

Avanzando en Delphi.

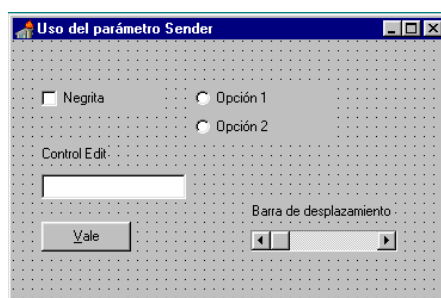
- 1. COMPARTIR GESTORES DE EVENTOS**
- 2. CUÁNDO NO ES NECESARIA UNA FICHA**
- 3. MENÚS.**
- 4. ERRORES EN EJECUCIÓN**
- 5. CUADROS DE DIÁLOGO DE USO COMÚN**
- 6. OTRAS COMPONENTES BÁSICAS**
- 7. FICHEROS EN DELPHI**
- 8. APLICACIONES MDI**

1. COMPARTIR GESTORES DE EVENTOS

Ya sabemos lo que es un gestor de eventos, cómo crearlo y asociarlo a un determinado componente.

Supongamos que deseamos diseñar un programa en el que el título de la ficha vaya cambiando, mostrando el título del control activo. Podríamos escribir un gestor para el evento *OnEnter* de cada componente, pero no es lo más adecuado, pues nos ocuparía mucho código. Lo más conveniente es escribir un solo gestor de evento que será utilizado por todos los componentes de la ficha. Para ello haremos uso del parámetro **Sender**, (que es una referencia al objeto que ha sido origen del evento) que reciben siempre en primer lugar todos los métodos de respuesta a eventos y de los moldeadores de tipo.

Veamos como se comparten los gestores de eventos con un ejemplo. Diseñemos una ficha como la que vemos en la siguiente figura:



Crearemos un método genérico de respuesta y lo asociaremos al evento *OnEnter* de cada uno de los controles de la ficha. Escribiremos el código necesario para determinar el tipo de control de que se trata, obteniendo el título o el nombre:

```
procedure TForm1.ComponenteEnter(Sender: TObject);
```

```
begin
```

```
    If Sender Is TCheckBox Then
```

```
        Caption:= 'Caja de selección : ' + (Sender As TCheckBox).Caption Else
```

```
    If Sender Is TRadioButton Then
```

```
        Caption:= 'Botón de radio : ' + (Sender As TRadioButton).Caption Else
```

```
    If Sender Is TButton Then
```

```
        Caption:= 'Control : ' + (Sender As TButton).Caption Else
```

```
        Caption := 'Control : ' + (Sender As Tcomponent).Name;
```

end;

Al ejecutar el programa podremos ver cómo a medida que nos desplazamos de un control a otro en la cabecera va apareciendo o el título o el nombre del control activo en cada momento.

ACCESO INDEXADO A LOS CONTROLES

Aunque el método habitual para acceder a un componente de la ficha y sus propiedades es utilizar el nombre del componente, **Delphi** nos permite acceder a ellos como elementos de un matriz.

Supongamos que en un ficha queremos mostrar el estado de una serie de señales distintas mediante componentes **CheckBox**, que pueden estar activos, desactivos o en estado desconocido.

Diseñemos una ficha como la que podemos ver seguidamente con 12 controles **CheckBox**, cada uno de los cuales tendrá como título el número de orden que le corresponda y como nombre ese mismo número precedido de la palabra **Señal**. A la derecha del grupo de controles insertaremos una lista **LSenales** donde mostraremos el estado de cada caja de selección y un botón **BEstado**.



Al pulsar el botón **BEstado** el programa deberá inspeccionar el estado de cada uno de los controles **CheckBox**, añadiendo a la lista su estado.

El listado siguiente muestra una primera solución que tendríamos que repetir para cada una de las cajas de selección:

```
procedure TForm1.BEstadoClick(Sender: TObject);
```

```
Var
```

```
  N: Integer;
```

```
begin
```

```
  LSenales.Clear; {Limpia la lista}
```

With LSenales.Items Do

If Senal1.State = cbChecked Then Add('Señal 1 <ACTIVA>') Else

If Senal1.State = cbUnchecked Add('Señal 1 <DESACTIVA>') Else

Add('Señal 1 <DESCONOCIDO>');

With LSenales.Items Do

If Senal2.State = cbChecked Then Add('Señal 2 <ACTIVA>') Else

If Senal2.State = cbUnchecked Add('Señal 2 <DESACTIVA>') Else

Add('Señal 2 <DESCONOCIDO>');

{Repetido para los diez controles restantes}

end;

Como vemos esta solución no es muy “elegante” y ocupa mucho código.

La solución la encontramos en las propiedades **Components**, **ComponentCount** y el método **FindComponent** que tiene cada ficha y la propiedad **ComponentIndex** que tiene cada control.

El listado siguiente muestra una solución más corta y eficiente y sin duda más clara y elegante:

```
procedure TForm1.BEstadoClick(Sender: TObject);
```

```
Var
```

```
  N: Integer;
```

```
begin
```

```
  LSenales.Clear; {Limpia la lista}
```

```
  {Recorrer los índices de los 12 controles}
```

```
  For N:=Senal1.ComponentIndex To Senal12.ComponentIndex Do
```

```
  {Convertir el componente en un CheckBox}
```

```
    With (Components[N] As TCheckBox) Do
```

Case State Of {Según la propiedad State}

cbChecked: LSenales.Items.Add('Señal '+Caption+ ' <ACTIVA>');

cbUnChecked: LSenales.Items.Add('Señal '+Caption+ ' <DESACTIVA>');

cbGrayed: LSenales.Items.Add('Señal '+Caption+ ' <DESCONOCIDO>');

End;

end;

En este caso se asume que los 12 controles **CheckBox** han sido insertados secuencialmente por orden. Sin embargo, puede no haber sido así por lo que con el siguiente código podemos solucionar el problema de que los controles estén en cualquier orden:

procedure TForm1.BEstadoClick(Sender: TObject);

Var

N: Integer;

Control: TComponent;

begin

LSenales.Clear; {Limpia la lista}

For N:=1 to 12 Do

Begin

{Buscar el componente que tiene el nombre SenalN}

Control:=FindComponent('Senal'+IntToStr(N);

{Convertir el componente en un CheckBox}

With (Control As TCheckBox) Do

Case State Of {Según la propiedad State}

cbChecked: LSenales.Items.Add('Señal '+Caption+ ' <ACTIVA>');

cbUnChecked: LSenales.Items.Add('Señal '+Caption+ ' <DESACTIVA>');

cbGrayed: LSenales.Items.Add('Señal '+Caption+ ' <DESCONOCIDO>');

End;

End;

end;

CREACIÓN DE CONTROLES EN TIEMPO DE EJECUCIÓN

Todos los objetos de **ObjectPascal**, incluidos los componentes, derivan de un tipo común que es **TObject**. Todos los controles, a su vez, están derivados de **TControl**, que es un tipo descendiente de **TComponent**. Todos los componentes comparten un constructor común, **Create**. Igual que podemos crear cualquier objeto en tiempo de ejecución podemos crear también un componente.

El constructor **Create** necesita como parámetro una referencia al componente en que se va a insertar el nuevo objeto. Dado que la llamada a este constructor se suele realizar desde algún método de la ficha en que se va a insertar el componente, el parámetro será **Self**, que representa al objeto actual. Una vez creado el componente el siguiente paso es establecer sus propiedades.

De igual forma podemos crear los métodos que actuarán como gestores de ciertos eventos. Para ello usaremos el nombre del evento como si se tratase de una propiedad y el nombre del método como si fuese el parámetro.

Partamos del programa anterior para ver un ejemplo. En lugar de insertar los 12 controles **CheckBox** en el grupo, insertaremos uno en la esquina superior izquierda.



Seguidamente escribiremos el código necesario para crear el resto de controles. Se insertará en el evento **OnCreate** de la ficha, de tal forma que al visualizarse la ficha ya aparezcan los controles en su interior.

En el siguiente listado puede verse cómo se van creando los controles, calculando su posición, título y nombre, estableciendo además un gestor común para el evento **OnClick**, para demostrar cómo un objeto creado dinámicamente puede responder a un cierto evento:

De igual forma podemos crear los métodos que actuarán como gestores de ciertos eventos. Para ello usaremos el nombre del evento como si se tratase de una propiedad y el nombre del método como si fuese el parámetro.

```
procedure TForm1.FormCreate(Sender: TObject);

Var

  N, Columna, Fila: Integer;

  Control: TCheckBox;

begin

  {Tomar las coordenadas del primer control a insertar}

  Columna := Senal1.Left;

  Fila := Senal1.Top + Senal1.Height;

  For N:=2 to 12 Do

    Begin

      {Crear el control}

      Control := TCheckBox.Create(Self);

      {Posicionarlo y darle anchura}

      Control.Left := Columna;

      Control.Top := Fila;

      Control.Width := Senal1.Width;

      {Asignar el título y el nombre}

      Control.Caption := IntToStr(N);

      Control.Name := 'Senal' + IntToStr(N);

      {El control estará contenido en el GroupBox}

      Control.Parent := Senal1.Parent;

      {Fijar el estado inicial del control}

      Control.AllowGrayed := True;

      Control.State := cbGrayed;

      {Establecer el gestor para el evento OnClick}

      Control.OnClick := Senal1Click;
```

{Calcular la posición del siguiente control}

Fila := Fila + Control.Height;

If N mod 6 = 0 Then {Si hemos terminado con la primera columna}

Begin

{Pasar al principio de la siguiente}

Columna := Columna + Control.Widht;

Fila := Senal1.Top;

End;

End;

end;

procedure TForm1.Senal1Click(Sender: TObject);

begin

Caption := 'Señal ' + (Sender As TCheckBox).Caption;

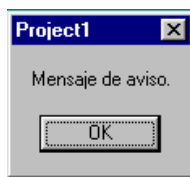
end;

2. CUÁNDO NO ES NECESARIA UNA FICHA

En ocasiones podemos pensar en diseñar una ficha para funciones tan simples como mostrar un mensaje, pedir un dato, etc... En estos casos no es necesario crear una ficha ya que existen una serie de funciones y procedimientos que facilitan todas estas operaciones.

• VISUALIZAR UN MENSAJE

Para mostrar cualquier mensaje de aviso, confirmación, etc..., que no necesite respuesta por parte del usuario utilizaremos el procedimiento **ShowMessage**, que toma como parámetro la cadena con el mensaje a visualizar. Por defecto, la ventana aparecerá en el centro de la pantalla. Si deseamos mostrar el mensaje en una cierta posición de la pantalla utilizaremos el procedimiento **ShowMessagePos**, que toma dos parámetros más, la columna y fila de la esquina superior derecha de la ventana, relativas a la pantalla.






• MENSAJES CON RESPUESTA

A veces el mensaje no es simplemente de aviso o confirmación, sino que da varias opciones. En estos casos usaremos la función **MessageDlg** que necesita cuatro parámetros:








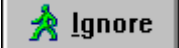
- Una cadena conteniendo el mensaje, con un máximo de 255 caracteres.
- Una constante indicando el tipo de ventana que queremos mostrar. Los valores aparecen en la **tabla 10.1**, dependiendo de la que seleccionemos aparecerá un icono distinto y el título será acorde con dicho icono.
- Un conjunto indicando los botones que deseamos aparezcan en el interior de la ventana, según los valores de la **tabla 10.2**.
- Un identificador asociado a una página de ayuda.

Una vez que el usuario pulsa cualquiera de los botones , ésta se cierra y la función devuelve un valor correspondiente a uno de los valores de la tabla 10.3. Igualmente existe la función **MessageDlgPos** que tomara además dos enteros con las coordenadas de la ventana.

Valor	Icono que aparecerá en el interior
mtCustom	Ninguno. El título será el nombre del programa
mtWarning	
mtError	
mtInformation	

mtConfirmation

tabla 10.1

Valor	Botón
<i>mbOK</i>	
<i>mbCancel</i>	
<i>mbYes</i>	
<i>mbNo</i>	
<i>mbHelp</i>	
<i>mbAbort</i>	
<i>mbRetry</i>	
<i>mbIgnore</i>	

mbAll



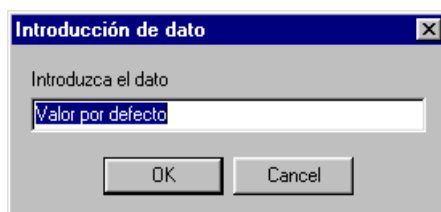
tabla 10.2

Valor	Botón que se ha pulsado
<i>mrNone</i>	Ninguno, la ventana ha sido cerrada
<i>mrOk</i>	mbOk
<i>mrYes</i>	mbYes
<i>mrNo</i>	mbNo
<i>mrCancel</i>	mbCancel
<i>mrAbort</i>	mbAbort
<i>mrRetry</i>	mbRetry
<i>mrIgnore</i>	mbIgnore
<i>mrAll</i>	mbAll

tabla 10.3

- **ENTRADA DE DATOS**

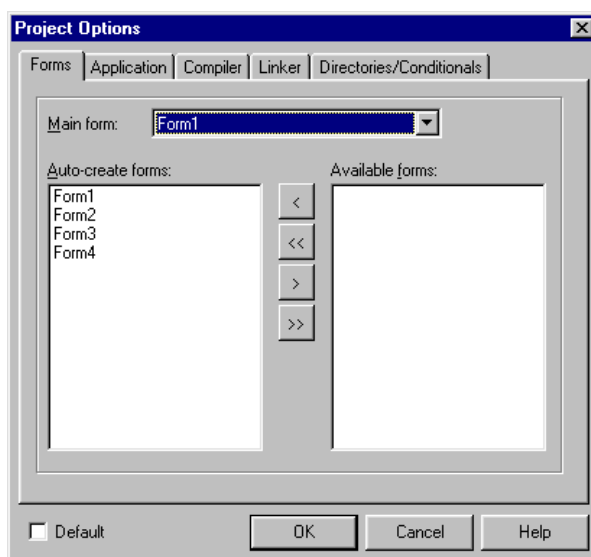
Si necesitamos solicitar información utilizaremos la función ***InputQuery***, que consta de tres parámetros, una cadena especificando el título de la cabecera de la ventana, otra con el texto con el que se solicitará el dato a introducir y una variable tipo **String** en la que será devuelto el dato introducido por el usuario. La función devuelva **True** o **False** dependiendo de que se pulse **<Intro>** o **<Escape>**.



GESTIÓN DE MÚLTIPLES FICHAS

En todo proyecto que cuenta con más de una ficha siempre existirá una que actúe como ficha principal, que será creada y mostrada automáticamente al ejecutar la aplicación. El resto de las fichas deberán ser mostradas a petición del código de nuestro programa, para lo que previamente habrán de ser creadas.

Las fichas del proyecto pueden ser creadas al inicio del programa, para ello las insertaremos en la lista **Auto-create forms**, de la página **Forms** de las opciones del proyecto. Las fichas que aparezcan en el apartado **Available forms** no serán creadas al inicio, por lo que antes de usarlas tendrán que ser creadas.



- **CREAR UNA FICHA**

Antes de poder usar una ficha tendremos que añadir a la cláusula **Uses** del módulo en que se va a incluir la referencia, el nombre del módulo asociado a la ficha.

Para crear una ficha tendremos que utilizar el constructor **Create**, tomando como parámetro una referencia al componente que se convertirá en padre de la nueva ficha.

Otro método es usar el objeto **Application**, del tipo **TApplication** y que se crea al comenzar el programa. Este objeto cuenta con el método **CreateForm**, al que pasándole como parámetros el tipo de ficha a crear y una variable donde guardar la referencia creará la ficha.

Cuando la ficha creada no la utilicemos más, podemos liberar memoria mediante el método **Free**.

- **VENTAJAS E INCONVENIENTES**

La ventaja de crear todas las fichas al cargar la aplicación es que siempre disponemos de ella sin preocuparnos de crearla o liberar memoria. Sin embargo, existen varios inconvenientes. La creación de todas las fichas al principio supone un gasto de memoria excesivo, así como el tiempo que tarda la aplicación en cargarse y mostrarse activa

MOSTRAR UNA FICHA

Independientemente del método de creación de la ficha, tendremos que hacerla visible. Un método consiste en dar el valor **True** a la propiedad **Visible**. El efecto es similar si llamamos al método **Show**, llamado si es necesario al método **BringToFront**. Cualquiera que sea el método estaremos visualizando una ventana no modal, por lo que podremos activar otras ventanas.

Para ocultar la ventana podemos llamar al método **Hide** o dar **False** a la propiedad **Visible**.

- **CUADROS DE DIÁLOGO MODALES**

Una ventana modal no tiene los botones para minimizar y maximizar, tampoco existe la posibilidad de modificar su tamaño en tiempo de ejecución y normalmente incorpora botones generales como **OK** o **Cancel**.

Para mostrar una ventana como modal la visualizaremos mediante el método **ShowModal** y daremos el valor **bsDialog** a la propiedad **BorderStyle**. Por último, para ocultarse, daremos un valor distinto de cero a la propiedad **ModalResult**, de esta forma podremos usar el valor devuelto de forma adecuada.

CREACIÓN DINÁMICA DE FICHAS

Aunque no es una técnica muy habitual, una ficha puede ser creada dinámicamente. Para ello declararemos una variable del tipo *TForm*, utilizando posteriormente el constructor **Create** o el método **CreateForm** para crearla. Luego determinaremos sus propiedades, como título, dimensiones, etc. El siguiente paso será la inclusión de los controles que necesitemos, que también habrán de crearse dinámicamente.

3. MENÚS.

Los Menús proporcionan al usuario una forma sencilla de ejecutar comandos agrupados lógicamente. El IDE de **Delphi** incluye un programa “experto” **Diseñador de Menús** que permite añadir fácilmente a la ficha un Menú personalizado:

- ♦ Barra de menú y menús desplegados **MainMenu**
- ♦ Menús emergentes **PopupMenu**

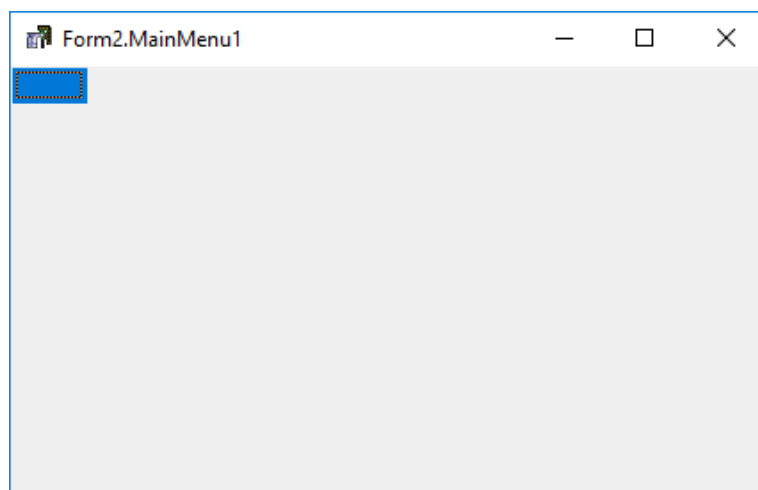
Desde modo diseño podemos: añadir, eliminar elementos, arrastrarlos y soltarlos para reorganizarlos, etc...

Terminología de Menús:

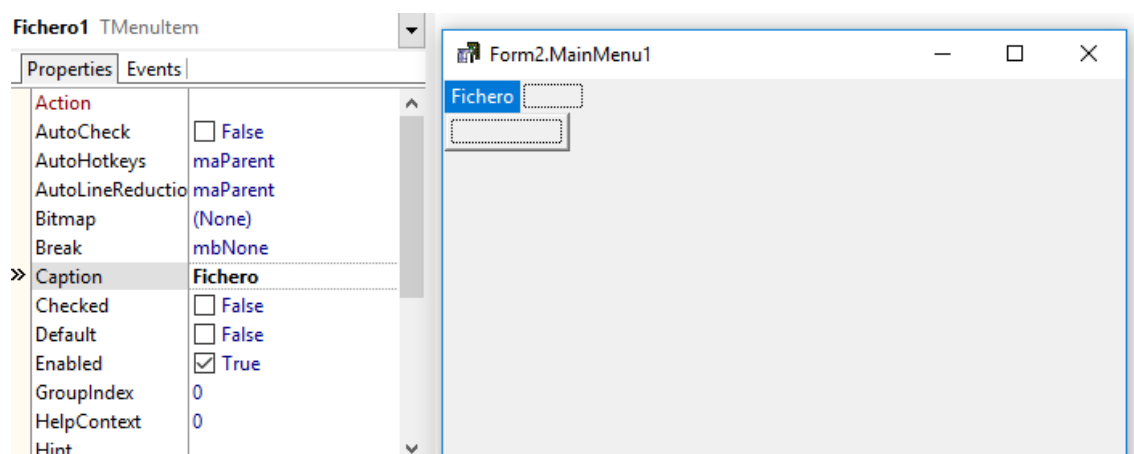
Para poner en marcha el **Diseñador de Menús**:

- ① Seleccionar componente **MainMenu** ó **PopupMenu** de la paleta **Standard**.
- ② Situarlo sobre la Ficha.
- ③ ♦ Pulsar doble ‘click’ sobre el componente
ó
♦ Activar la propiedad **Items** del componente.

A continuación aparecerá en pantalla el entorno del programa Diseñador de menús.



Como podemos observar en “el volcado” de la imagen, sobre la **Ficha** que estamos diseñando se ha superpuesto el entorno visual que utilizaremos para la creación de los elementos del objeto **Menú**.



Para crear las clásicas barras separadoras que hay en los menús, solo tienes que poner en la propiedad Caption un guión. Así consigues el efecto que te muestro en la imagen a la derecha de este texto cuando tu programa este ejecutándose. Mira que a la derecha de los menús hay teclas de acceso rápido. Las teclas de acceso rápido son combinaciones de teclas que realizan el mismo efecto que si pulsarás ese menú. Por ejemplo en Word cuando pulsas Ctrl+P accedes al cuadro de impresión, eso es porque el menú de impresión tiene asociado esas teclas. Si quieres que tus menús tengan teclas de acceso rápido (también llamados atajos de teclado), tienes que poner la combinación en la propiedad ShortCut del menú o submenú que desees. Si despliegas esta propiedad verás gran cantidad de combinaciones predefinidas. Observa que apenas hay alguna combinación definida con la tecla Alt. Esto es porque las combinaciones de Alt más una tecla esta asociadas al carácter subrayado del texto que muestran los controles a través de su propiedad Caption. Este carácter subrayado se obtiene poniendo antes del carácter deseado el símbolo &. Creo que ya lo he usado en algún ejemplo aunque no lo había comentado. Prueba a poner un menú o submenú con este símbolo incluido en su nombre,

por ejemplo: &Cerrar. Tampoco existe la combinación Alt+F4, la cual es conocida por todo usuario de Windows con algo de experiencia, y que sirve para cerrar una aplicación.

Una pequeña trampa para relacionar nuestro menú de salida con este atajo de Windows, es ponerlo su propiedad ShortCut, pero como no aparece en la lista pues simplemente teclea la combinación. He dicho teclear, no la ejecutes, sino escribe Alt+F4. Ten en cuenta que si usas este atajo tu programa es el responsable de cerrar la aplicación.

Otra propiedad interesante es Checked, la cual funciona de la misma manera que en el CheckBox que usamos en la alarma. Existen otras propiedades como RadioButton y GroupIndex, las cuales te explicaré cuando toque el tema de los grupos de botones, ya que funcionan igual y lo entenderás mejor.

Te habrás fijado que hay programas que dentro de los menús tienen otros menús que salen hacia un lado, esta indicado con una flecha. Pues para crear un menú de este tipo, solo tienes que ponerte encima del submenú y pulsas en tu teclado la tecla Control, más la tecla cursor hacia la derecha, y ya esta.

Te comento que para borrar un menú o submenú, selecciónalo y pulsa Control más la tecla suprimir. Para insertar pues Control más insertar. También los puedes cambiar de posición o de menú arrastrándolo.

Hasta ahora mucha teoría, pero quizás te estés preguntado que como se controlan los menús y submenús, pues muy fácil, si te digo que responden a un solo evento, funcionan de manera análoga a la de los botones, pues ya esta todo dicho. Pues en ese único evento que tienen los menús es donde debes escribir tu código.

MENÚ DEL TIPO POPUP.

Todo lo que te he contado hasta ahora es aplicable a un menú tipo emergente (popup), son iguales, pero lo único que los diferencia es que los menús emergentes se despliegan al pulsar sobre un control con el botón derecho del ratón. Pues para usar un menú de este tipo lo único de que debes hacer es colocar un control de este tipo sobre el formulario que contiene el control que hará uso del menú emergente (puede ser casi cualquier control, o el mismo formulario), configúralo como si se tratase de un menú normal, y ahora en el control que desea que tenga este menú emergente, selecciónalo en su propiedad PopUpMenu, y ya esta. Prueba tu programa y pulsa sobre el componente o formulario con el botón derecho del ratón.

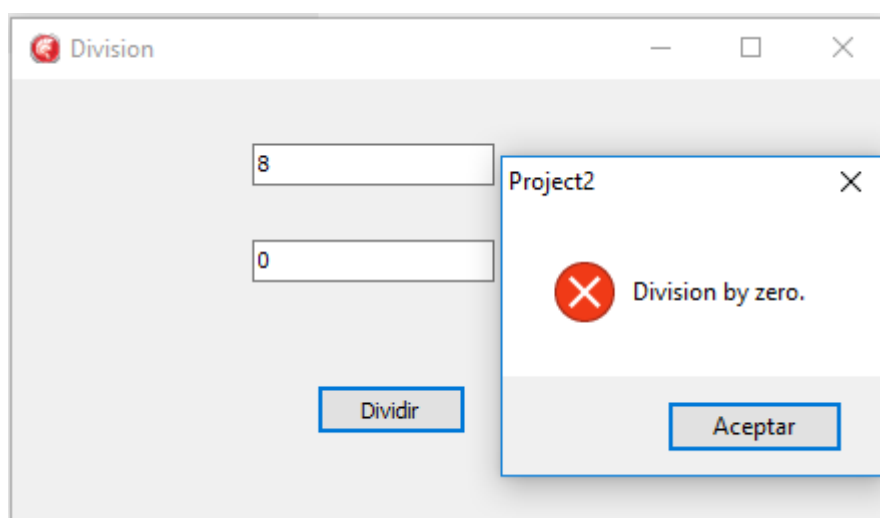
4. ERRORES EN EJECUCIÓN

Cada método, procedimiento o función puede encontrarse en cierto momento en una situación errónea al ejecutar una orden o un grupo de ordenes; ejemplo:” realizar una operación sobre una tabla no creada, pasarse de rango en una operación de cálculo, etc. “.

Dichos errores crearían una interrupción en la aplicación rompiendo su ejecución y funcionamiento, pudiendo incluso crear una parada del sistema. La mejor solución sería subsanar el error sin tener que acabar la aplicación o acabar la aplicación de forma correcta “cerrando todas las puertas que hemos dejado abiertas”.

SOLUCIÓN: LAS EXEPCIONES:

El sistema puede detectar el error (avisando posiblemente con un mensaje) pero no tiene ni idea de que decisión tomar el respecto.



En **Delphi** cuando en un fragmento de código se detecta una situación excepcional, caso de una operación incorrecta con una orden, y en la aplicación hemos utilizado el uso de la unidad **SysUtils**, los errores son convertidos automáticamente en las llamadas excepciones, que nos permitirán tomar las medidas necesarias para resolver la situación; gracias a que el **Delphi** dispone de un manejo de excepciones en la biblioteca **VCL**, mediante la cual el propio motor del **Delphi**, por defecto, controla el error realizando una parada de la aplicación avisándonos mediante un mensaje.

¿QUÉ ES UNA EXCEPCIÓN?

Una excepción es una señal provocada por un error. Esta señal puede ser interceptada por el propio programa, procediendo a su tratamiento o bien por el código adicional añadido por el compilador, que se encargará de avisar mediante un mensaje de que se ha producido una excepción, interrumpiendo la ejecución del programa.

Generalmente una excepción dispone de la información necesaria como para conocer la causa y el punto en el que se ha producido un error, ayudando así en su resolución. Dicha información la podemos aprovechar en nuestro código, si interceptamos la excepción. O bien será mostrada en un mensaje antes de que el programa se interrumpa, si no la interceptamos.

EL OBJETO EXCEPTION:

Todas las clases de excepciones heredan de la clase **Exception**, definida en la unidad **SysUtils** (que deberemos llamar en la cláusula **Uses** de nuestra unidad). La clase **Exception** se declara de la misma forma que cualquier clase; de esta forma cada excepción estará representada por un objeto o mejor dicho será un tipo objeto, teniendo así las siguientes ventajas:

- Tendremos información del lugar donde se produce y donde será tratada, puesto que al ser un objeto tiene esas cualidades.
- Se podrán agrupar (ya que al ser objetos disponen de la cualidad de la herencia).
- Por último, añadir nuevas excepciones sin afectar el código existente.

Unit datos;

Interface

Uses SysUtils;

Type

//Declaración de tipos de excepciones

Emaldata = Class (Exception);

EmalRango = Class (Exception);

.....

Procedure Errores;

Begin

.....

//Mal palabra: avisamos con una excepción

Raise Emaldata.CreateFmt (' palabra mal: %S',[Dato]);

.....

Function Errores2;

Begin

.....

If Numero >123 then R:=8

Else // *Mal número: avisamos con una excepcion*

Raise EmalRango.CreateFmt(' número mal: %d', [Numero]);

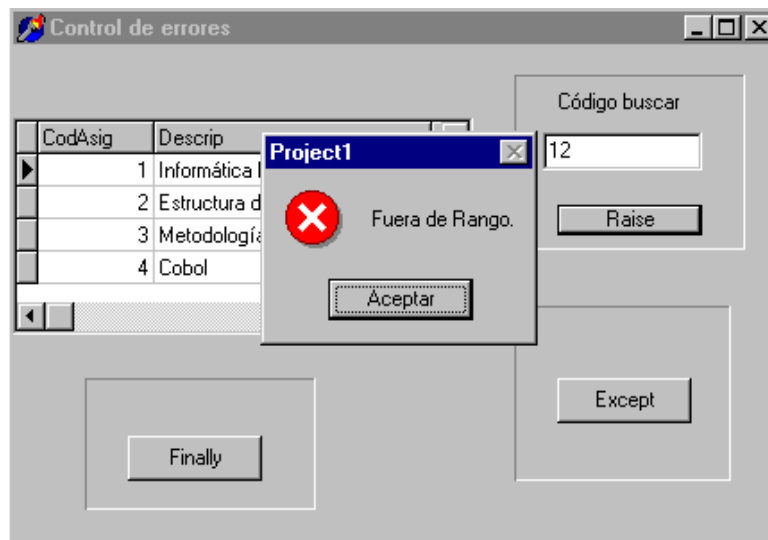
.....

MANEJO Y USO DE EXCEPCIONES:

Lo esencial es detectar en que bloques de programa (orden o conjunto de instrucciones), se puede producir error. Para gestionar el error podemos utilizar tres métodos:

1. Generar nosotros mismos la excepción , por medio de la sentencia **Raise** (levantar), que provoca una interrupción de la ejecución del programa, terminando todas las funciones pendientes agrupadas en la pila de ejecución generada por las diversas llamadas a procedimientos y funciones, es decir, termina el procedimiento que la ha provocado.

Cada vez que se piense que pueden ir mal las cosas se utilizará, facilitando como único parámetro un objeto de la clase de excepción que queremos generar, cuya instancia puede crearse en esa misma sentencia, llamando al constructor **Create** con una cadena de caracteres, que definirá el mensaje contenido en la excepción.



Raise Objeto-error.Create ('Mensaje de aviso');

Program Genexcep;

Uses

Dialogs, SysUtils;

Var

Numero: String;

Begin

InputQuery('Valor filtrado','Introduzca un número entre 1 y 10: ',Numero);

If (StrToInt(Numero) < 1) Or (StrToInt(Numero) > 10) Then

Raise ERangeError.Create('Fuera de rango');

ShowMessage('El número era válido');

End.

En caso de no conocer el **objeto-clase-error** se puede aplicar **Exception** que es el objeto general que representa a cualquier error.

Raise Exception.Create ('Mensaje de aviso');

Aunque **Delphi** tiene definida una jerarquía de clases de excepciones bastante completa, no siempre todas nuestras necesidades se verán cubiertas. Dado que **Exception** es un tipo de objeto, podemos usarlo como base para derivar nuevos tipos de excepciones. Para ello sólo hará falta crear un nuevo tipo de objeto, especificando como base **Exception**, tras lo cual podremos crear instancias del nuevo tipo y generar excepciones de la forma vista anteriormente.

Program Exceprop;

Uses

Dialogs, SysUtils;

Type

EMesValido=Class(Exception);

Var

Mes: String;

Begin

InputQuery('Valor filtrado','Introduzca un número entre 1 y 12: ',Mes);

If (StrToInt(Mes)< 1) Or (StrToInt(Mes)> 12) Then

Raise EMesValido.Create('Mes Fuera de rango');

ShowMessage('El mes era válido');

End.

Mediante **Raise**, las instrucciones siguientes no se ejecutan , de tal forma que incluso se puede aplicar sin un parámetro ya que en este caso el propio **Delphi** mostrará el mensaje de error y cerrará los procedimientos abiertos afectados por el error, liberando memoria y evitando el llenado de la pila que controla las interrupciones no provocando un posible error de memoria insuficiente.

2. La instrucción **Try.....Finally.....End**, se utilizará no para gestionar la excepción, pero si para liberar al ordenador de recursos después de ejecutar un bloque de ordenes.

El bloque de ordenes que intentamos ejecutar con posibilidad de producir una excepción se colocan en la parte de la instrucción **Try** (intentar), si se produce error se interrumpe la ejecución, visualizando el sistema el mensaje de aviso de la anomalía producida, no sin antes ejecutar las ordenes que se encuentran en el apartado de **Finally**. En caso de no existir excepción también se ejecutarán las ordenes expuestas en **Finally** pero sin interrumpir el programa.

Procedure Control;

Var

BookMark:TbookMark;

Begin

Table1.DisableControls; {Instrucciones que piden un recurso}

BookMark:=Table1.GetBookMark;

Try **{Inst. que trabajan con el recurso y pueden fallar}**

Table1.First;

Finally **{Inst. que devuelven el recurso liberando al sistema}**

Table1.GotoBookMark(BookMark);

Table1.FreeBookMark(BookMark);

Table1.EnableControls;

End;

Try

Try

New(PNumero);{Se asigna memoria para un entero}

PNumero^:=Dividendo Div Divisor;

Finally

Dispose(PNumero); {Liberar la memoria asignada}

End;

Except

ShowMessage('Ha ocurrido una excepción');

End;

3. **Try.....Except.....End**; el primer paso para tratar excepciones es determinar qué sentencias o bloques de sentencias de nuestro código pueden generar algún error. Para evitar que la excepción generada por el programa sea gestionada por **Windows** o **Delphi** debemos aislar el bloque a proteger entre la palabra **Try** y la palabra **Except**, situando detrás de ésta las sentencias a ejecutar en caso de que se produzca una excepción. La estructura completa de control de la excepción finalizará con la palabra **End**.

Try

// Instrucciones que pueden errar

Except

// Bloque de instrucciones que se ejecutarán en caso de

// error, el programa continuará a la siguiente orden

End;

Procedure Fichero;

Begin

AssignFile(Fichero,'Datos');

Try

//Intentar abrir el fichero

Reset(Fichero);

Except

//Se ha producido error no existe el fichero

//Se crea

Rewrite(Fichero);

End;

Al producirse una excepción, automáticamente se crea un objeto conteniendo la información acerca de ella. Este objeto pertenecerá a una determinada clase

y nos puede servir para determinar qué error se ha producido. Las distintas clases de excepciones se definen en el módulo **SysUtils** que tendremos que enumerar en la cláusula **Uses** de nuestro programa.

Con el fin de diferenciar en nuestro código las sentencias que se han de ejecutar dependiendo del tipo de excepción (debido a que el bloque de ordenes pueden generar varios errores), tras la palabra **Except** introduciremos uno o más apartados, cada uno de los cuales se iniciará con la palabra **On** seguida de la clase de excepción a la que se desea responder y la palabra **Do**, tras la cual dispondremos la sentencia o bloque de sentencias a ejecutar.

Try

// Instrucciones a ejecutar con posibilidad de errores

Except

On Clase-Objeto-Excepción1 Do

// Acción al error

.....

On Clase-Objeto-ExcepciónN Do

// Acción al error

End;

Program Divide2;

Uses

Dialogs, SysUtils;

Var

Dividendo, Divisor: String;

Cociente: Real;

Error: Boolean;

Begin

While True Do

Begin


```
Error:=False;
```

```
Try
```

```
    InputQuery('Valor del dividendo','Entre el dividendo: ',Dividendo);
```

```
    If Dividendo = '0' Then Break;
```

```
    InputQuery('Valor del divisor','Entre el divisor: ',Divisor);
```

```
    Cociente:=StrToFloat(Dividendo) / StrToFloat(Divisor);
```

```
Except
```

```
    On EZeroDivide Do      {División por cero}
```

```
        Begin
```

```
            ShowMessage('No se puede dividir por cero');
```

```
            Error:=True:
```

```
        End;
```

```
    On EOverflow Do      {Desbordamiento}
```

```
        Begin
```

```
            ShowMessage('Uno de los operandos es demasiado grande');
```

```
            Error:=True:
```

```
        End;
```

```
    End;
```

```
If Not Error Then
```

```
    ShowMessage('Cociente= ',FloatToStr(Cociente));
```

```
End;
```

```
End.
```

Cuando deseemos controlar otras excepciones que puedan producirse en nuestro programa sin conocer de antemano cuáles pueden ser, al final de la estructura **Except...On** podemos añadir la palabra **Else**, seguida de la sentencia que se ejecutará en caso de que la excepción no pertenezca a ninguna de las clases especificadas con anterioridad.

```
Try
```

```
    // Instrucciones a ejecutar con posibilidad de errores
```

Except

On Clase-Objeto-Excepción1 Do

// Acción al error

.....

On Clase-Objeto-ExcepciónN Do

// Acción al error

Else

//instrucciones para cualquier otro error

End;

Except

On EZeroDivide Do {División por cero}

Begin

ShowMessage('No se puede dividir por cero');

Error:=True;

End;

On EOverflow Do {Desbordamiento}

Begin

ShowMessage('Uno de los operandos es grande');

Error:=True;

End;

Else

ShowMessage('Se ha producido una excepción desconocida');

End;

Todas las clases de excepciones están derivadas del objeto **Exception**. Este tipo contiene los miembros, datos y métodos necesarios para la gestión de una excepción. Uno de los miembros de este objeto es **Message**, mediante el cual

podemos obtener un mensaje identificativo de la excepción que se ha generado, permitiendo facilitar información adicional sobre el error que ha provocado la excepción.

En la estructura **Except...On** del ejemplo anterior no tenemos información sobre el error que se produzca distinto a la división por cero o desbordamiento. Para poder tener dicha información, tras la palabra **On** dispondremos de un identificador que irá seguido de dos puntos y la clase de excepción tal y como si estuviéramos declarando una variable de ese tipo. A partir de ahí podemos usar ese objeto para

acceder a sus miembros, por ejemplo a **Message**. Este tipo de construcción no está permitida detrás de la parte **Else** del bloque

Except, pero puesto que todas las excepciones están derivadas de **Exception**, podemos modificar el código.

Try

//Instrucciones con posibles errores

Except

//Objeto excepción que captura el error

On E: Clase-Objeto-Excepción Do

//Instrucciones que gestionan el error

.....

// Objeto Exception que suple a cualquier tipo error

On E: Exception Do

//Instrucciones a ejecutar en caso de producirse

// un error no gestionado

End;

Except

On EZeroDivide Do {División por cero}

Begin

```
    ShowMessage('No se puede dividir por cero');

    Error:=True;

End;

On EOverflow Do    {Desbordamiento}

    Begin

    ShowMessage('Uno de los operandos es demasiado grande');

        Error:=True;

        End;

    On X: Exception Do

        Begin

    ShowMessage('Se ha producido la excepción X.Message');

            Error:=True;

            End;

        End;
```

PARTICULARIDADES: GENERALES

La función **ExceptObject** también recupera el objeto que se eleva como error, retornando un objeto de tipo TObject.

Try

//Instrucciones

Finally

//Instrucciones que se ejecutarán tanto si se

//produce excepciones como si no las hay

If ExceptObject <> Nil Then Limpiar; // Llamar a

// una función que soluciona el error

End;

También podemos realizar la gestión de la siguiente forma:

Try

//Instrucciones

Except

Limpiar; //Función que libera recursos

Raise;

End;

Si se esta trabajando con un **Delphi** en inglés y se quiere mostrar errores en castellano se puede utilizar la herramienta de Lenguaje **Pack** (que cuesta su dinero), aunque, como hemos aprendido, podemos sustituir el mensaje en inglés por castellano utilizando:

Try

//Instrucciones que pueden fallar

Except

Raise Exception.Create ('Este es el nuevo mensaje');

End;

Cuando no deseamos ver mensaje alguno podemos ejecutar el procedimiento **Abort**, generando la llamada excepción silenciosa.

Try

//Instrucciones

Except

On E: Clase-Error Do

Abort;

End;

PARTICULARIDADES: REFERENTES A DASES DE DATOS:

Cuando un conjunto de datos se encuentra en el estado **dsBrowse**, no es posible asignar valores a los campos, pues se produce una excepción. Para realizar modificaciones, hay que cambiar el estado del conjunto de datos a uno de los estados **dsEdit** ó **dsInsert**.

Es una precondition de **Post** que el conjunto de datos se encuentre alguno de los estados de edición; de no ser así, se produce una excepción. Es importante, cuando trabajamos con bases de datos locales, garantizar que una tabla abandone el estado **Edit**, ya que para las tablas locales **Edit** pide un bloqueo, que no es devuelto hasta que se llame a **Cancel** o **Post**.

```
Table1.Edit;  
  
Try  
  
    // Asignación a campos  
  
Table1.Post;  
  
Except  
  
    Table1.Cancel;  
  
    Raise;  
  
End;
```

Cuando se necesita imponer una condición sobre varios campos de un mismo registro, se puede realizar en el evento **BeforePost**., verificando en su interior la condición, y si no se cumple se aborta la operación con una excepción; también se puede realizar con el evento **OnValidate**.

```
Procedure TmoduloDatos.Table1BeforePost(DataSet: TDataSet);  
  
Begin  
  
    If Tabla1.State = dsInsert Then  
  
        If Tabla1.Edad > TopeEdad Then  
  
            DatabaseError('Mal la edad');  
  
    // Para lanzar una excepción se puede utilizar la función //DatabaseError que es  
    // equivalente a la instrucción:  
  
    // Raise EdatabaseError.Create ('mensaje de explicación')  
  
End;
```

```
Procedure TdataModule1.Table1Validate(Sender: Tfield);
```

```
Begin
```

```
  If not Table1.Locate('Numero', Sender.Value, []) Then
```

```
    DatabaseError('Número de alumno incorrecto');
```

```
End;
```

Para solucionar problemas a la hora de trabajar con bases de datos relacionales y en un entorno de red, será necesario tener conocimientos sobre los temas de **Transacciones** y control de concurrencia, y actualizaciones en **Caché**.

Encerrar en instrucciones **Try.....Except** todas las llamadas que pudieran fallar, no es siempre posible pues algunos métodos, como **Post**, son llamados implícitamente por otras operaciones. Será necesario utilizar los eventos de detección de errores , que se disparan cuando se produce un error en alguna operación de un conjunto de datos, pero antes de que se eleve la excepción asociada **OnEditError**, **OnPostError** y **OnDeleteError**.

Estos eventos cuando se generan lo hacen después de que ha ocurrido el evento **Before**, pero antes de que ocurra el evento **After**. Dentro del manejador del evento se puede corregir la situación de error, e indicar que la operación se reintente; para esto hay que asignar **daRetry** al parámetro **DataAction**. Se puede dejar que la operación **fallay** que el **Delphi** muestre el mensaje de error, asignando **daFail** a **DataAction** (valor inicial). Para mostrar el mensaje dentro del manejador y que **Delphi** aborte la operación sin mostrar mensajes adicionales, se tendrá que asignar **daAbort** a **DataAction**.

// Cuando se produce un error de bloqueo se dispara el

// Evento OnEditError

```
Procedure TmoduloDatos.Table1EditError( DataSet: TdataSet;
```

```
  E: EdatabaseError; var Action: TdataAction );
```

```
Var
```

```
  C: Integer;
```

```
Begin
```

```
  //GetTickCount devuelve los milisegundos transcurridos
```

```
  // desde el arranque del Delphi, hacemos una espera
```

```
  C:=GetTickCount+700+Random(700);
```

```
While GetTickCount < C do
```

```
    Application.ProcessMessages;
```

```
    // Reintentar la operación
```

```
    Action:=daRetry;
```

```
End;
```

En el caso de errores producidos por **Post** y **Delete**, las causas de error pueden ser varias, con lo cual se tendrá que utilizar la información que pasa el parámetro **E**, que es la excepción que está a punto de generar el sistema e interpretar el tipo **EdatabaseError**. La mayoría de los errores de bases de datos disparan una excepción de tipo **EDBEngineError**. Un objeto de esta clase de excepción tiene, además de las propiedades heredadas de **EdatabaseError**, las propiedades **ErrorCount: Integer** y **Errors[Index]: TDBError**.

El tipo **TDBError** corresponde a cada error individual. Las propiedades disponibles en esta clase son:

ErrorCode :El código de error, de acuerdo al BDE

Category : Categoría a la que pertenece el error

SubCode : Subcódigo del error

NativeError: Si el error es producido por el servidor, el código de error devuelto

Message : Cadena con el mensaje asociado al error

// Creación de un procedimiento general compartido

// por varias tablas en que realizan su llamada en sus

// eventos OnPostError para detectar claves repetidas

// será necesario añadir Bde a la cláusula Uses

```
Procedure TmoduloDatos.ErrorGeneral(DataSet: TdataSet;
```

```
    E: EdatabaseError; var Action: TdataAction);
```

```
Begin
```

```
    //El parámetro E es de tipo EDBEngeniError
```

```
    If E is EDBEngineError Then
```

```
        Begin
```


// El parámetro **E** sirve para determinar la causa real //del error, puede extraer códigos de error específicos

Case EDBEngineError(E).Errors[0].ErrorCode Of

DBIERR_KEYVIOL:

ShowMessage('Clave repetida en la tabla '+ Ttable(DataSet).TableName);

DBERR_FOREIGNKEYERR:

ShowMwssage('Error en clave externa tabla '+ Ttable(DataSet).TableName);

Else

Exit;

End;

Action:=daAbort;

End;

End;

Con el parámetro de error **E**, además de lo mostrado en el ejemplo se pueden extraer un mensaje de error por ejemplo para mostrar en un label:

ErrorLabel.Caption:=E.Message;

También podemos como el ejemplo comprobar un error con que causa esta relacionado:

If EDBEngineError(E).Errors[ErrorCount – 1].ErrorCode = DBIERR_KEYVIOL

Then UpdateAction := uaSkip { violación de clave no tener en cuenta ese registro} Else UpdateAction:=uaAbort {al no saber que ocurre, se interrumpe la actualización}

5. CUADROS DE DIÁLOGO DE USO COMÚN

Delphi proporciona en la paleta **Dialogs** una serie de cuadros de diálogo que son de uso común en la mayoría de las aplicaciones. Estos componentes tienen una serie de características comunes, como ser la de no disponer de un tamaño ni un elemento de interfaz en tiempo de diseño, o de no ser visibles en tiempo de ejecución.

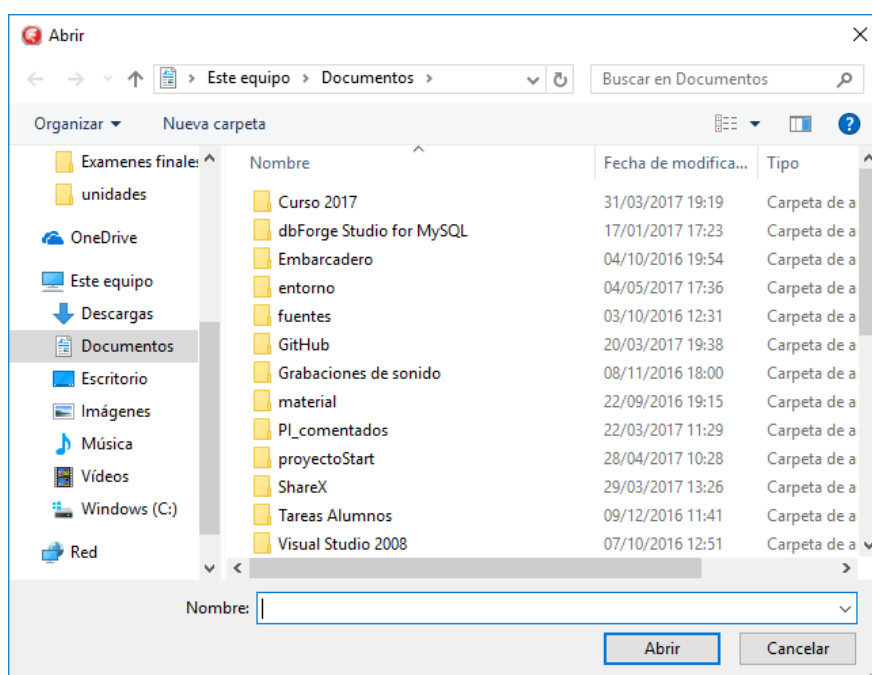
Para hacer visibles estos componentes tendremos que utilizar el método **Execute**, que no toma parámetro alguno y puede devolver **True** o **False**,

indicando que el cuadro de diálogo se ha cerrado por la pulsación del botón **OK** o bien por otro método distinto.

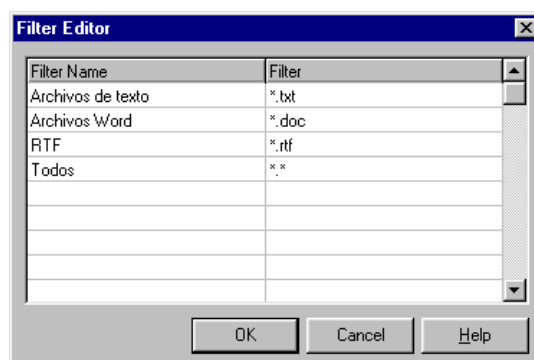
CARGAR ARCHIVOS (OpenDialog) TOpenDialog

Permite seleccionar un archivo para abrir. Básicamente el usuario puede cambiar la unidad y directorios actuales hasta encontrar el archivo que se desea abrir.

Por defecto el cuadro de diálogo tendrá un título genérico que proporcionamos mediante la propiedad **Title**. El nombre de archivo que aparezca inicialmente seleccionado, en el apartado **Nombre de archivo**, dependerá de la propiedad **FileName**. Una vez que se ha seleccionado el archivo y se cierre el cuadro de diálogo podremos obtener el nombre y todo el camino completo, incluyendo la letra de la unidad, mediante la misma propiedad **FileName**.



El método más habitual de selección de un archivo es la selección directa en la lista que ocupa la mayor parte de la ventana. La lista de tipos de archivo la podemos facilitar mediante la propiedad **Filter**, a la que asignaremos una serie de literales y a éstos una máscara de archivos que cumplan un determinado filtro. A un literal se le puede asociar más de una máscara separadas por punto y coma.



Cuando se facilitan varios filtros para un cuadro de diálogo, podemos hacer aparecer el que corresponda con la propiedad **FilterIndex**, que por defecto vale 1.

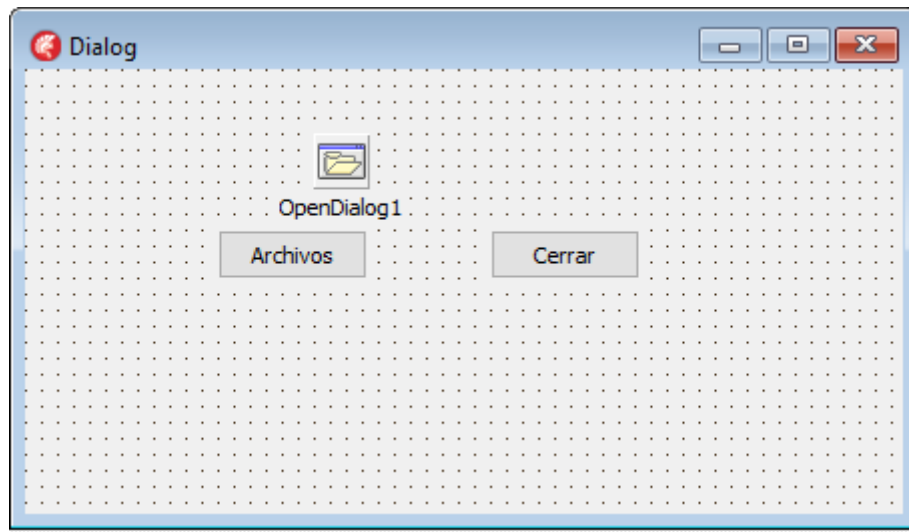
La propiedad **DefaultExt** permite añadir una extensión por defecto cuando el usuario no la haya especificado. Mediante **InitialDir** podemos indicar cuál es el directorio que aparecerá inicialmente abierto.

La propiedad **FileEditStyle** determina si el usuario se encontrará con un **Edit** o un **Combo** a la hora de especificar el archivo a recuperar, si el valor es **fsComboBox** los nombres de archivo los asignaremos a la propiedad **HistoryList**, que es un objeto del tipo **TString**.

El aspecto y comportamiento del cuadro de diálogo **OpenDialog** puede ser modificado en la propiedad Options, que es un conjunto del tipo **TOpenOptions**, que por defecto es un conjunto vacío. Los valores posibles son los siguientes, entre otros:

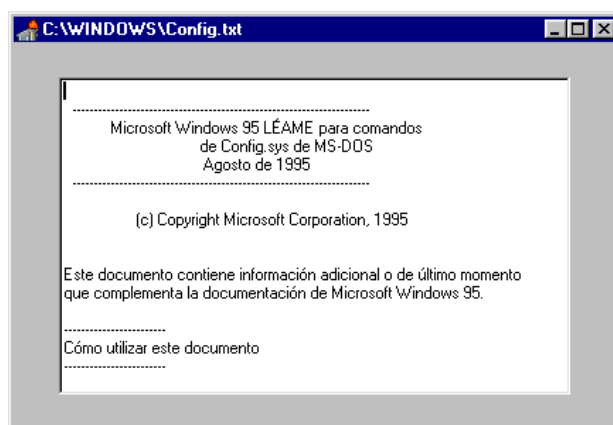
Valor	Significado
ofAllowMultiSelect	Permite seleccionar varios archivos.
ofCreatePrompt	En caso de que el archivo no exista, se mostrará una ventana preguntando si se desea crear.
ofFileMustExist	Se fuerza a que el archivo a elegir exista.
ofHideReadOnly	Oculto los archivos de sólo lectura.
ofPathMustExist	Similar al anterior pero referente a caminos de acceso.
ofNoChangeDir	Impide cambiar de directorio.
ofNoValidate	Permite introducir cualquier carácter.
ofOverwritePrompt	Indica mediante una caja que el archivo será sobrescrito
ofShowHelp	Aparece un botón de ayuda.

Veamos como se crean ventanas de dialogo con una serie de ejemplos. Partiremos de una ventana principal en la que ubicaremos tres elementos, dos botones y un componente **OpenDialog**.



El botón **Archivo** permitirá elegir un determinado archivo y visualizar su contenido en un control **Memo** de otra ventana. El método de respuesta al evento **OnClick** del botón **Archivo** tendrá la forma siguiente:

```
procedure TForm1.ArchivoClick(Sender: TObject);  
  
begin  
    if OpenDialog1.Execute then  
        begin  
            OpenDialog1.InitialDir:="";  
            Form2:=TForm2.Create(Form1);  
            Form2.Caption:=OpenDialog1.Filename;  
            Form2.Memo1.Lines.LoadFromFile(OpenDialog1.Filename);  
            Form2.ShowModal;  
        end;  
end;
```

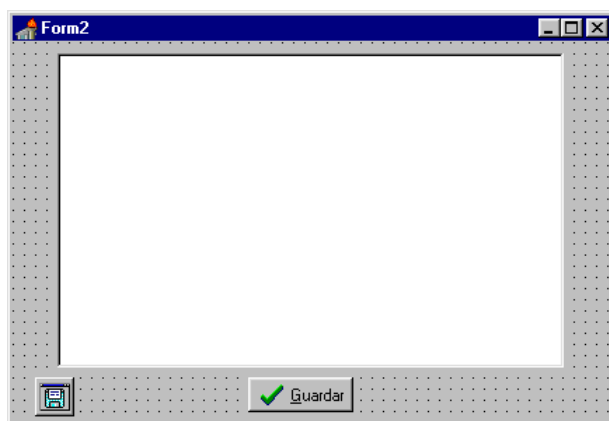


SALVAR ARCHIVOS (SaveDialog) TSaveDialog

Todo lo dicho para el componente anterior es válido para este, por lo que todas las propiedades estudiadas son válidas y aplicables en este componente.

Eso sí, no todas las opciones de la propiedad Options tienen validez, ya que **ofFileMustExist** no tiene sentido para salvar un archivo. En este caso tiene validez la opción **ofOverwritePrompt** que comprueba si se desea o no sobrescribir el contenido de un fichero existente.

Podemos mejorar el ejemplo anterior añadiendo un botón al segundo formulario que llamaremos **Guardar** que permita guardar el contenido del control **Memo** una vez finalizado el trabajo con él, por supuesto, también añadiremos un control **SaveDialog**:



El botón **Guardar** permitirá guardar el archivo, pudiéndose cambiar el nombre, si es necesario. El método de respuesta al evento **OnClick** del botón **Guardar** tendrá la forma siguiente:

```
procedure TForm2.GuardarClick(Sender: TObject);
```

```
begin
```

```
    SaveDialog1.FileName:=Caption;
```

```

if SaveDialog1.Execute then
begin
    SaveDialog1.InitialDir:="";

    Memo1.Lines.SaveToFile(SaveDialog1.Filename);

end;

```

TIPOS DE LETRA (FontDialog)

Al mostrar este cuadro de diálogo, el tipo de letra, estilo. Tamaño y color seleccionados por defecto dependerán de los valores actuales de la propiedad **Font**.

Una vez que el cuadro de diálogo es visualizado, el usuario puede alterar cualquiera de los componentes, y al cerrar la ventana podremos obtener todos los datos del tipo seleccionado.

Utilizando la propiedad **Device** podemos indicar que dispositivo será destino del tipo de letra, limitando las fuentes que se listarán a sólo aquellas que existan para ese dispositivo. Los valores pueden ser:

Valor	Dispositivo de destino
fdScreen	Pantalla
fdPrinter	Impresora
fdBoth	Ambos

Los valores y finalidad de las opciones de la propiedad **Options** son los siguientes, entre otros:

Valor	Significado
fdAnsiOnly	No se mostrarán las fuentes que sean símbolos.
fdEffects	Permite dotar de efectos a las fuentes
fdFixedPitchOnly	Sólo aparecerán letras de espaciado fijo.
fdForceFontExist	No se permitirá la entrada de un tipo de letra inexistente.
fdScalableOnly	Sólo aparecen fuentes escalables, sean TrueType o no.
fdTrueTypeOnly	Sólo aparecen fuentes TrueType.

fdShowHelp	Muestra un botón de ayuda.
fdWysiwig	Permite fuentes que aparezcan con el mismo aspecto en la impresora y en la pantalla

Puesto que ya disponemos de un sitio donde visualizar texto, añadiremos un control **FontDialog** para cambiar el tipo de letra con el que se muestra, y un botón que llamaremos **Fuentes**. El código necesario para responder al evento **OnClick** será:

```
procedure TForm2.FuentesClick(Sender: TObject);  
  
begin  
  
    FontDialog1.Execute;  
  
    Memo1.Font:=FontDialog1.Font;  
  
end;
```

COLORES (ColorDialog) TColorDialog

La propiedad **Color** de este componente nos servirá para obtener el color que el usuario haya seleccionado en el cuadro de diálogo.

Podemos facilitar una serie de colores definidos a medida mediante la propiedad **CustomColors**, que es un objeto del tipo **TStrings**. A ella añadiremos una cadena por cada color conteniendo la palabra **Color** seguida de una letra de la **A** al la **P**, un signo igual y una secuencia hexadecimal especificando los componentes RGB del color.

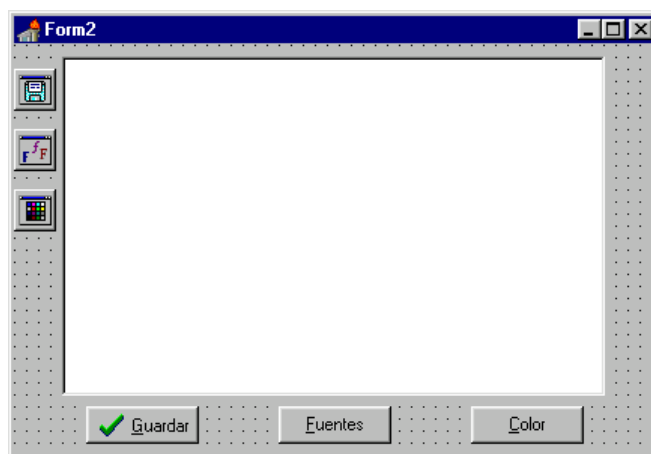
La propiedad **Options** cuenta con los valores siguientes:

- **cdPreventFullOpen**: No permite seleccionar un color a partir de sus componentes.
- **cdFullOpen**: Muestra la ventana de definición de colores personalizados.
- **cdShowHelp**: Muestra un botón de ayuda.

Ampliamos nuestro ejercicio añadiendo un control **ColorDialog** para cambiar el color de la letra con el que se muestra el texto, y un botón que llamaremos **Color**. El código necesario para responder al evento **OnClick** será:

```
procedure TForm2.ColorClick(Sender: TObject);  
  
begin  
  
    ColorDialog1.Execute;  
  
    Memo1.Font.Color:=ColorDialog1.Color;
```

end;



IMPRESIÓN (PrintDialog) TPrintDialog

Permite elegir las opciones previas a la impresión de un texto solicitando al usuario cierta información.

Mediante la propiedad **PrintRange** podemos elegir el trozo de texto a imprimir, puede tomar los valores siguientes:

Valor	Información a imprimir
prSelection	La selección actual
prPageNum s	Las páginas especificadas
prAllPages	Todo el documento

El número de copias se indica en la propiedad **Copies**, que por defecto es uno. El orden en que habrán de imprimirse se determina por la propiedad **Collate**, que puede ser **True**, con lo que las copias serán consecutivas de cada página, o **False**, con lo que se imprimirá una copia completa cada vez. La propiedad **PrintToFile** tomará el valor **True** si se permite realizar la impresión sobre un archivo. Los valores de la propiedad **Options** son los siguientes:

Valor	Significado
poHelp	Aparece un botón de ayuda.
poPageNum	Permite seleccionar un rango de páginas en los valores MinPage y MaxPage.
poPrintToFile	El usuario puede imprimir en un archivo.
poSelection	Permite al usuario que elija imprimir la selección actual de texto en el documento.
PoWarning	Avisa que no existe una impresora instalada.
PoDisablePrintToFile	No permite imprimir en un fichero

Añadamos un control **PrintDialog** a nuestro ejemplo y un botón que llamaremos **Imprimir**. El código necesario para responder al evento **OnClick** será:

```

procedure TForm2.ImprimirClick(Sender: TObject);

var

    nLinea: Integer; PrintText: TextFile;

begin

    If PrintDialog1.Execute then

        begin

            AssignPrn(PrintText);

            Rewrite(PrintText);

            Printer.Canvas.Font:=Memo1.Font;

            for nLinea:=0 to Memo1.Lines.Count - 1 do

                Writeln(PrintText, Memo1.Lines[nLinea]);

            System.CloseFile(PrintText);

        end:

    end;

```

La función **AssignPrn()** se encarga de asignar un archivo auxiliar al dispositivo de impresión, después de esto se fuerza una primera escritura que provoca la apertura física del archivo declarado con la función **Rewrite()**. Por fin el contenido es actualizado en la impresora por medio de la propiedad **Canvas** de la variable **Printer** (variable que se crea automáticamente para tener una instancia de la clase **TPrinter**). Una vez definidos los pasos iniciales pasamos a recorrer el texto a imprimir:

```
for nLinea:=0 to Memo1.Lines.Count - 1 do
```

```
    Writeln(PrintText, Memo1.Lines[nLinea]);
```

CONFIGURACIÓN DE LA IMPRESORA (PrinterSetupDialog)

Permite cambiar las propiedades de la impresora, aunque se pueda hacer directamente desde la ventana de diálogo que contiene las opciones de impresión.

Esta ventana no cuenta con ninguna propiedad que no conozcamos ya, por lo que para utilizarla sólo tendremos que llamar al método **Execute**.

BUSQUEDA DE TEXTO (FindDialog)

Permite solicitar información al usuario para iniciar un proceso de búsqueda.

La propiedad **Options** permite configurar las opciones disponibles en el cuadro de diálogo, sus valores son:

Valor	Significado
frDisableMatchCase	Habilita o no la búsqueda respetando el orden de mayúsculas y minúsculas.
frDisableUpDown	Habilita o no la dirección de la búsqueda.
frDisableWholeWord	Habilita o no la opción Match Whole Word
frDown	Indica que la búsqueda es hacia abajo.
frHideMatchCase	Oculto la opción Match Case.
frHideWholeWord	Oculto la opción Match Whole Word Only.
frHideUpDown	Oculto la opción Direction.
frMatchCase	Indica que la opción Match Case ha sido activada por el usuario.
frShowHelp	Muestra un botón de ayuda en la ventana.
frWholeWord	Indica que ha sido activada por el usuario la opción Match Whole Word.

Cada vez que el usuario pulse el botón **Find Next**, el componente generará un evento **OnFind**, en cuyo gestor podemos añadir el código necesario para obtener el texto a buscar, de la propiedad **FindText**. El cuadro de diálogo no se oculta, podemos usar el método **CloseDialog** o pulsar el botón **Cancel** para hacerlo. Mientras esté en marcha el proceso de búsqueda podemos utilizar la propiedad **Position** para desplazar la ventana a otro punto.

Si añadimos un control **FindDialog** y un botón **Buscar** tendremos que codificar dos procedimientos. Uno que llamaremos **BuscarClick** asociado al evento **OnClick** del botón y otro que llamaremos **Encontrado** asociado al evento **OnFind** del control **FindDialog**.

```

procedure TForm2.BuscarClick(Sender: TObject);
begin
    nLineaActual:=0;
    FindDialog1.Position:=Point(30,30);
    FindDialog1.Execute;
end;

procedure TForm2.Encontrado(Sender: TObject);
var
    I, J, nPos, nCarSaltar: Integer;
begin
    I:=nLineaActual;
    while I <= Memo1.Lines.Count -1 do
    begin
        nPos:=Pos(FindDialog1.FindText,Memo1.Lines[I]);
        if nPos <> 0 then
            begin
                begin
                    nCarSaltar:=0;
                    for J:=0 to I -1 do
                        nCarSaltar:=nCarSaltar + Length(Memo1.Lines[J]);
                    nCarSaltar:= nCarSaltar + (I*2);
                    nCarSaltar:=nCarSaltar + nPos -1;
                    Memo1.SetFocus;
                    Memo1.SelStart:=nCarSaltar;
                    Memo1.SelLength:=Length(FindDialog1.FindText);
                    nLineaActual:=I+1;
                    I:=Memo1.Lines.Count;
                end
                else MessageBeep(MB_ICONHAND);
                I:=I+1;
            end;
        end;
    end;
end;

```

En el primer método se inicializa una variable, que declararemos privada, llamada **nLineaActual** en la que almacenaremos en qué línea se ha quedado la última búsqueda y reposicionamos la ventana en la pantalla a fin de que no se solapen.

En el segundo se desarrolla el algoritmo que se encarga de encontrar el valor de la cadena, almacenada en **FindText**, averiguar su posición, sobreiluminar el texto y recordar cuál fue la última línea donde se halló la cadena deseada; en caso de no encontrar la ocurrencia se provoca un tono de aviso mediante la función del **API** de **Windows MessageBeep(MB_ICONHAND)**.

Hay que tener en cuenta que la aparición repetida de una cadena en la misma línea no se hallaría, pues sólo se tiene en cuenta la línea por la que se va y no la columna. Queda por cuenta del alumno realizar una versión más completa.

SUTITUCIÓN DE TEXTO (ReplaceDialog)

La apariencia y funcionamiento es similar al anterior componente. Permite sustituir un texto buscado por otro valor.

ReplaceText contendrá el texto que sustituirá al buscado. Al pulsar el botón **Replace** o **Replace All** se producirá un evento **OnReplace**. La opción **Options** contiene además las constantes **frReplace** o **frReplaceAll**.

6. OTRAS COMPONENTES BÁSICAS

EVENTOS PERIÓDICOS (Timer)

Mediante este componente que encontramos en la paleta **System** podemos programar un evento que se generará periódicamente, con la frecuencia que determinemos.

Es un componente no visual, por lo que en tiempo de ejecución no aparece en la ficha. Para determinar el tiempo entre dos eventos utilizamos la propiedad **Interval**, que es un valor dado en milisegundos. Por defecto vale 1000. Cada vez que transcurra el tiempo indicado en Interval se generará un evento **OnTimer**.

La propiedad **Enabled** permite que el evento se genere si vale **True**. Con el valor **False** no se produce.

AREAS DE DIBUJO (PaintBox)

Este componente que esta en la paleta **System** permite dibujar imágenes restringiendo el área de dibujo a las dimensiones que hayamos asignado al componente.

La ventaja de usar un componente **PaintBox** sobre un componente **Image** las encontramos en la menor necesidad de memoria, ya que el primero lo único que hace es limitar el área de dibujo a una porción de la ficha, sin incorporar ninguna función adicional.

REJILLAS DE DATOS (DrawGrid)

Este componente se encuentra en la paleta **Additional** y permite crear matrices o rejillas de datos en las que es posible crear títulos o editar el contenido.

- **DIMENSIONES DE LA REJILLA**

El número de filas y columnas viene determinado por las propiedades **RowCount** y **ColCount**. Algunas de estas filas y columnas pueden estar fijas, es decir que permanecen siempre en el margen izquierdo o superior de la

rejilla sin desplazarse. El número de filas fijas se determina por **FixedRows** y el de columnas por **FixedCols**.

El tamaño de las celdillas se fija en **DefaultColWidth** y **DefaultRowHeight**, que contienen el número de puntos de ancho y alto. Se puede establecer el tamaño de una celdilla individual mediante las propiedades **ColWidths** y **RowHeights**, que son matrices de enteros, con tantos elementos como columnas o filas existan. El grosor de la línea con la que se dibuja la cuadrícula está en la propiedad **GridLineWidth**. El número de filas y columnas visibles lo proporcionan las propiedades **VisibleRowCount** y **VisibleColCount**. El valor de la primera columna visible está en **LeftCol** y la primera fila visible en **TopRow**.

• COLORES

La propiedad **Color** especifica el fondo de la rejilla, y se puede establecer el color de las filas y columnas fijas mediante **FixedColor**.

• CELDILLA ACTIVA

Mediante **Row** y **Col** sabemos cuál es la celdilla activa que aparecerá destacada de las demás mediante un recuadro más grueso.

• OPCIONES

El funcionamiento del componente está controlado por la propiedad **Options** que es un conjunto con los posibles valores siguientes:

Valor	Descripción
goFixedHorzLine	Mostrar líneas de separación de filas en las columnas fijas
goFixedVertLine	Mostrar líneas de separación de columnas en las filas fijas
goHorzLine	Dibujar líneas horizontales de separación
goVertLine	Dibujar líneas verticales de separación
goRowSizing	Permitir el cambio de altura de las filas
goColSizing	Permitir el cambio de anchura de las columnas
goRowMoving	Permitir el movimiento de las filas
goColMoving	Permitir el movimiento de las columnas
goRangeSelect	Permitir la selección de un grupo de celdillas
goDrawFocusSelected	Mostrar la celdilla seleccionada del mismo color de las otras
goEditing	Permitir la edición de las celdillas
goAlwaysShowEditor	Activar siempre el modo de edición
goTabs	Permitir el uso del tabulador para desplazarse
goRowSelect	Permitir sólo la selección de filas completas
goThumbTracking	Desplazar el contenido de la rejilla a medida que se

arrastra el cursor de las barras de desplazamiento

Si se permite la selección de un rango de celdillas, las celdillas seleccionadas están disponibles en la propiedad **Selection**, que es un objeto del tipo **TGridRect**.

- **EVENTOS DE DRAWGRID**

No existe propiedad mediante la cual podamos establecer el contenido de cada celdilla. Debemos utilizar una matriz o un archivo para presentar en cada celdilla los datos que necesitemos.

Cuando dibujemos el contenido de una celda, se generará un evento **OnDrawCell**, pasando los siguientes parámetros:

- **ACol** y **ARow**: Número de columna y fila de la celdilla.
- **ARect**: Un objeto del tipo **TRect** conteniendo el área de dibujo correspondiente a esa celdilla.
- **AState**: Estado de la celdilla. Conjunto con los valores: **gdSelected**, si la celdilla está seleccionada, **gdFocused**, si es la celdilla activa y **gdFixed** si es una celdilla fija.
- **Edición de las celdillas**

En el momento en que se inicia la operación de edición se generan los eventos **OnGetEditMask** y **OnGetEditText**. El primero obtiene una máscara de entrada de datos y el segundo para facilitar al componente el valor de la celdilla. Estos eventos reciben los mismos parámetros **ACol** y **ARow** para conocer qué celdilla se va a editar y **Value** que devolverá la máscara o el propio dato.

Cuando se realiza la introducción o modificación de un dato se generan eventos **OnSetEditText**, mediante los cuales recibiremos los parámetros **ACol** y **ARow** y el parámetro **Text**, conteniendo el valor actual introducido por el usuario.

- **Selección de celdillas**

Cuando se realiza un desplazamiento se genera un evento **OnSelectCell**, que cuenta con los parámetros **ACol** y **ARow** así como el parámetro **CanSelect** que es un valor booleano que permite o no el desplazamiento a dicha celdilla.

- **Movimiento de columnas y filas**

Cuando se desplacen columnas o filas se generan eventos **OnColumnMove** o **OnRowMove**, a partir de los cuales obtendremos los parámetros **FromIndex** y **ToIndex** para saber que columna ha sido desplazada y a dónde se ha soltado.

REJILLAS DE CADENAS (StringGrid)

Este control que se encuentra en la paleta **Additional** permite la gestión del contenido de cada una de las celdillas de una manera más cómoda.

Si activamos la opción de edición podemos introducir datos en las celdillas. Cada celdilla puede contener una cadena de texto, además de tener asociado cualquier objeto que nos interese. A las celdillas podemos acceder individualmente mediante **Cells**, o bien en forma de listas de cadenas mediante **Cols** y **Rows**. Los objetos asociados a cada celdilla están accesibles mediante **Objects**. Tanto **Cells** como **Objects** son matrices bidimensionales siendo el primer índice la columna y el segundo la fila.

Todas las opciones, propiedades y eventos vistos para **DrawGrid** están disponibles para **StringGrid**. Por parte del programador es más cómodo el uso de **StringGrid**, al no tener que preocuparse del almacenamiento de datos y su suministro al control, pero el uso de **DrawGrid** ahorra memoria. El uso de uno u otro control dependerá del tamaño de la rejilla y si la cantidad de memoria ahorrada merece el trabajo adicional de codificación.

LISTAS JERÁRQUICAS (Outline)

Este control se encuentra en la página **Additional** y permite crear una lista jerárquica, es decir, una lista en la que cada elemento puede a su vez contener otros elementos.

• CONTENIDO DE LA LISTA

Cada uno de los elementos del control **Outline** es un objeto del tipo **TOutlineNode**, capaz de contener un texto y un dato adicional asociado. Además cada nodo tiene un nivel. En la lista es posible mostrar el nodo cerrado o abierto, desplegando todos los elementos que contiene.

Una lista contiene tantos nodos como indique la propiedad **ItemCount** y es posible acceder a cada uno de ellos mediante **Items**, que es una matriz de objetos **TOutlineNode**, siendo su base 1 y no cero.

El dato asociado a cada elemento está almacenado en la propiedad **Data** del objeto **TOutlineNode**. El número de nivel de cada elemento está en la propiedad **Level** del objeto **TOutlineNode**. Si un elemento tiene otros en su interior la propiedad **HasItems** vale **True**. La propiedad **Index** del objeto **TOutlineNode** indica el índice de un cierto nodo y **TopParent** indica cuál es el índice del nodo padre. Un nodo puede ser desplazado a un nivel superior o inferior pasándole 1 o -1 al método **ChangeLevelBy**.

La propiedad **Expanded** indica si el nodo está abierto o cerrado. El método **Expand** abre el nodo, el método **FullExpand** abre el nodo y todos los que existan en su interior. El método **Collapse** cierra el nodo.

• AÑADIR ELEMENTOS A LA LISTA

El método **Add** toma como primer parámetro el índice de un elemento del mismo nivel que el que se desea añadir, y como segundo parámetro el título que aparecerá en la lista. De forma similar **AddChild** toma los mismos parámetros pero el nuevo elemento se convertirá en hijo del elemento facilitado.

Podemos usar los métodos **AddObject** y **AddChildObject** equivalente a los anteriores pero con un tercer parámetro, que es una referencia a la propiedad **Data** del objeto **TOutlineNode** que se creará al añadir el nuevo elemento.

Para insertar un elemento en una posición específica usaremos **Insert** o **InsertObject** de la misma forma que los métodos anteriores.

En todos los casos obtendremos un valor de retorno en el que se indica el índice del nuevo elemento.

• VISUALIZACIÓN DE LA LISTA

Mediante **OutlineStyle** indicamos cómo va a ser visualizada la lista, tomando los valores siguientes:

Valor	Se visualiza
osText	Sólo el texto que actúa como título
osTreeText	Los títulos y líneas formando ramas
osPictureText	Títulos con iconos a la izquierda
osTreePictureText	Combinación de los dos anteriores
osPlusMinusText	Títulos y los símbolos + y -
osPlusMinusPictureText	El anterior y los iconos de osPictureText

Los iconos usados pueden ser modificados mediante las propiedades **PictureOpen**, **PictureClosed**, **PictureLeaf**, **PicturePlus** y **PictureMinus**, que son objetos **TBitmap**.

• SELECCIÓN ACTUAL

Podemos conocer el índice del elemento seleccionado mediante la propiedad **SelectedItem**.

7. FICHEROS EN DELPHI

La verdadera potencia de un ordenador reside en su capacidad de tratamiento de datos. Pero no tiene ningún sentido sino podemos almacenar los datos, ya que entonces cada vez que necesitáramos hacer algo deberíamos introducir los datos, y además no podríamos conservar los resultados.

En Delphi podemos tratar ficheros tanto en Ascii como en binario. Los ficheros que vamos a tratar son secuenciales, ya que para almacenar datos en otras estructuras mejores, como estructuras indexadas, tenemos las bases de datos como Dbase o Paradox, entre otras, las cuales Delphi maneja a la perfección.

FICHEROS ASCII

Los ficheros en formato Ascii son las más sencillas de todos. Un clásico fichero Ascii en el Config.sys o el Autoexec.bat, o casi cualquier Script de configuración de cualquier dispositivo. Son ficheros que se pueden visualizar con la orden type del Msdos, o con el Notepad de Windows, suelen llevar extensión txt.

Para leer o escribir un fichero de este tipo lo primero que tenemos que hacer es indicar que se trata de un fichero de tipo texto, luego crearemos una variable con la cual leeremos o escribiremos, y como estamos tratando ficheros de tipo ascii, esta tendrá que ser una cadena de caracteres (un string). Lo siguiente es indicarle a Delphi como se llama nuestro fichero y con que nombre se conocerá en el programa. Luego lo abrimos, hay tres formas de hacerlo, en modo lectura, en modo escritura para añadir, o en modo reescritura. El modo lectura no tiene nada que comentar, en cambio los otros dos modos tienen su peculiaridad, así en modo escritura abrimos un fichero para añadir cadenas a él, y el modo reescritura crea el fichero y si este ya existiera con anterioridad sería sobrescrito, así que mucho cuidado.

Vamos a crear un ejemplo donde introduzcas tu nombre y este sea grabado en un fichero de texto. Para ello pon un componente Edit y un botón en un formulario y asocia este código a su evento OnClick.

```
procedure TForm1.Button1Click(Sender: TObject);
```

Var

MiFichero : TextFile;

Nombre : **String**;**begin**

Nombre := Edit1.Text;

AssignFile (MiFichero,'nombre.txt');

Rewrite (MiFichero);

writeln(MiFichero,Nombre);

CloseFile (mifichero);

end;

Si pruebas este programa verás que por muchos nombre que metas solo el último es grabado, el motivo es que cada vez que invocas este procedimiento el fichero es abierto con la instrucción Rewrite, la cual crea el fichero sin importarle si existe o no. Observa la secuencia de las instrucciones, dentro de la sección de la variables lo importante es la asignación es la variable TextFile, la cual determina el tipo de fichero que vamos a utilizar. Dentro del bloque de instrucciones tenemos la instrucción AssignFile, que es para indicar como se llama el fichero de texto que vamos a usar, y lo asigna a Mifichero, que es el nombre con el que será conocido de ahora en adelante. Luego lo abrimos con la orden Rewrite, la cual ya os comenté que crea el fichero exista o no exista. Después grabamos los datos en nuestro fichero con la orden Writeln indicando que lo grabamos en nuestro fichero, ya que podemos tener más de un fichero abierto y de diferentes tipos, sin queremos grabar más cosas debemos hacerlo antes de cerrar el fichero, ya que sino cuando lo volvamos a abrir este se creará.

Yo he usado la orden Writeln, que graba una cadena de caracteres con un retorno de carro al final, con lo que si grabáramos otra cadena de caracteres esta se grabaría en la línea de abajo. En cambio si usas la orden Write, no se graba la cadena con el retorno de carro al final, con lo que si grabas otra cadena de caracteres esta sería grabada a continuación. Prueba a cambiar la orden y ver las diferencias en el fichero creado.

Para leer el fichero, el proceso es similar, la diferencia está en que no podemos usar una orden de escritura sino que debemos usar una de lectura. Coloca un nuevo botón en el proyecto que estamos construyendo y una etiqueta, dentro del manejador Onclick del botón escribe lo siguiente:

```
procedure TForm1.Button2Click(Sender: TObject);  
Var  
MiFichero : TextFile;  
Nombre : String;  
begin  
AssignFile (MiFichero,'nombre.txt');  
Reset (MiFichero);  
Read(MiFichero,Nombre);  
CloseFile (mifichero);  
Label1.Caption := Nombre;  
end;
```

Prueba el programa, y verás que después de haber grabado el nombre, este es leído. Pero ten en cuenta que si tratas de abrir un fichero para leer, y este no existe obtendrás un error. Para controlar este error lo primero que se te puede venir a la cabeza es preguntar si existe el fichero primero, lo cual estaría bien si decides tomar medidas al respecto, como puede ser crearlo, pero si solo deseas controlar el error, y evitar un mensaje en inglés lo puedes hacer con las instrucciones Try Except, las cuales explique en el capítulo de errores. Esta última opción es la que tomé yo en el programa de ejemplo que acompaña a esta página.

Para preguntar si existe un fichero debes usar la función FileExists, la cual devuelve True en el caso de que exista el fichero, la manera de usarla es como sigue:

```
If FileExists ('Nombre.txt') Then  
showmessage ('El fichero existe')  
Else  
Showmessage ('El fichero no existe');
```

Para recorrer un fichero (del tipo que sea), existe la orden Seek, la cual avanza por el fichero, en el caso de un fichero de texto la orden ReadLn avanza línea a línea, haciendo el mismo efecto. Seek, esta orientado más a fichero de tipo binario, lo veremos más adelante, aunque repito que la orden de lectura ya avanza el puntero de lectura. Para ilustrarlo mejor, desde una aplicación nueva, vamos hacer un programa que cuente el número de veces que aparece la palabra Rem, en el autoexec.bat de tu sistema. Coloca y botón, y dos etiquetas, una con el texto:"Número de rem:", y la otra la pones a continuación y sin texto. Ahora en evento Onclik del botón sitúa este código.

```
procedure TForm1.Button1Click(Sender: TObject);  
Var  
Fichero : TextFile;  
Linea : String;  
Contador: Integer;  
begin  
Contador := 0;  
AssignFile (Fichero,'c:\autoexec.bat');  
Reset (Fichero);  
While Eof(Fichero) = False Do  
Begin  
ReadLn(Fichero,Linea);  
If Pos ('rem ',lowerCase(Linea)) > 0 Then  
Inc (Contador);  
End;  
CloseFile (Fichero);  
Label2.Caption := IntToStr (Contador);  
End;
```

Un detalle, a la hora de gestionar fichero, es la manera de construir el bucle que recorre el fichero. Esta hecho con un While, también se podía hacer con un repeat until. La diferencia radica en que un while comprueba la condición antes de entrar en el bucle, y un repeat ejecuta el bloque y luego comprueba la condición con lo cual por lo menos una vez en leído el fichero. Pero que pasa, si el fichero esta vacío, pues que la mejor decisión es averiguar si el fichero

esta vacío antes de leer nada, y evitarme trabajo inútil, y quizás sorpresas, por eso el bucle esta hecho con un While. He usado una función que es Pos la cual devuelve valores según encuentre una cadena dentro de otra. Así en el ejemplo si la cadena Rem esta dentro de la línea que hemos leído, pues el valor devuelto será mayor de cero, siendo este el valor del desplazamiento dentro de la cadena. Esto viene a que Delphi (por lo menos en la versiones de 32 bits) considera las cadenas como un tabla Matriz unidimensional, donde cada carácter puede ser leído por su posición, empezando por uno, ya que el supuesto carácter cero contiene información usada por Delphi para almacenar la cadena. Entonces si en una cadena quieres leer el carácter que esta en la posición 2 puedes hacer esta asignación:

carácter := Cadena [2];

Ojo que si lees más allá de la longitud de la cadena Dios sabe que devolverá el sistema, así que para saber la longitud de una cadena usa Lenght (cadena), que devuelve la longitud de la cadena. Otra función que he usado es LowerCase, la que convierte toda a minúsculas una cadena, lo hago para asegurarme que lo que haya en la cadena este en minúsculas y así el posible rem que busco coincida. También se podía hacer al revés con UpperCase, que convierte a mayúsculas una cadena.

Por último la última función que he usado es IntToStr, que convierte un número entero a cadena, esta función genera un error si no es posible. Tiene un primo cercano que es StrToInt, que convierte de Cadena a entero. Creo que no hace falta comentar que Inc(contador) en lo mismo que Contador := Contador +1. Ten en cuenta que Inc solo se permite con variables numéricas no con propiedades.

La tercera forma de abrir un fichero es para añadir datos, para lo cual debes usar Append (fichero), en lugar de Read o Write. Append abre un fichero existente para escribir en él, y todo lo que escribas lo hace al final de este. La mecánica es idéntica a los ejemplo anteriores.

FICHEROS BINARIOS.

Los ficheros binarios son todo lo contrario a los Ascii, en el sentido que no pueden ser leído con un Type, o con un procesador de Texto, solo pueden ser leídos por el programa que los haya creado, o en su defecto uno que haya sido programado para leerlos. Para poder trabajar con ellos debemos leer/escribir bytes, si queremos podemos definir un estructura, la cual nos servirá para leer un serie de Bytes seguidos, y así poder interpretar la información y manejarla con mayor soltura.

La ventaja es que tenemos dos de funciones que no había en los fichero anteriores, como son Filesize, y FilePos. Un ejemplo puede ser el siguiente, en un formulario coloca un botón, una etiqueta, y un campo Edit, de nuevo en el manejador Onclik del botón coloca los siguiente:

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
Var
```

```
Fichero : File Of Byte;
```

```
Valor : Byte;
```

```
begin
```

```
AssignFile (Fichero,Edit1.text);
```

```
Reset (Fichero);
```

```
Seek(Fichero,11);
```

```
Read (Fichero,Valor);
```

```
CloseFile (Fichero);
```

```
Label1.Caption := IntToStr (Valor);
```

```
end;
```

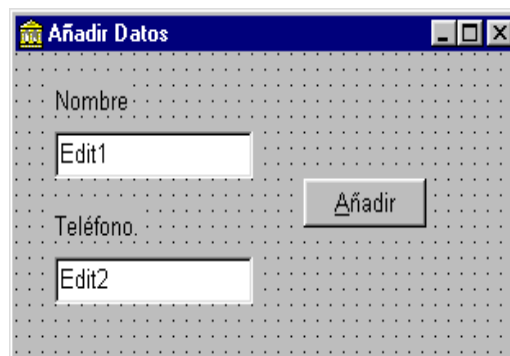
Lo que conseguimos con este fichero es leer el Byte 12 de un fichero que hemos introducido su nombre en el campo Edit. He dicho byte 12 porque se cuentan desde 0.

Lo interesante de este sistema es poder crear un fichero binario con un tipo de datos determinado por nosotros, para lo cual debemos declarar una estructura de datos, luego indicar que el fichero debe usar ese registro y todo solucionado. Para ello debes declarar la estructura al principio de la unidad,

justo después de la cláusula `uses` y la declaración del formulario. Yo he declarado un registro como el que sigue:

```
Type TMiRegistro = Record  
Nombre : String [20];  
Telefono : String [7];  
End;
```

He hecho este registro porque vamos a hacer un ejemplo para grabar unos datos en fichero. En un formulario coloca dos etiquetas, un botón y un par de campos edit, como la imagen que hay debajo de este párrafo.



Debes tener en cuenta que al crearse el formulario es el momento de abrir el fichero, y cuando la aplicación se cierra debes cerrar el fichero. Y como no, en el evento `OnClick` del botón pondremos las ordenes necesarias para grabar los datos. Así el código fuente de los tres eventos queda como sigue.

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
AssignFile (Fichero,'Datos.dat');  
Rewrite (Fichero);  
end;  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
With MiRegistro Do  
Begin  
Nombre := Edit1.Text;  
Telefono := Edit2.Text;  
End;
```

```
Write (Fichero,MiRegistro);  
end;  
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);  
begin  
  CloseFile (Fichero);  
end;
```

Fíjate que no he declarado las variables dentro de ningún procedimiento, sino que lo he hecho en la sección Private de la unidad, para que así estén disponibles en todos los procedimientos. Estas son las declaraciones de la sección Private:

```
private  
{ Private declarations }  
MiRegistro : TMiRegistro;  
Fichero : File of TMiRegistro;
```

He construido un programa a parte que lee los datos creados por este programa. Su estructura es casi idéntica, piensa lo que cambiaría y como lo harías.

La principal función, a mi modo de ver, de los ficheros binarios es el de almacenar algún tipo de datos que se leen secuencialmente y sirven para un motivo en concreto dentro de un programa. Porque si quieres almacenar datos en general, y gestionarlos eficazmente pues es mejor usar una base de datos. Incluso yo le veo más uso a los ficheros de texto (ascii) ya que siempre se usan para los Scripts, o los ini, aunque Delphi presenta una forma de leer y escribir los ficheros ini, muy efectiva que veremos más adelante. En el próximo capítulo veremos como leer y escribir un fichero de texto con un control de Delphi, el cual es un pequeño editor de texto, así como usar los cuadros de dialogo estándar de Windows para abrir o cerrar un fichero, y espero que alguna sorpresa, depende del tiempo que tenga.

8. APLICACIONES MDI

Determinados tipos de aplicaciones, como son los editores de texto, hojas de cálculo, etc., tienen la capacidad de editar múltiples documentos simultáneamente. Para ello disponen de una ventana principal en cuyo interior aparecen ventanas a las que se denominan hijas.

Nosotros también podemos construir aplicaciones de este tipo mediante el uso de **MDI (Interfaz de Documento Múltiple)**, que es el nombre de una serie de servicios **Windows** que facilitan esta técnica.

LA VENTANA PRINCIPAL

Una aplicación **MDI** cuenta con una ficha principal, que servirá como marco de trabajo para todas las demás ventanas del programa. Esta ventana se distingue porque su propiedad **FormStyle** tiene el valor **fsMDIForm**, no existiendo otra ventana con este atributo.

La ficha principal **MDI** puede contener otros controles, pero lo normal es que en ella sólo insertemos componentes no visuales, como menús o cuadros de diálogo y uno o dos controles Panel, que servirán como barra de botones y línea de estado. Cualquier otro control se superpondrá a las ventanas hijas, interfiriendo en su funcionamiento.

VENTANAS HIJAS

La ficha que actúe como ventana hija de una aplicación **MDI**, se caracteriza por tener el valor **fsMDIChild** en la propiedad **FormStyle**, lo que asegura su correcta asociación con la ventana principal.

Dado que el uso del **MDI** tiene como finalidad el trabajo con múltiples documentos, lo que implica abrir múltiples ventanas, la ficha que creamos en tiempo de diseño servirá como plantilla de ventana a crear en tiempo de ejecución. Normalmente, las fichas que actúan como hijas no son creadas al cargar la aplicación, sino que se crean en tiempo de ejecución conforme son necesarias.

Las fichas que actúan como hijas pueden contener un menú de opciones. En tiempo de ejecución las opciones siempre aparecerán en la ventana principal, nunca en una ventana hija. Para controlar que las opciones del segundo menú aparezcan en el primer menú usaremos la propiedad **GroupIndex** estudiada en el tema de Menús.

GESTIÓN DE LAS VENTANAS HIJAS

En cualquier momento podemos conocer el número de ventanas hija existentes consultando la propiedad **MDIChildCount**. Con este número podemos acceder a cualquier de ellas mediante la propiedad **MDIChildren** que es una matriz de objetos **TForm**.

- **LA VENTANA HIJA ACTIVA**

De todas las ventanas abiertas sólo una de ellas puede ser la activa. Podemos obtener una referencia a ella mediante **ActiveMDIChild**, de tal forma que una cierta acción de, por ejemplo, una opción del menú principal, se lleve a cabo sobre la ventana que esté en ese momento activa.

Podemos cambiar de ventana activa con una pulsación de ratón o teclado. Mediante los métodos **Previous** y **Next** de la ficha principal podemos activar por código la ventana siguiente o anterior a la actual.

- **DISPOSICIÓN DE LAS VENTANAS**

A medida que se crean ventanas, en tiempo de ejecución, éstas irán tomando una posición y dimensiones por defecto. Mediante los métodos **Cascade** y **Tile** podemos modificar la posición y dimensiones de todas las ventanas hijas, disponiéndolas en forma de cascada, o en forma de mosaico.

Las ventanas minimizadas en el espacio de la ventana principal no se verán afectadas por estos dos métodos, pero podemos usar el método **ArrangeIcons** para ordenar los iconos que representan a estas ventanas.

- **VENTANAS HIJAS Y MENÚS**

Habitualmente entre las opciones del menú principal suele existir una llamada **VENTANA**, en la que existen una serie de opciones para establecer la disposición de las ventanas, ordenar los iconos, etc., también suele completarse con una lista de todas las ventanas hijas, facilitando así la activación de una cierta ventana.

Para crear y mantener esta lista debemos dar el valor adecuado a la propiedad **WindowMenu** de la ficha principal, asignándole una referencia a la opción de menú correspondiente.

- **OTRAS CONSIDERACIONES**

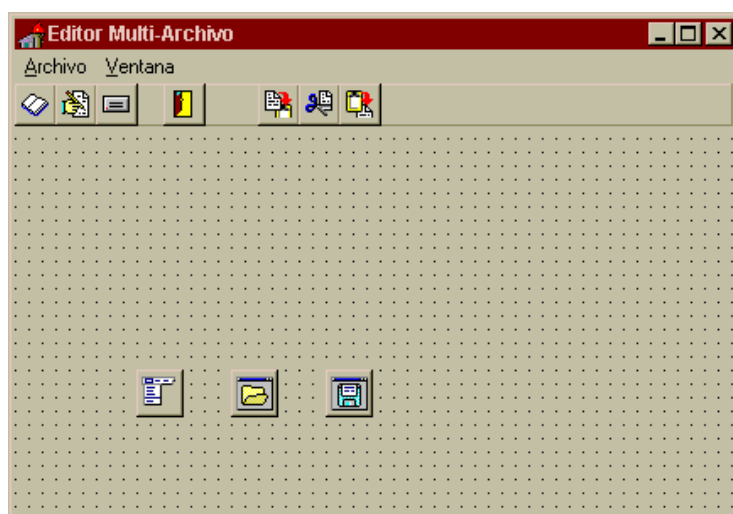
Dado que las ventanas hijas de una aplicación **MDI** son fichas creadas en tiempo de ejecución no basta con utilizar el método **Close** para eliminarla, ya que esto tendrá el efecto de minimizarla en la ventana principal.

Al igual que otros componentes creados dinámicamente deberemos usar el método **Free**. Hecho esto, la ventana desaparece de **MDIChildren**, la propiedad **MDIChildCount** será decrementada y se actualizará la lista de ventanas del menú apuntado por **WindowMenu**.

UN EJEMPLO

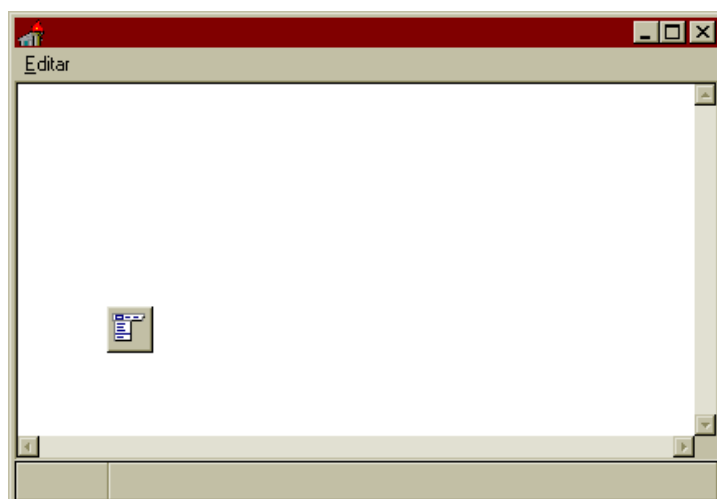
Hemos visto hasta ahora lo simple que es crear una aplicación MDI con Delphi, basta con usar cinco propiedades y otros tantos métodos. Veámoslo ahora en la práctica, escribiendo una aplicación que nos permita editar múltiples archivos de texto. Para ello partiremos de un nuevo proyecto, cuya ficha inicial actuará como ventana principal de la aplicación, incorporando un menú con las opciones principales y una barra de botones.

La disposición de los controles y componentes sobre la ficha principal de la aplicación sería la siguiente:



El control **Panel** se ajusta a la parte superior de la ventana, de tal forma que aunque ésta sea redimensionada, los botones siempre aparezcan colocados correctamente. Los iconos utilizados en estos botones han sido tomados del directorio IMAGES\BUTTONS, que se encuentra en el directorio de trabajo de Delphi.

A continuación insertaremos una nueva ficha en el proyecto, que será la que actúe como plantilla de ventana hija. En esta ficha insertaremos los siguientes elementos: un componente menú con opciones de edición, un control **Memo**, que servirá para la edición de texto, y un control **Header**, que en este ejemplo lo vamos a utilizar como si se tratase de un control **Panel**. En la siguiente figura se puede observar el aspecto de esta ficha.



El último paso será la creación de todo el código necesario para responder a las opciones de los menús, la pulsación de los botones de la barra y algún evento más. El listado PRINCIPA.PAS contiene el código correspondiente a la ficha principal, y que por lo tanto es genérico para toda la aplicación.

```
{ Módulo de la ficha principal }
```

```
unit Principal;
```

```
interface
```

```
uses
```

```
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms,  
  Dialogs, Menus, EditText, Buttons, ExtCtrls, StdCtrls;
```

```
type
```

```
  TFPPrincipal = class(TForm)
```

```
    MenuPrincipal: TMainMenu;
```

```
    Archivo1: TMenuItem;
```

```
    Nuevo1: TMenuItem;
```

```
    Salvar1: TMenuItem;
```

```
    N1: TMenuItem;
```

```
    Salir1: TMenuItem;
```

```
    Ventana1: TMenuItem;
```

```
    Cascada1: TMenuItem;
```

Mosaico1: TMenuItem;

Ordenariconos1: TMenuItem;

N2: TMenuItem;

Minimizartodas1: TMenuItem;

Abrirtodas1: TMenuItem;

Cerrartodas1: TMenuItem;

Abrir1: TMenuItem;

CDAbrir: TOpenDialog;

CDSalvar: TSaveDialog;

Panel1: TPanel;

BAbrir: TSpeedButton;

BNuevo: TSpeedButton;

BSalvar: TSpeedButton;

BSalir: TSpeedButton;

BCopiar: TSpeedButton;

BCortar: TSpeedButton;

BPegar: TSpeedButton;

procedure Nuevo1Click(Sender: TObject);

procedure Cascada1Click(Sender: TObject);

procedure Mosaico1Click(Sender: TObject);

procedure Ordenariconos1Click(Sender: TObject);

procedure Minimizartodas1Click(Sender: TObject);

procedure Abrirtodas1Click(Sender: TObject);

procedure Cerrartodas1Click(Sender: TObject);

procedure Salir1Click(Sender: TObject);

procedure Abrir1Click(Sender: TObject);

```
procedure Salvar1Click(Sender: TObject);

procedure BCopiarClick(Sender: TObject);

procedure BCortarClick(Sender: TObject);

procedure BPegarClick(Sender: TObject);

private

    { Private declarations }

public

    procedure ActualizaInterfaz; { Procedimiento que activa/desactiva opciones y botones
    según el estado actual del programa }

    { Public declarations }

end;

var

    FPrincipal: TFPrincipal;

implementation

{$R *.DFM}

    { Crear un nuevo archivo }

procedure TFPrincipal.Nuevo1Click(Sender: TObject);

Var

    Ficha: TFEditor; { Variable para crear una nueva ventana hija }

begin

    Ficha := TFEditor.Create(Self); { Crear la ventana }

    { Establecer un título }

    Ficha.Caption := 'SINOMBRE.TXT';
```

```
Ficha.Show; { y mostrarla }

{ Activar las opciones y botones oportunos }

ActualizaInterfaz;

end;

procedure TFPrincipal.Cascada1Click(Sender: TObject);

begin

    Cascade; { Disponer las ventanas en forma de cascada }

end;

procedure TFPrincipal.Mosaico1Click(Sender: TObject);

begin

    Tile; { Disponer las ventanas en forma de mosaico }

end;

procedure TFPrincipal.Ordenariconos1Click(Sender: TObject);

begin

    Arrangelcons; { Ordenar los iconos de las ventanas minimizadas }

end;

procedure TFPrincipal.Minimizartodas1Click(Sender: TObject);

Var

    N: Integer;

begin

    { Recorrer todas las ventanas }

    For N := 0 To MDIChildCount-1 Do

        { minimizándolas }
```

```
MDIChildren[N].WindowState := wsMinimized;

{ Minimizar también la ventana activa }

ActiveMDIChild.WindowState := wsMinimized;

end;


procedure TFPrincipal.Abrirtodas1Click(Sender: TObject);

Var

    N: Integer;

begin

    { Devolver todas las ventanas a su situación normal }

    For N := 0 To MDIChildCount-1 Do

        MDIChildren[N].WindowState := wsNormal;

end;


procedure TFPrincipal.Cerrartodas1Click(Sender: TObject);

Var

    N: Integer;

begin

    { Cerrar todas las ventanas }

    For N := 0 To MDIChildCount-1 Do

        MDIChildren[0].Close;

end;


procedure TFPrincipal.Salir1Click(Sender: TObject);

begin

    { Antes de salir cerrar las ventanas }

    Cerrartodas1Click(Self);
```



```
Close; { cerrar la ventana principal }

end;

{ Se ha elegido la opción Abrir }

procedure TFPrincipal.Abrir1Click(Sender: TObject);

Var

    Ficha: TFEditor; { Variable para crear la ficha }

begin

    If CDAbrir.Execute Then { Si se ha elegido un archivo }

        Begin

            { Creamos la ficha }

            Ficha := TFEditor.Create(Self);

            Ficha.Caption := CDAbrir.FileName; { Establecemos el título }

            Try

                { Intenta cargar el archivo }

                Ficha.CTexto.Lines.LoadFromFile(CDAbrir.FileName);

            Except On Exception Do { Si no es posible }

                ShowMessage('No es posible cargar ' + CDAbrir.FileName); { indicarlo }

            End;

            Ficha.CTexto.Modified := False;

            Ficha.Show; { Mostrar la ficha }

            { Mostrar el número de líneas existentes en el archivo }

            Ficha.LineaEstado.Sections[1]:=IntToStr(Ficha.CTexto.Lines.Count)+' líneas';

        End;

        { Activar las opciones y botones }

        ActualizaInterfaz;

    end;
```

```
{ Salvar el contenido de la ventana actual }

procedure TFPrincipal.Salvar1Click(Sender: TObject);

begin

    With ActiveMDIChild As TFEditor Do{ Asumir la referencia a la ventana actual }

        Begin

            { Inicialmente el nombre de archivo será el

                título de la ventana }

            CDSalvar.FileName := Caption;

            If CDSalvar.Execute Then { Si se elije un nombre de archivo }

                Try

                    CTexto.Lines.SaveToFile(CDSalvar.FileName); { Intentar salvar }

                    Caption := CDSalvar.FileName;

                    CTexto.Modified := False; { Eliminar el indicador de modificado }

                    LineaEstado.Sections[0] := '      ';

                Except On Exception Do { Si se produce un error }

                    ShowMessage('No es posible salvar en '+CDSalvar.FileName);{comunicarlo}

                End;

            End;

        end;

    procedure TFPrincipal.BCopiarClick(Sender: TObject);

    begin

        { Copiar el texto seleccionado al portapapeles }

        (ActiveMDIChild As TFEditor).CTexto.CopyToClipboard;

    end;

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

```
begin
    { Cortar el texto seleccionado }

    (ActiveMDIChild As TFEEditor).CTexto.CutToClipboard;

end;

procedure TFPrincipal.BPegarClick(Sender: TObject);

begin
    { Pegar el texto del portapapeles a la ventana }

    (ActiveMDIChild As TFEEditor).CTexto.PasteFromClipboard;

end;

{ Este procedimiento activa o desactiva opciones y botones
según que haya o no abierta alguna ventana }

Procedure TFPrincipal.ActualizaInterfaz;

Var

    Estado: Boolean;

Begin

    Estado := MDIChildCount <> 0; { True si hay alguna ventana abierta }

    Salvar1.Enabled := Estado; { Activar o desactivar la opción }

    BSalvar.Enabled := Estado;

    BCopiar.Visible := Estado; { Mostrar u ocultar los botones }

    BCortar.Visible := Estado;

    BPegar.Visible := Estado;

End;

end.
```

Mientras que EDITEXTO.PAS almacena el código específico de un tipo de ventana hija, en este caso el único que existe, cuya finalidad es facilitar la edición de texto.

```
{ Módulo de la ficha hija }

unit Editexto;
```

interface

uses

SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
Forms, Dialogs, Menus, StdCtrls, ExtCtrls;

type

TFEEditor = class(TForm)

MenuEditor: TMainMenu;

Editar1: TMenuItem;

Copiar1: TMenuItem;

Cortar1: TMenuItem;

CTexto: TMemo;

Pegar1: TMenuItem;

LineaEstado: THeader;

procedure FormClose(Sender: TObject; var Action: TCloseAction);

procedure CTextoChange(Sender: TObject);

procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);

procedure Copiar1Click(Sender: TObject);

procedure Cortar1Click(Sender: TObject);

procedure Pegar1Click(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

FEditor: TFEEditor;

implementation

Uses Principal; { Necesario para hacer referencia a la ficha principal. Se dispone aquí, y no en la cláusula Uses de Interface, para evitar una referencia circular }

{ \$R *.DFM }

procedure TFEditor.FormClose(Sender: TObject; var Action: TCloseAction);

begin

Free; { Destruir la ventana }

FPrincipal.ActualizaInterfaz; { y actualizar botones y opciones }

end;

{ Cada vez que se realice una modificación }

procedure TFEditor.CTextoChange(Sender: TObject);

begin

{ Mostrar la información adecuada en la línea de estado }

If CTexto.Modified Then

LineaEstado.Sections[0] := 'Modificado'; { indicarlo }

LineaEstado.Sections[1] := IntToStr(CTexto.Lines.Count) + ' líneas';

end;

{ Antes de cerrar la ficha }

procedure TFEditor.FormCloseQuery(Sender: TObject; var CanClose: Boolean);

Var

Respuesta: Word;

begin

If CTexto.Modified Then { Si el texto está modificado }

Begin

{ Preguntar si se desea salvar }

Respuesta:=MessageDlg('¿ Salvar ' + Caption + ' antes de cerrar?',mtConfirmation, [mbYes, mbNo, mbCancel], 0);

If Respuesta = mrYes Then

```
        { en caso afirmativo hacerlo }

        (Owner As TFPrincipal).Salvar1Click(Self)

    Else If Respuesta = mrCancel Then { Si se ha pulsado Cancelar }

        CanClose := False; { no cerrar la ventana }

    End;

end;

procedure TFEditor.Copiar1Click(Sender: TObject);

begin

    CTexto.CopyToClipboard; { Copiar el texto al portapapeles }

end;

procedure TFEditor.Cortar1Click(Sender: TObject);

begin

    CTexto.CutToClipboard; { Cortar el texto }

end;

procedure TFEditor.Pegar1Click(Sender: TObject);

begin

    CTexto.PasteFromClipboard; { Pegar texto desde el portapapeles }

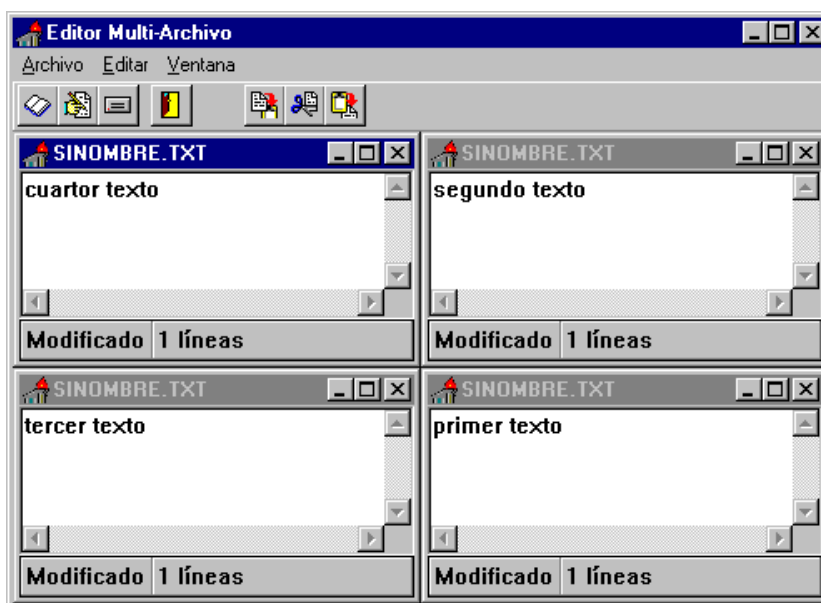
end;

end.
```

En la definición de la ficha principal hemos añadido, en la parte pública, el procedimiento **Actualizarterfaz**, cuya finalidad será la de activar o desactivar ciertas opciones y botones, dependiendo de que haya o no ventanas de edición abiertas en ese momento.

El resto del código está ampliamente comentado, y prácticamente se basa en llamadas a los métodos y referencias a las propiedades que hemos visto anteriormente.

El resultado de ejecutar la aplicación podría ser el siguiente:



ANEXO

1 Nueva Aplicación.

2 Creación del Formulario 'Padre'.

Propiedades:	FormStyle fsMDIForm
	• Generalmente ...
	Position poScreenCenter
	WindowState wsMaximized

3 Creación del Módulo de Datos.

Para cada Conjunto de Datos...	TTable,TQuery, TStoredProc
	TDataSource

4 Secuencia por cada Formulario Hijo ...

4.1 Nuevo Formulario.

Propiedades:	FormStyle fsMDIChild
	Generalmente ...

Position poScreenCenter
WindowState wsNormal

4.2 Cláusula USES del Formulario 'Padre'.

Incluir el nombre de la unidad 'Hijo'

4.3 Si la unidad 'Hijo' hace referencia y/o uso de algún Componente del Formulario 'Padre' u otro Formulario...

En la sección IMPLEMENTATION de la unidad 'Hijo' añadir una cláusula USES con el nombre de la Unidad 'Padre' y otras.

4.4 Determinar el procedimiento de creación del Formulario 'Hijo'.

Activar la opción: Project Manager del menú View (Ver).

Activar la opción: Options.

Si queremos que el Formulario 'Hijo' se cree automáticamente en tiempo de carga de la Aplicación ...

Dejarlo en la sección: Auto-create Forms.

Si queremos que el Formulario 'Hijo' sea creado en tiempo de ejecución, a partir de la ejecución de un 'suceso' provocado por el usuario o el programa...

Pasarlo a la sección: Available Forms.

En este segundo caso ...

En un evento provocaremos la creación del Formulario 'Hijo',

Application.CreateForm(TForm2,Form2);

y ...

Lo mostraremos:

Form2.Show; Ventana no Modal

Ojo !

Tener muy presente el modo en que queremos que se 'comporte' la ventana (formulario). Según hayamos definido el tipo de Ventana (determinado por las propiedades: **BorderStyle** y **FormStyle**).

Form2.ShowModal; Ventana Modal.