write Set_Nombre;
    property Nif: WideString read Get_Nif write Set_Nif;
    property Saldopte: WideString read Get_Saldopte write
Set_Saldopte;
    property Diaspago: WideString read Get_Diaspago write
Set_Diaspago;
  end;

{ Forward Decls }

  TXMLClientesType = class;
  TXMLClienteType = class;

{ TXMLClientesType }

  TXMLClientesType = class(TXMLNodeCollection, IXMLClientesType)
  protected
    { IXMLClientesType }
    function Get_Cliente(Index: Integer): IXMLClienteType;
```

```
    function Add: IXMLClienteType;
    function Insert(const Index: Integer): IXMLClienteType;
  public
    procedure AfterConstruction; override;
  end;

{ TXMLClienteType }

  TXMLClienteType = class(TXMLNode, IXMLClienteType)
  protected
    { IXMLClienteType }
    function Get_Nombre: WideString;
    function Get_Nif: WideString;
    function Get_Saldopte: WideString;
    function Get_Diaspago: WideString;
    procedure Set_Nombre(Value: WideString);
    procedure Set_Nif(Value: WideString);
    procedure Set_Saldopte(Value: WideString);
    procedure Set_Diaspago(Value: WideString);
  end;

{ Global Functions }

function GetClientes(Doc: IXMLDocument): IXMLClientesType;
function LoadClientes(const FileName: WideString): IXMLClientesType;
function NewClientes: IXMLClientesType;

const
  TargetNamespace = '';

implementation

{ Global Functions }

function GetClientes(Doc: IXMLDocument): IXMLClientesType;
begin
  Result := Doc.GetDocBinding('Clientes', TXMLClientesType,
TargetNamespace) as IXMLClientesType;
end;

function LoadClientes(const FileName: WideString): IXMLClientesType;
begin
  Result := LoadXMLDocument(FileName).GetDocBinding('Clientes',
TXMLClientesType, TargetNamespace) as IXMLClientesType;
end;

function NewClientes: IXMLClientesType;
begin
  Result := NewXMLDocument.GetDocBinding('Clientes', TXMLClientesType,
TargetNamespace) as IXMLClientesType;
end;

{ TXMLClientesType }

procedure TXMLClientesType.AfterConstruction;
begin
  RegisterChildNode('Cliente', TXMLClienteType);
  ItemTag := 'Cliente';
  ItemInterface := IXMLClienteType;
  inherited;
end;
```

```
function TXMLClientesType.Get_Cliente(Index: Integer):
IXMLClienteType;
begin
  Result := List[Index] as IXMLClienteType;
end;

function TXMLClientesType.Add: IXMLClienteType;
begin
  Result := AddItem(-1) as IXMLClienteType;
end;

function TXMLClientesType.Insert(const Index: Integer):
IXMLClienteType;
begin
  Result := AddItem(Index) as IXMLClienteType;
end;

{ TXMLClienteType }

function TXMLClienteType.Get_Nombre: WideString;
begin
  Result := ChildNodes['nombre'].Text;
end;

procedure TXMLClienteType.Set_Nombre(Value: WideString);
begin
  ChildNodes['nombre'].NodeValue := Value;
end;

function TXMLClienteType.Get_Nif: WideString;
begin
  Result := ChildNodes['nif'].Text;
end;

procedure TXMLClienteType.Set_Nif(Value: WideString);
begin
  ChildNodes['nif'].NodeValue := Value;
end;

function TXMLClienteType.Get_Saldopte: WideString;
begin
  Result := ChildNodes['saldopte'].Text;
end;

procedure TXMLClienteType.Set_Saldopte(Value: WideString);
begin
  ChildNodes['saldopte'].NodeValue := Value;
end;

function TXMLClienteType.Get_Diaspago: WideString;
begin
  Result := ChildNodes['diaspago'].Text;
end;

procedure TXMLClienteType.Set_Diaspago(Value: WideString);
begin
  ChildNodes['diaspago'].NodeValue := Value;
end;

end.
```

UTILIZANDO LA NUEVA INTERFAZ PARA CREAR DOCUMENTOS XML

La unidad creada por el asistente la he guardado con el nombre UInterfazClientes.pas lo que significa que hay que introducirla en la sección uses del formulario donde vayamos a utilizarla.

Vamos a ver como damos de alta un cliente utilizando dicha interfaz:

```
var
  Clientes: IXMLClientesType;
begin
  XML.FileName := ExtractFilePath( Application.ExeName ) +
'clientes.xml';
  XML.Active := True;
  Clientes := GetClientes( XML );

  with Clientes.Add do
  begin
    Attributes['id'] := '3';
    Nombre := 'FERNANDO RUIZ BERNAL';
    Nif := '58965478T';
    Saldopte := '150.66';
    Diaspago := '45';
  end;

  XML.SaveToFile( ExtractFilePath( Application.ExeName ) +
'clientes5.xml' );
end;
```

He cargado el documento XML como siempre y utilizando la interfaz IXMLClientesType cargo la lista de clientes del documento clientes.xml. Después doy de alta un cliente y grabo el documento como clientes5.xml.

Una cosa cosa interesante que se puede hacer es activar la propiedad doAutoSave que está dentro de la propiedad Options del componente XMLDocument lo que implica que el documento se guardará automáticamente conforme vayamos modificando los elementos del documento XML.

La ventaja de convertir nuestro documento XML en una interfaz persistente es que ahora podemos manejar la lista de clientes como si fuera una base de datos utilizando los métodos get y set de cada uno de los campos así como las rutinas para alta, baja o modificación de elementos dentro del documento. Sólo es cuestión de echarle una mirada a la nueva unidad creada para ver las posibiliades de la misma.

Pruebas realizadas en Delphi 7.

Creando informes con Rave Reports (I)

Rave Reports es una herramienta externa asociada a Delphi que permite la creación de informes a partir de una base de datos. Se pueden crear toda variedad de informes, desde un simple folio con una banda hasta otros más complejos con múltiples bandas. Incluye las siguientes características:

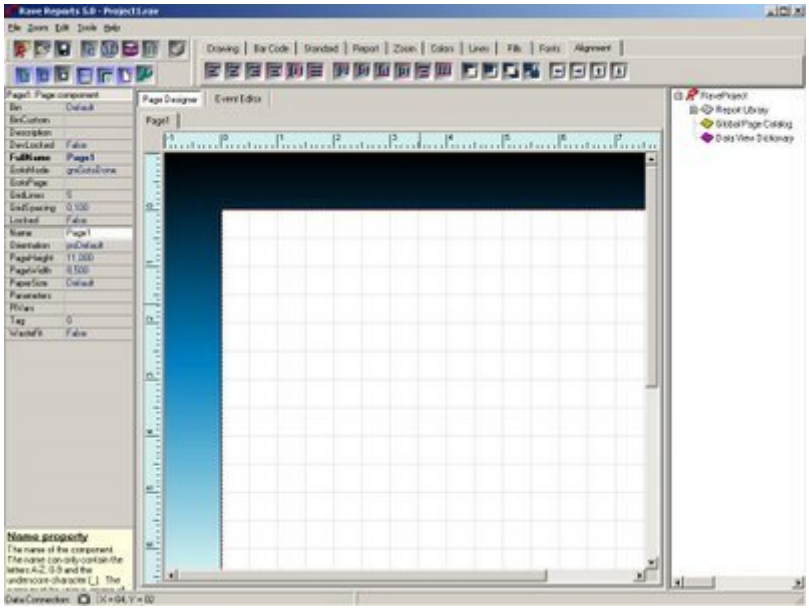
- Integración de gráficos.
- Párrafos justificados.
- Modificación de opciones de impresora.
- Control de las fuentes.
- Vista previa antes de imprimir.
- Exportación a PDF, HTML, RTF y archivos de texto.

CREANDO UN INFORME PARA NUESTRA APLICACION

Vamos a suponer que tengo una base de datos creada en Firebird 2.0 cuyo nombre es BaseDatos.fdb. Esta base de datos tiene la siguiente tabla:

```
CREATE TABLE CLIENTES (
ID INTEGER NOT NULL,
NOMBRE VARCHAR(100),
DIRECCION VARCHAR(100),
POBLACION VARCHAR(50),
CP VARCHAR(5),
PROVINCIA VARCHAR(50),
IMPORTEPTE DOUBLE PRECISION,
NIF VARCHAR(15),
ALTA TIMESTAMP,
HORALLAMAR TIME,
ULTIMOPAGO DATE,
NUMPAGOS INTEGER,
OBSERVACIONES BLOB
);
```

Entonces vamos a crear un informe para sacar un listado de clientes. Para ello abrimos el diseñador de informes Rave desde Delphi a través de la opción Tools -> Rave Designer. Nos aparecera el siguiente entorno:



Los informes que genera este diseñador de informes tienen extensión RAV. Por defecto, el nombre del informe que vamos a crear se llama Project1.rav. Le vamos a cambiar el nombre seleccionando File -> Save As... con el nombre listado_clientes.rav.

Antes de comenzar a diseñar el listado tenemos que establecer una conexión con nuestra base de datos Firebird llamada BaseDatos.fdb para poder extraer la información.

CONFIGURANDO LA CONEXION ODBC

Como vamos a utilizar una conexión ADO para vincularlo a este informe, primero tenemos que establecer el orígen de datos ODBC. Al ser mi base de datos es Firebird me he instalado el driver ODBC que se encuentra en la página:

http://www.firebirdsql.org/

El driver se encuentra en el apartado Development -> Firebird ODBC Driver.

Aparte de utilizar este driver para conexiones ADO nos puede ser muy útil para vincular nuestras bases de datos a prográmas ofimáticos tales como Word, Excel y Access.

Una vez descargado e instalado el driver ODBC vamos a realizar los siguientes pasos:

1. Abrimos el panel de control de Windows.

2. Hacemos doble clic en el icono Herramientas Administrativas.

3. Hacemos doble clic en el icono Orígenes de datos (ODBC).

4. Teniendo seleccionada la pestaña DSN de usuario pulsamos el botón

Agregar.

5. Seleccionamos Firebird/Interbase(r) Driver y pulsamos el botón Finalizar.

6. Se abrirá la siguiente ventana:

7. Vamos a rellenar los siguientes parámetros:

Nombre de Origen de Datos (DSN): BaseDatos_Rave
Description: BaseDatos_Rave
Base de Datos: D:\Desarrollo\DelphiAlLimite\Rave\BASEDATOS.FDB (donde esté el archivo FDB)
Cliente: C:\Firebird2\bin\fbclient.dll (donde esté instalado Firebird 2.0)
Cuenta de Base de Datos: SYSDBA
Contraseña: masterkey

8. Pulsamos el botón Comprobar conexión y debe mostrar ¡La conexión fue exitosa!

9. Pulsamos el botón Aceptar y veremos en la ventana anterior nuestra nueva conexión: BaseDatos_Rave.

10. Pulsamos el botón Aceptar y cerramos el panel de control de Windows.

Una vez configurada la conexión ODBC en Windows vamos a vincularla a nuestro informe.

CONECTANDO CON LA BASE DE DATOS DESDE RAVE REPORTS

Para establecer una conexión con nuestra base de datos Firebird nos tenemos

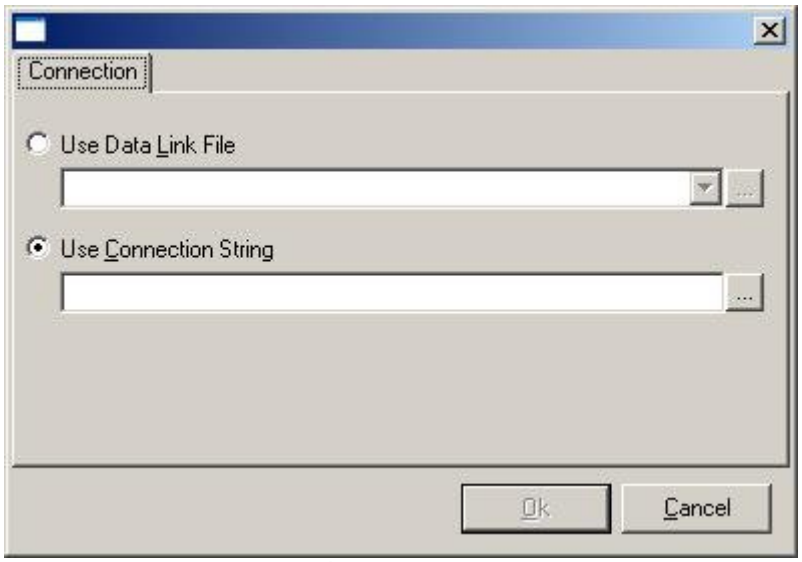
que ir a la barra de botones que se encuentra en la esquina superior izquierda del diseñador de informes Rave y pulsar el botón New Data Object:

 Al pulsarlo nos aparecen las siguientes opciones:

Seleccionamos Database Connection y pulsamos Next. La conexión sólo se puede realizar a través de ADO, BDE o DBExpress, aunque pueden añadirse otros drivers de conexión dentro del directorio:

C:\Archivos de programa\Borland\Delphi7\Rave5\DataLinks\

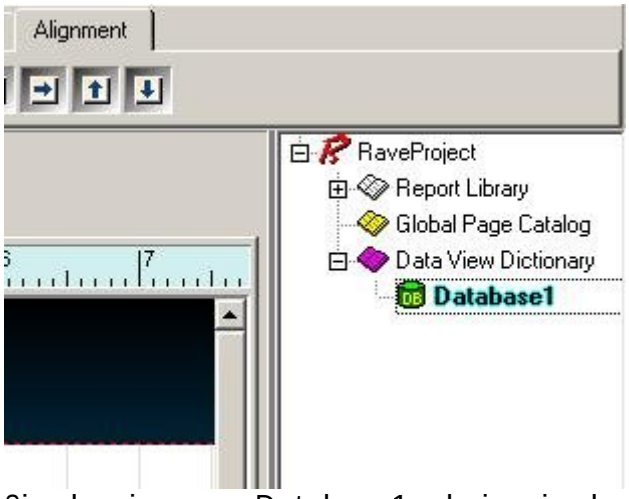
Seleccionamos ADO y pulsamos el botón Finish. Nos aparecerá la siguiente ventana:



Seleccionamos la opción Use Connection String y pulsamos el botón [...] que aparece a la derecha. Aparecerá la ventana de las propiedades de vínculo de datos:

Seleccionamos la pestaña Conexión y en la opción Usar el nombre de origen de datos seleccionamos BaseDatos_Rave. Pulsamos el botón Probar Conexión y nos saldrá el mensaje: La prueba de conexión fue satisfactoria. Pulsamos el botón Aceptar y en la ventana anterior pulsamos Ok.

Después de mucho sacrificio (es más fácil recompilar el núcleo de Linux) ya tenemos en el diseñador Rave la conexión directa con nuestra base de datos:



Si seleccionamos Database1 a la izquierda veremos sus propiedades:

Database1: Database component	
AuthDesign	(Auth)
AuthRun	(Auth)
Description	
DevLocked	False
FullName	**Database1**
LinkPoolSize	0
LinkType	**ADO**
Locked	False
Name	Database1
Tag	0

Vamos a renombar las propiedades FullName y Name a BaseDatos.

En el próximo artículo vamos a vincular la tabla clientes y vamos a realizar el informe.

Pruebas realizadas con Firebird 2.0 y Rave Reports 5.0.

Creando informes con Rave Reports (II)

Una vez establecida la conexión con la base de datos vamos a crear un objeto DriverDataView para vincularlo a nuestro informe.

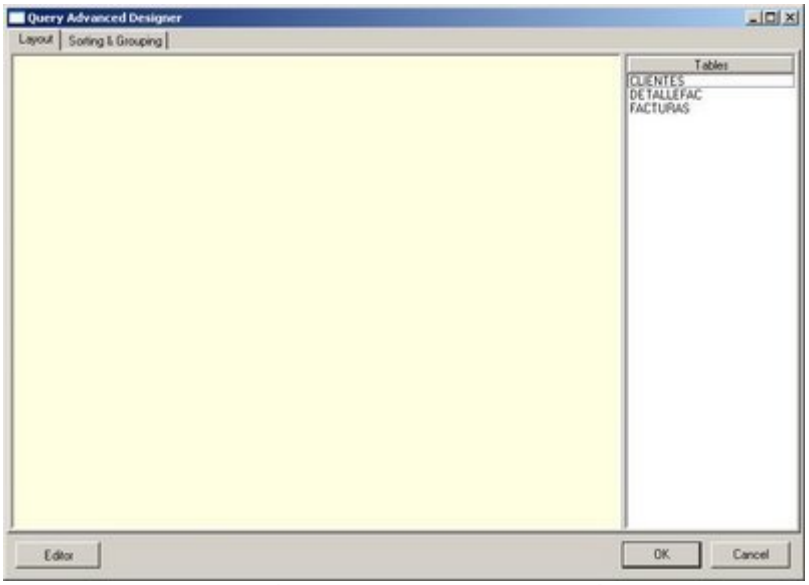
CREANDO UN VINCULO PARA LA TABLA CLIENTES

Para enlazar la tabla de clientes hay que hacer lo siguiente:

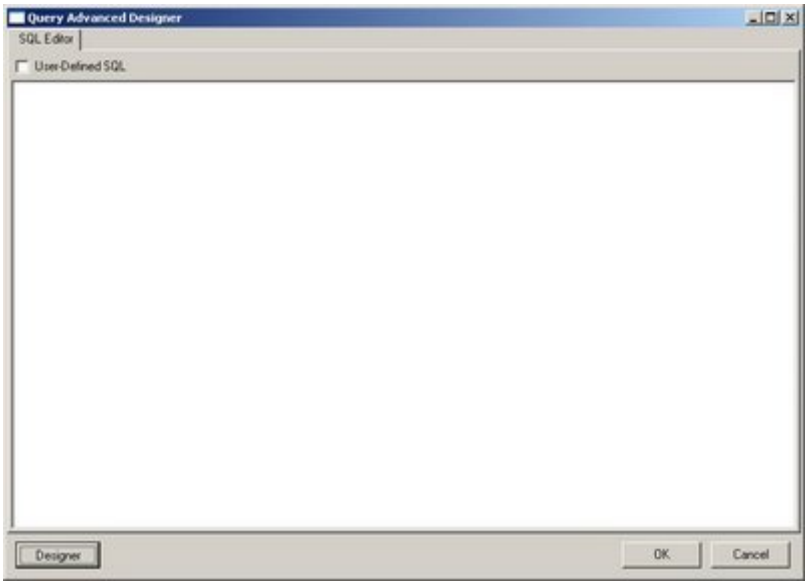
1. Pulsamos el botón NewDataObject y pulsamos el botón Next.

2. Seleccionamos Driver Data View y pulsamos el botón Next.

3. Seleccionamos BaseDatos y pulsamos el botón Finish. Se abrirá la siguiente ventana:



4. Pulsamos el botón Editor y se mostrará de esta forma:



5. Escribimos la siguiente SQL:

SELECT * FROM CLIENTES

6. Pulsamos el botón Ok.

7. Si nos fijamos a la derecha del editor a aparecido el objeto DriverDataView1. Lo seleccionamos y en la parte izquierda del editor cambiamos su propiedad Name a Clientes.

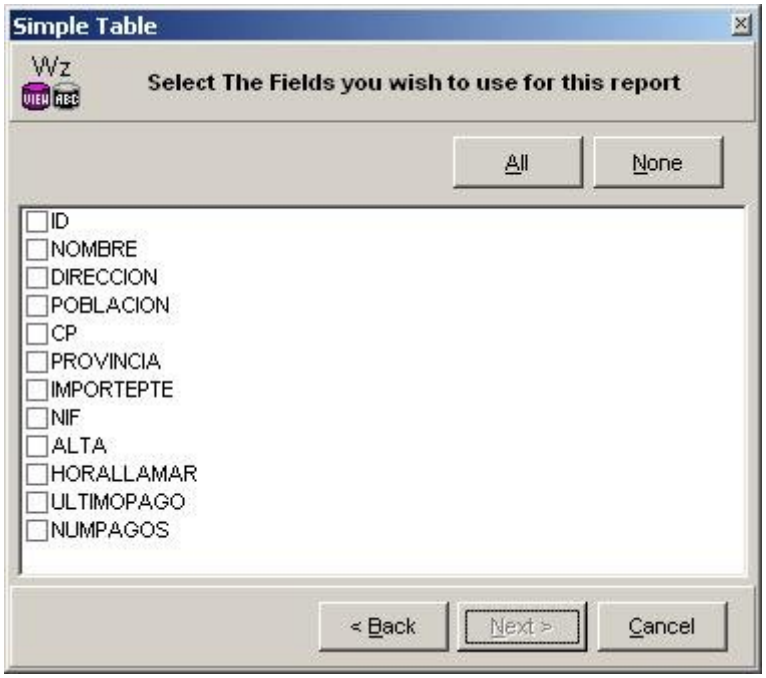
Con esto ya tenemos vinculada la tabla CLIENTES a nuestro informe.

UTILIZANDO EL ASISTENTE PARA GENERAR INFORMES

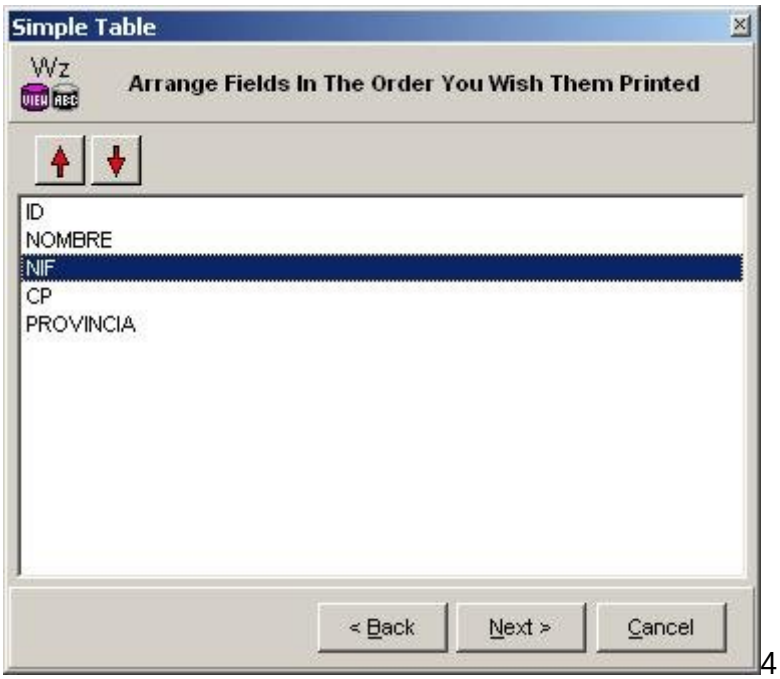
Vamos a ver los pasos para generar el listado de clientes utilizando el Wizard:

1. Seleccionamos Tools -> Report Wizards -> Simple Table. Aparecera esta ventana:

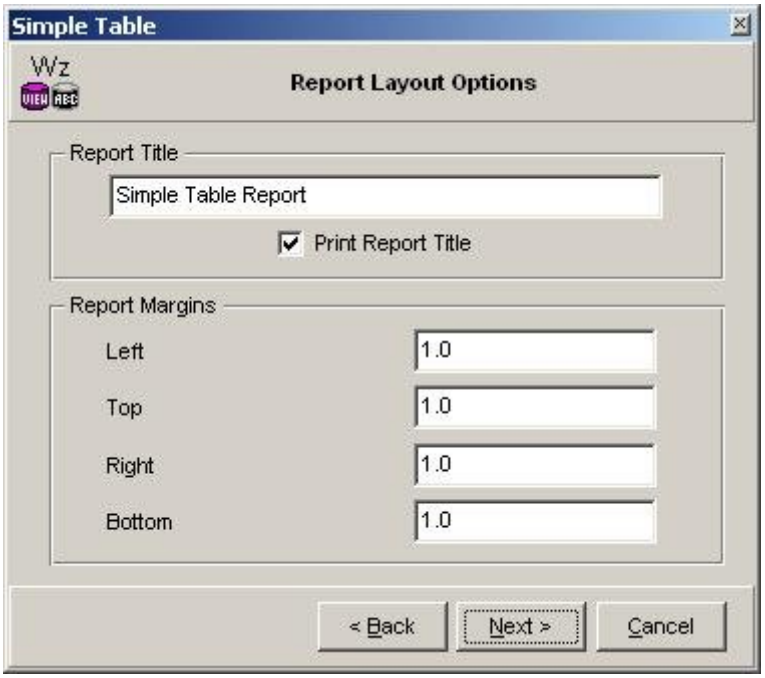
2. Seleccionamos Clientes y pulsamos Next:



3. Vamos a seleccionar los campos ID, NOMBRE, CP, PROVINCIA y NIF. Pulsamos Ok.

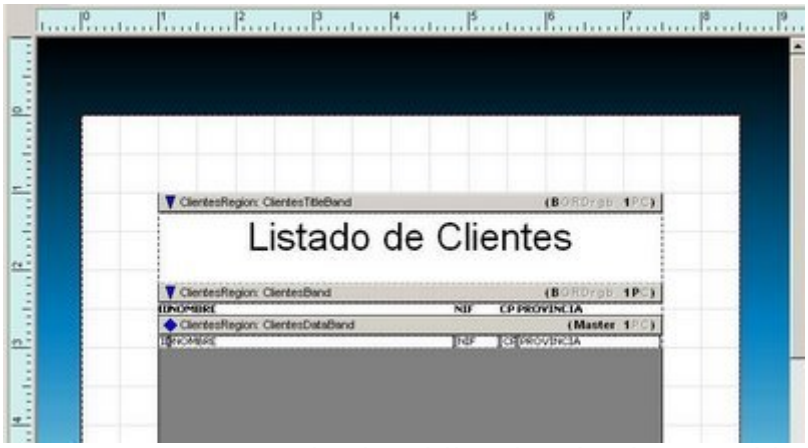


4. En la siguiente ventana se nos da la opción de ordenar los campos a nuestro gusto. En este caso he puesto el NIF después del NOMBRE. Al pulsar Next nos aparecera esta ventana:



5. En el campo Report Title ponemos Listado de Clientes. También podemos configurar los márgenes a nuestro gusto, aunque en este caso lo dejamos como está y pulsamos el botón Next. La siguiente ventana es esta:

6. Aquí podemos cambiar la fuente para el título, la cabecera de los campos y la fuente de los campos. En la cabecera de los campos le he configurado una fuente a Tahoma tamaño 10 y negrita. Para los campos lo he dejado en Tahoma 10. Al pulsar el botón Generate y aparecerá lo siguiente:



Con esto ya tenemos hecho el listado. Si pulsamos el botón Execute Report (la impresora) y seleccionamos Preview podemos ver como queda la vista previa del listado.

Como puede verse a primera vista los campos nos aparecen demasiado pegados los unos de los otros. Lo que hay que hacer es ajustar a mano en el editor las columnas para que queden a nuestro gusto.

Una de las cosas que mas me gustan de este editor es que se pueden seleccionar a la vez componentes de distintas bandas para moverlos o redimensionarlos. Con la tecla Mayúsculas pulsada y con los cursores del teclado podemos ampliar o reducir el tamaño de los campos. Y con la tecla Control pulsada se pueden mover varios campos a la vez. El comportamiento es muy similar al editor de formularios de Delphi.

ANALIZANDO EL INFORME CREADO

Si nos fijamos bien en el informe creado aparecen los siguientes objetos:

1. Primero crea un objeto Region (pestaña Report) para colocar todo el diseño encima.

2. Dentro de la región creada introduce tres bandas mediante el componente Band que se encuentra en la pestaña Report:

- ClientesTitleBand: La primera banda para el título Listado de Clientes.

- ClientesBand: La segunda banda contiene los títulos de los campos.

- ClientesDataBand: La tercera banda incluye los campos que se van a imprimir. Esta última banda es de tipo DataBand (también en la pestaña Report).

Lo bueno de tenerlo todo dentro de un componente Region es que si tenemos problemas con los márgenes de impresión, ajustando la región a nuestro gusto no aperecán los típicos problemas de folios duplicados en impresoras laser o multifunción que hace que salgan dos copias.

En el siguiente artículo vamos a crear un listado para tablas maestro/detalle.

Pruebas realizadas con Firebird 2.0 y Rave Reports 5.0.

Creando informes con Rave Reports (III)

Una vez que sabemos crear listados a partir de una tabla vamos a ver como se haría para tablas maestro/detalle, como puede ser una factura.

Nuestra factura consta de dos tablas. La cabecera de la factura está implementada en esta tabla:

```
CREATE TABLE FACTURAS (
ID          INTEGER NOT NULL,
IDCLIENTE   INTEGER,
BASEIMP     DOUBLE PRECISION,
IVA         DOUBLE PRECISION,
IMPORTEIVA  DOUBLE PRECISION,
TOTAL       DOUBLE PRECISION,
PRIMARY KEY (ID)
);
```

Y el detalle de la misma en esta otra:

```
CREATE TABLE DETALLEFAC (
ID          INTEGER NOT NULL,
IDFACTURA   INTEGER,
UNIDADES    DOUBLE PRECISION,
```

```
ARTICULO    VARCHAR(100),
PRECIO      DOUBLE PRECISION,
TOTALLINEA  DOUBLE PRECISION,
PRIMARY KEY (ID)
);
```

Cada línea de detalle está vinculada a la cabecera mediante el campo IDFACTURA.

VINCULANDO LAS TABLAS MAESTRO/DETALLE AL INFORME

Primero tenemos que incluir dos objetos DriverDataView para las tablas de cabecera y detalle. Hacemos lo siguiente:

1. Pulsamos el botón New Data Object.

2. Seleccionamos Driver Data View y pulsamos el botón Next.

3. Seleccionamos BaseDatos y pulsamos el botón Finish.

4. En la ventana que aparece pulsamos el botón Editor.

5. Escribimos una SQL que nos permita traernos los datos del cliente:

```
SELECT * FROM FACTURAS
LEFT JOIN CLIENTES ON FACTURAS.IDCLIENTE=CLIENTES.ID
```

6. Pulsamos el botón Ok.

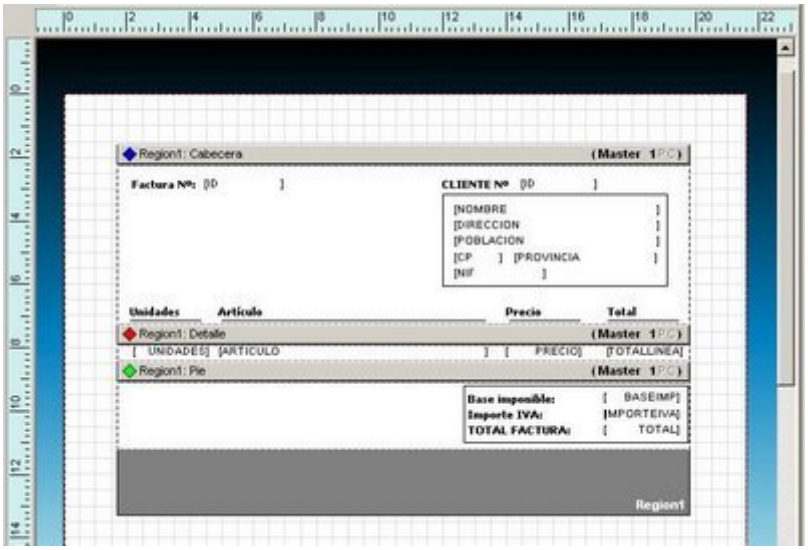
7. Estando en el editor seleccionamos a la derecha el nuevo DriverDataView1 creado y a la izquierda ponemos su propiedad Name a Facturas.

Siguiendo los mismos pasos tenemos que vincular la tabla DETALLEFAC con la SQL:

```
SELECT * FROM DETALLEFAC
```

CREANDO INFORMES PARA TABLAS MAESTRO/DETALLE

En esta ocasión vamos a ver como crear una factura utilizando directamente el editor sin asistentes. El resultado sería el siguiente:



Los pasos para crear esta factura serían los siguientes:

1. Insertamos un componente Region que esta situado en la pestaña Report.

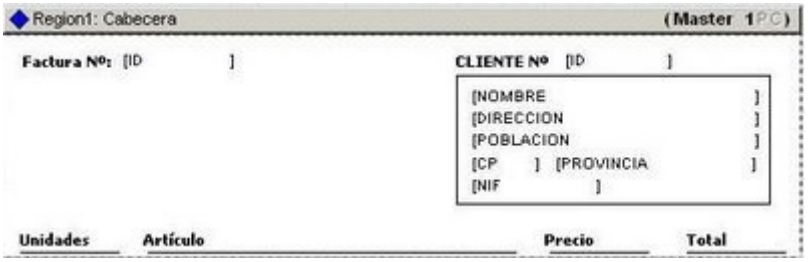
2. Dentro de la región creada insertamos tres componentes de tipo DataBand (pestaña Report).

3. A las tres bandas creadas las vamos a llamar Cabecera, Detalle y Pie.

4. A la propiedad DataView de la bandas Cabecera y Pie le asignamos la tabla Facturas.

5. A la propiedad DataView de la banda Detalle le asignamos la tabla DetalleFac.

6. La cabecera se compone de:



- 6 etiquetas en negrita de tipo Text situadas en la pestaña Standard.

- 8 campos de tipo DataText situados en la pestaña Report para los campos: ID, NOMBRE, DIRECCION, POBLACION, CP, PROVINCIA y NIF.

- 4 líneas utilizando el componente Line situado en la pestaña Drawing.

7. El detalle se compone de:

- 4 componetes de tipo DataText para los campos UNIDADES, ARTICULO,

PRECIO y TOTALLINEA. Para que los campos UNIDADES, PRECIO y TOTALLINEA salgan con formato a dos decimales tenemos que seleccionar en el árbol del proyecto a la derecha (debajo de DetalleFac) los campos DetalleFacUNIDADES, DetalleFacPRECIO y DetalleFacTOTALLINEA y establecer su propiedad DisplayFormat con el valor ###,###,#0.00. A dichos campos también hay que ponerle su propiedad FontJustify con el valor pjRight (eso se hace en el folio) para que aparezcan alineados a la derecha.

8. El pie tiene el siguiente diseño:

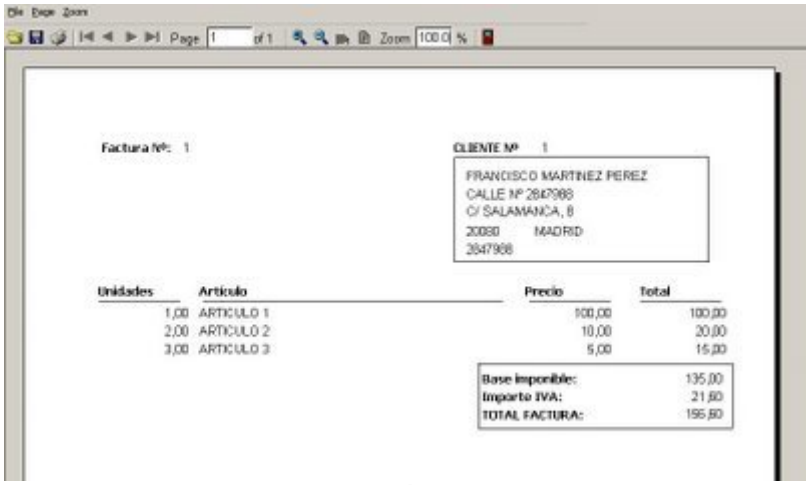
- 3 etiquetas de tipo Text.

- 3 campos de tipo DataText para los campos BASEIMP, IMPORTEIVA y TOTAL, dando el formato de número real como hemos hecho con los campos moneda del detalle.

- 1 rectángulo de tipo Rectangle situado en la pestaña Drawing.

Y por último fuera de la región en la parte superor del folio he añadido una etiqueta con la palabra FACTURA.

Al ejecutar el informe este sería el resultado:



Como puede verse es muy fácil crear informes utilizando Rave Reports siempre que tengamos claro de que tabla extraer la información.

En el próximo artículo veremos como ejecutar estos informes desde Delphi.

Pruebas realizadas con Firebird 2.0 y Rave Reports 5.0.

Creando informes con Rave Reports (IV)

Después de crear los informes utilizando el programa Rave Designer ahora vamos a ver como abrir esos informes con extensión RAV dentro de nuestro proyecto de Delphi.

ABRIENDO INFORMES RAVE DESDE DELPHI

Para llamar a los informes creados desde Delphi vamos a insertar en nuestro formulario los componentes de la clase TRvProject y TRvSystem situados en la pestaña Rave. Al objeto RvProject lo vamos a llamar Proyecto y al componente RvSystem lo llamaremos Sistema.

También vamos a vincular el componente Sistema al componente Proyecto a través de su propiedad Engine. Como estamos utilizando una conexión ADO tenemos que añadir en la sección uses de nuestro formulario la unidad RvDLADO, porque en caso contrario nos mostraría el error:

```
No DATA Link drivers have been loaded.
```

El objeto RvSystem realmente no es obligatorio utilizarlo, pero es recomendable porque con el mismo podemos personalizar todo lo relacionado con la impresión de documentos, desde el título, hasta los mensajes, la forma de como mostrar la vista previa, etc.

Pues con esto ya podemos lanzar la vista previa del informe:

```
begin
  Proyecto.ProjectFile := ExtractFilePath( Application.ExeName ) +
'factura.rav';
  Proyecto.Execute;
end;
```

Primero se nos abre una ventana para seleccionar el tipo de impresión y la impresora por donde va a salir:

Si queremos quitar esta ventana y que lance directamente la vista previa del informe tenemos que desactivar la propiedad ssAllowSetup que se encuentra dentro de la propiedad SystemSetups del componente RvSystem (que hemos llamado Sistema).

Para lanzar directamente el informe a la impresora sin vista previa hay que configurar la propiedad DefaultDest del componente RvSystem con el valor rdPrinter.

INSTALANDO NUESTRA APLICACION EN OTRO PC

Uno de los problemas que suelen ocurrir al llevarnos la aplicación a otro ordenador es que siempre falta que configurar algo para que pueda imprimirse el informe.

En mi caso he probado a ejecutar la aplicación en una máquina virtual limpia (VMWare con Windows XP) y sólo he tenido que hacer lo siguiente para que imprima el informe:

- Hay que instalar el driver ODBC de Firebird 2.0 como mencioné en otro artículo anterior.

- Hay que ir a herramientas administrativas dentro del panel de control de Windows y configurar nuestra conexión hacia el archivo BaseDatos.fdb. El alias de la conexión tiene que llamarse igual como lo hicimos anteriormente: BaseDatos_Rave.

- Me ha ocurrido que al hacer un test de conexión me faltaba la siguiente librería:

```
MSVCR71.DLL
```

Para solucionarlo lo copiamos de otro Windows XP que lo tenga y lo metemos en la carpeta C:\Windows\System32\ y listo. O bien lo bajamos de Internet.

Y con esto ya tenemos nuestros informes funcionando en otra máquina.

Naturalmente damos por hecho de que tiene instalado el motor de bases de datos Firebird 2.0.

En el próximo artículo seguiremos viendo más cosas interesantes relacionadas con Rave Reports.

Pruebas realizadas en Delphi 7 y Firebird 2.0.

Creando informes con Rave Reports (V)

Hasta ahora hemos visto como crear informes utilizando una conexión ADO, pero surgen los siguientes inconvenientes:

- Hay que instalar un driver ODBC en el cliente.

- Hay que configurar la conexión ODBC para que apunte a nuestra base de datos.

- Cada vez que el programa va a imprimir se abre una nueva conexión a la base de datos (imaginaos 20 usuarios trabajando a la vez).

- Puede faltar alguna DLL a la hora de instalar el programa en el cliente relacionada con conexiones ODBC.

Para solucionar esto vamos a ver como utilizar los objetos Direct Data View en el diseñador Rave Reports que nos van a evitar todos estos problemas.

PASANDO DE ADO, BDE y DBX

Los componentes Direct Data View que se encuentran en el editor Rave Designer permiten vincular los informes directamente con los componentes TRvDataSetConnection que se encuentren en nuestro programa Delphi, sin tener que establecer ninguna conexión de base de datos ni preocuparnos por drivers externos.

Funciona exactamente igual que los informes generados por QuickReport, donde el origen de datos es nuestro mismo programa, ahorrándonos conexiones extras. Vamos a ver un ejemplo de cómo generar un listado de clientes realizando una conexión directa entre Delphi y Rave Reports.

Los siguientes pasos serían para crear una conexión a Firebird desde Delphi:

1. Abrimos Delphi y creamos un formulario de prueba.

2. Insertamos un componente de la clase TIBDatabase en el formulario con el nombre BaseDatos.

3. Hacemos doble clic en el componente TIBDatabase y rellenamos los siguientes datos:

Connection: Remote
Server: 127.0.0.1
Database: D:\Desarrollo\DelphiAlLimite\Rave\BaseDatos.fdb
UserName: SYSDBA
Password: masterkey
Login Prompt: Desactivado

4. Añadimos un componente TIBTransaction y lo llamamos Transaccion.

5. Asociamos el componente Transaccion al componente BaseDatos a través de su propiedad DefaultTransaction.

6. En la propiedad DefaultDatabase del objeto TIBTransaction seleccionamos BaseDatos.

7. Insertamos en el formulario un componente TIBQuery con el nombre TClientes. En su propiedad Database seleccionamos BaseDatos. Y en su propiedad SQL ponemos:

```
SELECT * FROM CLIENTES
```

8. Hacemos doble clic sobre el componente TIBQuery y pulsamos la combinación de teclas CTRL + A para traernos los campos. Al hacer esto habrá dejado la base de datos abierta (BaseDatos). La dejamos así por ahora.

9. Insertamos en el formulario un componente TRvProject y lo llamamos Proyecto.

10. Insertamos otro componente de TRvSystem y lo llamamos Sistema. Asociamos este componente con TRvProject con su propiedad Engine.

11. Añadimos al formulario un componente TRvDataSetConnection que llamaremos DSClientes. En su propiedad DataSet elegimos TClientes.

Con toda esta movida ya tenemos una conexión a una base de datos Firebird y un DataSet especial para que Rave pueda recoger los datos.

CREANDO EL INFORME EN RAVE REPORTS

Dejando Delphi abierto con nuestra base de datos conectada (BaseDatos) ejecutamos el programa Rave Designer. Una vez estamos en el diseñador vamos a efectuar los siguiente pasos:

1. Pulsamos el botón New Data Object.

2. Seleccionamos Direct Data View y pulsamos Next.

3. Nos aparecerá una lista de conexiones abiertas en Delphi donde se estén utilizando componente TRvDataSetConnection. En nuestro caso aparecera

DSClientes (DT). Lo seleccionamos y pulsamos Finish.

4. Renombramos el objeto *DataView1* creado y lo llamamos Clientes.

5. Seleccionamos en el menú de arriba Tools -> Report Wizards -> Simple Table.

6. Seleccionamos Clientes y pulsamos Next.

7. Seleccionamos los campos ID, NOMBRE, DIRECCION, NIF, PROVINCIA y pulsamos Next.

8. Dejamos la ordenación como está y pulsamos Next.

9. En Report Title ponemos Listado de Clientes y pulsamos Next.

10. Seleccionamos la fuente a nuestro gusto y pulsamos Generate.

11. Guardamos el informe en el mismo directorio que nuestro proyecto de Delphi con el nombre Clientes.rav.

Si pulsamos el botón de la impresora y mostramos una vista previa veremos como se trae directamente de Delphi los datos y los imprime.

La potencia que nos da esta forma de trabajar es enorme, ya que es nuestro proyecto Delphi el que suministra los datos, siendo posible filtrar y ordenar tablas de bases de datos a nuestro antojo sin que Rave Reports no tenga que enterarse de donde proceden los datos.

Ya podemos cerrar el programa Rave Designer. Ahora vamos a ver como lanzar el informe desde Delphi.

LANZANDO EL INFORME RAVE DESDE DELPHI

En Delphi ya podemos desconectar la base de datos en tiempo de diseño. Para lanzar el listado hacemos lo siguiente:

```
begin
  BaseDatos.DatabaseName := '127.0.0.1:' + ExtractFilePath(
Application.ExeName ) + 'BaseDatos.fdb';

  try
    BaseDatos.Open;
  except
    raise;
  end;

  TClientes.Open;
  Proyecto.ProjectFile := ExtractFilePath( Application.ExeName ) +
'clientes.rav';
  Proyecto.Execute;
end;
```

Esto abrirá la base de datos Firebird, también la tabla clientes y ejecutará el informe.

A partir de aquí podemos hacer lo que nos venga en gana para modificar el listado de clientes: modificar la SQL para cambiar la ordenación, filtrar por ID, NIF, etc.

FILTRANDO TABLAS MAESTRO/DETALLE EN RAVE REPORTS

Utilizando las tablas FACTURAS y DETALLEFAC que vimos anteriormente podemos filtrar el detalle de la factura a partir de la cabecera. Resumiendo un poco, los pasos serían los siguientes:

1. Vinculamos las tablas FACTURAS y DETALLEFAC en el informe utilizando componentes Direct Data View.

2. Volvemos a vincular los campos con estos dos componentes.

3. Seleccionamos la banda Detalle y configuramos las siguientes propiedades:

MasterDataView: Facturas
MasterKey: ID
DetailKey: IDFACTURA

Con estos simples pasos imprimiremos las facturas desde Delphi sin tener que preocuparnos de filtrar la tabla detalle. También habría que crear otro objeto Direct Data View para enlazar a la tabla CLIENTES y poder traernos los datos. Para imprimir la factura desde Delphi filtrando al cliente:

```
begin
  BaseDatos.DatabaseName := '127.0.0.1:' + ExtractFilePath(
Application.ExeName ) + 'BaseDatos.fdb';

  try
    BaseDatos.Open;
  except
    raise;
  end;

  // Filtramos la factura
  TFacturas.SQL.Clear;
  TFacturas.SQL.Add( 'SELECT * FROM FACTURAS WHERE ID=2' );
  TFacturas.Open;

  // Filtramos el cliente según la factura
  TClientes.SQL.Clear;
  TClientes.SQL.Add( 'SELECT * FROM CLIENTES WHERE ID=' +
TFacturas.FieldByName( 'IDCLIENTE' ).AsString );
  TClientes.Open;

  // Abrimos el detalle (ya se encarga Rave de filtrarlo respecto a la
cabecera)
  TDetalleFac.Open;
```

```
  // Lanzamos el informe
  Proyecto.ProjectFile := ExtractFilePath( Application.ExeName ) +
'factura.rav';
  Proyecto.Execute;
end;
```

Se supone que hemos creado en el formulario de Delphi dos componentes TIBQuery llamados TFactura y TDetalleFac para la cabecera y detalle de la factura.

Con esto finalizamos la introducción al editor de informes Rave Reports.

Pruebas realizadas en Delphi 7, Rave Reports 5.0 y Firebird 2.0.

Como manejar excepciones en Delphi (I)

Las excepciones son condiciones excepcionales que se producen en nuestra aplicación en tiempo de ejecución y que requieren un tratamiento especial. Un ejemplo de excepciones podría ser las divisiones por cero o los desbordamientos de memoria. El tratamiento de excepciones proporciona una forma estándar de controlar los errores, descubriendo anticipadamente los problemas y posibilitando al programador anticiparse a los fallos que puedan ocurrir.

Cuando ocurre un error en un programa, se produce una excepción, lo que significa que crea un objeto excepción y situa el puntero de la pila en el primer punto donde se ha provocado la excepción. El objeto excepción contiene información sobre todo lo que ha ocurrido.

Esto nos permite crear aplicaciones más robustas ya que se puede llegar a averiguar el lugar en concreto donde se ha producido el error, particularmente en áreas donde los errores puedan causar la perdida de datos y recursos del sistema.

Cuando creamos una respuesta a la excepción tenemos que hacerlo en dentro de bloques de código, los cuales se llaman bloques de código protegidos.

DEFINIENDO BLOQUES DE CODIGO PROTEGIDOS

Los bloques de código protegidos comienzan con la palabra reservada try. Si ocurre algún error dentro del bloque de código protegido, el tratamiento del error se introduce en un bloque de código que comienza con except.

Vamos a ver un ejemplo que provoca una excepción al abrir un archivo que no existe:

```
var
  F: TextFile;
begin
  AssignFile( F, 'c:\nosexiste.txt' );
```

```
  try
    Reset( F );
  except
    on E: Exception do
      Application.MessageBox( PChar( E.Message ), 'Error', MB_ICONSTOP
);
  end;
end;
```

La primera parte de un bloque protegido comienza con la palabra try. El bloque try contiene el código que potencialmente puede provocar la excepción. Al provocar la excepción saltará directamente al comienzo del bloque de código que comienza con la palabra reservada except.

Como puede apreciarse en el código anterior hemos creado un objeto E que representa la excepción creada. El objeto E pertenece a la clase Exception que a su vez hereda directamente de la clase TObject. Este objeto contiene propiedades y métodos para manejar la excepción provocada.

PROVOCANCO NUESTRA PROPIA EXCEPCION

Nosotros también podemos crear nuestras propias excepciones que hereden de la clase Exception. Por ejemplo, voy a crear una excepción si una variable de tipo string está vacía. Primero defino el tipo especial de excepción:

```
type
  ECadenaVacia = class( Exception );
```

Y ahora la provoco en el programa:

```
var
  sCadena: String;
begin
  if sCadena = '' then
    raise ECadenaVacia.Create( 'Cadena vacia.' );
end;
```

El comando raise provoca a propósito la excepción para detener la ejecución del programa. No es necesario que creemos nuestros tipos de excepción. También podía haber sido así:

```
  if sCadena = '' then
    raise Exception.Create( 'cadena vacia' );
```

Cuando se provoca una excepción la variable global ErrorAddr declarada dentro de la unidad System contiene un puntero a la dirección de memoria donde se ha provocado el error. Esta variable es de sólo lectura a título informativo.

CONTROLANDO UNA EXCEPCION SEGUN SU TIPO

Dentro de un mismo bloque de código podemos controlar que tipo de excepción queremos controlar. Por ejemplo, si ejecutamos el código:

```
var
  s: string;
  i: Integer;
begin
  s := 'prueba';
  i := StrToInt( s );
end;
```

Mostrará el error:

```
'prueba' not is a valid integer value
```

Si queremos saber el tipo de excepción podemos sacarla por pantalla:

```
var
  s: string;
  i: Integer;
begin
  s := 'prueba';

  try
    i := StrToInt( s );
  except
    on E: Exception  do
      Application.MessageBox( PChar( E.ClassName + ': ' + E.Message ),
'Error', MB_ICONSTOP );
  end;
end;
```

Hemos incluido en el mensaje de la excepción la clase y el mensaje de error. Este sería el resultado:

```
EConvertError: 'prueba' not is a valid integer value
```

Así, mediante la propiedad ClassName de la clase Exception podemos averiguar la clase a la que pertenece la excepción. Ahora mediante la sentencia on podemos aislar la excepción de la forma:

on tipo do sentencia

En nuestro caso sería así:

```
  try
    i := StrToInt( s );
  except
    on E: EConvertError  do
      Application.MessageBox( 'Error de conversión', 'Error',
MB_ICONSTOP )
    else
      Application.MessageBox( 'Error desconocido', 'Error',
MB_ICONSTOP );
  end;
```

Si se produjera una excepción que no fuese de la clase EConvertError mostraría el mensaje Error desconocido.

De este modo podemos aislar dentro de un mismo bloque de código los distintos tipos de excepción que se puedan producir.

Otro ejemplo podría ser la división de dos números enteros:

```
try
  Resultado = b div c;
except
  on EZeroDivide do Resultado := MAXINT;
  on EIntOverflow do Resultado := 0;
  on EIntUnderflow do Resultado := 0;
end;
```

Aquí hemos aislado cada uno de los casos que se puedan producir al dividir dos números enteros, alterando el resultado a nuestro antojo.

En el próximo artículo seguiremos viendo más características de las excepciones.

Pruebas realizadas en Delphi 7.

Como manejar excepciones en Delphi (II)

Una vez que hemos visto lo que es una excepción y cómo proteger nuestro código usando bloques protegidos vamos a pasar a ver como pueden meterse unas excepciones dentro de otras para dar más seguridad a nuestro código. Es lo que se llaman excepciones anidadas.

EXCEPCIONES ANIDADAS

Vamos a ver un ejemplo de como anidar una excepción dentro de otra:

```
var
  F: TextFile;
begin
  AssignFile( F, 'C:\noexiste.txt' );

  try
    Reset( F );

    try
      CloseFile( F );
    except
      on E: Exception do
        ShowMessage( 'Excepción 2: ' + E.Message );
    end;

  except
    on E: Exception do
      ShowMessage( 'Excepción 1: ' + E.Message );
  end;
end;
```

En este ejemplo hemos metido un bloque protegido dentro de otro, donde cada uno controla su propia excepción. En este caso provocaría la excepción 1 ya que el archivo no existe.

DETENIENDO LA EXCEPCION

Cuando se provoca una excepción, una vez la hemos procesado con la sentencia on E: Exception, la ejecución continua hacia el siguiente bloque de código. Si queremos detener la ejecución del programa debemos utilizar el comando raise:

```
var
  F: TextFile;
begin
  AssignFile( F, 'C:\noexiste.txt' );
  ShowMessage( '1' );

  try
    Reset( F );

  except
    on E: Exception do
      raise;
  end;

  ShowMessage( '2' );
end;
```

En este ejemplo nunca llegaría a ejecutarse el segundo ShowMessage ya que raise detiene la ejecución del procedimiento.

FORZANDO A QUE FINALICE LA EJECUCION

Hay bloques de código en los cuales cuando se provoca una excepción ni podemos continuar con la ejecución ni podemos cortar la ejecución. Por ejemplo, supongamos que abro un archivo en módo sólo lectura e intento escribir en el mismo. Esto provocaría una excepción, pero lo que no puedo hacer es detener en seco la ejecución del programa ya que hay que cerrar el archivo que hemos abierto.

Para solucionar esto, los bloques protegidos permiten finalizar la ejecución en el caso de que se produzca una excepción mediante la claúsula finally. En nuestro ejemplo nos interesa que se cierre el archivo abierto:

```
var
  F: TextFile;
begin
  AssignFile( F, 'C:\prueba.txt' );

  try
    Reset( F );

    try
      WriteLn( F, 'intentando escribir' );
```

```
    finally
      ShowMessage( 'Finalizando...' );
      CloseFile( F );
    end;

  except
    on E: Exception do
      raise;
  end;
end;
```

Tenemos dos excepciones anidadas: una para abrir el archivo con una sentencia except que detiene la ejecución y otra dentro que utiliza la sentencia finally para cerrar el archivo en el caso de que se produzca un error.

TRATANDO EXCEPCIONES EN COMPONENTES VCL

Los componentes VCL también pueden provocar muchas excepciones si no sabemos utilizarlos correctamente. Un error típico es el intentar acceder a un elemento que no existe dentro de un componente ListBox llamado Lista:

```
begin
  Lista.Items.Add( 'PABLO' );
  Lista.Items.Add( 'MARIA' );
  Lista.Items.Add( 'CARLOS' );

  try
    ShowMessage( Lista.Items[4] );
  except
    on E: EStringListError do
      ShowMessage( 'La lista sólo tiene tres elementos.' );
  end;
end;
```

En este caso se ha provocado una excepción de la clase EStringListError, aunque bien se podría haber controlado de este modo:

```
  try
    ShowMessage( Lista.Items[4] );
  except
    on E: Exception do
      ShowMessage( 'La lista sólo tiene tres elementos.' );
  end;
```

Los componentes VCL disponen principalmente de estas clases de excepciones:

EAbort: Finaliza la secuencia de eventos sin mostrar el mensaje de error.

EAccessViolation: Comprueba errores de acceso a memoria inválidos.

EBitsError: Previene intentos para acceder a arrays de elementos booleanos.

EComponentError: Nos informa de un intento inválido de registar o renombar

un componente.

EConvertError: Muestra un error al convertir objetos o cadenas de texto string.

EDatabaseError: Especifica un error de acceso a bases de datos.

EDBEditError: Error al introducir datos incompatibles con una máscara de texto.

EDivByZero: Errores de división por cero.

EExternalException: Significa que no reconoce el tipo de excepción (viene de fuera).

EIntOutError: Representa un error de entrada/salida a archivos.

EIntOverflow: Especifica que se ha provocado un desbordamiento de un tipo de dato.

EInvalidCast: Comprueba un error de conversión de tipos.

EInvalidGraphic: Indica un intento de trabajar con gráficos que tienen un formato desconocido.

EInvalidOperation: Ocurre cuando se ha intentado realizar una operación inválida sobre un componente.

EInvalidPointer: Se produce en operaciones con punteros inválidos.

EMenuError: Controla todos los errores relacionados con componentes de menú.

EOleCtrlError: Detecta problemas con controles ActiveX.

EOleError: Especifica errores de automatización de objetos OLE.

EPrinterError: Errores al imprimir.

EPropertyError: Ocurre cuando se intenta asignar un valor erroneo a una propiedad del componente.

ERangeError: Indica si se intenta asignar un número entero demasiado grande a una propiedad.

ERegistryExcepcion: Controla los errores en el resigtro.

EZeroDivide: Controla los errores de división para valores reales.

EXCEPCIONES SILENCIOSAS

Para dar un toque profesional a un programa hay ocasiones en que nos interesa controlar la excepción pero que no se entere el usuario del programa. Lo que no se puede hacer es abandonar la excepción con los comandos Break o con Exit ya que puede ser peor el remedio que la enfermedad.

Para salir elegantemente de una excepción hay que utilizar el comando Abort:

```
try
  { sentencias }
except
  Abort;
end;
```

De este modo se controla la excepción y el usuario no ve nada en pantalla.

Con esto finalizamos el tratamiento de excepciones en Delphi.

Pruebas realizadas en Delphi 7.

Creando aplicaciones multicapa (I)

En este artículo que estará separado en varias partes vamos a ver como crear aplicaciones de bases de datos multicapa cliente/servidor. Este tipo de aplicaciones esta dividido en unidades lógicas, llamadas capas, las cuales se ejecutan en distintas máquinas. Las aplicaciones multicapa distribuyen los datos y se comunican en una red de área local o bien sobre Internet. Esto proporciona muchas ventajas tales como centralizar la lógica de negocio en un sólo servidor y donde varios clientes van tirando de él. Además podemos crear aplicaciones que comuniquen varios centros de trabajo se estén separados geográficamente a través de Internet.

Una aplicación multicapa queda particionada de la siguiente manera:

- Aplicación Cliente: se encarga de mostrar la interfaz de usuario.

- Servidor de aplicaciones: reside en la red local central y es accesible por todos los clientes donde reciben datos directamente de este servidor.

- Servidor de bases de datos: en este servidor es donde está instalado el motor de bases de datos (Interbase, Firebird, Oracle, etc.), aunque el servidor de aplicaciones y el servidor de bases de datos pueden ser la misma máquina.

En este modelo a tres capas los clientes sólo pueden comunicarse con el servidor de aplicaciones y en ningún caso directamente con el motor de bases de datos, como ocurre en las aplicaciones cliente/servidor habituales.

Este tipo de aplicaciones multicapa no tiene porque estar compuesto sólo de tres capas, podría constar de varios servidores de bases de datos y servidores

de aplicaciones.

VENTAJAS DE CREAR UN MODELO MULTICAPA

En este modelo de bases de datos la aplicación cliente sólo se dedica a mostrar los datos al usuario, no sabe nada sobre como los datos son actualizados y mantenidos.

El servidor de aplicaciones (capa media) coordina y procesa las peticiones y actualizaciones de múltiples clientes. El servidor maneja todos los detalles, define el conjunto de datos e interactua con el servidor de bases de datos.

Las ventajas de este modelo multicapa son las siguientes:

- Encapsulación de lógica de negocio. Diferentes clientes de la aplicacion pueden acceder al mismo servidor intermedio. Esto permite evitar la redundancia (y coste de mantenimiento) de duplicar las reglas de negocio para cada aplicación cliente separada.

- Aplicaciones clientes pequeñas. Al delegar las tareas más pesadas en la capa media las aplicaciones clientes ocupan menos y consumen menos procesador y memoria, permitiendo instalarse en máquinas de bajo rendimiento. Esto trae la ventaja de que por muchos clientes que accedan a la aplicación, el motor de bases de datos sólo tiene una conexión, que va directamente al servidor de aplicaciones, evitando así problemas de concurrencia o latencia de datos entre distintas aplicaciones cliente. Estas aplicaciones clientes también pueden funcionar a través de Internet ya que su consumo de ancho de banda es mínimo, al contrario de conectar directamente con el motor de bases de datos.

- Procesar datos distribuidos. Distribuir el trabajo de una aplicación entre varias máquinas puede mejorar la ejecución, ya que el balanceo de carga permite reducir la carga de las máquinas que funcionan como servidor de aplicaciones. Por ejemplo, si vemos que una aplicación de gestión se relentiza podemos distribuir en una máquina las compras, en otra las ventas y la gestión de recibos en otra.

- Incrementar la seguridad. Podemos aislar la funcionalidad en las capas dando restricciones de seguridad. Esto proporciona unos niveles de seguridad configurables y flexibles. Las capas intermedias pueden limitar los puntos de entrada a material protegido, permitiendo controlar el control de acceso más fácilmente. Si usamos HTTP o COM+, podemos utilizar los modelos de seguridad que soportan.

COMPOSICION DE LAS APLICACIONES DE BASES DE DATOS MULTICAPA

Las aplicaciones multicapa usan los componentes de la pestaña DataSnap, los de la pestaña Data Access y a veces los de la pestaña WebServices, más un módulo de datos remoto que es creado por el asistente de la pestaña Multitier o WebServices del cuadro de diálogo New Items. Estos componentes

proporcionan las funcionalidades para empaquetar información transportable sólo con los datos que se hayan modificado.

Los componentes que necesitamos para aplicaciones multicapa son los siguientes:

Módulo de bases de datos remoto: Los módulos de datos pueden actual como un servidor COM o implementar un servicio web para dar a las aplicaciones clientes acceso a los datos. Este componente es utilizado en el servidor de aplicaciones en la capa intermedia (el servidor de aplicaciones).

Provider: Es el que proporciona los datos creando paquetes de datos y resolviendo las actualizaciones de los clientes. Este componente es utilizado en el servidor de aplicaciones en la capa intermedia (servidor de aplicaciones).

ClientDataSet: es un dataset especializado que usa la librería midas.dll o midaslib.dcu para manejar los datos almacenados en los paquetes que se envían y reciben. Este componente es utilizado por la aplicación cliente. Tiene una caché local y sólo envía al servidor los datos que han cambiado.

Connection: Es una familia de componentes que estan localizados en el servidor y que utilizan la interfaz IAppServer para leer las peticiones de las aplicaciones clientes. Cada componente de conexión está especializado en protocolo particular de comunicaciones.

Los componentes proveedores y clientes requiren las librerías midas.dll o midaslib.dcu cuando se va a distribuir la aplicación en otros equipos.

FUNCIONAMIENTO DE UNA APLICACION A TRES CAPAS

Los siguientes pasos ilustran la secuencia normal de eventos para una aplicación a tres capas:

1. El usuario cliente arranca la aplicación. El cliente conecta con el servidor de aplicaciones (el cual puede ser elegido en tiempo de ejecución). Si el servidor de aplicaciones no estuviera arrancado, lo arranca automáticamente. Los clientes reciben una interfaz IAppServer para comunicarse con el servidor de aplicaciones.

2. El cliente solicita los datos al servidor de aplicaciones, donde puede requerir todos los datos de una vez o pedir una parte de ellos poco a poco.

3. El servidor de aplicaciones lee la información solicitada por el cliente del motor de bases de dtaos (estableciendo una conexión con si fuera necesario), empaqueta la información y devuelve la información al cliente. La información adicional (por ejemplo, las características de los campos) pueden ser incluida en los metadatos del paquete de datos. Este proceso de empaquetamiento de datos es llamado providing.

4. El cliente decodifica el paquete recibido y muestra los datos al usuario.

5. Cuando el usuario de la aplicación cliente realiza modificaciones sobre los datos (añadir, borrar o modificar registros) estas modificaciones son almacenadas en un log temporal.

6. Finalmente los clientes envian las actualizaciones al servidor de aplicaciones, el cual responde normalmente a cada acción del usuario. Para aplicar los cambios, los paquetes de los clientes leerán y cambiarán el log antes de enviar sus paquetes de datos al servidor.

7. El servidor de aplicaciones decodifica el paquete y efectua los cambios (en el contexto de una transacción cuando sea apropiada). Si un registro no puede ser actualizado (por ejemplo, porque otra aplicación cambie el registro antes de que el cliente lo solicite y después de que el cliente haya aplicado los cambios), el servidor de aplicaciones intenta reconciliar los cambios de los clientes con los datos actuales, y guarda los registros que podrían no ser actualizados. Este proceso de guardar y resolver problemas con los registros es llamado resolving.

8. Cuando el servidor de aplicaciones finaliza el proceso de actualización de registros, devuelve los registros no actualizados al cliente para que pueda resolverlos por el mismo.

9. El cliente resuelve los registros que el servidor no ha podido actualizar. Hay muchas maneras de que un cliente pueda resolver estos registros. Generalmente el cliente intenta corregir la situación asegurándose de que los registros estén validados antes de enviarlos. Si la situación puede ser rectificada, entonces vuelve a enviar los datos al servidor.

10. El cliente desempaqueta los datos que vienen del servidor y refresca su caché local.

En la siguiente parte de este artículo veremos la estructura de una aplicación cliente.

Pruebas realizadas en Delphi 7.

Creando aplicaciones multicapa (II)

Después de tener una visión global de cómo funcionan las aplicaciones multicapa vamos a ver cada una de sus partes.

LA ESTRUCTURA DE LA APLICACION CLIENTE

El usuario que va a utilizar la aplicación cliente no notará la diferencia entre una aplicación cliete/servidor y una aplicación multicapa, ya que el acceso a la información se realiza a través de los componentes estándar TClientDataSet.

El componente ClientDataSet se comunica con el proveedor de datos a través de la interfaz IAppServer. Se pueden seleccionar diferentes protocolos de comunicación según el componente de conexión que se utilice, donde tenemos los siguientes componentes:

Componente	Protocolo
TDCOMConnection	DCOM
TSocketConnection	Windows sockets (TCP/IP)
TWebConnection	HTTP
TSOAPConnection	SOAP (HTTP y XML)

LA ESTRUCTURA DE LA APLICACION SERVIDOR

Una vez instalado el servidor de aplicaciones, cuando se ejecuta por primera vez no establece una conexión con los clientes. Más bien son los clientes los que inician y mantienen la conexión con el servidor de aplicaciones. Todo esto sucede automáticamente sin que tengamos que manejar solicitudes o administrar interfaces.

La base de un servidor de aplicaciones es el módulo de datos remoto, el cual esta especializado en soportar la interfaz IAppServer (para los servidores de aplicaciones que tienen servicios web, el módulo de datos remoto soporta la interfaz IAppServerSOAP, y usa como preferencia IAppServer).

Las aplicaciones cliente usan las interfaces de módulos de bases de datos remotos para comunicarse con los componentes Provider del servidor de aplicaciones. Cuando el módulo de datos remoto usa IAppServerSOAP, el componente de conexión se adapta a este para la interfaz IAppServer que el que usa el componente ClientDataSet.

Hay tres tipos de módulos de datos remotos:

TRemoteDataModule: Se usa este tipo si los clientes van a utilizan protocolos DCOM, HTTP, sockets o una conexión OLE hacia el servidor de aplicaciones, a menos que queramos instalar el servidor de aplicaciones con COM+.

TMTSDataModule: Se utiliza este tipo si vamos a crear un servidor de aplicaciones como librería DLL que esté instalada con COM+ (or MTS). Se puede utilizar este módulo de datos remoto MTS con protocolos DCOM, HTTP, sockets u OLE.

TSoapDataModule: Este es un módulo de datos que implementa una interfaz IAppServerSOAP en una aplicación de servicios web. Utilizaremos este tipo de módulo de datos para proveer datos a los clientes con acceso a servicios web.

Si el servidor de aplicaciones es distribuido mediante COM+ (o MTS), el módulo de datos incluye eventos para cuando el servidor de aplicaciones sea activado o desactivado. Esto permite conectar automáticamente con los

motores de bases de datos cuando se active y desconectarlas cuando se desactive.

EL CONTENIDO DEL MODULO DE BASES DE DATOS REMOTO

Como cualquier otro módulo de datos, se puede incluir cualquier componente no visual en el módulo de datos remoto. Pero hay que tener en cuenta ciertos aspectos:

- Para cada dataset que tiene el módulo de datos remoto en los clientes, debemos incluir un DataSetProvider. Un DataSetProvider parte la información en paquetes que son enviados a los ClientDataSet y aplican las actualizaciones de las bases de datos contra el servidor de aplicaciones.

- Para cada documento XML que el módulo de datos remoto envía al cliente, debemos incluir un proveedor XML. Un proveedor XML actua como un DataSetProvider, exceptuando que la actualización de los datos se efectua a través de documentos XML en vez de ir al motor de bases de datos.

No hay que confundir los componentes de conexión a bases de datos con los componentes de conexión a servidores de aplicaciones, ya que estos últimos utilizan los componentes de las pestañas DataSnap y WebServices.

MODULOS DE DATOS REMOTOS TRANSACCIONALES

Si vamos a crear un servidor de aplicaciones que va a utilizar los protocolos COM+ o MTS entonces podemos sacar ventaja de esto creando módulos de datos transaccionales (Transactional Data Module) en lugar de un módulo de datos remoto ordinario (Remote Data Module). Esto sólo se puede hacer en sistemas opetarivos Windows 2000 en adelante.

Al utilizar un módulo de datos transaccional tenemos las siguientes ventajas:

- Seguridad: COM+ (o MTS) proporciona unas reglas de seguridad en nuestro servidor de aplicaciones. A los clientes se les asignan reglas, las cuales determinan como pueden acceder a la interfaz MTS del módulo de datos.

- Los módulos de datos transaccionales permiten mantener la conexión de los clientes abierta cuando se conectan y desconectan muchas veces hacia el servidor de aplicaciones. De este modo, mediante unas pocas conexiones podemos controlar a muchos clientes que se conectan y se desconectan continuamente (como si fuera un servidor web).

- Los módulos de datos transaccionales pueden participar en transacciones que abarquen múltiples bases de datos o incluir funciones que no están implementadas en las bases de datos.

- Pudemos crear nuestro servidor de aplicaciones como un módulo de datos remoto cuya instancia es activada y desactivada según se necesite. Cuando se usa la activación en tiempo real, nuestros módulos de datos remotos son

instanciados solamente si los necesitan los clientes. Esto evita de gastar recursos que no se vayan a utilizar.

Pero no todo son ventajas. Con una simple instancia de un módulo de datos el servidor de aplicaciones se puede manejar todas las llamadas a bases de datos a través de una simple conexión de bases de datos. Pero si se abusa de esto se puede crear un cuello de botella y puede impactar en la ejecución cuando hay muchos clientes, con lo tenemos que mantener un equilibrio entre el número de clientes que van a acceder al servidor de aplicaciones y el número de módulos de datos remotos que se van instanciar.

También puede ocurrir el efecto contrario: si se utilizan múltiples instancias de un módulo de bases de datos remotas, cada instancia puede mantener una conexión a bases de datos independiente. De modo que si hay muchas instancias del módulos de datos remotos se abrirían demasiadas conexiones con la base de datos, disminuyendo el rendimiento del motor de bases de datos como si fuera la típica aplicación cliente/servidor con muchos usuarios.

AGRUPAMIENDO DE MODULOS DE DATOS REMOTOS

El agrupamiento de objetos (pooling) nos permite crear una caché de módulos de datos remotos que estarán disponibles en memoria para las llamadas de las aplicaciones cliente. De esta manera se conservarán recursos en el servidor y evitará el tener que instanciar y destruir de memoria los módulos de datos remotos cuando se utilicen o se dejen de utilizar.

Cada vez que no estamos usando el módulo de datos transaccional obtenemos todas las ventajas que proporciona el tener una cache de objetos agrupados tanto si la conexión se efectua a través de DCOM como si es mediante el componente TWebConnection. Además podemos limitar el número conexiones a bases de datos.

Cuando el servidor de aplicaciones web recibe una petición del cliente, este las pasa a su primer módulo de datos remoto disponible en la caché de módulos de datos remotos. Si no hay disponible ninguna instancia de módulo de datos remoto, el servidor crea una nueva (no sobrepasando el máximo de módulos especificado por nosotros). Esto proporciona una gran flexibilidad permitiendo manejar varios clientes a través de un simple instancia de módulo de datos remoto (el cual puede actuar como un cuello de botella) e irá creando nuevas instancias según se vaya necesitando.

Cuando una instancia de un módulo de datos remoto que está en la caché no recibe ninguna petición de clientes durante un tiempo prolongado, es automáticamente liberado. Esto evita que el servidor de aplicaciones se sature con muchas instancias abiertas.

En la siguiente parte de este artículo abarcaremos los tipos de conexión que se pueden realizar entre las aplicaciones clientes y el servidor de aplicaciones.

Pruebas realizadas en Delphi 7.

Creando aplicaciones multicapa (III)

Vamos seguir con la base teórica de como funcionan las aplicaciones multicapa antes de comenzar a crear un ejemplo práctico.

ELIGIENDO EL PROTOCOLO DE CONEXION

Cada protocolo de comunicación que podemos usar para conectar de las aplicaciones cliente a las aplicaciones servidor tienen sus propias ventajas e inconvenientes. Antes de elegir un protocolo tenemos que considerar que servicio va a prestar la aplicación, cuantos clientes va a ser atendidos, que reglas de negocio se van a establecer, etc.

USANDO CONEXIONES DCOM

DCOM es el protocolo que proporciona el acceso más directo y rápido de comunicación, no requiriendo aplicaciones externas aparte del servidor de aplicaciones.

Este protocolo proporciona servicios de seguridad cuando se utiliza un módulo de datos transaccional. Cuando usamos DCOM podemos identificar quien llama al servidor de aplicaciones (ya sea con COM+ o MTS). Por tanto, es posible determinar que reglas de acceso vamos a asignar a las aplicaciones cliente.

Son los clientes los que instancian directamente el módulo de datos remoto para realizar la comunicación, no requiriendo ninguna aplicación externa que haga de intermediario.

USANDO CONEXIONES SOCKET

La comunicación mediante sockets nos permiten crear clientes muy ligeros. Se suele utilizar este protocolo si no tenemos la seguridad que los sistemas clientes soporten DCOM. Los sockets proporcionan un sistema comunicación simple para establecer conexiones entre los clientes y el servidor.

En lugar de instanciar el módulo de datos remoto desde el cliente (como sucede con DCOM), los sockets usan una aplicación separada en el servidor (ScktSrvr.exe), la cual acepta las peticiones de los clientes e instancia el módulo de datos remoto usando COM. El componente de conexión en el cliente y ScktSrvr.exe en el servidor son los responsables de organizar las llamadas mediante la interfaz IAppServer.

El programa ScktSrvr.exe también puede ejecutarse como un servicio NT. Para ello habría que registrarlo como servicio usando línea de comandos. También se puede eliminar de la lista de servicios del mismo modo.

Uno de los inconvenientes que tienen los sockets es que no hay protección en el servidor contra fallos en la conexión con los clientes. Mientras este

protocolo de comunicación consume menos ancho de banda que el protocolo DCOM (el cual envía periódicamente mensajes de comprobación y mantenimiento), no puede detectar si un cliente se está ausente o no.

USANDO CONEXIONES WEB

Una las ventajas de utilizar el protocolo de comunicación HTTP es que los clientes pueden comunicarse con el servidor de aplicaciones aunque este protegido por un cortafuegos. Al igual que las conexiones socket, los mensajes HTTP proporcionan un sistema sencillo y de poco consumo de ancho de banda para comunicar los clientes y el servidor.

En lugar de instanciar el módulo de datos remoto desde el cliente (como sucede con DCOM), las conexiones basadas en HTTP pueden usar un servidor de aplicaciones web (como Apache) o el servidor puede ser la librería httpsrvr.dll, el cual acepta las peticiones de los clientes e instancia el módulo de datos remoto usando COM. El componente de conexión en la máquina cliente y httpsrvr.dll en el servidor son los responsable de organizar las llamadas a la interfaz IAppServer.

Las conexiones web aportan también la ventaja de la seguridad SSL suministrada por la librería wininet.dll (una librería de utilidades de internet que corre en los sistemas cliente). Una vez que hemos configurado el servidor web en la máquina que hace de servidor de aplicaciones, podemos establecer los nombre y claves de acceso para los usuario aprovechando las propiedades de conexión que aporta el componente web.

Las conexiones web tienen la ventaja de tener en una memoria caché los objetos de módulos de datos que se van instanciando, conocido como pooling. Esto permite que nuestro servidor cree un número de instancias de módulos remotos limitado para dar servicio a los clientes, sin tener que estar instanciando y destruyendo objetos sin parar cada vez que se conecta o desconecta un cliente.

A diferencia de otras conexiones con componentes, no podemos crear funciones que permitan comunicar directamente el servidor de aplicaciones con las aplicaciones clientes, lo que se llaman funciones callback.

USANDO CONEXIONES SOAP

El protocolo SOAP es el estándar utilizado para crear aplicaciones de servicios web. Este protocolo envia y recibe mensajes codificados en documentos XML, y los envía utilizando el protocolo HTTP.

Las conexiones SOAP tienen la ventaja de que son multiplataforma, ya que son soportadas por prácticamente todos los sistemas operativos actuales. Al utilizar como transporte el protocolo HTTP tienen las mismas ventajas: permite servir a los clientes a través de un cortafuegos y pueden utilizarse diversos servidores HTTP.

CONSTRUYENDO UNA APLICACION MULTICAPA

Resumiendo a grandes rasgos, los pasos generales para crear una aplicación de bases de datos multicapa son los siguientes:

1. Crear el servidor de aplicaciones.

2. Registrar el servidor de aplicaciones como un servicio o instalarlo y ejecutarlo como una aplicación (recomendado).

3. Crear la aplicación cliente.

El orden de creación es importante. Debemos crear y ejecutar el servidor de aplicaciones antes de crear el cliente. Esto se debe a que cuando vayamos a construir la aplicación cliente, en tiempo de diseño tenemos que conectar con el servidor de aplicaciones para realizar pruebas de conexión. Aunque se también se podría un crear un cliente sin especificar el servidor de aplicaciones en tiempo de diseño, pero no lo recomiendo porque es más incómodo.

Si no estamos creando la aplicación cliente en el mismo equipo que el servidor, y estamos usando una conexión DCOM, podemos registrar el servidor de aplicaciones en la máquina cliente. Esto hace que los componentes de conexión se enteren de donde está el servidor de aplicaciones en tiempo de diseño, así podemos elegir el componente Provider desde el inspector de objetos.

CREANDO UN SERVIDOR DE APLICACIONES DCOM

La mayor diferencia entre crear un servidor de aplicaciones y la típica aplicación de bases de datos cliente/servidor reside en el módulo de datos remoto. Vamos a crear una aplicación EXE que va a hacer de servidor de aplicaciones para una base de datos Firebird 2.0 que tiene una tabla llamada CLIENTES.

Para crear un servidor de aplicaciones, hay que seguir los pasos:

1. Crear un nuevo proyecto: File -> New -> Application.

2. Guardar el proyecto como ServidorDatos.exe

3. Añadimos un módulo de datos remoto al proyecto: File -> New -> Other.

4. Nos vamos a la pestaña Multitier, seleccionamos Remote Data Module y pulsamos Ok.

5. Rellenamos los campos:

CoClass Name: ServidorDCOM

Instancing: Multiple Instance

Threading Model: Apartment

6. Pulsamos Ok. Nos aparecerá una nueva ventana que representa el nuevo módulo de datos remoto creado.

7. Guardamos el módulo de datos remoto en disco con el nombre UServidorDCOM.pas

8. Insertamos en el módulo de datos remoto el componente TIBDatabase con el nombre BaseDatos.

9. Hacemos doble clic sobre el componente BaseDatos y configuramos lo siguiente:

Connection: Remote
Protocol: TCP
Database:
127.0.0.1:D:\Desarrollo\DelphiAlLimite\Multicapa\DCOM\BaseDatos.fdb
LoginPrompt: Desactivado

10. Pulsamos Ok.

11. Añadimos al módulo de datos remoto un componente TIBTransaction llamado Transaccion.

12. Vinculamos el objeto Transaccion al componente TIBDatabase a través de su propiedad DefaultTransaction.

13. Asignamos BaseDatos en la propiedad DefaultDatabase del componente Transaccion.

14. Insertamos un componente TIBQuery llamado TClientes. En su propiedad Database ponemos BaseDatos. Y su propiedad SQL escribimos:

```
SELECT * FROM CLIENTES
```

15. Hacemos doble clic en el componente TClientes y pulsamos la combinación de teclas CTRL + A para añadir todos los campos.

16. Insertamos un componente TDataSetProvider llamado DSPClientes. En su propiedad DataSet seleccionamos TClientes.

Con esto ya tenemos nuestro propio servidor de aplicaciones conectado a la base de datos Firebird. Como puede apreciarse es casi lo mismo que crear una aplicación cliente/servidor.

La diferencia está en que no es necesario instanciar en memoria el módulo de

datos remoto ya que la aplicación cliente se encargará de hacerlo.

En la siguiente parte de este artículo crearemos la aplicación cliente encargada de conectarse con este servidor de aplicaciones que hemos creado.

Pruebas realizadas en Delphi 7.

Creando aplicaciones multicapa (IV)

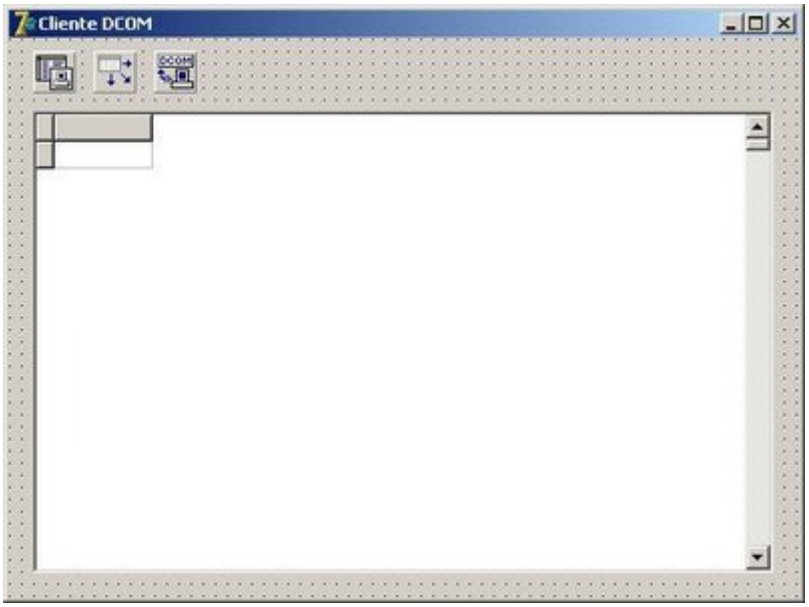
Una vez que hemos creado nuestro servidor DCOM vamos a implementar nuestra aplicación cliente que se va a encargar de llamarlo.

CREANDO LA APLICACION CLIENTE

Para conectar un cliente multicapa al servidor DCOM vamos a utilizar un componente de la clase TDOMConnection que buscará en tiempo de diseño el servidor.

Como se verá a continuación no es necesario añadir un componente de la clase TDataSetProvider ni ningún otro relacionado con motores de bases de datos específicos. Eso ya se lo hemos dejado al servidor de aplicaciones. Tampoco es necesario que el servidor de aplicaciones que creamos en el artículo anterior se esté ejecutando mientras diseñamos el cliente.

Para realizar las pruebas vamos a crear un nuevo proyecto que contenga un sólo formulario y los siguientes componentes:



- Un componente TDBGrid llamado ListadoClientes destinado a listar todos los campos de la tabla CLIENTES.

- Un componente TClientDataSet llamado TClientes.

- Un componente TDataSource que lo vamos a llamar DSClientes.

- Un componente TDCOMConnection que lo llamaremos Conexion.

Ahora vamos a vincular unos componentes a otros:

- Vinculamos el componente TDataSource llamado DSClientes con la rejilla ListadoClientes a través de su propiedad DataSource.

- Vinculamos el componente TClientDataSet llamado TClientes al componente TDataSource llamado DSCliente en su propiedad DataSet.

- Asociamos el componente TDOMConnection llamado Conexion al componente TClientDataSet llamado TClientes utilizando su propiedad RemoteServer.

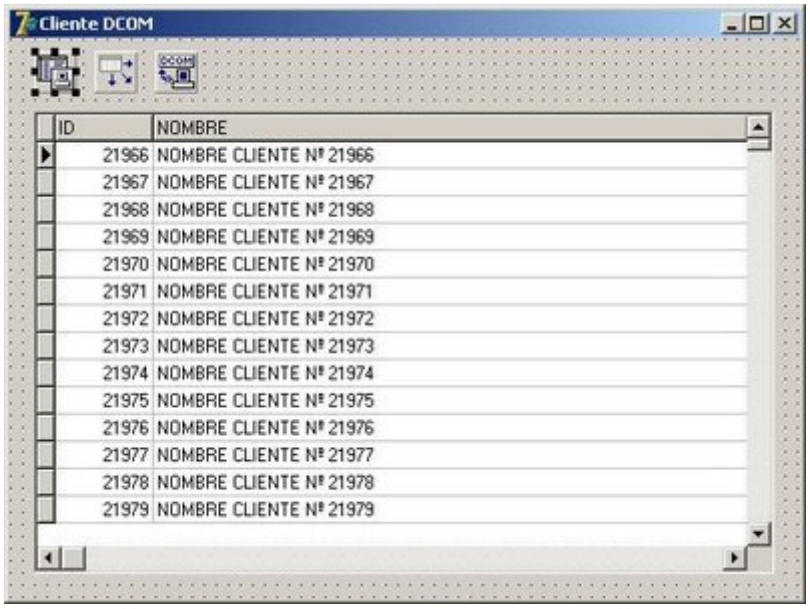
Aquí nos detenemos para analizar un problema. El componente TClientDataSet no mostrará nada que no venga de un componente TDataSetProvider. Como dicho componente se encuentra en el servidor de aplicaciones, lo primero que hay que hacer es conectar con el servidor de aplicaciones para poder vincular su DataSetProvider:

- Seleccionamos el componente TDOMConnection y en su propiedad ServerName elegimos ServidorDatos.ServidorDCOM. Al hacer esto debe rellenarnos automáticamente el campo ServerGUID, el cual es un identificador de la interfaz del módulo de datos remoto que creamos en el servidor de aplicaciones.

- Activamos la propiedad Connected en el componente TDOMConnection y veremos que se ejecuta automáticamente el servidor de aplicaciones mostrándonos su formulario principal.

- Dejando el servidor de aplicaciones en ejecución seleccionamos el componente TClientDataSet y en su propiedad ProviderName seleccionamos DSPClientes.

Con sólo realizar estos pasos, si activamos la propiedad Active del componente TClientDataSet nos mostrará en tiempo de diseño los datos de la tabla clientes:



Al compilar y ejecutar nuestro programa cliente ya tenemos un programa que maneja los datos del servidor sin preocuparse del motor de bases de datos o de otros usuarios conectados.

El componente TDOMConnection tiene la propiedad llamada ComputerName la cual contiene el nombre del equipo donde se va a alojar el servidor de aplicaciones. Si no seleccionamos ninguno se asume que el servidor de aplicaciones y la aplicación cliente residen en la misma máquina.

Si intentamos cerrar el servidor de aplicaciones mientras está conectado el cliente saldrá este mensaje:

There are still active COM objects in this application. One o more clients may have references to these objects, so manually closing this application may cause those client application(s) to fail.

Are you sure want to close this application?

Lo que traducido al castellano sería:

Todavía hay objetos COM activos en esta aplicación. Uno o más clientes podrían estar utilizando estos objetos, así que si cierra esta aplicación podría causar un error de conexión en los clientes.

¿Esta seguro de cerrar esta aplicación?

Esto significa que nunca debemos cerrar el servidor mientras quede algun cliente conectado a nosotros. Además, si el último cliente que estaba conectado al servidor de aplicaciones desconecta entonces el servidor de aplicaciones se cerrará automáticamente al ver que no hay ninguna conexión abierta.

En la siguiente parte de este artículo veremos como crear un servidor de

aplicaciones y una aplicación cliente utilizando otros protocolos distintos a DCOM.

Pruebas realizadas en Delphi 7.

Enviando un correo con INDY

Vamos a crear un procedimiento para mandar correos electrónicos utilizando el componente TIdSMTP de la paleta de componentes INDY CLIENTS.

El componente no hace falta ponerlo en el formulario, ya que lo creo en tiempo real dentro del procedimiento. Sólo hace falta añadir en el apartado USES de nuestro formulario lo siguiente:

```
uses
  IdSMTP, IdMessage;
```

Vamos con el procedimiento que envía un mensaje de correo electrónico:

```
procedure EnviarMensaje( sUsuario, sClave, sHost, sAdjunto, sAsunto,
sDestino, sMensaje: String );
var SMTP: TIdSMTP;
   Mensaje: TIdMessage;
   Adjunto: TIdAttachment;
begin
  // Creamos el componente de conexión con el servidor
  SMTP := TIdSMTP.Create( nil );
  SMTP.Username := sUsuario;
  SMTP.Password := sClave;
  SMTP.Host := sHost;
  SMTP.Port := 25;
  SMTP.AuthenticationType := atLogin;

  // Creamos el contenido del mensaje
  Mensaje := TIdMessage.Create( nil );
  Mensaje.Clear;
  Mensaje.From.Name := sDestino;
  Mensaje.From.Address := sDestino;
  Mensaje.Subject := sAsunto;
  Mensaje.Body.Text := sMensaje;
  Mensaje.Recipients.Add;
  Mensaje.Recipients.Items[0].Address := sDestino;

  // Si hay que meter un archivo adjunto lo creamos y lo asignamos al
mensaje
  if sAdjunto <> '' then
  begin
    if FileExists( sAdjunto ) then
      Adjunto := TIdAttachment.Create( Mensaje.MessageParts, sAdjunto
);
  end
  else
    Adjunto := nil;

  // Conectamos con el servidor SMTP
```

```
  try
    SMTP.Connect;
  except
    raise Exception.Create( 'Error al conectar con el servidor.' );
  end;

  // Si ha conectado enviamos el mensaje y desconectamos
  if SMTP.Connected then
  begin
    try
      SMTP.Send( Mensaje );
    except
      raise Exception.Create( 'Error al enviar el mensaje.' );
    end;

    try
      SMTP.Disconnect;
    except
      raise Exception.Create( 'Error al desconectar del servidor.' );
    end;
  end;

  // Liberamos los objetos creados
  if Adjunto <> nil then
    FreeAndNil( Adjunto );

  FreeAndNil( Mensaje );
  FreeAndNil( SMTP );

  Application.MessageBox( 'Mensaje enviado correctamente.',
    'Fin de proceso',MB_ICONINFORMATION );
end;
```

Y este es un ejemplo de envío de mensajes:

```
EnviarMensaje( 'juanito33', 'djeuE21', 'smtp.terra.es',
             'c:\documento.zip', 'Te envio mi documento',
             'felipe8843@terra.es', 'Adjunto archivo: documento.zip'
);
```

Con un poco de imaginación se puede hacer que muestre el estado de la conexión en la barra de estado e incluso una barra de progreso para ver cuanto queda por terminar de enviar.

Pruebas realizadas en Delphi 7.

Leyendo el correo con INDY

Vamos a dividir el proceso de descargar el correo en dos partes. Primero leemos las cabeceras de nuestros mensajes y después descargamos los mensajes que nos interesen. Esto puede ser útil para descartar el correo spam desde el mismo servidor.

Vamos a crear dos componentes de la paleta INDY en tiempo real, con lo cual hay que añadir al comienzo de nuestra unidad:

```
uses
  IdPOP3, IdMessage;
```

A continuación vamos a crear un procedimiento donde le pasamos los datos de nuestra cuenta de correo y vuelca el contenido en un ListView (el cual se supone que contiene tres columnas: Dirección, Asunto y Fecha/hora).

```
procedure LeerCorreo( sServidor, sUsuario, sClave: String; Mensajes:
TListView );
var
  POP3: TIdPOP3;
  Mensaje: TIdMessage;
  i: Integer;
begin
  // creamos el objeto POP3
  POP3 := TIdPOP3.Create( nil );
  POP3.Host := sServidor;
  POP3.Username := sUsuario;
  POP3.Password := sClave;
  POP3.Port := 110;

  // conectamos con el servidor
  try
    POP3.Connect;
  except
    raise Exception.Create( 'Error al conectar con el servidor.' );
  end;

  Mensaje := TIdMessage.Create( nil );
  for i := 1 to POP3.CheckMessages do
  begin
    // Leemos la cabecera del mensaje
    Mensaje.Clear;
    POP3.RetrieveHeader( i, Mensaje );

    Mensajes.Items.Add;
    Mensajes.Items[i-1].SubItems.Add( Mensaje.From.Address ); //
dirección
    Mensajes.Items[i-1].SubItems.Add( Mensaje.Subject );      //
asunto
    Mensajes.Items[i-1].SubItems.Add( DateTimeToStr( Mensaje.Date ) );
// Fecha-hora
  end;

  FreeAndNil( Mensaje );
  FreeAndNil( POP3 );
end;
```

Una vez tenemos las cabeceras del mensaje en nuestra lista ListView supongamos que haciendo doble clic se descarga el mensaje del servidor a nuestro disco duro. Antes de eso vamos a crear una clase llamada TMensaje que contenga el mensaje descargado del servidor. La implementación de la misma sería:

```
type
  TMensaje = class
    iNumero: Integer;      // Nº de mensaje dentro de nuestro buzón de
correo
```

```
    sServidor, sUsuario, sClave: String;
    sAsunto, sMensaje: String;
    Adjuntos: TStringList;
    sRutaAdjuntos: String; // Ruta donde se guardaran los archivos
adjuntos

    constructor Create;
    destructor Destroy; override;
    function Descargar: Boolean;
  end;
```

Y aquí viene la implementación de sus métodos:

```
constructor TMensaje.Create;
begin
  Adjuntos := TStringList.Create;
end;

destructor TMensaje.Destroy;
begin
  FreeAndNil( Adjuntos );
  inherited;
end;

function TMensaje.Descargar: Boolean;
var
  POP3: TIdPOP3;
  Mensaje: TIdMessage;
  i: Integer;
  sAdjunto: String; // Nombre del archivo adjunto
begin
  // creamos el objeto POP3
  POP3 := TIdPOP3.Create( nil );
  POP3.Host := sServidor;
  POP3.Username := sUsuario;
  POP3.Password := sClave;
  POP3.Port := 110;

  // Conectamos con el servidor
  try
    POP3.Connect;
  except
    raise Exception.Create( 'Error al conectar con el servidor.' );
  end;

  // Leemos todo el mensaje
  Mensaje := TIdMessage.Create( nil );
  try
    POP3.Retrieve( iNumero, Mensaje );
  except
    raise Exception.Create( 'Error al leer el mensaje.' );
  end;

  // y desconectamos del servidor
  POP3.Disconnect;

  // Mostramos el mensaje en otro formulario
  sAsunto := Mensaje.Subject;

  // ¿Tiene mensajes algún mensaje adjunto?
  if Mensaje.MessageParts.Count > 0 then
```

```
  begin
    // Leemos todas las partes del mensaje
    for i := 0 to Mensaje.MessageParts.Count - 1 do
    begin
      // ¿Esta parte es de texto?
      if ( Mensaje.MessageParts.Items[i] is TIdText ) then
        sMensaje := TIdText( Mensaje.MessageParts.Items[i] ).Body.Text
      else
        // ¿Esta parte es un archivo binaro adjunto?
        if ( Mensaje.MessageParts.Items[i] is TIdAttachment ) then
        begin
          // Guardamos el nombre del archivo adjunto en una variable
para hacerlo más legible
          sAdjunto := TIdAttachment( Mensaje.MessageParts.Items[i]
).FileName;

          // Si ya existe el archivo adjunto lo borramos para que no
de error
          if FileExists( sRutaAdjuntos + sAdjunto ) then
            DeleteFile( sRutaAdjuntos + sAdjunto );

          // Guardamos el archivo adjunto y lo añadimos a la lista de
adjuntos
          TIdAttachment( Mensaje.MessageParts.Items[i] ).SaveToFile(
sRutaAdjuntos + sAdjunto );
          Adjuntos.Add( sAdjunto );
        end
    end
  end
  else
    // Tomamos todo el mensaje como un mensaje de texto
    sMensaje := Mensaje.Body.Text;

  FreeAndNil( Mensaje );
  FreeAndNil( POP3 );

  Result := True;
end;
```

Si nos fijamos en el código fuente vemos que el método Descargar controla si el mensaje lleva archivos adjuntos o no. Aunque los mensajes de correo electrónico suelen codificarse de cuarenta leches distintas, generalmente hay dos tipos:

1º Texto plano o página web sin contenido.
2º Multiparte, donde cada parte puede ser texto plano, página web, archivo binario, etc.

En ambos casos la codificación utilizada es MIME.

Si el archivo que nos envían tiene una plantilla de página web (casi todos hoy en día) hay que complicarse un poco la vida y sacarlo mediante un navegador (browser). Eso lo dejaré para otra ocasión.

Pues bien, por último creamos el método que hace doble clic en nuestra lista de mensajes y muestra el contenido del mensaje en otro formulario:

```
procedure TFFormPrincipal.MensajesDblClick( Sender: TObject );
var i: Integer;
    Mensaje: TMensaje;
begin
  // ¿Ha selecionado un mensaje de la lista?
  if Mensajes.Selected <> nil then
  begin
    Mensaje := TMensaje.Create;
    Mensaje.iNumero := Mensajes.Selected.Index+1;
    Mensaje.sServidor := Servidor.Text;
    Mensaje.sUsuario := Usuario.Text;
    Mensaje.sClave := Clave.Text;
    Mensaje.sRutaAdjuntos := ExtractFilePath( Application.ExeName ) +
'Adjuntos\';

    if Mensaje.Descargar then
    begin
      Application.CreateForm( TFMensaje, FMensaje );
      FMensaje.Asunto.Text := Mensaje.sAsunto;
      FMensaje.Mensaje.Text := Mensaje.sMensaje;

      for i := 0 to Mensaje.Adjuntos.Count-1 do
        FMensaje.Adjuntos.Items.Add( Mensaje.Adjuntos[i] );

      FMensaje.ShowModal;
    end;

    FreeAndNil( Mensaje );
  end;
end;
```

Esta es la base de un programa lector de correo POP3 aunque realmente hay que controlar muchas más cosas, como por ejemplo el borrar del servidor los mensajes ya descargados (Ahora no lo hace).

Pruebas realizadas en Delphi 7.

Subiendo archivos por FTP con INDY

Para subir archivos por FTP utilizaremos el objeto TIdFTP de la paleta de componentes Indy Clients. Para poder utilizar dicho objeto debemos añadirlo en la seccion interface:

```
uses
  Windows, Messages, ......, IdFTP, IdComponent;
```

La unidad IdComponent la utilizaremos luego para controlar eventos del componente FTP. El objeto lo creamos en tiempo de ejecución:

```
procedure SubirArchivo( sArchivo: String );
var
  FTP: TIdFTP;
begin
  FTP := TIdFTP.Create( nil );
```

Antes de subir un archivo hay que conectar con el servidor dando usuario y password:

```
  FTP.Username := 'usuario';
  FTP.Password := 'miclave';
  FTP.Host := 'miftp.midominio.com';

  try
    FTP.Connect;
  except
    raise Exception.Create( 'No se ha podido conectar con el servidor
' + FTP.Host );
  end;
```

Ahora ya estamos listos para enviar el archivo, pero antes debemos ir al directorio del servidor donde deseamos subir el archivo:

```
  FTP.ChangeDir( '/misarchivos/copiaseguridad/' );
```

Para subir un archivo tenemos el método Put, el cual toma como primer parámetro la ruta y nombre del archivo a subir, como segundo parámetro el nombre que va a tener el archivo en el servidor (se supone que el mismo) y como tercer parámetro si deseamos añadir a un archivo ya existente o crear el archivo de nuevo:

```
  FTP.Put( sArchivo, ExtractFileName( sArchivo ), False );
```

En nuestro caso hemos subido el archivo con el mismo nombre y si hubiera otro igual lo sobrescribe. Por último nos desconectamos del servidor y eliminamos el objeto.

```
  FTP.Disconnect;
  FTP.Free;
end;
```

Como suelo comentar en artículos anteriores los componentes Indy no controlan bien la multitarea, por lo tanto si vamos a subir archivos relativamente grandes recomiendo meter el método Put dentro de un hilo de ejecución, ya que si no es así, mientras que el componente TIdFTP no termine de subir el archivo, nuestra aplicación da el aspecto de estar colgada (no se puede ni mover la ventana).

Tampoco estaría mal mostrar al usuario mediante una barra de progreso el estado de la subida del archivo. Para ello el objeto TIdFtp posee el evento OnWork el cual nos informa de los bytes subidos al servidor. El cambio es sencillo.

Primero creamos el evento OnWork:

```
procedure TFVentana.FTPWork( Sender: TObject; AWorkMode: TWorkMode;
const AWorkCount: Integer );
begin
```

```
  Barra.Position := AWorkCount div 1024;
end;
```

Se supone que Barra es un componenete TProgressBar donde vamos acumulando el tamaño del archivo enviado. En este caso he dividido los bytes subidos entre 1024 para que me devuelva la información en kilobytes.

Y ahora asociamos el evento al componente después de crearlo:

```
  FTP := TIdFTP.Create( nil );
  FTP.OnWork := FTPWork;
```

¿Cómo averiguamos el tamaño del archivo para medir el progreso? Muy fácil:

```
procedure SubirArchivo( sArchivo: String );
var
  FTP: TIdFTP;
  F: File of byte;
begin
  AssignFile( F, sArchivo );
  Reset( F );
  Barra.Max := FileSize( F ) div 1024;
  CloseFile( F );
  ....
```

Con esto le decimos a la barra de progreso que la longitud máxima de la misma es la longitud del archivo en kilobytes. Una vez comience a subir el archivo la barra de progreso se va incrementando sola según el evento OnWork.

Y por supuesto nunca se os olvide controlar en todo momento el estado de la conexión, tanto para conectar, subir el archivo o desconectarse del servidor capturando las excepciones oportunas e informando al usuario de lo que ocurre (por ejemplo con una barra de estado en la parte inferior de la ventana).

En el próximo artículo mostraré como descargar un archivo por FTP.

Y lo se, a veces los componentes Indy pueden sacar de quicio a cualquiera.

Pruebas realizadas en Delphi 7.

Descargando archivos por FTP con INDY

El compononente IdFTP que utilizamos en el artículo anterior para subir archivos es el mismo que vamos a utilizar para descargarlos. Añadimos a la sección interface:

```
uses
  Windows, Messages, ......, IdFTP, IdComponent;
```

Y creamos el procedimiento para descargar el archivo:

```
procedure DescargarArchivo( sArchivo: String );
var
  FTP: TIdFTP;
begin
  FTP := TIdFTP.Create( nil );
  FTP.OnWork := FTPWork;
  FTP.Username := 'usuario';
  FTP.Password := 'miclave';
  FTP.Host := 'miftp.midominio.com';

  try
    FTP.Connect;
  except
    raise Exception.Create( 'No se ha podido conectar con el servidor
' + FTP.Host );
  end;

  FTP.ChangeDir( '/misarchivos/copiaseguridad/' );
```

Antes de comenzar la descarga hay que averiguar el tamaño del archivo en el servidor:

```
  Barra.Max := FTP.Size( ExtractFileName( sArchivo ) ) div 1024;
```

Donde Barra es un objeto TProgressBar que colocamos para mostrar al usuario el progreso de la descarga. Ahora nos aseguramos de que el archivo a descargar no haya sido descargado anteriormente, ya que podría producir un error:

```
  if FileExists( sArchivo ) then
    DeleteFile( sArchivo );
```

Para descargar el archivo utilizaremos el método Get, el cual toma como primer parámetro la ruta y nombre del archivo a descargar en local, como segundo parámetro el nombre que va a tener el archivo en el servidor, como tercer parámetro si deseamos añadir a un archivo ya existente y como último parámetro si deseamos la opción RESUME (en caso de interrumpirse la conexión si queremos que continue por donde iba, siempre y cuando el servidor soporte dicho modo).

```
  FTP.Get( ExtractFileName( sArchivo ), sArchivo, False, False );
```

Para finalizar nos desconectamos del servidor y eliminamos el objeto.

```
  FTP.Disconnect;
  FTP.Free;
end;
```

También creamos el evento OnWork para controlar el progreso de la descarga:

```
procedure TFVentana.FTPWork( Sender: TObject; AWorkMode: TWorkMode;
const AWorkCount: Integer );
```

```
begin
  Barra.Position := AWorkCount div 1024;
end;
```

Para terminar recomiendo meter el método Get en un hilo de ejecución por si el componente se queda bloqueado en la descarga.

Pruebas realizadas en Delphi 7.

Operaciones con cadenas de texto (I)

Delphi posee un amplio repertorio de funciones para el análisis y manipulación de cadenas de texto que nos harán la vida mucho más fácil una vez las conozcamos. Comencemos con ellas:

function AnsiCompareStr(const S1, S2: string): Integer;

Esta función compara dos cadenas de texto carácter a carácter y nos dice si son iguales (diferencia mayúsculas y minúsculas). Si ambas cadenas son iguales devuelve un cero. Devolverá un 1 si la cadena S1 es superior a la cadena S2 y devuelve un -1 si la cadena S1 es inferior a la cadena S2. Veamos un ejemplo:

```
AnsiCompareStr( 'HOLA', 'HOLA' ) devuelve  0
AnsiCompareStr( 'HOLA', 'HOLa' ) devuelve  1
AnsiCompareStr( 'HOLa', 'HOLA' ) devuelve -1
```

¿Cuando se considera una cadena de texto superior a otra? Pues el orden es el siguiente:

```
Letras mayúsculas > Letras minúsculas
Letras minúsculas > Números
```

function AnsiCompareText(const S1, S2: string): Integer;

Esta función es similar a AnsiCompareStr a diferencia de que no distingue entre mayúsculas y minúsculas. En el caso anterior:

```
AnsiCompareText( 'HOLA', 'HOLA' )  devuelve 0
AnsiCompareText( 'HOLA', 'HOLa' )  devuelve 0
AnsiCompareText( 'HOLa', 'HOLA' )  devuelve 0
AnsiCompareText( 'HOLA', 'HOLLA' ) devuelve -1
AnsiCompareText( 'HOLLA', 'HOLA' ) devuelve 1
```

El orden entre cadenas se define por:

```
Letras > Números
```

function AnsiContainsStr(const AText, ASubText: string): Boolean;

Comprueba si la cadena ASubText esta dentro de la cadena AText. Por ejemplo:

```
AnsiContainsStr( 'DELPHI AL LIMITE', 'LIMITE' ) devuelve True
AnsiContainsStr( 'DELPHI AL LIMITE', 'LIMITe' ) devuelve False
```

function AnsiContainsText(const AText, ASubText: string): Boolean;

Esta función es igual a AnsiConstainsStr salvo que no diferencia mayúsculas de minúsculas. Veamos un ejemplo:

```
AnsiContainsText( 'DELPHI AL LIMITE', 'LIMITE' ) devuelve True
AnsiContainsText( 'DELPHI AL LIMITE', 'LIMITe' ) devuelve True
AnsiContainsText( 'DELPHI AL LIMITE', 'LIMITES' ) devuelve False
```

function AnsiEndsStr(const ASubText, AText: string): Boolean;

La función nos devuelve True si la cadena AText termina en la cadena ASubText. He aquí un ejemplo:

```
AnsiEndsStr( '.avi', 'C:\Archivos de
programa\Emule\Incoming\pelicula.avi' ) devuelve True
AnsiEndsStr( '.AVI', 'C:\Archivos de
programa\Emule\Incoming\pelicula.avi' ) devuelve False
```

Para este caso es mejor utilizar la función:

function AnsiEndsText(const ASubText, AText: string): Boolean;

Esta función obtiene el mismo resultado que AnsiEndsStr pero sin diferenciar mayúsculas de minúsculas. En el caso anterior:

```
AnsiEndsText( '.avi', 'C:\Archivos de
programa\Emule\Incoming\pelicula.avi' ) devuelve True
AnsiEndsText( '.AVI', 'C:\Archivos de
programa\Emule\Incoming\pelicula.avi' ) devuelve True
```

Como vemos es ideal para comprobar extensiones de archivos.

En el próximo artículo seguiremos con muchas más funciones.

Pruebas realizadas en Delphi 7.

Operaciones con cadenas de texto (II)

Continuamos viendo funciones para manejo de cadenas de caracteres:

function AnsiIndexStr(const AText: string; const AValues: array of string): Integer;

Esta función comprueba si alguna de las cadenas contenidas en el array AValues coincide extactamente con la cadena AText (distingue mayúsculas y minúsculas). Si la encuentra nos devuelve el número de índice del array donde se encuentra (empieza por cero), en caso contrario nos devuelve -1. Por ejemplo:

```
AnsiIndexStr( 'JUAN', ['CARLOS','PABLO','JUAN','ROSA'] ) devuelve  2
AnsiIndexStr( 'JUAN', ['CARLOS','PABLO','Juan','ROSA'] ) devuelve -1
```

Como vemos en el primer ejemplo nos dice que JUAN se encuentra en la posición 2 del array. Sin embargo en el sedungo ejemplo devuelve -1 porque JUAN es distinto de Juan.

function AnsiIndexText(const AText: string; const AValues: array of string): Integer;

Funciona exactamente igual que la función AnsiIndexStr salvo que no distingue mayúsculas y minúsculas. Según el ejemplo anterior ambas llamadas a la función devuelven idéntico resultado:

```
AnsiIndexText( 'JUAN', ['CARLOS','PABLO','JUAN','ROSA'] ) devuelve 2
AnsiIndexText( 'JUAN', ['CARLOS','PABLO','Juan','ROSA'] ) devuelve 2
```

function AnsiLeftStr(const AText: AnsiString; const ACount: Integer): AnsiString;

Esta función devuelve la parte izquierda de la cadena AText según el número de caracteres que le indiquemos en ACount. Sería algo así:

```
AnsiLeftStr( 'PROGRAMANDO CON DELPHI', 6 )  devuelve PROGRA
AnsiLeftStr( 'PROGRAMANDO CON DELPHI', 11 ) devuelve PROGRAMANDO
AnsiLeftStr( 'PROGRAMANDO CON DELPHI', 18 ) devuelve PROGRAMANDO CON
DE
```

function AnsiLowerCase(const S: string): string;

Devuelve la cadena S convertida toda en minúsculas (si tiene letras, naturalmente). Veamos un par de ejemplos:

```
AnsiLowerCase( 'Programando con Delphi' ) devuelve:

  programando con delphi

AnsiLowerCase( 'MANIPULANDO CADA CARÁCTER DE LA CADENA' ) devuelve:

  manipulando cada carácter de la cadena
```

Como vemos nos ha respetado la tílde en la palabra carácter.

function AnsiMatchStr(const AText: string; const AValues: array of string): Boolean;

Esta función nos dice si alguna de las cadenas contenidas en el array AValues coincide exactamente con la cadena AText (comprobando mayúsculas y minúsculas). Aunque esta función pueda parecer igual a AnsiIndexStr se diferencia en que sólo responde True o False si la encontrado o no, al contrario que AnsiIndexStr que nos devuelve que la posición donde la ha encontrado. Con un ejemplo se ve mas claro:

```
AnsiMatchStr( 'JUAN', ['CARLOS','PABLO','JUAN','ROSA'] ) devuelve True
AnsiMatchStr( 'JUAN', ['CARLOS','PABLO','Juan','ROSA'] ) devuelve
False
```

Nota: La ayuda de Delphi 7 dice que esta función devuelve un Integer y realmente devuelve un Boolean, será un error de documentación (ya estamos acostumbrados a la 'magnífica' documentación de Borland). En cambio si está corregido en la función:

function AnsiMatchText(const AText: string; const AValues: array of string): Boolean;

Similar a la función anterior AnsiMatchStr pero sin diferenciar mayúsculas y minúsculas. Siguiendo el mismo ejemplo:

```
AnsiMatchText( 'JUAN', ['CARLOS','PABLO','JUAN','ROSA'] ) devuelve
True
AnsiMatchText( 'JUAN', ['CARLOS','PABLO','Juan','ROSA'] ) devuelve
True
```

function AnsiMidStr(const AText: AnsiString; const AStart, ACount: Integer): AnsiString;

Devuelve un trozo de la cadena AText cuya posición comienza en AStart (el primer elemento es el 1) y cuyo número de caracteres viene determinado por ACount. Por ejemplo:

```
AnsiMidStr( 'PROGRAMANDO CON DELPHI', 7, 5 ) devuelve MANDO
AnsiMidStr( 'PROGRAMANDO CON DELPHI', 17, 6 ) devuelve DELPHI
```

function AnsiPos(const Substr, S: string): Integer;

Devuelve la posición de la cadena Substr que está dentro de la cadena S. Si no la encuentra devuelve un cero (el primer elemento comienza por 1). También distingue entre mayúsculas y minúsculas. Veamos como funciona:

```
AnsiPos( 'PALABRA', 'BUSCANDO UNA PALABRA' ) devuelve 14
AnsiPos( 'Palabra', 'BUSCANDO UNA PALABRA' ) devuelve  0
```

function AnsiReplaceStr(const AText, AFromText, AToText: string): string;

Esta función nos devuelve la cadena AText reemplazando las palabras que contenga según la variable AFromText sustituyéndolas por AToText. Tiene encuentra mayúsculas y minúsculas:

```
AnsiReplaceStr( 'CORRIGIENDO TEXTO DENTRO DE UNA FRASE', 'TEXTO', 'UNA
PALABRA' ) devuelve:

  CORRIGIENDO UNA PALABRA DENTRO DE UNA FRASE

AnsiReplaceStr( 'CORRIGIENDO TEXTO DENTRO DE UNA FRASE', 'Texto', 'UNA
PALABRA' ) devuelve:
```

```
  CORRIGIENDO TEXTO DENTRO DE UNA FRASE
```

Como vemos en el segundo ejemplo al no encontrar Texto por contener minúsculas ha dejado la frase como estaba.

function AnsiReplaceText(const AText, AFromText, AToText: string): string;

Igual a la función AnsiReplaceStr sin distinguir mayúsculas y minúsculas:

```
AnsireplaceText( 'CORRIGIENDO TEXTO DENTRO DE UNA FRASE', 'TEXTO',
'UNA PALABRA' ) devuelve:

  CORRIGIENDO UNA PALABRA DENTRO DE UNA FRASE

AnsireplaceText( 'CORRIGIENDO TEXTO DENTRO DE UNA FRASE', 'Texto',
'UNA PALABRA' ) devuelve:

  CORRIGIENDO UNA PALABRA DENTRO DE UNA FRASE
```

El próximo artículo continuará con más funciones de manipulación de texto.

Pruebas realizadas en Delphi 7.

Operaciones con cadenas de texto (III)

Seguimos con funciones para manejo de cadenas de caracteres:

function AnsiReverseString(const AText: AnsiString): AnsiString;

Esta función devuelve la cadena AText con los caracteres invertidos (de derecha a izquierda). Por ejemplo:

```
AnsiReverseString( 'Al revés' ) devuelve:

  séver lA

AnsiReverseString( 'INVIRTIENDO UNA CADENA DE TEXTO' ) devuelve:

  OTXET ED ANEDAC ANU ODNEITRIVNI
```

function AnsiRightStr(const AText: AnsiString; const ACount: Integer): AnsiString;

Esta función devuelve la parte derecha de la cadena AText según el número de caracteres que le indiquemos en ACount. Por ejemplo:

```
AnsiRightStr( 'PROGRAMANDO CON DELPHI', 6 )  devuelve DELPHI
AnsiRightStr( 'PROGRAMANDO CON DELPHI', 11 ) devuelve CON DELPHI
AnsiRightStr( 'PROGRAMANDO CON DELPHI', 18 ) devuelve RAMANDO CON
DELPHI
```

function AnsiStartsStr(const ASubText, AText: string): Boolean;

Devuelve True si la cadena ASubText está al comienzo de la cadena AText (distingue mayúsculas y minúsculas). Veamos como funciona:

```
AnsiStartsStr( 'C:\', 'C:\Mis documentos\' ) devuelve True
AnsiStartsStr( 'c:\', 'C:\Mis documentos\' ) devuelve False
```

function AnsiStartsText(const ASubText, AText: string): Boolean;

Esta función es igual a AnsiStartsStr salvo que no distingue mayúsculas de minúsculas. En el caso anterior:

```
AnsiStartsText( 'C:\', 'C:\Mis documentos\' ) devuelve True
AnsiStartsText( 'c:\', 'C:\Mis documentos\' ) devuelve True
```

function AnsiUpperCase(const S: string): string;

Devuelve la cadena S convertida toda en mayúsculas. Veamos un par de ejemplos:

```
AnsiUpperCase( 'Programando con Delphi' ) devuelve:

  PROGRAMANDO CON DELPHI

AnsiUpperCase( 'manipulando cada carácter de la cadena' ) devuelve:

  MANIPULANDO CADA CARÁCTER DE LA CADENA
```

procedure AppendStr(var Dest: string; const S: string); deprecated;

Este procedimiento esta obsoleto. Añade a la cadena Dest el contenido dela cadena S. Utilizar en su lugar el operador + o la función Contat.

function CompareStr(const S1, S2: string): Integer;

Compara las cadenas de texto S1 y S2 y devuelve cero si son iguales. Si son distintas devuelve un número positivo o negativo según la diferencia de caracteres. La operación de diferencia de caracteres se realiza a nivel de cada carácter utilizando su valor equivalente en ASCII. Por ejemplo:

```
CompareStr( 'HOLA', 'HOLA' ) devuelve   0
CompareStr( 'HOLA', 'HOLa' ) devuelve -32
CompareStr( 'HOLa', 'HOLA' ) devuelve  32
CompareStr( 'HOLA', 'HOYA' ) devuelve -13
```

Esta función esta obsoleta. Se recomienda utilizar AnsiCompareStr en su lugar.

function CompareText(const S1, S2: string): Integer;

Similar a la función CompareStr pero sin tener en cuenta mayúsculas y minúsculas. Según el ejemplo anterior:

```
CompareText( 'HOLA', 'HOLA' ) devuelve   0
CompareText( 'HOLA', 'HOLa' ) devuelve   0
CompareText( 'HOLa', 'HOLA' ) devuelve   0
CompareText( 'HOLA', 'HOYA' ) devuelve -13
```

Esta función esta obsoleta. Se recomienda utilizar AnsiCompareText en su lugar.

function Concat(s1 [, s2,..., sn]: string): string;

Esta función devuelve concatenadas un número indeterminado de cadenas de texto que le pasemos como parámetro, aunque se recomienda utilizar el operador + en su lugar. Veamos un ejemplo:

```
Concat( '12', '34', '56', '78' ) devuelve 12345678
Concat( 'COMO', ' ESTÁN', ' USTEDES' ) devuelve COMO ESTÁN USTEDES
```

function Copy(S; Index, Count: Integer): string;
function Copy(S; Index, Count: Integer): array;

Ambas funciones devuelven una subcadena de la cadena S en la posición Index (el primer elemento es 1) y con una longitud de Count. Por ejemplo:

```
Copy( '123456', 1, 3 ) devuelve 123
Copy( '123456', 4, 2 ) devuelve 45
Copy( 'CAPTURANDO UNA PALABRA', 16, 7 ) devuelve PALABRA
```

También puede utilizarse para copiar elementos entre arrays de cualquier tipo. Por ejemplo vamos a crear un array de enteros y vamos a copiar una parte del mismo a otro:

```
var Origen, Destino: array of Integer;
    i: Integer;
begin
  SetLength( Origen, 6 );
  Origen[0] := 10;
  Origen[1] := 20;
  Origen[2] := 30;
  Origen[3] := 40;
  Origen[4] := 50;
  Origen[5] := 60;
  Destino := Copy( Origen, 2, 4 );
  for i := 0 to Length( Destino ) - 1 do
    Memo.Lines.Add( Format( 'Destino[%d]=%d', [i,Destino[i]] ) );
end;
```

El resultado lo hemos volcado a pantalla dentro de un campo Memo mostrando lo siguiente:

```
Destino[0]=30
Destino[1]=40
Destino[2]=50
Destino[3]=60
```

En el próximo artículo seguiremos con más funciones.

Pruebas realizadas en Delphi 7.

Operaciones con cadenas de texto (IV)

Vamos a seguir con las funciones para tratamiento de cadenas de texto:

procedure Delete(var S: string; Index, Count:Integer);

Este procedimiento elimina de la cadena S (pasada por variable) un número de caracteres situados en la posición Index y cuya longitud viene determinada por Count. El primer elemento comienza por 1. Por ejemplo:

```
var
  sTexto: String;
begin
  sTexto := 'CORTANDO UNA CADENA DE TEXTO';
  Delete( sTexto, 10, 14 );
end;
```

Ahora la variable sTexto contiene:

```
CORTANDO TEXTO
```

porque hemos eliminado UNA CADENA DE.

function DupeString(const AText: string; ACount: Integer): string;

Esta función devuelve la cadena AText repetida tantas veces como se indique en ACount. Por ejemplo:

```
DupeString( 'HOLA ', 3 ) devuelve HOLA HOLA HOLA
```

function High(X);

Devuelve el valor más alto de un tipo de variable. Vamos a ver un ejemplo que utiliza esta función con distintos tipos de variable y muestra el resultado en un Memo:

```
var
  Numeros: array[1..4] of Integer;
begin
  Memo.Lines.Add( 'High( Char )=' + IntToStr( Ord( High( Char ) ) ) );
  Memo.Lines.Add( 'High( Integer )=' + IntToStr( High( Integer ) ) );
  Memo.Lines.Add( 'High( Word )=' + IntToStr( High( Word ) ) );
  Memo.Lines.Add( 'High( DWord )=' + IntToStr( High( DWord ) ) );
  Memo.Lines.Add( 'High( Boolean )=' + BoolToStr( High( Boolean ),
True ) );
  Numeros[1] := 13;
  Numeros[2] := 6;
```

```
  Numeros[3] := 16;
  Numeros[4] := 5;
  Memo.Lines.Add( 'High( Numeros )=' + IntToStr( High( Numeros ) ) );
end;
```

El resultado que muestra sería:

```
  High( Char )=255
  High( Integer )=2147483647
  High( Word )=65535
  High( DWord )=4294967295
  High( Boolean )=True
  High( Numeros )=4
```

Como vemos en el último caso, nos devuelve el último índice del del array Numeros.

procedure Insert(Source: string; var S: string; Index: Integer);

Este procedimiento inserta dentro de la cadena S (pasada como variable) la cadena Source en la posición Index. Por ejemplo:

```
var
  sTexto: String;
begin
  sTexto := 'AMPLIANDO TEXTO';
  Insert( ' UNA CADENA DE', sTexto, 10 );
end;
```

Ahora sTexto contiene:

```
AMPLIANDO UNA CADENA DE TEXTO
```

function LastDelimiter(const Delimiters, S: string): Integer;

Esta función nos devuelve la posición del último delimitador determinado por Delimiters que encuentre en la cadena S. Con unos ejemplos se ve mas claro:

```
  LastDelimiter( ',', 'PABLO,JUAN,MARIA,MARCOS' ) devuelve 17
  LastDelimiter( '.', '1.2.3.4.5' ) devuelve 8
  LastDelimiter( '-:', 'A:1-B:2-C:3' ) devuelve 10
```

En el último ejemplo hemos utilizado dos delimitadores. Esta función sería muy interesante para crear un parser de código fuente HTML buscando delimitadores <>.

function Length(S): Integer;

Devuelve la longitud máxima de una cadena de texto o de un array. Por ejemplo:

```
var
  Valores: array of Integer;
begin
  ShowMessage( Length( 'HOLA' ) );
```

```
  SetLength( Valores, 3 );
  ShowMessage( IntToStr( Length( Valores ) );
end;
```

Los comandos ShowMessage nos mostraría los valores 4 para la cadena HOLA y el valor 3, que es la longitud del array dinámico.

function LowerCase(const S: string): string;

Convierte los caracteres de la cadena S a minúsculas. Se recomenda utilizar en su lugar la función AnsiLowerCase para caracteres internacionales de 8 bits. Por ejemplo:

```
LowerCase( 'Programando con Delphi' ) devuelve:

  programando con delphi

LowerCase( 'MANIPULANDO CADA CARÁCTER DE LA CADENA' )

  manipulando cada carÁcter de la cadena
```

Como vemos en este último ejemplo la palabra CARÁCTER ha dejado la Á en mayúsculas. Por ello se recomienda utilizar la función AnsiLowerCase para evitar estas incidencias.

procedure Move(const Source; var Dest; Count: Integer);

Este procedimiento (que debería llamarse quizás Copy) copia la cantidad de bytes Count de Source a la variable Dest. Por ejemplo:

```
var
  sOrigen, sDestino: String;
begin
  sOrigen  := 'COPIANDO TEXTO';
  sDestino := '--------------';
  Move( sOrigen[10], sDestino[3], 5 );
end;
```

El valor de la variable destino sería:

```
 --TEXTO-------
```

function Pos(Substr: string; S: string): Integer;

Devuelve la posición de la cadena Substr dentro de la cadena S (Distingue mayúsculas de minúsculas). Si no la encuentra nos devuelve un cero (el primer elemento es el 1). Por ejemplo:

```
Pos( 'PALABRA', 'BUSCANDO UNA PALABRA' ) devuelve 14
Pos( 'palabra', 'BUSCANDO UNA PALABRA' ) devuelve  0
```

procedure ProcessPath(const EditText: string; var Drive: Char; var DirPart: string; var FilePart: string);

Este interesante procedimiento divide la cadena EditText la cual se supone que es una ruta como:

```
C:\Archivos de programa\MSN Messenger\msnmsgr.exe
```

en unidad, directorio y archivo. Vamos un ejemplo volcando la información en un Memo:

```
var
  sRuta, sDirectorio, sArchivo: String;
  cUnidad: Char;
begin
  sRuta := 'C:\Archivos de programa\MSN Messenger\msnmsgr.exe';
  ProcessPath( sRuta, cUnidad, sDirectorio, sArchivo );
  Memo.Lines.Add( 'sUnidad=' + cUnidad );
  Memo.Lines.Add( 'sDirectorio=' + sDirectorio );
  Memo.Lines.Add( 'sArchivo=' + sArchivo );
end;
```

El resultado sería:

```
sUnidad=C
sDirectorio=\Archivos de programa\MSN Messenger
sArchivo=msnmsgr.exe
```

Nota: para que funcione esta función la ruta que le pasemos debe ser real, en caso contrario da un error.

En el próximo artículo terminaremos con este tipo de funciones.

Pruebas realizadas en Delphi 7.

Operaciones con cadenas de texto (V)

Aquí finalizamos con las funciones de manipulación de cadenas de texto:

procedure SetLength(var S; NewLength: Integer);

Establece el tamaño de una cadena de texto o un array dinámico. Por ejemplo:

```
var
  sTexto: String;
begin
  sTexto := '123456789';
  SetLength( sTexto, 4 );
end;
```

Después de ejecutar esto sTexto será igual a 1234 (hemos recortado la cadena).

También se utiliza para establecer el tamaño de un array dinámico antes de utilizarlo.

```
var
  Valores: array of Integer;
begin
  SetLength( Valores, 3 );
  Valores[0] := 10;
  Valores[1] := 20;
  Valores[2] := 30;

  // Ampliamos el tamaño del array
  SetLength( Valores, 5 );
  Valores[3] := 40;
  Valores[4] := 50;
end;
```

procedure SetString(var S: string; Buffer: PChar; len: Integer);

El procedimiento SetString fija la longitud de la cadena S antes de copiar el número de caracteres determinado por len de la cadena Buffer. Vamos a ver un ejemplo que copia el contenido de un array de caracteres a un string:

```
var
  Original: array[1..3] of char;
  sDestino: String
begin
  Original[1] := 'A';
  Original[2] := 'B';
  Original[3] := 'C';
  SetString( sDestino, PChar( Addr( Original ) ), 3 );
end;
```

Ahora la variable sDestino vale 'ABC'.

function StringOfChar(Ch: Char; Count: Integer): string;

Esta función devuelve una nueva cadena de texto creada con el carácter Ch y repetido tantas veces como determine el parámetro Count. Por ejemplo:

```
  StringOfChar( '0', 5 ) devuelve 00000
  StringOfChar( 'A', 6 ) devuelve AAAAAA
```

Esta función viene bien para completar con ceros por la derecha o por la izquierda los números que se pasan a formato texto.

function StringReplace(const S, OldPattern, NewPattern: string; Flags: TReplaceFlags): string;

Esta función devuelve una cadena de texto reemplazando las subcadenas encontradas dentro de la cadena S que coincidan con OldPattern y serán sustituidas por la cadena NewPattern. Los valores del parámetro Flags pueden ser:

rfReplaceAll: reemplaza todas las cadenas encontradas
rfIgnoreCase: no distingue mayúsculas de minúsculas

Por ejemplo:

```
StringReplace( 'CAMBIANDO-EL-TEXTO', '-', '.', [] ) devuelve:

  CAMBIANDO.EL-TEXTO

StringReplace( 'CAMBIANDO-EL-TEXTO', '-', '.', [rfReplaceAll] )
devuelve:

  CAMBIANDO.EL.TEXTO

StringReplace( 'CAMBIANDO EL TEXTO', 'EL', 'mi',
[rfReplaceAll,rfIgnoreCase] ) devuelve:

  CAMBIANDO mi TEXTO

StringReplace( 'CAMBIANDO EL TEXTO', 'EL', 'mi', [rfReplaceAll] )
devuelve:

  CAMBIANDO mi TEXTO

StringReplace( 'cambiando el texto', 'EL', 'mi', [rfReplaceAll] )
devuelve:

  cambiando el texto

StringReplace( 'cambiando el texto', 'EL', 'mi',
[rfReplaceAll,rfIgnoreCase] ) devuelve:

  cambiando mi texto
```

function StrScan(const Str: PChar; Chr: Char): PChar;

Devuelve un puntero al primer carácter que encuentre dentro de la cadena Str. Por ejemplo:

```
StrScan( 'BUSCANDO UN CARÁCTER', 'T' ) devuelve TER
StrScan( 'BUSCANDO UN CARÁCTER', 'U' ) devuelve USCANDO UN CARÁCTER
```

function StuffString(const AText: string; AStart, ALength: Cardinal; const ASubText: string): string;

Esta función devuelve la cadena AText insertando dentro de la misma ASubText en la posición AStart y con la longitud ALength (aunque realmente no inserta sino que es una sustitución del texto que hay en esa misma posición). Veamos un ejemplo:

```
StuffString( 'INSERTANDO UN TEXTO EN UNA CADENA', 12, 2, 'EL' )
devuelve:

  INSERTANDO EL TEXTO EN UNA CADENA

StuffString( 'INSERTANDO UN TEXTO EN UNA CADENA', 12, 15, 'UNA' )
devuelve:

  INSERTANDO UNA CADENA
```

Si nos fijamos en el segundo ejemplo hemos sustituido UN TEXTO EN UNA por UNA (15 caracteres).

function Trim(const S: string): string; overload;
function Trim(const S: WideString): WideString; overload;

La función Trim devuelve la cadena S eliminando los espacios en blanco y los caracteres de control que halla a la izquierda y derecha (pero no en medio). Por ejemplo:

```
  Trim( '  ESTO ES UNA PRUEBA  ' ) devuelve 'ESTO ES UNA PRUEBA'
```

function TrimLeft(const S: string): string; overload;
function TrimLeft(const S: WideString): WideString; overload;

Esta función es similar a Trim salvo que sólo elimina los espacios en blanco y caracteres de control por la izquierda. Por ejemplo:

```
TrimLeft( '  ESTO ES UNA PRUEBA  ' ) devuelve 'ESTO ES UNA PRUEBA  '
```

function TrimRight(const S: string): string; overload;
function TrimRight(const S: WideString): WideString; overload;

Esta función es igual a Trim salvo que sólo elimina los espacios en blanco y caracteres de control por la derecha. Por ejemplo:

```
TrimRight( '  ESTO ES UNA PRUEBA  ' ) devuelve '  ESTO ES UNA PRUEBA'
```

function UpCase(Ch: Char): Char;

Convierte un carácter a mayúsculas. Por ejemplo:

```
UpCase( 'a' ) devuelve A
UpCase( 'ú' ) devuelve ú
```

Como vemos en el segundo ejemplo no funciona correctamente con caracteres Ansi. Mejor utilizar en su lugar la función AnsiUpperCase.

function UpperCase(const S: string): string;

Convierte la cadena S a mayúsculas. Por ejemplo:

```
UpperCase( 'Hola' ) devuelve HOLA
UpperCase( 'Programación en Delphi' ) devuelve PROGRAMACIóN EN DELPHI
```

El segundo ejemplo no respeta las vocales con tilde. Mejor utilizar AnsiUpperCase.

function WrapText(const Line, BreakStr: string; nBreakChars: TSysCharSet; MaxCol: Integer):string; overload;
function WrapText(const Line, MaxCol: Integer = 45):string; overload;

La función WrapText parte la cadena Line en múltiples líneas de texto separadas por defecto con los caracteres de control #13 y #10, en tamaños máximos de palabra definidos por MaxCol. Veamos algunos ejemplos:

```
WrapText( 'Partiendo una cadena de texto', 15 ) devuelve

  Partiendo una
  cadena de texto

WrapText( 'Partiendo una cadena de texto', 10 ) devuelve:

  Partiendo
  una
  cadena de
  texto

WrapText( 'Partiendo una cadena de texto', 5 ) devuelve:

  Partiendo
  una
  cadena
  de
  texto
```

Es decir, parte la cadena en frases cuyo tamaño máximo es definido por MaxCol pero respetando en ancho de cada palabra (aunque se pase del límite). También podemos definir con que caracteres vamos a separar las palabras así como que deseamos de separador. Si quisieramos separar la frase mediante por punto sería de la siguiente manera:

```
WrapText( 'Partiendo.una.cadena.de.texto', '-', ['.'], 5 ) devuelve

  Partiendo.-una.-cadena.-de.-texto
```

Hemos definido el separador (quitando #13 y #10) y también le hemos indicado el carácter que divide cada palabra (el punto).

Con esto finalizan las funciones de manipulación de texto.

Pruebas realizadas en Delphi 7.

El objeto StringList (I)

Un objeto de la clase TStringList es muy útil para procesar listas de cadenas de texto en memoria. Cada elemento de la lista puede ordenarse, insertar nuevos elementos, eliminar otros, etc.

La lista pueder ser construida de vacío o bien se pueden cargar los elementos desde un archivo de texto separando cada elemento por comas.

La clase TStringList hereda de la clase TStrings, aunque no se recomienda esta última por ser algo incompleta.

CREANDO UNA LISTA

Veamos un sencillo ejemplo para crear una lista de clientes:

```
var Clientes: TStringList;
begin
  Clientes := TStringList.Create;
  Clientes.Add( 'PEDRO SANCHEZ GARCIA' );
  Clientes.Add( 'MARIA PALAZÓN PÉREZ' );
  Clientes.Add( 'CARLOS ABENZA MARTINEZ' );
  Clientes.Add( 'ANA GONZALEZ RUIZ' );
end;
```

ACCEDIENDO A LOS ELEMENTOS DE LA LISTA

A los elementos de la lista se accede como si fuera un array dinámico (el primer elemento es el 0). Si escribimos:

```
ShowMessage( Clientes[2] );
```

nos mostrará CARLOS ABENZA MARTINEZ.

En cualquier momento se puede modificar un elemento de la lista:

```
Clientes[3] := 'ROSA GUILLÉN LUNA';
```

Se puede recorrer cada elemento y mostrarlo en un campo Memo:

```
var
  i: Integer;
begin
  for i := 0 to Clientes.Count - 1 do
    Memo.Lines.Add( Clientes[i] );
end;
```

Como muestra arriba se puede averiguar el número de elementos de la lista mediante la propiedad Count. Aunque hay una manera mucho más fácil de mostrar los elementos. Un objeto StringList dispone de la propiedad Text que nos devuelve todos los elementos de la lista separados en líneas. Por ejemplo:

```
ShowMessage( Clientes.Text ) nos devuelve:
```

```
  PEDRO SANCHEZ GARCIA
  MARIA PALAZÓN PÉREZ
  CARLOS ABENZA MARTINEZ
  ROSA GUILLÉN LUNA
```

También podemos obtener cada elemento separado por comas:

```
ShowMessage( Clientes.CommaText ) nos devuelve:

"PEDRO SANCHEZ GARCIA","MARIA PALAZÓN PÉREZ","CARLOS ABENZA
MARTINEZ","ROSA GUILLÉN LUNA"
```

Nos ha devuelto el resultado separando cada elemento con comas además de encerrar cada uno con comillas dobles. Si nos interesa cambiar las comillas dobles por comillas simples, un StringList dispone de la propiedad QuoteChar la cual define el carácter delimitador de la cadena. Después solo hay que consultar el resultado mediante el valor DelimitedText en lugar de Text:

```
  Clientes.QuoteChar := '''';  // o también Clientes.QuoteChar := #39;
  ShowMessage( Clientes.DelimitedText );
```

El resultado sería:

```
'PEDRO SANCHEZ GARCIA','MARIA PALAZÓN PÉREZ','CARLOS ABENZA
MARTINEZ','ROSA GUILLÉN LUNA'
```

Y por supuesto se puede cambiar el separador de los elementos de la lista (la coma):

```
  Clientes.Delimiter := '-';
  ShowMessage( Clientes.DelimitedText );
```

Lo cual mostraría:

```
'PEDRO SANCHEZ GARCIA'-'MARIA PALAZÓN PÉREZ'-'CARLOS ABENZA MARTINEZ'-
'ROSA GUILLÉN LUNA'
```

ALMACENANDO VALORES DOBLES

Otra característica que hace extremadamente potente a los StringList es la posibilidad de almacenar valores dobles, es decir, cada elemento de la lista puede contener un nombre y un valor a la vez. Veamos un ejemplo:

```
var
  Diccionario: TStringList;
begin
  Diccionario := TStringList.Create;
  Diccionario.CommaText := 'BOOK=LIBRO, MOUSE=RATON, TABLE=MESA';
  Memo.Lines.Add( 'BOOK = ' + Diccionario.Values['BOOK'] );
  Memo.Lines.Add( 'MOUSE = ' +Diccionario.Values['MOUSE'] );
  Memo.Lines.Add( 'TABLE = ' +Diccionario.Values['TABLE'] );
  Diccionario.Free;
end;
```

El resultado que nos mostraría dentro del campo Memo sería:

```
  BOOK = LIBRO
  MOUSE = RATON
  TABLE = MESA
```

Se podría también utilizar para almacenar un registro de una base de datos en memoria antes de ejecutar la SQL:

```
var
  Tabla: TStringList;
begin
  Tabla := TStringList.Create;
  Tabla.CommaText := 'ID=1, NOMBRE=CARLOS, DNI=65872841R,
SALDO=130.25';
  Memo.Lines.Add( 'ID = ' + Tabla.Values['ID'] );
  Memo.Lines.Add( 'NOMBRE = ' + Tabla.Values['NOMBRE'] );
  Memo.Lines.Add( 'DNI = ' +Tabla.Values['DNI'] );
  Memo.Lines.Add( 'SALDO = ' + Tabla.Values['SALDO'] );
  Tabla.Free;
end;
```

Su resultado sería:

```
  ID = 1
  NOMBRE = CARLOS
  DNI = 65872841R
  SALDO = 130.25
```

OPERACIONES QUE SE PUEDEN REALIZAR DENTRO DE UNA LISTA

Una de las operaciones más básicas de una lista es la ordenación mediante el comando Sort:

```
  Clientes.Sort;
  Memo.Lines.Add( Clientes.Text );
```

La lista quedaría de la siguiente manera:

```
  CARLOS ABENZA MARTINEZ
  MARIA PALAZÓN PÉREZ
  PEDRO SANCHEZ GARCIA
  ROSA GUILLÉN LUNA
```

Para añadir elementos hemos visto el comando Add:

```
Clientes.Add( 'LUIS GARCIA ROJO' );
```

También se puede utilizar el comando Append:

```
Clientes.Append( 'MANUEL ESCUDERO BERNAL' );
```

La diferencia esta en que Add devuelve la posición donde se ha insertado el elemento, en cambio Append no devuelve nada. La lista quedaría de la siguiente manera:

```
CARLOS ABENZA MARTINEZ
MARIA PALAZÓN PÉREZ
```

```
PEDRO SANCHEZ GARCIA
ROSA GUILLÉN LUNA
LUIS GARCIA ROJO
MANUEL ESCUDERO BERNAL
```

Para añadir elementos a la lista también disponemos del comando Insert el cual toma como primer parámetro en que posición de la lista deseamos insertar el elemento y como segundo parámetro el elemento en cuestión. Por ejemplo vamos a introducir un cliente entre CARLOS y MARIA:

```
Clientes.Insert( 1, 'PASCUAL CAMPOY FERNANDEZ' );
```

Y la lista quedaría así:

```
CARLOS ABENZA MARTINEZ
PASCUAL CAMPOY FERNANDEZ
MARIA PALAZÓN PÉREZ
PEDRO SANCHEZ GARCIA
ROSA GUILLÉN LUNA
LUIS GARCIA ROJO
MANUEL ESCUDERO BERNAL
```

En el próximo artículo seguiremos viendo más cosas que se pueden hacer con un StringList.

Pruebas realizadas en Delphi 7.

El objeto StringList (II)

Vamos a seguir viendo las características de la clase TStringList.

ELIMINANDO ELEMENTOS DE LA LISTA

Con el método Delete se elimina un elementos de la lista la cual vuelve a reagrupar sus elemento asignando un nuevo índice. Actualmente la lista de clientes contiene:

```
CARLOS ABENZA MARTINEZ
PASCUAL CAMPOY FERNANDEZ
MARIA PALAZÓN PÉREZ
PEDRO SANCHEZ GARCIA
ROSA GUILLÉN LUNA
LUIS GARCIA ROJO
MANUEL ESCUDERO BERNAL
```

Vamos a eliminar a ROSA de la lista:

```
Clientes.Delete( 4 );
```

Ahora el cliente LUIS tiene un índice de 4 (en vez de 5 como antes), es decir, al contrario de un array dinámico, si en un objeto StringList se elimina un elemento de la lista no queda ningún hueco, ya que vuelven a agruparse sus elementos.

Un error común para eliminar todos los elementos de la lista sería hacer lo siguiente:

```
var
  i: Integer;
begin
  for i := 0 to Clientes.Count - 1 do
    Clientes.Delete( i );
end;
```

Nos daría el error:

List index out of bounds(3)

¿Por qué ocurre esto? Pues por la sencilla razón de que cada que se elimina un elemento, al reagruparse el resto de los elementos se va también disminuyendo el número de los mismos y sus índices, con lo cual cuando intenta acceder a un elemento que ya no existe provoca un error. La manera correcta de hacerlo sería:

```
var
  i: Integer;
begin
  for i := 0 to Clientes.Count - 1 do
    Clientes.Delete( 0 );
end;
```

Como vemos arriba se elimina sólo el primer elemento y como se van eliminando como si fuera una pila de platos al final desaparecen todos. Naturalmente una clase StringList dispone del método Clear que elimina todos los elementos de la lista:

```
Clientes.Clear;
```

MOVIENDO ELEMENTOS EN LA LISTA

Supongamos que tenemos la anterior lista de clientes:

```
CARLOS ABENZA MARTINEZ
PASCUAL CAMPOY FERNANDEZ
MARIA PALAZÓN PÉREZ
PEDRO SANCHEZ GARCIA
LUIS GARCIA ROJO
MANUEL ESCUDERO BERNAL
```

Vamos a mover a MARIA a la primera posición de la lista:

```
Clientes.Move( 2, 0 );
```

El comando Move acepta como primer parámetro en que posición esta el elemento que deseamos mover y como segundo parámetro su destino. En el caso anterior llevamos a MARIA de la posición 2 a la 0:

```
MARIA PALAZÓN PÉREZ
```

```
CARLOS ABENZA MARTINEZ
PASCUAL CAMPOY FERNANDEZ
PEDRO SANCHEZ GARCIA
LUIS GARCIA ROJO
MANUEL ESCUDERO BERNAL
```

Ahora bien, no hay que confundirse el mover elementos en la lista con intercambiarlos. El mover un elemento sólo lo elimina de la posición actual y lo lleva a la posición deseada, pero mantiene el orden natural de la lista. Si lo que queremos es intercambiar dos elementos para ello disponemos del método Exchange el cual tiene los mismos parámetros que Move salvo que lo que hace es intercambiar las posiciones de ambos elementos.

Vamos a ver como cambiar la posición de MANUEL (que es la 5) con la de CARLOS (que es la 1):

```
Clientes.Exchange( 5, 1 );
```

Su resultado sería:

```
MARIA PALAZÓN PÉREZ
MANUEL ESCUDERO BERNAL
PASCUAL CAMPOY FERNANDEZ
PEDRO SANCHEZ GARCIA
LUIS GARCIA ROJO
CARLOS ABENZA MARTINEZ
```

BUSCANDO ELEMENTOS EN LA LISTA

Se puede averiguar la posición de un elemento en la lista con con los métodos IndexOf e IndexOfName:

```
function IndexOf( const S: string ): Integer; override;
function IndexOfName(const Name: string): Integer; virtual;
```

El método IndexOf toma como parámetro el nombre del elemento que se desea buscar y devuelve su índice si lo encuentra (el primer elemento es el cero) y -1 si no lo encuentra. Por ejemplo:

```
Clientes.IndexOf( 'PEDRO SANCHEZ GARCIA' ) devuelve 3
Clientes.IndexOf( 'PEDRO SANCHEZ' ) devuelve -1
```

La función IndexOf sólo busca el primer elemento encontrado en la lista, de modo que si hay dos elementos iguales sólo muestra el índice del primero (de ahí la importancia de tener la lista ordenada).

Después tenemos la función IndexOfName destinada a buscar elementos de tipo NOMBRE = VALOR como vimos anteriormente con el diccionario:

```
Diccionario.IndexOfName( 'BOOK' )  devuelve 0
Diccionario.IndexOfName( 'MOUSE' ) devuelve 1
Diccionario.IndexOfName( 'TABLE' ) devuelve 2
```

Por último tenemos la función Find utilizada para buscar elementos sólo si la lista esta ordenada. Por ejemplo:

```
var
  iPosicion: Integer;
  bEncontrado: Boolean;
begin
  Clientes.Sort; // ordenamos la lista
  bEncontrado := Clientes.Find( 'PEDRO SANCHEZ GARCIA', iPosicion );
end;
```

La función Find es similar a función IndexOf añadiendo como segundo parámetro una variable donde depositar la posición del elemento buscado y cuyo valor devuelto en la función será True o False dependiendo si ha encontrado el elemento en la lista.

Después de ejecutar este código la lista quedaría de la siguiente manera:

```
CARLOS ABENZA MARTINEZ
LUIS GARCIA ROJO
MANUEL ESCUDERO BERNAL
MARIA PALAZÓN PÉREZ
PASCUAL CAMPOY FERNANDEZ
PEDRO SANCHEZ GARCIA
```

La variable bEncontrado valdría True y iPosicion sería igual a 5.

En el próximo artículo terminaremos de ver todas las funcionalidades de un StringList.

Pruebas realizadas en Delphi 7.

El objeto StringList (III)

IMPORTAR Y EXPORTAR ELEMENTOS DE LA LISTA

Se pueden guardar los elementos de la lista en un archivo de texto utilizando para ello el método SaveToFile:

procedure SaveToFile(const FileName: string); virtual;

Por ejemplo vamos a guardar nuestra lista de clientes en el mismo directorio donde ejecutamos el programa:

```
Clientes.SaveToFile( ExtractFilePath( Application.ExeName ) +
'clientes.txt' );
```

Si más adelante deseamos recuperar el listado utilizamos el procedimiento:

procedure LoadFromFile(const FileName: string); virtual;

Por ejemplo:

```
Clientes.LoadFromFile( ExtractFilePath( Application.ExeName ) +
'clientes.txt' );
```

Si tenemos dos listas en memoria también podemos copiar elementos de una lista a otra. Por ejemplo:

```
var
  Animales, Animales2: TStringList;
begin
  Animales := TStringList.Create;
  Animales2 := TStringList.Create;

  Animales.Add( 'PERRO' );
  Animales.Add( 'GATO' );
  Animales.Add( 'LEÓN' );

  Animales2.Add( 'VACA' );
  Animales2.Add( 'PATO' );
  Animales2.Add( 'ÁGUILA' );

  // Añadimos a la lista de ANIMALES el contenido de ANIMALES2 y
ordenamos la lista
  Animales.AddStrings( Animales2 );
  Animales.Sort;

  // Mostramos el resultado en un campo Memo
  Memo.Lines.Add( Animales.Text );

  Animales2.Free;
  Animales.Free;
end;
```

El resultado de la lista Animales después de ordenarla sería:

```
ÁGUILA
GATO
LEÓN
PATO
PERRO
VACA
```

Otra de las cosas que se pueden hacer es asignar los elementos de una lista a otra, eliminando los elementos de la lista original (sería algo así como copiar y pegar). Por ejemplo si queremos sustituir todos los elementos de la lista Animales por los de Animales2 habría que hacer:

```
Animales.Assign( Animales2 );
```

lo cual elimina todos los elementos de la lista Animales e inserta todos los de Animales2.

LISTAS ORDENADAS AUTOMÁTICAMENTE

Anteriormente vimos como una lista podía ordenarse con el método Sort, pero

resulta algo molesto y lento tener que volver a ordenar la lista cada vez que se inserta, modifica o elimina un elemento de la misma. Para evitar eso los objetos StringList disponen de la propiedad booleana Sorted la cual una vez esta activa ordenará todos los elementos de la lista automáticamente independientemente de las operaciones que se realicen en la misma.

Vamos a crear una nueva lista ordenada:

```
var
  Vehiculos: TStringList;
begin
  Vehiculos := TStringList.Create;
  Vehiculos.Sorted := True;
  Vehiculos.Add( 'RENAULT' );
  Vehiculos.Add( 'PEUGEOT' );
  Vehiculos.Add( 'MERCEDES' );
  Memo.Lines.Add( Vehiculos.Text );
  Vehiculos.Free;
end;
```

Al ejecutar este código mostrará en el campo Memo los elemento ya ordenados:

```
MERCEDES
PEUGEOT
RENAULT
```

Tanto si insertamos o eliminamos elementos, la lista seguirá ordenada:

```
Vehiculos.Delete( 3 );
Vehiculos.Add( 'BMW' );
```

Quedaría se la siguiente manera:

```
BMW
MERCEDES
PEUGEOT
```

Lo que no se puede hacer en una lista ordenada es modificar sus elementos:

```
Vehiculos[1] := 'AUDI';
```

Ya que provocaría el error:

Operation not allowed on sorted list (operación no permitida en una lista ordenada).

Para ello habría que desconectar la propiedad Sorted, modificar el elemento y volver a activarla:

```
  Vehiculos.Sorted := False;
  Vehiculos[1] := 'AUDI';
  Vehiculos.Sorted := True;
```

Ahora si tendríamos el efecto deseado:

```
AUDI
BMW
PEUGEOT
```

Otra de las operaciones que no se pueden hacer en una lista ordenada es añadir elementos duplicados. En el caso anterior si hacemos:

```
Vehiculos.Add( 'BMW' )
```

no hará absolutamente nada. Para poder añadir elementos duplicados en una lista ordenada hay que modificar la propiedad Duplicates la cual puede contener los siguientes valores:

dupIgnore -> Ignora el añadir elementos duplicados (así es como está por defecto)
dupAccept -> Acepta elementos duplicados
dupError -> Da un error al añadir elementos duplicados

Si hacemos lo siguiente:

```
Vehiculos.Duplicates := dupAccept;
Vehiculos.Add( 'BMW' );
```

Entonces si funcionará correctamente:

```
AUDI
BMW
BMW
PEUGEOT
```

Pero si activamos:

```
Vehiculos.Duplicates := dupError;
Vehiculos.Add( 'BMW' );
```

Nos mostrará el mensaje:

String list does not allow duplicates (StringList no acepta duplicados).

ASOCIANDO OBJETOS A LOS ELEMENTOS DE UNA LISTA

Una de las características más importantes de una lista TStringList es la de asociar a cada elemento de la lista la instacia de un objeto creado. Vamos a ver un ejemplo donde voy a crear la clase TProveedor y voy a utilizarla para crear una lista de proveedores donde cada elemento tiene su propio objeto:

```
type
  TProveedor = class
    sNombre: String;
    cSaldo: Currency;
    bPagado: Boolean;
  end;

var
```

```
  Proveedores: TStringList;
  Proveedor: TProveedor;
begin
  Proveedores := TStringList.Create;

  Proveedor := TProveedor.Create;
  with Proveedor do
  begin
    sNombre := 'SUMINISTROS PALAZÓN';
    cSaldo := 1200.24;
    bPagado := False;
  end;

  Proveedores.AddObject( 'B78234539', Proveedor );

  Proveedor := TProveedor.Create;
  with Proveedor do
  begin
    sNombre := 'ELECTRIDAUTON, S.L.';
    cSaldo := 2460.32;
    bPagado := True;
  end;

  Proveedores.AddObject( 'B98654782', Proveedor );

  Proveedor := TProveedor.Create;
  with Proveedor do
  begin
    sNombre := 'FONTANERIA MARTINEZ, S.L.';
    cSaldo := 4800.54;
    bPagado := False;
  end;

  Proveedores.AddObject( 'B54987374', Proveedor );
end;
```

Hemos creado una lista de proveedores e instanciado tres proveedores asociandolos a la lista mediante el C.I.F. del proveedor. Ahora bien, ¿como accedemos a cada proveedore de la lista? Pues muy fácil. El objeto StringList dispone de la propiedad Objects la cual nos devuelve la referencia a un objeto en cuestión. Por ejemplo vamos a buscar a FONTANERIA MARTINEZ por su C.I.F.:

```
var
  iProveedor: Integer;
begin
  iProveedor := Proveedores.IndexOf( 'B54987374' );

  if iProveedor > -1  then
  begin
    Proveedor := Proveedores.Objects[iProveedor] as TProveedor;
    Memo.Lines.Add( '' );
    Memo.Lines.Add( 'BUSCANDO AL PROVEEDOR: B54987374' );
    Memo.Lines.Add( '' );
    Memo.Lines.Add( 'NOMBRE: ' + Proveedor.sNombre );
    Memo.Lines.Add( 'SALDO: ' + CurrToStr( Proveedor.cSaldo ) );
    Memo.Lines.Add( 'PAGADO: ' + BoolToStr( Proveedor.bPagado, True )
);
  end;
end;
```

El resultado sería:

```
BUSCANDO AL PROVEEDOR: B54987374

NOMBRE: FONTANERIA MARTINEZ, S.L.
SALDO: 4800,54
PAGADO: False
```

Esto es algo mucho más útil que utilizar un array:

```
var
  Proveedores: array of TProveedor;
  ....
```

ya que es mucho más flexible y no hay que preocuparse por los huecos al añadir o eliminar registros. Esto nos permite volcar información de registros de la base de datos a memoria mapeando el contenido de un registro (Clientes, Facturas, etc) dentro de su clase correspondiente (Por ejemplo: se podrían cargar una cantidad indeterminada de albaranes del mismo cliente para sumarlos y pasarlos a factura).

Para añadir objetos a un StringList aparte de la función AddObject se pueden insertar objetos mediante el procedimiento:

procedure InsertObject(Index: Integer; const S: string; AObject: TObject); override;

el cual inserta el elemento en la posición que deseemos desplazando el resto de la lista. Otra de las ventajas de asociar objetos a un StringList en vez de utilizar un array es que al liberarlo de memoria también elimina cada instancia de objeto asociado a cada elemento de la lista:

```
Proveedores.Free;
```

no teniendo así que preocuparnos con errores de liberación de memoria.

Con esto terminados de vez las características más importantes de la clase TStringList.

Pruebas realizadas en Delphi 7.

Convertir cualquier tipo de variable a String (I)

Vamos a ver de que funciones dispone Delphi para pasar cualquier tipo de variable a una cadena de texto (String):

function CurrToStr(Value: Currency): string; overload;

Convierte un valor Currency a String. Por ejemplo:

```
CurrToStr( 1234.5678 ) devuelve 1234,5678
```

function CurrToStrF(Value: Currency; Format: TFloatFormat; Digits: Integer): string; overload;

Convierte un valor Currency a String usando un formato específico determinado por el parámetro Format y por el número de dígitos Digits. Por ejemplo:

```
CurrToStrF( 1234.5678, ffCurrency, 2 ) devuelve 1.234,57 €
CurrToStrF( 1234.5678, ffCurrency, 4 ) devuelve 1.234,5678 €
CurrToStrF( 1234.5678, ffCurrency, 0 ) devuelve 1.235 €
```

function DateTimeToStr(DateTime: TDateTime): string; overload;

Convierte un valor TDateTime a String. Por ejemplo:

```
var
  dt: TDateTime;
begin
  dt := StrToDateTime( '20/06/2007 11:27' );
  ShowMessage( DateTimeToStr( dt ) );
  dt := StrToDateTime( '20/06/2007 00:00' );
  ShowMessage( DateTimeToStr( dt ) );
end;
```

El resultado de ejecutar esto sería:

```
20/06/2007 11:27:00
20/06/2007
```

procedure DateTimeToString(var Result: string; const Format: string; DateTime: TDateTime); overload;

Convierte un valor TDateTime a String usando un formato especificado por el parámetro Format y devolviendo el valor en la variable Result. Por ejemplo:

```
var
  dt: TDateTime;
  s: String;
begin
  dt := StrToDateTime( '20/06/2007 11:27' );
  DateTimeToString( s, 'd/m/y', dt );
```

```
  ShowMessage( s );
  DateTimeToString( s, 'dd/mm/yy', dt );
  ShowMessage( s );
  DateTimeToString( s, 'dd/mm/yyyy', dt );
  ShowMessage( s );
  DateTimeToString( s, 'dd-mm-yyyy', dt );
  ShowMessage( s );
  DateTimeToString( s, 'ddd, d ''de'' mmm ''de'' yyyy', dt );
  ShowMessage( s );
  DateTimeToString( s, 'dddd, d ''de'' mmmm ''de'' yyyy', dt );
  ShowMessage( s );
  DateTimeToString( s, 'dd/mm/yyyy hh:nn:ss', dt );
  ShowMessage( s );
  DateTimeToString( s, 'dd/mm/yyyy hh:nn:ss am/pm', dt );
  ShowMessage( s );
  DateTimeToString( s, 'ddddd', dt );
  ShowMessage( s );
  DateTimeToString( s, 'dddddd', dt );
  ShowMessage( s );
end;
```

Mostraría los siguientes resultados:

```
20/6/07
20/06/07
20/06/2007
20-06-2007
mié, 20 de jun del año 2007
miércoles, 20 de junio del año 2007
20/06/2007 11:27:00
20/06/2007
miércoles, 20 de junio de 2007
```

Estas son todas las letras para dar formato a un campo fecha/hora:

```
y        = Los dos últimos dígitos del año (sin completar con cero por
la izquierda)
yy       = Los dos últimos dígitos del año (completando con cero por
la izquierda)
yyyy     = Los 4 dígitos del año
m        = Los dos dígitos del mes (sin completar con cero por la
izquierda)
mm       = Los dos dígitos del mes (completando con cero por la
izquierda)
mmm      = El nombre del mes en formato corto (Ene, Feb, Mar, etc.)
mmmm     = El nombre del mes en formato largo (Enero, Febrero, Marzo,
etc.)
d        = Los dos dígitos del día (sin completar con cero por la
izquierda)
dd       = Los dos dígitos del día (completando con cero por la
izquierda)
ddd      = El nombre del día en formato corto (Lun, Mar, Mié, etc.)
dddd     = El nombre del día en formato largo (Lunes, Martes,
Miercoles, etc.)
ddddd    = Fecha en formato abreviado (20/06/2007)
dddddd   = Fecha en formato extendido (miércoles, 20 de junio de 2007)

c     = Formato corto de fecha y hora (20/06/2007 11:27:00)
h     = Los dos dígitos de la hora (sin completar con cero por la
izquierda)
```

```
hh    = Los dos dígitos de la hora (completando con cero por la
izquierda)
n     = Los dos dígitos de los minutos (sin completar con cero por la
izquierda)
nn    = Los dos dígitos de los minutos (completando con cero por la
izquierda)
s     = Los dos dígitos de los segundos (sin completar con cero por la
izquierda)
ss    = Los dos dígitos de los segundos (completando con cero por la
izquierda)
z     = Los dígitos de los milisegundos (sin completar con cero por la
izquierda)
zzz   = Los 3 dígitos de los segundos (completando con cero por la
izquierda)
t     = Formato abreviado de hora (11:27)
tt    = Formato extendido de hora (11:27:00)

am/pm   = Formato de hora am/pm
a/p     = Formato de hora a/p
ampm    = Igual que a/p pero con TimeAMString, TimePMString
/       = Sustituido por el valor de DateSeparator
:       = Sustituido por el valor de TimeSeparator
```

Y estas son las variables de los formatos predeterminados para fecha/hora:

```
DateSeparator             = /
TimeSeparator             = :
ShortDateFormat           = dd/mm/yyyy
LongDateFormat            = dd mmm yyyy
TimeAMString              = AM
TimePMString              = PM
ShortTimeFormat           = hh:mm
LongTimeFormat            = hh:mm:ss
ShortMonthNames           = Jan Feb ...
LongMonthNames            = Enero, Febrero ...
ShortDayNames             = Lun, Mar ...
LongDayNames              = Lunes, Martes ...
TwoDigitCenturyWindow     = 50
```

function DateToStr(Date: TDateTime): string; overload;

Convierte un valor TDateTime a String. Por ejemplo:

```
var
  d: TDate;
begin
  d := StrToDate( '20/08/2007' );
  ShowMessage( DateToStr( d ) );  // Mostraría 20/08/2007
end;
```

En el próximo artículo continuaremos viendo más funciones de conversión.

Pruebas realizadas en Delphi 7.

Convertir cualquier tipo de variable a String (II)

Vamos a seguir viendo funciones para la conversión de cualquier tipo de variable a String:

function FloatToStr(Value: Extended): string; overload;

Convierte un valor Extended en String. Por ejemplo:

```
FloatToStr( 1234.5678 ) devuelve 1234,5678
```

function FloatToStrF(Value: Extended; Format: TFloatFormat; Precision, Digits: Integer): string; overload;

Convierte un valor Extended en String usando el formato, precisión y dígitos según los parámetros Format, Precision y Digits respectivamente. Por ejemplo:

```
FloatToStrF( 1234.5678, ffCurrency, 8, 2 ) devuelve 1.234,57 €
FloatToStrF( 1234.5678, ffCurrency, 8, 4 ) devuelve 1.234,5678 €
FloatToStrF( 1234.5678, ffGeneral, 8, 2 )  devuelve 1234,5678
FloatToStrF( 1234.5678, ffGeneral, 8, 4 )  devuelve 1234,5678
FloatToStrF( 1234.5678, ffExponent, 8, 2 ) devuelve 1,2345678E+03
FloatToStrF( 1234.5678, ffExponent, 8, 4 ) devuelve 1,2345678E+0003
FloatToStrF( 1234.5678, ffFixed, 8, 2 )    devuelve 1234,57
FloatToStrF( 1234.5678, ffFixed, 8, 4 )    devuelve 1234,5678
FloatToStrF( 1234.5678, ffNumber, 8, 2 )   devuelve 1.234,57
FloatToStrF( 1234.5678, ffNumber, 8, 4 )   devuelve 1.234,5678
```

function Format(const Format: string; const Args: array of const): string; overload;

Esta función es similar al comando printf del lenguaje C. Su misión es dar múltiples formatos a una misma cadena de texto según sus argumentos. Con algunos ejemplos se verá más claro:

```
Format( '%d', [1234] )      devuelve -1234
Format( '%e', [1234.5678] ) devuelve 1,23456780000000E+003
Format( '%f', [1234.5678] ) devuelve 1234,57
Format( '%g', [1234.5678] ) devuelve 1234,5678
Format( '%n', [1234.5678] ) devuelve 1.234,57
Format( '%m', [1234.5678] ) devuelve 1.234,57 €
sTexto := 'Prueba';
Format( '%p', [sTexto] )    devuelve 0012FE14
Format( '%s', [sTexto] )    devuelve Prueba
Format( '%u', [1234] )      devuelve 1234
Format( '%x', [1234] )      devuelve 4D2
```

Cada los siguientes tipos de formato:

```
d     = Decimal (Integer)
e     = Científico
```

```
f     = Punto fijo
g     = General
m     = Moneda
n     = Número (Real)
p     = Puntero (La dirección de memoria)
s     = String
u     = Decimal sin signo
x     = Hexadecimal
```

Dentro de una misma cadena de formato se pueden introducir distintos parámetros:

```
var
  sSerie: String;
  iNumero: Integer;
  rImporte: Real;
begin
  sSerie := 'A';
  iNumero := 5276;
  rImporte := 120.35;
  ShowMessage( Format( 'Factura nº %s-%d -> Importe %f', [sSerie,
iNumero, rImporte] ) );
end;
```

Al ejecutarlo muestra:

```
  Factura nº A-5276 -> Importe 120,35
```

De otra manera tendríamos que hacer:

```
ShowMessage( 'Factura nº ' + sSerie + '-' + IntToStr( iNumero ) + ' ->
Importe ' + FloatToStr( rImporte ) );
```

function FormatCurr(const Format: string; Value: Currency): string; overload;

Convierte un valor Currency a String según el formato determinado por el parámetro Format. Por ejemplo:

```
FormatCurr( '#####', 1234.5678 )    devuelve 1235
FormatCurr( '00000', 1234.5678 )    devuelve 01235
FormatCurr( '########', 1234.5678 ) devuelve 1235
FormatCurr( '00000000', 1234.5678 ) devuelve 00001235
FormatCurr( '0.####', 1234.5678 )   devuelve 1234,5678
FormatCurr( '0.00000', 1234.5678 )  devuelve 1234,56780
```

function FormatDateTime(const Format: string; DateTime: TDateTime): string; overload;

Esta función es similar al procedimiento DateTimeToString para dar formato a un campo TDateTime. Por ejemplo:

```
var
  dt: TDateTime;
begin
  dt := StrToDateTime( '20/06/2007 11:27' );
  FormatDateTime( 'd/m/y', dt );      // devuelve 20/6/07
```

```
  FormatDateTime( 'dd/mm/yyyy', dt ); // devuelve 20/06/2007
  FormatDateTime( 'dd-mm-yyyy', dt ); // devuelve 20-06-2007
end;
```

function FormatFloat(const Format: string; Value: Extended): string; overload;

Convierte un valor Extended a String utilizar el formato determinado por Format. Es similar a la función FormatCurr. Por ejemplo:

```
FormatFloat( '#####', 1234.5678 )    devuelve 1235
FormatFloat( '00000', 1234.5678 )    devuelve 01235
FormatFloat( '########', 1234.5678 ) devuelve 1235
FormatFloat( '00000000', 1234.5678 ) devuelve 00001235
FormatFloat( '0.####', 1234.5678 )   devuelve 1234,5678
FormatFloat( '0.00000', 1234.5678 )  devuelve 1234,56780
```

function IntToHex(Value: Integer; Digits: Integer): string; overload;
function IntToHex(Value: Int64; Digits: Integer): string; overload;

Estas dos funciones convierten un valor Integer o Int64 a String en formato hexadecimal, especificando el número de dígitos a través del parámetro Digits. Por ejemplo:

```
IntToHex( 123456, 6 ) devuelve 01E240
IntToHex( 123456, 5 ) devuelve 1E240
IntToHex( 123456, 4 ) devuelve 1E240
```

function IntToStr(Value: Integer): string; overload;
function IntToStr(Value: Int64): string; overload;

Ambas funciones convierten un valor Integer o Int64 a String en formato decimal. Por ejemplo:

```
IntToStr( 123456 ) devuelve 123456
```

procedure Str(X [: Width [: Decimals]]; var S);

Este procedimiento da un formato a la cadena de texto S la cual esta en una variable. Además da la posibilidad de especificar el ancho en dígitos del número y sus decimales. Por ejemplo:

```
var
  sTexto: String;
begin
  Str( 1234, sTexto );           // sTexto = '1234'
  Str( 1234:10, sTexto );        // sTexto = '      1234'
  Str( 1234.5678:10:2, sTexto ); // sTexto = '   1234.57'
  Str( 1234.5678:10:4, sTexto ); // sTexto = ' 1234.5678'
end;
```

function WideCharToString(Source: PWideChar): string;

Convierte una cadena de texto Unicode (16 bits) a String (8 bits). Por ejemplo:

```
var
  Unicode: array[0..4] of WideChar;
  sTexto: String;
begin
  Unicode[0] := 'H';
  Unicode[1] := 'o';
  Unicode[2] := 'l';
  Unicode[3] := 'a';
  Unicode[4] := #0;  // Terminamos la cadena
  sTexto := WideCharToString( Unicode );
end;
```

Pruebas realizadas en Delphi 7.