

Introducción general

En este capítulo veremos algunas generalidades de Delphi y su Entorno de Desarrollo que nos permitirán comenzar a explotar sus posibilidades. Tendremos también un primer contacto con los archivos generados por el sistema y veremos cuáles son indispensables para trasladar un programa a otra máquina. Por último, presentaremos algunos conceptos de Programación Orientada a Objetos que nos serán útiles para comprender y aprovechar las ventajas de Delphi.

¿Qué es Delphi?

Cuando comencé a programar en PC compatibles, las opciones de lenguajes de programación eran más o menos las siguientes: si uno buscaba facilidad de aprendizaje, tenía el BASIC. Para programar sistemas comerciales había versiones varias de COBOL. Cuando se manejaban cantidades de datos relativamente grandes (¡que sobrepasaban a veces varios diskettes de 360 kB!) se pensaba en programar DBASE III+. Cuando la velocidad y acceso a los recursos básicos del Sistema Operativo eran más importantes que la facilidad de programación, se disponía de los lenguajes C y Assembler. Entonces Borland dijo “hágase la luz”.

...Y nació Turbo Pascal.

Hasta entonces, el lenguaje Pascal -que había sido creado con fines académicos, para enseñar los fundamentos de la programación estructurada- era poco menos que una curiosidad universitaria. El producto de Borland ponía en las manos de los programadores un lenguaje poderoso, un compilador sumamente rápido y un Entorno Integrado de Desarrollo desde el que se podía editar el código, compilarlo e incluso ejecutarlo sin necesidad de utilizar programas externos.

De ahí en más el desarrollo del lenguaje y el entorno fue rápido. Siempre adaptándose a las nuevas tecnologías, incorporó los conceptos de Programación Orientada a Objetos en cuanto se vio que el nuevo paradigma había llegado para quedarse. Las extensiones realizadas al Pascal original seguían la filosofía del lenguaje, haciéndolas fáciles de incorporar para los programadores. A partir de la aparición de los objetos se conoce al lenguaje como Object Pascal.

Y llegamos a las versiones para Windows. Los primeros intentos fueron poco más que traducciones del entorno de desarrollo al nuevo Sistema Gráfico, prácticamente sin cambios en el lenguaje. No obstante, desde las primeras versiones para Windows, Borland ofrecía junto con el compilador una biblioteca de objetos muy bien construida -Object Windows Library, OWL- que hacía de capa intermedia entre la Application Programming Interface (API) de Windows y el programador. No obstante, todavía era necesario escribir muchas líneas de código y conocer bastante de Programación Orientada a Objetos.

Y entonces apareció Visual Basic de Microsoft, que revolucionó el mercado. A partir de ese momento, quedó claro que los entornos de desarrollo bajo Windows tenían que ser parecidos al de ese producto. Miles de programadores se pasaron a las filas de Visual Basic, y muchos más que nunca se habían atrevido a pensar en escribir programas para Windows comenzaron a desarrollar aplicaciones que tenían incluso cierto valor comercial.

Borland se unió a esta ola de programación visual y lanzó al mercado Delphi, basado en el lenguaje Pascal pero con un entorno de programación que hace más fácil el desarrollo de aplicaciones en entorno Windows. Entonces, ahora tenemos más opciones a la hora de programar en el entorno gráfico que se impone.

Enhorabuena. Surge ahora una nueva pregunta: ¿por qué Delphi?. La respuesta no es fácil. Podríamos decir que la herramienta adecuada para un trabajo es la que permite la consecución correcta y a tiempo del mismo. Pero además de estas condiciones mínimas indispensables, cada lenguaje tiene un “espacio” donde se desempeña mejor; por ejemplo, Clipper fue concebido para manejar bases de datos con formato dBase, y es ese el lugar en el que claramente se destaca. Delphi no tiene un lugar tan claramente definido; es lo que se llama un *lenguaje de propósito general*. Esto significa que se comporta bien ante tipos diferentes de problemas, y Borland ha puesto mucho empeño en lograr que el rendimiento sea el mejor posible.

Delphi se puede usar para casi cualquier tipo de programa, obteniéndose un rendimiento excelente con facilidad.

Destaquemos entonces algunas características de Delphi:

- ? Posee un Entorno de Desarrollo Integrado para Windows (IDE), con características de programación visual. Esto significa que la mayor parte del programa se hace gráficamente con el mouse o interactuando con los objetos en tiempo de diseño sin necesidad de compilar cada vez para ver los resultados, ahorrando tiempo y esfuerzo.
- ? El lenguaje Pascal ha sido ampliado y mejorado, sobre todo para adaptarlo a la programación con Objetos.
- ? La programación se hace más intuitiva y sencilla con el uso de componentes.
- ? Incluye un soporte de Bases de Datos poderoso y fácil de usar.

- ? Genera ejecutables nativos, sin necesidad de librerías de run-time (salvo el caso de los programas que usan Bases de Datos, para los que hay que instalar otro componente de libre distribución que viene con el paquete de Delphi).
- ? Se pueden crear componentes nuevos que se integran en el entorno de la misma manera que los nativos.

Versiones de Delphi

En el momento de escribir estas líneas, hay en el mercado cinco versiones del producto: Delphi 1 (también llamado Delphi 16) que compila código en 16 bits para Windows 3.1 y superiores, y los siguientes Delphi 2, 3, 4 y 5, que compilan en 32 bits para Windows 95 y NT. Además, de cada una de las versiones hay diferentes opciones que varían en los componentes y herramientas que traen y por supuesto... el precio. En general, cuando programemos para Windows 95 o NT aprovecharemos las ventajas extras que brindan estos sistemas (múltiples hilos de ejecución, modelo plano de memoria, etc) por lo que podemos decir que migrar un sistema a Windows 95 o NT no es sólo recompilar viejas aplicaciones (aunque puede hacerse para salir del paso) sino rescribir todo. Está planeado para mediados del año 2000 el lanzamiento de la primera versión de Delphi para el sistema operativo Linux.

Los programas de Delphi: proyectos

Un programa en Pascal está compuesto de un archivo principal (con el identificador **program**) y diferentes unidades (con el identificador **unit**). Esta estructura se mantiene, pero ahora se denomina **proyecto** al programa completo. El archivo que empieza con **program** sigue siendo el principal, el que se ejecuta en primer lugar; se puede ver con la opción de menú **View|Project Source**. Identificamos entonces los programas o aplicaciones como *proyectos*.

El entorno integrado de desarrollo (IDE)

El entorno de programación de Delphi incluye un cómodo editor de texto ASCII con posibilidad de trabajar con varios archivos a la vez, un visor de propiedades y eventos y un potente debugger. Además, vienen con el compilador una versión reducida del Paradox for Windows para trabajar con las tablas y un editor de recursos (bitmaps, cursores, iconos, etc). En la figura 1 podemos ver una imagen del IDE con las partes más importantes señaladas.

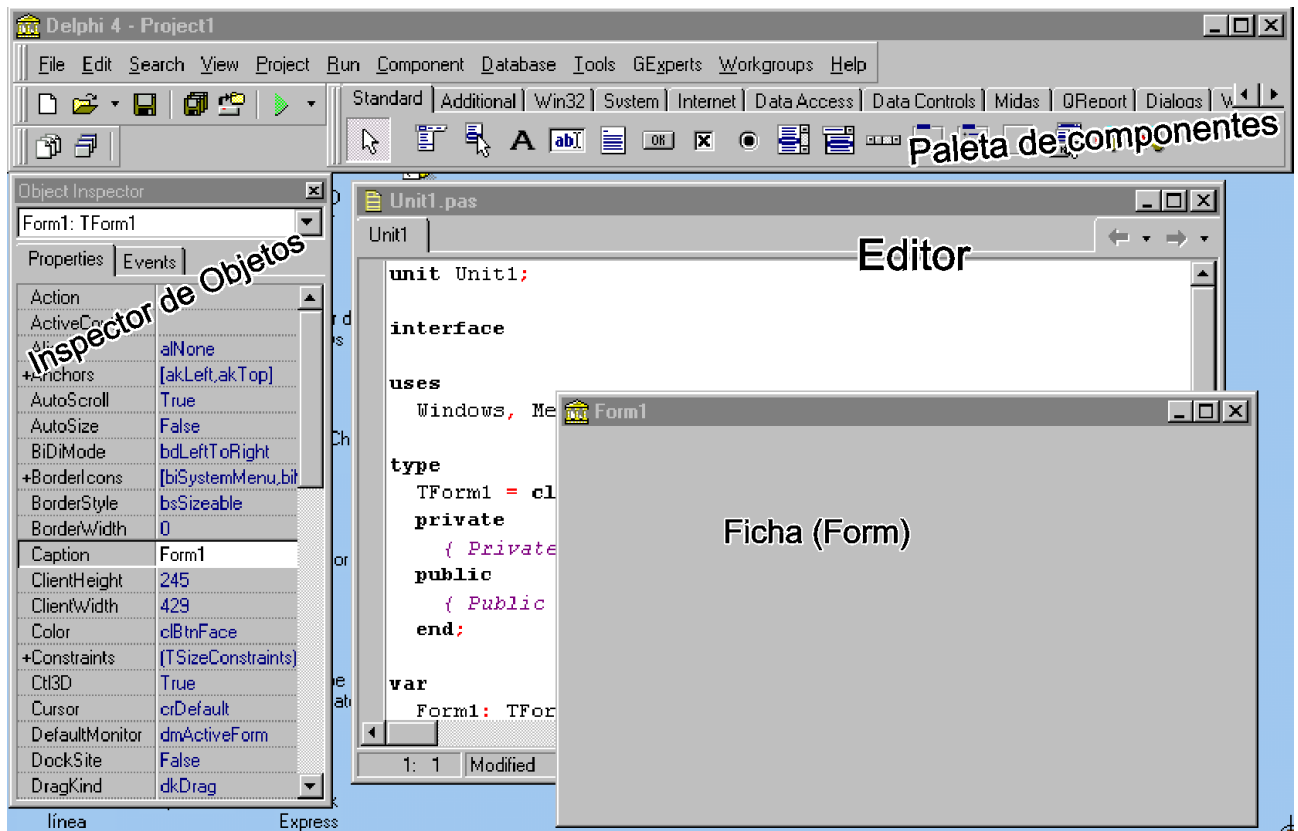


Figura 1: Entorno Integrado de Desarrollo (IDE) de Delphi 4

Explicaremos las diferentes partes que componen el IDE a medida que las utilizemos.

¡Manos a la tecla!

Si no lo ha hecho aún, arranque Delphi. Seleccione la opción de menú **File|New Project** para crear un nuevo proyecto. Luego seleccione **Run|Run** para compilar y correr el mínimo programa generado por defecto.

¡Felicitaciones! ¡Acaba de crear un programa en entorno Windows!

Veremos una ventana vacía que se puede redimensionar, minimizar, maximizar, cerrar con una doble pulsación en el ángulo superior izquierdo o la combinación de teclas **Alt+F4**, etc. Este es el comportamiento por defecto de las ventanas creadas por Delphi, que a partir de ahora llamaremos también **forms** o **fichas** (Fig. 2).

Al compilar un programa, Delphi *siempre* crea un archivo ejecutable .EXE (no existe más la compilación en memoria) y luego lo ejecuta aprovechando las capacidades de multitarea de Windows. Cuando está corriendo un proyecto, en la barra de título de Delphi aparece la palabra *running* y desaparece el Inspector de Objetos; al cerrar la nueva aplicación volvemos al IDE en modo normal.

NOTA: Se puede indicar al IDE que se minimize cuando corre un programa; seleccione la opción **Tools|Environment Options|Minimize on run**.



Figura 2: un form vacío

Cuando un programa está corriendo podemos editar el código fuente en el editor; no obstante, no podremos volver a compilar hasta que cerremos la aplicación previamente compilada. Además la ventana de la

aplicación aparece por defecto en la misma posición, lo que puede llevar a confusión. Para distinguir cuando estamos en diseño y cuando en ejecución, tenemos una serie de indicadores que veremos a continuación. En la Figura 3 vemos el IDE con el proyecto mínimo listo para compilar; no obstante, tenemos varios indicadores de que la aplicación no se está ejecutando:

- ? En la barra de título de la ventana principal se ve el nombre del proyecto y nada más
- ? Se ve el Inspector de Objetos
- ? El comando “Ejecutar” está activo, mientras que el de Pausa no lo está.

Normalmente se verán los puntos de la grilla de diseño en la ficha; aunque se puede configurar el IDE para que no aparezcan (Tools|Environment Options|Display Grid). También se puede pedir a Delphi que minimice el IDE y las ventanas de diseño cada vez que se ejecuta un proyecto: en la misma página de configuración, las opciones **Minimize on run** y **Hide Designers on run**.

En la figura 3 se ve una imagen del IDE con un proyecto cargado, antes de ejecutarlo.

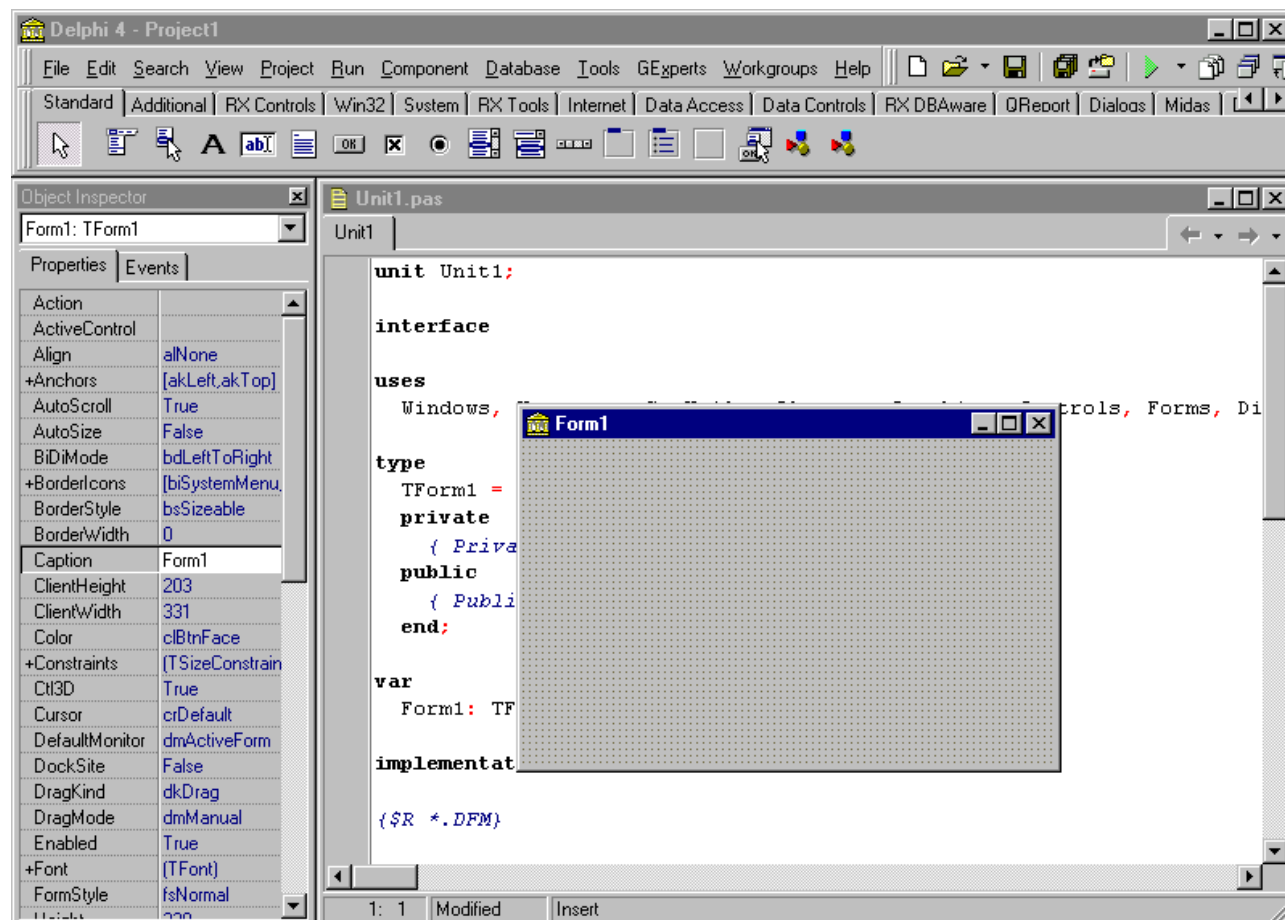


Figura 3 : Indicaciones en el IDE de que un proyecto no está corriendo

Cuando ejecutamos la aplicación, cambian algunas cosas:

- ? En la barra de título dice “(Running)” (corriendo). Este es el indicador principal que nos dice que la aplicación está activa.
- ? El Inspector de Objetos no se ve¹.
- ? El comando “Pause” está activo, mientras que el de ejecutar no lo está.

Notemos (Fig. 4) que la ficha de la aplicación aparece en el mismo lugar en que la dejamos cuando estábamos en el IDE; esto dificulta un poco el distinguir los dos estados.

¹ En Delphi 4 se puede mostrar un *Inspector de Objetos de tiempo de ejecución*, que nos permitirá mirar y cambiar los valores de las propiedades de los componentes fácilmente con fines de depuración del código.

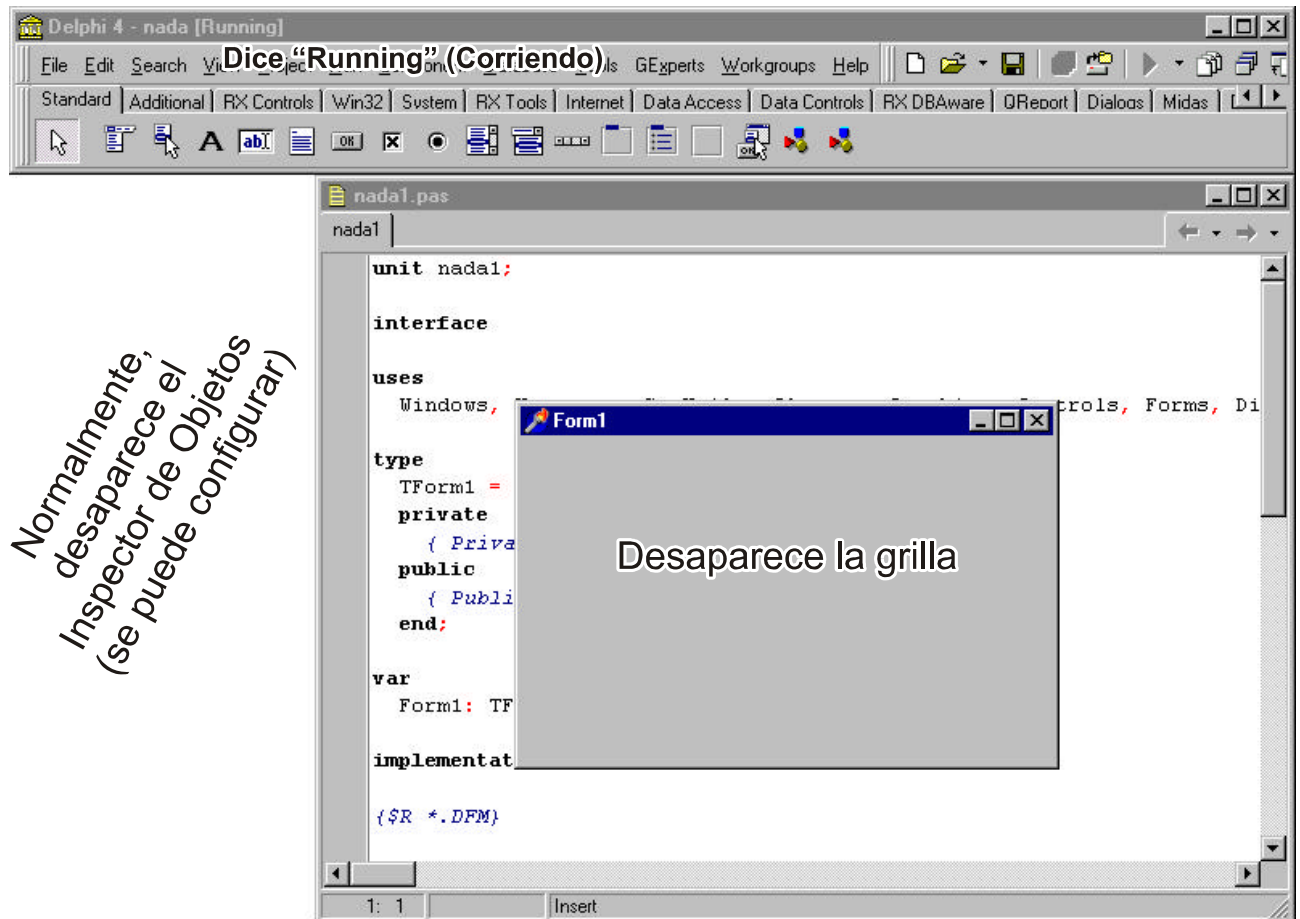


Figura 4: el IDE cuando ejecutamos un proyecto

? Pruébalo YA!

¿Por qué es tan importante distinguir cuando un programa está ejecutándose? Para entenderlo, corremos el programa y después con **Alt+Tab** pasamos a Delphi (notemos el “Running” en la barra de título). Podemos escribir en el editor, modificando el código. Pero recordemos que Pascal es un lenguaje *compilado*; para reconocer nuestros cambios, hay que recompilar el programa. *Y esto no se puede hacer mientras se está ejecutando la versión anterior!* De hecho, ni siquiera estará disponible el comando **Ejecutar**.

Una vez que está corriendo, podemos detener la aplicación para inspeccionar (y eventualmente modificar) el valor de las propiedades o variables en tiempo de ejecución. Para ello tenemos el comando **Run|Program Pause** que provoca una detención inmediata de la aplicación, volviendo al IDE. En este momento podemos pedir a Delphi que nos muestre el valor actual de las variables con **View|Watches** o podemos también ver y *modificar* el valor de algún símbolo con **Run|Evaluate/Modify**. En la figura 5 vemos un ejemplo de inspeccionar variables con la aplicación detenida. La línea roja que se ve en el código indica que se ha colocado un *Punto de ruptura de secuencia (Breakpoint)*. Cuando el programa se está ejecutando y llega a esta línea, se detiene y vuelve a Delphi para que podamos inspeccionar las variables.

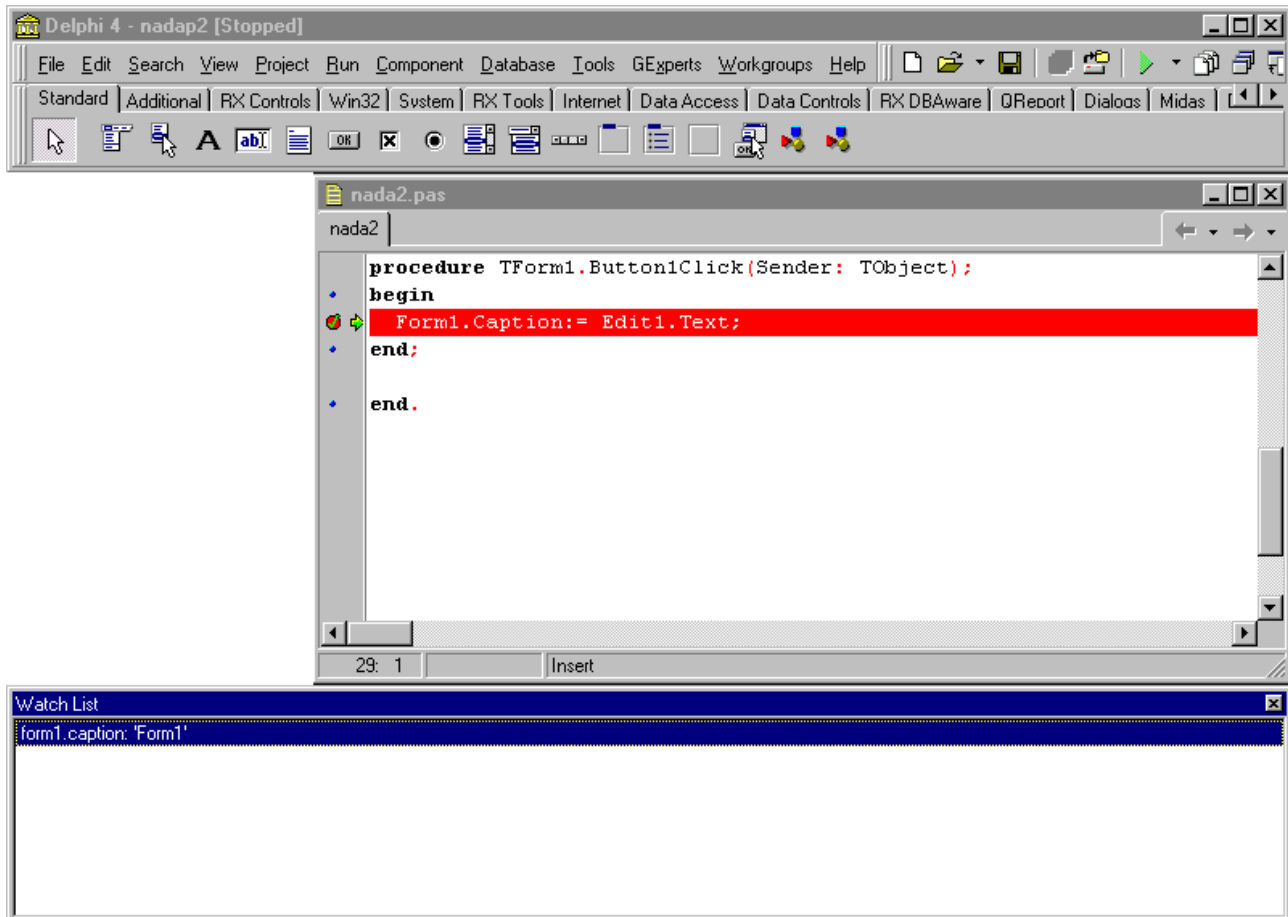


Figura 5: Un programa detenido para ver los valores de las variables

Después de ver los valores de las variables que nos interesaban, debemos continuar con la ejecución del programa; otra vez el comando **Run**. *Esto es muy importante*, ya que el sistema queda detenido de una manera especial cada vez que inspeccionamos variables. Una vez que el programa está activo nuevamente, lo cerramos de la manera habitual.

Un programa Windows puede tener varias ventanas que se activan en diferentes momentos; pero siempre habrá una ventana que identifica al programa -es la que cerramos para terminar la aplicación. Esta ventana se denomina **ventana o form principal**. En Delphi, por defecto el primer form que se crea se toma como principal, el caso de la ventana única que teníamos en el ejercicio anterior. Por ese motivo, al cerrar esta ventana se termina la aplicación.

Archivos generados por Delphi

Los programas grandes generalmente se dividen en módulos para hacer más manejable el código. En Delphi, esto se hace automáticamente. Cada ventana que declaramos tiene asociado un bloque de código Pascal, que se almacena en un archivo separado. Estos archivos se denominan *unidades* (Unit).

Las unidades no son otra cosa que archivos de texto con código Pascal. La gran ventaja es que tenemos separadas las partes -si tengo que corregir algo en la segunda ventana, sólo tengo que abrir el archivo de la Unit correspondiente.

Veamos un poco los archivos generados por Delphi para el programa anterior:

La ficha está asociada con una **unit**, que fue escrita automáticamente y puede accederse trayendo al frente el editor (**View|Units...**). En el listado 1 vemos el código generado para la ficha inicial.

NOTA: Se puede cambiar el foco entre una ficha y su código asociado presionando F12

En este archivo vemos la definición de una *clase* (los conceptos de Programación Orientada a Objetos serán discutidos pronto; por ahora digamos que una clase es un tipo de datos parecido a un registro sólo que puede contener además de campos de datos, procedimientos y funciones que trabajen con ellos) llamada **tForm1** descendiente de la clase **tForm**, que no agrega nada a lo heredado de ésta. Por lo tanto, el comportamiento y las propiedades de la ventana que vemos están determinadas en la clase **tForm**.

¿Esto es todo? En realidad, no. Necesitamos también un programa principal que comience con la palabra clave **program**, como en el lenguaje Pascal tradicional. Este archivo también es generado por Delphi y se denomina **Fuente del proyecto** (project source). Se puede abrir en el editor con la opción de menú **Project|View source** (listado 2).

Veamos alguna información que nos presenta este archivo:

- ? El programa generado se llamará **project1.exe**, dado que este es el nombre del proyecto (el nombre que sigue al identificador **program**)
- ? Utiliza dos units por lo menos: **Forms** (interna de Delphi, contiene las definiciones básicas) y **Unit1**, en la que se define una ventana llamada **Form1** (nuestra ficha principal).
- ? Existe algún objeto llamado **Application**, que contiene un procedimiento para crear la ficha (*Create Form*) y otro para correr el programa (*Run*).
- ? La ficha principal es **Form1**, porque es la primera que se crea (en este caso es también la única).

Grabación del proyecto. Archivos intervinientes

Los nombres del programa y de las units se asignan por defecto hasta la hora de grabar por primera vez el proyecto. En ese momento si no hemos dado un nombre a los archivos componentes se nos preguntará por cada uno.

NOTA: Conviene pedir a Delphi que grabe automáticamente los archivos que modifiquemos cada vez que corremos el programa -el sistema puede “colgarse” y perderemos los cambios desde la última vez que grabamos. Para hacerlo, entramos al menú **Tools|Environment Options** y seleccionamos “Editor Files” de la sección *Auto Save*.

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

end.
```

Listado 1: la unidad donde se declara la ficha (form) principal de la aplicación.

```
program Project1;

uses
  Forms,
  Unit1 in 'UNIT1.PAS' {Form1};

{$R *.RES}

begin
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Listado 2: programa principal

Una vez grabado el proyecto, tendremos varios archivos en el disco. Los que **no** se pueden recuperar en caso

de pérdida de información son:

- | | |
|-----------------------------|--|
| ? <Nombre del proyecto>.dpr | Listado principal del proyecto (el que contiene la cláusula <i>program</i>) |
| ? <nombre unit>.pas | Código fuente de la unit. Uno por cada unit utilizada. |
| ? <nombre unit>.dfm | Imagen binaria de un form. Uno por cada form. |

Estos tipos de archivos son los más importantes, ya que sin ellos el programa no puede compilar.

NOTA: Para trasladar el proyecto a otro equipo o directorio y seguir trabajando con Delphi, debemos llevar los archivos que tengan las extensiones **dpr**, **pas** y **dfm**.

Cambiar el nombre de un proyecto

Después de grabar el proyecto por primera vez, los archivos fuente del programa principal y de las units tienen existencias independientes. Algunos casos comunes con que nos encontramos al querer cambiar el nombre de alguna parte del proyecto son los siguientes:

- ? Cambio de nombre de una unit, que sigue perteneciendo al proyecto.
Seleccionar la opción de menú **File|Save File As...** No renombre el archivo **.pas** manualmente porque el proyecto perdería el rastro y mostraría un error de archivo no encontrado.
- ? Cambio de nombre del proyecto, utilizando las mismas units.
Seleccionar la opción de menú **File|Save Project As...** Es válida la consideración anterior sobre el renombrado directo del archivo **.dpr**.
- ? Cambio de nombre del proyecto y las units, obteniendo una copia del programa.
Hay que grabar por un lado las units una por una (**Save File As...**) y el proyecto (**Save Project As...**) *en ese orden*.

NOTA: siempre se pueden conocer (y cambiar) las especificaciones de los archivos que intervienen en un proyecto editando el archivo fuente del mismo (**Project|View Source**)

El archivo de proyecto guarda información de la estructura de directorios en la que hay que buscar las units, por lo que no es válido copiar directamente todos los archivos a otro directorio. No obstante se puede hacer, teniendo cuidado de indicar luego al proyecto dónde encontrar los archivos. Veremos esto luego al tratar el Gestor de Proyectos (Project Manager).

Una vez que dimos nombre a los archivos correspondientes al proyecto, la grabación automática utilizará los mismos cada vez que solicitemos grabar.

Para comprender mejor los programas creados en Delphi debemos conocer algunos conceptos de Programación Orientada a Objetos; discutiremos los más comunes enseguida y dejaremos para después una discusión más profunda del tema.

Introducción a la Programación Orientada a Objetos (OOP)

La Programación Orientada a Objetos es un paso más de estructuración del código que tiene algunas ventajas conceptuales y de programación y se ha transformado en la “niña mimada” de los medios informáticos.

En cuanto a la implementación en Delphi, Borland ha modificado el lenguaje Pascal añadiéndole sentencias para definir y trabajar los objetos. En esta versión se agregan más comandos que en la versión 7.0, que nos dan nuevas posibilidades y facilidades a la hora de diseñar y codificar.

Al programar en Delphi *utilizamos* objetos, que no es lo mismo que *programar orientado a objetos*. Claro que también se puede hacer de esa manera, pero no es mandatorio.

Veamos algunas definiciones fundamentales.

? **Clase:** la clase describe los campos de datos y el código que actúa sobre los datos. Es sólo la declaración del tipo.

Ejemplo: en el ejercicio anterior creamos una clase **tForm1**, descendiente de la clase **tForm**, que no agrega nada a la base:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Listado 7: Declaración de una clase descendiente de TForm

? **Instancia:** es una materialización de la clase. Por ejemplo, podemos decir que una clase de tipo *reloj* casio modelo CS-2000 tiene como características que indica la hora en forma digital, tiene cronómetro, calendario, alarmas, etc. Cada uno de los relojes de este tipo (el mío, el de un vecino, etc.) es una instancia de esta clase. Se pueden tener muchas instancias diferentes pero con el mismo comportamiento, definido por la clase. En un lenguaje de programación, para usar una clase debemos crear una instancia de la misma, a la que se accede generalmente con una variable del tipo correspondiente.

Ejemplo: para acceder a la clase **tForm1**, Delphi declara una variable **Form1**:

```
var
  Form1: TForm1;
```

En el programa utilizaremos la *variable* **Form1** para acceder a una *instancia* de **tForm1**.

? **Objeto:** lo mismo que una instancia. Aquí debemos hacer una aclaración porque en el Pascal anterior -hasta la versión 7.0- se declaraba una clase con la palabra reservada **object**, lo que puede llevar a confusiones. Ahora en Delphi existe la palabra reservada **class** para definir las clases. Todavía se puede usar la declaración como **object** para conservar compatibilidad con los programas ya escritos, pero perdemos algunas características del nuevo modelo que son muy útiles en general e indispensables a la hora de crear componentes.

Todos los objetos en Delphi son creados en forma dinámica. Esto significa que no es suficiente con declarar la variable (que en realidad es un puntero) sino que hay que crear el objeto en memoria y apuntarlo con ella. En el programa anterior esto se hace en el método **Application.CreateForm**. A partir de esta llamada ya podemos acceder a nuestro objeto usando la variable **Form1**. Si tratamos de usarla antes se generará un

error porque el puntero **Form1** no tiene un valor asignado.

? **Propiedades:** son variables internas del objeto que definen *características* del mismo. En la práctica son variables que pueden ser de cualquier tipo definido en Pascal (incluso clases). La diferencia con las variables comunes es que las propiedades sólo pueden ser accedidas a través del objeto al que corresponden, lo que también se conoce como *encapsulamiento*.

Ejemplo: En la clase `tControl` se define una propiedad llamada *caption*, de tipo `string`, para contener el texto que aparece en los controles; por ejemplo, en los forms especifica el texto que aparece en la barra de título.

```
property Caption: string;
```

Esta propiedad, como muchas otras, se puede modificar directamente en tiempo de diseño mediante el Inspector de Objetos. No obstante, hay propiedades que no se ven en el Inspector de Objetos: sólo se pueden acceder por programa en tiempo de ejecución y se ven en la ayuda como *run-time properties*.

Para acceder a una propiedad de una instancia desde programa usamos un punto después del nombre de la variable. Por ejemplo, para asignar un valor a la propiedad *caption* de **form1**, haríamos

```
form1.caption:= 'alguna cadena';
```

Las propiedades son <i>variables</i> comunes de Pascal, internas al objeto.

? **Métodos:** son procedimientos internos de los objetos, que definen el *comportamiento* de éstos. Al igual que las propiedades, sólo se pueden acceder a través del objeto al que pertenecen. En la práctica son **procedimientos** y **funciones** comunes de Pascal, que actúan sobre los datos del objeto -las propiedades.

Ejemplo: en la clase `tWinControl` se define un método para actualizar la imagen en pantalla llamando a la función **UpdateWindow** de la API de Windows. Este método no necesita parámetros.

```
procedure Update;
```

por ejemplo, si queremos que el **form1** se redibuje inmediatamente, hacemos

```
form1.update;
```

? **Alcance o scope:** es el entorno en el cual vemos o tenemos acceso a las propiedades y métodos de un objeto. Vienen definidos por palabras reservadas como **public**, **private**, **published** y **protected**. El alcance determinará que podamos o no acceder a determinados métodos o propiedades desde otros objetos o unidades, o en tiempo de diseño a través del Inspector de Objetos. Hablaremos más sobre los controladores de acceso en el capítulo dedicado a la Programación Orientada a Objetos.

? **Mensaje:** el hecho de ejecutar un método de un objeto se conoce como *enviar un mensaje* al mismo. Por lo tanto, si decimos que le enviamos un mensaje al objeto para que se pinte a si mismo en la pantalla, en realidad estamos llamando al método de dibujar del objeto. El mensaje define **qué** es lo que queremos que el objeto haga, y el método **cómo** lo hace.

Hay que distinguir el concepto *mensaje* de la Programación Orientada a Objetos de los mensajes del Sistema Operativo. En Windows se puede hablar de mensajes del sistema a las aplicaciones o ventanas, que son verdaderos mensajes comunicando algún evento especial -por ejemplo que Windows se va a cerrar, permitiendo a las aplicaciones cerrar sus archivos y limpiar la memoria; o simplemente consultando a las ventanas abiertas por su título, para encontrar una determinada.

El significado del término debería resultar claro del contexto en que se utilice; si hablamos de mensajes entre objetos estamos empleando la definición teórica de la Programación Orientada a Objetos, en cambio si hablamos de mensajes de Windows son notificaciones de que ocurrió algún evento en el sistema.

? **Evento:** un concepto que está muy ligado a la Programación Orientada a Objetos es el de **evento**. Los eventos son hechos, cosas que suceden, como por ejemplo la pulsación de una tecla o el movimiento del ratón. Los programas **orientados a objetos** son por lo general **manejados por eventos**, es decir que los objetos hacen algo sólo en respuesta a eventos.

El sistema operativo Windows captura los eventos que ocurren en el sistema, pone los datos relevantes en

una estructura propia y envía mensajes a las aplicaciones indicando la ocurrencia del evento. Las aplicaciones escritas con Delphi capturan los mensajes del sistema y generalmente los traducen en llamadas a procedimientos especialmente definidos para contestarlos. Llamaremos a estos procedimientos **de respuesta a eventos**, y puede ser escritos por nosotros. De hecho, esta es la forma normal de especificar un comportamiento a nuestro programa -mediante respuestas propias a los eventos del sistema.

? **Componentes:** La programación en Delphi se hace mayormente colocando objetos en forms y relacionándolos unos con otros. Borland ha definido ya una cantidad de clases que modelan los controles usuales de Windows, como los botones o barras de desplazamiento; todas estas clases componen la **Visual Component Library (VCL)** que podríamos traducir como **Biblioteca de controles visuales**. La mayoría de las clases que utilizamos normalmente sobre los forms descienden de *tComponent*, y llamamos a los objetos **componentes**. Por lo tanto éste es un término que referencia una clase perteneciente a la VCL, descendiendo de *tComponent*.

Después de este pequeño paseo por los conceptos de la programación con objetos, volvamos al camino inicial para poner un poco de vida a nuestra aplicación.

La paleta de componentes

Los componentes están situados en la **paleta de componentes**, que tiene el siguiente aspecto²:



Para colocar un componente sobre una ficha, simplemente presionamos con el puntero sobre el botón que lo representa (que se “hunde”, indicando que está seleccionado) y luego el lugar de la ficha donde queremos situar la esquina superior izquierda (figura 7). Si efectuamos una pulsación simple, el componente se crea del tamaño por defecto; podemos indicar un tamaño propio al componente con el ratón al momento de colocarlo arrastrando el puntero sin soltar el botón hasta que llegamos a la esquina inferior derecha.

² Esta paleta de componentes está modificada con respecto a la original

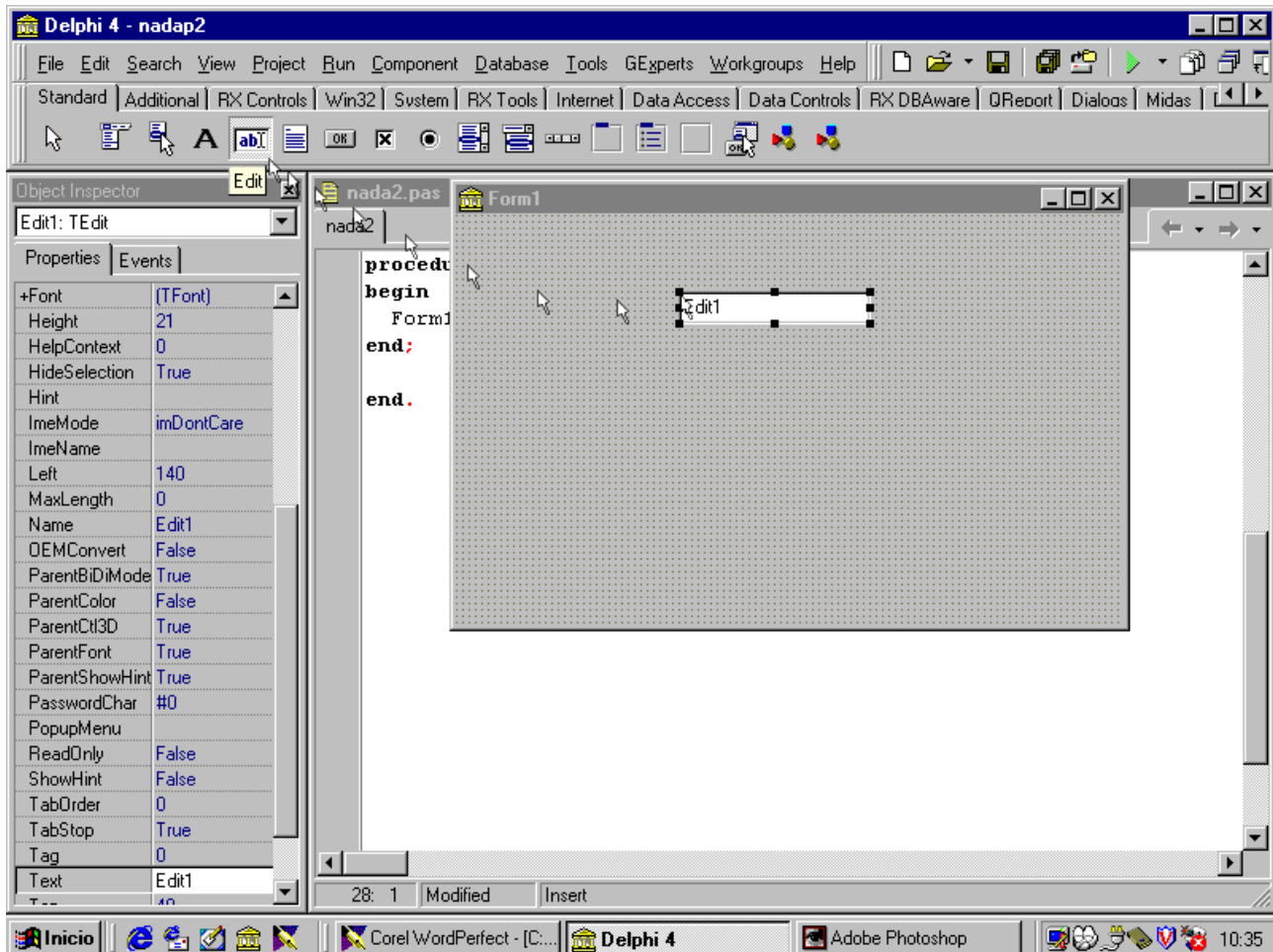


Figura 7: colocar un componente

Una vez colocado, si queremos cambiar las características de algún componente hay que seleccionarlo presionando el ratón sobre él. Aparecerán una serie de cuadrados negros alrededor (los manejadores), que sirven para modificar las dimensiones del componente arrastrándolos; las propiedades del componente se ven en el Inspector de Objetos.

Modificar el aspecto de los componentes: las propiedades

Las propiedades de un objeto especifican su estado interno, del cual podemos conocer algo mediante el **aspecto** exterior del componente y a través de su comportamiento. En el IDE de Delphi se pueden cambiar directamente algunas de las propiedades de los componentes, y ver el cambio inmediatamente sin necesidad de compilar el programa.

La herramienta para trabajar con las propiedades en diseño es el *Inspector de Objetos*. En él vemos dos páginas: **Properties** (propiedades) y **Events** (eventos). Figura ???.

Seleccionando la página **Properties** vemos las propiedades modificables directamente del componente que está seleccionado; en la página **Events** podemos asignar *procedimientos de respuesta a eventos*.

Por ejemplo, en la ficha de nuestra aplicación anterior tenemos la propiedad **Caption**, que indica el título de la ventana. Al cambiar su valor -es un string- cambia automáticamente el texto de la barra superior de la ficha.

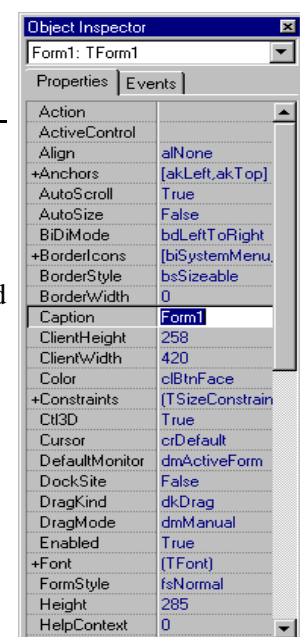


Figura 8: el Inspector de Objetos

? Pruébalo YA!

Modifique el valor de la propiedad *Caption*. Ves los cambios en la barra de título de la ficha. Trate de modificar de la misma manera otra propiedad, como por ejemplo *BorderStyle*. ¿Qué sucede cuando se presiona [Enter]? En la siguiente sección veremos los distintos tipos de propiedades y la forma de editarlas.

Diferentes tipos de propiedades. Los editores.

Delphi nos deja interactuar con los valores de las propiedades en el Inspector de Objetos de diferente manera según el tipo de propiedad.

Unas palabras de precaución: muchos componentes tienen las mismas propiedades y eventos; por ejemplo, el Form y un botón tienen ambos la propiedad **Caption** (con el mismo significado, el texto que se muestra sobre el componente). Entonces, ¿cómo sabemos si estamos modificando el texto de uno o del otro?

Notemos que el Inspector de Objetos nos muestra las propiedades y eventos *de un solo componente a la vez*: el que está seleccionado. Podemos distinguirlo por las marcas de selección -pequeños cuadrados negros- sobre el borde del componente, salvo para el caso del Form. También hay otro indicador del componente que está seleccionado: en la parte superior del Inspector de Objetos tenemos una lista (ComboBox) que nos mostrará el nombre del componente seleccionado en cada momento.

Ahora sí, pasemos a ver los diferentes tipos de propiedades en detalle.

? *Propiedades simples*

En el caso más simple como el del **Caption**, la edición del valor se hace directamente en la columna de la derecha del Inspector de Objetos, y podemos poner cualquier valor que concuerde con el tipo de dato (figura 9).

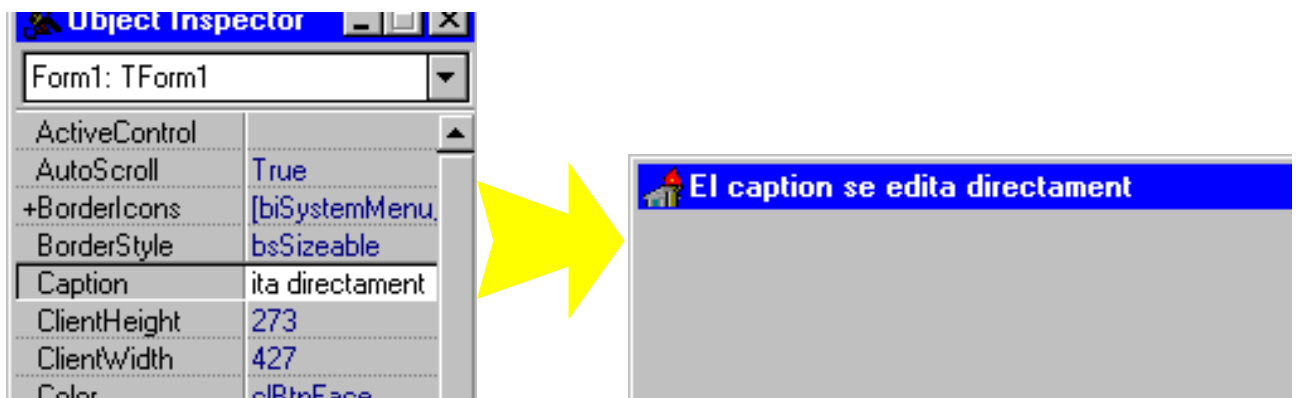


Figura 9: al escribir un nuevo caption, se ve inmediatamente en el form

? *Propiedades enumeradas*

Son variables que pueden tomar un valor de un conjunto finito de posibilidades (un tipo enumerado). Los valores posibles se despliegan en una lista en la columna de la derecha del Inspector de Objetos. Un caso por ejemplo es la propiedad **FormStyle** de los forms (figura 11).

NOTA: haciendo *double click* en la columna de la derecha (donde va el valor) se selecciona el siguiente valor posible.

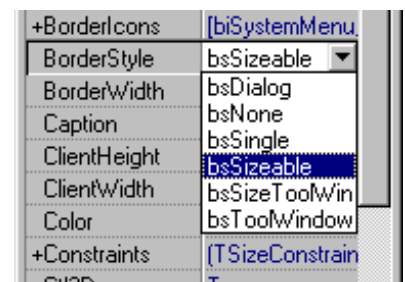


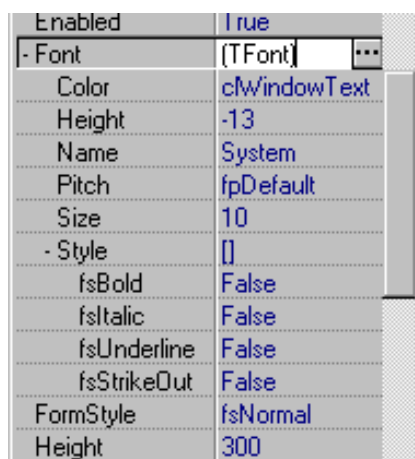
Figura 10: una propiedad enumerada

Algunas propiedades que ofrecen las dos posibilidades: elegir el valor de la lista o escribirlo directamente (siempre y cuando lo que escribimos concuerde con uno de los valores

posibles). Ejemplo: la propiedad **visible** de cualquier control, que puede ser **true** o **false**. Podemos elegir el valor en la lista, hacer doble click para cambiar al otro o bien escribir la palabra directamente.

? *Propiedades de conjunto*

Se presentan al usuario como un árbol de opciones, indicado por un signo + a la izquierda del nombre de la propiedad. Haciendo doble click sobre este nombre se despliega el árbol mostrando las subpropiedades (ramas) que son las que toman valores particulares. Para cada una de las ramas debemos especificar un valor. Un ejemplo es la propiedad **Font** de los componentes visuales (y dentro de ésta también la propiedad **style**).



Enabled	True
- Font	(TFont) ...
Color	cWindowText
Height	-13
Name	System
Pitch	fpDefault
Size	10
- Style	[]
fsBold	False
fsItalic	False
fsUnderline	False
fsStrikeOut	False
FormStyle	fsNormal
Height	300

Figura 11: dos propiedades de conjunto -Font y Style.

? *Propiedades especiales*

Estas son propiedades que necesitan un editor especial para darles valor. En el Inspector de Objetos se reconocen por mostrar a la extrema derecha de la columna de valores un botón con tres puntos que abre el editor para la propiedad. Por ejemplo, la propiedad **Font** que mostramos en la figura 11 se puede también modificar con un editor especial -la conocida caja de diálogo de seleccionar fuente- que se abre al presionar el botón con tres puntos (o haciendo doble click).

Modificar el comportamiento de los componentes: los eventos

Recalquemos la diferencia entre los programas tradicionales y los manejados por eventos, ya que es un concepto fundamental. En los primeros, el programa seguía un hilo de ejecución generalmente regido por un bucle de lectura del estado de los periféricos (teclado, ratón, etc) que llamaba a los procedimientos correspondientes a cada suceso o evento. En Windows esta tarea la realiza el sistema, y nos informa cada vez que se produce algún suceso que incumbe a nuestra aplicación por medio de un *mensaje*. Delphi captura estos mensajes del sistema y los transforma en *Eventos*. Nosotros debemos estar listos a responder en cualquier momento a los eventos que nos interesan, y programamos únicamente estas respuestas.

Los eventos son, como ya vimos, cosas que suceden. En un programa manejado de esta manera, la actividad de los componentes se da en respuesta a eventos. Por ejemplo, es común colocar botones en las ventanas para dar al usuario una forma de indicar al programa que desea que se haga algo. Al presionar el botón, el programa recibe una notificación de que ha sucedido tal evento. El modelo de objetos utilizado en Delphi nos brinda una forma de indicar la respuesta a tales sucesos a través del proceso de **delegación**.

El programador de una aplicación escribe rutinas que deben ejecutarse en respuesta a los eventos, y lo indica a Delphi a través del Inspector de Objetos. Cuando se produce el evento, el componente mira si hay algún procedimiento designado para actuar en respuesta y si es así lo ejecuta, delegándole la responsabilidad; caso

contrario, se actúa de la manera prevista por defecto.

Las rutinas de respuesta a eventos son procedimientos comunes de Pascal, pero con parámetros determinados por el evento. Así por ejemplo, todos los eventos incorporan un parámetro llamado *Sender* de tipo **TObject** -la clase antecesora de todas- que cuando se invoca el procedimiento apunta al componente que produjo el evento. Si el evento es la pulsación del botón llamado **button1**, *Sender* estará apuntando a **button1** y el resultado de la expresión *sender=button1* es verdadero.

Para asignar un procedimiento a un evento, escribimos el nombre del procedimiento en la columna de la derecha y hacemos [Enter]. Delphi escribe la cabecera y coloca el cursor en el cuerpo.

Nuevamente Delphi tiene definido un mecanismo para ahorrar trabajo; para crear un procedimiento que responda a un evento basta con hacer doble clic en la columna de la derecha del Inspector de Objetos, en la página de eventos. Delphi escribirá la cabecera de un procedimiento con los parámetros correctos utilizando un nombre generado automáticamente, colocará las palabras reservadas **begin...end** y dejará el cursor en el medio, listo para agregar nuestro código de manejo del evento³. Una vez asignado, bastará con hacer doble clic nuevamente en el Inspector de Objetos para colocar el cursor al principio del procedimiento correspondiente.

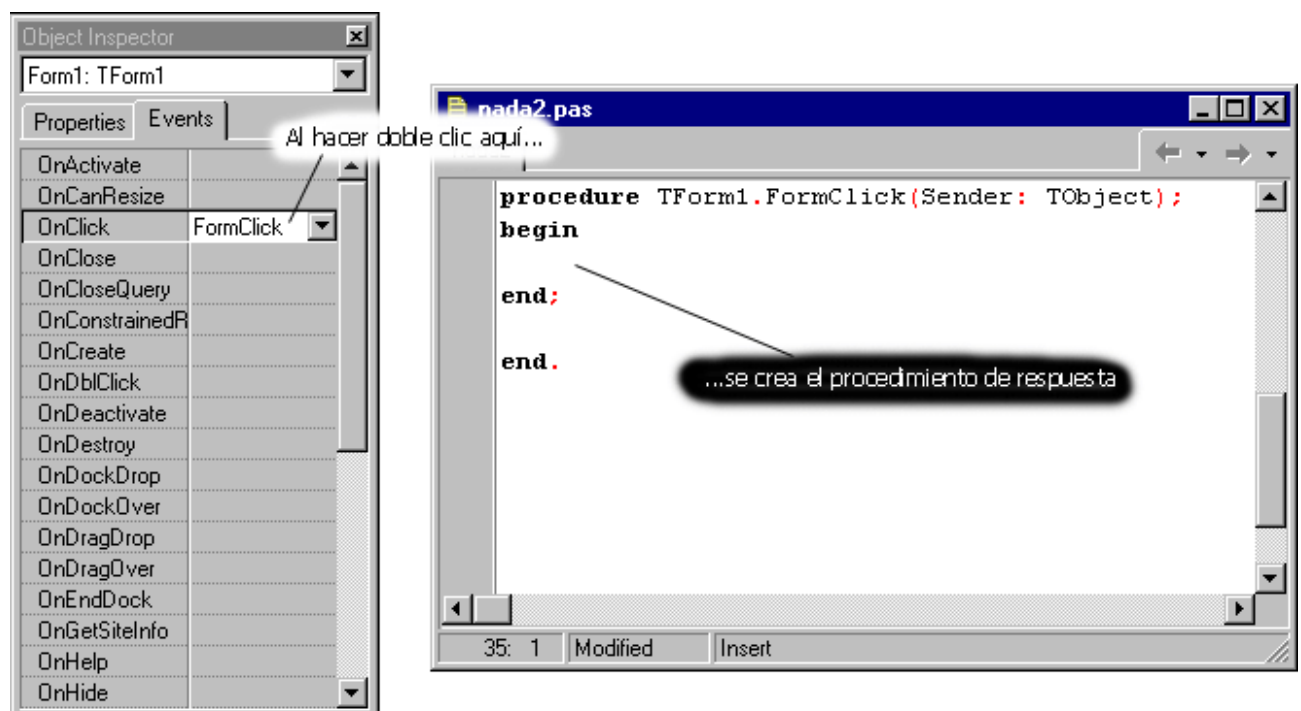


Figura 12: Crear un procedimiento de respuesta a un evento

Cuando tenemos que responder a eventos similares con las mismas actividades, podemos asignar a todos ellos un mismo procedimiento. Para esto, en lugar de escribir el nombre de un procedimiento nuevo en cada evento desplegamos la lista de métodos disponibles presionando la flecha que aparece a la derecha `OnClick` `FormClick` y seleccionamos el procedimiento deseado. Los procedimientos deben ser compatibles, con los mismos parámetros; caso contrario, no aparecerán en la lista y cualquier intento de enlazarlos provocará un error del compilador.

Veamos un par de ejemplos simples.

Ejemplo 1: Cerrar la aplicación con un botón

El objetivo es agregar a nuestro form vacío un botón e indicar que cuando se presione se cierre la aplicación.

- ? Seleccionar **File|New project** para crear un proyecto en blanco.
- ? Agregar un botón y cambiar su **Caption** a "Cerrar". Esto cambia el texto del botón (fig. 14).
- ? Crear un procedimiento de respuesta al evento **OnClick** del botón, escribiendo sólo la palabra **Close**. Con esto estamos llamando al método *Close* del form, que efectivamente cierra la ventana.

³ Como nos estás malcriando, Borland...

Dado que es la ventana principal de la aplicación, al cerrarla se termina la aplicación. Notemos que no es necesario escribir "Form1.Close" dado que el procedimiento pertenece al objeto **form1** y por lo tanto tiene acceso a todos los métodos y propiedades del mismo, tal como el método **Close**.

? Correr el programa y comprobar que el botón termina la aplicación correctamente.

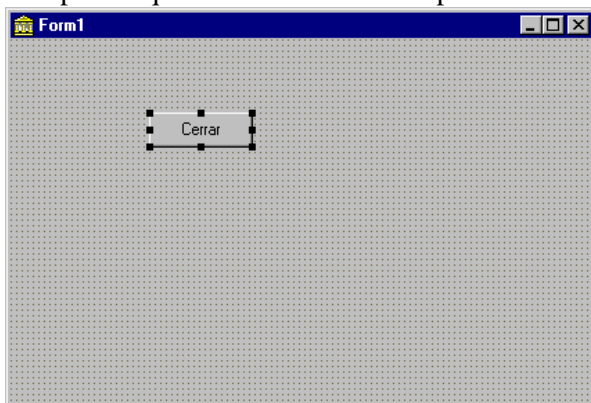


Figura 14: la ficha del ejemplo 1

Ejemplo 2: cambiar el título del form escribiendo en un editor

El objetivo es poder cambiar el texto de la barra superior de la ficha escribiéndolo directamente en un editor.

? En la misma aplicación anterior, poner sobre la ficha un editor.

? Crear un procedimiento de respuesta al evento **OnChange** del editor⁴, en el cual asignaremos la propiedad **text** del mismo a la propiedad **Caption** de la ficha:

```
form1.caption:= edit1.text;
```

NOTA: igual que en el ejemplo anterior, se puede eliminar la referencia a **Form1** delante de la propiedad **Caption**.

? Correr el programa y escribir algo en el editor.

Para probar: ¿Qué cambia si en lugar de 'form1.caption' escribimos 'button1.caption'?

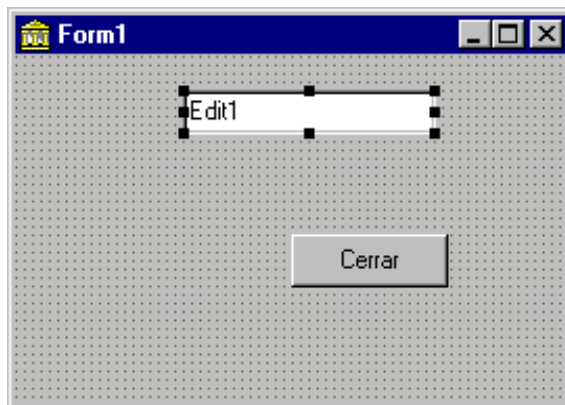


Figura 15: la ficha del ejemplo 2

⁴ Hay que seleccionar el editor antes de escribir el evento. Comprobar con el nombre que aparece en la parte superior del Inspector de Objetos.

Sumario

En esta introducción hemos visto algunas de las características que hacen que la programación en Delphi sea rápida y cómoda. Ahora Ud. debería saber

- ? Cómo determinar si el proyecto está corriendo o en tiempo de diseño.
- ? Cómo crear procedimientos de respuesta a eventos.
- ? Cuales son los archivos que hay que trasladar para seguir trabajando con un proyecto en otro equipo.
- ? Cómo cambiar el nombre de un proyecto o parte de él.
- ? Distinguir los distintos tipos de propiedades y la forma de trabajar con ellas en el Inspector de Objetos.

En los próximos capítulos nos concentraremos en los componentes, que son el motor de las aplicaciones y la clave del éxito de Delphi.

Programar en Delphi

En este capítulo veremos los componentes fundamentales de toda ventana de entrada de datos, mediante un ejemplo concreto del proyecto BIBLIO. Asimismo presentaremos algunas técnicas de trabajo con los componentes como el alineado, el orden de secuencia, las teclas de atajo, etc. que darán un toque profesional a nuestras aplicaciones.

Los componentes básicos

Veremos ahora los controles básicos, aquellos que están presentes en prácticamente todas las aplicaciones de Windows, y los componentes de Delphi que los modelan. Aprovecharemos, además, para practicar la interacción con el IDE.

El objetivo aquí no es crear una aplicación completa y funcional, sino sólo mostrar la forma de programar una parte de la aplicación, cómo están interrelacionados el diseño en la pantalla y el comportamiento de los componentes.

En este capítulo trabajaremos sobre un form de entrada de datos de BIBLIO (un sistema para llevar datos de libros o artículos), que tiene el siguiente aspecto una vez terminado:

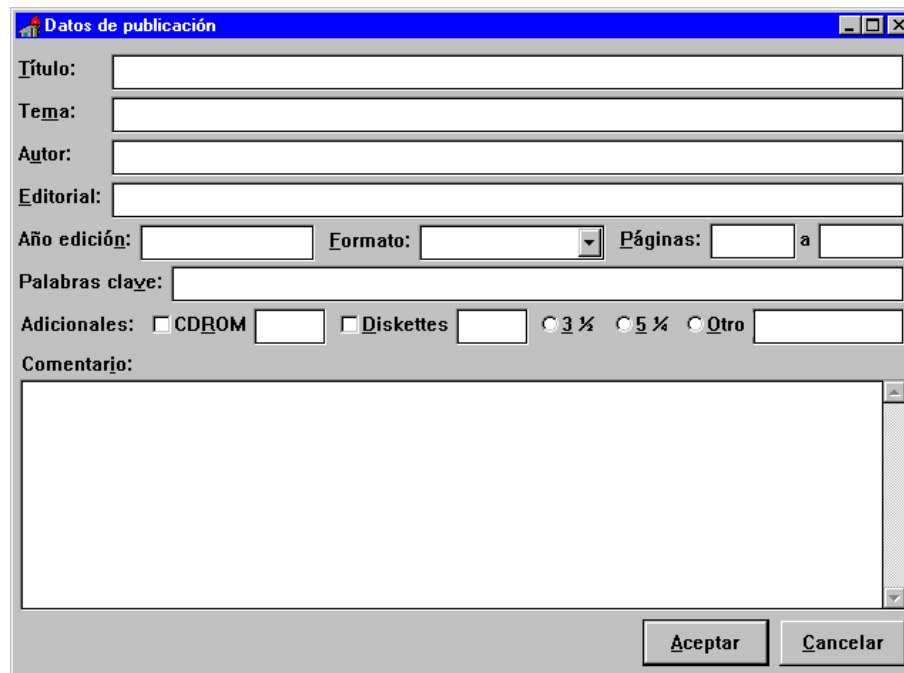


Figura 1: la ficha de entrada de datos una vez terminado

Trabajaremos sobre los distintos componentes individualmente, indicando en cada caso algunas particularidades. Asimismo, comentaremos sobre la marcha algunas herramientas del IDE para trabajar con los componentes.

Etiquetas



Labels (etiquetas) Cumplen una función meramente informativa, indicándonos qué es lo que vemos en una lista, qué dato espera un editor, etc. Clase: TLabel.

Ejemplo:

Pondremos algunas de las etiquetas del form de entrada de datos de BIBLIO.

Colocar las cuatro etiquetas de la izquierda del form, espaciándolos igualmente dejando lugar para los editores.

Necesitamos alguna herramienta que nos deje alinear varios componentes. Pero primero debemos seleccionar los componentes sobre los que vamos a trabajar. Esto se puede hacer de dos maneras:

- ? seleccionar uno por uno manteniendo la tecla **shift** presionada
- ? “dibujar” un recuadro con el mouse alrededor de los componentes deseados

NOTA: para seleccionar un componente que contiene a otro y está cubierto por este último, seleccionar el de arriba y presionar ESC repetidas veces. Cada vez se selecciona el componente contenedor del actual (el último es el form).

Los comandos de alineación disponibles son los siguientes:

Horizontal			Vertical		
	No change	No cambia la posición de los componentes		No change	No cambia la posición de los componentes
	Left Sides	Coloca los bordes izquierdos en una misma línea		Tops	Coloca los bordes superiores en una misma línea
	Centers	Coloca los centros en una misma línea		Centers	Coloca los centros en una misma línea
	Right Sides	Coloca los bordes derechos en una misma línea		Bottoms	Coloca los bordes inferiores en una misma línea
	Space equally	Divide el espacio entre el primero y el último componente seleccionado entre todos, de manera que queden equiespaciados		Space equally	Divide el espacio entre el primero y el último componente seleccionado entre todos, de manera que queden equiespaciados
	Center in Window	Centra el grupo de componentes seleccionados en el ancho total del contenedor		Center in Window	Centra el grupo de componentes seleccionados en el ancho total del contenedor

Tenemos dos vías en Delphi para acceder a los comandos de alineación:

? la opción de menú **View|Alignment palette**

Con esta ventana cubierta de botones accedemos fácilmente a los comandos de alineación, los cuales aparecerán en un recuadro amarillo si nos detenemos encima de un botón por unos segundos.

? el menú contextual en el área del form, y la opción **Align...** que nos trae una caja de diálogo con los comandos de alineación.

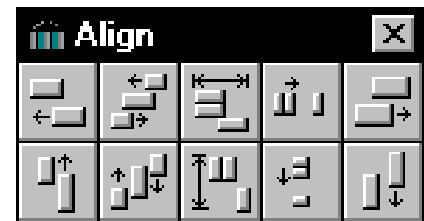


Figura 2: paleta de alineación

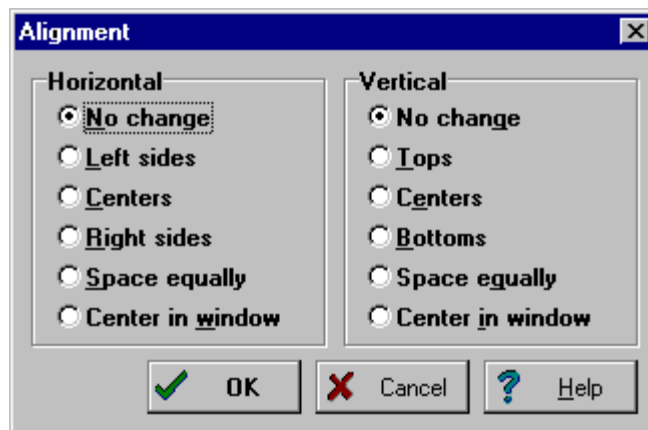


Figura 3: las opciones de alineación de componentes

Para colocar las etiquetas procedemos así:

1. Dimensionar el form al tamaño aproximado que tendrá definitivamente. En nuestro caso lo hacemos

NOTA: es conveniente diseñar los programas de uso general para ser ejecutados en una pantalla VGA común, ya que hacerlo para SVGA impondría un requerimiento que no todos los usuarios pueden cumplir.

- del ancho total de la pantalla en VGA (640 pixels).
- Colocar las etiquetas y cambiar el texto (*caption*) de cada uno como muestra la fig. 19. Note que al cambiar el caption se ajusta automáticamente el tamaño de los componentes; esto es debido a una propiedad de las etiquetas llamada **AutoSize**. Colóquela en *False* para ver el efecto.

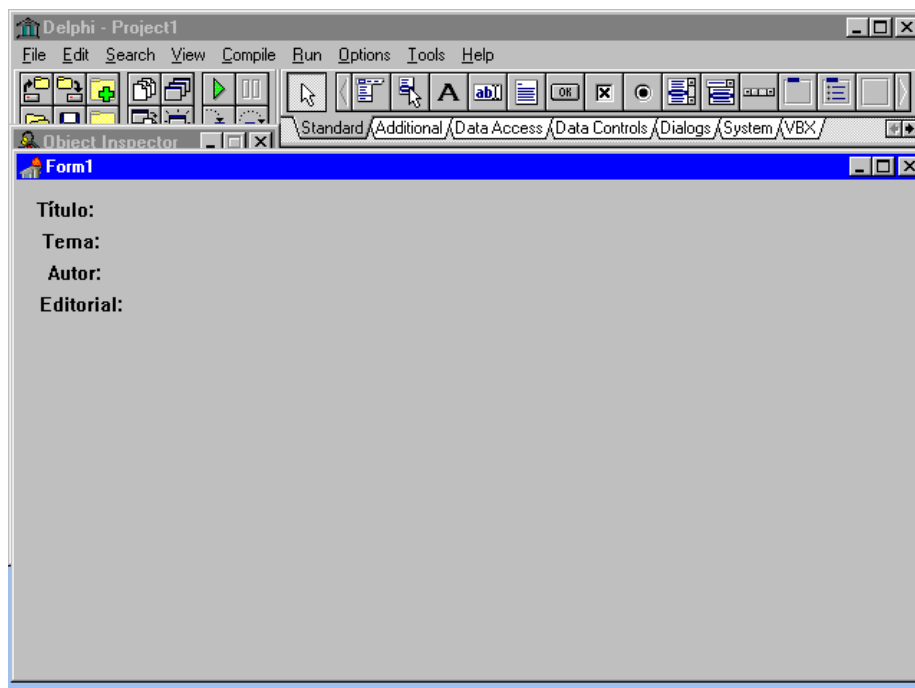


Figura 4: El form con las primeras etiquetas colocadas

- Seleccionar todas las etiquetas

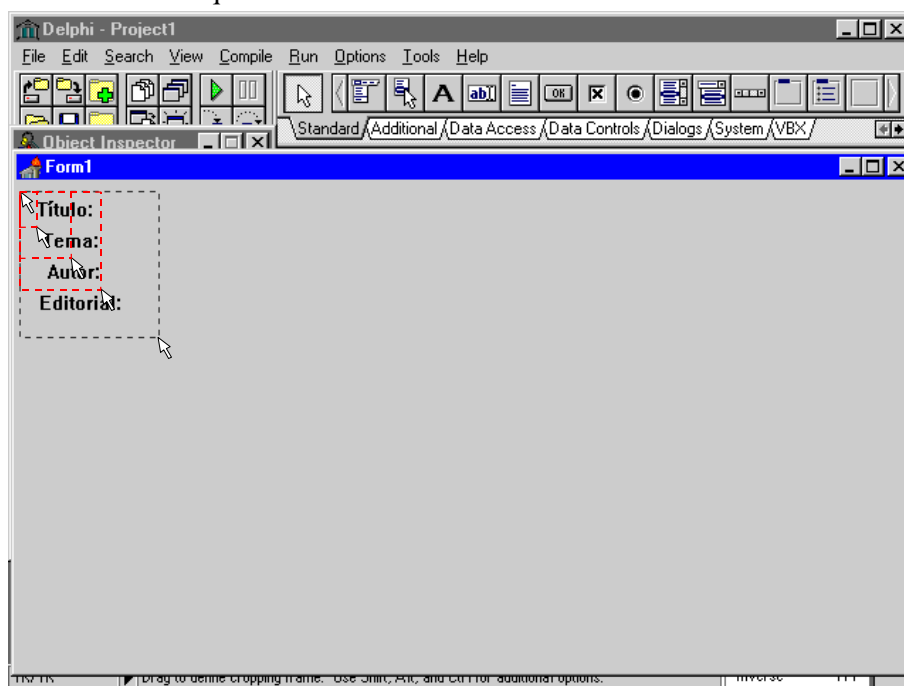


Figura 5: Selección de varios componentes

NOTA: para seleccionar un conjunto de componentes no es necesario encerrarlos completamente en el recuadro del mouse, sólo tocarlos con el mismo.

- Presionar el botón derecho para mostrar el menú local, y seleccionar **Align...**

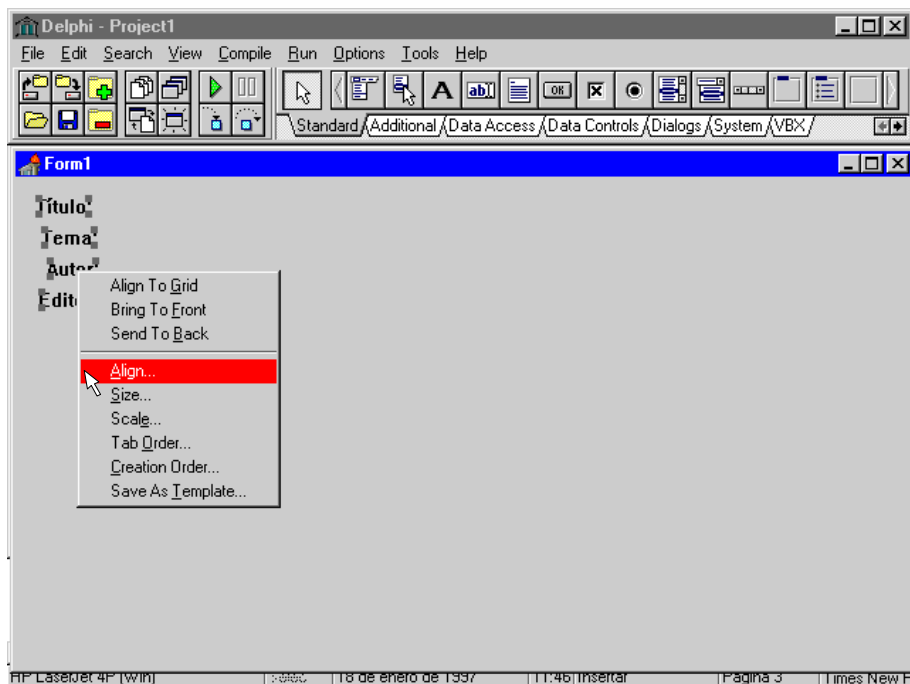


Figura 6: con el botón derecho del ratón tenemos acceso al comando de alineación

5. En el diálogo de alineación, seleccionar **Left Sides** para alinear los bordes izquierdos
6. Hacer lo mismo para lograr un equiespaciado en vertical (opción Vertical|Space equally)

NOTA: estas dos operaciones se pueden realizar en forma simultánea dado que una es en sentido horizontal y la otra en vertical.

Veamos ahora otro tipo de controles muy usados, que nos permitirán introducir texto.

Editores



Editor

Es una línea de entrada con características de edición limitadas, como por ejemplo interacción con el clipboard, manejo de cursores, selección, etc.
Clase: TEdit

Los editores simples tienen ya incorporadas algunas capacidades sencillas pero útiles, como ser la selección de texto con **Shift**, la posibilidad de copiar y pegar, etc.

Ejemplo:

1. Agregar al form anterior los editores de **Nombre**, **Tema**, **Autor** y **Editorial**.

Se colocan igual que las etiquetas, salvo que no tienen las propiedades **Caption** ni **AutoSize**. El texto que contienen está en la propiedad **Text**.

2. Borrar el contenido de los editores.
3. Redimensionar los componentes para que ocupen casi hasta el borde derecho.

El form debe quedar como indica la figura 23.

Antes de continuar, un pequeño ejercicio: coloque las siguientes etiquetas y editores de manera que el aspecto de la ventana sea el de la fig.23:

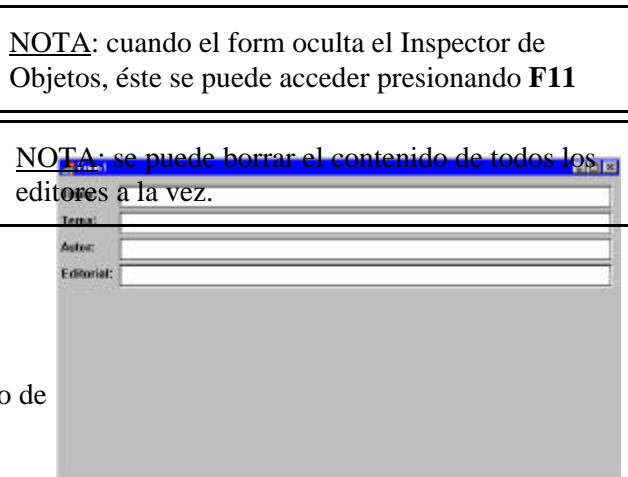


Figura 7: el formulario cuando ya se han agregado los editores

Figura 8: el formulario con todos los editores y etiquetas

Check Box



Cuadro o caja de selección (check box)

Permiten seleccionar opciones no excluyentes (cuando es correcto seleccionar más de una a la vez). Clase: `tCheckBox`

Ejemplo:

Utilizaremos estos controles para la parte de Adicionales (CDROM / Diskettes) porque se puede dar el caso que una publicación tenga los dos adicionales.

1. Seleccionar de la paleta la herramienta de *Check Box*
2. Colocar en el form dejando lugar para la palabra “Adicionales”
3. Cambiar el **caption** para que diga “CDROM”
4. Cambiar el tamaño para que no se solape con el control de la derecha.

Hacer lo mismo con el otro cuadro, “Diskettes”. Agregar también la palabra “Adicionales” a la izquierda. El form debe quedar como en la figura 24

Figura 9: Después de agregar los check boxes

Otro tipo de opciones son aquéllas en que sólo podemos seleccionar una a la vez; para ello existe un tipo diferente de control denominado Botón Radial (`RadioButton`) por su semejanza con los botones selectores de canales de radio, en los que sólo se permite la selección de uno a la vez.

Botones radiales (Radio Buttons)



Botones radiales (Radio Button) Permiten seleccionar una opción entre varias excluyentes (sólo una de ellas puede estar seleccionada). Clase: `tRadioButton`.

Los botones radiales se utilizan generalmente en grupos, dentro de los cuales sólo uno puede estar marcado. Para crear más de un grupo de botones en un mismo form, utilice el contenedor **GroupBox**. También hay un control especial: **RadioGroup**, que permite crear un grupo de botones radiales modificando algunas propiedades.

Ejercicio:

Colocar en el form los botones radiales de tamaño de diskette: **3 ½**, **5 ¼** y **Otro**. Se procede de la misma manera que con los check box.

Una vez colocados los botones radiales, los textos y editores en la línea de *Adicionales*, el form queda como se muestra en la figura 25:

The screenshot shows a window titled 'Form1' with the following controls:

- Título: [Text input field]
- Tema: [Text input field]
- Autor: [Text input field]
- Editorial: [Text input field]
- Año edición: [Text input field] Páginas: [Text input field] a [Text input field]
- Palabras clave: [Text input field]
- Adicionales: CDROM [Text input field] Diskettes [Text input field] 3 ½ 5 ¼ Otro [Text input field]

Figura 10: el formulario casi terminado

Pondremos ahora el editor multilínea de la parte inferior del formulario. La diferencia con el Editor simple es que este componente acepta varias líneas de texto.

Editor de notas (memo)



Memo Editor multilínea simple con soporte de portapapeles. Clase: `tMemo`

Después de agregar este editor al form, queda de la siguiente manera (fig 26):

The screenshot shows a Windows form titled "Form1" with the following fields and options:

- Título:** [Text input field]
- Tema:** [Text input field]
- Autor:** [Text input field]
- Editorial:** [Text input field]
- Año edición:** [Text input field] **Páginas:** [Text input field] a [Text input field]
- Palabras clave:** [Text input field]
- Adicionales:**
 - CDROM [Text input field]
 - Diskettes [Text input field]
 - 3 1/2
 - 5 1/4
 - Otro [Text input field]
- A large empty text area at the bottom.

Figura 11: Después de agregar el memo

A diferencia de los editores simples, el contenido del control Memo se encuentra en una propiedad llamada **Lines**. Accediendo al editor de la propiedad con un click en el botón con tres puntos, podemos borrar el contenido inicial (el nombre del componente). Después veremos cómo acceder a las líneas de texto desde el programa en tiempo de ejecución.

Otra propiedad útil del memo es **ScrollBars**. Es una propiedad enumerada y los valores posibles se presentan en una lista. Con estos valores podemos elegir que el control tenga una barra de desplazamiento horizontal (*ssHorizontal*), vertical (*ssVertical*), ambas (*ssBoth*) o ninguna (*ssNone*, valor por defecto). La presencia de las barras no implica que no se pueda desplazar el texto usando el teclado, son sólo para usar con ratón.

También utilizaremos las propiedades **WantReturns** y **WantTabs**, que indican si el control acepta las teclas *Return* y *Tab* respectivamente. Dado que estas teclas tienen respuestas especiales dentro de una caja de diálogo (*Return* activa el botón por defecto y *Tab* pasa el foco al control siguiente) hay que indicar expresamente al memo que queremos utilizar estas teclas como en un editor normal (es decir, como saltos de línea y de tabulación). Si ponemos la propiedad **WantTabs** en *True* sólo saldremos del control por medio de teclas de atajo o pulsando con el ratón sobre otro control.

Si damos valor *False* a estas propiedades, aún podemos insertar saltos de línea o tabulaciones presionando *Ctrl* junto con *Return* o *Tab*.

Una propiedad más, llamada **WordWrap**, controla si las palabras que excedan el borde derecho del memo deben escribirse en la línea siguiente (*True*) o no (*False*). Esta propiedad únicamente funciona si no especificamos barra de desplazamiento horizontal.

En nuestro form de entrada de datos pondremos una barra vertical, con corte de palabras (**WordWrap**) y dejaremos la tecla *Tab* para pasar de un control a otro mientras que tomaremos el *Return* como salto de línea. Para ver el resultado visual (sólo veremos la barra de desplazamiento) remitimos al lector a la figura 16 al principio del capítulo.

Veamos ahora cómo implementar la entrada del *formato* del artículo. Queremos que, para evitar errores de escritura (como por ejemplo **libro** y **Libro**), el programa ofrezca al usuario un conjunto de opciones predeterminadas. Para ello necesitamos una lista de opciones y un lugar donde mostrar la elegida.

Utilizaremos un tipo especial de editor que permite la especificación de una lista de opciones: los **combo box**.

Editor con opciones (Combo Box)



Editor con opciones (combo box) Una línea de edición con una lista asociada. Podemos desplegar la lista con el botón de la derecha y seleccionar un ítem que pasará al editor. Clase: `tComboBox`.

Para colocar el editor con opciones de Formato, procedemos de la siguiente manera:

1. Colocamos un componente **Combo Box** en el lugar adecuado, redimensionándolo de manera que ocupe el espacio correctamente.
2. La lista de palabras se mantiene en una propiedad llamada **ítems**. Con doble click o presionando el botón con tres puntos accedemos al editor.
3. Escribir los formatos predefinidos siguientes en el editor de ítems:
 - a. Libro
 - b. Revista
 - c. Fotocopias
 - d. CD RomCerrar el editor.
4. Cambiar la propiedad **style** a `csDropDownList`. Esto hace que no se pueda escribir en el editor, sólo permite que se elija una palabra de la lista. Veremos estas propiedades con más detalle en un capítulo posterior.
5. Cambiar la propiedad **text** a **Libro**. Este es el valor que aparecerá por defecto al correr la aplicación. El aspecto del form después de este agregado es el siguiente:

The screenshot shows a Windows form titled 'Form1'. It contains several input fields: 'Título:', 'Tema:', 'Autor:', 'Editorial:', 'Año edición:', 'Formato:' (with a dropdown menu showing 'ComboBox1'), 'Páginas:', 'Palabras clave:', and 'Adicionales:' (with checkboxes for 'CDROM' and 'Diskettes', and radio buttons for '3 1/2', '5 1/4', and 'Otro'). There is a large empty text area at the bottom.

Figura 12: se agregó el combobox

Sólo nos queda ahora agregar los botones que servirán para aceptar o cancelar las modificaciones de los datos: los típicos botones.

Botones de comando (Button)



Botón (Button) Nos permiten activar o desactivar procesos, abrir o cerrar ventanas, etc. Los hay de varios tipos; este es el común que todas las aplicaciones de Windows poseen, denominado **push button**. Clase: `tButton`.

Pondremos dos botones en el form, uno con caption **Aceptar** y el otro **Cancelar**. Cambiando un par de propiedades más lograremos que estos botones se comporten correctamente ante las pulsaciones de teclas como Enter y Escape:

En el botón **Aceptar**:

- ? ponemos la propiedad *Default* a **true**. Esto hace que responda al Enter; no es necesario presionar el botón con el ratón.
- ? En la propiedad *ModalResult* elegimos de la lista el valor **mrOk**. Este valor hace que el botón cierre el form y además nos permite saber que se presionó **Aceptar**.

En el botón **Cancelar**:

- ? ponemos la propiedad *Cancel* a **true**. Esto hace que responda al Escape (presionar la tecla Escape será lo mismo que presionar este botón con el ratón)
- ? en la propiedad *ModalResult* elegimos el valor **mrCancel**.

Tenemos casi terminado nuestro form (fig. 28); sólo faltan unos detalles para permitir la navegación con el teclado, sin usar el ratón.

NOTA: Es muy importante siempre permitir que el usuario pueda manejarse sin el ratón; puede suceder que éste no funcione bien o que resulte engorroso separar la mano del teclado y apuntar con el puntero a un control cuando está escribiendo.

Figura 13: la parte visual está terminada

Si corremos la aplicación generada hasta ahora, veremos el form que creamos como ventana principal; los botones no servirán para cerrarla porque están pensados para un form secundario de tipo **modal** (este concepto será explicado luego; un poco de paciencia). Para cerrar el form y con él la aplicación debemos usar **Alt+F4** o el menú de sistema que aparece a la izquierda de la barra de título.

Las teclas de atajo

Las *teclas de atajo* o *shortcuts* son un medio para reemplazar un click del ratón con el teclado. Se utilizan en varios tipos de controles, con comportamientos un poco diferentes en cada caso. Por ejemplo, en los menús o botones se toma la pulsación del atajo como si hiciéramos click con el ratón, mientras que en las etiquetas sirve para cambiar el foco a un control asociado. En cualquier caso, la letra que usaremos para el atajo se indica en el texto del caption precediéndola del carácter **&**. En el form se verá subrayada.

El control que recibirá el foco se indica en la propiedad **FocusControl** de la etiqueta. El editor de esta propiedad es del tipo “combo box”, nos muestra en una lista los controles que podemos utilizar con esta técnica (no se listan por ejemplo las otras etiquetas, que no pueden recibir el foco).

Implementaremos por lo tanto esta facilidad en el form de entrada de datos. Observe el form terminado al principio del capítulo y cambie los captions de las etiquetas y los botones. Los controles asociados a las etiquetas son los editores correspondientes. En el

NOTA: en los atajos que podamos, conviene usar la primera letra del texto. Como no puede haber dos atajos iguales, a veces tendremos que utilizar otras letras. Como norma general, trate de respetar los atajos de los botones a través de todas las cajas de diálogo del programa para no confundir al usuario.

caso de los botones radiales o cuadros de opción, cada uno trae su propia etiqueta; con sólo agregar el caracter **&** delante del caracter correspondiente en la propiedad caption es suficiente.

El orden es importante

Es importante guardar un cierto orden en la distribución de los componentes, que nos permita navegar por el form con Tab y Shift+Tab fácilmente. Si colocamos los componentes en el orden correcto, este es el comportamiento por defecto. Caso contrario, debemos especificar el orden en que se recorren los controles. Este ordenamiento se denomina *Tab Order*, en alusión a la tecla Tab que sirve para pasar de uno a otro. Para cambiar el orden de recorrido, pulsamos el botón derecho del ratón en algún lugar del form; en el menú local que aparece seleccionamos *Tab Order*:

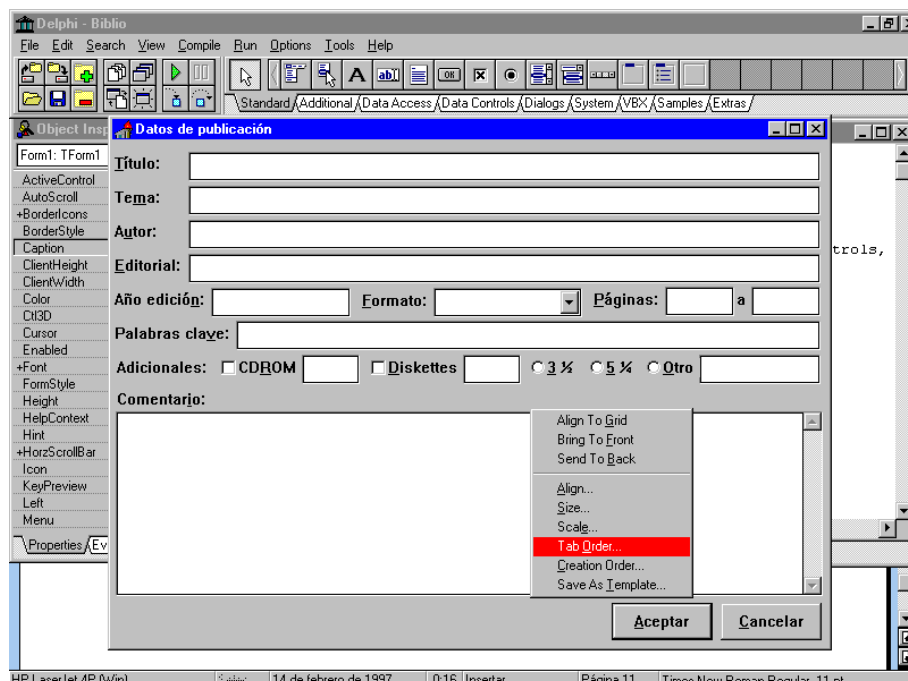


Figura 14: desde el menú contextual se accede al orden de tabulación de los componentes

Se nos presenta una lista con los componentes que posee el form en el orden actual. Para cambiarlo debemos arrastrar con el ratón el componente al lugar deseado de la lista.

Por ejemplo, en la fig. 30 se ve la lista de componentes del form de entrada de datos ¹ y el cambio necesario para que el editor de Título (*edTitulo*) se acceda antes que el de Tema (*edTema*).

¹ Los componentes han sido renombrados para facilitar su identificación (propiedad *name*)

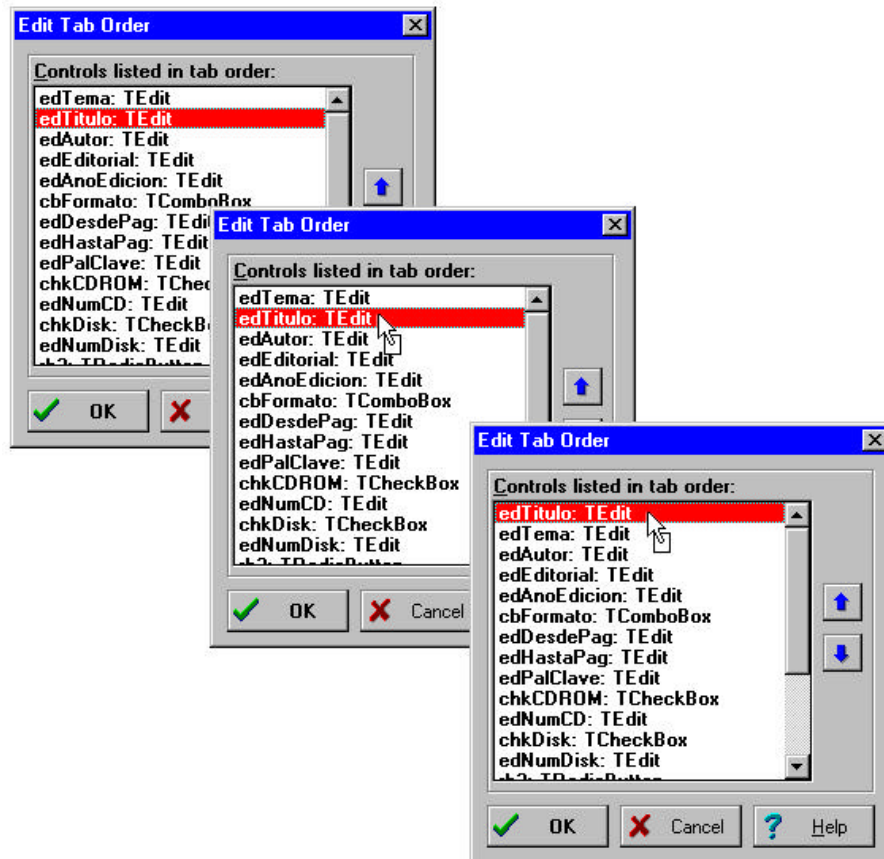


Figura 15: Cambio del orden de tabulación de un componente

Repetimos que el correcto orden de secuencia de los componentes es muy importante; basta con utilizar un poco el form para darse cuenta.

Después de completar las modificaciones, compile y ejecute el programa para comprobar que el comportamiento sea el esperado.

Sumario

En este capítulo hemos hecho una recorrida por las técnicas básicas de creación de forms con Delphi; aprendimos a alinear los diferentes componentes y a ordenarlos en una secuencia lógica. Además pasamos revista a una serie de consideraciones válidas para cualquier programa, como las teclas de atajo y los botones por defecto. A partir de ahora tocaremos temas más complicados que hacen a cualquier programa más complejo que una ventana de ingreso de datos.

VCL - Componentes y eventos básicos

Veremos aquí los componentes básicos de la VCL (Visual Components Library, Biblioteca de Componentes Visuales), sus propiedades y eventos. Esto nos permitirá crear programas medianamente sofisticados, con todas las características de los programas comerciales bajo Windows.

Los botones y el evento OnClick

Para realizar los ejemplos de esta sección, tendremos que escribir algo de código (después de todo eso es lo que estamos estudiando, no?). Ahora bien, hay que ver el lugar donde se escribe y el momento en que se ejecutará nuestro código.

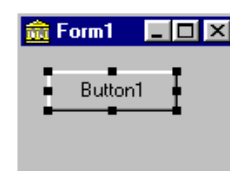
Normalmente, las porciones de programa que escribimos responden a algún *evento* del sistema: un movimiento del mouse, la presión de uno de los botones del mismo (**click**), etc. Windows captura estos eventos y los envía a las aplicaciones que estén funcionando en ese momento. Delphi procesa estos mensajes del sistema y los transforma en llamadas a procedimientos especiales que llamaremos *procedimientos de respuesta a eventos*; Una denominación muy imaginativa, por cierto.

En los siguientes ejemplos utilizaremos un botón como desencadenante de la acción deseada; el evento será apretar el botón izquierdo del ratón sobre este botón, un evento llamado **OnClick**.

Al igual que con las propiedades, vemos los eventos en el Inspector de Objetos. Debemos asegurarnos cada vez que estamos viendo los eventos del componente deseado, seleccionándolo primero.

Veamos un ejemplo: queremos que cuando se presione el botón “hola” en la siguiente ventana aparezca un mensaje de saludo. Para ello:

1. Seleccionar el botón haciendo click sobre él. Aparecen los manejadores (pequeños cuadrados negros) y en la parte superior del Inspector de Objetos podemos ver el nombre del componente -normalmente, Button1.



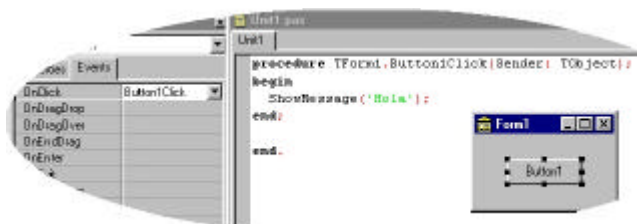
2. Pasar a la página “Events” en el Inspector de Objetos.

3. En la columna izquierda -nombre del evento- buscamos **OnClick**. En la columna derecha -nombre del procedimiento de respuesta- hacemos *dobles click* en la parte blanca (donde podemos escribir)¹. Delphi crea un procedimiento de respuesta al evento, le asigna un nombre formado por el nombre del componente y el evento en cuestión, escribe las declaraciones necesarias en el editor de código y coloca el cursor entre las sentencias **begin** y **end**, listo para escribir nuestro programa.

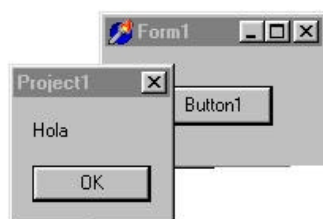


4. Escribimos la siguiente línea:

```
ShowMessage( 'Hola' );
```



Ya hemos creado nuestro programa. Ahora seleccionamos la opción “Run” del menú “Run” o bien presionamos la tecla F9 para compilarlo y ejecutarlo. Cuando aparece la ventana, presionamos el botón y obtenemos el mensaje.



¹ También podemos escribir un nombre para el procedimiento, que se creará automáticamente al presionar <Enter>

Ahora sí, ya podemos centrarnos en las propiedades. Escribiremos los programas de muestra como respuesta al evento `OnClick` de un botón, de la misma manera que en el ejemplo anterior. Luego discutiremos los eventos con mayor profundidad.

Posición y tamaño en pantalla

Todos los componentes que son visibles tienen al menos algunas características comunes: las que indican la posición en pantalla y el tamaño de la región que ocupan en la misma. Estas características se mantienen en las siguientes propiedades:

- `Left` (*integer*): coordenada horizontal del borde izquierdo del componente
- `Top` (*integer*): coordenada vertical del borde superior del componente
- `Width` (*integer*): ancho del componente
- `Height` (*integer*): altura del componente

Hay que tener en cuenta la diferencia entre el sistema de coordenadas cartesianas que utilizamos en matemática y el sistema que usa Windows: en este sistema, el eje Y se sitúa sobre el borde superior del área de interés, y crece de arriba hacia abajo. Podemos verlo mejor en la figura 5.

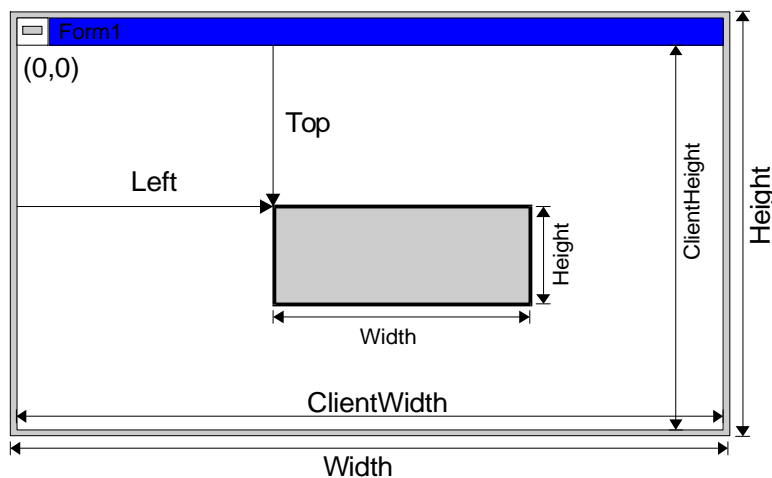


Figura 5: Sistema de coordenadas

Las coordenadas de cada componente son relativas al componente en el cual está inserto; por ejemplo, las del form son relativas al escritorio de Windows (la pantalla total) y las de los controles que están dentro del form, relativas al form mismo.

¿Qué podemos hacer con estas propiedades? Dado que indican la posición y el tamaño de la imagen del componente en pantalla, podemos utilizarlas para mover el componente por la pantalla y para redimensionarlo.

Ejemplo 1

Se trata de cambiar la posición de un botón en la pantalla. Inicialmente aparece en una esquina, y cada vez que lo presionamos se mueve a la esquina siguiente en sentido de la rotación de las agujas del reloj.

Comenzamos creando un nuevo proyecto. En la ventana principal (vacía) colocaremos un botón, inicialmente en la esquina superior izquierda. Para ello será necesario editar directamente los valores de las propiedades `Left` y `Top` del botón en el Inspector de Objetos, asignándoles valor 0 (cero).

El algoritmo de movimiento se podría esquematizar como se ve en la figura 6.

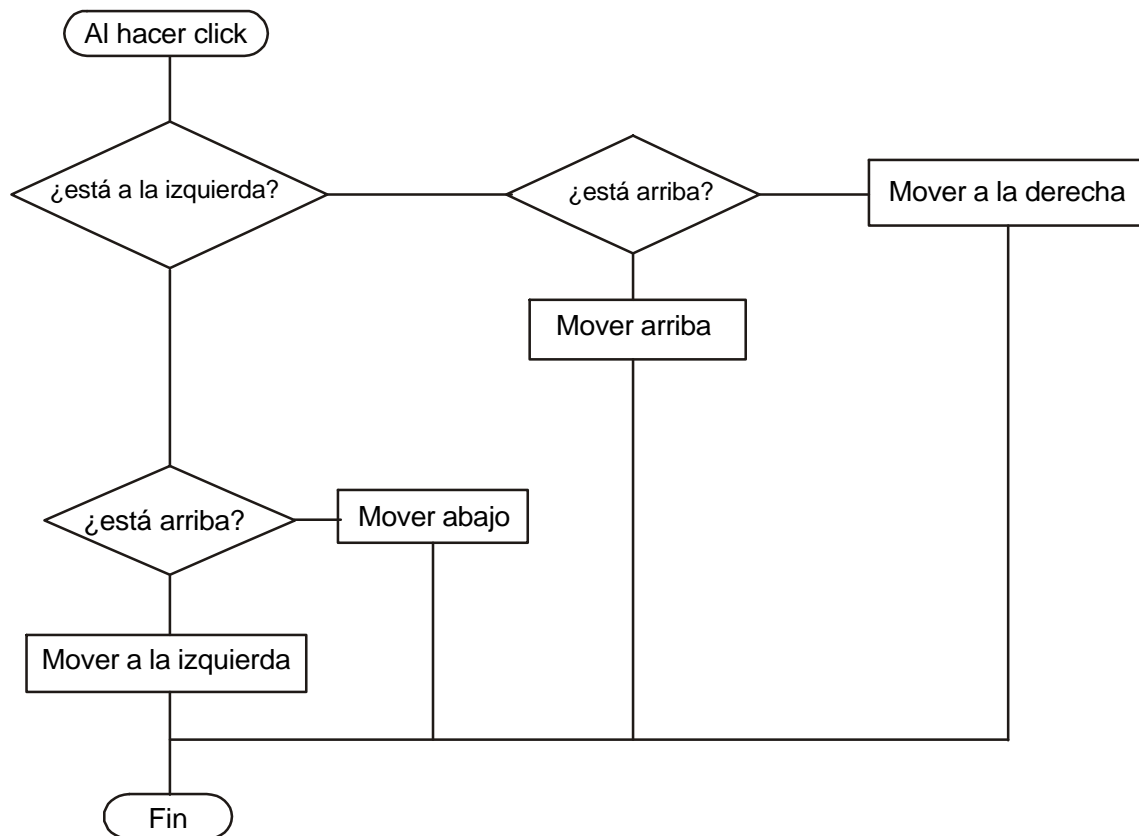


Figura 6: algoritmo utilizado para mover el botón

Para escribir este algoritmo necesitamos saber *cuándo* va a activarse el código, es decir qué evento debemos esperar para contestar con el algoritmo. La respuesta la tenemos en las especificaciones del problema: dice “cada vez que lo presionamos” (al botón). Por lo tanto, el evento será *OnClick del botón*. El siguiente código realiza la tarea deseada:

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  if Button1.Left=0 then begin
    if Button1.top=0 then
      Button1.Left:= ClientWidth-button1.Width
    else
      Button1.Top:= 0;
  end else begin
    if Button1.top=0 then
      Button1.Top:= ClientHeight-Button1.Height
    else
      Button1.Left:= 0;
  end;
end;

```

Notemos que debemos indicar a Delphi *de quién* son las propiedades que vamos a modificar; para ello, en el código tenemos que agregar el *nombre* del control y separarlo de la propiedad con un punto.

Ejercicio
Modifique el programa anterior para que el botón “gire” en sentido contrario

?

La propiedad Name

El nombre de un componente es otra propiedad del mismo: la propiedad *Name*. Es de tipo String y se puede cambiar en el Inspector de Objetos -y se ve en la parte superior del mismo.

Hay que tener en cuenta un par de reglas al cambiar la propiedad Name:

1. Debe ser única para cada control dentro del mismo form.
 2. No puede quedar vacía.
-

Por ejemplo, para referirnos a la propiedad **Left** del botón llamado **Button1** escribiremos **Button1.Left**.

Las propiedades ClientWidth y ClientHeight

Notemos también la utilización de dos propiedades que no hemos comentado hasta el momento: **ClientWidth** y **ClientHeight**. Son propiedades del form (no es necesario decir a quién pertenecen porque el botón está dentro del form) e indican el ancho y la altura del área útil del form, respectivamente. El área útil es la porción del form que podemos utilizar directamente para nuestros fines; difiere del área total de la ventana porque ésta tiene un título y bordes en los otros costados, lugares que no podemos acceder directamente². Se denomina Área del Cliente.

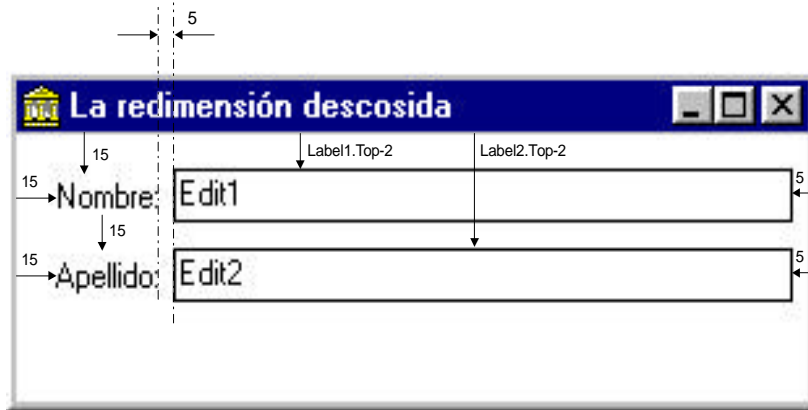
En la figura 5 se pueden ver las propiedades ClientWidth y ClientHeight.

Utilizando estas propiedades nos aseguramos que el botón se mantendrá siempre en los límites del form, aunque éste sea redimensionado.

² Por supuesto, es posible en definitiva acceder a estas áreas de pantalla; pero debemos hacerlo a través de funciones especiales de la API de Windows.

Ejemplo 2

Se trata de un programa que debe redimensionar otros componentes para que guarden entre sí una relación de posición y tamaño, aunque el form se redimensione. Al presionar un botón, se deben reacomodar los controles como en la figura:



Creamos primero un nuevo proyecto agregando al form en blanco varios componentes, como en la figura 8. A continuación cambiamos los textos que muestran los componentes, utilizando la propiedad caption.

Bien, en esta aplicación hay algunos datos fijos y otros variables; por ejemplo, la posición vertical de los controles está fijada de antemano a una cierta cantidad de pixels (unidad de medida) de distancia de la barra de título. En cambio, el ancho de los editores se debe ajustar al ancho que tenga la ventana cada vez que apretamos el botón: nuevamente llega a nuestro auxilio la propiedad ClientWidth del form.

El código del evento OnClick del botón es el siguiente:

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Top:= 15;
  Label1.Left:= 15;
  Label2.Top:= Label1.Top+Label1.Height+15;
  Label2.Left:= 15;
  Edit1.Top:= Label1.Top-2;
  Edit1.Left:= Label1.Left+Label1.Width+5;
  Edit1.Width:= ClientWidth-Edit1.Left-5;
  Edit2.Top:= Label2.Top-2;
  Edit2.Left:= Label2.Left+Label2.Width+5;
  Edit2.Width:= ClientWidth-Edit2.Left-5;
end;
    
```



Figura 8: El form principal una vez modificados los textos

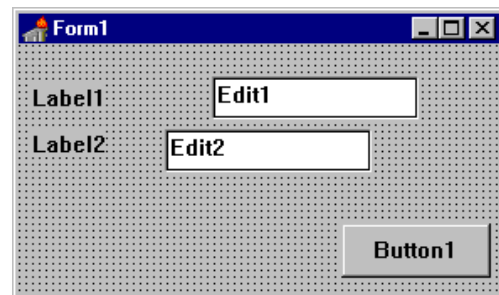


Figura 9: El form principal con los componentes

Aprovechando el hecho que las propiedades anteriores están en todos los componentes visibles, podemos incluso trabajar con componentes que no conocemos.

Ejemplo 3

Por ejemplo, haremos un programa que haga rebotar una “pelota” en el fondo del form. Para la pelota utilizaremos un control “Shape” que se encuentra en la página *Additional* de la paleta de componentes.

Por defecto, este control asume la forma de un rectángulo; para lograr el círculo que necesitamos para emular la pelota hay que cambiar la propiedad **shape** al valor *stCircle*. Agregamos además el consabido botón y escribimos el código siguiente en el evento *OnClick*:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: integer;
begin
  for i:= Shape1.Top to ClientHeight-Shape1.Height do
  begin
    Shape1.Top:= i;
    shape1.update;
  end;
  for i:= shape1.top downto 0 do
  begin
    Shape1.Top:= i;
    shape1.update;
  end;
end;
```

Veamos un poco el programa en un diagrama de flujo (fig 10).

Utilizamos aquí dos bucles. Nuestra “pelota” caerá y se levantará rápidamente. En realidad, lo hace tan rápidamente que hay que indicar a la figura que queremos ver cada uno de los pasos; esa es la causa de la llamada al método **update**.

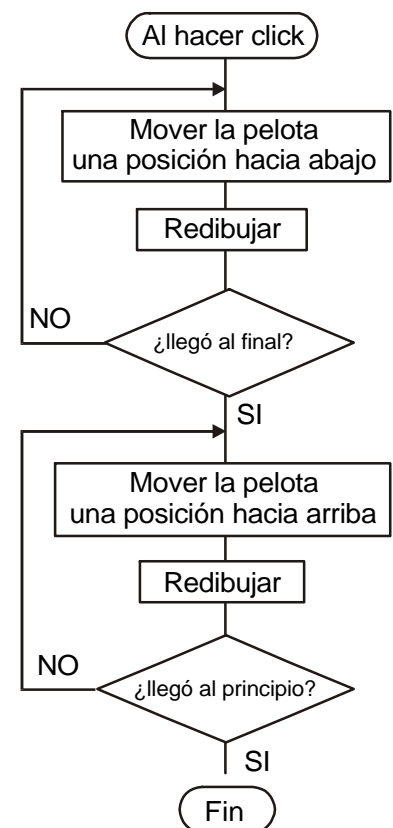


Figura 10: algoritmo del rebote

El sistema Windows redibuja la pantalla cuando tiene tiempo; si no le dejamos un ratito entre cada paso, directamente descarta los cuadros que no puede mostrar y nos presenta el final. Con el método **update** (que tienen todos los controles visuales) forzamos al sistema a dibujar el control *en ese instante*, interrumpiendo lo que pudiera estar haciendo.

El texto de los componentes: Caption y Text

Siguiendo con las características de presentación en pantalla, podemos ver que muchos de los componentes visibles presentan algún texto. Dos de las propiedades asociadas a esta característica son

- **Caption:** propiedad de tipo String que almacena el texto que se muestra en los controles que no son editables por el usuario, por ejemplo botones, etiquetas, forms, etc.
- **Text:** propiedad de tipo String que almacena el texto contenido en los controles editables (es decir: lo que el usuario escriba en uno de estos controles lo podremos leer en esta propiedad). Como ejemplos podemos citar los editores comunes (Edit), editores con opciones (ComboBox), etc.

Tenga en cuenta que estas propiedades son generales para los controles que contienen *una sola línea de texto*; los editores multilinea (memos) por ejemplo, almacenan todas las líneas en otro tipo de propiedad que veremos en breve.

Ejemplo 4

Se trata de un programa que calcule el área de un triángulo dadas la base y la altura. Utilizaremos editores para el ingreso de datos y una etiqueta para mostrar el resultado.

El form principal se puede ver en la figura 11.

Al presionar el botón “Calcular”, se deben tomar los números que el usuario escribió en los editores y utilizarlos para calcular la superficie, mostrando el resultado en la etiqueta llamada “Label4”. Veamos el código:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  base, altura, sup: real;
begin
  base:= StrToFloat(edit1.text);
  altura:= StrToFloat(Edit2.text);
  sup:= base*altura/2;
  Label4.Caption:= FloatToStrF(sup, ffFixed, 15, 2);
end;
```



Figura 11: El form de la aplicación en diseño

Podemos ver aquí la utilización de las funciones de conversión entre cadenas y números reales.

Recapitulando, podemos cambiar el texto que se muestra en los controles no editables cambiando el valor de la propiedad *caption* de los mismos. Incluso las ventanas tienen texto en la barra de título, que el usuario no puede modificar -tienen una propiedad *caption*.

Ejercicio

Modifique la propiedad *caption* de la ventana principal, en el Inspector de Objetos, y verifique que el cambio es visible incluso sin ejecutar el programa.

?

Habrán visto que en los programas comerciales de Windows las etiquetas generalmente tienen una letra subrayada. ¿Para qué sirve? Para acceder al control correspondiente (editor, lista, etc), generalmente el que está al lado. Se preguntarán cómo hacerlo en Delphi... y si no se lo preguntaron, ¡háganlo!

Para esta tarea Windows ha reservado un símbolo: el “&”. Se encuentra normalmente en la misma tecla que el número 7. Cuando se escribe uno de estos símbolos como parte de la propiedad **Caption** de un control, el carácter siguiente aparece subrayado. Y no es sólo una coquetería: esta letra automáticamente pasa a jugar el papel de “tecla de atajo”. Presionando la tecla <Alt> junto con la letra o símbolo subrayado, hacemos que el cursor (el foco de teclado) pase al control asociado (por ejemplo un editor). En el ejercicio anterior, podríamos poner como **Caption** de la primera etiqueta la cadena “&Base”, con lo cual aparecerá la letra B subrayada y presionando <Alt>+ pasamos el control al editor asociado. ¿Y cómo sabemos cuál es el editor asociado? ¿Será el de la derecha? ¿Y si cambiamos los lugares? ¿Y si ponemos el editor debajo de la etiqueta? Para evitar ambigüedades, Delphi define una nueva propiedad que nos dice a qué control se pasará el foco cuando se presione la tecla de atajo.

Les dejo a Uds. el placer de encontrar esta propiedad... y probarla.

Cuando hay mucho para decir

Hay varios componentes que no sólo tienen una línea de texto, sino que tienen *muchas*, por ejemplo **ListBox**, **ComboBox**, **Memo**. Todos estos componentes tienen en común que trabajan con varias líneas de texto, por lo que en un entorno de programación orientado a objetos ya nos suena una campanita... ¿se podrá crear una clase general para usar en todos los componentes que tengan esta necesidad? La respuesta, como ya habrán imaginado, es **Si**. Existe una clase así en la VCL, denominada **TStrings**. Es una clase muy útil, ya que nos permite mantener una lista de **strings** en cualquier lugar que la necesitemos (no sólo en los componentes).

La clase **TStrings** es en realidad una clase abstracta; define las características que debe tener una lista de cadenas, pero no tiene forma de almacenarlas. Es como diríamos, el teórico del grupo. Esta clase, por lo tanto, no se utiliza directamente. En su lugar utilizamos la clase descendiente **TStringList** que implementa todos los métodos abstractos de la clase **TStrings** y sabe cómo almacenar las cadenas, ordenarlas, cargarlas y guardarlas en disco, etc. Por lo tanto, todo lo que se hable sobre listas de cadenas es aplicable -en teoría- a la clase **TStrings**, pero si queremos utilizar una usaremos **TStringList**.

Las cadenas se almacenan como... en realidad no nos importa si se utiliza una lista enlazada, un árbol o un grafo; externamente, se nos presenta la lista de cadenas como un *vector* (array) de **Strings**. Por lo tanto, podemos utilizar la sintaxis normal de subíndices para acceder a una cadena particular. Por ejemplo: en un componente **ListBox** la lista se accede a través de la propiedad **Items**, por lo que para acceder a la tercera cadena haríamos (NOTA: el vector empieza en cero)

```
var
  s:string;
begin
  s:= ListBox1.Items[2];
```

Y ya está. Leer es muy fácil, pero ¿qué pasa si quiero *agregar* una cadena a la lista? El vector ¿es de longitud fija, o crece automáticamente para acomodar las nuevas cadenas? Pues resulta, pequeño saltamontes, que la lista crece automáticamente; sólo tenemos que llamar a un método para agregar una cadena, que se llama... **Add**, por supuesto!

El método **Add** espera como parámetro un **String**, y lo agrega a la lista, devolviendo la posición en que se insertó:

```
function Add(const S: string): Integer
```

Cuando no necesitamos el índice, podemos descartarlo sin más o utilizar el método **Append** que hace lo mismo pero sin intentar devolver nada.

Veamos un ejemplo de agregar una cadena a un **ListBox**:

```
Var
  s:string;
begin
  s:= 'Otra cadena';
  ListBox1.Items.Add(s);
```

Podemos ver si se agregó la cadena chequeando la propiedad **Count**, que nos da la cantidad de cadenas que contiene la lista.

También podemos borrar una cadena, por supuesto; para esto se define el método **Delete**:

```
procedure Delete(Index: Integer);
```

Notemos que este método espera que le proporcionemos el *índice* de la cadena a borrar. ¿Y si tenemos la cadena pero no el índice? Pues podemos **buscarla**, claro

```
function IndexOf(const S: string): Integer;
```

Esta función busca la cadena pasada como parámetro y si la encuentra devuelve el índice de la misma (entre 0 y **Count**-1). Si no se encuentra, devuelve -1. Notemos que si la cadena aparece más de una vez en la lista, esta función sólo encuentra la primera ocurrencia.

Si lo que queremos es borrar *toda* la lista de una sola vez, podemos utilizar el método **Clear**.

Y esto es todo lo que necesitamos por el momento para utilizar las listas de cadenas. La clase TStrings y sus descendientes pueden también almacenar *objetos* asociados a las cadenas, pero ese tema se verá cuando tratemos las colecciones.

La propiedad ItemIndex

Cuando trabajamos con los componentes ListBox y ComboBox, podemos seleccionar una cadena; visualmente es fácil saber cuál es. Ahora bien, ¿cómo sabemos *desde el programa* cuál es la cadena seleccionada? Existe una propiedad que contiene esta información, pero **no se ve en el Inspector de Objetos**: se denomina **ItemIndex**.

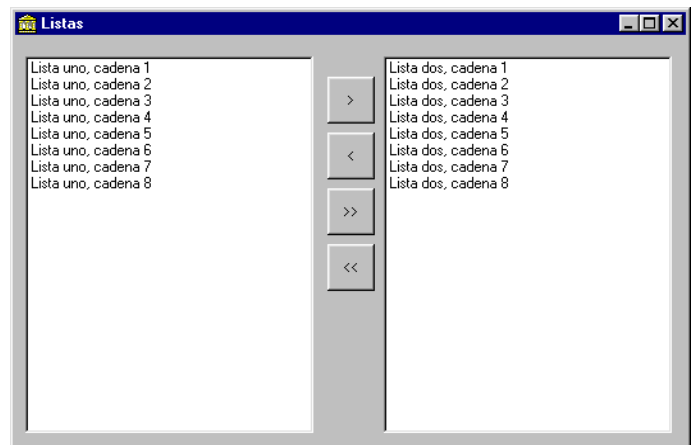
Esta propiedad es de tipo entero, y no contiene otra cosa que el índice de la cadena seleccionada en ese momento. En un **ListBox** esta cadena se ve seleccionada directamente en la lista, mientras que en un **ComboBox** la cadena se copia de la lista al editor; aunque la lista desaparece de la vista sigue estando en memoria (propiedad **Items**).

Con esta información, será sencillo realizar la siguiente aplicación de ejemplo:

Ejemplo 5

Realizar una aplicación que tenga en la ventana principal dos listas y algunos botones, como se ve en la figura. Al presionar los botones marcados con “>” o “<” se debe *mover* (copiar y borrar del original) la palabra seleccionada de una lista hacia la otra, según indique la dirección de la lista. Los botones con doble flecha (“>>” y “<<”) transfieren *todas* las cadenas en una sola operación.

En el primer botón (>) ponemos el código para pasar la cadena seleccionada en la lista de la izquierda (que llamaremos ListBox1) a la de la derecha (ListBox2):



Ejemplo de trabajo con listas

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ListBox1.ItemIndex >= 0 then
  begin
    ListBox2.Items.Add(ListBox1.Items[ListBox1.ItemIndex]); //copia la cadena a la lista2
    ListBox1.Items.Delete(ListBox1.ItemIndex); //Borra el original
  end;
end;
```

la primera línea comprueba que haya efectivamente una cadena seleccionada; caso contrario obtenemos un error al utilizar ItemIndex como índice al “vector” Items. Si todo está bien, copiamos la cadena de una lista a la otra y luego borramos la cadena de la lista original.

Fácil, ¿no? les dejo a Uds. el resto de los procedimientos, para que se diviertan un poco.

NOTA: al realizar los bucles para trasladar todas las cadenas de una lista a la otra, tengan en cuenta que al borrar una cadena todas las que siguen se “corren” para no dejar espacios vacíos.

Ejercicio

Agregar al ejemplo anterior un mecanismo para poder insertar nuevas cadenas en cada lista, y otro para borrarlas (algún editor, botones...)

?

Propiedades referentes a la tipografía

Cuando se muestra un texto hay que especificar el *tipo* de caracteres a utilizar, o tipografía. Esta característica se almacena en los componentes de Delphi como una propiedad llamada **Font** (Fuente).

Esta propiedad tiene una característica particular: es, en sí misma, un objeto. Es una instancia de la clase Tfont, que contiene las propiedades necesarias para especificar completamente la tipografía a utilizar.

¿Y cómo accedemos a las propiedades del objeto Tfont de un componente particular, si ya el Inspector de Objetos está ocupado mostrando las propiedades del componente? Bueno, si miramos bien la línea de la propiedad **Font**, veremos que la palabra está precedida por un signo “más” (+Font). Es un indicativo de que la propiedad es compuesta, que tiene sus propias características que serán en este caso *Subpropiedades* del componente; se muestran como dependientes de la propiedad principal (Font), indentadas hacia la derecha. Se utiliza aquí la misma metáfora del “árbol” de directorios: el signo “+” indica que esta rama tiene características ocultas³.

Para abrir la rama y ver las propiedades de la fuente debemos hacer doble click *en la columna izquierda* del Inspector de Objetos, es decir, sobre la palabra “+Font”. Ahora el signo (+) cambia por (-) y se muestran las subpropiedades de la fuente. Veamos cada una de estas por separado.

- **Name:** el nombre de la tipografía. Es una variable de tipo **string** que debe tomar el nombre de una fuente instalada en Windows (Arial, Times New Roman, System, etc). Si el valor asignado no corresponde a una fuente instalada, se utilizará la más cercana... según la opinión de Windows.
- **Color:** el color de la fuente. Es una variable de tipo Entero Largo, en la que normalmente utilizaremos las constantes ya definidas por Delphi (que se muestran en una lista en el Inspector de Objetos): clGreen, clRed, etc.
- **Pitch:** indica si la tipografía es de ancho variable o fijo. Los valores posibles son fpDefault (depende del tipo de letra), fpVariable (ancho variable, las “i” ocupan menos espacio que las “m”) y fpFixed (Ancho fijo, todas las letras ocupan el mismo espacio). En las impresoras gráficas se utilizan preferentemente letras de ancho variable, por ser más elegantes y ocupar menos espacio; en cambio, en las viejas

³ En la introducción (capítulo 1) se muestra esta propiedad y la forma de acceder a las subpropiedades.

impresoras de agujas o margarita todas las letras tenían el mismo ancho.

- **Size:** entero. Expresa el tamaño de la fuente en puntos (unidad de medida tipográfica). No representa directamente la cantidad de pixels de pantalla que utilizará la fuente (esto está dado en la propiedad Height) sino el tamaño en puntos tipográficos que deseamos que tenga. Delphi calcula automáticamente el tamaño en pantalla (propiedad Height).
- **Height:** entero. Indica el tamaño de la fuente en la pantalla (tamaño en pixels). Se puede calcular en base a las propiedades Size y PixelsPerInch de la fuente.
- **PixelsPerInch:** entero. Afecta a las fuentes de impresora y no debería ser modificada. Es un factor de escala que garantiza que el tamaño de las fuentes impresas sea igual al de las fuentes de pantalla.

La fórmula que liga estas tres propiedades es la siguiente:

$$\text{Font.Size} = -\text{Font.Height} * 72 / \text{Font.PixelsPerInch}$$

Por lo tanto, al asignar un valor positivo a la propiedad Height se dará un valor negativo a la propiedad Size, y viceversa.

- **Style:** Indica el estilo de la tipografía, es decir, si es itálica, negrita, subrayada, etc. Es una propiedad de tipo conjunto, con los siguientes valores posibles:
 - *fsBold* - la tipografía aparece en negrita
 - *fsItalic* - itálica
 - *fsUnderline* - subrayada
 - *fsStrikeout* - tachada

Dado que es una propiedad de tipo conjunto, se puede asignar a la misma una lista de estilos (utilizando las constantes mostradas arriba). Esta lista puede tener cero o más elementos.

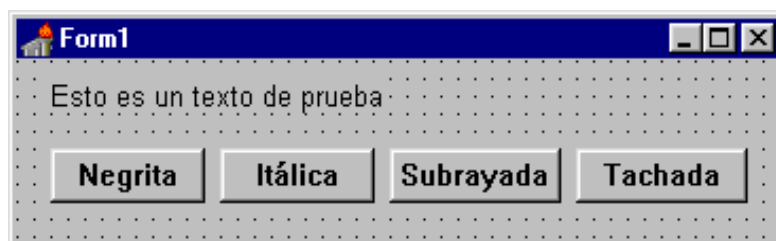
Como se vio en la introducción general -capítulo 1- en el Inspector de Objetos no se edita directamente este tipo de propiedades; se crea una nueva “rama” con cada una de los valores posibles como variables booleanas. Las características que se reflejarán en la tipografía serán las que tengan valor “TRUE”.

No obstante, desde un programa debemos acceder a la propiedad como una variable de tipo conjunto, utilizando para manipularla las diversas funciones y operadores definidas en Pascal.

Ejemplo 6

Colocaremos una etiqueta y varios botones, cada uno con una función diferente. Al presionar cada uno se alterará la forma de la letra de la etiqueta. El form principal es como el siguiente⁴:

⁴ Asegúrese que la etiqueta utiliza letra tipo “Arial”, para ver bien los cambios de estilo.



Como habrá adivinado, al presionar el botón “Negrita” la letra se volverá negrita; lo mismo para la forma itálica, subrayada y doble golpe. Veamos una primera codificación:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  label1.font.style:= [fsBold];
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Label1.Font.Style:= [fsItalic];
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  Label1.Font.Style:= [fsUnderline];
end;
```

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  Label1.Font.Style:= [fsStrikeOut];
end;
```

Ahora bien, este código funciona pero no aprovecha las posibilidades de los tipos conjunto de Pascal. Cada vez que se presiona un botón nuevo, el estilo anterior se pierde y es reemplazado completamente con el nuevo.

Una nueva versión del programa intercambia los estilos; si el estilo no estaba siendo utilizado se agrega, y si no se elimina:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if fsBold in label1.font.style then
    label1.font.style:= label1.font.style-[fsBold]
  else
    label1.font.style:= label1.font.style+[fsBold]
end;
```

```

procedure TForm1.Button2Click(Sender: TObject);
begin
  if fsItalic in label1.font.style then
    label1.font.style:= label1.font.style-[fsItalic]
  else
    label1.font.style:= label1.font.style+[fsItalic]
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  if fsUnderline in label1.font.style then
    label1.font.style:= label1.font.style-[fsUnderline]
  else
    label1.font.style:= label1.font.style+[fsUnderline]
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  if fsStrikeOut in label1.font.style then
    label1.font.style:= label1.font.style-[fsStrikeOut]
  else
    label1.font.style:= label1.font.style+[fsStrikeOut]
end;

```

Ahora si, se puede mostrar la etiqueta con cualquier combinación de estilos.

Ejemplo 7

Modificaremos un poco el ejemplo anterior, de manera que el usuario pueda especificar el *nombre* de la fuente y el *tamaño* de la misma. Para ello agregamos un par de editores, quedando el form como en la figura 14.

Cuando se presiona el botón “Cambiar...” el programa asignará los valores escritos en los editores a la fuente de la etiqueta superior (el estilo no cambia, se maneja en forma independiente con los botones superiores). Este es el código:

```

procedure TForm1.Button5Click(Sender: TObject);
begin
  label1.font.Name:= edit1.Text;
  label1.Font.Size:= StrToInt(edit2.text);
end;

```

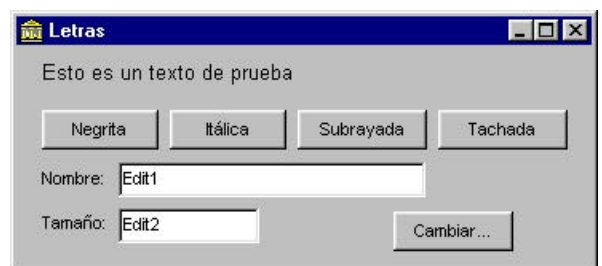


Figura 14: El form principal del ejemplo 5

Como vemos, es bastante simple modificar las características del texto en Delphi.

Componentes contenedores: los paneles

Los componentes de tipo *Panel* son nada más que contenedores de otros controles; cumplen una función más bien social, agrupando los controles con una función similar. Poseen también la posibilidad de mostrar un borde y un texto en su interior.

Enseguida veremos la relación especial que un panel guarda con los controles que contiene; mostremos antes cómo se utilizan.

Para indicar a Delphi deseamos que un control permanezca dentro de un panel -generalmente junto a otros que se refieren a un mismo tema dentro de la ventana-, debemos seleccionar el panel antes de colocar el control encima. De esta manera, la posición de los controles internos se referirá al panel contenedor y no al form (por ejemplo, la esquina superior izquierda de un panel se convierte en el centro de coordenadas para los controles que contiene). De esta manera, al mover un panel se mueven con él todos sus componentes.

Padres, hijos y otras cosas que nos da la vida

Entre los componentes en una aplicación se establece una relación de tipo filial. Podríamos decir que entre cualquier par de entes que convivan en un espacio de menos de 1000 puntos de ancho existirá una relación semejante; pero dejaremos estas disquisiciones filosóficas para los filósofos.

Las relaciones que nos interesan ahora se refieren a la posición relativa entre los componentes. Dicho rápidamente: todos los componentes en Windows están contenidos dentro de otros -en última instancia, dentro del escritorio del sistema. Por ejemplo, cada ventana contiene a sus controles (botones, etiquetas, etc) mientras que la ventana misma está contenida en otra ventana o bien en el escritorio.

El componente que contiene a otro *visualmente* se denomina *Padre* (Parent), y el control contenido es el *Control Hijo* (Child Control).

Todos los componentes visibles tienen la propiedad **Parent** para señalar a su contenedor en la pantalla (esta propiedad no es visible en el Inspector de Objetos). Esto es un requerimiento del sistema Windows para administrar los distintos controles y ventanas que coexisten en la pantalla.

Padres vs. Dueños

Hay otro tipo de relación entre los componentes de Delphi, específica de este lenguaje: la relación de *pertenencia*. Esta relación permite que al destruir un objeto se destruyan automáticamente los que le pertenecen. Por ejemplo, en una ventana cualquiera los controles son poseídos por el form; al destruirse éste

y liberarse los recursos ocupados, se destruyen también los controles. *Todos* los componentes de Delphi (no solamente los visibles) poseen una propiedad que apunta al componente dueño, llamada **Owner**. Esta propiedad no se ve en el Inspector de Objetos.

No hay que confundir las propiedades *Owner* y *Parent*. La primera se refiere al **objeto** que posee a otro y se utiliza para *liberar recursos* automáticamente; mientras que la segunda indica la **ventana** o **control** que visualmente contiene a otra ventana o control, y se utiliza para *redibujar* automáticamente los controles.

Por ejemplo, cuando ponemos controles dentro de un panel que está a su vez sobre un form, el Padre de los controles (Parent) es el panel, mientras que el Dueño de los mismos (Owner) es el form. El form es también Padre y Dueño del panel.

Otras propiedades comunes

- **Visible**

Es una propiedad de tipo lógico (boolean). Si tiene valor verdadero (TRUE), el componente está visible. Se utiliza para mostrar algunos componentes en ocasiones especiales, por ejemplo cuando se cumple alguna condición. Todos los componentes visuales tienen esta propiedad -inclusive los componentes de campo que veremos en la parte de Bases de Datos.

- **Enabled**

Es otra propiedad de tipo lógico. Si tiene valor verdadero, el componente está habilitado para recibir la atención del usuario; si el componente está deshabilitado, no responde a los eventos. Generalmente se puede ver el estado deshabilitado por un cambio en la presentación del componente: es muy común mostrarlo *grisado* o en colores más pálidos para indicar este estado.

- **Color**

Es una variable de tipo Entero Largo que contiene el valor de color a utilizar para el fondo del componente. Generalmente se indica con una de las constantes predefinidas en Delphi, de las que se puede ver una lista en la columna derecha del Inspector de Objetos. No obstante, también es posible especificar un color que no esté en esta lista (aunque no recomendable): haciendo doble click en la columna derecha del Inspector de Objetos aparecerá el cuadro de diálogo estándar de selección de color de Windows. Aquí podemos elegir cualquiera de los colores ya definidos o crear uno nuevo.

- **Hint**

La propiedad Hint es una variable de tipo String que contiene un texto que será mostrado en un pequeño rectángulo amarillo que aparecerá cerca del componente pasado cierto tiempo que el puntero está sobre el mismo (ayuda emergente). Se utiliza para dar al usuario una ayuda rápida y simple sobre la acción o cometido de cada control. Por defecto no se verá esta ayuda aunque haya un valor en la propiedad; hay que habilitar esta característica con la propiedad siguiente.

- **ShowHint**

Es una variable de tipo lógico que indica si se debe mostrar o no la ayuda emergente (Hint). Está relacionada con las propiedades Hint y ParentShowHint. Valor por defecto: FALSE.

- **ParentShowHint**

Variable de tipo lógico. Si tiene valor verdadero indica que el valor para la propiedad ShowHint se debe tomar del componente padre (Parent). Se pone automáticamente en FALSE cuando cambiamos la propiedad ShowHint. Valor por defecto: TRUE.

Se utiliza para lograr que todos los componentes contenidos por otro mantengan un comportamiento consistente.

- **ParentFont**

Variable de tipo lógico. Si tiene valor verdadero indica que el valor para la propiedad Font se debe tomar del componente padre (Parent). Se pone automáticamente a FALSE si cambiamos la propiedad Font. Valor por defecto: TRUE.

- **ParentColor**

Igual que las anteriores, para el color del componente. Por defecto: TRUE.

- **Ctl3D**

Variable lógica que indica si el control tendrá un aspecto tridimensional o no. Por defecto: TRUE.

Además de estas propiedades que están en casi todos los componentes de uso común, cada uno tiene sus particularidades. Antes de estudiar en detalle cada uno de los componentes, veremos los eventos más usados.

Eventos

Los eventos son, como dijimos en la introducción, cosas que suceden. Ya sea que se presiona un botón del ratón, se redimensiona una ventana o nos tocan el timbre, Delphi procesa estos sucesos (bueno, tal vez el último no) y si hemos definido una respuesta para el mismo, la activa automáticamente.

El código que escribimos en Delphi se ejecuta mayormente en respuesta a eventos.

No obstante, no todos los componentes pueden responder a todos los eventos. Por ejemplo, dado que las etiquetas no pueden ser editadas por el usuario final, estos componentes no responderán a las pulsaciones de

teclas. Veremos ahora los eventos más comunes.

El proceso de crear un procedimiento de respuesta a un evento es el mismo que utilizamos al principio de esta sección, cuando creamos un procedimiento de respuesta al evento `OnClick` de un botón. Los pasos generales son:

- Seleccionar el componente en el que se producirá el evento
- En el Inspector de Objetos, mostrar la página de eventos
- Buscar el evento que deseamos
- En la columna de la derecha (que inicialmente está en blanco) escribir un nombre para el procedimiento y presionar `<Enter>` o hacer “Doble Click”: Delphi escribirá un nombre automáticamente. El cursor pasa a la ventana de edición, entre las palabras reservadas `Begin` y `End`, listo para escribir nuestro código.

Por último, una advertencia: es muy común equivocarse en el componente que señalamos para responder a determinado evento. Por ejemplo, queremos que un determinado código se ejecute en respuesta al click del ratón sobre un botón; vamos al Inspector de Objetos y buscamos el evento, escribimos el código y todo bárbaro, pero al ejecutar nos damos cuenta que no responde a la pulsación del ratón sobre el botón sino sobre el form. ¿Qué pasó? Pues que inadvertidamente hemos tomado el evento `OnClick del form` en lugar del botón. Siempre antes de escribir un procedimiento de estos hay que comprobar que tenemos seleccionado el componente adecuado, mirando la parte superior del Inspector de Objetos (en el `ComboBox` aparece el nombre del componente seleccionado).

Sobre la administración de los procedimientos de respuesta a eventos

Una regla de oro: *lo que Delphi escribe, Delphi lo borra*. Olvidar esta regla puede traer dolores de cabeza al más guapo.

Cuando se crea un procedimiento de respuesta a un evento Delphi escribe unas cuantas cosas, no solamente la cabecera del procedimiento. Para eliminar un procedimiento de estos, hay que eliminar todas esas cosas. Se puede hacer a mano; estudiaremos todos estos cambios cuando entremos de lleno al tema de los objetos. No obstante, es bastante engorroso. Por eso los programadores de Borland en su infinita sabiduría han inculcado a Delphi el conocimiento necesario para hacerlo solo.

Para borrar un procedimiento de respuesta a un evento, borramos el *cuerpo* del procedimiento y cualquier declaración de variables locales que hayamos escrito. Dejamos las cosas tal como Delphi las escribió: la cabecera y las sentencias `Begin..End`. La próxima vez que compilemos el proyecto este procedimiento será dado de baja limpiamente.

Tipos de procedimientos y parámetros

Al responder a un evento, es útil conocer algunos datos; por ejemplo, cuando detectamos que se presionó una tecla es interesante saber qué tecla fue. Asimismo, a veces es importante saber qué componente es el que detectó el evento (dado que es posible asignar el mismo procedimiento a eventos distintos de diferentes componentes). Delphi resuelve esta dificultad con facilidad utilizando parámetros en los procedimientos de respuesta a eventos.

Así por ejemplo, todos los procedimientos de respuesta a eventos tienen un parámetro llamado *Sender* que indica cuál fue el componente que llamó al procedimiento -es decir, el que detectó el evento; el procedimiento creado en respuesta a un evento `OnKeyPress` agrega un parámetro de tipo `Char` que contiene la letra de la tecla presionada, y así sucesivamente. Al llamar al procedimiento, Delphi llena automáticamente estos parámetros con los valores correspondientes y así podemos nosotros acceder a los

mismos.

Es importante entonces conocer además de los eventos disponibles, la información que éstos nos brindan. Por ejemplo, si queremos detectar las coordenadas donde se hace click con el ratón no podremos utilizar el evento `OnClick` (que sería la primera idea que nos viene a la mente), ya que este evento no proporciona ese dato. Tendremos que utilizar `OnMouseDown` o, mejor todavía, `OnMouseUp`.

Eventos del ratón

OnClick

Se produce al presionar y soltar el botón principal del ratón. Este evento existe en prácticamente todos los componentes visibles, dado que siempre es posible presionar el botón principal del ratón sobre ellos.

OnDblClick

Se produce al presionar y soltar el botón principal del ratón rápidamente dos veces sobre el mismo lugar (aproximadamente, con un margen pequeño de movimiento entre cada pulsación). La velocidad de pulsación necesaria para que el sistema considere los dos clicks como uno solo doble se configura en el Panel de Control de Windows.

Generalmente se asigna a este evento la acción por defecto de una caja de diálogo, de manera que el usuario experimentado pueda acelerar la ejecución.

OnMouseDown

Se produce al presionar uno cualquiera de los botones del ratón. En los parámetros del procedimiento se indica el botón presionado, la posición en que se encontraba el puntero cuando se produjo el evento, y si había alguna tecla especial (`Shift`, `Control`, `Alt`) presionada en ese momento.

OnMouseUp

Se produce al soltar uno cualquiera de los botones del ratón. El procedimiento ofrece los mismos parámetros que `OnMouseDown`.

Estos tres eventos se producen en sucesión cuando se presiona el botón principal del ratón. Primero se genera `OnMouseDown`, y al soltar el botón se producen `OnClick` y `OnMouseUp`.

OnMouseMove

Se produce cada vez que se mueve el puntero del ratón sobre la superficie considerada. Los parámetros incluyen las coordenadas del puntero al momento de producirse el evento y las teclas especiales que se encontraban presionadas.

Ejemplo 8

Se trata de un programa que muestre en la barra de título de la ventana principal, las coordenadas del ratón al moverse sobre ella.

El evento a programar debe pertenecer al form, y generarse cada vez que se mueve el puntero del ratón. Este evento existe y se llama OnMouseMove. Las coordenadas vienen dadas en los parámetros X, Y del procedimiento de respuesta. Una posible versión sería la siguiente:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  Caption:= '('+IntToStr(X)+', '+IntToStr(Y)+' )';
end;
```

Ejercicio

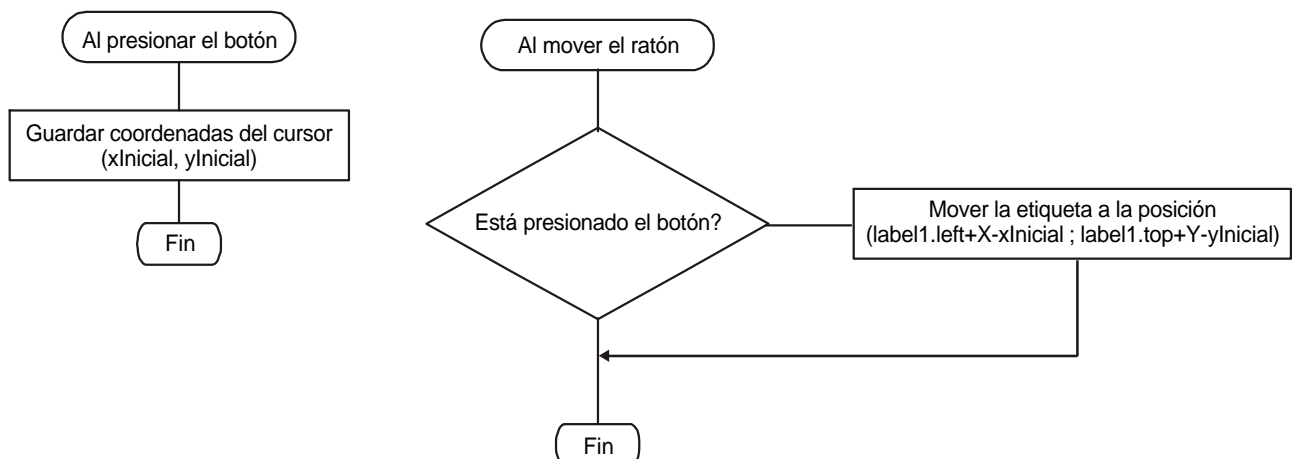
Coloque un botón sobre el form. Ahora pase por encima del botón con el puntero del ratón. ¿Se ven las coordenadas? ¿Cómo lo solucionarías?

?

Ejemplo 9

Escribiremos un programa que permita “arrastrar” una etiqueta. Por arrastrar entendemos que el usuario presiona el botón principal del mouse sobre la etiqueta, lo mantiene apretado mientras mueve el puntero (la etiqueta sigue al puntero) y al soltar el botón la etiqueta queda fija.

Una forma de plantearlo sería, en diagrama de bloques:



Un gráfico nos ayudará a comprender la fórmula de cálculo de las coordenadas. La clave está en darse cuenta que las coordenadas X, Y que nos llegan en el evento MouseMove son relativas a la esquina superior

izquierda del Label *cuando estaba en la posición anterior* al movimiento (Fig. 16)

Como podemos ver del gráfico, necesitamos mantener las coordenadas en las que se presionó el ratón cuando comenzó el arrastre. Para ello declararemos dos variables de tipo entero llamadas *Xinicial* e *Yinicial* en la sección de declaración de variables, antes del inicio de la sección de implementación.

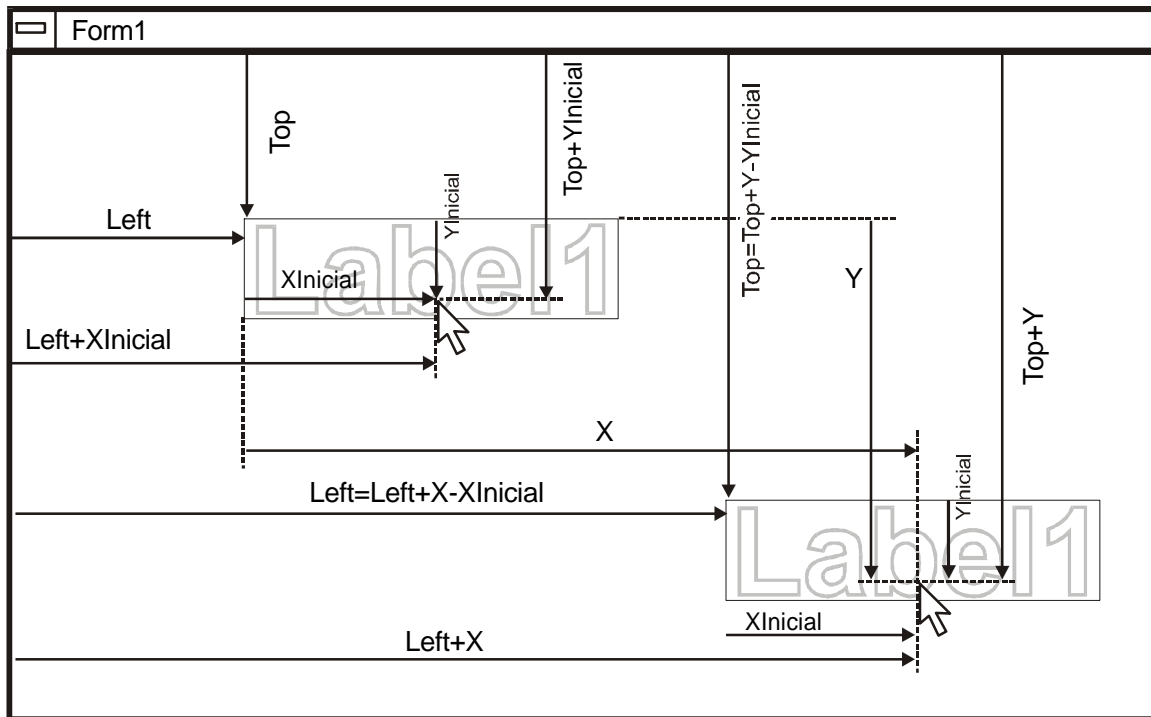


Figura 16: Variables que intervienen en el ejemplo de arrastre de una etiqueta

Los dos procedimientos que escribimos responden a los eventos *OnMouseDown* y *OnMouseMove* de la *Label*:

```
var
  Form1: TForm1;
  xInicial, yInicial: integer;

implementation

{$R *.DFM}

procedure TForm1.Label1MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  if ssLeft in Shift then begin
    Label1.left := x + Label1.left - xInicial;
    Label1.Top := y + Label1.Top - yInicial;
  end;
end;
```

```
end;  
end;  
  
procedure TForm1.Label1MouseDown(Sender: TObject; Button: TMouseButton;  
  Shift: TShiftState; X, Y: Integer);  
begin  
  xInicial:= x;  
  yInicial:= y;  
end;
```

Existen otros eventos relacionados con el ratón: aquellos que permiten la operación de *Arrastrar* y *Soltar* (Drag&Drop); dado que esta operación involucra otras acciones, la veremos por separado más adelante.

Eventos de teclado

OnKeyPress

Se produce al presionar y soltar una tecla que corresponda a un caracter simple; no se producirá por ejemplo en respuesta a una tecla de función o alguna de las teclas modificadoras: Shift, Control, Alt.

Parámetros:

- Key: tipo Char, contiene el código ASCII de la tecla presionada.

OnKeyDown

Se produce al presionar una tecla. Ofrece los siguientes parámetros (además de Sender):

- Key: tipo Word. Contiene el *código virtual* de la tecla presionada. Para ver los códigos virtuales busque en la ayuda el tópico *virtual key codes*.
- Shift: es un conjunto de constantes que indican las teclas especiales modificadoras que estaban presionadas al producirse el evento, así como los botones del ratón presionados. Los valores posibles son
 - *ssShift* cualquiera de las teclas Shift está presionada
 - *ssCtrl* tecla Control presionada
 - *ssAlt* tecla Alt presionada
 - *ssLeft* el botón izquierdo del ratón está presionado
 - *ssMiddle* ídem para el botón del medio
 - *ssRight* lo mismo para el botón derecho
 - *ssDouble* los botones izquierdo y derecho del ratón están ambos presionados

Para determinar por ejemplo si estaba presionada la tecla <Ctrl> utilizamos el operador IN:

```
if ssCtrl in Shift then ...
```

Notemos que el evento **OnKeyDown** se repetirá con la velocidad de repetición de teclado configurada en el panel de control.

Ejemplo 10

Realizaremos una aplicación que nos permita mover una “nave espacial” a través de la ventana, utilizando el teclado.

Como nave espacial utilizaremos componentes Shape. La nave se llamará por supuesto “Enterprise” (Fig. ???)

Los componentes Shape no reciben el foco del teclado. Entonces, ¿dónde ponemos el código de respuesta a la presión de una tecla? ¿Quién tiene el foco de teclado cuando no hay nada sobre la ventana que pueda recibirlo? Pues... la ventana misma! Por lo tanto, escribiremos nuestro código en respuesta al evento **OnKeyDown** del form principal.

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  case key of
    vk_Left: begin
      Shape1.Left:= Shape1.Left-1;
      Shape2.Left:= Shape2.Left-1;
      Shape3.Left:= Shape3.Left-1;
    end;
    vk_Right: begin
      Shape1.Left:= Shape1.Left+1;
      Shape2.Left:= Shape2.Left+1;
      Shape3.Left:= Shape3.Left+1;
    end;
    vk_Up: begin
      Shape1.Top:= Shape1.Top-1;
      Shape2.Top:= Shape2.Top-1;
      Shape3.Top:= Shape3.Top-1;
    end;
    vk_Down: begin
      Shape1.Top:= Shape1.Top+1;
      Shape2.Top:= Shape2.Top+1;
      Shape3.Top:= Shape3.Top+1;
    end;
  end;
end;
```

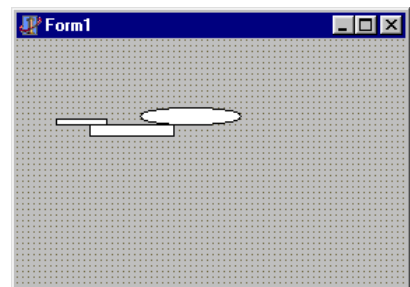


Figura 17: la Enterprise

```
end;  
end;
```

Notemos la utilización de las constantes `vk_XXXX` que representan cada tecla; la lista completa se puede ver en la ayuda.

Ejercicio
Modificar el programa anterior para que la nave no pueda salirse de los límites de la ventana

?

OnKeyUp

Este evento se produce cuando se suelta una tecla. Tiene los mismos parámetros que el evento `OnKeyDown`.

Estos tres eventos se producen en sucesión: cuando presionamos una tecla se produce `OnKeyDown`; al soltarla se genera un evento `KeyPress` (si corresponde) y enseguida `KeyUp`.

El foco de atención

En Windows hay controles que aceptan eventos de teclado y otros que no; normalmente tendremos varios del primer tipo en una sola ventana. Por lo tanto, hay que definir de alguna manera a quién van a parar los eventos de teclado.

Sólo un control a la vez puede recibir estos eventos: se dice que este control tiene el *foco* de atención por parte del sistema.

Generalmente es muy fácil reconocer visualmente al control que tiene el foco: en los controles que aceptan texto aparecerá el cursor parpadeante, y los demás se muestran rodeados por una línea de puntos.

Cuando el foco de atención pasa de un control a otro, se producen dos eventos: `OnEnter` y `OnExit`

OnEnter

Se genera en un control cuando éste recibe el foco. No tiene otros parámetros aparte de `Sender`.

OnExit

Se genera en un control cuando el mismo pierde el foco. No tiene otros parámetros aparte de Sender.

Cambia, todo cambia...

Hay otro evento que se produce en varios componentes al momento de cambiar su valor, como por ejemplo los editores al cambiar el texto o las barras de desplazamiento al cambiar la posición del indicador. Es llamado, previsiblemente, OnChange. No tiene otros parámetros además de Sender.

Ejemplo

Se desea permitir al usuario cambiar el texto de la barra de título de una ventana, escribiéndolo en un editor.

Escribiremos el programa de dos maneras:

1) con un botón que copie el texto del editor al título de la ventana

Creamos un procedimiento de respuesta al evento OnClick del botón, en el cual copiamos el texto:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Caption:= edit1.text;
end;
```

2) el título cambia a medida que se escribe en el editor

Ahora tomamos el evento OnChange del editor, y escribimos el mismo código:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
    Caption:= edit1.text;
end;
```

Eventos del form

La clase TForm, de la que descienden todos las ventanas de Delphi, define algunos eventos especiales que no se encuentran en otros componentes.

OnActivate - OnDeactivate

Se generan cuando se activa y desactiva la ventana respectivamente, es decir cuando recibe o pierde el foco de atención.

Por ejemplo, si la ventana está cubierta por otra no tiene el foco de atención; si pulsamos el ratón sobre ella la traemos al frente, le damos el foco y se produce el evento OnActivate. Si luego la cerramos o traemos otra ventana al frente, se pierde el foco y se genera el evento OnDeactivate.

OnCloseQuery

Se genera cuando se va a cerrar un form, es una manera de “pedir permiso” al programa para cerrar la ventana. Es muy común utilizarla para preguntar al usuario si se deben grabar datos que se hayan modificado, y según su respuesta permitir o no el cierre de la ventana.

El permiso para cerrar la ventana se da cambiando el valor de un parámetro de tipo lógico llamado **CanClose**. Si le damos valor verdadero (TRUE), la ventana se cierra. Este es el valor por defecto que toma la variable si no le asignamos otro.

Ejemplo 11

Tenemos una aplicación que mantiene datos del usuario (por ejemplo un procesador de textos). Cuando se quiere cerrar la misma, si los datos han sido modificados -el programa pone una variable tipo lógica en TRUE; digamos que se llama *Modificado*- se debe dar aviso al usuario y ofrecer la opción de Grabar y Salir, Salir sin Grabar o Cancelar la operación de cerrado.

El lugar ideal es el evento OnCloseQuery de la ventana principal; el código podría ser como el siguiente:

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
var
  Respuesta: word;
begin
  if Modificado then begin
    respuesta:= MessageDlg('Los datos han sido modificados'#13'¿Desea grabarlos?',
      mtConfirmation,[mbYes,mbNo,mbCancel],0);
    case respuesta of
      mrYes: {Grabar los datos}
      mrCancel: CanClose:= false;
    end;
  end;
end;
```

En este código se presenta al usuario un cuadro de diálogo que pregunta si se desean grabar los datos antes de cerrar. Si el usuario presiona el botón “Si”, se graban los datos y sigue el proceso de cerrado normal. Si el usuario presionó el botón “No”, la ventana se cierra y los datos se pierden. También puede contestar con el botón “Cancelar”, con lo cual se corta el proceso de cerrado de la ventana y el usuario puede seguir trabajando.

Como el valor por defecto que toma el parámetro CanClose es TRUE, no hace falta asignarlo

específicamente por ejemplo en el caso de haberse seleccionado el botón “No”.

OnClose

Este evento es la continuación del anterior: cuando se cierra una ventana se genera primero `OnCloseQuery`, con el que le damos permiso a la ventana para cerrarse; inmediatamente se genera `OnClose` para darnos la oportunidad de especificar el acción a tomar con el *objeto* de la ventana, es decir, con la estructura interna que está en la memoria.

La acción deseada se especifica asignando un valor al parámetro **Action**, que puede ser una de las constantes siguientes:

- `caNone`: indicamos que no se permite cerrarse a la ventana, por lo tanto es equivalente a poner `CanClose=FALSE` en el evento `OnCloseQuery`
- `caHide`: el form no se cierra, sólo se oculta a la vista. Todavía tenemos acceso a sus propiedades y métodos.
- `caFree`: el form se cierra y el objeto se destruye, es decir, se liberan todos los recursos que pueda haber ocupado (como la memoria).
- `caMinimize`: el form se minimiza, no se cierra. Es el comportamiento por defecto de las ventanas tipo `MDIChild`.

La acción por defecto de un form común es `caHide`, por lo que al cerrar una ventana -salvo que sea la principal, que cierra la aplicación completa- en realidad la estamos ocultando. Todavía podemos acceder a sus controles y propiedades.

La acción `caFree` es muy utilizada cuando el form es creado en tiempo de ejecución; de esta manera, nos aseguramos que al cerrar la ventana se libera la memoria y otros recursos ocupados.

OnCreate - OnDestroy

El evento `OnCreate` se genera al crearse una ventana. Es el lugar ideal para dar valores iniciales a otros controles, como por ejemplo llenar un `Listbox` o un `ComboBox`, poner valores iniciales en los editores, crear estructuras propias, etc.

El evento `OnDestroy` se produce al destruirse una ventana; aquí deben destruirse todas las estructuras y variables propias que fueron creadas en el evento `OnCreate`.

NOTA: el evento `OnCreate` se produce siempre al crearse la ventana, antes de aparecer en la pantalla. No obstante, hay que tener en cuenta que la acción por defecto al cerrar un form es su ocultación (vea arriba el evento `OnClose`) por lo que en ese caso no se producirá `OnDestroy` al cerrar una ventana. Si en el evento `OnClose` asignamos el valor `caFree` al parámetro `Action`, entonces sí se producirá `OnDestroy`.

OnShow - OnHide

Estos eventos se producen justo antes que se muestre o se oculte un form, respectivamente. En el momento de mostrar una ventana (llamando a los métodos `Show` o `ShowModal`, que veremos luego) se produce `OnShow`. No obstante, *no* se produce cuando la ventana vuelve a mostrarse luego de quedar tapada por otra (en este caso, se produciría `OnActivate`).

Como regla práctica: OnShow se produce cuando se muestra por primera vez la ventana, o cuando sale de una ocultación previa. OnHide se produce cuando ocultamos la ventana (si recordamos que al cerrar una ventana en realidad se oculta, veremos que también al “cerrar” una ventana se produce OnHide).

OnPaint

Este evento se produce cada vez que se necesita redibujar una ventana. Se utiliza para dibujar algo especial en la superficie de la ventana.

Bajo Windows, cada ventana es responsable por su propio contenido. Esto implica también que cada ventana se debe redibujar cuando sea necesario. Por ejemplo, cuando movemos una ventana que estaba tapando una parte de otra, es necesario redibujar la parte que se reveló de la ventana inferior. Entonces Windows envía un mensaje a la ventana inferior indicando que es necesario redibujar la superficie; Delphi intercepta este mensaje y lo transforma en un evento OnPaint.

Notemos que al redibujar la superficie de una ventana no es necesario hacerlo con los controles que están sobre la misma; esto se hace automáticamente.

Cuando veamos la sección de gráficos utilizaremos profusamente este evento.

OnResize

Este evento se produce cada vez que cambia el tamaño de la ventana. Se utiliza mayormente para redimensionar los controles de forma acorde con el nuevo tamaño, de manera que sigan manteniendo las proporciones.

Ejercicio

Modifique el ejemplo 2 para que el ajuste de posiciones y tamaños se realice automáticamente al cambiar el tamaño de la ventana

?


El menú, por favor...

Todos los que trabajamos con Windows conocemos los menús; especialmente si trabajamos con la nueva interface de Windows 95, donde el acceso principal a los comandos del sistema se realiza a través de un menú desplegable.

El uso de un menú como vía de acceso a los comandos es una norma de diseño emanada de un estudio sobre usabilidad; es una forma práctica de tener disponibles una cantidad elevada de comandos en un pequeño espacio.

Existen dos tipos principales de menús: el principal (la *barra* de menús), que se puede ver inmediatamente debajo de la línea de título de la ventana; y el menú *contextual*, que se mantiene oculto y sólo se despliega en respuesta a una acción del usuario, normalmente el botón derecho del ratón.

El menú principal

El menú principal (*Main Menu* en inglés) contiene todos los comandos disponibles en la aplicación, y se ubica en la parte superior de la ventana debajo del título. En Delphi disponemos de un componente para crear estos menús: previsiblemente, la clase se denomina *TMainMenu*. Este componente se encuentra normalmente en la página *Standard* de la paleta de componentes ().

Este componente es el primero que encontramos que no se comporta normalmente en el editor de formularios: una vez colocado en una ficha, no se puede cambiar su tamaño. Para crear los desplegables con los comandos debemos hacer *doble click* sobre el icono que queda en la ficha. Se abre entonces el **Editor de menús**.

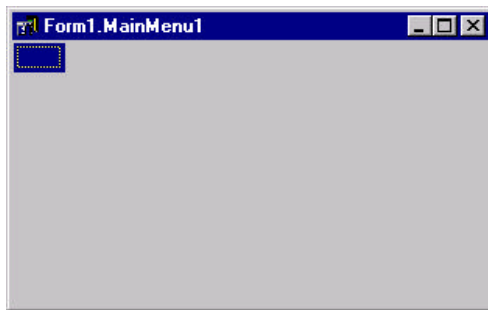


Figura 19: el editor de menús antes de empezar

Como se habrán imaginado, cada línea de un menú (llamado *ítem*) es un objeto, que tiene propiedades y eventos que podemos ver en el Inspector de Objetos. Una de estas propiedades es **Caption**, que contiene el texto a mostrar en ese ítem. Por ahora esa es la única propiedad que nos interesa, ya que el nombre (la propiedad **Name**) se asigna automáticamente.

En el editor de menús veremos un lugar vacío para cada ítem; al escribir el texto del **Caption** éste se crea automáticamente y se agrega otro espacio vacío. Siempre queda un espacio vacío al pie de cada menú desplegable y a la derecha del último ítem de la

línea superior.

Bien, es muy fácil crear un menú bonito; pero ¿cómo hacemos que funcione, que responda a la elección del usuario? ¡Pues con un evento, claro! Cada ítem de menú tiene un solo evento, para reflejar la única opción que se puede realizar sobre ellos: seleccionarlos, ya sea con el botón principal del ratón o presionando <Enter> cuando se encuentra resaltada.

Ejemplo 12

Vamos a un ejemplo. Haremos un programa simple para editar archivos de texto, con las opciones comunes en un menú principal; luego agregaremos otros refinamientos a la interface, como una barra de herramientas y una línea de estado. Lo llamaremos *Gutenberg*.

La ficha principal tiene sólo dos componentes: un Memo y un *MainMenu*. El espacio de escritura es el memo, mientras que los comandos estarán en el menú principal (el componente Memo es un componente común de Windows, que de hecho es lo que forma el Bloc de Notas). Hagamos que el memo ocupe todo el espacio disponible en la ventana poniendo la propiedad **Align** en *alClient*. Hasta que no escribamos los ítems, el menú no aparecerá en la ventana. En la figura 20 se ve el aspecto de la ventana hasta este punto.

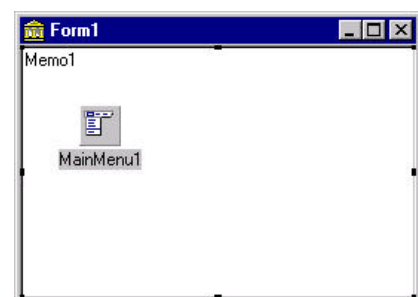


Figura 20: la semilla de Gutenberg

Ahora empezamos a escribir el menú. Hacemos doble click en el icono para abrir el Editor de Menús, y creamos los siguientes menús:

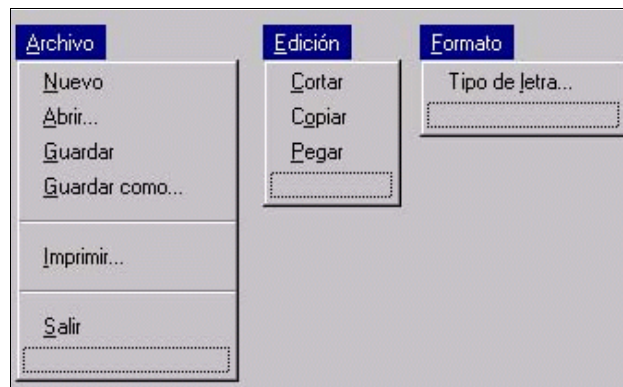


Figura 21: Menús de Gutenberg

Notemos en el gráfico que los ítems tienen una letra subrayada: es la letra de acceso directo o atajo de teclado, que se utiliza para acceder a los comandos mediante el teclado. Los ítems superiores se acceden con la tecla <Alt> más la letra subrayada, mientras que al desplegarse un menú sólo es necesaria la letra subrayada.

El subrayado se consigue poniendo el signo de libra (&) antes de la letra en cuestión. Por ejemplo, en la figura anterior el Caption del primer menú es “&Archivo”.

Hay que cuidar que no se repitan las letras elegidas como atajos de teclado dentro de un mismo nivel; por ejemplo, en la barra superior o en un mismo menú desplegable. ¿Qué sucederá en el menú “Archivo” al presionar la letra “A” una vez desplegado? ¿Y si presionamos la “G”?

Los comandos comunes como los de esta aplicación siguen normalmente un orden, *por convención*, que es el que mostramos en la figura. No es obligatorio seguirlo, pero sí muy recomendable porque el usuario se acostumbra a verlos en esas posiciones relativas y luego de un tiempo ya no lee los textos.

Notemos que en la ficha se crea espacio para la barra del menú, y que éste funciona correctamente incluso en tiempo de diseño; sólo que si seleccionamos ahora una opción se inserta un procedimiento de respuesta para el evento OnClick!

Ahora veamos como responder al comando “Salir”, que es sencillo de implementar.

Para ello podemos

- ? Seleccionar la opción en el Editor de menús, ir a la página de Eventos en el Inspector de Objetos y hacer doble click en el evento OnClick.
- ? Seleccionar la opción en el menú de la ficha, como si estuviéramos utilizando la aplicación terminada.

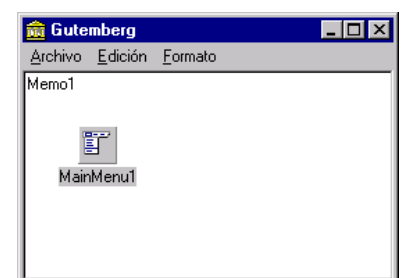


Figura 22: la barra de menú ya creada

En cualquier caso, se crea el procedimiento de respuesta al evento OnClick de esa opción de menú. En él escribimos como ya sabemos, la llamada al método Close:

```
procedure TForm1.Salir1Click(Sender: TObject);
begin
  Close;
end;
```


De la misma manera crearemos los procedimientos de respuesta a las demás opciones del menú.

Comencemos con las opciones del menú Edición. La clase Tmemo tiene métodos especiales para trabajar con el portapapeles (*clipboard*), como por ejemplo “Cortar al clipboard”, la opción “Cortar” del menú **Edición**:

```
procedure TForm1.Cortar1Click(Sender: TObject);
begin
    Memo1.CutToClipboard;
end;
```

Dejamos al lector descubrir en la ayuda y codificar las otras opciones del menú de Edición (Copiar y Pegar).

En la sección siguiente desarrollaremos el Bloc de Notas completo, con todas las opciones habilitadas. Mientras tanto, sigamos repasando las distintas partes de la interface.

Barras de herramientas

Las barras de herramientas son conjuntos de botones que permiten acceder rápidamente con el ratón a los comandos más comunes de un programa. Han tomado por asalto al mundo de Windows; recuerdo cuando empezaron a aparecer, eran toda una novedad que hacía pensar que la empresa que las usaba era muy de avanzada. Actualmente, son muy raros los programas que no incorporan alguna forma de estas “botoneras”.

Delphi acompaña a los tiempos acercando la posibilidad de hacer barras de herramientas de distintos tipos, desde las más simples -esencialmente un panel con botones dentro- hasta las más sofisticadas, con dibujos de fondo y botones planos que se levantan cuando pasamos el cursor del ratón sobre ellos. Veremos ahora como crear las más simples, y dejaremos la parte avanzada para más tarde.

El objetivo es llegar a la ventana siguiente:

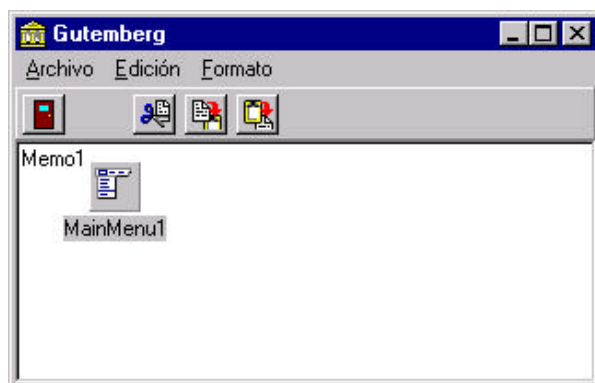


Figura 23: Gutenberg con barra de herramientas

La barra de herramientas que usamos en la imagen anterior es la más simple que podemos pensar: un simple

panel con botones adentro. Primero colocamos el panel en la ficha, y ponemos su propiedad **Align** en *alTop* para que se “pegue” al menú principal y desplace el memo hacia abajo. Le damos una altura adecuada (los botones que usaremos tienen un tamaño por defecto de 23 pixels de ancho y 22 de alto) y borramos el **Caption**.

A continuación colocamos los botones: utilizaremos unos botones especiales para barras de herramientas, llamados **SpeedButtons**, que se encuentran en la página **Additional** de la paleta de componentes. Estos botones *no pueden recibir el foco de atención del teclado*, por lo que sólo podrán accederse con el ratón. Además del texto (como siempre en la propiedad **Caption**) pueden contener un gráfico pequeño, que es lo que normalmente se muestra. La propiedad que almacena el gráfico se llama **Glyph**⁵.

Ahora todo está más lindo, pero todavía nos falta lo importante que es el *comportamiento* de los botones. Como todos los botones, los **SpeedButtons** tienen un evento **OnClick**; podemos verlo en el Inspector de Objetos. Dado que los botones ejecutan algún comando de los que están en los menús, podríamos crear un procedimiento de la manera usual y escribir nuevamente las mismas líneas que en el menú (o copiar y pegar el código).

Pero hay una forma mucho más simple. Veámoslo con el primer botón, el de **Salir**.

Una vez posicionado el botón en la barra, vamos al Inspector de Objetos y en el evento **OnClick** *desplegamos la lista del editor* (fig. 24). Aquí aparecen los nombres de los procedimientos que están ya creados y *son compatibles* con este evento particular. Podemos seleccionar cualquiera de ellos, y automáticamente ese procedimiento será llamado también en respuesta a este evento. En nuestro ejemplo seleccionamos el procedimiento **Salir1Click**.

¡Y ya está! No hizo falta escribir absolutamente nada.

Como siempre, la parte divertida queda para Uds: asignar los procedimientos correspondientes a los otros botones.

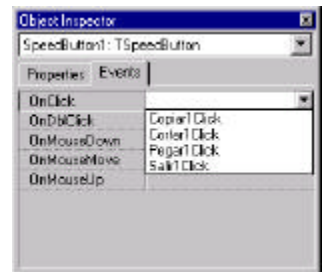


Figura 24: reutilizar un procedimiento de respuesta a un evento

¡Manos al ratón!

Ejercicios

- 1) Agregue un grupo de dos botones radiales a la ficha del ejemplo 1, con los cuales el usuario podrá decidir el sentido de giro del botón
- 2) Modifique el programa anterior para que el botón se “deslice” en lugar de saltar de una posición a la siguiente.
- 3) Modificar el ejemplo 10 de la nave para que al desaparecer por un borde de la pantalla, la nave aparezca por el otro (salto por el hiperespacio)

⁵ La instalación normal de Delphi incluye una serie bastante extensa de gráficos para usar en botones, en el directorio “Borland Shared\Images\Buttons”. Además podemos crear nuestros propios dibujos.

- 4) Modificar el programa anterior para que la nave pueda disparar torpedos de fotones, al presionar la barra espaciadora por ejemplo

Técnicas de interfaces

Veremos aquí algunas técnicas muy útiles y necesarias a la hora de programar aplicaciones: utilización de más de un form tanto en una interfaz SDI como en una MDI, cuadros de diálogo estándar, crear una pantalla de presentación para nuestro programa, etc.

Creación de ventanas secundarias

Las aplicaciones Windows normalmente tendrán varias ventanas. Por ejemplo, cuando en una aplicación de uso común como un procesador de textos pedimos Abrir un archivo se nos muestra una ventana secundaria donde podemos ver la estructura de nuestro sistema de archivos y elegir el que nos interesa abrir. Esta forma de requerir información del usuario presentando una nueva ventana es muy conveniente, ya que elimina la necesidad de colocar todo en la ventana principal (algo que muchas veces es sencillamente imposible).

Toda aplicación tiene una *ventana principal*; las demás ventanas (opcionales) son llamadas *ventanas secundarias*.

Las ventanas secundarias difieren en algunos aspectos de la principal; por ejemplo, no deben aparecer en la barra de aplicaciones del escritorio. También hay una relación de pertenencia con la principal: cuando ésta se cierra, se cierran automáticamente todas las secundarias que le pertenecen.

Agregar una ficha a la aplicación

Podemos agregar a la aplicación una ficha que ya tenemos grabada, o crear una nueva (posiblemente basándonos en otra). Comencemos por crear una ficha nueva, limpieta de culpa y cargo.

En el menú **File** seleccionamos la opción **New form**. Esto nos crea una nueva ficha con su unit asociada y la muestra en pantalla para poder agregarle controles de la misma manera que hicimos con la principal.

Si no se ha cambiado la especificación por defecto de Delphi¹, la nueva ventana estará completamente vacía y se le asignará el nombre "FormN" donde N es el número de ventana que le corresponde en secuencia: el primero que se crea será "Form1", el segundo "Form2" y así sucesivamente.

Podemos agregar ahora algunos controles a la nueva ventana; no obstante, si corremos el programa notaremos que únicamente aparece la ventana principal (la primera que se crea). ¿Cómo hacemos para ver nuestra nueva ventana?

Seleccionar la ventana principal

Las aplicaciones Delphi pueden contener más de una ventana pero una de ellas debe ser la principal, la que representa a la aplicación, que termina el programa cuando la cerramos. En las aplicaciones Delphi *la ventana que se crea primero es la principal*. Por lo tanto, para cambiar la ventana primaria debemos cambiar el orden de creación de las mismas.

¹ Para cambiar las especificaciones iniciales debemos configurar el "almacén" de objetos: el *repository*, tema que trataremos en breve.

Podemos ver el orden de creación de las ventanas en el código fuente del proyecto (**Project|View source...**), como se muestra en el listado 1 en letra resaltada.

En este listado se ve que la primera ficha que se crea -y por lo tanto se toma como principal- es la llamada **Form1**.

Podríamos cambiar el orden de creación de las fichas sencillamente cambiando de lugar las sentencias de creación de las mismas (las dos líneas resaltadas), pero hay una forma más simple y *visual*: modificar las **Opciones del proyecto**.

Seleccionando la opción de menú **Project|Options**, veremos un cuadro de diálogo como el de la fig. 1. Aquí podemos configurar muchas cosas del proyecto, pero la que nos interesa en este momento es la página “Forms”.

En esta página vemos dos listas de ventanas: las que figuran en la izquierda serán creadas automáticamente por Delphi al iniciar la aplicación, mientras que las de la derecha pertenecen al proyecto (se compilarán junto con el ejecutable) pero la responsabilidad de crearlas corre por nuestra cuenta. Podemos pasar las ventanas de una a otra lista con los botones del medio; por el momento dejaremos las cosas como están.

Arriba de la ventana vemos una lista desplegable con los nombres de los forms de creación automática que componen nuestra aplicación; *la ficha que*

seleccionemos aquí será tomada como la principal de la aplicación. Por ejemplo, podemos seleccionar ahí la segunda ventana que creamos anteriormente (`form2`) y veremos que en la lista de la izquierda se altera el orden colocando el `form2` en primer lugar; esto indica que se creará primero y por lo tanto será considerado el principal. Este cambio se refleja en el archivo fuente del proyecto, que vemos como antes con la opción **Project|View Source** (listado 2).

```

program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm2, Form2);
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.

```

Listado 2: código fuente del proyecto, donde vemos que el `form2` se crea primero (y se toma como principal)

```

program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.

```

Listado 1: código fuente del proyecto indicando la creación automática de las fichas. La ficha `Form1` se toma como principal porque se crea en primer lugar.

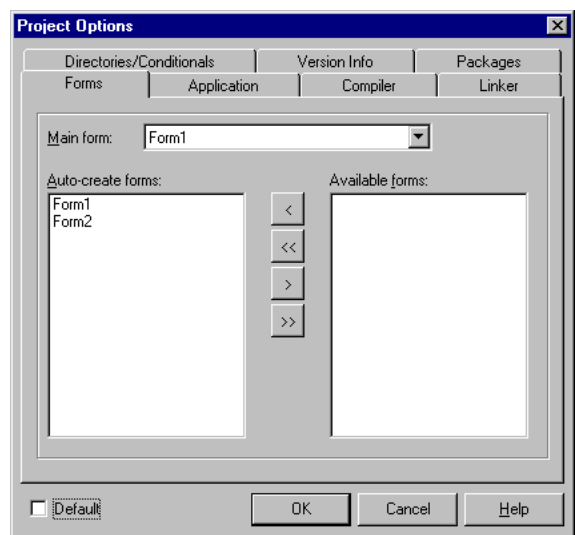


Figura 1: Opciones del Proyecto

Si ahora ejecutamos nuevamente la aplicación, veremos aparecer el `form2` en lugar del `form1`.

Ahora bien, ¿adónde está la ficha que falta? Un poco de paciencia, que ya llegamos.

Por defecto, Delphi crea todos nuestros forms al iniciar el programa (todas las fichas están en la lista de la izquierda del cuadro de diálogo de Opciones), pero sólo muestra en pantalla el principal (aunque los demás tengan la propiedad **Visible en True**).

Con esto se consume más memoria (ya que aunque no las usemos las ventanas estarán creadas en memoria) pero ganamos en otras cosas:

- ? Desde el punto de vista del usuario final, la demora de la creación de las estructuras se dará al principio antes que aparezca nada en la pantalla. Un usuario generalmente preferirá esperar más para *empezar* a trabajar antes que detenerse cada vez que va a realizar un proceso².
- ? Desde el punto de vista del programador, será más fácil trabajar con las ventanas secundarias: ya están creadas, se pueden usar sin hacer nada más.

Una vez creadas, sólo es necesario hacer visibles a las ventanas para que aparezcan en pantalla. Veamos a continuación cómo se logra esto.

Mostrar una ventana secundaria

Las ventanas secundarias que están creadas en memoria no se ven hasta que nosotros indicamos a Windows que las muestre. Hay dos maneras de mostrar una ventana, llamadas **modal** y **no modal**.

Ventanas modales son aquellas que centralizan la atención del usuario; no podemos hacer otra cosa con la aplicación hasta que cerremos la ventana (por supuesto que siempre podemos cambiar a otra aplicación que esté corriendo al mismo tiempo). Se utilizan generalmente para pedir datos necesarios para completar una operación, en forma de *Cuadro de diálogo*. Para mostrarlas se usa el método **ShowModal** de la misma ventana.

Por ejemplo: para mostrar una ventana llamada form3 en forma modal haríamos **Form3.ShowDialog**.

Ventanas no modales son ventanas independientes que se pueden dejar abiertas mientras trabajamos con otra parte de la aplicación. Se usan generalmente para mostrar resultados o ingresar datos al tiempo que hacemos otras operaciones. Para mostrarlas se usa el método **Show** de las ventanas.

Por ejemplo: si queremos mostrar la ventana form3 en forma no modal, hacemos **form3.Show**.

Para comprender el funcionamiento de los distintos tipos de ventanas, crearemos una pequeña aplicación de ejemplo.

Ejemplo

Crear una aplicación que contenga en el form principal tres botones:

² Claro que habrá excepciones: generalmente un buen porcentaje de las ventanas secundarias se utilizan muy poco, de manera que será preferible crearlas “al vuelo” en el momento de usarlas y luego destruirlas para liberar memoria y recursos gráficos.



Además, crear un segundo form (**File|New form**) con tres botones como se ve en la fig. 3.

En el evento **OnClick** del botón “Cerrar” ejecutamos la orden **Close** para cerrar la ventana.

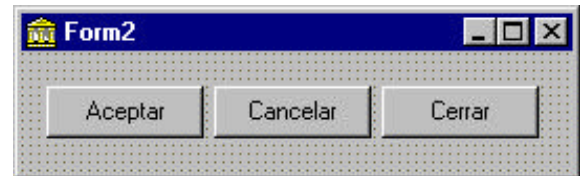


Figura 3: ventana secundaria

Con los botones de la primera ventana vamos a mostrar la segunda en forma modal o no modal, según el botón.

En el botón “Cerrar” de la segunda ventana también debemos ejecutar el método **Close**.

PREGUNTA: ¿qué diferencia hay en el funcionamiento del botón “Cerrar” de la ventana secundaria y el de la ventana principal?

Antes de compilar, hay que tener en cuenta una cosa: desde el form1 estamos llamando a procedimientos del form2. Ésto sólo es posible si incluimos la unit2, donde está definido el form2, en la cláusula **uses** de la unit1. Delphi se da cuenta de esto y nos pregunta si deseamos que realice ese paso automáticamente! Luego de contestar afirmativamente, podemos compilar la aplicación sin problemas (se agrega la nueva **unit** en la cláusula **uses**).

```

procedure TForm1.Button1Click(Sender:
TObject);
begin
    form2.ShowModal;
end;

procedure TForm1.Button2Click(Sender:
TObject);
begin
    form2.Show;
end;

procedure TForm1.Button3Click(Sender:
TObject);
begin
    close;
end;

```

Listado 3: código de los botones de la ventana principal

Prueba el comportamiento de la aplicación. Trata de crear más de una ventana secundaria al mismo tiempo ¿Es posible con las ventanas modales? ¿Y con las no modales? ¿Qué efecto tienen los botones de la segunda ventana? ¿Hay alguna diferencia cuando ésta es modal o no modal?

Como habrán comprobado, al mostrar la ventana secundaria en forma modal tenemos la *obligación* de cerrarla para poder volver al form principal; en cambio, cuando se la muestra no modal podemos simplemente hacerla a un lado y seguir trabajando en la ventana principal -y en cualquier otra, si hubiera más.

NOTA: cuando se cierra la ventana principal se cierran automáticamente todas las secundarias, ya que pertenecen a la aplicación.

Ahora bien, los botones “Aceptar” y “Cancelar” de la ventana secundaria no hacen nada. Están puestos ahí para dar opciones al usuario; deberían cerrar la ventana al tiempo que avisan al programa principal cuál fue el botón presionado. Normalmente se utilizan sólo en las ventanas Modales, en las otras usamos el típico botón “Cerrar”.

Las ventanas modales se cierran al poner en su propiedad **ModalResult** un valor distinto de *mrNone* (0). Entonces, si cada botón pone un valor distinto, podemos leer desde el programa principal el valor de esta propiedad para saber qué botón se presionó.

Esta operación es tan común que ha sido encapsulada en los objetos de la VCL: los botones tienen una propiedad llamada precisamente *ModalResult*, y automáticamente este valor se pone en la propiedad del mismo nombre de la ficha. Si ésta era modal, se cerrará y podremos ver el código del botón causante del hecho.

Pero todavía es más fácil: en realidad el método **ShowModal** es una función, que devuelve un valor... efectivamente, el contenido de la propiedad **ModalResult** de la ficha al cerrarse.

Como ejemplo, modificaremos el código del programa anterior para incluir una comprobación del botón usado para cerrar la ventana modal. Como primera medida debemos poner valores distintos de *mrNone* en la propiedad **ModalResult** de los botones “Aceptar” y “Cancelar” de la ventana secundaria. Cualquier valor distinto de cero está bien, pero hay unos cuantos de uso común ya definidos que se pueden seleccionar de la lista desplegable. Seleccionamos para el botón “Aceptar” el valor *mrOk*, y para el botón “Cancelar” el valor *mrCancel*.

Ahora si, modificamos el código de llamada en la ficha principal:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if form2.ShowModal = mrOk then
    ShowMessage('Se presionó Aceptar');
  if form2.ModalResult = mrCancel then
    ShowMessage('Se presionó Cancelar');
end;
```

Notemos que la función **ShowModal** se llama sólo una vez (sólo la mostramos una vez); la segunda comprobación tenemos que hacerla leyendo directamente la propiedad **ModalResult** de la ficha.

PREGUNTA: ¿Qué hubiera pasado si utilizáramos las dos veces “ShowModal”?

Otro ejemplo: tenemos una aplicación del tipo Editor de texto, y queremos que antes de cerrar verifique si el documento en edición ha sido modificado, y en caso afirmativo presentar un aviso al usuario dándole oportunidad de grabar y salir, no grabar y salir, o cancelar la operación y seguir editando el documento. Esta funcionalidad es muy común en los programas que trabajan con documentos.

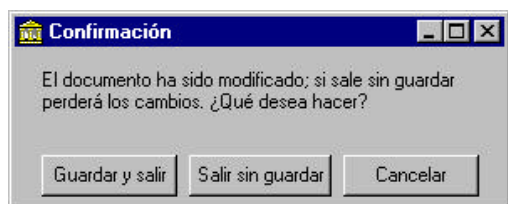


Figura 4: ventana de confirmación (form2)

“Salir sin guardar” tiene el valor **mrNo** y el botón “Cancelar” el valor **mrCancel**.

La comprobación de modificación del documento debe hacerse en algún evento que se produzca al intentar cerrar la aplicación: tenemos disponible el evento **OnCloseQuery**

Suponemos que tenemos una ventana secundaria con el mensaje para el usuario y los botones de selección, como la que aparece en la fig. 4, llamada **form2**. El botón “Guardar y salir” tiene el valor **mrYes** en la propiedad **ModalResult**, el botón

```
procedure TForm1.FormCloseQuery(Sender:
TObject; var CanClose: Boolean);
var
  resp: integer;
begin
  if Modificado then
  begin
    resp:= form2.ShowModal;
    if resp=mrYes then GrabarDoc;
    CanClose:= resp<>mrCancel;
  end
  else
    CanClose:= true;
end;
```

Listado 5: pedir confirmación antes de cerrar

del form principal³. Una codificación posible para resolver el ejemplo se muestra en el listado 5.

Creación de fichas: automática y manual

Hasta ahora no nos hemos tenido que preocupar sobre como crear las ventanas secundarias; Delphi se encarga de eso detrás de bambalinas. No obstante, hay ocasiones en que es conveniente tener el control de cuándo crear las ventanas, y eso es justamente lo que vamos a aprender a continuación.

Recordemos que las ventanas son *clases*, descendientes de la clase base **Tform**; nosotros utilizamos *instancias* de estas nuevas clases. Podemos ver esto claramente en el código asociado con las fichas; por ejemplo, para la primera ventana que se llama por defecto **Form1**, vemos en su unit lo siguiente:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

Vemos aquí la declaración de la clase TForm1 (el valor que ponemos en la propiedad **Name** de la ficha, con una “T” adelante) como descendiente de la clase Tform; a medida que agreguemos controles sobre la ficha veremos aparecer declaraciones de *variables de instancia* (propiedades) para cada control.

En la parte de declaración de variables vemos que Delphi ya ha declarado una variable para almacenar una instancia de la clase: el nombre de esta variable es el que ponemos en la propiedad **Name** de la ficha (en este caso *Form1*).

Ahora podemos entender lo que hace Delphi cuando crea las fichas automáticamente: las asigna a las variables que él mismo declara en las units correspondientes. Si queremos crear nuestras propias instancias y accederlas en forma independiente de las demás, necesitaremos una variable para acceder a cada una.

Necesitaremos entonces como primera medida una variable del tipo correcto, y luego llamar al constructor de la clase -en la VCL se llama **Create**- asignando el resultado de la construcción -es decir, la instancia- a la

³ Podemos también escribir la comprobación en la opción “Cerrar” o “Salir” que seguramente tendrá la aplicación; pero hay otras formas de cerrar una aplicación (por ejemplo, con la combinación Alt+F4) en donde no se comprobarán las modificaciones. En cambio, el evento **OnCloseQuery** se produce *siempre* que se quiera cerrar la ventana, no importa con qué método.

variable.

Este constructor espera un parámetro: el componente *dueño (owner) de la ficha*. Este parámetro indica a Windows que el componente dueño es responsable del nuevo, esto es, que al cerrar o minimizar el componente dueño se hará lo mismo automáticamente con los dependientes. En el caso más común utilizaremos como dueño de todas las fichas al objeto **Application**, que representa la aplicación. Entonces, cuando la aplicación se cierra o minimiza -a través de su ventana principal- todas las fichas dependientes harán lo mismo automáticamente.

Por ejemplo, el siguiente código crearía una instancia nueva de un form de tipo Tform1, referenciándolo con la variable MiVentana:

```
Var
  MiVentana: Tform1;
begin
  MiVentana:= tForm1.Create(Application);
  MiVentana.Caption:= 'Esta es mi ventana';
  MiVentana.Show;
end;
```

Notemos en el código anterior que para trabajar con la ventana usamos la variable MiVentana. La nueva ventana tendrá todas las características que pusimos en el Form1, pero el título será “Esta es mi ventana”.

¡A Probar!

Podemos poner este código como respuesta al evento OnClick de un botón para probarlo. Cada vez que presionemos el botón se creará una instancia de la ventana, cada una idéntica a la anterior pero con contenido independiente (por ejemplo: si la clase tiene un editor, cada instancia tendrá un editor pero éstos podrán contener textos distintos).

Algunas preguntas para pensar: ¿qué pasa con el objeto una vez que lo creamos? La variable MiVentana es local, por lo que al salir del procedimiento se pierde. ¿Qué pasa entonces con la ventana? ¿Se cierra? Si no se cierra y ya no tengo la variable para accederla, ¿cómo le digo que se cierre? Si no se puede cerrar, ¿qué pasa con la memoria y los recursos gráficos que ocupa?

Trata de contestar a estas preguntas antes de seguir; por más que las respuestas sean equivocadas, al corregirlas será más fácil recordarlas.

En efecto, la variable que referencia a la nueva ventana (MiVentana) se pierde al salir del procedimiento. Y la ventana que ya estaba en pantalla queda flotando, perdida. Ya no podemos accederla para cerrarla (no podemos hacer “MiVentana.Close”)⁴, pero todavía podemos cerrarla con el ratón. En cuanto a los recursos gráficos y de memoria, permanecen ocupados claro hasta que se cierra la ventana.

De hecho, estas ventanas no quedan huérfanas del todo; recordemos que al crearlas indicamos que pertenecían a la aplicación. Como dijimos antes, esto significa que la aplicación misma se encargará de destruir las ventanas y devolver los recursos al sistema una vez que se cierre. La casa está en orden...

⁴ En realidad, se guarda una referencia en una lista global de todas las ventanas de la aplicación -la propiedad Screen.Forms.

Ahora bien, hay una cuestión pendiente. Recordemos que podemos acceder a los controles de una ventana aunque no la veamos, siempre que tengamos una variable que la referencie. Esto significa que la ventana *existe*, aunque no la veamos. Incluso después de cerrarla con **Close**, todavía podemos acceder a sus componentes... entonces quiere decir que no se destruye, ¿verdad? Y si no se destruye, no se libera la memoria ni los recursos ni nada. ¡Catástrofe!

En realidad no es para tanto: es cierto, con la instrucción **Close** no se destruye el objeto en memoria -sólo se oculta la ventana. Aún así no es tan grave, porque los recursos gráficos *si* se liberan y lo demás... bueno, es cuestión de balancear la conveniencia de tener “a mano” los controles con algunos recursos ocupados, o perder los controles para recuperar recursos. Generalmente es preferible tener los controles a mano aunque mantengamos ocupados algunos recursos.

Las ventanas en Delphi se *ocultan* en lugar de cerrarse. Esta es la acción por defecto en las ventanas que no tienen el estilo *fsMDIChild* -es decir, que no son ventanas internas de una aplicación MDI. En este último caso, el comportamiento es aún más extraño: las ventanas se *minimizan* al cerrarlas!

Podemos cambiar este comportamiento por defecto de una manera muy sencilla. Todas las ventanas tienen un evento **OnClose**, que se produce cuando se está por cerrar la ventana -una vez que ya dimos el visto bueno en el evento **OnCloseQuery**. En este evento **OnClose** tenemos una oportunidad de decidir qué hacer con el objeto de la ventana: el parámetro **Action**.

La acción que se tome sobre el objeto de la ventana depende del valor que tome esta variable. Los valores posibles y su efecto son los siguientes:

- ? **caHide**: el objeto no se destruye; la ventana se oculta de la vista. Es la acción por defecto en las ventanas no MDI.
- ? **caMinimize**: el objeto no se destruye; la ventana se minimiza dentro de la ventana contenedora. Es la acción por defecto en las fichas MDI.
- ? **caNone**: no se hace nada; ni siquiera se oculta la ventana, en efecto impide que se cierre.
- ? **caFree**: se destruye el objeto automáticamente, por lo que se libera todo y la ventana desaparece de la vista... y de la memoria!

Hay casos en que es conveniente usar la última opción: cuando tenemos una ficha que consume muchos recursos -por ejemplo abre archivos, crea estructuras en memoria, tiene muchos controles que ocupan recursos gráficos- pero usamos muy poco en el trámite normal del programa. Entonces conviene crear esta ventana “al vuelo” y destruirla apenas la dejamos de usar. Lo más conveniente es decirle que se destruya ella misma, poniendo en el evento **OnClose** la siguiente línea

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    Action:= caFree;
end;
```

Creación de una ventana de presentación

Como un ejemplo práctico de los conceptos vertidos anteriormente, crearemos una pantalla de presentación para nuestra aplicación. Esta presentación deberá mostrarse mientras se carga el programa, por lo que debe

ser la primera ventana que se crea, debe mostrarse mientras se crean las demás y debe destruirse antes de empezar a utilizar la aplicación. Todo esto debemos hacerlo “a mano”; Delphi no nos ofrece ninguna ayuda en este caso, así que... a trabajar!

? Paso 1: creación de la ventana de presentación

Lo primero que tenemos que hacer es crear la ventana. Para ello procedemos como siempre: **File|New form** y ponemos los componentes que deseamos. Generalmente se utiliza una imagen como presentación. También es deseable que la ventana de presentación *no tenga barra de título*, por lo que procederemos a quitarla poniendo la propiedad **BorderStyle** en *bsNone*.

Podemos agregar un panel para crear un borde elevado a la imagen; ponemos el panel alineado al cliente y la imagen dentro de él también ocupando todo el espacio. El efecto total es una imagen con un borde “tridimensional” gracias al panel, como en la siguiente imagen:

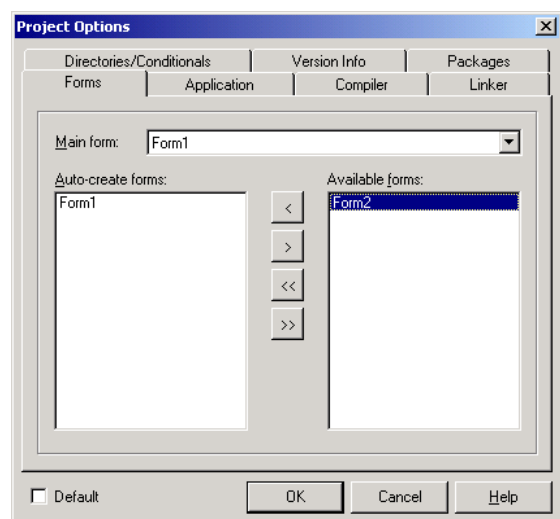


? Paso 2: indicar a Delphi que la ventana no será creada automáticamente

Dado que esta ventana será creada (y destruida) fuera de la secuencia normal de operación de la aplicación, debemos quitarla de la lista de auto creación (donde se colocó automáticamente) en las opciones del proyecto.

Paso 3: modificar el proyecto para crear, mostrar y destruir la ventana de presentación

Debemos modificar el código del proyecto (View|Project source) y agregar las instrucciones necesarias para crear,



Indicamos a Delphi que el Form2 no se debe crear automáticamente

mostrar y destruir la ventana de presentación. Una posibilidad sería la siguiente:

```
program pSplash;

uses
  Forms,
  uSplash1 in 'uSplash1.pas' {Form1},
  uSplash in 'uSplash.pas' {Form2};

{$R *.RES}

begin
  Application.Initialize;
  //Creamos la ventana de presentación
  Form2:= TForm2.Create(nil);
  Form2.Show; //la mostramos
  Form2.Update; //y obligamos a Windows a mostrarla en pantalla ahora
  //Aca viene la creación normal (automática) de ventanas
  Application.CreateForm(TForm1, Form1);
  form2.Free;
  Application.Run;
end.
```

Ahora si, al ejecutar nuestro programa se verá la pantalla de presentación mientras se crean las otras ventanas. En el ejemplo anterior solamente hay una ventana para crear, de manera que casi no se percibe la demora en el arranque y la técnica descrita no es muy visible. Cuando el proyecto tenga muchas ventanas para crear, el tiempo de arranque puede ser considerable -y en ese caso, un indicador como el explicado es muy valioso.

Ejercicio: agregar a la ventana de presentación un “indicador de progreso” que indique aproximadamente cuánto falta para que la aplicación arranque.

Esto está oculto

Distintos tipos de ventanas: normas de diseño

<<<Agregar algo sobre los distintos tipos de ventanas (cuadros de diálogo, ventanas comunes, toolboxes) y como crear cada una; además, en qué momento se deberían usar de acuerdo con las normas de diseño>>>

Aplicaciones MDI

Las aplicaciones MDI son aquellas que pueden abrir varios documentos simultáneamente en diferentes ventanas, que quedan restringidas a la ventana principal (por ejemplo el Word o el mismo Administrador de Archivos en Windows 3.1 o 3.11). Veremos las operaciones básicas para crear una aplicación MDI con Delphi.

Las ventanas de una MDI se denominan ‘ventanas hijas’ (child windows) que comparten un mismo espacio de trabajo (una ventana) que en Delphi siempre será la ventana principal de la aplicación, a la que llamaremos ‘padre’ (parent) o ‘marco’ (frame). Las ventanas hijas pueden ser de distintas clases, pero el frame es uno solo.

Veamos paso a paso cómo crear una aplicación MDI, que usaremos para editar archivos de texto.

Creación del marco

Para crear el marco sólo hay que cambiar una propiedad del form principal, el valor **fsMDIForm** en la propiedad *FormStyle*.

Creación de las ventanas hijas

Haremos un form para cada tipo de ventana hija que vayamos a usar. Luego construiremos instancias de este form para crear las ventanas en run-time.

Para indicar que un form es para ser usado como *child*, ponemos el valor **fsMDIChild** en la propiedad *FormStyle*. Además, debemos eliminar el form de la lista de ventanas de auto creación (en las opciones del proyecto).

El proceso de creación de una nueva ventana es el normal, con el tipo de form que definimos como *Child*.

Ejercicio 2-1

Crear una aplicación MDI. En la ventana marco colocar un menú con un comando que permita crear las ventanas hijas; por lo demás, los forms estarán vacíos. Cada vez que se seleccione la opción se debe poner una nueva ventana hija en el marco.

Vea el comportamiento de las ventanas hijas cuando las queremos cerrar. ¿Se le ocurre alguna manera de hacer que se destruya la ventana en lugar de minimizarse?

?

Acomodar las ventanas

Para acomodar las ventanas en las conocidas formas Cascada y Mosaico o acomodar los iconos en la parte inferior, sólo son necesarias unas pocas líneas de código.

Generalmente tendremos los comandos de ventana en un menú especial, el cual podemos usar también para llevar una lista de las ventanas abiertas para poder acceder rápidamente a ellas.

Ejercicio 2-2

Agregar a la aplicación del ejercicio anterior un menú llamado **Ventana**, con las opciones **Cascada**, **Mosaico** y **Alinear Iconos**.

Crear procedimientos de respuesta a estas opciones con las siguientes líneas (sólo lo que está en negrita es el cuerpo del procedimiento):

Para Cascada: **Cascade**

Para Mosaico: **Tile**

Para Alinear Iconos: **ArrangeIcons**

Ahora en la propiedad *WindowMenu* del form marco seleccione el nombre del menu *ventana*. Correr la aplicación y comprobar los cambios.

?

Esto está oculto

<<<Esto no estoy seguro que quede bien aca>>>

Drag&Drop

La operación de Drag&Drop (arrastrar y soltar) es cada vez más común en los programas de Windows. La implementación de la operación básica en Delphi es muy simple, sólo hay que seguir los siguientes pasos:

- ? Poner la propiedad **DragMode** del componente fuente en *dmAutomatic* para que Delphi maneje los eventos del mouse y cambie el cursor automáticamente.
- ? Crear un procedimiento de respuesta al evento **OnDragOver** del componente destino, donde debemos decir a Delphi si este componente acepta la operación o no. El parámetro **var Accept** debe ponerse en **True** para indicar que aceptamos la operación. Generalmente esto se hace comprobando que el componente fuente de la operación (parámetro **source** en el evento) es un determinado componente o por lo menos de un tipo específico.
- ? Si el parámetro **Accept** del evento **OnDragOver** es **True**, al soltar el botón del mouse con algo que arrastráramos sobre el destino se provoca el evento **OnDragDrop**. En este evento debemos ejecutar la acción que se pretendía con la operación de arrastre, para lo cual podemos acceder al componente donde se inició el arrastre con el parámetro **source**.

Ejercicio 2-3

Crear una aplicación con un form que contenga dos listas. Se deben poder pasar cadenas de una lista a otra arrastrando y soltando.

?

Cuadros de diálogo estándar

<<<Repasar el texto, los gráficos ya están actualizados>>>

Windows tiene definidas algunas cajas de diálogo de uso común, buscando una uniformidad en la presentación de las aplicaciones que redunde en un aprendizaje acelerado por parte de los usuarios: si todas las aplicaciones hacen las mismas tareas con la misma interface hacia el usuario, si manejamos una podemos manejar todas.

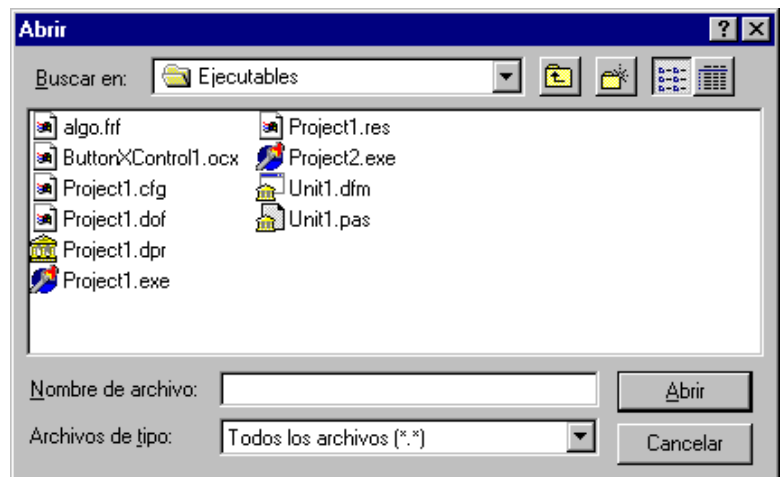
Con esta idea en mente, Windows incluye las interfaces para Abrir Archivos, Grabar Como, Elegir una Fuente, Elegir un Color, Impresión, Configuración de Impresoras, Buscar y Reemplazar. Como de costumbre, Delphi incluye componentes que encapsulan las propiedades y acciones de estos diálogos, en la página *Dialogs* de la paleta de componentes.

Generalidades

Los componentes de diálogo estándar ya están diseñados; sólo es necesario crear una instancia, ejecutarlos llamando al método **Execute** y listo! Las características que se pueden cambiar están en el Inspector de Objetos como propiedades. Veamos cada uno por separado:



Abrir archivo:



cuadro de diálogo estándar para abrir archivo

Para usarlo, llamamos al método **Execute** y después leemos en la propiedad **Filename** el nombre del archivo seleccionado.

Las propiedades más útiles son:

DefaultExt: es la extensión que se adicionará al nombre si el usuario no especifica una. Se compone de tres letras. No incluir el punto.

FileEditStyle: Indica si el editor de nombre es un editor propiamente dicho o un combo box con una lista de los archivos abiertos anteriormente. OJO: si elegimos esta última opción, el mantenimiento de la lista histórica de archivos es nuestra responsabilidad. Vea el Help para mayores detalles.

Filename: Es el nombre de archivo que aparecerá al principio en el editor de nombre

Filter: contiene una lista de filtros posibles, que aparecerán en el combo box de *Mostrar Archivos de Tipo*. El filtro que aparece inicialmente es el especificado en la propiedad **FilterIndex** (empieza en 1).

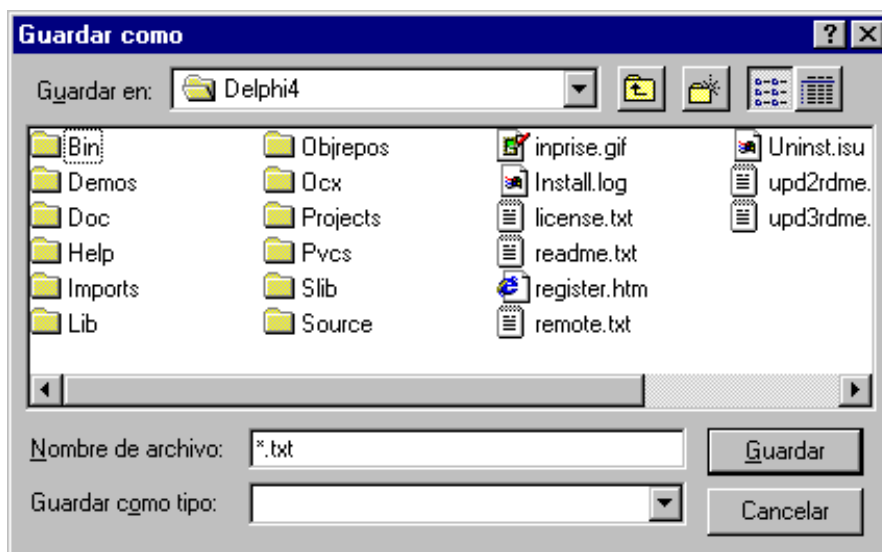
InitialDir: es el directorio inicial.

Options: varias opciones para configurar la caja de diálogo. Ver en la ayuda los detalles de cada una.

Title: El título (caption) de la caja de diálogo.



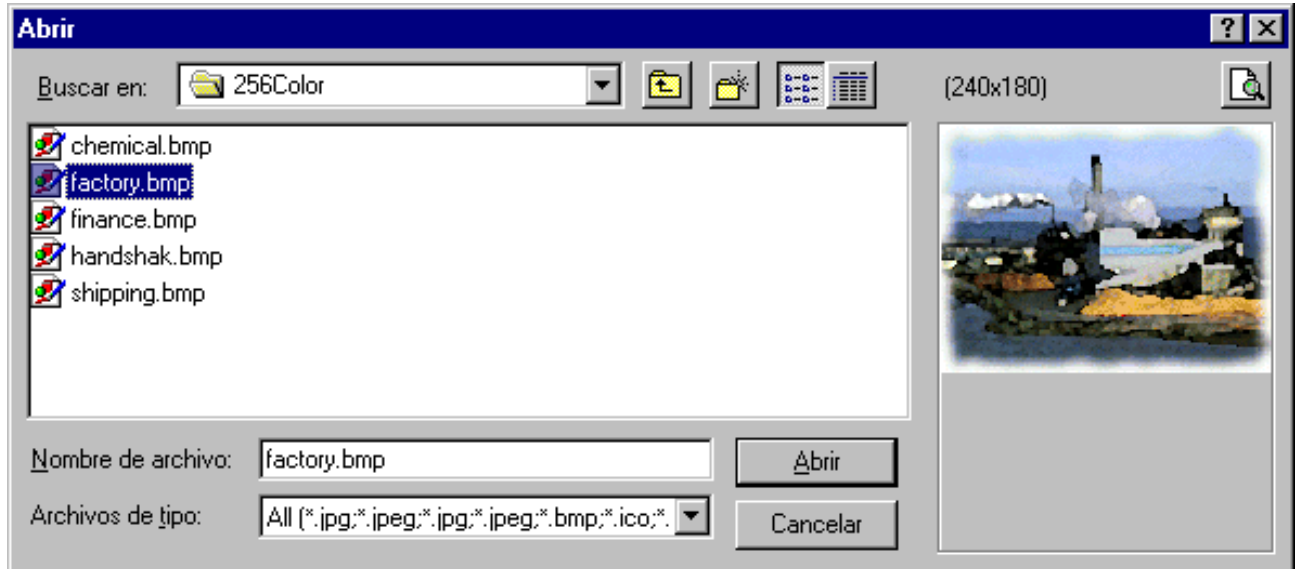
Guardar como



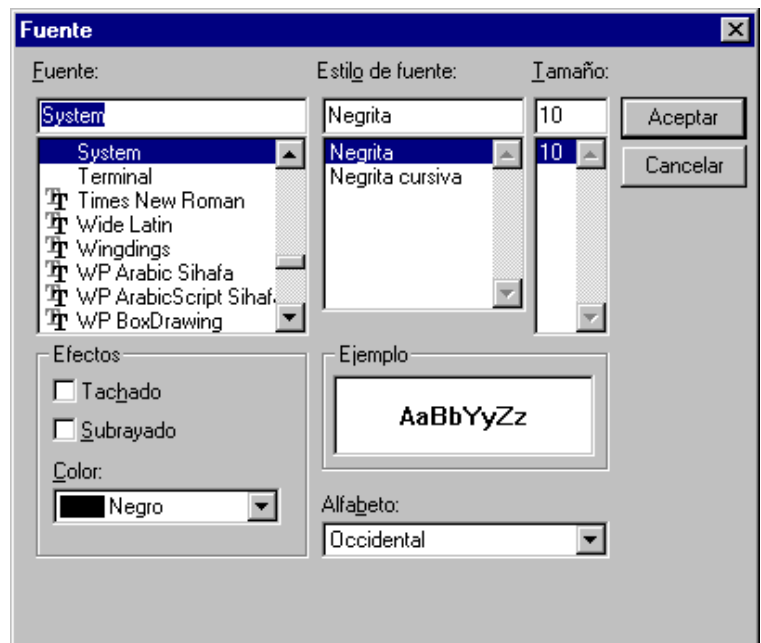
Se utiliza igual que el de Abrir; las propiedades que vimos para aquel componente valen también para éste.

En Delphi 4 se agregan dos cuadros de diálogo especiales, para abrir y guardar imágenes. Son iguales a los cuadros de diálogo vistos anteriormente, pero con un cuadro de vista previa a la derecha. De este modo, con sólo seleccionar el archivo de gráfico deseado (.BMP, .WMF, .ICO) lo veremos sin necesidad de abrirlo.

Vemos en la siguiente figura el cuadro de Abrir Imagen (OpenPictureDialog, <<<Agrega icono>>>); el de Guardar Imagen (SavePictureDialog, <<<Agrega Icono>>>) es exactamente igual, sólo cambia el título.



Tipo de letra

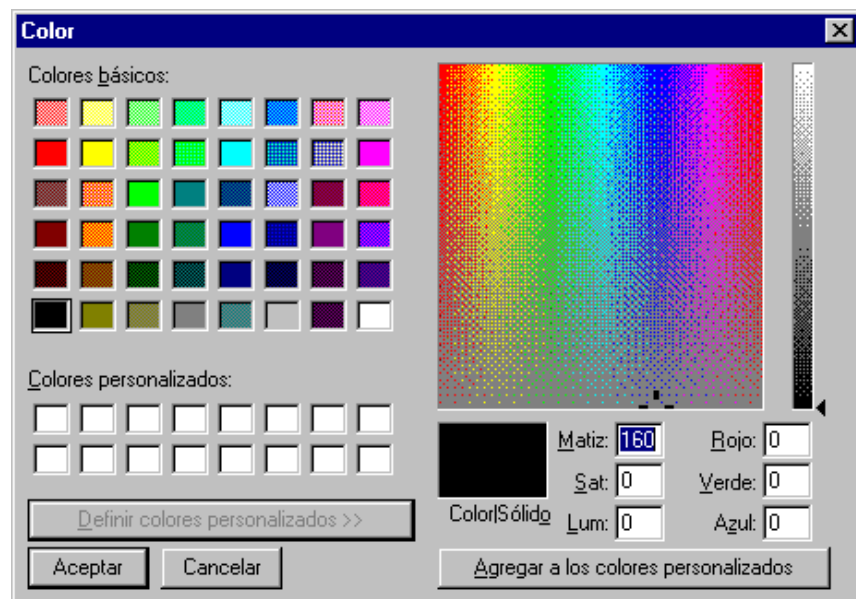


Después de ejecutarlo, la fuente seleccionada estará disponible en la propiedad **Font**. La propiedad más útil es la **Options**, donde se especifican varias opciones como por ejemplo si se muestran sólo las fuentes TrueType o todas. Una vez más, los significados correctos están listados en la ayuda.

Hay un evento especial en este cuadro de diálogo, llamado **OnApply**. Si asignamos un procedimiento de respuesta a este evento, aparecerá un nuevo botón etiquetado **Aplicar**, que disparará el evento. Típicamente, en este procedimiento aplicaremos los cambios pedidos pero en forma temporal, *sin cerrar el cuadro de diálogo y con la posibilidad de cancelar los cambios*.



Colores

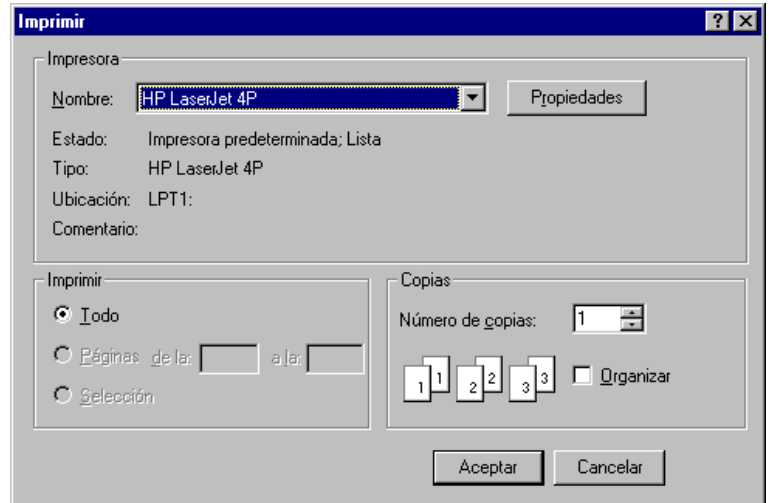


Permite seleccionar un color de la paleta de Windows o crear uno nuevo. Después de ejecutarlo, si el usuario presiona el botón OK se guarda el color elegido en la propiedad **Color**.

La propiedad **CustomColors** es una lista de strings que mantiene un formato igual al utilizado por Windows en el CONTROL.INI. Se puede utilizar esta propiedad para guardar y recuperar una lista de hasta 16 colores personalizados, para tenerlos disponibles en la caja de diálogo.



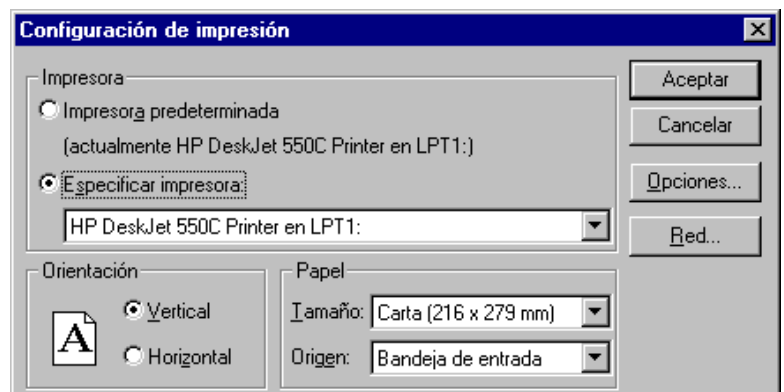
Imprimir



Este componente nos permite especificar un rango de impresión, a qué impresora va dirigido, e inclusive permite la configuración de la impresora. Al presionar OK, se pueden consultar los valores seleccionados por el usuario en las propiedades correspondientes, por ejemplo ver el rango de páginas en **From Page** y **ToPage**. *No se efectúa la impresión.*



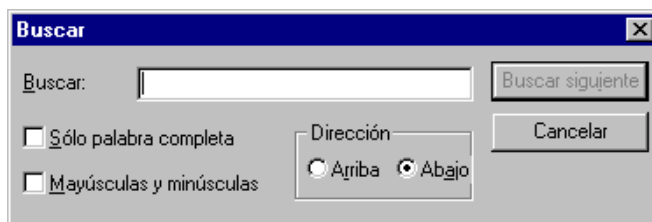
Config. de impresoras



Este diálogo no devuelve nada, simplemente nos permite configurar la impresora. Los datos que aparecen en el diálogo dependen de la impresora, pero normalmente tendremos disponible a primera vista el tamaño de papel y el origen del mismo, así como su orientación (que son los ajustes más comunes). Para una configuración más detallada, presionar el botón **Opciones**.



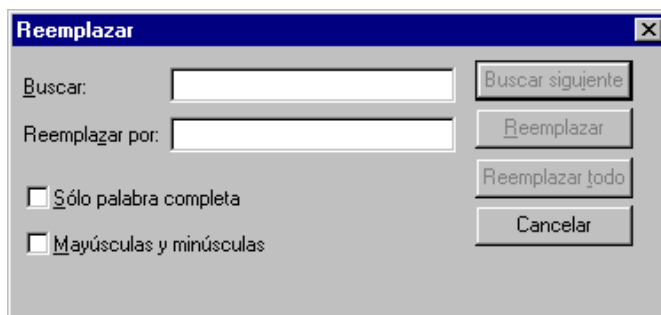
Buscar



Permite la búsqueda de palabras en un texto. Cuando el usuario presiona el botón de buscar, se produce un evento **OnFind**. Aquí es donde nosotros debemos poner el código de búsqueda propiamente dicho. Los valores de las opciones seleccionadas se pueden consultar en la propiedad **Options**, y el texto ingresado en el edit estará en la propiedad **FindText**.



Reemplazar



Es igual al diálogo de encontrar texto, pero agrega la posibilidad de especificar un string de reemplazo. Ahora tenemos dos eventos, **OnFind** y **OnReplace**. El segundo se produce cuando seleccionamos Replace o ReplaceAll, por lo cual debemos chequear en el evento cuál de los dos ha sido elegido.

Tratamiento de errores.

Excepciones

En todos los programas alguna vez se producirá un error. Esta afirmación, digna de Murphy, expresa algo que los programadores conocen bien: no hay que preguntarse si un sistema va a fallar, sino *cuando* va a fallar.

Hemos terminado el mejor sistema de nuestra carrera. Todo anda bien y rápido en nuestro equipo y en otros con diversas configuraciones de memoria y procesador. Hemos tenido en cuenta todo lo imaginable; incluso lo ha manejado la abuela del vecino -la máquina más moderna que ha manejado en su vida es una escoba- y pudo hacerlo funcionar. No puede estar más listo para su entrega, pensamos (el programa, no la abuela) y llamamos al cliente que nos cita para mañana a la mañana. Cuando llegamos allá, instalamos todo... perfecto. Lo probamos haciendo algunas transacciones simples... perfecto. Una vez que tenemos el OK del jefe, presionamos el botón que vacía todas las tablas e inicializa el sistema para comenzar a utilizarlo. Y nos vamos. El jefe se sienta a su máquina luciendo su mejor sonrisa de juguete nuevo... y antes de llegar a la puerta nos vemos detenidos por una orden tajante de que no nos dejen abandonar el lugar. El sistema no funciona. ¿Cómo es posible?

Después de días y noches de buscar por las miles de líneas de código el error y nuestro prestigio, encontramos la causa: el sistema no estaba preparado para trabajar con tablas vacías, una condición extraña que no había sido tenido en cuenta... hasta que fue demasiado tarde.

Este escenario no es tan ficticio o extraño, como sabe cualquiera que haya programado un tiempo. Por lo tanto, es indispensable contar con algún método para protegerse de los errores.

Delphi proporciona el suyo mediante las **excepciones**.

Cuando ocurre un error en tiempo de ejecución Delphi genera una *excepción*; crea una instancia de una clase especial denominada **Exception** o algún descendiente, el flujo del programa se altera y únicamente se ejecuta el *código de respuesta a excepciones*. Este código puede tratar el error (procesarlo y responder si corresponde) o no, en cuyo caso se genera un evento **OnException** en la aplicación. Haremos aquí un estudio de las excepciones, cómo protegernos de ellas e incluso utilizarlas para nuestro provecho.

Cómo se detectan y procesan

Una excepción se mantiene hasta que se atiende o termina el programa. Normalmente procesaremos las excepciones que se produzcan en nuestro código para recuperarnos sin problemas de los errores; veremos además que Delphi nos da una vía para escapar airosos de las situaciones no previstas, cuando se producen errores inesperados que harían caer nuestro sistema sin remedio, tal vez dejando archivos abiertos y con pérdida de datos.

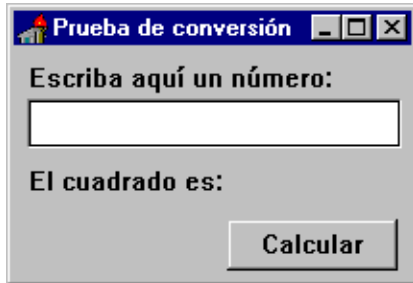


Figura 1: Programa para probar el comportamiento de Delphi ante un error

Notemos primero el tratamiento por defecto de los errores en Delphi. Utilizaremos un programa muy simple que calcula el cuadrado de un número entero. A continuación vemos el form principal (fig. 1) y el único método del programa, asignado al evento OnClick del botón (listado 1):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i:integer;
begin
  i:= StrToInt(edit1.text);
  label1.caption:= format('El cuadrado es: %d',[i*i]);
end;
```

Listado 1: Código de conversión sin protección ante errores

Cuando presionamos el botón, se convierte el texto tipeado en el editor en un número entero, se lo eleva al cuadrado y se muestra el resultado en el label. No obstante, si escribimos algo que no sea un número entero -incluso si dejamos vacío el editor- se producirá un error de conversión. Delphi procesa este error mostrando un pequeño mensaje indicativo (Fig. 2).

(Entre comillas figura el texto escrito en el editor, que provocó el error; el título del mensaje es el nombre de la aplicación).

Como vemos, no pasa nada grave. La respuesta por defecto de Delphi a las excepciones es sólo mostrar el mensaje correspondiente en una caja de diálogo como la anterior. En una aplicación simple no es necesario mucho más, pero sería agradable para el usuario que se le explique un poco mejor qué ha sucedido. Además, no hay nada más frustrante que un mensaje donde se nos dice “Se ha producido un error”. La única opción que tiene es presionar el único botón que puede ver, para hacer desaparecer este mensaje. Y los cambios que había realizado en los datos. En la sección siguiente cambiaremos el comportamiento por defecto de Delphi mostrando un mensaje adecuado al tipo de error.



Figura 2: Mensaje de error por defecto

Cambiar la respuesta por defecto a las excepciones

Para modificar la respuesta estándar de Delphi ante un error debemos escribir un procedimiento de respuesta al evento **OnException**, que se produce en el objeto **Application**.

El objeto **Application** no es un componente visible; se crea automáticamente cada vez que se ejecuta una aplicación y se referencia con la variable global *Application*. Para indicar que queremos que se ejecute un método al producirse el evento **OnException**, debemos utilizar el hecho que *los eventos son propiedades*. Como tales, se les puede asignar un valor (en el caso de los eventos, un procedimiento que se ejecutará al producirse el evento). La definición de la propiedad **OnException** nos indica qué tipo de método podemos asignarle. En el caso de **OnException**:

```
property OnException: TExceptionEvent;
```

El tipo **TExceptionEvent** es un tipo procedural:

```
TExceptionEvent = procedure (Sender: TObject; E: Exception) of object;
```

Por lo tanto, el procedimiento debe esperar dos parámetros, uno de tipo **TObject** y otro de tipo **Exception**, y debe ser un método de un objeto¹. Escribamos entonces un método nuevo al form principal (listado 2):

```
procedure TForm1.TratarExcepciones(sender:TObject; e:Exception);
begin
  MessageDlg('Se ha producido un error. Por favor intente de nuevo',
    mtError,[mbOk],0);
end;
```

Listado 2: procedimiento de respuesta a una excepción

La declaración de este método debe ser escrita en la definición del tipo **Tform1** (cuando asignamos un método a un evento a través del Inspector de Objetos Delphi escribe esta declaración por nosotros). En el listado 3 se ve la definición de la clase entera

¹ El método puede pertenecer a cualquiera de los forms que componen el proyecto.

```

TForm1 = class(TForm)
  Button1: TButton;
  Edit1: TEdit;
  Label1: TLabel;
  Label2: TLabel;
  procedure Button1Click(Sender: TObject);
private
  procedure TratarExcepciones(sender:tObject; e:Exception);
public
  { Public declarations }
end;

```

Listado 3: Declaración del nuevo método

Haremos la asignación al crear el form principal, en el evento **OnCreate**:

Y bien, ahora el programa presenta un mensaje más amable cuando se produce un error *de cualquier tipo*. El paso siguiente es "filtrar" los errores, respondiendo de manera diferente a cada uno de ellos.

```

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnException:= TratarExcepciones;
end;

```

Listado 4: Asignación del nuevo método al evento

Podemos ver de qué tipo es una excepción mirando la clase a que pertenece el parámetro **E**. Debemos primero repasar la jerarquía de clases de excepciones definida en Delphi.

Categorías y clases de excepciones

Las excepciones en Delphi se pueden dividir en las siguientes categorías:

1. Conversión de tipo

Se producen cuando tratamos de convertir un tipo de dato en otro, por ejemplo utilizando las funciones `IntToStr`, `StrToInt`, `StrToFloat`. Delphi dispara una excepción **EConvertError**.

2. Tipo forzado (typecast)

Se producen cuando tratamos de forzar el reconocimiento de una expresión de un tipo como si fuera de otro usando el operador **as**. Si no son compatibles, se dispara la excepción **EInvalidCast**.

3. Aritmética de punto flotante

Se producen al hacer operaciones con expresiones de tipo real. Existe una clase general para este tipo de excepciones -**EmathError**- pero Delphi utiliza sólo los descendientes de ésta:

- **EInvalidOp**: el procesador encontró una instrucción inválida.
- **EzeroDivide**: división por cero.
- **Eoverflow**: se excede en más la capacidad aritmética (números demasiado grandes).
- **Eunderflow**: se excede en menos la capacidad aritmética (números demasiado pequeños).

4. Aritmética entera

Se producen al hacer operaciones con expresiones de tipo entero. Existe una clase general definida para este tipo de excepciones llamada **EIntError**, pero Delphi sólo utiliza los descendientes:

- **EDivByZero**: división por cero.
- **ERangeError**: número fuera del rango disponible según el tipo de dato. La comprobación de rango debe estar activada (indicador **\$R**).
- **EIntOverflow**: se excede en más la capacidad aritmética (números demasiado grandes). La comprobación de sobrepasamiento debe estar activada (indicador **\$O**).

5. Falta de memoria

Se producen cuando hay un problema al acceder o reservar memoria. Se definen dos clases:

- **EOutOfMemory**: no hay suficiente memoria disponible para completar la operación.
- **EInvalidPointer**: la aplicación trata de disponer de la memoria referenciada por un puntero que indica una dirección inválida (fuera del rango de direcciones permitido a la aplicación). Generalmente significa que la memoria ya ha sido liberada.

6. Entrada/Salida

Se producen cuando hay un error al acceder a dispositivos de entrada/salida o archivos. Se define una clase genérica **EInOutError** con una propiedad que contiene el código de error **ErrorCode**.

7. Hardware

Se producen cuando el procesador detecta un error que no puede manejar o cuando la aplicación genera intencionalmente una interrupción para detener la ejecución. *El código para manejar estas excepciones no se incluye en las DLL compiladas, sólo en las aplicaciones.* Se define una clase base que no es directamente utilizada **EProcessorException**. Las clases útiles son los descendientes:

- **EFault**: es una clase base para todas las excepciones de faltas del procesador.
- **EGPFFault**: error de Protección General, cuando un puntero trata de acceder posiciones de memoria protegidas.
- **EStackFault**: acceso ilegal al segmento de pila del procesador.
- **EPageFault**: el manejador de memoria de Windows tuvo problemas al utilizar el archivo de intercambio.
- **EInvalidOpcode**: el procesador trata de ejecutar una instrucción inválida.
- **EBreakpoint**: la aplicación ha generado una interrupción de la ejecución (punto de ruptura, utilizado por el *debugger* de Delphi para inspeccionar las variables en un punto).
- **ESingleStep**: la aplicación ha generado una interrupción de ejecución paso a paso. Luego de cada paso de programa se produce la interrupción. Es utilizada también por el *debugger* de Delphi.

8. Excepciones silenciosas

Se disparan intencionalmente por la aplicación para interrumpir el flujo de ejecución; no generan mensajes de error. Se define una clase: **EAbort**. Esta excepción es automáticamente generada cuando invocamos el procedimiento global **Abort**.

NOTA: La excepción silenciosa provocada por el procedimiento **Abort** puede utilizarse para *interrumpir* la ejecución del programa en cualquier punto.

Como podemos ver en el *Browser* (menú **View**), la jerarquía de clases descendientes de **Exception** es bastante amplia; no obstante, muchos de los descendientes son de uso interno de los componentes y no los experimentaremos directamente. La jerarquía completa de excepciones se ve en la fig. 3:

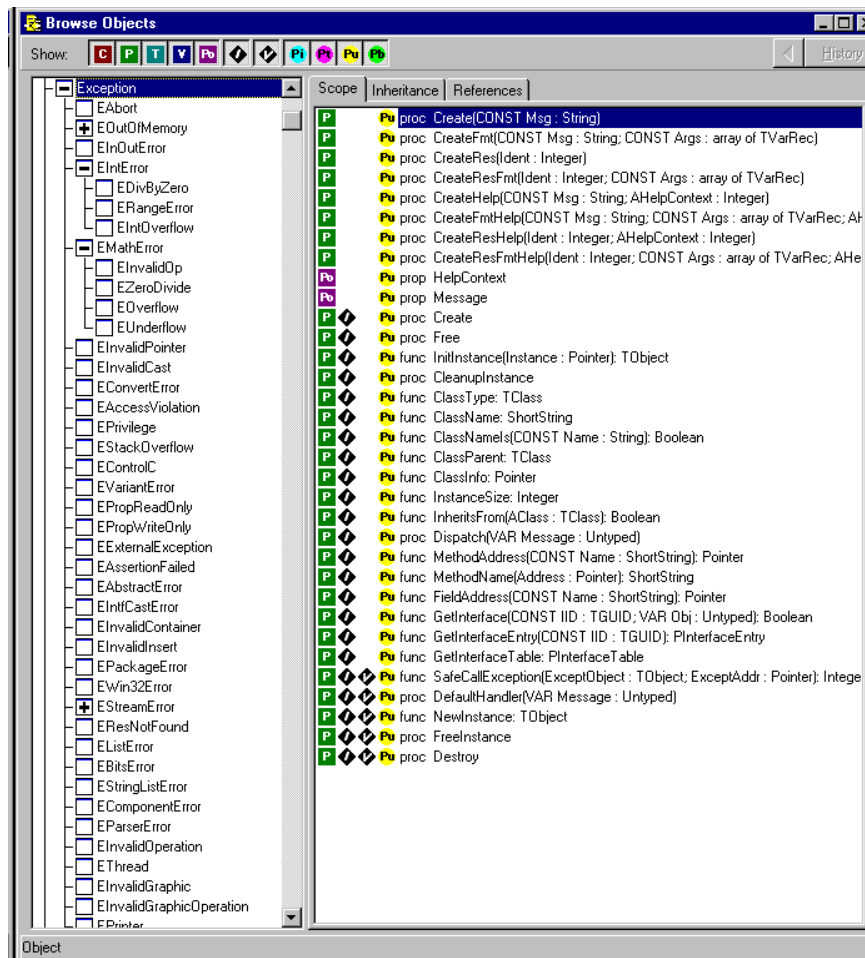


Figura 3: Parte de la jerarquía de excepciones como se ve en el Browser de Delphi 3

Con esta información podemos "filtrar" las excepciones que se produzcan en nuestros programas y responder a cada clase de manera diferente si fuera necesario. Este tratamiento específico se utiliza por lo general para responder a distintas excepciones en el lugar donde se producen, de manera tal de recuperarnos del error y poder proseguir el código. El ejemplo más común es el error de lectura de un archivo: normalmente permitiremos al usuario reintentar la operación además de cancelarla. Pero recordemos que el evento **OnException** se produce en el objeto **Application**, después de lo cual la ejecución queda a la espera de nuevos eventos. Debemos encontrar una forma de detectar y corregir el error sin abandonar el procedimiento en curso. Esto se logra mediante la *protección de bloques de código*.

Protección de bloques de código. *Try..Except*

Para proteger una porción de código debemos encerrarla en un bloque **try...except**. Entre estas dos palabras reservadas se ubica el código que está expuesto a errores; después de **except** se procesan estos últimos, cerrando todo el bloque con **end**. La sintaxis es

```
try
:
  {Bloque de código propenso a errores}
:
except
  on <clase de excepción> do
  :
    {una sola instrucción o un bloque begin..end}
  on <otra excepción diferente> do
  :
end;
```

Listado 6: Sintaxis del bloque try..except

Veamos un ejemplo:

En la aplicación que creamos al principio, cuando se producía un error la ejecución saltaba al procedimiento *TratarExcepciones*, y el texto del label 1 no se actualizaba. Una versión más elaborada podría cambiar el mensaje del label cuando se produce la excepción; podemos hacerlo fácilmente cambiando el código de respuesta al botón "Convertir" por el del listado 7:

```

procedure TForm1.Button1Click(Sender: TObject);
var
  i:integer;
begin
  try
    try
      i:= StrToInt(edit1.text);
      labell.caption:= format('El cuadrado es: %d',[i*i]);
    except
      on eConvertError do
        Labell.Caption:= 'Número erróneo';
    end;
  end;
end;

```

Listado 7: Tratamiento de una excepción en el lugar donde se produce

Los bloques protegidos forman como *capas* al resto del programa; cuando se produce una excepción en un lugar del código la ejecución salta directamente a la capa de protección donde se está ejecutando. Si el error no se procesa en esta capa se pasa a la siguiente, y así sucesivamente. Las aplicaciones de Delphi corren dentro de un bloque protegido, por lo que todas las excepciones no tratadas especialmente se procesan en el objeto Application.

En la Fig. 4 vemos un esquema de una aplicación con varios bloques protegidos y cómo se redirige la ejecución cuando se produce un error.

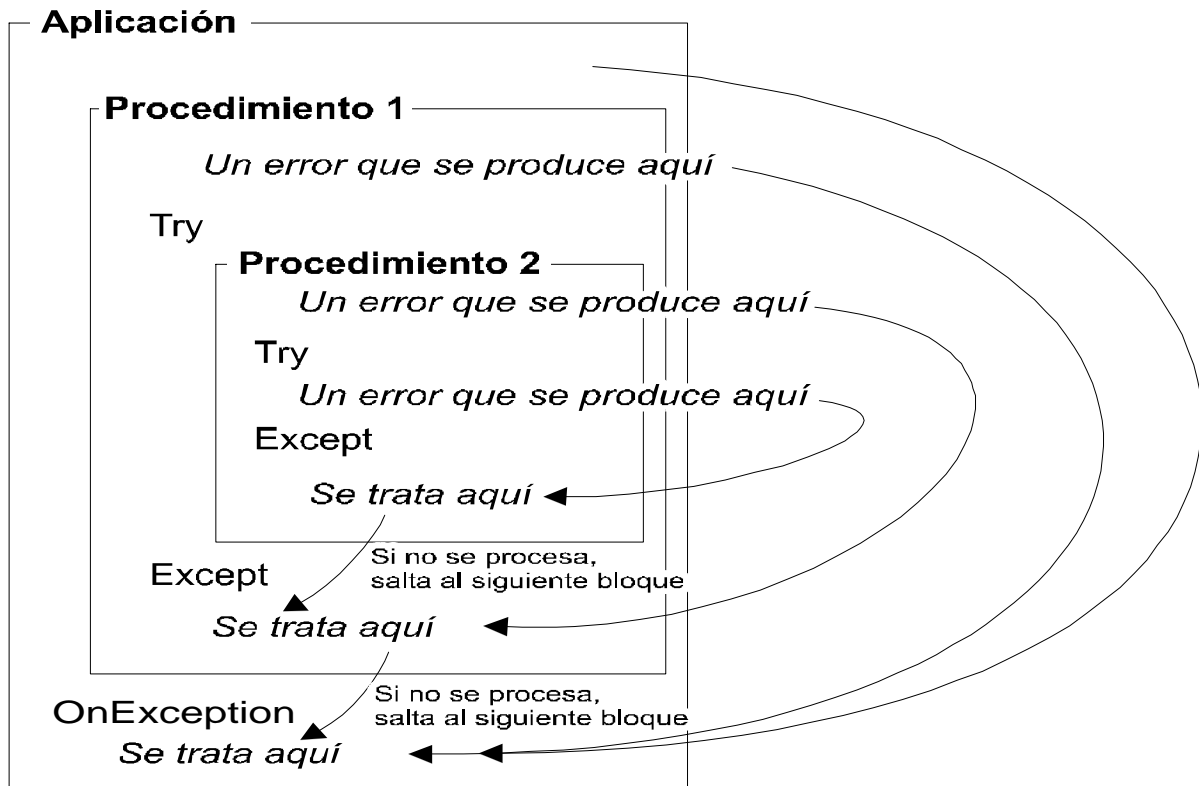


Figura 4: Varios bloques de protección anidados

En el gráfico vemos que los errores que se producen *fuera* de las estructuras **try..except** son tratadas en la parte correspondiente al bloque anterior; en última instancia se procesan en el evento OnException de la aplicación. En efecto, es como si toda la aplicación estuviera encerrada en un gran bloque *try..except*.

La técnica de declarar como clases a las excepciones permite el *tratamiento jerárquico* de las mismas. Esto quiere decir que un manejador de una clase de excepciones trata también las clases descendientes, con un código como el del listado 8.

```
Try
  {código}
except
  on <clase de la excepción> do
    {manejar la excepción o sus descendientes}
  on <otra clase> do
    :
end;
```

Listado 8:
tratamiento de una
clase de errores

por ejemplo si queremos atrapar cualquier error aritmético que se pueda producir en una operación, podemos escribir:

```
try
  {operación}
except
  on eMathError do
    ShowMessage('Error de punto flotante');
  on eIntError do
    ShowMessage('Error aritmético entero');
end;
```

Listado 9:
ejemplo de
tratamiento de
clases de errores

Podemos hacer algo más que mostrar un mensaje, por ejemplo asignar al resultado de la operación un valor por defecto.

Veremos a continuación una serie de situaciones comunes donde es necesario un control de errores y cómo se corrigen los mismos.

División por cero

Cada vez que hacemos una división estamos propensos al error que se produce si el divisor es cero. Según se trate de una división entera (**div**) o de una de punto flotante, se dispara una excepción **eDivByZero** o **eZeroDivide**, respectivamente.

Las siguientes funciones (listado 10) permiten probar un método de recuperación del error de

división por cero. En este caso no se presenta ningún mensaje al usuario, sólo se devuelve un valor por defecto.

```
Function DivisionEntera (a,b: integer): integer;
begin
  try
    Result:= a div b;
  except
    on eDivByZero do
      Result:= 0;
    end;
  end;

function DivisionReal (a,b:real): real;
begin
  try
    Result:= a/b;
  except
    on eZeroDivide do
      Result:= 0.0;
    end;
  end;
end;
```

Listado 10: ejemplo de tratamiento de excepciones de división por cero, tanto en el caso de nros enteros como punto flotante

Acceso a un archivo

1) Abrir un archivo y leer datos.

El siguiente procedimiento (listado 11) abre un archivo y lee 50 bytes en un array, mostrando un mensaje si se produce cualquier error en el proceso.

```
var
  f: file;
  b: array[0..49 of byte];
begin
  try
    AssignFile(f,'nombre');
    ResetFile(f,1);
    BlockRead(f,b,50);
    CloseFile(f);
  except
    on eInOutError do
      ShowMessage('Error al trabajar con el archivo');
    end;
  end;
end;
```

Listado 11: trabajo con archivos en un bloque protegido

NOTA: vea la sección sobre *finally* antes de ejecutar el código anterior

2) Abrir un archivo. Bloques anidados

El procedimiento siguiente (listado 12) se puede usar cuando necesitamos utilizar un archivo; si existe, se abre y si no existe se crea.

Como vemos en el listado 12, tenemos dos bloques protegidos anidados, uno dentro del otro. El interior captura un error al abrir (por ejemplo, cuando no existe el archivo); mientras que el exterior intercepta un posible error de creación, presentando un mensaje al usuario y abandonando inmediatamente el procedimiento. Si todo va bien, podemos utilizar el archivo abierto (con posible comprobación de errores al leer o escribir) y finalmente debemos cerrar el archivo.

```
Var
  f: file;
begin
  try
    try
      AssignFile(f, 'ARCH.TXT');
      Reset(f, 1);
    except
      On eInOutError do
        Rewrite(f, 1);
      end;
    except
      on eInOutError do begin
        ShowMessage('No se puede crear el archivo');
        Exit;
      end;
    end;
    {utilizar el archivo}
    CloseFile(f);
  end;
```

Listado 12: creación de un archivo si no se puede abrir

MUY IMPORTANTE

Notemos que de producirse un error al abrir o leer el archivo, la instrucción de cierre no se ejecutará. En el caso de una lectura no hay mayores problemas, pero cuando modificamos el contenido del archivo es *indispensable* cerrarlo correctamente. Veremos más sobre esto al tratar el bloque de protección **try..finally**.

La estructura *try..finally*

Como vimos en el ejemplo anterior de acceso a archivos, hay veces que es necesario ejecutar una porción de código *suceda un error o no*. Para esto, existe en Delphi la estructura **try..finally**. Cuando se produce un error dentro de este bloque, se suspende el tratamiento de la excepción para ejecutar el código que sigue a **finally**. Luego de terminado, se sigue con el proceso normal de proceso del error.

La estructura es una variación de los bloques de protección que vimos antes, donde en lugar de

except utilizamos la palabra reservada **finally** (listado 14).

```
try
  {código expuesto a errores}
finally
  {código de ejecución obligatoria}
end;
```

Listado 14: el bloque *try..finally*

Un ejemplo simple: cuando realizamos una tarea que puede extenderse en el tiempo, es bueno indicarlo al usuario mediante el cursor de espera (el reloj de arena, en Delphi *crHourglass*); al terminar la operación debemos restituir el cursor anterior. Ahora bien, si ocurre un error durante el proceso, la instrucción de restitución del cursor no se ejecutará nunca... y el usuario puede quedarse horas esperando que el cursor le indique que puede seguir trabajando. Podemos evitar esta enojosa situación (que sin embargo es muy común) con el código del listado 15:

```
try
  screen.cursor:= crHourglass; {ponemos el cursor de espera}
  {aquí se hace el proceso}
finally
  screen.cursor:= crDefault; {restituimos el cursor por defecto}
end;
```

Listado 15:
cambio seguro
del cursor

Advertencia: si en la sección **finally** se produce un error, la ejecución saltará inmediatamente a la siguiente capa de protección (sección **except** correspondiente o al manejador por defecto de la aplicación). Por lo tanto, debemos evitar en esta sección utilizar código propenso a errores, o desactivar la detección de los mismos.

Los bloques **try..except** y **try..finally** pueden anidarse. Podemos ver un ejemplo de esta técnica en el programa *Visor Hexadecimal*.

El programa Visor Hexadecimal nos permite ver en una ventana el texto de un archivo junto a los códigos hexadecimales que corresponden a cada caracter. El form principal se ve en la fig. 5; el programa completo está en el listado 17.

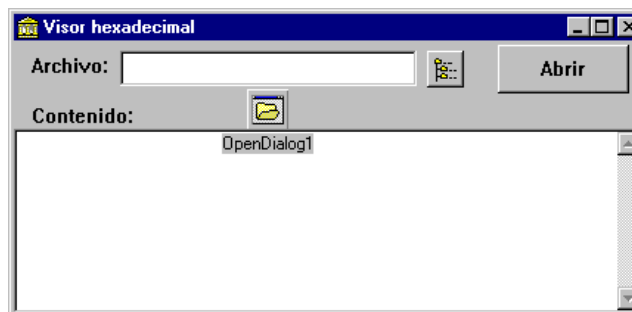


Figura 5: form principal del programa *visor hexadecimal*

Listado 17: programa *visor hexadecimal*

```
unit Excep_1;

interface

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

const
  Linea = 10;
type
  tArra = array[0..linea*10-1] of byte;

  TForm1 = class(TForm)
    Edit1: TEdit;
    Label1: TLabel;
    Mem1: TMemo;
    Button1: TButton;
    SpeedButton1: TSpeedButton;
    OpenDialog1: TOpenDialog;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure SpeedButton1Click(Sender: TObject);
    procedure FormResize(Sender: TObject);
  private
    procedure MuestraHexa(b:tArra; Cant:word);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.MuestraHexa(b:tArra; Cant:word);
var
  i, j: word;
  s, s2: string;
  bt: byte;
begin
  i:= 0;
  while i*linea<Cant do begin
    s:= '';
    s2:= '';
    for j:= 0 to linea-1 do begin
      if i*linea+j>cant then break;
      bt:= b[i*linea+j];
      s:= s+IntToHex(bt,2)+' ';
      if bt>31 then
        s2:= s2+Chr(bt)
      else
        s2:= s2+' ';
    end;
    try
      Mem1.Lines.add(s+s2);
    except
      on Exception do begin
        ShowMessage('No se pueden ingresar más líneas');
        abort;
      end;
    end;
  end;
end;
```

```

        end;
        inc(i);
    end;
end;

procedure TForm1.Button1Click(Sender: TObject);
Var
    f: file;
    buf: tArra;
    leidos:integer;
begin
    try
        AssignFile(f,Edit1.text);
        Reset(f,1);
    except
        On eInOutError do begin
            ShowMessage('No se puede abrir el archivo');
            exit;
        end;
    end;
    try
        try
            while not eof(f) do begin
                BlockRead(f,buf,SizeOf(buf),Leidos);
                MuestraHexa(buf,leidos);
            end;
        except
            on eInOutError do
                ShowMessage('Error al leer el archivo');
            end;
        finally
            CloseFile(f);
        end;
    end;
end;

procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
    if OpenFileDialog1.Execute then
        edit1.text:= OpenFileDialog1.FileName;
end;

procedure TForm1.FormResize(Sender: TObject);
begin
    Mem1.height:= ClientHeight-64;
end;

end.

```

Uso de *finally*: ejemplos comunes

Repasaremos a continuación los casos más comunes que necesitan código de ejecución obligatoria.

Cerrar un archivo

Retomemos el ejemplo de lectura de archivos que vimos antes. Como mencionamos allí, si se produce un error mientras accedemos al archivo o procesamos sus datos, no se ejecutará la orden **CloseFile** y el archivo permanecerá abierto. Esto puede ocasionar pérdida de datos ya que el sistema de directorios se vuelve inestable. Agreguemos el código para cerrar el archivo aunque se produzca un error:

```
var
  f: file;
  b: array[0..49 of byte];
begin
  try
    try
      AssignFile(f,'nombre');
      ResetFile(f,1);
      BlockRead(f,b,50);
    except
      on eInOutError do
        ShowMessage('Error al trabajar con el archivo');
    end;
  finally
    CloseFile(f);
  end;
end;
```

Listado 17: el bloque try..finally asegura que el archivo se cierre correctamente aunque se produzca un error

Liberar memoria

Si creamos una estructura dinámica, es nuestra responsabilidad devolver la memoria utilizada al sistema. Veamos un ejemplo utilizando **GetMem** y **FreeMem**:

```
type
  aInt= array[0..0] of integer;
  paInt = ^aInt;
var
  a: paInt;
begin
  GetMem(a,SizeOf(integer)*10000); {solicitamos memoria para 10000 enteros}
  try
    {utilización de la memoria}
  finally
    if assigned(a) then FreeMem(a,SizeOf(integer)*10000);
  end;
end;
```

Listado 18: liberar memoria

Devolver recursos gráficos

En Windows, los recursos deben ser compartidos entre todas las aplicaciones que corren en un determinado momento. Esto es especialmente crítico para los recursos gráficos, ya que el sistema posee pocos y siempre son solicitados. Por lo tanto, es indispensable devolverlos después de

usarlos.

Veremos un ejemplo que utiliza un Device Context para acceder a la pantalla:

```
var
  MiDC: hDC;
begin
  try
    GetDC(0); {pide un manejador para la pantalla entera}
    {código que grafica en la pantalla usando la API}
  finally
    ReleaseDC(0); {libera el DC al sistema}
  end;
end;
```

Listado 19: liberar recursos gráficos

Utilizar el objeto de la Excepción

Cuando se dispara una excepción se crea una instancia de la clase correspondiente. Por lo tanto, podemos en principio tener acceso a las propiedades y métodos de la misma.

La definición de la clase Exception es la siguiente (SYSUTILS.PAS):

```
Exception = class(TObject)
private
  FMessage: PString;
  FHelpContext: Longint;
  function GetMessage: string;
  procedure SetMessage(const Value: string);
public
  constructor Create(const Msg: string);
  constructor CreateFmt(const Msg: string; const Args: array of const);
  constructor CreateRes(Ident: Word);
  constructor CreateResFmt(Ident: Word; const Args: array of const);
  constructor CreateHelp(const Msg: string; AHelpContext: Longint);
  constructor CreateFmtHelp(const Msg: string; const Args: array of const;
    AHelpContext: Longint);
  constructor CreateResHelp(Ident: Word; AHelpContext: Longint);
  constructor CreateResFmtHelp(Ident: Word; const Args: array of const;
    AHelpContext: Longint);
  destructor Destroy; override;
  property HelpContext: Longint read FHelpContext write FHelpContext;
  property Message: string read GetMessage write SetMessage;
  property MessagePtr: PString read FMessage;
end;
```

Listado 20: La definición de la clase Exception

Como vemos en el listado anterior, tenemos disponibles tres propiedades: una indica el contexto de ayuda, y las otras dos dan acceso al mensaje de error en los dos formatos utilizados en Windows. Veamos cómo utilizar estas propiedades para mostrar un mensaje personalizado.

Para acceder al objeto de la excepción debemos utilizar una variable temporal. La forma de hacerlo es la siguiente:

```
try
  {Algún proceso}
except
  on e: eDivByZero do
    ShowMessage('Error!'#13+e.Message);
end;
```

Listado 21: acceder al objeto de la excepción

Con este código no estamos *creando* una nueva instancia de la excepción; únicamente definimos una variable que apunta al objeto creado por Delphi. La clase Exception (y la mayoría de los descendientes directos) tienen pocas propiedades de utilidad, pero nada nos impide en los descendientes creados por nosotros agregar otras nuevas. Esto es precisamente lo que hace Delphi con la clase `eInOutError`, definiendo una propiedad que almacena el código de error devuelto por el sistema:

```
EInOutError = class(Exception)
public
  ErrorCode: Integer;
end;
```

Por lo tanto, podemos saber cual fue el error que provocó la excepción. El siguiente fragmento indica cómo mostrar este código al usuario, a la vez que muestra una forma de asegurar el cierre del archivo:

```
var
  f:file;
  st: array[0..50] of char;
begin
  try
    try
      AssignFile(f,'PRUEBA.TXT');
      Reset(f,1);
      BlockRead(f,st,SizeOF(st));
    except
      on e: eInOutError do
        ShowMessage(format('Error %d al acceder al archivo PRUEBA.TXT',
          [e.ErrorCode]));
      end;
    finally
      {$I-} {desactiva detección de errores}
      CloseFile(f); {puede dar error si f no está abierto}
      {$I+} {activa la detección de errores de nuevo}
    end;
  end;
end;
```

Listado 22: utilización de una propiedad de la excepción

Note la utilización de `{$I-}` y `{$I+}` en la parte Finally; si el archivo no se pudo abrir se produce

el error, se muestra el mensaje... y se ejecuta CloseFile sobre un archivo que no está abierto, lo que genera otro error. Para evitar que la ejecución salte al bloque protegido superior con este último error, desactivamos la detección de los mismos cuando tratamos de cerrar el archivo. De esta manera si el archivo está abierto se cierra normalmente, y si no está abierto no pasa nada.

Una aplicación más elaborada podría mantener una lista con los mensajes de error correspondientes y utilizar el código de error como índice.

Provocar una excepción

Hay ocasiones en que es conveniente llamar a una capa de tratamiento de errores superior; por ejemplo, podríamos mostrar un mensaje en un bloque local y después dejar que la capa superior libere recursos. Pero una vez que hemos tratado una excepción, ésta se da por terminada y el código de los bloques protegidos superiores no se ejecuta. Debemos indicar a Delphi que deseamos mantener la excepción para que pueda tratarla el bloque superior: algo así como llamar a un método heredado desde una clase descendiente.

Para lograrlo, el lenguaje contempla la opción de *re-lanzar* la excepción, como si se volviera a producir. Sólo debemos insertar en el código la palabra reservada **raise**:

```
try
  StrToInt('a45');
except
  on exception do
  begin
    ShowMessage('Se ha producido un error');
    raise;
  end;
end;
```

Listado 23:
Utilización de **raise**

En el ejemplo anterior utilizamos **raise** para disparar la misma excepción que estábamos procesando. La ejecución salta inmediatamente al bloque protegido anterior.

También utilizamos **raise** para disparar nuestras propias excepciones, como veremos a continuación.

Definir nuevas excepciones

Cuando se produce un error en un procedimiento nuestro (ya sea de un objeto o no), podemos tratar de recuperarnos en el lugar o bien lanzar una excepción indicando que algo anduvo mal.

Esta segunda alternativa es la preferida, porque le permite al programador tratar todos los errores del mismo modo.

Para definir una nueva excepción, debemos crear un descendiente de **Exception** o alguna clase más especializada. En este descendiente podemos definir nuevas propiedades o métodos de la manera usual, lo cual nos permitirá un mejor tratamiento del error.

Vayamos a la práctica. Supongamos que implementamos una función que transforma un número entero en una cadena de unos y ceros que conforman su representación binaria, pero no queremos tratar los números negativos. Podemos entonces definir una excepción nueva y dispararla cuando detectamos que el número es incorrecto. No necesitamos ninguna propiedad ni método nuevo, por lo que la excepción puede ser un descendiente directo de **Exception**. No obstante, como se trata en realidad de un error durante una conversión, haremos que descienda de **eConvertError**:

```
type
  eSoloPosit = class (eConvertError)
  end;
```

y ahora escribimos el procedimiento de conversión:

```
function DecToBin(v:integer): string;
var
  temps: string;
begin
  if v<0 then
    raise eSoloPosit.Create('No se permiten números negativos');
  if v>0 then temps:= ''
    else temps:= '0';
  while v>0 do begin
    temps:= Chr(v Mod 2+48)+temps;
    v:= v div 2;
  end;
  result:= temps;
end;
```

El proyecto BINARIO.DPR muestra la llamada de esta función desde un bloque protegido, en una aplicación simple de conversión:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  try
    Label3.caption:= DecToBin(StrToInt(edit1.text));
  except
    on e: eConvertError do
      ShowMessage('Error! '+e.Message);
  end;
end;
```

Note que el tratamiento jerárquico nos permite procesar también los errores de la conversión a entero con el mismo bloque.

Sumario

Hemos visto los conceptos claves del tratamiento de errores en Delphi. La técnica de las excepciones permite un proceso claro, limpio y seguro de los errores que se producen en las aplicaciones o el sistema. A lo largo del texto hemos discutido procedimientos comunes para proteger nuestros programas de los errores más comunes.

Un aspecto importante de esta implementación es la definición de **clases** de excepciones. Esto nos habilita para tratar en forma jerárquica los errores e incluso definir nuestras propias clases especializadas como descendientes de las existentes.

En definitiva, una técnica poderosa para tratar un aspecto muy importante de la programación de aplicaciones.

Dibujar en Windows

Windows es un entorno gráfico. Esta afirmación, que no debería sorprender a nadie, tiene algunas implicaciones en el tema que nos ocupa: por su propia naturaleza, es de esperar que el Sistema Operativo nos brinde facilidades a la hora de hacer gráficos. Por suerte, esta vez nuestras esperanzas se confirman. La graficación en Windows se basa en una interface llamada GDI (Graphics Device Interface), que se presenta al programador en forma de estructuras de datos, tipos y funciones; lo que llamamos API (Application Programming Interface). La función de esta API es actuar como una capa intermedia entre los dispositivos físicos (pantalla, placas de video, etc.) y nuestros programas, lo que significa que no tendremos que crear *drivers* y otras cosas raras para permitir a nuestros programas trabajar con tal o cual placa de video; si el equipo está ejecutando Windows, entonces nuestro programa puede correr en ese equipo¹.

Esta API nos simplifica un montón las cosas... aunque todavía pueden ser más simples, como lo demuestra Delphi al encapsular las estructuras y funciones en objetos. Nos dedicaremos aquí a mostrar la forma en que se trabaja con gráficos en Delphi, incluyendo cuando sea necesario alguna interacción con la API.

Hay dos maneras de poner gráficos en las aplicaciones de Delphi:

- ? Usar componentes gráficos en tiempo de diseño (como los componentes `TImage` o los botones de clase `TBitBtn`)
- ? Dibujarlos en tiempo de ejecución

Hemos visto ya el uso de los componentes `TImage` y los botones `TBitBtn`. Ahora nos centraremos en la creación de gráficos en tiempo de ejecución.

Para graficar en Delphi trabajamos con un objeto llamado *Canvas* (se podría traducir como “Lienzo”). Representa la superficie de graficación, con propiedades tales como la pluma o el pincel que vamos a usar. Además, tiene métodos que nos permiten dibujar primitivas -rectángulos, círculos, elipses, texto, etc- directamente sobre su superficie.

Coordenadas

Es importante entender cómo se toma el sistema de ejes coordenados en la pantalla. Para cualquier componente, la coordenada X corre desde el borde izquierdo hacia el derecho, igual que en matemática; en cambio, la coordenada Y se toma desde arriba hacia abajo, al contrario de la norma seguida usualmente por los matemáticos. Esto significa que el centro de coordenadas está en la esquina superior izquierda del componente (Fig. ???).

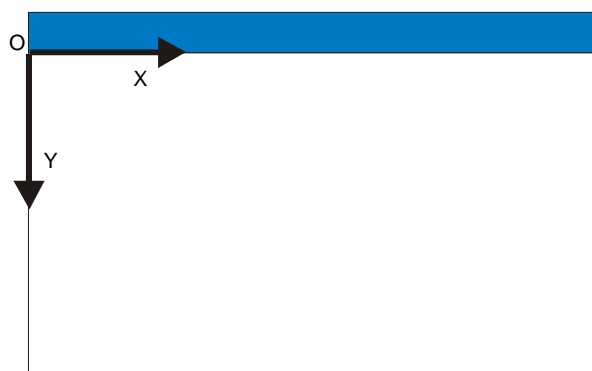


Figura 1: sistema de coordenadas gráfico

Para definir un punto se necesitan dos coordenadas, x e y . Se pueden pedir las dos coordenadas por separado, como dos parámetros de tipo entero, o bien utilizar una estructura creada al efecto: el registro `Tpoint`. Está

¹ Teóricamente. Nunca falta alguna desviación de los estándares por parte de los fabricantes, por lo que puede suceder que hagan falta algunos retoques en los drivers para llevar todo a buen puerto. Normalmente, con un mensaje a los fabricantes basta para tener en pocos días un controlador actualizado disponible en Internet.

definido en la unit *Windows*, de la siguiente manera:

```
type TPoint = record
  X: Longint;
  Y: Longint;
end;
```

Como vemos, nada más que una forma cómoda de tener las dos coordenadas “en un solo paquete”. Existe incluso una función que toma dos números y los devuelve encapsulados en un Tpoint:

```
function Point(AX, AY: Integer): TPoint;
```

Lo mismo sucede con los rectángulos: podemos definirlos con cuatro coordenadas, las de la esquina superior izquierda y las de la esquina inferior derecha. Existe también una estructura, llamada en este caso Trect.

```
TRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;
```

Se trata de un registro variante. Según como se lo pidamos, nos entregará su contenido en cuatro números enteros o en dos registros Tpoint. Algo así como la dualidad onda-partícula de la mecánica cuántica, o los discursos de los políticos.

Supongamos que tenemos las coordenadas de un rectángulo en una estructura como la anterior, llamada **R**; en tal caso, las siguientes sentencias son equivalentes:

```
Canvas.Rectangle(R.Left, R.Top, R.Right, R.Bottom);
```

```
Canvas.Rectangle(R.TopLeft.X, R.TopLeft.Y, R.BottomRight.X, R.BottomRight.Y);
```

También en este caso existe una función que toma cuatro números y los devuelve empaquetados en un registro Trect:

```
function Rect(ALeft, ATop, ARight, ABottom: Integer): TRect;
```

Ahora sí, vamos a dibujar!

En dónde dibujamos

Todavía nos queda algo por aclarar: en dónde ponemos el código. Y la respuesta más correcta sería: “depende de en dónde queremos graficar”. Como dijimos anteriormente, cada componente que muestra una cara en la pantalla tiene su propio **Canvas**, que tendremos que utilizar para dibujar en ese componente. ¿Cuándo? Cuando lo deseemos, puede ser al presionar un botón o al mover el ratón o en cualquier otro momento.

Hay sin embargo un evento especial para dibujar la ventana; este evento se produce cada vez que Windows determina que hay que actualizar el contenido de la ventana, por ejemplo si se mueve otra ventana que tapaba una parte de la nuestra. Este evento existe solamente en la clase TForm, y se denomina **OnPaint**. Todos los ejemplos en que no se indique lo contrario se deberían escribir en este evento.

Recalquemos un punto: *la ficha es responsable por mantener su aspecto*. La responsabilidad de Windows llega hasta avisar a la ventana que se tiene que redibujar, y punto. Pilatos, un poroto en comparación. Esto significa que la ventana tiene que poder redibujar su frente en cualquier momento. Normalmente esto requiere que almacenemos en algún lugar los datos necesarios. Por ejemplo: si tenemos un programa que dibuje las coordenadas del puntero cada vez que se presiona un botón en el ratón, debemos almacenar las coordenadas de todos los puntos donde pinchamos y graficar todas juntas en el evento **OnPaint**. Caso contrario, si tapamos a nuestra ventana llena de coordenadas con otra, y después la destapamos... veremos como se esfumaron los textos y aparece la superficie limpiita.

Primitivas

Se llama **Primitivas** a las formas básicas de todo dibujo: rectángulos, círculos, líneas, puntos. Dependiendo del sistema de graficación tendremos más o menos primitivas -por ejemplo, bajo un entorno de graficación 3D podríamos tener una primitiva **cubo**. Windows nos da acceso a sus primitivas a través de funciones de la API. No obstante, Delphi encapsula estas funciones dentro de la clase TCanvas, y en especial las funciones de la API son métodos de la clase TCanvas. Todos los componentes que permiten que les dibujemos encima tienen una instancia de esta clase, como una propiedad llamada *Canvas* (después dicen que los programadores no tenemos imaginación para elegir nombres).

En muchas de las funciones para dibujar primitivas gráficas necesitaremos expresar un rectángulo; normalmente lo haremos con las coordenadas de los puntos superior izquierdo e inferior derecho. Empezamos por las primitivas más sencillas: por ejemplo, un punto.

No hay una primitiva que grafique un punto.

¿¿Cómo?? El gráfico más simple posible, y no hay forma de hacerlo? Claro que hay formas de dibujar un punto, sólo que no hay un procedimiento exclusivo para eso.

Podemos dibujar un punto de por lo menos dos maneras diferentes:

- ? accediendo al pixel individual para cambiarle el color directamente (propiedad *Pixels*)
- ? haciendo una *recta* de un punto de largo.

Lo que sí nos permite la API de Windows es el mover la pluma *sin graficar*, algo así como decirle a la máquina “apoya el lápiz en las coordenadas (x,y) para empezar a dibujar”. *No se grafica nada con esta orden*, solamente se posiciona la punta del lápiz.

La declaración de este método es por supuesto muy simple:

```
? procedure TCanvas.MoveTo(x,y: integer);
```

¿Y cómo sabemos si la pluma se movió realmente? Pues resulta que en cualquier momento podemos saber en qué lugar está la “punta del lápiz” leyendo la propiedad **PenPos**, que es de tipo **Tpoint**.

Como un ejemplo, cambiaremos la pluma a la posición (100,45) y comprobaremos inmediatamente que se realizó el movimiento:

```
Canvas.MoveTo(100,45);
if (Canvas.PenPos.x=100) and (Canvas.PenPos.y=45) then
    ShowMessage('Todo OK, colega!')
else
    ShowMessage('La pluma no se encuentra en la posición indicada');
```

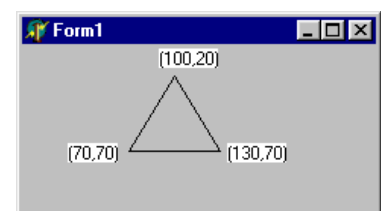
La capacidad de mover la pluma es importante porque no tenemos disponible una función para dibujar una línea entre dos puntos; en su lugar, existe la función **LineTo** que toma las coordenadas del punto *final* de la línea... tomando como inicial el punto donde esté la pluma en el momento de dibujar.

```
? procedure LineTo(x,y: integer);
```

Esto nos permite hacer una línea continua partiendo de un punto, simplemente llamando a **LineTo** por cada tramo.

Ejemplo 1: dibujar un triángulo como el mostrado en la figura:

```
Canvas.MoveTo(100,10);
Canvas.LineTo(130,70);
Canvas.LineTo(70,70);
Canvas.LineTo(100,10);
```



Ejemplo 2: crear una función que permita dibujar una línea especificando los dos puntos extremos. Podemos definir la función **Linea** como sigue:

```
function Linea(ElCanvas: Tcanvas; P0,P1: Tpoint);
begin
    ElCanvas.MoveTo(P0.x,P0.y);
    ElCanvas.LineTo(P1.x,P1.y);
end;
```

Notemos la necesidad de incluir el canvas sobre el cual se desea dibujar como un parámetro; si esta función estuviera definida en la clase Tcanvas esto no sería necesario. Les propongo como un ejercicio interesante la creación de una clase descendiente de Tcanvas que implemente la función anterior.

Ejemplo 3: dibujar el mismo triángulo anterior, ahora utilizando la nueva función *Linea*.

```
Linea(Canvas,Point(100,20),Point(130,70));
Linea(Canvas,Point(130,70),Point(70,70));
Linea(Canvas,Point(70,70),Point(100,20));
```

De la misma manera que en el ejemplo, podemos dibujar un rectángulo o en general un polígono de N lados. Pero hay formas más fáciles de hacerlo. Tenemos la función **Rectangle**, que dibuja un rectángulo, y la función **Polygon** que dibuja un polígono.

? **procedure Rectangle(x1,y1,x2,y2: integer);**

Dibuja un rectángulo tomando (x1,y1) como la esquina superior izquierda y (x2,y2) como la esquina inferior derecha.

? **procedure Polygon(points: array of tPoint);**

Dibuja una serie de líneas conectando los puntos del array, *uniendo el último punto con el primero*. Para indicar los puntos en la línea de comandos es muy útil la función **Point** de Delphi.

Ejemplo 4:

Para dibujar un cuadrado con vértices en los puntos (10,10); (100,10); (100,100); (10,100) podemos hacer lo siguiente:

```
Canvas.Polygon([Point(10,10),point(100,10),point(100,100),point(10,100)]);
```

o bien, utilizando Rectangle:

```
Canvas.Rectangle(10,10,100,100);
```

o bien, usando líneas:

```
with Canvas do
begin
    MoveTo(10,10);
    LineTo(100,10);
    LineTo(100,100);
    LineTo(10,100);
    LineTo(10,10);
end;
```

Pero **¡CUIDADO!** Hay una diferencia fundamental entre hacer una figura utilizando las funciones **Rectangle** o **Polygon** y hacerlas simplemente dibujando líneas: las funciones mencionadas generan figuras *cerradas*, por lo que automáticamente se pintan con el pincel activo (por defecto, color blanco sólido).

Usando líneas, por más que éstas se intersecten y formen una figura cerrada, no se pintará esa superficie porque en lo que a Windows concierne se trata de una serie de líneas y nada más.

En lo que sigue indicaremos entonces para cada método de **Canvas** si se genera una figura abierta o cerrada. Recordemos que en el último caso se pinta automáticamente la superficie interior, por lo que si no tomamos recaudos se tamará lo que había debajo.

Hay funciones que producen variantes sobre las anteriores:

? **procedure FrameRect(const rect:tRect);**

Dibuja un rectángulo sin relleno. Note el uso de una estructura Trect, a diferencia de **Rectangle**. Genera una figura cerrada pero con “relleno transparente” (a todos los usos prácticos, *sin relleno*). *Se utiliza el color de pincel (Brush) como color del cuadro.*

? **procedure RoundRect(x1,y1,x2,y2,ancho,alto: integer);**

Dibuja un rectángulo entre los puntos $(x1,y1)$ y $(x2,y2)$, redondeando las puntas con cuartos de elipse de anchura *Ancho* y altura *Alto*. Genera una figura cerrada.

? **procedure Polyline(points: array of tPoint);**

Dibuja una serie de líneas conectando los puntos del array. Genera una figura abierta.

Notemos que no hay una función especial para graficar un cuadrado, ya que éste es un caso especial de rectángulo.

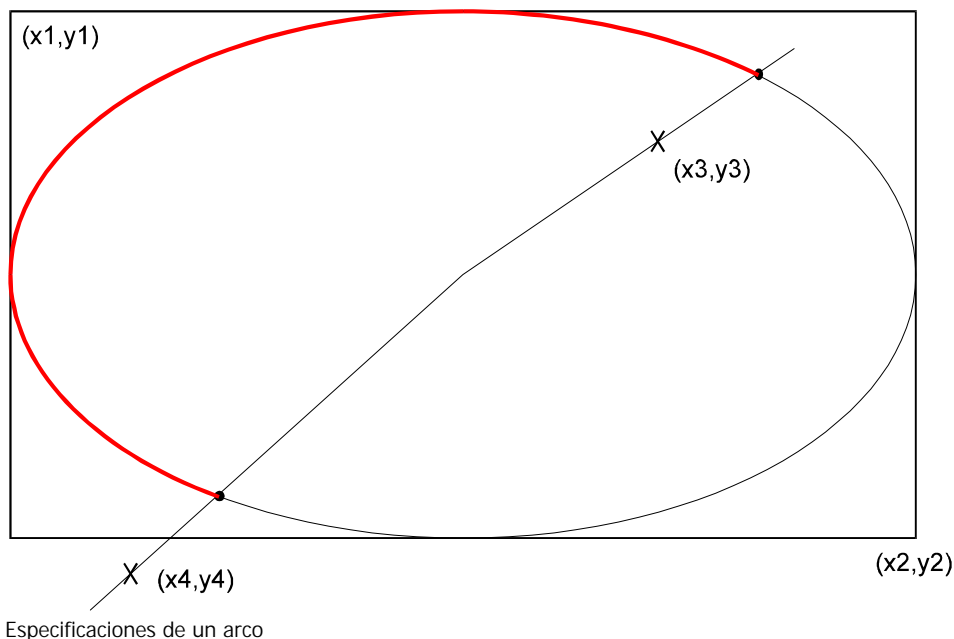
Otra serie de funciones se ocupan de las figuras *curvas*: elipses, arcos, etc.

? **Procedure ellipse(x1,y1,x2,y2: integer);**

Dibuja una elipse inscrita en el rectángulo definido por los dos puntos (figura cerrada). También se utiliza para dibujar círculos.

? **procedure arc(x1,y1,x2,y2,x3,y3,x4,y4: integer);**

Dibuja un arco de elipse. La elipse se toma encerrada por el rectángulo (especificado por los dos primeros puntos, $(x1;y1)$ y $(x2;y2)$); el comienzo y final del arco son determinados por la intersección de una línea desde el centro de la elipse, que pase por los puntos 3 $(x3;y3)$ y 4 $(x4;y4)$ respectivamente.

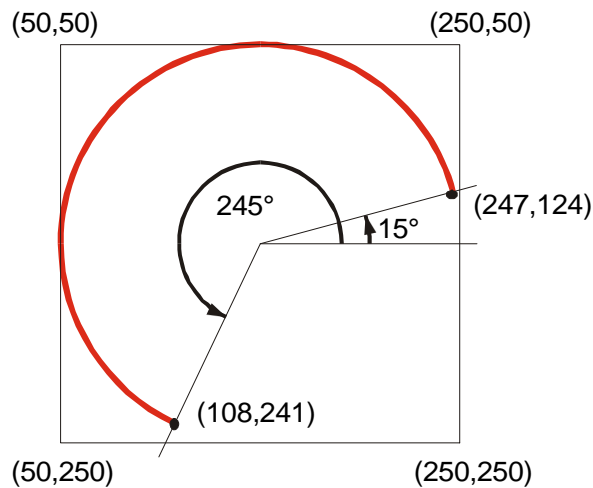


Es una forma un tanto extraña de expresar un arco, ¿no? Pero bueno, es lo que hay. Un par de ejemplos pueden ayudar a aclarar la situación:

Ejemplo 6:

Graficar un arco de circunferencia de radio 100 pixels, que comience a los 15 grados y termine a los 245 grados.

Usando un poquito de trigonometría básica (esa que aprendimos a los 10 años, más o menos) podemos llegar a una situación como la siguiente:



Usando la información del gráfico anterior, llegamos a la orden necesaria para dibujar la elipse (aproximadamente, recordemos que tenemos que redondear a nros. enteros):

```
Canvas.Arc(50,50,250,250,247,124,108,241);
```

Ejemplo 7:

Grafiquemos un arco de una elipse de eje mayor igual a 100 pixels y eje menor igual a 70 pixels, que vaya desde los 90 grados hasta los 180 (un cuarto de elipse). La situación es más fácil que la anterior, ya que ni siquiera es necesario calcular las funciones trigonométricas de esos ángulos:

```
Canvas.Arc(0,0,100,70,50,0,0,35);
```

Hay otra instrucción muy parecida, pero esta vez se cierra el arco con una línea entre los extremos y se rellena con el pincel activo (propiedad **Brush**):

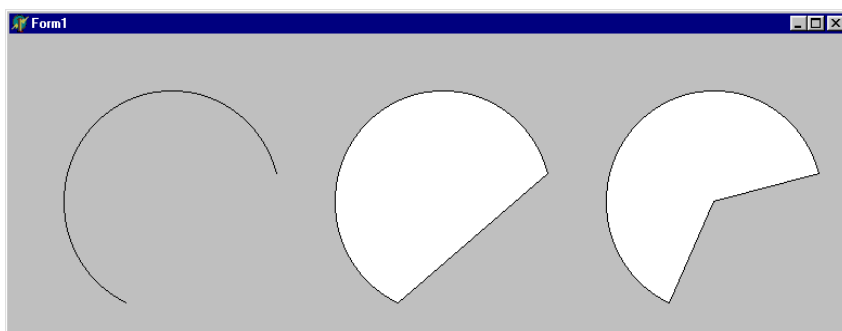
```
? procedure chord(x1,y1,x2,y2,x3,y3,x4,y4: integer);
```

Haciendo los mismos ejemplos que antes cambiando **Arc** por **Chord** podemos notar la diferencia entre los dos procedimientos. Y todavía hay una variante más:

```
? procedure Pie(x1,y1,x2,y2,x3,y3,x4,y4: Longint);
```

Dibuja una porción de una tarta, definida de la misma manera que para Chord o Arc.

La diferencia entre estas dos últimas funciones está en la manera en que cierran la figura, como podemos comprobar dibujando con las mismas coordenadas. En la figura siguiente se muestra el resultado del ejemplo ?, realizado con las tres funciones anteriores.



El gráfico del ejemplo ?, hecho con **Arc** (izquierda), **Chord** (centro) y **Pie** (derecha)

Otro grupo de funciones se utilizan para pintar:

? Procedure fillRect(const rect:tRect);

Rellena el rectángulo especificado con el pincel actual (sin borde).

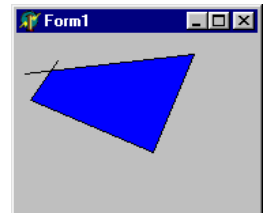
? procedure FloodFill(x,y:integer; color:tColor; FillStyle:tFillStyle);

Rellena la superficie que contiene al pixel (x,y).

si FillStyle es **fsBorder**, se rellena hasta que se encuentra el color especificado.

Si FillStyle es **fsSurface**, se rellena la superficie mientras el color sea igual al especificado en el parámetro **Color**; se detiene cuando encuentra un color distinto.

Ejemplo 8: Queremos rellenar con color azul la superficie limitada por un polígono; pero resulta que no usamos **Polygon** para construir éste, sino que lo hicimos con líneas sueltas. No nos queda más remedio que usar **FloodFill**:



```
with Canvas do begin
  Pen.Color:= clBlack; //Color de las líneas de borde
  MoveTo(30,20);
  LineTo(10,50);
  LineTo(100,90);
  LineTo(130,15);
  LineTo(5,30);
  Brush.Color:= clBlue; //Con este color se pinta
  FloodFill(20,40,clBlack,fsBorder);
end;
```

Aquí he utilizado un poco los objetos de pluma (Pen) y pincel (Brush) para asegurarme que el resultado será el esperado; enseguida estudiaremos estos objetos.

Métodos para trabajar con texto

? procedure TextOut(x,y: integer; const text: string);

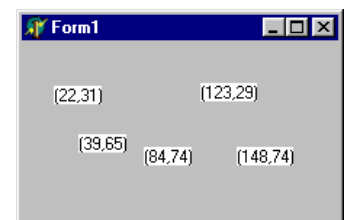
Escribe el string comenzando en la posición (x,y) con el font actual.

Ejemplo 9:

Hagamos un programa que cuando se presione el botón izquierdo del ratón en la superficie de la ventana nos escriba *ahí mismo* las coordenadas en que se presionó el botón.

El primer evento que se nos viene a la mente es el **OnClick**; bueno, pues destiérrenlo de sus mentes inmediatamente porque este evento no nos brinda la información de la posición del cursor en el momento de producirse. En su lugar, debemos usar **OnMouseDown** o **OnMouseUp**. He aquí el código, y una muestra del resultado:

```
procedure TForm1.FormMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState;
  X, Y: Integer);
begin
  canvas.TextOut(x,y,Format('(%d,%d)', [x,y]));
end;
```



? procedure TextRect(Rect: TRect; X, Y: Integer; const Text: string);

Esta función escribe un texto limitándolo a un determinado rectángulo. Si el texto se sale de los límites, no aparece en pantalla.

Ejemplo 10:

Modificaremos el ejemplo anterior para que solamente se escriban las coordenadas si estamos dentro de un rectángulo de esquinas (50,50) y (200,150):

```

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  canvas.TextRect(Rect(50,50,200,150),x,y,Format('%d,%d',[x,y]));
end;

```

Ahora se dibuja el rectángulo primero (se borra todo lo que está debajo) y si el texto se sale de los límites del rectángulo, se recorta.

? function TextHeight(const Text: string): Integer;

Esta función devuelve la altura en pixels del texto enviado como parámetro.

? function TextWidth(const Text: string): Integer;

Esta función devuelve el ancho en pixels de una determinada cadena.

? function TextExtent(const Text: string): TSize;

Devuelve los dos datos anteriores juntos en una estructura Tsize, definida como

```

type
  TSize = record
    cx: Longint;
    cy: Longint;
  end;

```

Ejemplo 11:

Vamos a escribir otra variación al ejemplo anterior de las coordenadas; ahora queremos que en lugar de escribir las coordenadas escriba "Hola, mundo!" recuadrado como se ve en la imagen. El código podría ser algo así:



```

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  s: string;
  r: tSize;
begin
  s:= 'Hola, mundo!';
  canvas.TextOut(x,y,s);
  r:= canvas.TextExtent(s);
  Canvas.FrameRect(Rect(x-3,y-3,x+r.cx+3,y+r.cy+3));
end;

```

Note que en el código anterior se deja un margen de 2 pixels vacíos entre el texto y el recuadro (el recuadro mismo ocupa un pixel de ancho, por lo que separamos uno de otro en 3 pixels).

La vida color de ¿¿clPink??

Los colores en Delphi se representan con números enteros de cuatro bytes, donde cada uno tiene un significado preciso. Para representar colores se ha definido el tipo Tcolor, que no es más que un subrango de los enteros.

Hay una serie de constantes predefinidas en la unidad Graphics, que representan colores comunes de la paleta de sistema o colores de elementos comunes definidos por el usuario en el Panel de Control (por ejemplo, clBtnFace es el color que elegimos en el Panel de Control para el frente de los botones -usualmente gris). Estas constantes empiezan con las letras **cl**, y en la ayuda se puede ver un listado con sus equivalencias buscando **Tcolor** en el índice.

Pero no estamos restringidos a las constantes predefinidas; podemos crear colores intermedios trabajando con el modelo RGB (Rojo, Verde, Azul).

Veamos el significado de cada byte de una variable TColor:

Byte	Significado
------	-------------

3 (más peso)	Paleta. 0= paleta del sistema 1= paleta actual 2= paleta lógica del dispositivo
2	Color Azul
1	Color Verde
0 (menos peso)	Color Rojo

así, el valor hexadecimal \$00FF0000 representa azul puro, \$0000FF00 verde puro y \$000000FF rojo puro. \$00000000 es negro y \$00FFFFFF es blanco. Los valores intermedios se “mapean” (se busca la mejor coincidencia) en la paleta de colores indicada por el byte más alto.

Sin entrar en detalles técnicos, los dispositivos gráficos pueden mostrar generalmente una cantidad limitada de colores distintos a la vez; por ejemplo, un adaptador de video puede ser capaz de representar cualquier color pero solamente 256 colores diferentes a la vez. Entonces se define una *paleta*, una lista de 256 colores que serán los que se verán simultáneamente. Podemos cambiar las entradas de esta paleta, pero no la cantidad. Cuando estamos ante este caso (el más común) se necesita el *mapeo* de colores: al presentar un color al sistema, éste lo busca en la paleta correspondiente. Si no lo encuentra, determina por algún método el color más cercano y selecciona éste.

Por suerte, en general no necesitaremos preocuparnos por este trabajo ya que el sistema lo realiza en forma automática y bastante bien. Delphi además modifica la paleta cuando se muestra una imagen por ejemplo en un componente TImage, para que los colores de ésta se vean sin distorsión. Es posible que los colores del resto de la pantalla cambien (a veces drásticamente), pero algo hay que resignar...

En todos los ejemplos que siguen utilizaremos las constantes predefinidas.

Cambiar el pincel, la pluma o la letra

La clase TCanvas tiene cuatro propiedades fundamentales que representan otros tantos objetos de dibujo:

- ? **Pen:** objeto que modela la pluma utilizada para graficar líneas, ya sean bordes de figuras o líneas sueltas.
- ? **Brush:** objeto que modela el pincel utilizado para pintar figuras.
- ? **Font:** objeto que representa la fuente con la que se escribe.
- ? **Pixels:** array bidimensional de colores. Cada entrada de la matriz representa el color de un pixel del canvas.

Los tres primeros son objetos en si mismos, con propiedades que hacen muy fácil cambiar las características del canvas. El último es un arreglo de elementos de tipo **TColor**.

Propiedades del objeto Pen (clase TPen)

- ? *Color:* indica el color de la pluma. El color se aplica a todas las líneas que se dibujen a continuación, hasta que se vuelva a cambiar o se destruya el Canvas.

Por ejemplo, para dibujar en rojo haríamos (antes de dibujar):

```
Canvas.Pen.Color := clRed;
```

- ? *Mode:* indica el modo de combinar los colores de la pluma y de la pantalla.

Para dibujar algo, debemos cambiar el color de algunos puntos de la pantalla; pero estos puntos ya tienen un color. Podemos decidir ser absolutistas y reemplazar directamente el color anterior con el nuevo, o ser un poco más democráticos y permitir que se mezclen de alguna manera.

La propiedad **Mode** indica la forma de mezclar los colores.

Los valores posibles son:

Modo	Color resultado
pmBlack	Siempre negro, no importa el color que demos a la línea
pmWhite	Siempre blanco
pmNop	Sin cambio; en la práctica, es como si la línea fuera transparente
pmNot	Inverso del color de fondo (resultado = NOT fondo)
pmCopy	Color de la pluma (valor por defecto)
pmNotCopy	Inverso del color de la pluma (resultado = NOT pluma)
pmMergePenNot	resultado = pluma AND (NOT fondo)
pmMaskPenNot	Combinación de colores comunes a la pluma y el inverso del fondo
pmMergeNotPen	resultado = fondo AND (NOT pluma)
pmMaskNotPen	Combinación de colores comunes al fondo y el inverso de la pluma
pmMerge	resultado = fondo AND pluma
pmNotMerge	Combinación inversa a pmMerge (NOT (fondo AND pluma))
pmMask	Combinación de colores comunes al fondo y a la pluma
pmNotMask	Combinación inversa a pmMask
pmXor	resultado = fondo XOR pluma
pmNotXor	Inversa de pmXor

? *Width*: ancho en pixels de la pluma.

? *Style*: estilo de la pluma (continua, punteada, a trazos, etc) *cuando se trabaja con pluma de 1 pixel de ancho*. Si la pluma es más ancha, siempre aparece continua!

Las constantes permitidas están listadas en la ayuda.

Propiedades del objeto Brush (clase tBrush)

Color: indica el color del pincel.

Style: estilo del pincel (sólido, con rayas verticales, horizontales, oblicuas, etc). Las constantes están listadas en la ayuda, con una muestra de las mismas aplicadas a un rectángulo.

Bitmap: un BMP de 8x8 pixels que se usará como patrón para el relleno.

Propiedades del objeto Font (clase tFont)

Color: color de la fuente a usar.

Name: nombre de la fuente (Arial, Times New Roman, etc)

Size: tamaño en pixels de la fuente.

Style: estilo de la fuente: es un conjunto que puede tener alguno de los valores siguientes (pueden ser varios):

? fsBold: negrita

? fsItalic: itálica

? fsUnderline: subrayada

? fsStrikeOut: tachada.

Impresión en Delphi

Hay dos formas principales de imprimir en Delphi: utilizando un motor de impresión -generalmente orientado a reportes de bases de datos, como Report Smith, Quick Report o Crystal Reports- o enviando los datos directamente al driver de la impresora. Veremos aquí los dos métodos.

Es tarea del programador determinar cuando es conveniente usar un reporte o generarlo “a mano”. Los dos métodos tienen puntos a favor y en contra. En general, podemos decir que para reportes simples o sobre pocos datos es preferible el enfoque directo por la velocidad de respuesta -es simplemente *irritante* cuando para obtener tres líneas impresas debemos esperar que se cargue el Report Smith, y después seguir esperando a que se descargue. En cambio, cuando la consulta sea complicada, implique gráficos, muchos datos, un formato especial de la página o cosas así, valdrá la pena la espera.

NOTA: en las versiones de 32 bits, Delphi provee un paso intermedio al Report Smith llamado Quick Report (en Delphi 3 ni siquiera viene incluido el Report Smith). Compuesto de varios componentes VCL, se integra en el ejecutable proveyendo facilidad de programación de reportes complejos con poca sobrecarga. Incluso hay una versión para Delphi 1, que se puede conseguir en el sitio WEB de QuSoft: <http://www.qusoft.com>.

Por supuesto, el último y definitivo determinante del método a utilizar será... el usuario final.

Impresión directa, sin utilizar un motor de impresión

Tengamos en cuenta que si no utilizamos un motor, cualquier subida parece más alta. Entonces ¿por qué no utilizar uno? Principalmente, por la sobrecarga que imponemos al sistema. Al momento de comenzar a imprimir, Delphi debe cargar en memoria los módulos que componen el “reportero”; esto no sólo limita los recursos disponibles sino que también demora más. Y a veces el resultado no justifica la espera. Cualquiera que haya utilizado Report Smith para hacer reportes sabe bien lo que estamos diciendo.

La impresión en Windows se hace a través de funciones de la GDI (Graphics Device Interface), que se encarga de ejecutar los comandos correspondientes al dispositivo específico que producirá la salida. Es como tener un subordinado a quien le decimos lo que queremos que haga, y que lo haga como pueda. Así que pueden sentirse un poco jefes... hasta que empiecen los problemas, que nunca faltan. Es parte del trabajo.

Cuando enviamos algo a la impresora, Windows se encarga de arrancar el Administrador de Impresión. Delphi define una clase para tratar con la impresión, otro intermediario más para hacer las cosas aún más fáciles. Esta clase se denomina **TPrinter** y está definida en la unit *Printers*, que debemos agregar manualmente a la cláusula **uses**.

Método 1: asignar la salida estándar a la impresora

Hay una forma fácil de enviar texto a la impresora: crear un *archivo de texto* y redireccionarlo al puerto de la impresora. A partir de entonces se puede escribir directamente utilizando los procedimientos estándar *Write* y *Writeln*; el tipo de letra, tamaño, espaciado, etc. son los seleccionados en la propiedad **font** del canvas de *Printer*. Sólo debemos escribir como hacíamos en DOS (que época...).

Para asociar un archivo de texto a la salida de impresora se utiliza el procedimiento **AssignPrn**¹. Notemos que debemos *crear* el “archivo” con **rewrite** antes de empezar a imprimir, y cerrarlo al terminar.

Ejemplo 1: imprimir el contenido de un memo en la impresora con el mismo tipo de letra de pantalla.

¹Se debe agregar la unit **Printers** a la cláusula **uses** para tener acceso al procedimiento *AssignPrn*.



Figura 1 El form principal de la aplicación de impresión directa

Creamos una aplicación muy sencilla; el form principal puede verse en la fig. 1 y el listado del evento `OnClick` del botón “Imprimir” en el listado 1. Note la utilización de un bloque de protección de código **try..finally** para asegurarnos que se libere la salida estándar aunque se produzcan errores en la impresión. El tema de los errores se tratará en el capítulo de *Excepciones*.

De esta manera se puede imprimir fácilmente un texto.

Método 2: el objeto `Printer`

Delphi provee una clase para facilitar el acceso directo a la impresora. En las primeras versiones se define una variable global, instancia de la clase `Tprinter`; en la versión 3 esta variable se hace local a la unit **printers** y se define en cambio una *función* llamada `Printer`. De esta manera se puede seguir usando la misma sintaxis. También en esta última versión se agrega una función llamada **SetPrinter** que sirve para cambiar el objeto de impresora global, pudiéndose definir varias instancias con diferentes parámetros.

Bueno, entonces la situación es la siguiente: tenemos una tarea que llevar a cabo -la impresión propiamente dicha- un “brazo ejecutor” de nuestras órdenes -la GDI- y un intérprete que traduce nuestro idioma de Delphinianos al lenguaje de funciones que entiende Windows -la instancia de la clase `TPrinter`. Por lo tanto, sólo tenemos que aprender las capacidades de esta última clase para lograr los resultados apetecidos. Y esto implica, como con cualquier clase, hablar de *Propiedades y Métodos*.

```
procedure TForm1.Button1Click(Sender:
TObject);
var
  t:TextFile;
  i: integer;
begin
  AssignPrn(t);
  rewrite(t);
  try
    Printer.Canvas.Font:= Mem1.Font;
    for i:= 0 to Mem1.Lines.Count-1 do
      writeln(t,mem1.lines[i]);
  finally
    CloseFile(t);
  end;
end;
```

Listado 1: El evento `OnClick` del botón de impresión

Propiedades de Tprinter

La propiedad principal de la clase tPrinter es el **Canvas**. Se utiliza de la misma manera que el canvas que contienen los objetos gráficos de la VCL -de hecho, es una instancia de la misma clase. Por lo tanto podemos utilizar las mismas rutinas gráficas (métodos de Tcanvas). No obstante, hay algunas diferencias con la salida de pantalla:

1. El plano sobre el que se miden las coordenadas de impresión es mucho más variable que el de un dispositivo de video; se hace necesario contar con alguna forma de interrogar a la impresora sobre su resolución, tamaño de página, en general sobre sus *capacidades*. Esto puede lograrse a través de una función de la API denominada **GetDeviceCaps**.
2. No hay garantía de que la impresora pueda recibir gráficos; los métodos de impresión de gráficos como *Draw*, *StretchDraw* o *CopyRect* pueden fallar.

Otras propiedades de interés son las que indican las dimensiones de la página: PageHeight (altura en pixels) y PageWidth (ancho en pixels). Un resumen de las propiedades de la clase Tprinter se da en la tabla siguiente:

Propiedad	Significado
Aborted	TRUE cuando el usuario ha cortado la impresión antes de terminar
Canvas	Superficie de graficación de la página
Capabilities (sólo 32 bits)	Indica capacidades de la impresora seleccionada: orientación, número de copias, intercalar copias.
Copies (sólo 32 bits)	Permite indicar o consultar el número de copias.
Fonts	Lista de las fuentes soportadas por la impresora
Handle	Manejador del contexto de dispositivo.
Orientation	Orientación del papel: <code>poPortrait</code> , <code>poLandscape</code>
PageHeight	Altura del canvas en pixels
PageNumber	Número de página actual
PageWidth	Ancho del canvas en pixels
PrinterIndex	Indice de la impresora seleccionada en la propiedad Printers

Propiedad	Significado
Printers	Lista de las impresoras instaladas en Windows (Strings)
Printing	Indica cuando un trabajo de impresión está en curso. Es puesta a True por el método BeginDoc y a False por EndDoc .
Title	El título del trabajo de impresión en el Administrador de Impresión

Los métodos de la clase tPrinter se sintetizan en la tabla siguiente:

Métodos de Tprinter

Método	Función
Abort	Llamando a este procedimiento se detiene un trabajo de impresión en curso
BeginDoc	Este método debe ser llamado siempre antes de comenzar a imprimir.
EndDoc	Este método termina un proceso de impresión.
GetPrinter	Devuelve la impresora actualmente en uso. No es necesario llamar a este método directamente, es más fácil acceder a la propiedad Printers.
NewPage	Inicia una nueva página. Incrementa el contador de páginas, pone la pluma en las coordenadas (0,0) y envía un código Form Feed a la impresora.
SetPrinter	Especifica la impresora a usar. No es necesario llamar a este método directamente, es más fácil acceder a través de las propiedades Printers y PrinterIndex.

Podemos esquematizar el proceso de imprimir usando este objeto en los siguientes pasos:

- 1) Iniciar el proceso de impresión llamando a Printer.BeginDoc
- 2) Dibujar el texto (TextOut) o gráficos (Rectangle, Ellipse, LineTo, etc) en el canvas de Printer teniendo cuidado de las coordenadas; posiblemente sea necesario utilizar las propiedades PageHeight y PageWidth.
- 3) Cuando termina una página (por ejemplo, cuando la línea siguiente se imprimiría en una coordenada Y mayor que PageHeight) llamar a Printer.NewPage para comenzar una nueva página en blanco.

4) Al terminar de imprimir, llamar a `Printer.EndDoc`.

Nota: normalmente, cuando se está imprimiendo se muestra una ventana con un botón que permite la cancelación del trabajo; se debe llamar al método `Printer.Abort`.

Veamos el trabajo con el objeto `Printer` en un ejemplo: imprimir los datos de un registro de una tabla, incluida una foto. Utilizaremos la tabla `CLIENTS.DBF` que viene con los ejemplos de Delphi y colocaremos algunos de los campos en un form como el de la figura 2.

La foto necesita un tratamiento especial: el campo `Picture` de la tabla contiene el nombre del archivo `.BMP`, que se encuentra en el mismo directorio que la tabla. Para mostrarlo utilizamos un componente `TImage` común, al cual tenemos que decirle que cargue la imagen correspondiente cada vez que nos paramos en un registro nuevo. Para ello utilizaremos el evento `OnDataChange` del `DataSource`, que se produce cada vez que cambian los datos que accede. El código se ve en el listado 2.

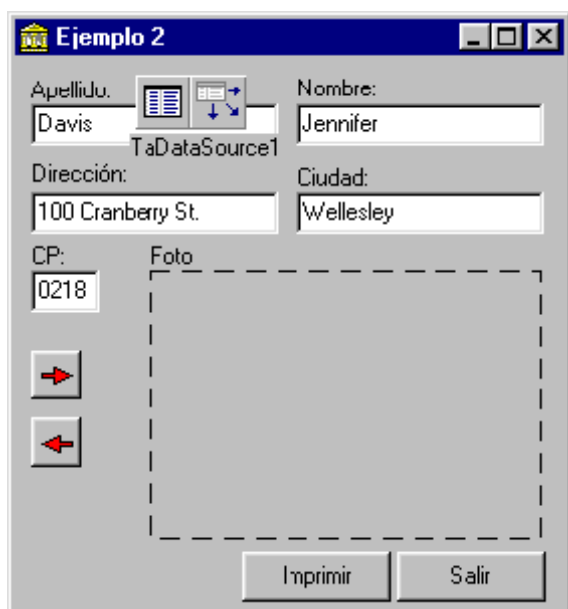


Figura 2 Form principal del ejemplo 2

Cuando vamos a imprimir, necesitamos algunos cálculos para posicionar los textos y la foto; una forma simple sería la indicada en el listado 3, que hace uso de la función `GetTextHeight` del canvas para determinar la altura de una línea de texto y le agrega 5 pixels para conseguir una separación entre líneas adyacentes. La imagen se redimensiona para que ocupe 1000x1000 pixels en una quinta línea.

```
procedure
TForm1.DataSource1DataChange(Sender: TObject;
Field: TField);
begin

Image1.Picture.LoadFromFile(Table1.DatabaseName+
'\'+Table1Picture.AsString);
end;
```

Listado 2: Cargar la foto cada vez que cambia el registro

```

procedure TForm1.Button1Click(Sender: TObject);
var
  i,tg:integer;
begin
  Printer.BeginDoc;
  try
    Table1.First;
    tg:= Printer.Canvas.TextHeight('tg')+5; //5 pixels de separacion entre lineas
    for i:= 0 to 0 do begin
      with Printer.Canvas do begin
        TextOut(10,tg*i*6,table1Last_Name.DisplayLabel);
        TextOut(600,tg*i*6,Table1Last_Name.AsString);
        TextOut(10,tg*(i*6+1),table1First_Name.DisplayLabel);
        TextOut(600,tg*(i*6+1),Table1First_Name.AsString);
        TextOut(10,tg*(i*6+2),table1Address_1.DisplayLabel);
        TextOut(600,tg*(i*6+2),Table1Address_1.AsString);
        TextOut(10,tg*(i*6+3),table1City.DisplayLabel);
        TextOut(600,tg*(i*6+3),Table1City.AsString);
        TextOut(10,tg*(i*6+4),table1ZIP.DisplayLabel);
        TextOut(600,tg*(i*6+4),Table1ZIP.AsString);
        StretchDraw(Rect(600,tg*(i*6+5),1600,tg*(i*6+5)+1000),Image1.Picture.Graphic);
      end;
    end;
  Printer.EndDoc;
end;

```

Listado 3: Impresión de la ficha

Utilización de Quick Report

Quick Report esta compuesto de una serie de componentes VCL que se deben *instalar* en Delphi 1; en las versiones de 32 bits viene incluido con el programa (no son hechos por Borland, están licenciados a QuSoft). En febrero de 1998, la versión existente es la 2.0. El hecho que Borland lo entregue integrado con Delphi ya habla en favor de la calidad de estos componentes. Son realmente muy buenos y poderosos; casi se diría que no necesitaremos más el Report Smith, y de hecho aquí ni siquiera hablaremos de este último.

Los reportes creados con Quick Report se integran por supuesto con el ejecutable -después de todo *son componentes*, no?, es decir que no necesitamos instalar nada extra en el equipo del cliente. Podemos hacer desde reportes muy simples hasta muy complicados, utilizando varias tablas, gráficos y memos. Además se puede mostrar una vista previa antes de imprimir. Y se rumorea que una próxima versión

podrá hacer caramelos, limpiar la casa y planchar la ropa. ¿Para qué más?

Algunos componentes se encargan de manejar el reporte completo (de manera similar a los componentes de acceso de datos, que se encargan de las tablas o bases de datos completas) mientras otros son los controles *imprimibles*, los que ponen la información en la impresora (equivalentes de los Controles de Datos).

Los componentes de acceso que tenemos disponibles son los siguientes:

Componente	Función
QuickRep	Representa el reporte en sí. Es el componente principal e indispensable. Contiene propiedades que controlan el tamaño de página a imprimir, los márgenes, etc.
QRSubDetail	Banda de "Sub Detalle", utilizada en reportes con tablas master/detail
QRBand	Banda del reporte. Las bandas más comunes en un reporte se pueden colocar automáticamente con la propiedad <i>Bands</i> del reporte.
QRChildBand	Banda de detalle que se estira en forma acorde con el contenido.
QRGroup	Banda de agrupación.
QRCompositeReport	Banda que permite combinar reportes.
QRPreview	Componente que permite crear vistas previas especiales.
QRStringsBand	Banda que toma los datos de una lista de Strings (Tstrings) en lugar de una Tabla.

Los componentes imprimibles son los siguientes:

Componente	Función
QRLabel	imprime el texto que tenga en la propiedad <i>Caption</i> . Es equivalente al Label pero sólo se ve en un QuickRep.
QRDBText	imprime el contenido de un campo de una tabla, equivalente al DBText. Sirve también para imprimir campos Memo. El formato de la salida se toma del formato ya definido para el campo, o de la propiedad <i>Mask</i> de este componente.
QRExpr	imprime el resultado de una expresión, ingresada en la propiedad <i>expr</i> .
QRSysData	imprime información del sistema, como el nro de página o la fecha.
QRMemo	imprime texto fijo que puede ocupar varias líneas. Equivalente al Memo.

Componente	Función
QRExprMemo	un memo con expresiones insertadas, que se evalúan al momento de imprimir.
QRShape	imprime una figura geométrica. Equivalente al Shape.
QRImage	imprime una imagen BMP, WMF o ICO. Equivalente a Image.
QRDBImage	imprime el contenido de un campo gráfico de una tabla. Equivalente al DBImage.
QRRichText (sólo 32 bits)	imprime texto con formato (Rich Text). Equivalente al RichEdit, se puede enlazar con un componente de éstos para imprimir su contenido.
QRDBRichText (sólo 32 bits)	imprime el contenido de un campo de texto con formato (Rich Text) de una tabla. Equivalente a DBRichEdit.

Existen además, a partir de la versión 3 de QR, los llamados *filtros* de exportación. Aunque se pueden crear otros, los más comunes vienen incluidos y son los siguientes:

Componente	Función
QRTextFilter	Permite exportar a un archivo de texto (.TXT)
QRCSVFilter	Permite exportar a un archivo delimitado por comas (.CSV)
QRHTMLFilter	Permite exportar a un archivo HTML (.HTM)

Al agregar cualquiera de estos últimos componentes a una ficha, aparecerá la opción correspondiente en el comando de grabación de la vista previa. Veremos un ejemplo de exportación por programa un poco más abajo.

Impresión de datos estáticos

La información anterior es sólo para referencia. Vayamos al grano: vamos a construir el reporte más simple posible, que muestre la frase que hizo famoso a Galileo Galilei (aunque algunos dicen que no pronunció nunca estas palabras, y aunque las haya dicho debe haber sido en voz muy baja para que no lo escuchen los del jurado y entonces uno se pregunta si realmente alguien las escuchó o es solamente una leyenda más, como la manzana que dicen que le cayó en la cabeza a Newton pero algunos investigadores lo niegan o el séptimo hijo varón que se transforma en *hombre lobo* en las noches de luna llena o... en fin, *Eppur si muove*)

Para hacer un reporte con Quick Report necesitamos como mínimo los siguientes componentes:

- * una ficha que servirá como “soporte” del reporte
- * un componente QuickRep
- * algún componente imprimible de Quick Report, como un QRLabel.

En nuestro pequeño ejemplo vamos a crear un form principal muy simple con dos botones, uno para cerrar la aplicación y otro para mostrar la vista previa del reporte. Una posible versión de esta ventana se ve en la fig. 3:

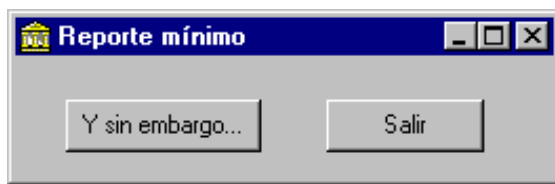


Figura 3 ventana principal para el ejemplo 1

En el evento `OnClick` del botón “Salir” hacemos que se cierre la aplicación. No hacen falta más explicaciones para esto, ¿o si?.

En el otro botón debemos pedir que se muestre la vista previa del reporte básico.

¿Y dónde está el reporte?

Para aquellos de Uds. que lo hayan adivinado, tenemos que crear el reporte antes de usarlo. Y para ello necesitamos como ya dijimos un form y un componente QuickRep. Entonces vamos a agregar otra ventana a la aplicación, en la cual pondremos solamente un componente QuickRep. En la fig. 4 se ve la ficha en tiempo de diseño.

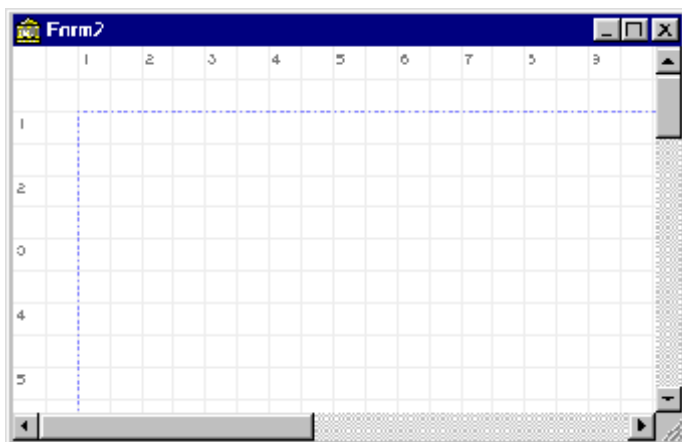


Figura 4 El form del reporte en diseño

Normalmente, no mostraremos en tiempo de ejecución la ventana de un reporte en la pantalla; sólo sirve como soporte para el diseño. En cambio debemos llamar a un método del componente QuickRep: *Preview* para mostrar una vista previa en la pantalla o *Print* para imprimir directamente en la impresora. (Si alguno de Uds. quiere probar cómo se ve un form con un reporte en ejecución, puede mostrarlo poniendo su propiedad *visible* en **true**. No es una vista muy edificante).

Veamos. En la ventana principal, asignamos el código del listado 4 al evento `OnClick` del botón que nos permitirá ver la vista previa. Al presionar este botón se nos presentará la ventana de vista previa que incluye por defecto el Quick Report (fig. 5). Desde

esta ventana podemos ya imprimir, presionando el botón correspondiente de la barra de herramientas.

La verdad que imprimir una hoja vacía no era nuestro objetivo, ¿no?. Agregamos entonces al reporte un componente QRLabel con el texto deseado en la propiedad *caption* del mismo. Ahora sí, la vista previa nos muestra el texto y si enviamos esto a la impresora, saldrá tal cual como se ve en pantalla.

```
procedure TForm1.Button1Click(Sender:
TObject);
begin
    Form2.QuickRep1.Preview;
end;
```

Listado 4: Código para ejecutar una vista previa

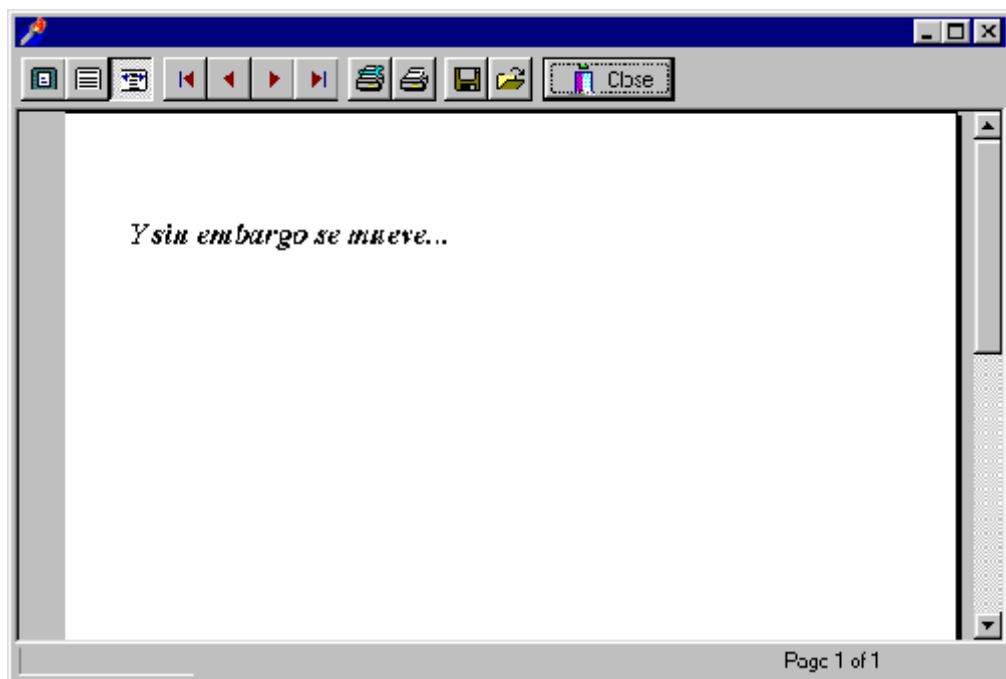


Figura 5 Vista previa del reporte

Esta técnica es una alternativa al acceso directo al dispositivo a través de la clase `tPrinter`. No obstante, tenga en cuenta que el solo hecho de utilizar el Quick Report nos agrega una sobrecarga de cerca de 400 Kb. al ejecutable. Para imprimir una frase o algo relativamente simple, tal vez sea más conveniente utilizar el método anterior.

Impresión de texto simple

A partir de la versión 3 de QuickReport, tenemos la opción de crear un reporte utilizando una lista de Cadenas en lugar de una tabla para obtener los datos; esto nos da la opción de obtener las líneas de un

componente Memo o mejor aún, de un archivo de texto.

La banda de Cadenas se comporta como una banda de detalle, imprimiéndose una por cada línea de texto. Estas líneas se almacenan en una propiedad llamada *Items*, de tipo **Tstrings**.

Ahora bien, ¿con qué componente mostramos el texto? La respuesta viene de la mano del componente QRExpr. Con él se puede acceder a las líneas individuales de texto de la banda de Cadenas: basta poner el nombre de la banda en la propiedad **Expression** del componente.

Entonces, se nos prende la lamparita para resolver un viejo problema de una forma nueva: la impresión de un archivo de texto. Basándose en lo que dijimos anteriormente, trate de crear una aplicación que lea un archivo de texto y lo imprima usando un reporte... antes de leer la solución siguiente!

Usaremos una ventana como la de la figura ??? para seleccionar el archivo a imprimir. En otra ficha colocamos el reporte, con la banda de cadenas y un componente QRExpr para acceder a las cadenas del archivo. El botón de imprimir debe abrir el archivo de texto en la propiedad *Items* de la banda de cadenas, y luego llamar al método *Preview* del reporte. En la figura ??? se ve la ficha del reporte y en la figura ??? la vista previa del archivo WIN.INI.

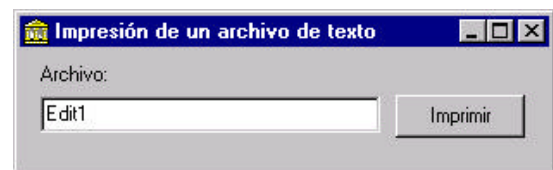


Figura 6 la ficha principal del ejemplo



Figura 7 ficha del reporte en tiempo de diseño

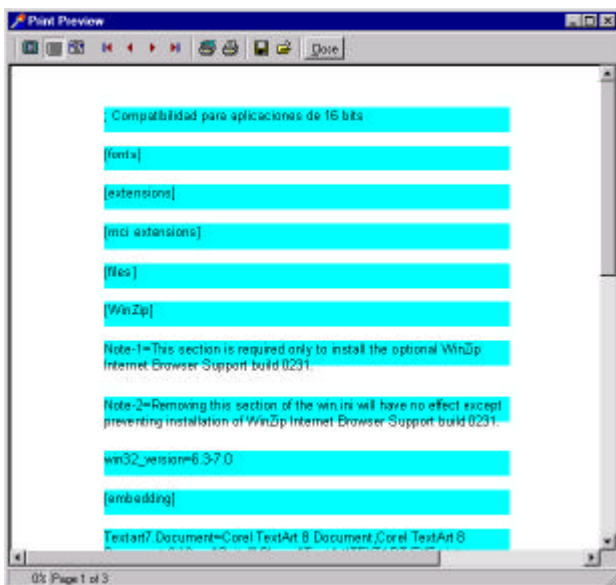


Figura 8 impresión del archivo WIN.INI

NOTA: en las imágenes del ejemplo anterior se ve la banda de cadenas con un color distinto para resaltarla

Por último, recordemos que también podemos poner “a mano” las líneas de texto que queremos escribir; simplemente agregamos (utilizando el método **Add**) las líneas a la propiedad **Items** de la banda de cadenas. Se escribirá una banda por cada línea de texto.

Reporte con datos de una tabla

El Quick Report fue pensado para simplificar la tarea de imprimir los datos de una tabla o resultado de una consulta; veremos ahora las facilidades que se nos brindan para ello.

Para crear un reporte basado en el contenido de una base de datos, necesitaremos también los elementos básicos mencionados anteriormente así como otros que introduciremos aquí. Comenzamos creando una ventana principal con el experto para formularios de bases de datos, tomando los datos de la tabla ANIMALS de los ejemplos que trae Delphi. Indiquemos al experto que arme un DataModule además del form, y que haga que la ventana creada sea la principal de la aplicación. Cuando compilamos y corremos la aplicación veremos la ventana con los datos. Hasta aquí todo normal (fig. 6).

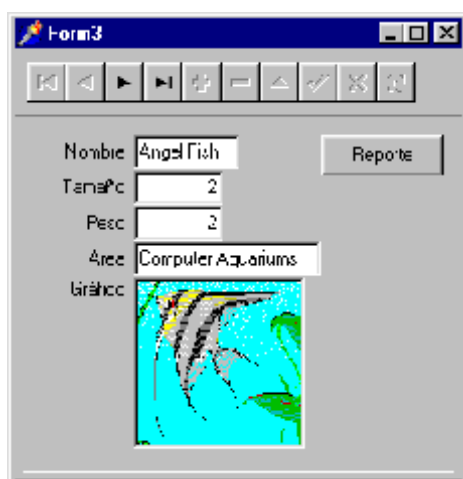


Figura 9 Ventana principal, creada con el experto

Ahora comenzamos con el reporte. Agreguemos a la ventana principal un botón para hacer una vista previa, y en otro form ponemos un componente QuickRep, un label y un QRDBText enlazado con el campo *nombre* de la tabla (será necesario indicar a Delphi que vamos a usar el DataModule con la opción File|Use Unit). Fig. 7.

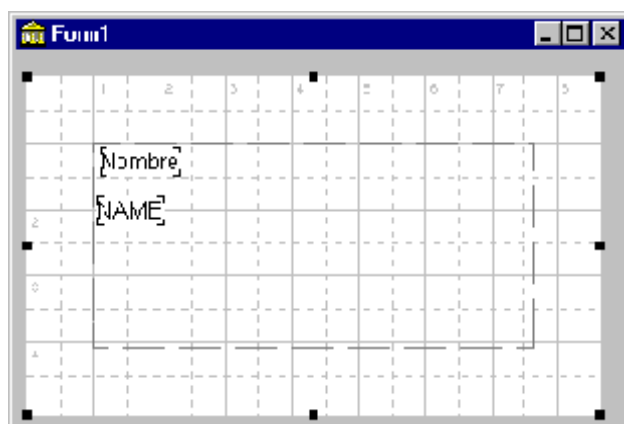


Figura 10 Ventana del reporte con los componentes ya colocados

Pruebe la aplicación.

No es lo que deseábamos, ¿ no? Podemos ver que se muestra un solo valor -el último de la tabla². Lo que es peor, la tabla queda en el último registro como podemos ver en el form principal³. Para ver todos los registros será necesario agregar *bandas* a nuestro reporte.

La división de un reporte en bandas no es algo nuevo; Access lo viene haciendo desde sus comienzos. El concepto es simple: **una banda es una parte del reporte, que se imprime sólo en un momento determinado**. Por ejemplo, tenemos una *banda de título* que se imprimirá una sola vez al principio del reporte, bandas de *encabezado de página* que se imprimen al comienzo de cada nueva página, y así. Los tipos de bandas disponibles y el momento en que se imprimen se ven en la tabla siguiente:

Tipo de banda	Momento de impresión
Título	Principio del reporte, después del primer encabezado de página si está activa la opción <i>FirstPageHeader</i> en <i>Options</i> del reporte.
Page Header (encabezado de página)	Al principio de cada página nueva. La opción <i>FirstPageHeader</i> de la propiedad <i>Options</i> del reporte indica si se imprimirá esta banda en la primera página del reporte.
Page Footer (pie de página)	Al final de cada página. La opción <i>LastPageFooter</i> de la propiedad <i>Options</i> del reporte indica si se imprime esta banda en la última página del reporte.
Detail (detalle)	Una vez por cada registro. Para que se imprima debe haber un enlace entre el reporte y un componente de acceso a datos como un Table, Query o StoredProc a través de la propiedad <i>DataSet</i> del reporte.
Column header (cabecera de columna)	Se imprime al principio de cada columna. Es el lugar ideal para poner los títulos de las columnas del detalle.
Summary (sumario, resumen)	Al final del reporte, antes del último pie de página si está permitido (ver Page Footer)
GroupHeader (cabecera de grupo)	Al principio de un grupo. En la propiedad <i>Expression</i> se escribe la expresión que define el grupo; cada vez que cambia el valor de la expresión se inicia un nuevo grupo y se imprime esta banda.
Group footer (pie de grupo)	Al final de un grupo.

² Cuando se especifica una tabla en la propiedad *DataSet* del Reporte, éste siempre recorrerá todos los registros de esa tabla. Si no usamos bandas, imprime el último.

³ Este último problema tiene una solución sencilla: utilizar otro componente Table o Query apuntando a la misma tabla

Ahora podemos crear nuestro reporte de la tabla, utilizando bandas. El objetivo es lograr un reporte como el de la fig. 8

Agregamos entonces un componente Query al form del reporte -usaremos uno diferente al que utilizamos para el form principal que muestra los datos en la pantalla. La instrucción SQL utilizada en este ejemplo es la siguiente:

```
select name,area from animals
```

Debemos enlazar la propiedad *DataSet* del reporte con este componente y **activarlo** para que funcione.

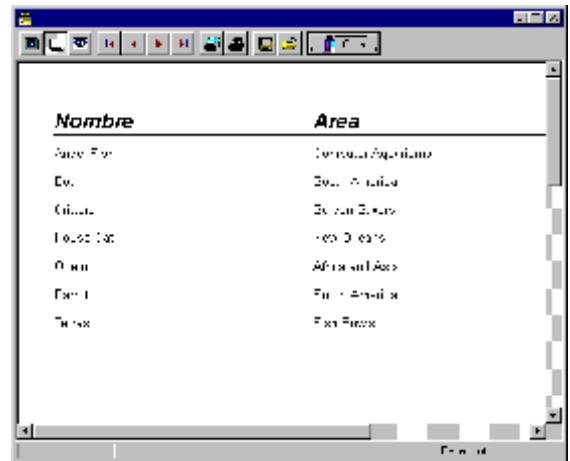


Figura 11 vista previa de un reporte utilizando una consulta SQL

Ahora agregamos las bandas para lograr el resultado deseado. Necesitamos una banda para los títulos y otra para los datos. Según consta en la tabla, pondremos una banda tipo *Column header* para los títulos y una tipo *Detail* para los datos.

Seleccionamos estas bandas en la propiedad **Bands** del reporte (fig. 9)

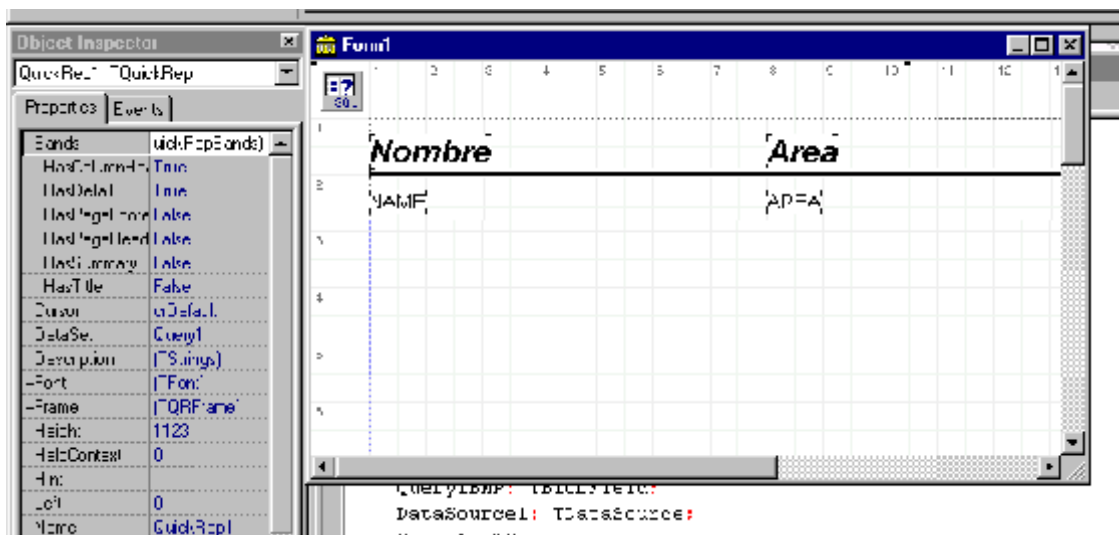


Figura 12 el reporte en modo de diseño, con las bandas y componentes imprimibles ya situados.

Note que ya hemos ubicado un par de etiquetas -QRLabels- y una línea horizontal -QRShape- en la banda de encabezados de columna, y los correspondientes QRDBText en la banda de detalle, para

mostrar el contenido de los campos indicados.

En los componentes QRDBText debemos indicar el origen de los datos en la propiedad *DataSet* (al igual que en el reporte) y el campo que queremos mostrar en la propiedad *DataField*. Y ya está, ya tenemos el reporte de todos los registros de la tabla. Pueden probarlo.

Como ejercicio, dejamos al lector la tarea de hacer que el form del reporte se cree en el momento en que lo vamos a usar -en otras palabras, que no se cree automáticamente al inicio del programa, que no es necesario- y hacer que sólo se impriman los registros de los animales que pesan menos de 10 kg. Asimismo, convendría que agreguen bandas de título, de encabezado y pie de página y de resumen para ver el comportamiento de las mismas.

Imprimir un solo registro de una tabla

Hay otro caso que se puede presentar: el de imprimir un solo registro de una tabla, que no sea el último.

Podemos solucionar este problema de varias maneras:

- C Si el reporte se basa en una consulta SQL (componente Query), podemos agregar una clausula WHERE que seleccione sólo el registro deseado y proceder comúnmente.

Por ejemplo, en la tabla **Animals.dbf** podríamos hacer

```
select * from animals where name="ocelot"
```

También podríamos haber utilizado un parámetro para el nombre.

- C Si el reporte se basa en una tabla, debemos filtrar la tabla aplicando un rango a los datos (SetRange) que seleccione sólo el registro deseado.

En la tabla **Animals.dbf**:

```
SetRange(['ocelot'], ['ocelot']);
```

Claro que habrá que seleccionar antes el índice **Name** para la tabla!

El valor que identifica al registro deseado (en nuestro ejemplo, el nombre del animal) puede venir de un editor o algún otro medio de interacción con el usuario.

Reporte Principal/Detalle (Master/Detail)

Haremos ahora un ejemplo más complejo: un reporte basado en dos tablas relacionadas (*master/detail*) y con los datos agrupados. Además agregaremos componentes QRExpr y QRSysData para mostrar datos calculados al momento de ejecutar el reporte.

El caso de reporte principal/Detalle que trataremos es el típico de una Orden de Pedido. La orden de pedido se compone de datos que están normalizados en dos tablas, **Orders** e **Items**. La primera actúa como tabla principal, aportando los datos globales del pedido; la segunda debe mostrar únicamente los registros que componen el detalle de *ese* pedido particular. Este filtrado se obtiene fácilmente enlazando la tabla *secundaria* con la principal a través de las propiedades **MasterSource**, **MasterFields** e **IndexFieldNames**.

Dejemos de lado el detalle de los pedidos, por el momento. El reporte de los datos de la tabla principal únicamente es igual a lo que hemos hecho antes con los animales, es decir: asociamos el reporte a la tabla de Pedidos y colocamos una banda de detalle con los componentes correspondientes, cada uno enlazado al campo que tiene que mostrar. Hasta aquí todo normal.

Ahora queremos agregar debajo de cada pedido un listado de los datos del detalle de ese pedido; como sabemos, estos datos están en la tabla **Items**. Suponemos que la tabla de Items ya está enlazada con la de Pedidos, de manera que al momento de imprimir los datos de un pedido la tabla secundaria ya se encuentra filtrada. Ahora, QuickReport debería recorrer toda la tabla secundaria imprimiendo los registros. Normalmente esta operación la realiza con la banda de detalle, pero ya está ocupada con la tabla principal. La banda que corresponde ahora es una llamada **SubDetalle**.

La banda de SubDetalle trabaja de la misma manera que la banda de detalle, pero con una tabla secundaria. Esta banda tiene su propiedad **Dataset** individual, que debemos asociar a la tabla secundaria. Por cada registro de la tabla principal, se recorrerá la tabla secundaria completa -recordemos que está filtrada- y se imprimirá una banda de subdetalle por cada registro de ésta.

Podemos incluso colocar un título antes de listar los registros de la tabla secundaria; en la banda de SubDetalle tenemos una propiedad llamada *HeaderBand* que puede apuntar a una banda que usaremos como Banda de Cabecera. Esta nueva banda -de tipo *rbGroupBand*- se imprime *después* de la banda de detalle y *antes* del grupo de bandas de SubDetalle. Aquí colocaremos controles de tipo *QRLabel* con los títulos de las columnas de la tabla secundaria que se imprimirán debajo.

Bueno, si han podido seguir la explicación hasta aca... los felicito! Les recomiendo intentar hacer el reporte sólo con esto, antes de ver la explicación paso a paso más adelante. Después de cometer los errores una vez, les será mucho más fácil evitarlos la próxima.

Desarrollo del ejemplo paso por paso

Usaremos las tablas de ejemplo que vienen con Delphi **Orders** (master) e **Items** (detail). Primero hacemos un form principal con las tablas relacionadas y un navegador para ver los datos antes de imprimir. Fig. 10.

El navegador permite moverse en la tabla principal. Ahora veamos el reporte que, como habrán adivinado, se activa al presionar el botón.

Los pasos son los siguientes:

1. Agregar un nuevo form al proyecto. *Todas las acciones siguientes se refieren a este form.*
2. Poner un componente QuickRep
3. Agregar un componente Table y asociar a la tabla Orders (Table1).
4. Agregar un componente DataSource (DataSource1) y asociar a la tabla anterior
5. Agregar otro componente Table, asociarlo a la tabla Items y relacionar con Table1 a través de la propiedad MasterSource por el campo OrderNo
6. Indicar al reporte que obtenga los datos para el detalle de la tabla principal, seleccionando Table1 en la propiedad DataSet

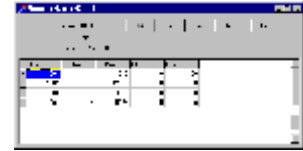


Figura 13 la ficha principal del ejemplo de reporte de tablas relacionadas.

Tenemos situados ya los componentes de acceso a los datos con los enlaces correspondientes. Vamos ahora a colocar las bandas.

1) Banda de detalle de la tabla principal

Aquí se mostrarán los registros de la tabla principal, es decir, los datos de las órdenes. En la propiedad *Bands* del reporte, ponemos en *True* la banda *detail*. Se agrega una banda al reporte.

Coloquemos los componentes imprimibles: dos QRDBText para mostrar los campos **OrderNo** y **ShipDate**; se pueden poner además etiquetas para indicar qué valores estamos viendo.

Ahora ya podemos hacer una vista previa; se muestran los datos de la tabla principal.

2) Banda de detalle de la tabla secundaria (sub detail)

Agregue un componente *QRSubDetail* desde la paleta de componentes. La propiedad *DataSet* debe apuntar a la tabla secundaria.

Agregamos ahora los componentes imprimibles: QRDBText para mostrar los campos **PartNo** (número de parte, código), **Qty** (cantidad) y **Discount**. (descuento).

Se puede hacer ya una vista previa y tendremos todos los datos. No obstante, faltaría un lugar para poner los títulos de las columnas del detalle secundario. Estos títulos se deben imprimir una vez antes del comienzo de cada grupo de registros secundarios: necesitamos una banda de *cabecera de grupo*.

Agregamos entonces al reporte una banda (QRBand) y ponemos su propiedad *BandType* en **rbGroupHeader**. Llamemos a esta banda **TitulosDeGrupo1**. Ahora seleccionamos la banda de detalle secundario (SubDetail) que pusimos antes y seleccionamos la banda de título recién creada en la propiedad *HeaderBand*. Las bandas se reacomodan. Agregamos etiquetas para títulos de las columnas y

una línea gráfica y llegamos a un resultado como el de la fig. 11.

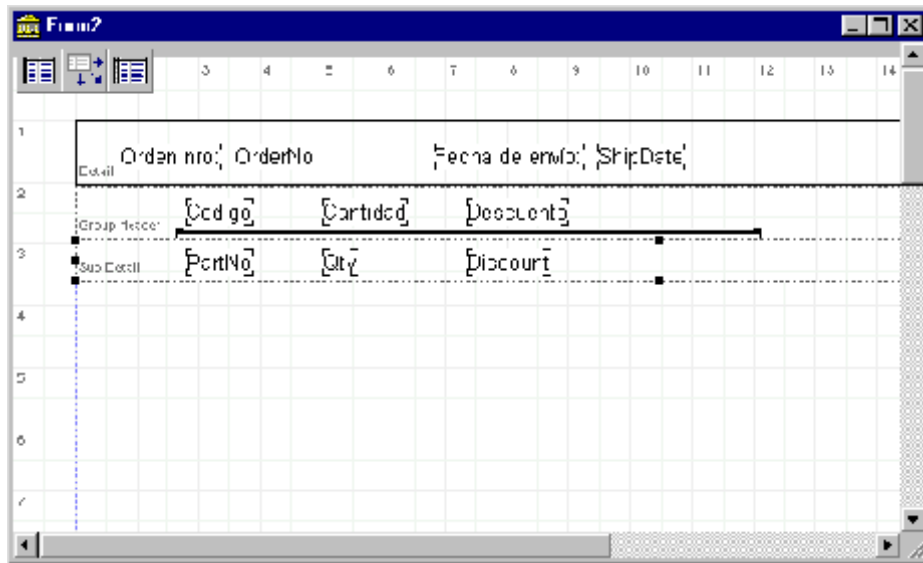


Figura 14 el reporte master/detail en diseño

El reporte está casi terminado; pueden admirar su obra en la vista previa.

Agregaremos un toquecito de “cosmética” que lo hará lucir más profesional: agregaremos un pie de página con el número de página.

En la propiedad *Bands* del reporte ponemos en True la opción **HasPageFooter**. Aparece la banda de pie de página, en la cual agregamos un componente **QRLabel** con el *caption* igual a “Página” y a su derecha un componente **QRSysData** con la propiedad *Data* en **qrsPageNumber**. Llevamos estos componentes a la derecha de la banda y ¡Voilà! Ya tenemos el número de página al pie de cada una.

De la misma manera podríamos agregar un encabezado de página con la fecha o algún otro dato de interés. Es así de simple... La vista previa resultante de este programa se ve en la fig. 12.

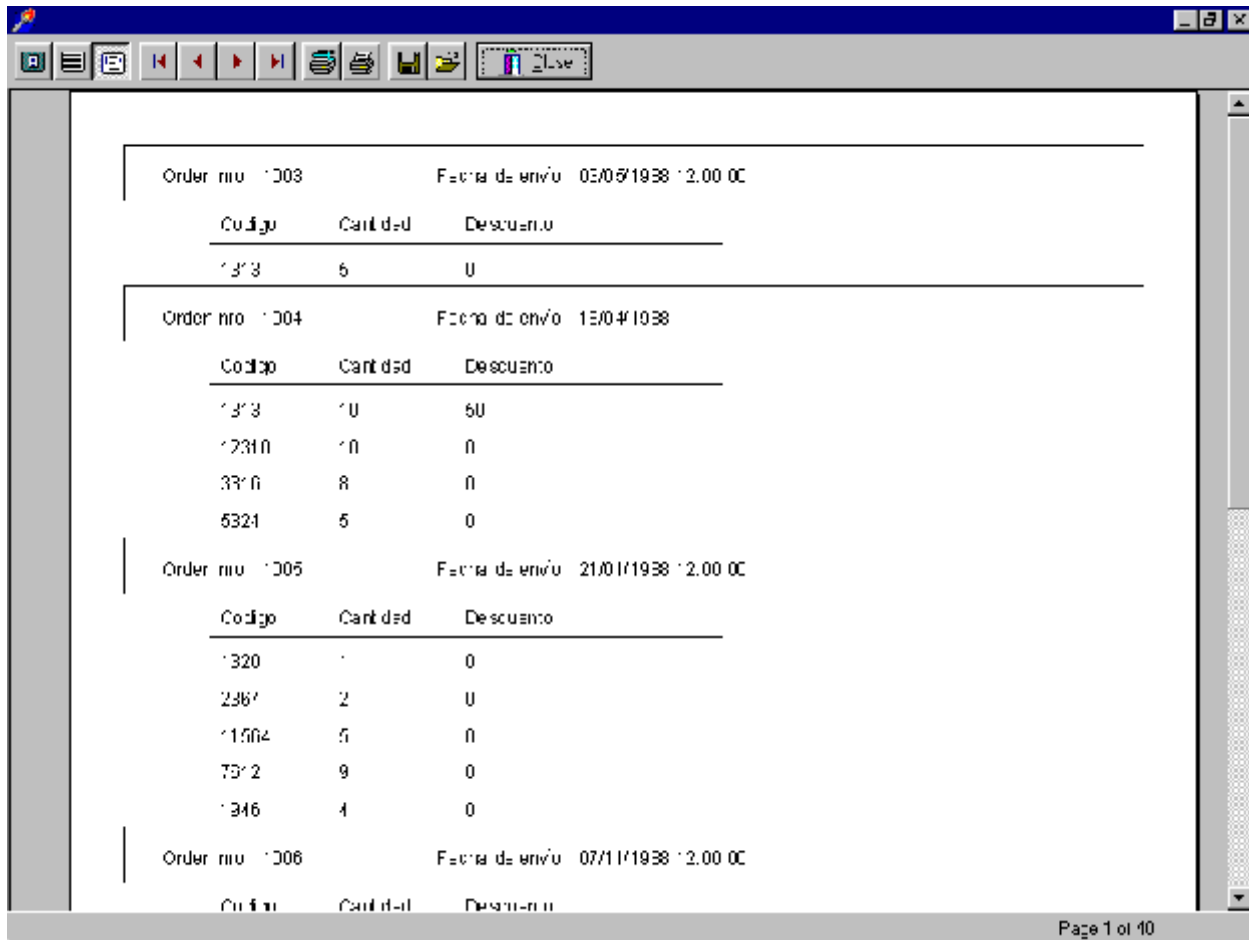


Figura 15 Vista previa del reporte Master/Detail

Impresión de órdenes seleccionadas

Muy bien, tenemos el listado de todas las órdenes. Pero ahora queremos imprimir sólo algunas, las que cumplan con un criterio establecido -por ejemplo, las que tienen fecha de envío en el mes actual; o solamente la que estamos mirando en la pantalla de entrada de datos. ¿Cómo podemos lograr eso con QuickReport?

Recordemos que ya nos topamos con este problema cuando trabajamos con una sola tabla -la de animales. La solución en este caso es exactamente igual, sólo hay que tener cuidado de aplicar el filtrado a la tabla *principal* (la secundaria se filtra sola gracias a la relación).

Por ejemplo, modifiquemos el programa anterior para que muestre la vista previa solamente de la orden que estamos mirando en la pantalla principal. El cambio viene en el evento `OnClick` del botón de Vista Previa:

```
procedure TForm3.Button1Click(Sender: TObject);
begin
    form2.table1.SetRange([form3.Table1.FieldByName('OrderNo').AsFloat],
        [form3.Table1.FieldByName('OrderNo').AsFloat]);
    Form2.QuickRepl.Preview;
    Form2.Table1.CancelRange;
end;
```

Listado 5: vista previa de una sola orden

Como vemos, la impresión se simplifica mucho con el uso del Quick Report. Hay mucho más todavía por descubrir: la jerarquía de componentes de Quick Report es bastante rica y nos ofrece la oportunidad de crear nuestros propios componentes imprimibles. Un ejemplo de extensión de las capacidades básicas es la serie de artículos de Keith Wood [Extending Quick Report-Delphi Informant Agosto/Septiembre 1997], donde se crea una grilla del estilo del DBGrid que se puede imprimir y además se modifica la ventana de vista previa agregando otras capacidades como diferentes grados de acercamiento (zoom).

Problemas

Asimismo, según podemos ver en los grupos de discusión especializados en Internet, el producto aún adolece de ciertas falencias -por ejemplo, varios usuarios han reportado una pérdida irremediable de recursos del sistema al hacer vistas previas en tiempo de diseño... con ciertas impresoras, en ciertos equipos, en determinadas situaciones. Es de esperar que pronto quSoft libere una corrección a ese y otros problemas que se han presentado.

Hay otro problema grave que se presenta en Delphi 4 (Quick Report 3) al usar bases de datos de Access 97: si la consulta que entrega los datos a QR involucra más de una tabla, la aplicación se cae... generalmente arrastrando en la caída a todo el sistema! Este problema ha sido reconocido por QuSoft, pero no ha sido solucionado. Según he visto en grupos de discusión en Internet, una posible solución viene de la mano de otro producto -que hay que pagar, por supuesto- que agrega capacidades de editar los reportes en tiempo de ejecución, incluso para el usuario final. Este producto se instala sobre QR y parece solucionar los problemas. Se llama QRDesign y se puede bajar una versión de prueba del sitio www.thsd.de. Aunque no tengamos el problema, vale la pena echar una ojeada al QRDesign; la capacidad de dar al usuario un generador de reportes es muchas veces importante.

A Mayo/1999, la última versión de Quick Report es la 3.04; la actualización se puede obtener de Internet.

Algunas Observaciones

- C Cuando se muestra en ejecución una ventana que contiene un reporte, se ve lo mismo que en tiempo de diseño.
- C Generalmente, se utilizan componentes de acceso a los datos independientes para los reportes, dado que el reporte debe navegar las tablas y si el usuario está trabajando con el mismo componente, se producirá un post automático y el cursor quedará en el último registro impreso.
- C Cuando se produce un reporte, se crea un componente llamado QRPrinter. Es un componente no visual que da acceso a la impresora directamente, por ejemplo para enviar un fin de página. Es equivalente al objeto *Printer*.
- C La grilla del QuickRep es exacta y depende de la propiedad *Units* si está en Pulgadas o MM.
- C La propiedad *Fonts* define la fuente por defecto para el reporte; NO SE LISTAN LAS FUENTES ESPECÍFICAS DEL DISPOSITIVO, como las fuentes comunes de las impresoras de agujas; hay que ponerlas "a mano" asignando el valor a la propiedad **Name** de **Fonts**.
- C Una banda se puede estirar para que entre el contenido de alguno de los componentes imprimibles (por ejemplo un QRDBMemo) siempre que la propiedad **CanExpand** de la banda esté en *True* y la propiedad **AutoStretch** (AutoAjuste) del componente esté en *True* también.
- C La posición y el tamaño de un componente imprimible se pueden especificar con precisión a través de la propiedad *Size*. Las unidades son las de la propiedad *Units* del reporte.

Con esto terminamos el tema de Impresión desde Delphi. Sólo hemos tocado la superficie, dado que es posible crear reportes mucho más complejos. A continuación ofrecemos una referencia de artículos relacionados para aquellos que deseen ahondar un poco en el tema.

QuickReport: an introduction to Delphi 2's alternative report generator. Cary Jensen, Delphi Informant Vol 2, Nro 8 (Agosto 1996). Una introducción simple como la que hemos dado aquí. Trata del Quick Report versión 1 que venía con Delphi 2; algunas cosas han cambiado en la versión 2.

QuickReport 2.0: Part I. Cary Jensen, Delphi Informant Vol 3, Nro 8 (Agosto 1997). Primera parte de la construcción de un programa de ejemplo de varios tipos de reportes. Utiliza *Child Bands* en uno de los reportes.

Creating Mailing Labels (Quick Report 2, Part II). Cary Jensen, Delphi Informant Vol 3, Nro 9 (Septiembre 1997). Técnicas avanzadas: reportes Master/Detail, etiquetas postales, ventanas de vista previa definidas por nosotros, y generación de reportes en tiempo de ejecución.

Extending QuickReport: Part I. Keith Wood, Delphi Informant Vol 3 Nro 8 (Agosto 1997). Técnicas avanzadas: creación de una vista previa propia con más posibilidades, y creación de un componente nuevo que muestra una grilla de base de datos (como el DBGrid) en el reporte.

Extending QuickReport: Part II. Keith Wood, Delphi Informant Vol 3 Nro 9 (Septiembre 1997). Agregados al componente de grilla creado en el artículo anterior.

Building a Master/Detail Report. En el sitio de QuSoft (www.qusoft.com), es un artículo con los fuentes correspondientes que explica cómo crear paso a paso un reporte master/detail. Va un poco más allá de lo presentado aquí, utiliza tres tablas y una consulta para presentar más datos.

QRDesign y QRPowerPack. En el sitio de Timo Hartmann Software Development (www.thsd.de) podemos encontrar estos dos productos; el primero es un editor de reportes para el usuario final (en tiempo de ejecución). Es pago. El PowerPack es un conjunto de componentes imprimibles *gratis*, que se instalan sobre QR y dan posibilidades avanzadas como reportes directos de grillas, etiquetas con ángulo, etc.

En el sitio WEB de QuSoft se pueden encontrar varios ejemplos, componentes shareware y freeware y algunos artículos sobre temas específicos, además de parches para las versiones actuales del producto. Conviene darse una vueltita de vez en cuando.

Bases de datos

Paradox

El acceso a bases de datos de Delphi es uno de sus puntos fuertes; está implementado en torno a la BDE, que es el mismo motor de acceso a datos usado por Paradox y dBase for Windows. Además, Delphi trae drivers nativos para acceder a datos en tablas de Access y de FoxPro, así como un servidor SQL (Interbase), para luego hacer la migración a Cliente/Servidor sin cambios en el programa.

Instalando aparte el SQL Links tendremos acceso transparente a bases de datos ORACLE, SYBASE, DB2, SQL SERVER e INFORMIX así como a cualquier origen de datos ODBC.

Algunos conceptos y Herramientas

Antes de comenzar a trabajar con bases de datos, conviene tener en claro una serie de conceptos claves:

Base de Datos (Database): es un conjunto de tablas. En Delphi una base de datos es un **directorio** donde estarán las tablas **-archivos-** y los índices.

Tabla (Table): es el lugar donde están realmente los datos. Se divide en *registros*, los cuales están formados por *campos*. Podemos hacer una analogía con una tabla, según la cual los registros serían las *filas* y los campos las *columnas*.

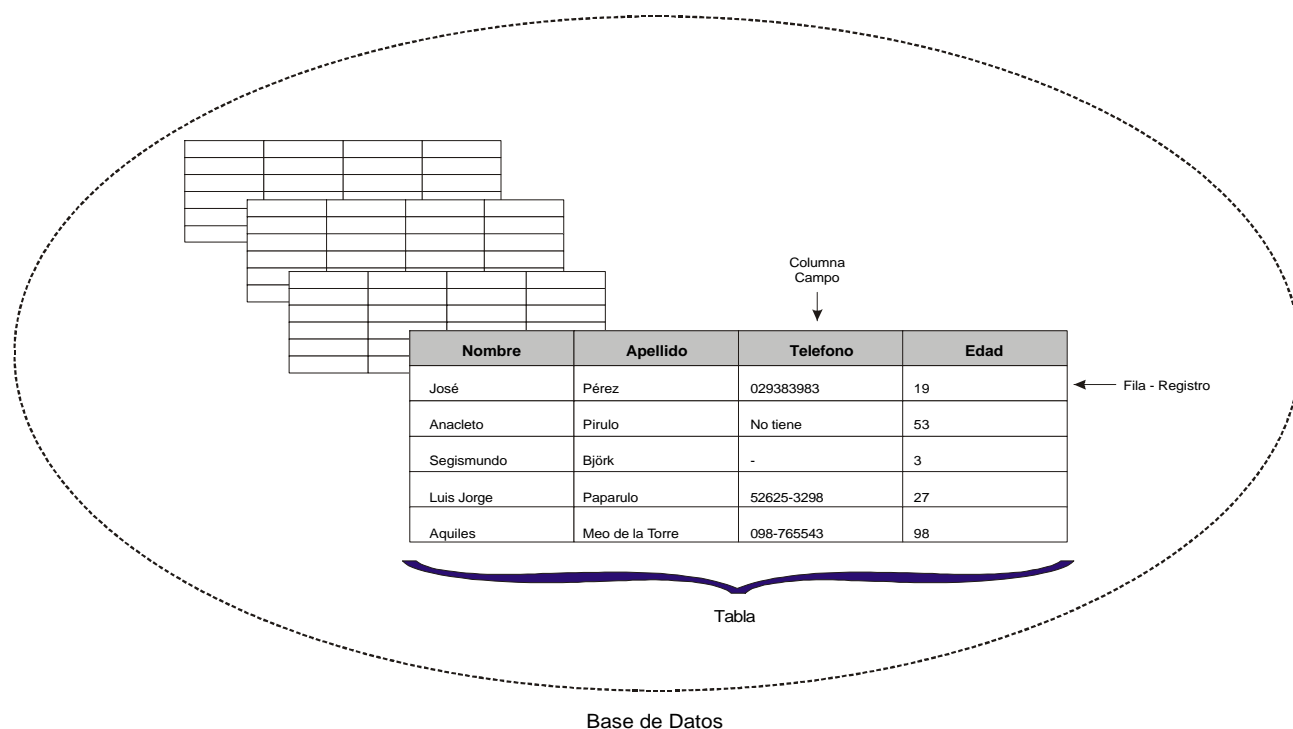


Figura 1: Base de datos, tablas, registros, campos

Índice (Index): es una estructura separada de la tabla que guarda ciertos campos de todos los registros, para acelerar las búsquedas y el ordenamiento. En Paradox y dBase se utilizan archivos adicionales para los índices.

Structured Query Language (SQL): lenguaje estructurado creado especialmente para realizar consultas a bases de datos.

Database Desktop (DBD): es una versión reducida del Paradox for Windows; nos permite ver, crear, modificar, borrar tablas. Se encuentra en el menú Tools|Database Desktop. También podemos desde aquí transformar una tabla de un formato a otro o hacer consultas SQL.

SQL Explorer: es una herramienta muy potente que nos permite navegar por las definiciones y datos de las distintas bases de datos conectadas a la BDE, no importa su formato. Lo utilizaremos para crear alias, para ver datos directamente de las tablas y para ejecutar consultas SQL inmediatas.

Database Desktop

Es una herramienta para manejo de tablas en los distintos formatos manejados por la BDE. Posee una interface al estilo Paradox -se lo puede ver como un “hermano pequeño” del Paradox for Windows. Usaremos esta herramienta para crear tablas, cambiar su estructura, modificar sus datos, etc. Se puede encontrar en el menú Tools de Delphi.

Crear una tabla usando el Database Desktop

Es la forma preferida de crear las tablas, que generalmente se definen en tiempo de diseño (es parte del diseño del programa la especificación de los campos de las tablas y su contenido). Sólo hay que seleccionar la opción de menú File|New|Table. Luego seleccionamos el formato deseado de tabla y aparecerá el editor de estructura (Field Roster) en el que definimos los campos.

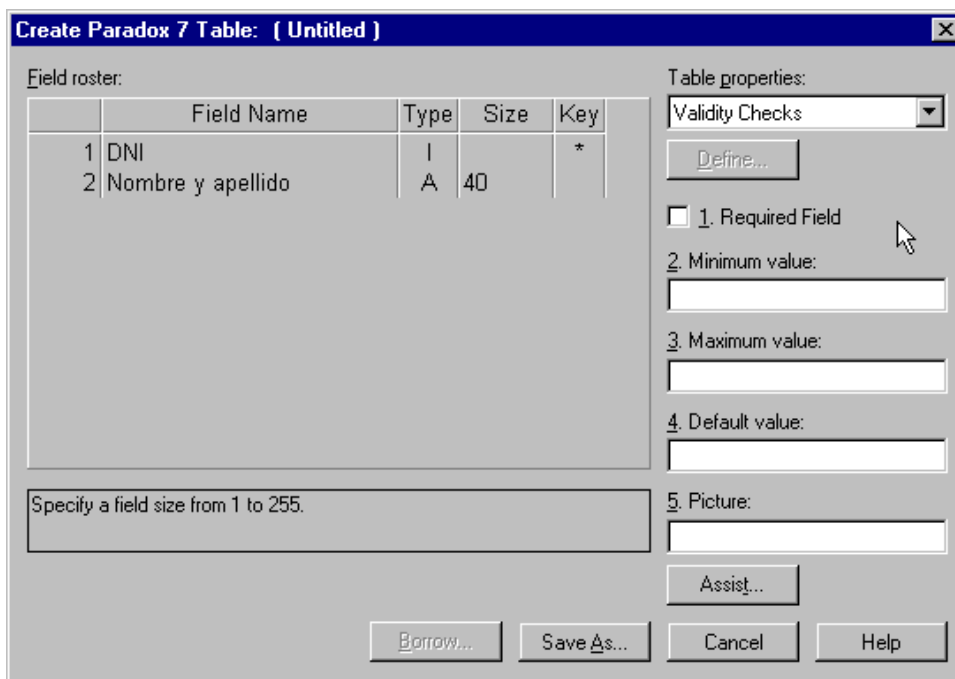


Figura 2: el Field Roster, donde se define la estructura de la tabla

Para definir completamente la tabla debemos indicar aquí los *nombres* de los campos, el *tipo de datos* que contendrá cada uno, la *longitud máxima* si corresponde (algunos tipos de datos tienen longitud fija, como los tipos numéricos de Paradox) y los campos que compondrán el *Índice principal* (o *Clave Primaria*).

El índice principal es una estructura aparte de la tabla¹, donde se mantienen únicamente los campos marcados en la definición. Se utiliza para optimizar el mantenimiento de los datos en el archivo principal.

No es *necesario* crear una clave principal, pero si *muy recomendable*. El subsistema de manejo de los datos funciona de manera mucho más eficiente si dispone de un índice, y hay algunas operaciones que no se pueden realizar si no se crea esta estructura (como el enlace de dos tablas en una relación, o la búsqueda eficiente de un registro).

La estructura del índice primario es una decisión muy importante, ya que define no sólo el orden de los registros sino también el criterio para diferenciar en forma unívoca cada registro. En otras palabras, **no puede haber dos registros con valores iguales en los campos del índice primario**. Por ejemplo: en una tabla que contenga datos de personas, no sería buena la elección del campo *nombre* para la clave primaria, ya que pueden existir dos personas con el mismo nombre. En este caso un buen candidato sería el campo DNI, ya que no puede haber dos personas con el mismo número. Postergamos la discusión de los índices para más tarde.

Los **tipos de datos** permitidos varían según el formato de la tabla; así por ejemplo tenemos para las tablas de Paradox los campos tipo *AutoIncrement* o *Time* que en el formato de dBase no existen (aunque se pueden emular con otros).

Nombres de campos en formato Paradox

? Pueden tener hasta 25 caracteres de largo.

? Deben ser únicos en la tabla.

? Pueden contener letras, números, y cualquier carácter imprimible excepto los siguientes: “ [] { } () # o la combinación “->”

? Se pueden usar espacios, salvo al principio.

¹ En Paradox o dBase, los índices se guardan en archivos separados de la tabla principal.

En el Apéndice II se ve una lista de los tipos de datos soportados por Paradox.

Una vez que definimos la estructura en el Field Roster, grabamos la tabla recién creada y podemos empezar a llenarla, simplemente abriéndola en el mismo Database Desktop y escribiendo los valores en los campos.

Ejercicio

Crear una tabla en formato Paradox con la siguiente definición:

Nombre	Tipo	Longitud	Clave primaria
DNI	I		*
Nombre y Apellido	A	40	
Dirección	A	40	
Teléfono	A	15	
Fecha de Nacimiento	D		
Comentarios	M	30	

Guardarla como "NOMBRES.DB". Abrir la tabla en el Database Desktop y agregar algunos registros.

NOTA: para empezar a editar una tabla hay que presionar F9 - para un campo particular, F2

?

Archivos creados: hasta ahora tenemos tres archivos creados, todos con el mismo nombre pero diferentes extensiones. La tabla en sí, con los datos, estará en el archivo NOMBRES.DB. Dado que hay un campo de tipo Memo (el campo *comentarios*), se crea también un archivo NOMBRES.MB. En el archivo principal se guardarán los primeros 30 caracteres de este campo²; si se pasa de esta cantidad, el resto irá a parar al archivo .MB.

El tercer archivo tiene extensión .PX: es el archivo del índice primario o principal.

La interface tipo "planilla de cálculo" del Database Desktop es muy conveniente para navegar los datos, comprobar cambios, editar registros, etc.

NOTA: para borrar un registro hay que presionar Ctrl+Del en la ventana de hoja de datos de la tabla. Se elimina el renglón sobre el que se encuentra el cursor.

Mantenimiento de la Base de Datos

Damos a continuación una referencia de algunas de las operaciones comunes que se pueden realizar con las tablas en el Database Desktop (salvo indicación expresa, los comandos se encuentran en el menú Tools|Utilities, y no es necesario abrir la tabla para ejecutar la acción sobre ella):

? Vaciar completamente una tabla: comando **Empty** del menú Utilities. Se nos pide confirmación, ya que la acción elimina todos los registros de la tabla y no se puede deshacer. No es necesario tener abierta la tabla para poder vaciarla.

² En realidad, se guardan 10 bytes más que lo indicado; vea el apéndice II

- ? Agregar o actualizar registros de una tabla a otra: comando **Add**. Si se elige actualizar, se cambiarán los registros de la tabla destino cuya clave coincida con la de los registros de la tabla fuente.
- ? Copiar una tabla en otra: comando **Copy**. Se duplican también los índices y otras características de la tabla origen, es decir, se crea una réplica idéntica. Se cambian tablas de formato entre dBase y Paradox, simplemente especificando la extensión de la copia. En la ayuda se especifican los cambios que sufren los tipos de datos al pasar de un formato a otro.
- ? Borrar una tabla y todos sus archivos asociados: comando **Delete**. Este comando es preferible al borrado manual a través del explorador o Administrador de Archivos, ya que elimina automáticamente *todos* los archivos asociados con la tabla: índices, reglas de validación, etc.
- ? Renombrar una tabla: comando **Rename**. Las mismas consideraciones que para el comando Delete.
- ? Información sobre la estructura de la tabla: comando **Info structure**. Presenta la ventana del Field Roster con toda la información de definición de los campos, sin posibilidad de modificarla. Esta información de estructura se puede guardar en una tabla.
- ? Cambiar la estructura de la tabla: comando **Restructure**. Presenta la ventana del Field Roster y permite cambiar cualquier atributo de la tabla.

NOTA: para poder reestructurar una tabla es necesario que la misma no esté abierta en otro programa, por ejemplo en Delphi. De ser así, se nos presenta un mensaje indicando que la tabla está en uso y no se puede abrir en forma exclusiva. Debemos cambiar a Delphi (Alt+Tab) y cerrar la tabla antes de reintentar.

- ? Ordenar una tabla: comando **Sort**. Si la tabla origen no tiene índice principal, los datos quedan en la misma tabla; caso contrario, se crea una nueva tabla *sin índice principal* con los registros colocados en el orden pedido.
- ? Eliminar los registros de una tabla si ya existen en otra: comando **Subtract**. Se comparan los registros de la tabla origen (subtract records in) con los de la tabla destino (subtract records from) y si los valores de los campos del índice principal coinciden se eliminan los registros de la tabla destino.

En definitiva, el Database Desktop es una muy buena herramienta para trabajar con las tablas, y es la preferida a la hora de *crearlas* o reestructurarlas, tareas que son bastante trabajosas de llevar a cabo por programa.

E **ejercicio**
Renombrar la tabla NOMBRES.DB a AGENDA.DB. _____ ?

E **ejercicio**
Agregar un campo EMAIL de tipo Alfanumérico, 50 caracteres de longitud. _____ ?

E **ejercicio**
Copiar la tabla AGENDA.DB a formato Dbase (.DBF). Observar las diferencias en la estructura. Iniciar el Explorador de Windows y reconocer los archivos de memo (.DBT) e índices (.MDX) de Dbase. _____ ?

Resumiendo: tenemos algo llamado Base de Datos (*Database*) que viene a ser como un gran recipiente donde se almacenan otras cosas: las tablas. Éstas tienen por fin los datos, organizados en registros compuestos de campos. Estas estructuras se pueden crear, modificar, llenar, borrar, etc. desde el Database Desktop. Hasta aquí todo bien.

Pero no todo es tan sencillo: hay otras consideraciones a tener en cuenta cuando se trabaja con bases de datos.

Índices

Comencemos por los índices. Un *índice* es una estructura extra a una tabla, que crea el sistema de administración de Bases de Datos para facilitar su tarea con esa tabla. No es obligatorio, **pero si muy recomendable**. O sea: *siempre* que podamos, crearemos índices para nuestras tablas.

Un índice es simplemente una lista ordenada de los valores de los campos que lo componen. En particular, Paradox crea automáticamente un índice para los campos de la *clave primaria* de la tabla³ -llamado *índice primario o principal*- pero se pueden crear fácilmente otros índices -llamados en este caso *secundarios*.

El efecto visible de los índices es el ordenamiento de los registros: normalmente, los programas Administradores de Bases de Datos ordenan los registros basándose en los valores de los campos que se incluyen en el índice. Veamos entonces cómo se crea un índice y de qué se compone.

Los índices se componen de un subconjunto de los campos de la tabla a la que pertenecen. Se definen como parte de la estructura de la tabla: en el Field Roster del Database Desktop, por ejemplo.

La **Clave primaria** (sólo puede haber una en cada tabla, posiblemente compuesta de varios campos) sirve para identificar unívocamente a cada registro; no puede haber dos registros con los mismos datos en los campos de la clave primaria. Se define en el Field Roster marcando la columna **Key** de aquellos campos que la conformen -para marcar esta columna presione <Espacio> o haga doble click con el ratón. Los campos marcados deberán ser los primeros de la tabla y todos contiguos.

En una tabla simple de datos de personas (Agenda), si utilizamos el campo Nombre solo como clave primaria, no podríamos cargar dos personas que se llamen igual. Si agregamos el apellido mejora un poco la situación, pero aún así hay muchos casos (por ejemplo padre e hijo) en que dos personas tienen el mismo nombre y el mismo apellido, y entonces no podrían coexistir en la tabla. Podemos entonces agregar el campo Dirección al índice; y también tendremos el mismo problema si el padre e hijo que se llaman igual viven en la misma casa. Como pueden ver, la definición de una clave es un tema delicado. En este ejemplo, hay un dato que es el candidato ideal para conformar el índice primario: el DNI. No puede haber dos personas con el mismo número, por lo que el requisito de unicidad se cumple perfectamente.

Una vez marcados los campos de la clave primaria, al grabar la estructura de la tabla se creará un archivo para contener un índice a los campos de la clave. Este archivo tiene el mismo nombre que la tabla, con extensión .PX.

NOTA: físicamente, los registros se almacenan en el orden dado por la clave primaria; por lo tanto, el tiempo de acceso es óptimo cuando se utiliza este índice.

³ Este índice es un poco especial internamente; vea el apéndice III

Podemos ya notar el efecto más visible: si ingresamos datos a la tabla, se ordenarán por DNI.

Para conseguir que se ordenen por ejemplo por nombre, debemos definir otros índices: los *índices secundarios*.

Los **índices secundarios** también se componen de campos, pero éstos pueden estar en cualquier posición de la tabla y pueden exigir valores únicos o no. Para definirlos debemos cambiar la estructura de la tabla en el Field Roster, seleccionando “Secondary Indexes” (Índices secundarios) del Combo “Table Properties” (Propiedades de la tabla) como se ve en la fig.3.

Se nos presentará entonces un cuadro de diálogo donde podremos elegir los campos que conformarán nuestro nuevo índice secundario, moviéndolos desde la lista de la izquierda a la de la derecha utilizando los botones con flechas. En la fig. 4 se ve un ejemplo: en la tabla Nombres creada anteriormente definimos un índice secundario sobre el campo “Nombre y Apellido”.

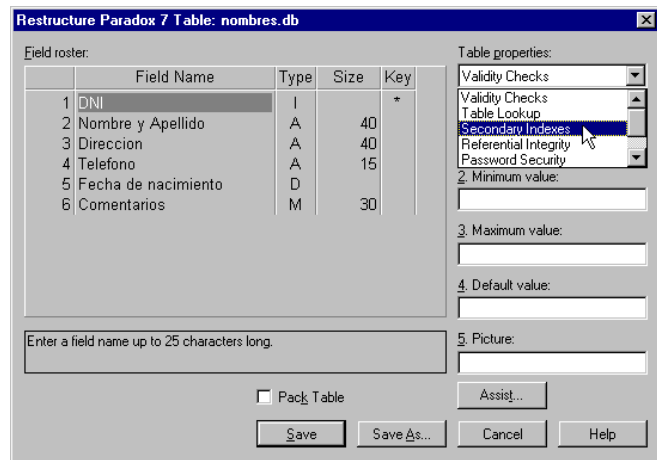


Figura 3: acceso a la definición de los índices secundarios

Las opciones de que disponemos a la hora de crear estos índices son las siguientes:

- ? **Unique:** indica si los valores en los campos del índice deben ser únicos, es decir, si se permitirá el ingreso de dos registros con los mismos valores para todos los campos del índice. Si marcamos esta opción y luego intentamos ingresar dos registros con los mismos valores en los campos del índice, el gestor de base de datos no lo permitirá.
- ? **Maintained:** indica que el índice sea automantenido o no. *Delphi sólo acepta índices automantenidos*, así que deje esta opción marcada. Los índices automantenidos se abren automáticamente al mismo tiempo que la tabla a la que pertenecen y se mantienen actualizados constantemente. Esto puede traer una ligera sobrecarga al sistema si definimos muchos índices, pero es más seguro.
- ? **Case sensitive:** si está marcado, la ordenación tomará en cuenta si el campo está en mayúsculas o minúsculas.
- ? **Descending:** si está marcado, la ordenación será en sentido inverso, de mayor a menor.

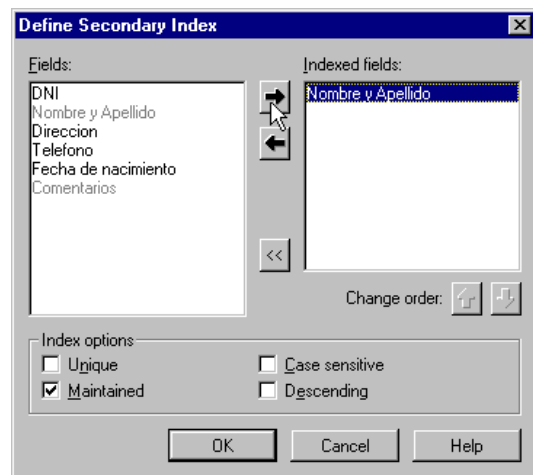


Figura 4: definición de un índice secundario por el campo “Nombre y Apellido”

Con las opciones marcadas como aparecen por defecto (igual que en la figura 4), los índices creados permitirán la existencia de múltiples registros con los mismos valores (que aparecerán uno después del otro), no se tendrá en cuenta si la escritura está en mayúsculas o minúsculas (será lo mismo “Primero” que “primero”), el índice se mantendrá automáticamente y el orden será el normal, de menor a mayor.

Aceptemos el índice secundario anterior con el botón OK; como nombre escribamos **iNombre**. Veremos un ejemplo de la utilización desde Delphi luego.

Los índices secundarios se almacenan en archivos separados de la tabla a la que pertenecen, con el mismo nombre pero extensión .X01, .Y01, X02, Y02, en general .X?? y .Y??.

El motor de Bases de Datos utiliza un sólo índice por vez: se denomina el **índice activo**⁴.

Los índices secundarios almacenan todo el contenido de los campos que lo componen, más los campos del índice primario para cada registro, y un número de bloque para acceder al mismo rápidamente; por lo tanto, son bastante más grandes que el índice primario.

Utilizaremos profusamente los índices secundarios cuando accedamos a las tablas desde Delphi, por lo que pospondremos su aplicación práctica hasta ese momento.

Reglas de validación

Las reglas de validación (*validity checks*) son restricciones a los datos que pueden ser ingresados en los campos. Las reglas posibles son las siguientes:

- ? **Campo requerido** (Required Field): indica que el campo no puede quedar vacío.
- ? **Valor mínimo** (Minimum Value): el valor del campo no puede ser menor que el especificado en este lugar.
- ? **Valor máximo** (Maximum Value): el valor del campo no puede ser mayor que el valor especificado en este lugar.
- ? **Valor por defecto** (Default Value): el campo toma este valor cuando se crea un nuevo registro.
- ? **Máscara** (Picture): cadena de caracteres que especifica el formato del campo. Se cuenta con un asistente que permite probar las máscaras cuando las generamos, y tiene unos cuantos ejemplos comenes listos para usar. El “lenguaje” de caracteres que utiliza Paradox para las máscaras es muy completo y complejo, por lo que se dejará al lector la encomiable tarea de aprenderlo por su cuenta. Damos a continuación una tabla con algunos ejemplos:

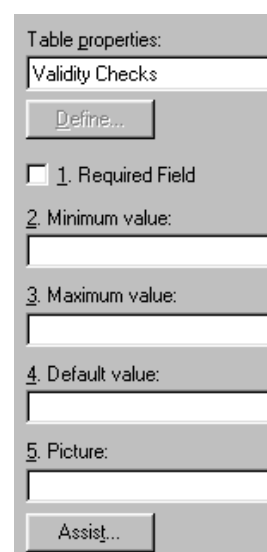


Figura 5: Reglas de validación

NOTA: las reglas de validación se almacenan en archivos separados de la tabla, con el mismo nombre que ésta pero extensión .VAL.

Caracter	Significado	Ejemplo
@	Cualquier caracter	@@: “4r”
#	un número (0 a 9)	###: “798”
?	Una letra (mayúscula o minúscula)	?: “aA”
&	una letra (convierte a mayúscula)	&&: “aA” es convertido a “AA”

⁴ No obstante, cuando se abre una tabla se abren *todos* los archivos de índice, y se actualizan al mismo tiempo que la tabla.

Caracter	Significado	Ejemplo
~	una letra (convierte a minúscula)	~~: “aA” es convertido a “aa”
!	Cualquier caracter; las letras se convierten a mayúsculas	!!!: “y7S” es convertido a “Y7S”
[]	Opcional	[#]#: uno o dos números
{ }	lista de entradas posibles	{“Uno”, “Dos”}: sólo se puede escribir “Uno” o “Dos”
*n	n repeticiones de un grupo o caracter (lo que sigue)	*3#: tres números
*	cualquier cantidad de repeticiones de lo que sigue	*#: cualquier cantidad de números
;	indica que el caracter que sigue se trate literalmente	;;#: el símbolo “#”
Otro caracter	se escribe literalmente	[#]#[#]/#####: máscara para fecha, el día y el mes se pueden escribir con un solo dígito y se agregan las barras automáticamente [#]#[#]#: máscara para hora: minutos

IMPORTANTE: cuando creamos los componentes de campo en Delphi para una tabla que contenga reglas de validación, las mismas *no son importadas*. Para lograr que se importen al menos las validaciones de valor mínimo y máximo, debemos primero ingresar la base de datos en el Diccionario de Datos. Veremos este tema al tratar el acceso a Bases de Datos desde Delphi.

Integridad Referencial

Cuando relacionamos dos tablas, una contiene los datos principales -por ejemplo, el nro. de factura y la fecha- mientras la otra contiene detalles que amplían la información. Esta relación puede ser del tipo Uno-a-Uno (por cada registro de la tabla principal hay uno solo de la tabla de detalle que cumple con la relación) o Uno-a-Muchos (por cada registro de la tabla principal puede haber más de un registro de detalle)⁵.

Al tener una relación entre dos tablas se plantean problemas de consistencia de datos: ¿qué sucede por ejemplo si se borra un registro de la tabla principal que tiene registros relacionados en la tabla de detalle? ¿O si modificamos el valor de los campos de la relación en un registro de la tabla principal que tenga registros relacionados? ¿Hay que cambiar correspondientemente los valores de enlace de los registros de detalle para mantener la relación? ¿O tal vez prohibir directamente la modificación o borrado de datos de la tabla

⁵ Hay una tercera forma teórica de relacionar tablas, llamada Muchos-a-Muchos. No obstante, en la implementación actual de los gestores de bases de datos relacionales más utilizados esta forma debe realizarse utilizando una tabla auxiliar y relaciones Uno-a-Muchos.

principal cuando tengan registros relacionados?

Para poder estudiar este problema y su solución, necesitaremos un par de definiciones:

- ? Clave externa (Foreign Key): cuando uno o más campos de una tabla (detalle) referencian a uno o más campos de la clave *primaria* de otra tabla (principal) se dice que los primeros forman una *clave externa*.
- ? Integridad Referencial (Referential Integrity): es una regla que dice que para cada registro de una tabla de detalle que esté en relación con otra principal *debe existir un valor de la clave primaria (en la tabla principal) para cada valor de la clave externa (en la tabla secundaria)*. Esto es, no pueden quedar registros “huérfanos” en el detalle, que no se correspondan con ningún registro de la tabla principal.

La Integridad Referencial asegura la consistencia de la relación entre las dos tablas.

Ahora bien, el problema se presenta generalmente cuando tratamos con relaciones uno-a-muchos, al modificar o borrar registros de la tabla principal que tengan registros relacionados en la tabla detalle. Hay dos formas de solucionar este problema:

1. Impedir los cambios en la tabla principal si hay registros relacionados en la de detalle
2. Propagar los cambios a todos los registros relacionados, de manera que el enlace se mantenga

Existe una tercera posibilidad en algunos gestores de Bases de Datos (no en Paradox): los campos de la clave externa toman un valor nulo cuando se elimina o modifica el registro principal al que referenciaban.

Paradox nos permite especificar las dos formas cuando se trata de una modificación, pero sólo la prohibición cuando hablamos de borrado de registros principales.

Para especificar la Integridad Referencial entre dos tablas en el Database Desktop, debemos seguir los siguientes pasos:

1. Crear la tabla principal, con una clave primaria. Por ejemplo, crearemos una tabla de facturas con la siguiente estructura (FACTURAS.DB):

Campo	Tipo	Tamaño	Clave	Comentario
NroFact	I		*	Clave Primaria
FechaFact	D			
Cliente	A	30		
Tipo	A	1		
FormaPago	A	10		

2. Crear la tabla de detalle o secundaria con campos que coincidan en tipo de datos con los de la clave primaria de la tabla principal. Éstos formarán la clave externa. En nuestro ejemplo, crearemos una tabla de detalle de facturas con la siguiente estructura (DETALLES.DB):

Campo	Tipo	Tamaño	Clave	Comentario
IDItem	+		*	Clave primaria
Cantidad	I			
Descripcion	A	40		
PrecioUnit	\$			

Factura	I			Clave externa a FACTURAS.DB
---------	---	--	--	--------------------------------

3. Crear la regla de Integridad Referencial entre las dos tablas.

Para crear la Integridad Referencial, mientras definimos la estructura **de la tabla detalle** (en el Field Roster) seleccionamos la opción “Referential Integrity” del combo “Table Properties”. A continuación, presionamos el botón “Define...” para definir la regla.

Se nos presenta un cuadro de diálogo como el de la figura 6. Aquí seleccionamos el o los campos de la clave externa deseada (en nuestro ejemplo, el campo “Factura”) y la tabla principal con la que deseamos relacionar (en nuestro ejemplo, FACTURAS.DB) de la cual se seleccionan automáticamente los campos de la clave primaria.

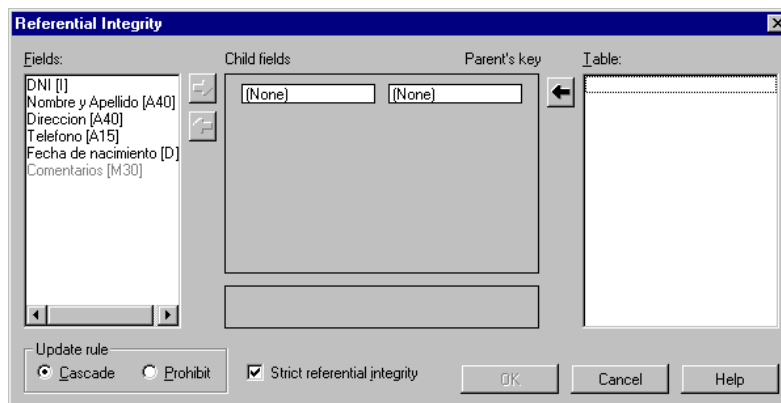


Figura 6: definición de Integridad Referencial

Elegimos luego el tipo de restricción a aplicar para las modificaciones -recordemos que el borrado de registros

se prohíbe, sin opción- entre Propagación en cascada (**Cascade**) o Prohibición de cambios (**Prohibit**). El cuadro de opción “Strict Referential Integrity” indica si se desea que las versiones anteriores de Paradox que no permitían Integridad Referencial puedan o no modificar las tablas. Si está marcado, no se podrán abrir las tablas en una versión de Paradox que no tome en cuenta la Integridad Referencial. Es la opción por defecto y la más recomendada, ya que si no sería posible modificar los datos de una sola tabla y perder la consistencia de la relación.

Una vez que hemos especificado todos los datos necesarios, damos un nombre a la regla (es posible crear más de una regla, por ejemplo para mantener Integridad Referencial con más de una tabla). A partir de ahora, no será posible agregar registros en la tabla de detalle cuyo número de factura no exista en la tabla de facturas, así como no podremos borrar un registro de facturas que tenga detalles. El comportamiento ante una modificación del campo NroFact de una factura que tenga detalle dependerá del tipo de restricción que aplicamos.

Ejercicio

Probaremos la regla de Integridad Referencial que acabamos de crear. Para ello, abriremos las dos tablas en la misma pantalla (Tile Top and Bottom) así podemos ver el efecto de las modificaciones. A continuación, crearemos por lo menos dos registros de facturas.

? ¿Es posible crear primero los registros de detalle para estas facturas?

Luego creamos por lo menos dos registros de detalle para cada factura (el valor del campo “Factura” debe coincidir con el “NroFact” de una factura existente).

¿Qué sucederá si

? modificamos el número de factura de un registro de “Facturas”?

? modificamos el nro. de factura de un registro de “Detalles”?

? borramos un registro de “Detalles”?

? borramos un registro de “Facturas”?

? agregamos un registro de “Facturas” que no tenga detalle?

? agregamos un registro de “Detalles” que no tenga factura?

?

La información de Integridad Referencial se guarda en los archivos .VAL de las dos tablas. Se crea automáticamente un índice secundario en los campos de la clave externa (en realidad, el nombre de la regla de Integridad Referencial que especificamos es el nombre de este índice).

BDE

El Motor de Bases de Datos de Borland (Borland Database Engine, a partir de ahora BDE) está implementado en una serie de librerías de enlace dinámico (DLL) que se instalan con Delphi, que también lo usa internamente. En las últimas versiones se la conoce como BDE, pero hace ya algunos años que está incorporada a varios productos (Quattro Pro, Autocad, Word Perfect, etc); anteriormente se la conoció como Open Database Application Programming Interface (ODAPI) o Independent Database Application Programming Interface (IDAPI).

La BDE es un intento de sentar un estándar para el acceso a bases de datos (en el que participaron Borland, Novell, IBM y WordPerfect), donde cada compañía puede crear sus propios drivers mientras cumpla con las especificaciones. El intento no prosperó comercialmente porque Microsoft lanzó su propio producto con esa meta: ODBC (Open DataBase Connectivity).

Como regla general, en una comparación entre la BDE y ODBC se hallará que los drivers de la primera son más veloces y con más funcionalidad que los de ODBC.

El lector atento podrá pensar en este momento (el resto de los lectores también pensará, generalmente, en este momento; pero también generalmente el tema no tendrá nada que ver con la BDE), y transcribo literalmente: “¿Para qué @#@\$#% gastar recursos y tiempo en ese paso intermedio entre el driver que accede a la tabla y mi programa?”. Bueno, hay algunas buenas razones. La más importante es la *libertad* que nos da al independizarnos del formato de la base de datos que accedemos. Podemos crear nuestro programa utilizando tablas de Paradox para el prototipo, y luego con pocos cambios migrar la base de datos a un servidor SQL tipo Oracle o SQL Server.

La meta de esta librería es tratar a todos los tipos de bases de datos para los que posee un driver de la manera más completa y eficiente posible; así por ejemplo, provee un mecanismo propio de proceso de transacciones para las tablas de Paradox y dBase, que no lo poseen. De esta manera, con el mismo procedimiento se puede iniciar una transacción en cualquiera de las bases de datos soportadas.

Los objetivos perseguidos al crear la BDE eran los siguientes:

- ? separar el lenguaje de programación del formato y acceso a los datos
- ? lograr el acceso a datos de naturaleza diversa con una interface unificada
- ? usar la misma API para formatos propios y de otras compañías
- ? unificar el acceso a bases de datos ISAM (Index Secuencial Access Method) como las utilizadas en los gestores de bases de datos comunes de PC -dBase, Paradox, etc- y las bases de datos basadas en SQL como los grandes servidores SQL
- ? el motor de acceso debe ser extensible
- ? no nivelar para abajo -es decir, no restringir los servicios del motor de acceso a las pocas actividades comunes a todos los formatos sino permitir la utilización de todas las características de cada formato

Delphi encapsula casi toda la funcionalidad de la BDE en los componentes y controles de datos, por lo que raramente se necesitará acceder a la misma directamente. De todas maneras, en la parte de técnicas y trucos veremos algunos casos en que será conveniente.

Nota

Al compilar un programa que utiliza tablas, no se incluye en el ejecutable la BDE; hay que instalarla por separado. En el paquete original de Delphi 1 se incluyen los dos discos de instalación para distribución libre de royalties, mientras que en las versiones de 32 bits se incluye en el paquete un *generador de instaladores* que permite instalar automáticamente las librerías adecuadas, además de crear los alias correctos (Install Shield Express).

Alias

Un *alias* es un nombre descriptivo que asignamos a una base de datos (nuevamente, en el trabajo normal con tablas de Paradox y Dbase identificamos Bases de Datos con Directorios). Los alias se definen en el programa de configuración de la BDE (menú Tools|BDE Config) y valen de ahí en más para todos los programas que utilicen la BDE.

Es una forma de facilitar la portabilidad: en lugar de indicar en el programa que las tablas están en un directorio determinado de un disco específico, definimos un alias. Entonces, al instalar el programa en otro equipo creamos en éste el mismo alias apuntando al directorio correcto (que puede diferir del original) y no es necesario cambiar el programa.

Para crear, ver o modificar los alias podemos utilizar varias herramientas: Database Desktop, BDEConfig o SQL Explorer. Aquí nos centraremos en esta última.

SQL Explorer

El SQL Explorer es un *explorador* de bases de datos, en el mismo sentido que el Explorador de Windows. Lo mismo que en éste, podemos ver a la izquierda un panel de tipo “árbol” y a la derecha un panel de información sobre la rama del árbol seleccionada (fig. 7).

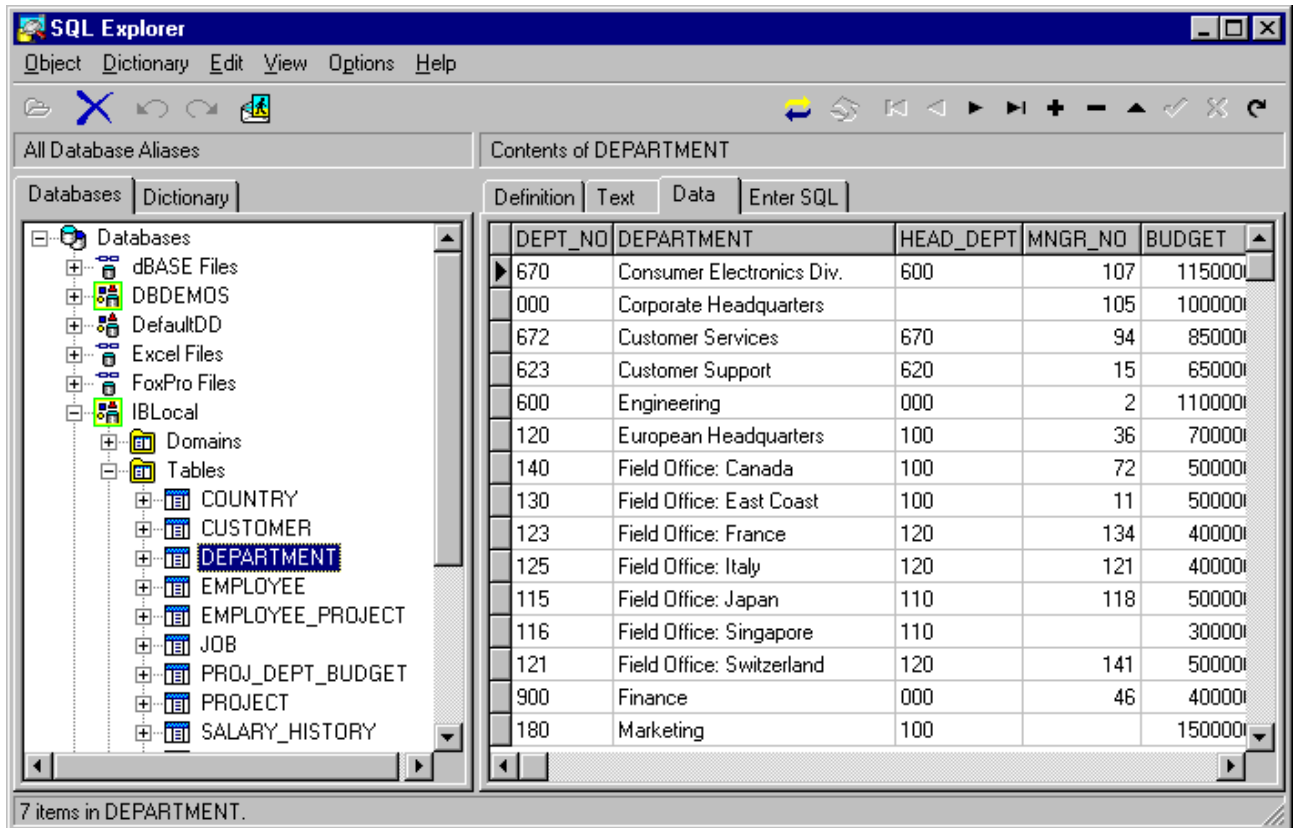


Figura 7: La pantalla principal de SQL Explorer, mostrando los datos de la tabla DEPARTMENT en la base de datos IBLOCAL

En el panel de la izquierda (el árbol) se ven los *alias* que tengamos definidos. Como cada alias representa una base de datos, hablaremos indistintamente de uno o de otro. Por ejemplo, podemos decir que “la base de datos cuyo alias es IBLOCAL tiene una tabla llamada COUNTRY” o simplemente “la base de datos IBLOCAL tiene una tabla llamada COUNTRY”. Vemos que al seleccionar la tabla DEPARTMENT podemos ver a la derecha los datos que la componen.

Vamos a crear un alias para acceder a nuestra tabla de agenda.

Nos posicionamos sobre la raíz del árbol de alias (donde dice “Databases”) y presionamos el botón derecho del ratón. Aparecerá un menú contextual (fig. 8) en el que elegimos “New...”.

NOTA: es importante que estemos sobre la raíz del árbol, ya que en otros lugares el mismo comando creará diferentes objetos.

Se nos pide ahora que ingresemos el tipo de alias (qué driver vamos a usar para acceder a los datos). Fig. 9. Para nuestro ejemplo seleccionamos Paradox (STANDARD).

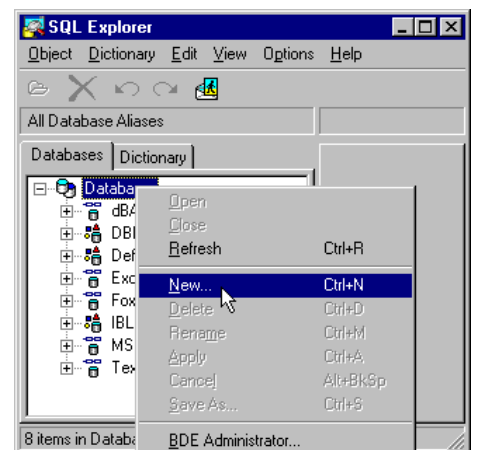


Figura 8: El primer paso para crear un alias es seleccionar “New...” del menú contextual.

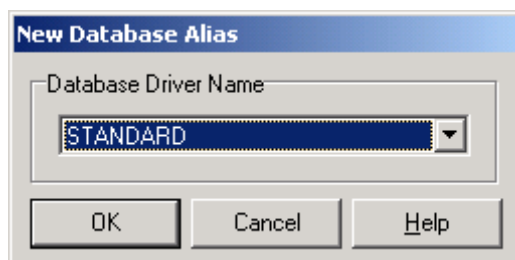


Figura 9: Driver a utilizar: STANDARD=PARADOX

Ahora se crea una nueva rama en el árbol de Bases de Datos, y podemos cambiarle el nombre. Llamémosla AGENDA (fig. 10).

La definición del alias está casi completa. Falta indicar al alias el lugar físico de la base de datos *en este equipo*.

Para el caso de las Bases de Datos de Paradox, debemos indicar un directorio; a la derecha, en la línea que dice “**Path**”.

Una vez escrito el camino que lleva a los datos, debemos confirmar los cambios presionando el botón *Apply* (Aplicar)

NOTA: *siempre* que cambiemos algo en los alias, debemos confirmar los cambios con el botón *aplicar* o las modificaciones se perderán.

Ahora si, ya estamos en condiciones de explorar nuestra base de datos con sólo hacer doble click en el alias recién creado.

Cuando hacemos doble click, estamos *conectando* con la base de datos que corresponde. Notaremos en el SQL Explorer que se hunde el botón de “Abrir”, indicando que tenemos una conexión abierta. Para cerrarla, solamente tenemos que presionar el mismo botón.

A navegar

Una vez que tenemos abierta la (conexión a la) Base de Datos, veremos algunas pestañas en el panel de la derecha que nos permitirán ver los datos de distintas maneras dependiendo del tipo de Base de Datos que trabajemos. Los siguientes ejemplos son válidos para una BD estándar, en particular trabajaremos con las tablas que vienen como ejemplo con Delphi.

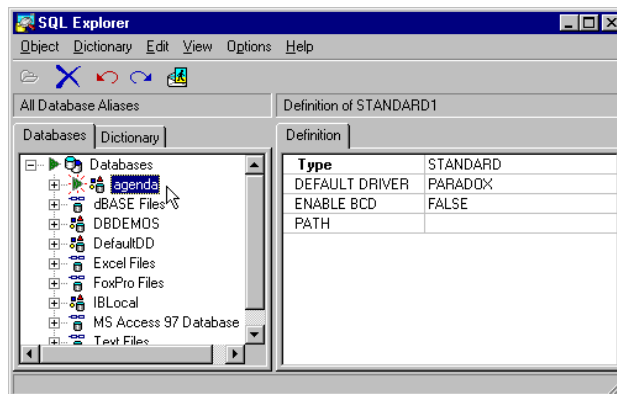


Figura 10: Poner nombre “Agenda” al nuevo alias

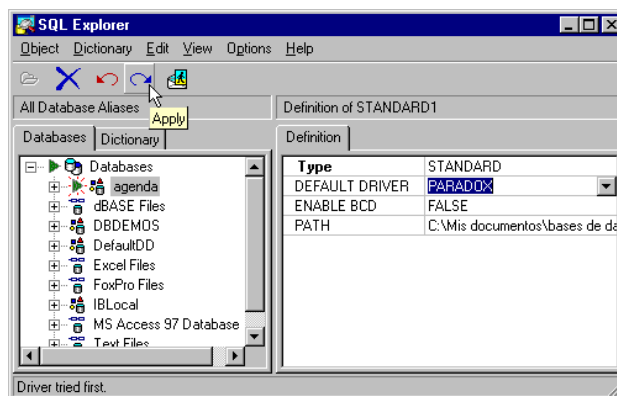


Figura 11: Aplicar (aceptar) los cambios

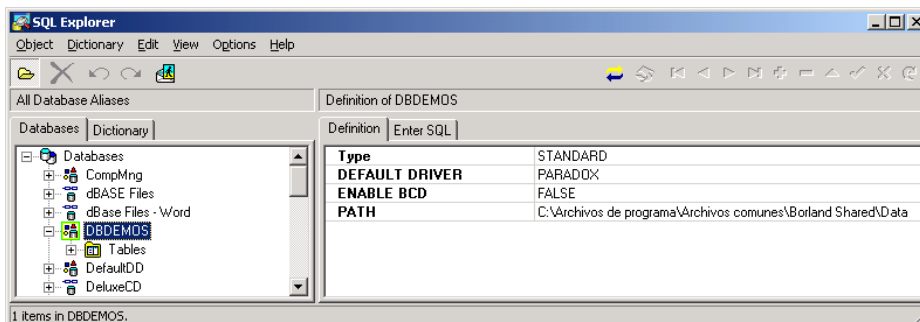


Figura 11: La Base de Datos de demostración que viene con Delphi está abierta.

Cuando tenemos seleccionado el alias en el panel de la izquierda solamente veremos dos pestañas: **Definition** (definición), en la que se muestran los parámetros de la conexión, y “Enter SQL” (Ingreso de consultas SQL) donde podemos escribir y ejecutar una consulta SQL sobre la BD.

Las cosas se ponen más interesantes cuando expandimos el nodo de las tablas:

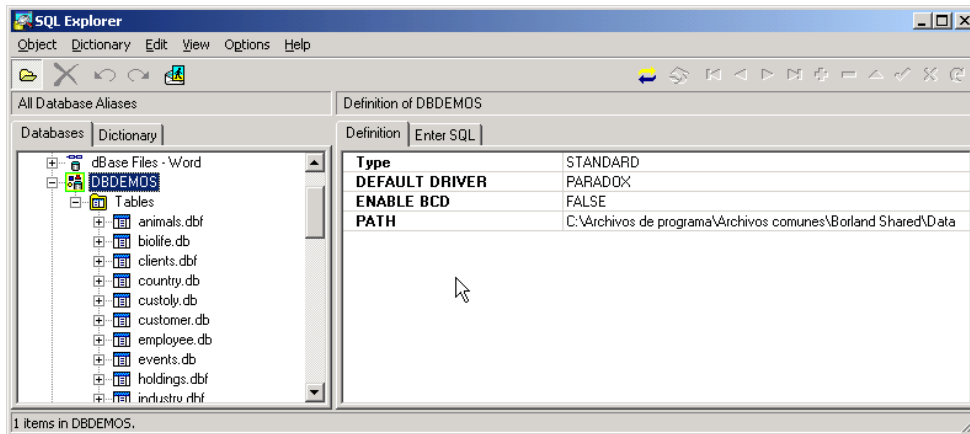


Figura 12: expansión del nodo de tablas

Ahora podemos seleccionar una tabla particular, y veremos una pestaña nueva: **Data** (Datos). Si la seleccionamos, veremos la tabla con todos sus datos en una grilla. Podemos modificar, agregar, borrar y navegar los datos.

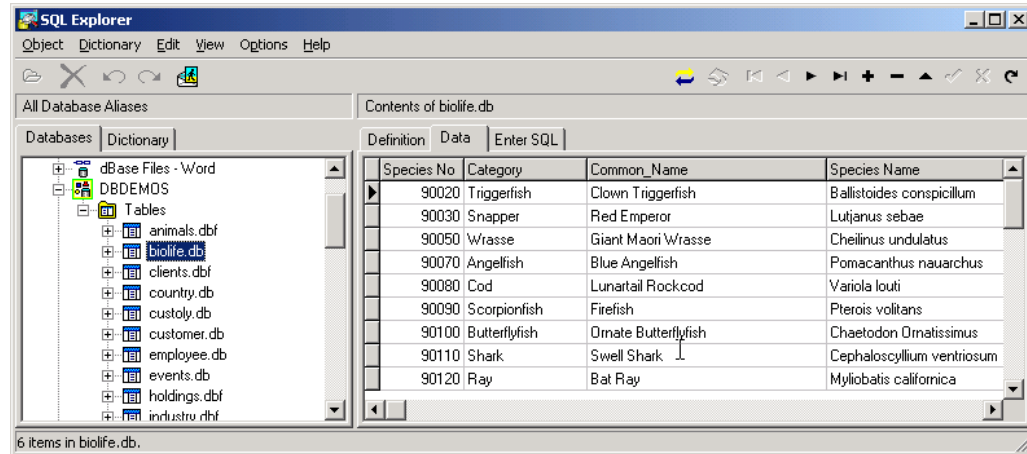


Figura 12: pestaña de datos

Como verán a la izquierda, el nodo de la tabla muestra también un signo +, indicando que se puede expandir todavía más. Veamos qué sucede:

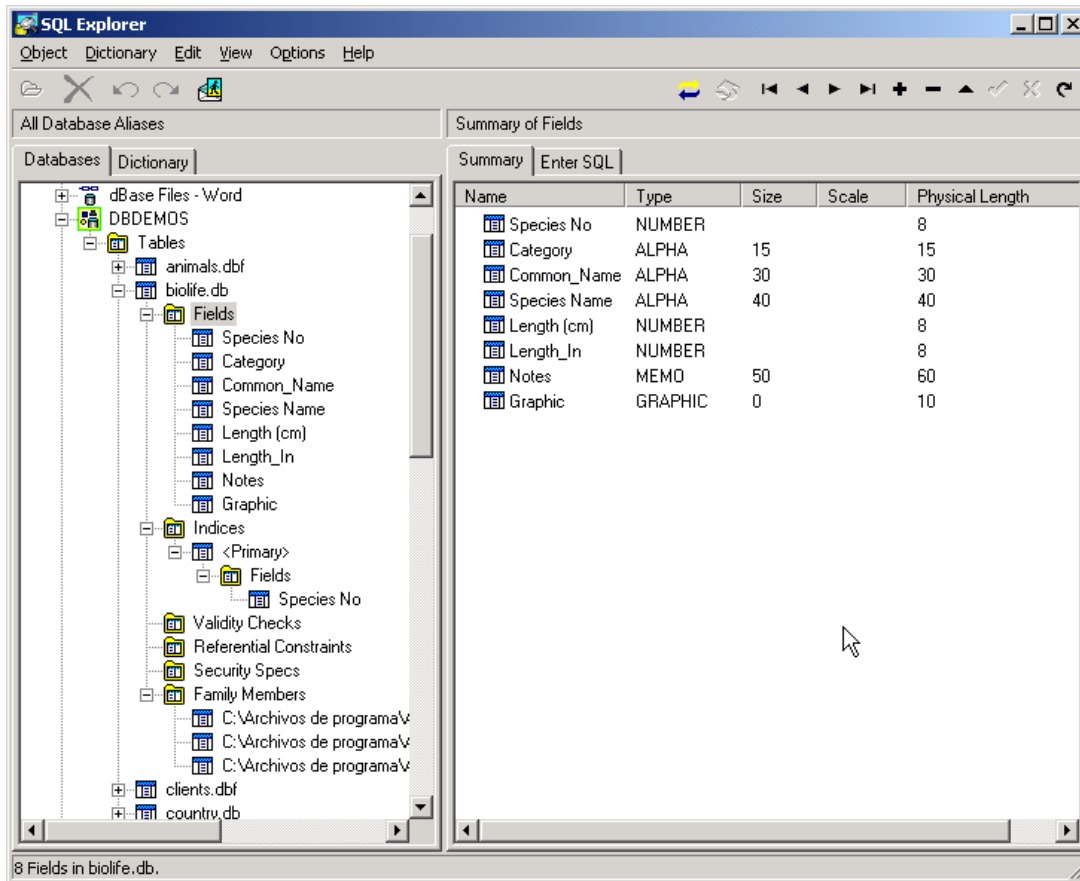


Figura 12: se han expandido todos los nodos de la tabla para ver los detalles de su constitución interna

Como vemos en la figura, se muestran los detalles internos de la tabla en cuestión. Si la BD es de tipo Servidor SQL (SQL Server, Interbase, Oracle, etc) veremos más nodos, mostrando objetos que las simples BD de Paradox o Dbase no poseen.

En definitiva: el Explorador de Bases de Datos es una potente herramienta que no debería faltar en ningún lugar donde se trabaje con Bases de Datos. Es una lástima que no exista una versión independiente del entorno de programación -es necesario instalar Delphi para que se instale el SQL Explorer.

Apéndice I

Los archivos generados por el motor de Paradox

Dado que el motor de Base de Datos de Paradox trabaja con el sistema de archivos de Windows, almacena cada objeto de la Base de Datos en archivos diferentes (uno o más), diferenciándolos por la extensión. A continuación listamos las extensiones de estos archivos y su significado.

NOTA: en todos los casos, el nombre del archivo es el nombre asignado a la tabla al grabarla.

Extensión	Significado
.DB	Definición de la tabla y todos sus datos excepto memos, gráficos y objetos binarios (BLOBs)
.MB	Datos de campos tipo Memo, Gráficos y Binarios (BLOBs)
.PX	Índice primario de la tabla
.Xnn y .Ynn	Cada par representa un índice secundario de la tabla.
.VAL	Definición de Validaciones, Búsqueda rápida (Lookup) e Integridad Referencial

Apéndice II

Tipos de datos soportados por el formato Paradox

NOTA: el formato que utiliza Paradox es del tipo *Registro de longitud fija*, lo que significa que si declaramos un campo como de 40 caracteres, aunque lo dejemos en blanco en realidad estará ocupando 40 bytes en el disco por cada registro.

Nombre Tipo	Tipo de datos - Espacio que ocupa en bytes	En Delphi
Alpha (A)	Alfanumérico - de 1 a 255 bytes	ShortString
LongInteger (I) Autoincrement (+)	Entero entre -2.147.483.648 y 2.147.483.647 (32 bits) - 4 bytes	Integer, Longint
Short (S)	Entero entre -32.768 y 32.767 (16 bits) - 2 bytes	SmallInt
Number (N)	Número real entre $5.0 \cdot 10^{-324}$ y $1.7 \cdot 10^{308}$ - 8 bytes	Double, Real (Delphi 4)
Money (\$)	Número real. Internamente se trabaja en 6 dígitos y en forma entera, independientemente de la cantidad de decimales mostrados. Por defecto se muestra con 2 decimales y el signo monetario. 8 bytes	Currency
BCD (#)	Número en formato BCD (Binary Coded Decimal). Mantenido por compatibilidad con otras aplicaciones que usen este formato, evita los errores de redondeo al trabajar con decimales. 17 bytes	
Timestamp (@)	Fecha y hora juntos - 8 bytes	tDateTime
Date (D)	Fecha - 4 bytes	tDateTime
Time (T)	Hora - 4 bytes	tDateTime
Logical (L)	Valor True o False - 1 byte	Boolean
Bytes (Y)	Campo de bytes, que puede contener hasta 255 bytes con cualquier información. De 1 a 255 bytes.	Byte
Binary (B) OLE (O)	Un conjunto de bytes de longitud arbitraria, por ejemplo para contener un sonido o un objeto OLE. En el archivo .DB se almacenan 10 bytes más lo que indiquemos como longitud, hasta 240. El resto va a un archivo .MB (máximo 64 Mb)	

Nombre Tipo	Tipo de datos - Espacio que ocupa en bytes	En Delphi
Memo (M) Formatted Memo (F)	Texto de longitud arbitraria. En el archivo .DB se almacenan 10 bytes más lo que indiquemos como longitud, hasta 240. El resto va a un archivo .MB (máximo 4 Mb para el memo, 64 Mb para el Memo con formato)	String *
Graphic (G)	Gráfico. En el archivo .DB se almacenan 10 bytes más lo que indiquemos como longitud, hasta 240. El resto va a un archivo .MB (máximo 64 Mb)	

* El tipo String de Delphi está restringido a algo mas de 4 millones de caracteres, lo que en la mayoría de los casos prácticos es más que suficiente y se puede considerar como que no tiene límite.

Apéndice III

Datos internos sobre la estructura y funcionamiento de las tablas de Paradox

- ? Los registros de una tabla de Paradox se almacenan en *bloques*; cada bloque puede ser de 1, 2, 4, 8, 16 o 32 Kb. El valor por defecto es 2 Kb. Este valor se puede cambiar en la configuración de la BDE.
- ? Los bloques -y los registros dentro de los mismos- se almacenan siguiendo el orden dado por el índice primario. Por lo tanto, cuando utilizamos este índice se reduce el tiempo de acceso a los datos.
- ? La cantidad máxima de bloques permitida para una tabla es de 65535; por lo tanto, el tamaño máximo estará dado en relación al tamaño de los bloques:

Tamaño de bloque	Tamaño máximo de tabla
1 Kb	64 Mb
2 Kb	128 Mb
4 Kb	256 Mb
8 Kb	512 Mb
16 Kb	1024 Mb
32 Kb	2048 Mb

En la práctica, mucho antes de alcanzar estos límites la tabla se volverá imposible de manejar (un proceso de reestructuración de una tabla de más de 100 Mb puede tomar *horas*). Paradox está pensado para manejar Base de Datos relativamente chicas, para aplicaciones medias o “de escritorio”.

- ? Cuando se busca un registro, se carga en memoria el bloque que lo contiene y se busca secuencialmente dentro del mismo.
- ? A medida que se insertan y borran registros, se van agregando bloques según sea necesario. Después de un tiempo, es muy posible que los bloques no estén ordenados físicamente como en el índice, y que muchos bloques tengan espacio desperdiciado. Esto se soluciona *compactando* la tabla (Restructure|Pack Table).
- ? Sólo se permite un campo autoincremental por tabla, que se almacena en la cabecera de la misma.
- ? El algoritmo de búsqueda en los índices es de tipo *búsqueda binaria*.
- ? El índice primario, creado automáticamente sobre los campos de la clave primaria de la tabla, almacena *solamente el primer registro de cada bloque*. Cuando se hace una búsqueda, se determina primero el bloque en el que el registro debe estar; se carga este bloque en memoria, y se hace una búsqueda secuencial dentro del mismo. De esta manera se logra un balance entre velocidad de operación y tamaño del índice.
- ? Internamente, el archivo de índice secundario de extensión *.X??* es una *tabla común de Paradox*.
- ? El archivo de índice secundario con extensión *.Y??* es un *índice primario* para la tabla *.X??* correspondiente.

Apéndice IV

Archivos creados por la BDE al usar tablas Paradox

PDOXUSRS.NET - Paradox Network Control File

Lleva la cuenta de los usuarios que acceden a una tabla compartida.

- Debe haber sólo uno en cada sesión de la BDE.
- El valor se da en la opción NET DIR de la configuración de la BDE, o por programa utilizando la propiedad NetFileDir del objeto tSession que corresponda a la sesión utilizada (este cambio se debe hacer antes de que se cree cualquier Database o Dataset)
- Las versiones de la BDE anteriores a la 3.0 *no permitían* asignar a este parámetro valores diferentes para cada usuario; las posteriores permiten hacerlo mientras apunten al mismo archivo físico (por ejemplo utilizando drives mapeados por la red).

PDOXUSRS.LCK - Lock Control File

Contiene información sobre las actividades permitidas en un *directorio* donde hay tablas, y qué está haciendo cada usuario que accede a una tabla. Se crea automáticamente la primera vez que se accede a una tabla en un directorio.

Hay tres tipos de archivos PDOXUSRS.LCK:

- En un directorio compartido el archivo contiene información sobre
 - tablas y registros bloqueados
 - usuario que puso el bloqueo
 - tipo de bloqueo
 - la sesión en que fue establecido el bloqueo
- En el directorio privado de cada usuario (por defecto el mismo de la aplicación), indica que otro usuario *no puede acceder* a las tablas en este directorio. El directorio privado se mantiene en la propiedad PrivateDir del objeto tSession
- En un directorio marcado como de sólo lectura, como cuando tenemos las tablas en un CD-ROM. El acceso es más rápido porque no se permiten los bloqueos (no puede haber escrituras). Se puede crear un archivo con esta marca usando la función de la BDE API DbAcqPersistTableLock: el archivo PARADOX.DRO (Directory Read Only)

PARADOX.NET, PARADOX.LCK, <tabla>.LCK

Archivos del sistema viejo de bloqueo de Paradox 1.0

Muchas veces estos archivos quedan desactualizados al terminar la aplicación en forma imprevista (por ejemplo por un corte de energía), por lo que los siguientes accesos pueden presentar errores de bloqueos o parecidos. Simplemente elimine los archivos *.lck y *.net (cuidado que no haya nadie accediendo a las tablas!).

2 - Bases de datos

Acceso desde Delphi

Los componentes de acceso a datos

El esquema de acceso a los datos utilizado en Delphi hace uso de varias *capas lógicas* entre el archivo físico y los controles (editores, etiquetas, etc) con los cuales interactúa el usuario. De esta manera se logra una cierta independencia no sólo del formato -lo que conseguimos con la BDE- sino también de la forma de acceder a los mismos.

¿Por qué es deseable lo anterior? Porque el programador puede disponer de la forma de acceso que considere adecuada a cada situación, mientras mantiene el núcleo de la interface sin cambios. Así si crea su programa utilizando tablas y luego decide utilizar consultas SQL o algún otro tipo de acceso más adecuado a las condiciones de ese momento puede hacerlo sin mayores complicaciones.

El esquema de acceso en capas se ve en la siguiente figura, donde se muestran los nombres de las clases correspondientes a cada capa. (Fig. 1).

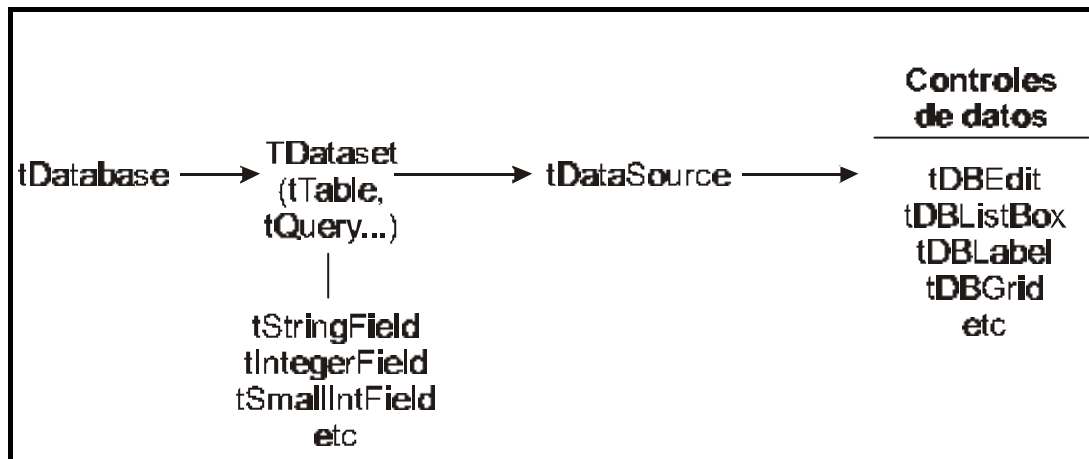


Figura 1: capas de acceso a los datos, con las clases intervinientes

Los únicos componentes visibles son los de la última capa (controles de datos), que son los equivalentes de los controles comunes tales como editores, etiquetas, etc. pero que muestran el contenido de un campo particular de una tabla. No hay que hacer nada más que especificar la fuente de datos (el componente de tipo tDataSource que utilizemos) y el campo que corresponde a cada uno para acceder a sus datos. La recuperación de información desde la tabla y la posterior grabación de los cambios se hacen automáticamente, como veremos.

Los objetos de acceso a los datos se encuentran en la página rotulada “Data Access” de la paleta de componentes. Son los siguientes:



DataSource: es el enlace entre los componentes de datos y los de acceso a tablas. Se relaciona con un tDataSet (Delphi trae definidas las clases tTable y tQuery, pero podemos crear el que necesitemos) a través de la propiedad **DataSet**.



Table: nos da acceso a una tabla con métodos para navegar por ella y acceder a los registros. Se relaciona con el archivo físico (la tabla real) a través de las propiedades **DatabaseName** y

TableName. Los controles de datos se comunican con estos componentes a través de un `tDataSource`.



Query: nos da acceso a los datos de una Base de Datos a través de comandos SQL, con los cuales podemos acceder a más de una tabla relacionada, crear campos calculados, etc. Las características SQL son las que pone a nuestra disposición el motor de datos, por lo que podremos aprovechar todas las opciones avanzadas que da cada servidor particular.



StoredProc: Permite la ejecución de procedimientos SQL almacenados en un servidor de datos. Se aplica comúnmente en el desarrollo de aplicaciones Cliente/Servidor.



Database: representación lógica de una base de datos (un directorio en la implementación actual para tablas de Paradox o Dbase).



BatchMove: permite trabajar con bloques de registros: mover, copiar, borrar, agregar registros de una tabla a otra, etc.



Session: es un objeto que encapsula una sesión de la BDE (luego veremos qué son las sesiones y para qué se utilizan).



UpdateSQL: utilizado con una metodología de acceso a los datos denominada *Cached Updates* (Actualizaciones retenidas). Lo veremos en la sección dedicada a esta forma de acceso.



NestedTables: permite el acceso a *tablas anidadas*, tablas que son campos de otras tablas. Un servidor de datos que implementa dicha característica es Oracle en su versión 8.

Iremos de lo fácil a lo difícil: primero utilizaremos la clase `tTable`, luego veremos un poco de SQL para usar la clase `tQuery` y finalmente veremos un poco la tecnología de Actualizaciones Retenidas (*Cached Updates*).

Acceder a los datos de una tabla

El componente `tTable` es muy simple de usar; sólo debemos poner ciertos valores en las propiedades que le dirán con qué Base de Datos trabajar, y dentro de ésta a qué tabla acceder.

Debemos indicar en la propiedad **DatabaseName** el nombre de la Base de Datos¹. Una vez que hemos

¹ En los casos de Paradox y Dbase, las bases de datos son simplemente directorios, por lo que en la propiedad `DatabaseName` podemos especificar directamente la ruta del directorio que contiene la tabla o un

definido la base de datos a usar, tenemos disponible en la propiedad **TableName** (Nombre de la Tabla) una lista con los nombres de las tablas que pertenecen a esa base de datos. Seleccionamos la deseada. Con estas dos propiedades el componente ya sabe con qué tabla trabajar y podremos navegar por los registros de la misma, filtrarla, limpiarla, etc.

El componente *Table* permite hacer operaciones *a nivel de tabla*, es decir que afectan a toda la tabla a la vez. En cuanto a los datos particulares, se utiliza un *cursor*, una estructura lógica interna que podríamos decir *apunta* hacia un registro particular. Podemos trabajar con los datos del registro apuntado por el cursor en cada momento, por lo tanto trabajamos con los datos *a nivel de registro*. No podemos con este componente leer o escribir el contenido de un solo campo, debemos hacerlo con el registro completo.

Aún no podemos ver los datos. Para acceder a los datos brindados por el *Table* debemos usar un *DataSource*. Éste está relacionado con la tabla a través de la propiedad **DataSet**, donde se puede asignar un *tTable* o un *tQuery*².

Una vez que hemos enlazado estos componentes entre sí y con la tabla física, ya tenemos acceso a los datos desde nuestro programa. Para permitir una sencilla interacción con el usuario, se definen controles análogos a los de la página “Standard” de la paleta, pero con propiedades especiales para comunicarse con una fuente de datos. Estos controles están en la página “Data Controls”.

Los **Controles de Datos** (Data-Aware controls) son componentes comunes como editores, etiquetas, listas, combos, etc. pero que muestran datos de una tabla. Además tenemos otros controles especiales para acceso a las tablas, como el Navegador o la Grilla.

Para comunicar un componente de datos con una tabla debemos indicar dos cosas, en las propiedades correspondientes del componente en cuestión:

DataSource indica la fuente de los datos (el componente *tDataSource*) que usamos.

DataField indica el campo al que accederá este componente dentro del registro. Cuando ya hemos indicado la fuente de los datos, nos aparece una lista de los campos disponibles.

En la figura 2 podemos ver los componentes y propiedades necesarias para acceder al campo “Nombre” de la tabla “Agenda.db” en la base de datos de alias “Agenda” con un editor:

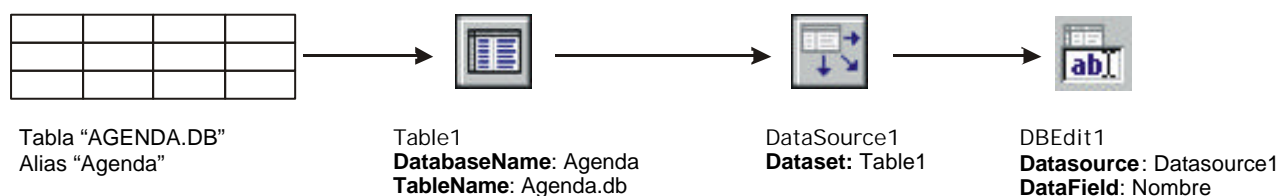


Figura 2: acceso a un campo específico dentro de una tabla

Veamos entonces la creación de una pantalla de entrada y visualización de datos, utilizando la tabla de agenda.

Ejemplo

alias definido para ese directorio (lo más recomendable)

² Técnicamente se espera una instancia de la clase *tDataset* o descendientes, por lo que se puede definir otro componente distinto del *Table* o el *Query*, por ejemplo para acceder a datos sin usar la BDE.

Crear una nueva aplicación. En el form principal agregar un componente DataSource y un Table.

Seleccionamos la tabla. En la propiedad **DatabaseName** debemos colocar el alias que creamos para la Base de Datos de Agenda.

Indicar la tabla AGENDA.DB en la propiedad **TableName**.

Enlazar el DataSource con la tabla, seleccionando la última en la lista que despliega la propiedad **Dataset**.

En este momento ya tenemos establecida una conexión entre la tabla en disco y Delphi. Ahora faltan los componentes que nos permitirán interactuar con los datos.

Agregar en el form cuatro editores de la página Data Controls (DBEdit), un DBMemo y un DBNavigator; enlazarlos con la fuente de datos seleccionando el DataSource en la propiedad del mismo nombre de todos los controles.

Cada uno de estos controles (salvo el Navigator) sirve para mostrar y modificar el contenido de un campo solamente. El nombre de ese campo debe especificarse en la propiedad **FieldName** de cada uno (convenientemente, si el control ya está enlazado a una fuente de datos Delphi muestra en una lista los campos posibles de ser asignados). Por ejemplo, en el primer DBEdit seleccionamos el campo “Nombre y Apellido”.

Además colocar un botón tBitBtn con *kind* = bkClose para cerrar la ventana. Colocar etiquetas y paneles de manera que el form tome la forma de la figura 3.

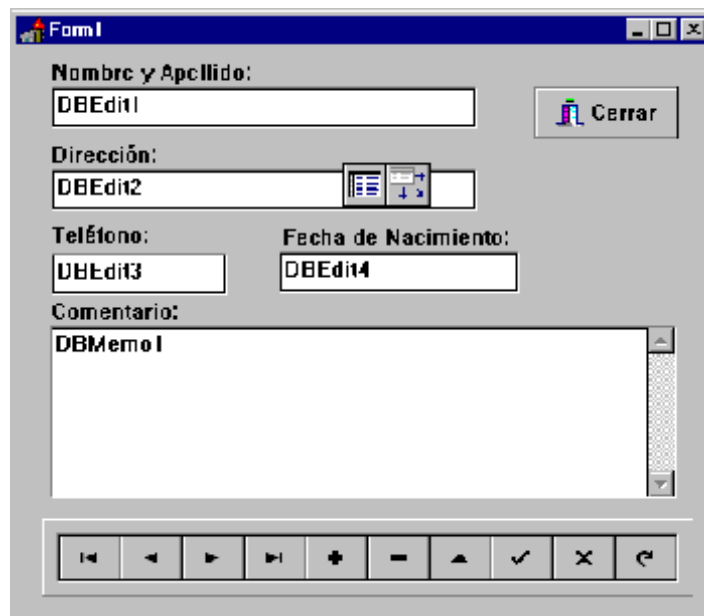


Figura 3: Ficha de Altas, Bajas y Modificaciones para la tabla Agenda

Ya tenemos todo en orden; ahora falta decirle a la tabla que queremos que nos muestre su interior, los registros que contiene. Para ello debemos abrir la tabla asignando el valor TRUE a la propiedad Active de la misma o bien llamando al método Open, de la siguiente manera: `Table1.Open`, por ejemplo en el evento `OnClick` de un botón. Y ¡voilà! Los controles muestran ya el contenido de un registro; si no hemos cambiado nada será el primero de la tabla.

Ejercicio 2-1

Agregar editores para los campos EMAIL y DNI de la tabla AGENDA

)

BIBLIO en funcionamiento

A continuación, transformaremos el proyecto Biblio que comenzamos al principio del curso (para almacenar datos de publicaciones) para que almacene los datos en una Base de Datos, tornándolo funcional.

Recordemos los datos que debemos almacenar, viendo la pantalla de datos:

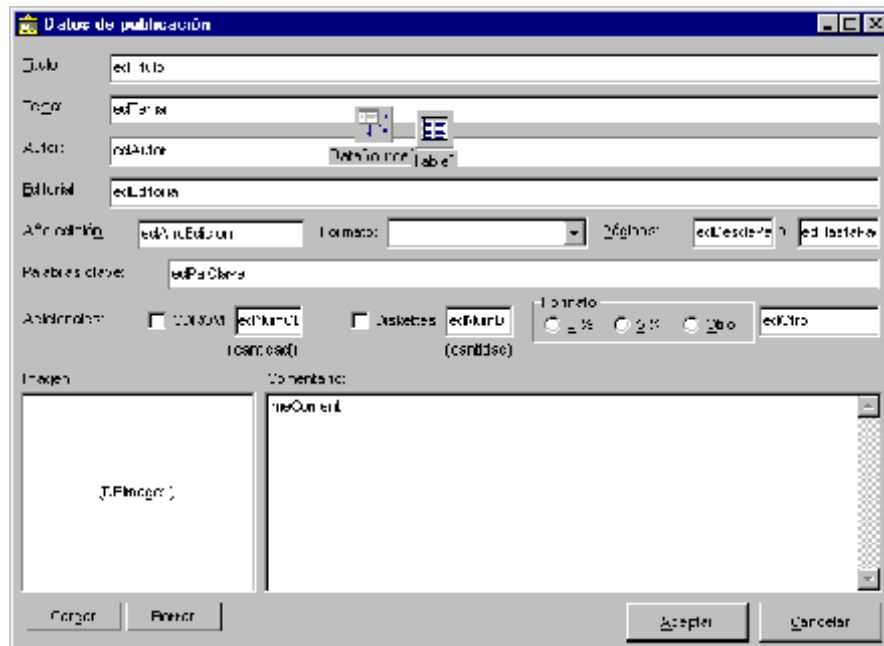


Figura 4: La pantalla de datos una vez terminada

Lo primero que hacemos es determinar la estructura de la tabla necesaria:

Campo	Tipo de dato	Tamaño	Comentarios
Título	A	50	
Tema	A	30	
Autor	A	30	
Editorial	A	30	
Año edición	I		>0, 4 números, por defecto=actual
Formato	A	10	
Página desde	I		>0
Página hasta	I		>0, >= Página desde
Palabras clave	A	50	
CDROM	L		Por defecto=FALSE
Discos	L		Por defecto=FALSE
Cant CD	I		>0

Campo	Tipo de dato	Tamaño	Comentarios
Cant Disc	I		>0
Tipo Disc	I		>=0, <=2
Desc Otro	A	10	
Comentario	M	50	
Imagen	G	0	

En la columna de comentarios están algunas de las restricciones que podemos ver en una primera inspección. Analicemos un poco más esta tabla:

- < Los campos **Título**, **Autor**, **Tema** y **Editorial** son campos alfanuméricos simples que almacenarán directamente lo que se escriba en los editores correspondientes.
- < El campo de **Formato** también es alfanumérico, pero como siempre serán los mismos (Libro, Revista, etc) convendría tener estos valores ya escritos y no permitir al usuario escribir otro que no exista ya. Utilizaremos para ello un componente DBComboBox, que funciona igual que un ComboBox pero almacena el resultado en un campo de la tabla.
- < En general, hasta que hablemos de los componentes de campo más adelante, definiremos las restricciones para los campos en la tabla (validity checks). De esta manera, las restricciones son forzadas por el motor de Bases de Datos y no por Delphi.
- < Los campos numéricos “**Año edicion**”, “**Pagina desde**”, “**Pagina hasta**”, “**Cant CD**”, “**Cant Disc**” y “**Tipo Disc**” no pueden ser negativos; indicamos pues una restricción (validity check) de valor mínimo 0 para cada uno.
- < El campo “**Año edicion**” debe entrarse con 4 dígitos; esto se indica con una restricción de máscara de entrada “####”.
- < El campo numérico “**Tipo Disc**” indica el tipo de disco que acompaña al material impreso. Utilizaremos una convención para hacer más fácil el tratamiento con los controles: “0” indicará discos de 3 ½ pulgadas, “1” indicará discos de 5 1/4 pulgadas y “2” indicará cualquier otro formato (por ejemplo, Zip o Jazz). Por ello, podemos indicar valor mínimo 0 y valor máximo 2.
- < Los campos lógicos “**CDROM**” y “**Discos**” indicarán con valor TRUE (verdadero) que se incluye ese soporte, caso contrario tomarán valor FALSE (Falso).
- < También hemos indicado los valores por defecto para algunos campos. Notemos que el valor por defecto del campo “**Año edicion**” (el año actual) no se puede especificar en la definición de la tabla, ya que Paradox no tiene provisiones para eso. Debemos programarlo en el sistema.

Nos falta todavía la definición de los campos de la clave principal. Para este caso podríamos elegir una combinación como “**Título**” + “**Autor**”. No obstante, utilizaremos otra técnica: agregaremos un campo de tipo Autonumérico con el único propósito de servir como clave principal (debe ser definido en primer lugar).

Ejercicio 2-2

- Crear la tabla del proyecto Biblio en formato Paradox 7, incluyendo todas las restricciones que sea posible especificar.
 - Definir al principio un campo llamado "ID" de tipo autoincremental. Marcarlo como Clave Primaria de la tabla.
 - Grabar la tabla como "BIBLIO.DB".
 - Crear un alias "Biblioteca" que apunte al directorio correspondiente.
-

Ahora podemos enlazar ya todos los componentes de la ventana y comenzar a trabajar con datos reales.

Ejercicio 2-3

- Colocar todos los componentes editores en la ventana de datos. Enlazarlos con los campos correspondientes de la tabla.
-

En la ventana de Biblio hay algunos componentes especiales, que veremos a continuación.

DBComboBox

El componente DBComboBox que utilizaremos para el campo "Formato" es una extensión del mismo componente estándar de Windows. Las propiedades relevantes son las siguientes:

- < **DataSource**: al igual que todos los controles de datos, indica el componente que provee el enlace con la tabla
- < **DataField**: nombre del campo que corresponde a este control.
- < **Items**: lista de valores predefinidos. No es necesario que el campo sea de tipo alfanumérico, las conversiones son realizadas automáticamente por Delphi.
- < **Style**: estilo del control. Los valores posibles son los siguientes
 - < DropDown: permite escribir en el editor, presenta la lista de opciones al presionar el botón
 - < DropDownList: no permite escribir en el editor; solamente elegir un valor de la lista
 - < OwnerDrawFixed: el programa debe encargarse de escribir (o dibujar) los valores que se mostrarán en la lista. Se asume que todos tienen la misma altura. Por cada elemento a dibujar se produce el evento OnDrawItem
 - < OwnerDrawVariable: el programa debe escribir o dibujar los valores de la lista. Se puede variar la altura de cada uno. Por cada elemento a dibujar se producen los eventos OnMeasureItem y OnDrawItem.

< Simple: se comporta igual que un editor común, sin mostrar el botón para desplegar la lista.

Coloquemos entonces el ComboBox del campo *Formato* con los siguientes items: Libro, Revista, Folleto, Fotocopias, Otros. Dado que no queremos que se puedan escribir otros valores diferentes a estos, ponemos el estilo igual a `csDropDownList`.

DBCheckBox

Los cuadros de opción utilizados para indicar si la publicación tiene discos o CD sirven para cualquier campo que sólo pueda tomar dos valores diferentes, como los de tipo Lógico en Paradox. Las propiedades especiales de estos controles son:

- < **DataSource, DataField:** igual que en los demás controles de datos.
- < **ValueChecked:** valor que se colocará en el campo cuando el cuadro se encuentre marcado. Asimismo, cuando el campo contenga este valor se verá una marca en el cuadro. Por defecto = TRUE, adecuado para campos de tipo lógico.
- < **ValueUnchecked:** valor que se pondrá en el campo cuando el cuadro no esté marcado. Al mostrar el contenido de un registro, si el campo contiene este valor se verá el cuadro sin marcar. Por defecto = FALSE, adecuado para campos de tipo lógico.

Vemos que la utilización de estos controles está adaptada por defecto a los campos de tipo lógico, pero no restringida a éstos. Por ejemplo, si tenemos un campo de tipo entero que debe tomar sólo los valores 0 y 1 podemos usar un control DBCheckBox sin problemas.

Ejercicio 2-4

Colocar los DBCheckBox para los campos “CDROM” y “Discos”

DBRadioGroup

Otro de los componentes estándar especialmente adaptado para trabajar con un campo de una tabla es el Grupo de botones radiales (RadioGroup). Representa un conjunto de valores mutuamente excluyentes (sólo se puede elegir uno de ellos) y permite una representación fácil de leer de un campo codificado. En nuestro ejemplo tenemos uno de esos campos: “Tipo Disc”, que acepta un valor entero entre 0 y 2. Cada uno de estos valores representa un tipo de disco, pero no queremos obligar al usuario a aprender esa codificación. Para él (o ella) debe ser transparente. Aquí entra en juego el componente DBRadioGroup.

Las propiedades más importantes son las siguientes:

- < **DataSource, DataField:** igual que en los demás controles de datos.
- < **Items:** lista de etiquetas para cada botón. Esto es lo que se lee al costado de cada botón, no lo que se

coloca en el campo de la tabla. Se colocan en orden, uno por línea.

- < **Columns**: cantidad de columnas en que se tienen que acomodar los botones. En nuestro ejemplo serán tres.
- < **Values**: valores que se colocarán en el campo al seleccionar un botón. Se toman en el orden en que están escritos, uno por línea.

Colocaremos entonces el grupo de botones para el campo “Tipo Disc”, indicando como

Items: “3 ½”, “5 ¼”, “Otros”

Values: “0”, “1”, “2”

Columns: 3

De esta manera el usuario selecciona “3 ½” y en la tabla se colocará un “0”. Dado que lo contrario también es cierto, el usuario nunca se entera de la codificación utilizada.

DBMemo

El campo de comentario es un campo de texto, pero un poco especial: no queremos restringir la libre expresión de la persona que introduzca el comentario a unos míseros 240 caracteres, no? Por eso definimos el campo como de tipo “Memo”, y debemos utilizar un componente acorde a esa definición. Como se habrán podido imaginar, el componente en cuestión es una extensión del Memo estándar y se llama... DBMemo. No tiene propiedades especiales además de **DataSource** y **DataField**.

Ejercicio 2-5

Agregar un DBMemo para el campo “Comentarios”

DBRichEdit

Existe otro componente para trabajar con los campos de texto, que permite especificar un formato además del texto: DBRichEdit. Este control asume que el texto tiene formato de texto enriquecido (RTF, Rich-Text Format) a menos que pongamos la propiedad **PlainText** en True. Es una opción a tener en cuenta si queremos ofrecer la opción de formatear cada párrafo por separado (por ejemplo, distintos tipos de letras o colores).

Este componente es la contrapartida del RichEdit estándar de Win32, y no tiene propiedades especiales además de **DataSource** y **DataField**.

DBImage

El último campo es un agregado que en otras épocas resultaba un lujo -y una característica que podía definir la compra de un sistema en lugar de otro: la posibilidad de agregar una imagen del documento, obtenida mediante un escáner o cámara digital. Ahora, bajo Windows, es algo muy fácil de hacer. Paradox incorpora un tipo de campo especial para almacenar imágenes, el tipo "G" (por Graphic), y Delphi tiene un control asociado: DBImage. El funcionamiento es el mismo que el componente estándar Image, pero como de costumbre los datos van a parar a la tabla.

Para trabajar con este control es necesario tener en cuenta las siguientes propiedades:

- < **DataSource, DataField:** igual que los demás controles de datos
- < **AutoDisplay:** si está en FALSE, los datos no se muestran automáticamente; hay que llamar por programa al método LoadPicture. Muy útil cuando no queremos demorar la navegación con imágenes muy grandes; por ejemplo, únicamente llamar a LoadPicture después de detenernos un par de segundos en un registro.
- < **Center:** indica que la imagen aparecerá centrada en el control
- < **Stretch:** la imagen se redimensiona al tamaño del control.
- < **QuickDraw:** si es TRUE, la imagen se muestra con los colores de la paleta del sistema; caso contrario, se crea una paleta especial para una mejor concordancia de los colores de cada imagen. El primer método resulta en una imagen que aparece más rápidamente pero tiene menor calidad.
- < **Picture:** contiene la imagen. Esta propiedad está disponible sólo en tiempo de ejecución, dado que se llena automáticamente a través del enlace a la fuente de datos (a diferencia de la misma propiedad del control Image). Si asignamos a esta propiedad un archivo gráfico o invocamos el método **Picture.LoadFromFile** efectivamente estamos cambiando el contenido del campo de la tabla.

NOTA: si agregamos la unit JPEG a la cláusula **uses** de nuestro programa, podremos cargar imágenes JPG; no obstante, el campo Graphic de Paradox no reconoce el formato y no podremos ingresar la imagen en la tabla.

Tenemos que dar al usuario la posibilidad de cargar una imagen desde un archivo externo y también de eliminarla si no se usa más; para eso son los dos botones inferiores. Para agregar o borrar datos de un registro debemos llamar a algunos métodos del componente Table, por lo que postergaremos la discusión hasta más tarde.

Ahora tenemos lista la pantalla de edición de datos de publicaciones. Aún queda por ver la forma en que indicamos a la tabla que queremos guardar los datos ingresados, o cancelar las modificaciones en caso contrario.

Navegando se llega a Roma (¿?)

Cursores

Los datos en una tabla se almacenan en registros, como ya dijimos; la BDE utiliza el concepto de *cursor* para acceder a los datos.

Un *cursor* es como un marcador de posición; se “ubica” sobre una fila de la tabla (un registro) y nos da acceso a sus datos. La única manera de acceder a los valores almacenados en un campo particular de un registro es posicionar un cursor sobre ese registro y a través de los controles de datos -que “ven” lo que el cursor les muestra- modificar el contenido de los campos.

Es posible indicar al cursor que se mueva al principio o al final de la tabla, o bien hacia adelante o hacia atrás de la posición actual. Pero tenemos que tener en cuenta que el concepto de “el registro número *n*” ya no es aplicable. Veamos por qué.

Supongamos que numero los registros en forma consecutiva, como se hacía antes por ejemplo en Dbase. Entonces, podré emitir alguna orden para editar el contenido del registro número 4, por decir un número cualquiera. Modifico algunos datos y luego lo grabo en la misma posición -la número 4. Ahora bien, ¿qué sucede si mientras estoy editando ese registro *alguien* -la mano negra- accede a la tabla a través de la red y me elimina el segundo registro de la tabla? El registro número 4 será ahora el que ocupaba la posición siguiente y por lo tanto modifiqué los datos equivocados. Por esto,

*No hay un método directo para indicar al cursor que se posicione en el registro número *n*.*

Hace un tiempo no era común tener varios equipos conectados en red, por lo que este problema no se veía en la mayoría de los casos. No obstante ahora generalmente nuestros programas se ejecutarán en una red local, aunque al principio no lo planeemos así.

Hay una forma de conocer el número secuencial de registro en el que estamos posicionados con el cursor, llamando a la BDE directamente; no hablaremos ahora de eso. Para movernos con seguridad entre los registros, utilizaremos *Marcadores* (Bookmarks) y métodos de búsqueda.

El Navegador

El *navegador* (navigator) es como un “control remoto” del cursor; una botonera que nos permitirá trabajar con los registros de una tabla. Podemos agregar, borrar, editar, aceptar o cancelar las modificaciones, recargar los datos desde el servidor y movernos (navegar) entre los registros.

Los botones del navegador y sus funciones son los siguientes:

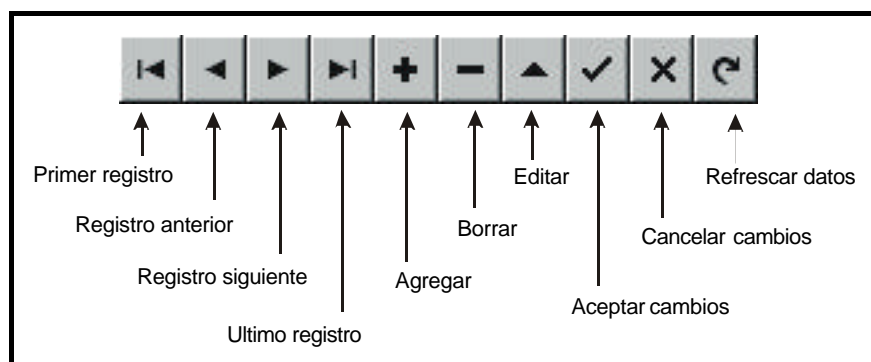


Figura 5: Navegador de tablas con la función de cada botón

Por ejemplo, si queremos agregar un registro a la tabla presionamos el botón “+”: la BDE crea un registro

nuevo en memoria -virtual- y posiciona el cursor en él. Vemos entonces en los controles de datos el contenido por defecto de los campos.

Para que los datos que ingresemos se graben en la tabla física, debemos confirmarlos: la acción se denomina “Post”. En el navegador hay un botón especial para esta operación, el que tiene una marca “U”. Si por el contrario no deseamos confirmar los cambios, disponemos del botón Cancelar: “X”. Una vez presionado este último botón, los datos ingresados se pierden y el cursor vuelve a la tabla real.

También es posible confirmar los datos de una forma indirecta; cuando hay modificaciones y nos movemos a otro registro, la BDE considera que hemos aceptado los cambios y los confirma, efectuando así un “Post implícito”.

Los datos de un registro no se pueden modificar directamente; hay que decirle a la BDE que deseamos hacerlo y entonces ésta creará una copia virtual del registro en memoria para que podamos modificarla. Esta operación se denomina “edición”, y el cursor tiene un comando para hacerlo. En el navegador disponemos del botón “• ”.

Luego de modificar los datos debemos aceptar los cambios con “Post” (U) o rechazarlos con “Cancel” (X).

Todos los botones del navegador llaman a procedimientos de la tabla correspondiente. Por ejemplo, si el navegador está enlazado al componente **Table1** y presionamos el botón “U”, el navegador llama al método **Table1.Post**. Por supuesto que nosotros también podemos llamar directamente a estos métodos de la tabla.

A continuación damos el nombre de cada método invocado por los botones de un navegador:



Con esta información, podemos armar nuestro propio navegador.

Ejercicio 2-6

Crear un navegador que se pueda utilizar con las teclas, y que muestre dibujos y también un texto en cada botón.

Veremos ahora algunos ejemplos, utilizando el form de entrada de datos de Biblio. Escribiremos los eventos correspondientes de los botones “Cargar” y “Borrar”, que se encargan de la gestión de la imagen del campo “Imagen”.

El botón “Cargar” debe llamar a `DBImage1.Picture.LoadFromFile` luego de obtener el nombre del archivo por algún medio (por ejemplo, invocando primero el cuadro de diálogo estándar de Abrir Archivo). Una primera versión del evento **OnClick** sería la siguiente:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if OpenPictureDialog1.Execute then
        DBImage1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

Hemos utilizado un cuadro de diálogo estándar de abrir archivo de imagen (con vista previa, está en la página “Dialogs”) llamado `OpenPictureDialog1`.

Hay un problema con el programa anterior: no podemos modificar el contenido de la tabla sin ponerla antes en modo de edición. Veremos la imagen en pantalla, pero no se guardará en la tabla. Modificaremos entonces nuestro evento, de la siguiente manera:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if OpenPictureDialog1.Execute then begin
        Table1.Edit;
        DBImage1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
    end;
end;
```

Ahora si, no hay problemas y la imagen se inserta en el registro actual de la tabla.

El botón “Borrar” debe eliminar la imagen. Para eso debemos asignar **nil** (nada) a la propiedad `Picture` del `DBImage`:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    Table1.Edit;
    DBImage1.Picture.Assign(nil);
end;
```

NOTA: no hay que olvidarse de aceptar o cancelar los cambios con **Post** o **Cancel**, respectivamente!

De grillas, tablas y otras yerbas

En los tiempos de Windows ya no es suficiente con mostrar un registro a la vez. El usuario ahora dispone de un arma de gran alcance: el ratón. Ya no está obligado a moverse línea por línea en una tabla, puede ir directamente a donde desee con sólo presionar un botón. Para aprovechar este poder que papá Bill ha dado al usuario, los programadores debemos rompernos la cabeza ideando formas nuevas de presentar la información... y de protegernos de un montón de posibilidades de error más que las que había antes!

La forma de presentación más intuitiva para una tabla es... una tabla. Sería bueno tener un componente que mostrara la información en forma de tabla. Lo mismo pensaron los muchachos en Borland y crearon el componente `tDBGrid` (📄). Este componente se ve gráficamente como una tabla formada por filas y columnas; cada fila tiene los datos de un registro, cada columna es un campo. Por lo tanto, cada celda es un campo de un registro determinado (fig. 7).

El usuario puede efectivamente posicionarse en cualquier celda (o en forma equivalente, en cualquier campo de cualquier registro) pulsando el ratón sobre ella. A continuación puede editar el contenido directamente.

Si bien es una interface cómoda y práctica, también es peligrosa; un usuario sin experiencia puede inadvertidamente modificar datos de una celda y luego pulsar en otra fila con lo cual el cursor se mueve y se efectúa un Post implícito, aceptándose los cambios. Por ello tenemos disponibles distintas opciones para personalizar este componente y eventualmente restringir el acceso a los datos.

La funcionalidad básica requiere muy pocos cambios: sólo hay que especificar la fuente de datos en la propiedad `DataSource` (que muestra una lista de las que estén disponibles) y ya tenemos acceso a la tabla. Es más, disponemos de la funcionalidad de un navegador dado que hay teclas especiales para insertar (**Insert**) y borrar (**Ctrl+Delete**) filas, así como para cancelar los cambios (**Escape**). Para aceptar las modificaciones basta con moverse a otra fila.

La primera fila de una grilla muestra los títulos de las columnas (por defecto, los nombres de los campos). Es posible modificar en tiempo de ejecución el orden de las columnas arrastrando los títulos, o el ancho de las mismas arrastrando la línea divisoria³ (Fig. 8).



Figura 7: Componente DBGrid

³ Note que las nuevas posiciones y tamaños de las columnas *no son permanentes*, es decir, cuando cerremos la ventana en la que aparece la grilla y la volvamos a abrir los valores serán otra vez los originales.

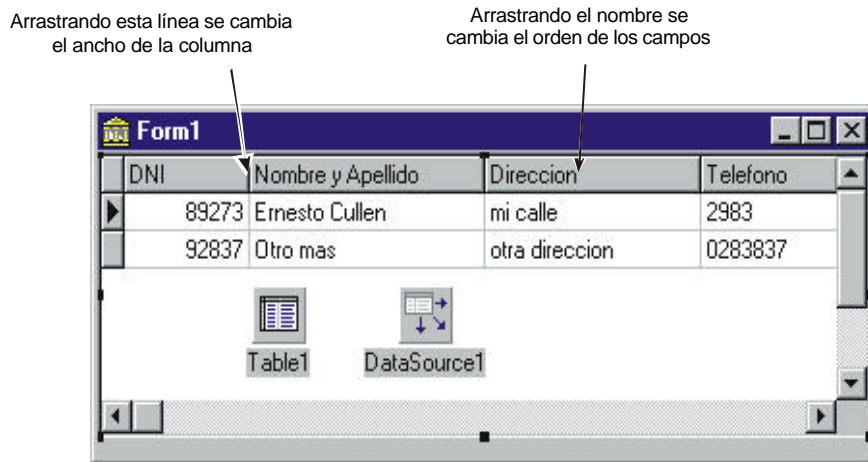


Figura 8: modificaciones que el usuario puede hacer al aspecto de una grilla

Modos de una tabla

La primera columna también es especial: presenta información acerca del estado de la tabla y la posición del cursor. Los símbolos que se muestran tienen los siguientes significados:

- < La tabla está en modo “navegación” (browse). Sólo estamos mirando, no hay modificaciones.
- I La tabla está en modo “edición” (edit). Se están modificando los datos de un registro existente.
- * La tabla está en modo “inserción” (insert). Se está trabajando en un registro nuevo.

Cuando se está trabajando sobre los datos (modos “edición” e “inserción”) se trabaja sobre una copia de los datos en memoria; para que esos datos vayan a parar al archivo en disco es necesario hacer **Post**. Si se cancela la operación, simplemente se elimina la copia en memoria. En cualquiera de los casos, se vuelve al modo navegación.

Ejercicio 2-7

Crear una aplicación con un form que contenga lo necesario para acceder a la tabla NOMBRES.DB (la agenda creada anteriormente), y una grilla para ver los datos. Tomar nota de los campos que se muestran en la grilla, los títulos, el ancho de las columnas, el orden, la posibilidad de edición. Intente “arrastrar” una columna a otra posición o redimensionarlas arrastrando la línea divisoria de los títulos, tanto en tiempo de diseño como en tiempo de ejecución.

Cambiar la presentación y el comportamiento por defecto de la grilla

En la mayoría de los programas reales, modificaremos el aspecto y/o el comportamiento de la grilla estándar. En lugar de dar aquí una lista de las propiedades implicadas y su significado (que se puede obtener

fácilmente en la ayuda), veremos cómo realizar tareas comunes y resolver problemas que se presentan a menudo.

Cambiar la letra de los títulos

Como todos los componentes visuales, la grilla posee una propiedad **Font** que contiene las especificaciones del tipo de letra que se usará para mostrar los datos. Existe además otra propiedad llamada **TitleFont**, que indica el tipo de letra que se usará para los títulos. Se utiliza de la misma manera que la propiedad **Font**.

Impedir cambios a los datos de la grilla

Se pueden prevenir algunos errores (sobre todo el borrado accidental de datos) poniendo la propiedad **ReadOnly** en **True**. A partir de ese momento, cualquier acción de edición de los datos en la grilla está prohibida (incluso si ponemos por programa a la tabla en modo de edición, no podremos escribir dentro de la grilla).

Se puede lograr el mismo resultado quitando la constante `dgEditing` de la propiedad **Options** (en el inspector de objetos le pondremos valor **False** a esa constante)

Opciones para modificar la edición

Tomemos un tiempo para ver cómo es el comportamiento normal de una grilla. Cuando el componente recibe el foco, se muestra una celda en otro color (seleccionada). Para realizar cambios a los datos contenidos en esa celda (un campo particular del registro actual) debemos presionar el botón **•** en el navegador o bien `<Enter>` en el teclado, con lo cual la tabla se pone en modo de edición. Una vez realizados los cambios, podremos aceptarlos moviéndonos a otras filas (otro registro) o bien cancelar los cambios presionando `<Esc>`. Antes de entrar en edición, podremos movernos de una celda a otra utilizando las teclas de cursor; en cualquier momento, pasamos al siguiente campo con `<Tab>` y volvemos al campo anterior con `<Shift>+<Tab>`.

¿Se puede hacer que se pase a la siguiente columna con `<Enter>`? **Si!!!** tenemos que capturar un evento de teclado y responder en consecuencia. Para cambiar el número de columna que contiene la celda seleccionada en una grilla, disponemos de la propiedad `SelectedIndex`. Por lo tanto, el evento `OnKeyPress` de la grilla puede codificarse como sigue:

```
procedure TForm1.DBGrid1KeyPress(Sender: TObject; var Key: Char);
begin
  if key=#13 then
    dbGrid1.SelectedIndex:= (dbGrid1.SelectedIndex + 1) mod dbGrid1.Columns.Count;
  end;
```

La función `Mod` evita que nos pasemos de la última columna; directamente salta a la primera, sin cambiar de registro.

Tenemos algunas posibilidades de modificación del comportamiento estándar de las grillas en la propiedad `Options`. Notemos que esta propiedad es de tipo conjunto, lo que significa que en el inspector de objetos

tendremos que agregar cada opción por separado dándole un valor verdadero mientras que por programa debemos asignar a esta propiedad una lista entre corchetes de las constantes predefinidas (las que empiezan con “dg”).

Las opciones que tienen efectos sobre la edición son `dgEditing` y `dgRowSelect`. Si se indica la segunda, la primera es ignorada y no se permiten cambios a los datos.

Opciones para modificar el aspecto

Hay varias opciones que modifican la presentación de la grilla. Pruebe el comportamiento de las opciones `dgRowLines`, `dgColLines`, `dgIndicator`, `dgAlwaysShowEditor`, `dgAlwaysShowSelection`, `dgTitles`. Estas opciones sólo tienen efecto sobre el aspecto de la grilla, no sobre la forma de trabajar con los datos.

Modificar las columnas

La opción `dgColumnResize` permite o prohíbe la modificación del ancho de las columnas en tiempo de ejecución. Existe una manera de cambiar las características de cada columna en forma permanente: la propiedad **Columns**.

La propiedad **Columns** es una *colección* de objetos, instancias de la clase `TColumn`, cada uno de los cuales representa una columna. Estos objetos tienen propiedades que se pueden ver y modificar en el Inspector de Objetos. Por ejemplo: el color, tipo de letra y título se pueden cambiar en tiempo de diseño para cada columna en la grilla. La clase `TColumn` tiene otras propiedades no visibles en el Inspector de Objetos, por ejemplo *Field* que representa el componente de campo con el que está asociada. Utilizaremos algunas de estas propiedades en los ejemplos que siguen.

Las distintas columnas se pueden acceder a través del índice, el número de orden en la lista, como si la colección fuese un vector de columnas (comenzado en 0). Así por ejemplo, la primera columna del **dbGrid1** será **dbGrid1.Columns[0]** y la última **dbGrid1.Columns[dbGrid1.Columns.count-1]**.

Si no cambiamos el valor inicial de la propiedad **Columns** (vacía), se mostrarán todas las columnas de la tabla con el formato por defecto. Si modificamos esta propiedad, se mostrará sólo lo que se indique en ella.

Cuando presionamos el botón de edición de la propiedad **Columns** aparece la lista de columnas de esa grilla (figura 9). Llamaremos a esta ventana el *Editor de columnas*.

Para agregar las columnas a la tabla podemos presionar el botón de “Agregar todos los campos” (el que tiene tres puntos), con lo cual se agrega una columna por cada campo de la tabla, con los valores por defecto de la definición. Para ver las propiedades de una columna determinada, la seleccionamos en el editor y miramos el Inspector de Objetos.

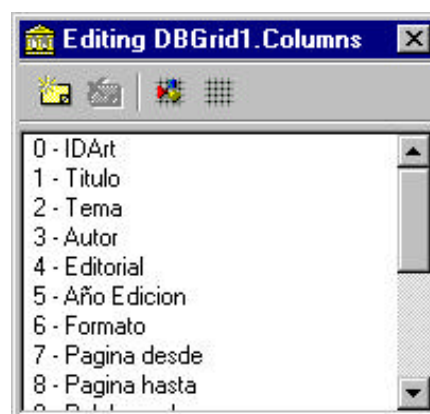


Figura 9: Editor de columnas con todas las columnas de la tabla

Ocultar y reordenar campos

Podemos ahora ocultar cualquiera de las columnas (campos) poniendo su propiedad **Visible** en **False**.

Para reordenar las columnas, simplemente las arrastramos en la lista tomándolas del número.

Cambiar el ancho de una columna

Las columnas tienen una propiedad **Width** que indica el ancho en pixels. Por defecto, este ancho se calcula en base al tamaño del dato que contiene el campo correspondiente. También, una vez que hemos definido las columnas, podemos redimensionarlas simplemente arrastrando las líneas divisoras de los títulos... y las nuevas dimensiones serán permanentes, cada vez que el programa arranque cada columna tomará el ancho especificado.

Formato de presentación de las columnas

Podemos cambiar varias características de presentación de las columnas:

- < El título (**Title**) es en sí mismo una estructura compleja con varias subopciones, que nos permiten cambiar la tipografía, el color, el texto y la alineación.
- < **Alignment** mantiene la alineación de los datos en la columna, en forma independiente del título.
- < **Font** contiene la fuente tipográfica de los datos de la columna.
- < Podemos hacer una columna de sólo lectura con **ReadOnly**.
- < La propiedad **PickList** mantiene una lista de palabras que se mostrará como un ComboBox en cada celda de la columna. Notemos que el Combo no aparece hasta que empezamos a editar el contenido.

Combinando estas posibilidades con los componentes de campo (el tema que veremos a continuación), podremos presentar grillas realmente muy elaboradas con poco esfuerzo.

Ejemplo 1: permitir personalización del ancho de una columna por el usuario

Se desea una aplicación que permita al usuario modificar el ancho de la primera columna de una grilla. El form principal podría ser como el de la figura 10. Al presionar el botón etiquetado "<", la primera columna se achicará un pixel; con el otro botón, se ensanchará.

El código de los botones podría ser el siguiente (BitBtn1: izquierda, BitBtn2: derecha):

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
    dbgrid1.Columns[0].Width:=
    dbgrid1.Columns[0].Width-1;
end;
```

```
procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
```

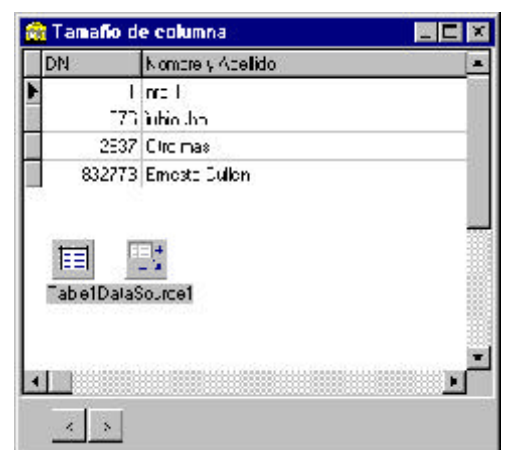


Figura 10: programa que permite modificar el ancho de la primera columna


```

    dbgrid1.Columns[0].Width:= dbgrid1.Columns[0].Width+1;
end;

```

Podemos ampliar este último ejemplo permitiendo al usuario que seleccione de una lista la columna sobre la que quiere actuar; para la selección utilizaremos un **ComboBox**, que tendremos que llenar con los nombres de las columnas visibles.

¿En qué evento llenaremos el Combo con los nombres de las columnas visibles? Dado que queremos que aparezca desde el inicio del programa, lo haremos en el evento **OnCreate**:

```

procedure TForm1.FormCreate(Sender: TObject);
var
    i: integer;
begin
    for i:= 0 to dbGrid1.Columns.Count-1 do
        If dbGrid1.Columns[i].Visible then
            ComboBox1.Items.Add(dbGrid1.Columns[i].Title.Caption);
    ComboBox1.Text:= dbGrid1.Columns[0].Title.Caption;
end;

```

Ahora bien: al presionar cualquiera de los botones, se modificará el ancho de la columna indicada en el **ComboBox**. Para ello necesitamos el *índice* de la columna. ¿Cómo podemos saber el índice de la columna cuyo nombre seleccionamos en el combo?

Una forma sencilla sería tomando directamente la propiedad **ItemIndex** del **ComboBox** como índice de la columna. Dado que fueron introducidas en el orden que se encontraban en la grilla, debe ser el mismo. Esta solución se ve en el siguiente listado, de los eventos **OnClick** de los dos botones:

```

procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
    dbgrid1.Columns[ComboBox1.ItemIndex].Width:=
dbgrid1.Columns[ComboBox1.ItemIndex].Width-1;
end;

```

```

procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
    dbgrid1.Columns[ComboBox1.ItemIndex].Width:=
dbgrid1.Columns[ComboBox1.ItemIndex].Width+1;
end;

```

El problema con esta aproximación simplista es que cuando tenemos columnas no visibles, éstas no se ingresan en el **ComboBox**; por lo tanto, el orden de los nombres en la lista ya no será igual que el orden de las columnas en la grilla. Pueden probarlo ocultando la primera columna (**Visible=False**) y modificando el

ancho de cualquiera de las que aparecen en el ComboBox. ¿Y ahora, quién podrá defendernos?⁴

Una solución sería crear una función que dado el nombre de una columna nos devuelva el índice de la misma en la colección **Columns**. Esta función podría ser algo como lo siguiente:

```
function TForm1.IndiceDeColumna(const NombreCol: string): integer;
var
    i: integer;
begin
    Result:= -1; //Valor que indicaria que no se encontro la columna
    for i:= 0 to dbGrid1.Columns.Count-1 do
    begin
        if dbGrid1.Columns[i].Title.Caption=NombreCol then
        begin
            Result:= i;
            break; //Termina el ciclo for
        end;
    end;
end;
```

Ahora si, utilizando esta función los eventos **OnClick** de los botones se hacen simples:

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
var
    Temp: integer;
begin
    Temp:= IndiceDeColumna(ComboBox1.Text);
    dbgrid1.Columns[Temp].Width:= dbgrid1.Columns[Temp].Width-1;
end;

procedure TForm1.SpeedButton2Click(Sender: TObject);
var
    Temp: integer;
begin
    Temp:= IndiceDeColumna(ComboBox1.Text);
    dbgrid1.Columns[Temp].Width:= dbgrid1.Columns[Temp].Width+1;
end;
```

Sencillo, ¿no?

⁴ Un tributo

Ejemplo 2: movimiento de columnas en tiempo de ejecución

Todavía podemos hacer más; pondremos otro par de botones que permitan *mover* de lugar la columna seleccionada en el ComboBox.

Para mover la columna, podemos tomar varios caminos; en el ejemplo siguiente he ilustrado dos de ellos.

El movimiento hacia atrás lo lleva a cabo intercambiando los componentes Tcolumn que representan la columna seleccionada en el ComboBox y la columna anterior:

```
procedure TForm1.SpeedButton3Click(Sender: TObject);
var
    temp: integer;
    ColTemp: TColumn;
begin
    //Mueve una columna hacia atras, copiando el objeto TColumn
    temp:= IndiceDeColumna(ComboBox1.Text);
    if temp>0 then begin
        ColTemp:= TColumn.Create(dbGrid1.Columns);
        ColTemp.Assign(dbGrid1.Columns[temp-1]);
        dbGrid1.Columns[temp-1].Assign(dbGrid1.Columns[temp]);
        dbGrid1.Columns[temp].Assign(ColTemp);
        ColTemp.Free;
    end;
end;
```

Como vemos, necesitamos un componente Tcolumn temporal para hacer el intercambio.

El movimiento hacia adelante se hace intercambiando las propiedades *FieldName* de los objetos Tcolumn que representan la columna seleccionada en el Combo y la siguiente. Necesitamos sólo un string auxiliar para el intercambio:

```
procedure TForm1.SpeedButton4Click(Sender: TObject);
var
    temp: integer;
    StrTemp: string;
begin
    //Mueve una columna para adelante, copiando la propiedad FieldName
    Temp:= IndiceDeColumna(ComboBox1.Text);
    if temp<dbGrid1.Columns.Count-1 then begin
        StrTemp:= dbGrid1.Columns[temp+1].FieldName;
        dbGrid1.Columns[Temp+1].FieldName:= dbGrid1.Columns[temp].FieldName;
        dbGrid1.Columns[Temp].FieldName:= StrTemp;
    end;
end;
```

```
end;  
end;
```

Hay otra posibilidad para modificar la presentación de los datos en una grilla: *dibujarlos* nosotros mismos. Exploraremos ahora esa posibilidad.

“Dibujar” los datos

Teniendo todas las posibilidades que tienen las grillas, ¿para qué podemos querer dibujar el contenido de las celdas? Hay varias ocasiones en que es deseable este grado de control: por ejemplo, para poner de distinto color las celdas con valores positivos y negativos; para mostrar el contenido de los campos gráficos en la grilla; para mostrar el texto de los campos memo, etc.

Lo primero es indicar a Delphi que deseamos escribir los datos por nuestra cuenta. Para esto, ponemos la propiedad **DefaultDrawing** en **False**. A partir de ahora, cada vez que la grilla vaya a graficar una celda generará un evento **OnDrawColumnCell**⁵. Es aquí donde tomamos el control.

Veamos un poco el evento y los datos que nos ofrece:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject; const Rect: TRect;  
    DataCol: Integer; Column: TColumn; State: TGridDrawState);
```

Sender: el objeto que genera el evento, como siempre

Rect: el rectángulo de pantalla que hay que dibujar. El tipo `TRect` es un registro variante, que podemos tratar como un conjunto de cuatro enteros (Left, Top, Right, Bottom) o dos `Tpoint` (TopLeft, BottomRight). Son las coordenadas de la esquina superior izquierda y de la esquina inferior derecha.

DataCol: el índice de la columna que se quiere dibujar.

Column: el objeto `Tcolumn` que representa la columna a dibujar.

State: el estado de la celda. Los estados posibles son:

- ◻ `gdSelected`: la celda está seleccionada (generalmente pintadas con el color
- ◻ `gdFocused`: la celda tiene el foco de atención del teclado
- ◻ `gdFixed`: la celda está en la zona fija de la grilla (la zona que no se mueve con las barras de desplazamiento, usualmente pintadas de gris)

Con estos datos es suficiente para graficar el interior de la celda. Uds. se preguntarán “¿y de donde sacamos los datos?”. Si no lo han hecho, háganlo ahora! La respuesta es: del campo asociado a esa columna. “¿Y cómo sabemos cuál es el campo, cómo accedemos a él?” Ayudados por el componente que representa esa columna: el parámetro *Column*.

⁵ Se mantiene un evento llamado `OnDrawDataCell` por compatibilidad, pero se recomienda el uso del nuevo `OnDrawColumnCell` en su lugar

Dijimos anteriormente que la clase Tcolumn tenía una propiedad que no se ve en el Inspector de Objetos que referencia al componente de campo correspondiente a la columna. Bien, ha llegado el momento de utilizarlo. Como todos los componentes de campo tiene las propiedades AsString, AsInteger, etc. -y es de ahí de donde obtendremos los datos.

Veamos un ejemplo: quiero mostrar el contenido de un campo memo en la grilla (por lo menos, lo que alcance con el ancho actual de la columna). Trabajando con la tabla de Agenda, tenemos el campo *Comentarios* de tipo memo.

Primero indicamos a la grilla que queremos dibujar nosotros su interior: la propiedad *DefaultDrawing* va a **False**. Ahora se generará el evento **OnDrawColumnCell**, en el que hacemos:

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject; const Rect: TRect;
  DataCol: Integer; Column: TColumn; State: TGridDrawState);
begin
  with dbGrid1 do begin
    if column.Field.DataType=ftMemo then
    begin
      if gdSelected in State then
        Canvas.brush.Color:= clHighlight //Color resaltado
      else
        Canvas.brush.Color:= Color; //Mismo Color que el Grid
      Canvas.Rectangle(rect.Left,rect.Top,rect.right,rect.bottom);
      Canvas.TextRect(rect,rect.Left+2,rect.Top+2,column.Field.asString)
    end else
      DefaultDrawColumnCell(rect,datacol,column,state);
    if gdFocused in State then
      Canvas.drawFocusRect(rect); //grafica el recuadro punteado
  end;
end;
```

El mayor problema de este programa es la extensión de los campos memo; difícilmente entre todo el texto en una sola línea. Sería bueno hacer más alta la línea según la cantidad de texto... pero es sumamente difícil, implica modificar el *componente* TDBGrid. Se puede encontrar el código fuente de un componente como ese en <http://www.marcocantu.com>.

Como ejemplo final, vamos a mostrar una imagen en la grilla. La forma más sencilla de hacerlo es creando un objeto Tbitmap temporal para mantener la imagen al sacarla de la tabla, y después dibujar esta imagen en la celda. Usaremos la famosa tabla Animals.dbf de los ejemplos de Delphi (alias DBDemos), donde tenemos el campo BMP con una foto del cada "animal".

```
procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject; const Rect: TRect;
  DataCol: Integer; Column: TColumn; State: TGridDrawState);
var
```

```

    g:tBitmap;
begin
    with dbGrid1 do begin
        if column.Field.FieldName='BMP' then
            begin
                if gdSelected in State then
                    Canvas.brush.Color:= clHighlight //Color resaltado
                else
                    Canvas.brush.Color:= Color; //Mismo Color que el Grid
                Canvas.Rectangle(rect.Left,rect.Top,rect.right,rect.bottom);
                g:= tBitmap.Create;
                g.Assign(column.field);
                Canvas.StretchDraw(rect,g);
                g.Free;
            end else
                DefaultDrawColumnCell(rect,datacol,column,state);
            if gdFocused in State then
                Canvas.drawFocusRect(rect);
            end;
        end;
    end;
end;

```

Nuevamente, los gráficos estarán muy deformados por la poca altura de las líneas.

Columnas con opciones

Otra de las opciones de que disponemos en las columnas de un DBGrid es la de especificar valores posibles para una columna; cada celda de esta columna se comportará efectivamente como un ComboBox (al entrar en edición).

Para esto sólo es necesario especificar los valores que se desplegarán en la lista, en la propiedad *PickList* de la columna deseada. Esta lista *no es excluyente*, lo que significa que se puede todavía escribir cualquier valor aunque no esté en la lista.

Una imagen vale más que 1001 palabras...

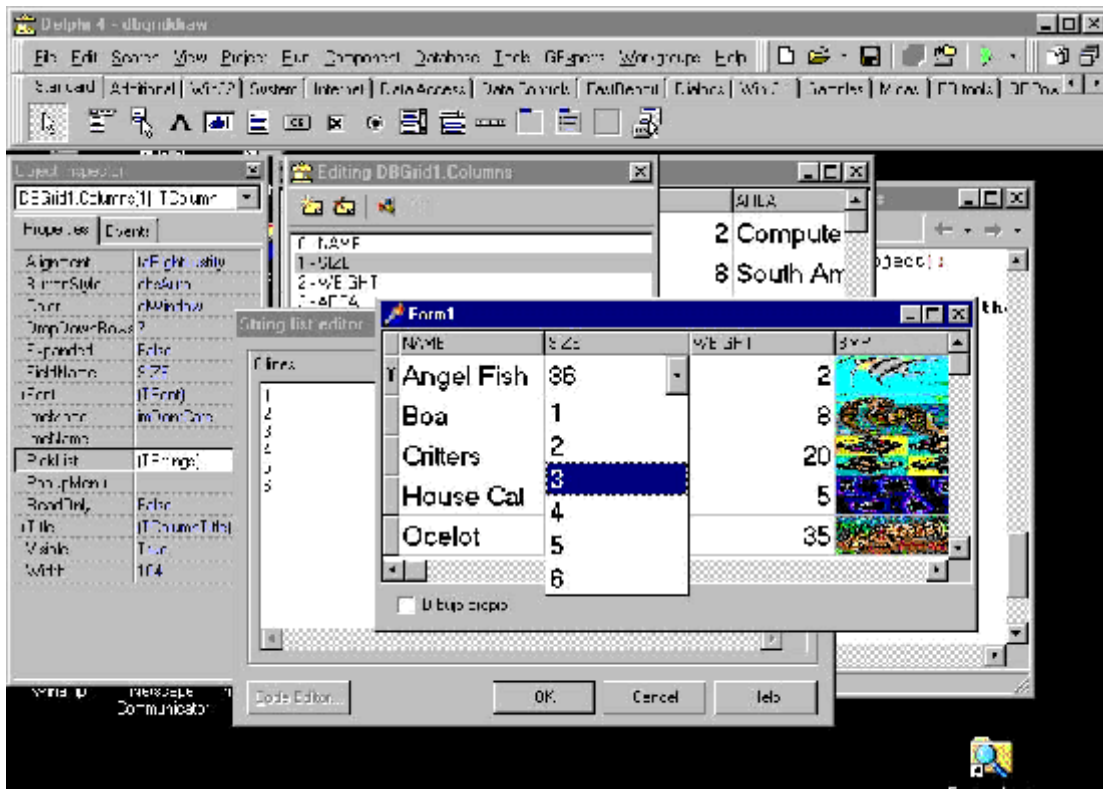


Figura 11: especificar una lista de valores en la propiedad PickList del campo SIZE, y su efecto en el programa final

Editores especiales

Todavía tenemos otra opción para las columnas de la grilla: si ponemos la propiedad *ButtonStyle* en **cbsEllipsis**, cuando entremos a editar ese campo aparecerá un botoncito en el extremo derecho de la celda. Recordaremos ese botón del Inspector de Objetos: allí nos permite invocar algún editor especial (como el de las listas). Aquí... nos genera un evento en la grilla.

El evento **OnEditButtonClick** se genera cada vez que presionamos ese botón. Aquí podemos abrir otra ventana, ir a otro lado, en fin... lo que nuestra imaginación nos permita. Como ya se me está acabando la imaginación y el lugar, haremos sólo un pequeño ejemplo que al presionar el botón pone la columna de un color elegido al azar entre cinco posibilidades:

```

procedure TForm1.DBGrid1EditButtonClick(Sender: TObject);
const
  colores: array[0..4] of tColor = (clRed,clBlue,clWindow,clGreen,clYellow);
begin
  dbgrid1.Columns[dbGrid1.SelectedIndex].Color:= colores[Round(random(5))];
end;

```

Alguna de las columnas debe tener la propiedad *ButtonStyle* en **cbsEllipsis** para que se llame alguna vez a este evento!



La grilla de controles

Hay aún otro control disponible para mostrar los datos de una tabla, una opción intermedia entre una grilla y una pantalla de controles. Se denomina *DBCtrlGrid* (*ControlGrid*, o *grilla de controles*).

La idea es tener la versatilidad de poder utilizar cualquier control de acceso a datos, pero mostrando más de un registro a la vez como en una grilla. En efecto, la grilla de controles es como una *matriz* de paneles, cada uno de los cuales puede contener cualquier control. Se muestra el contenido de un registro de la tabla en cada panel.

Crearemos una grilla de controles para la tabla *biolife.db*, de los ejemplos de Delphi. Coloquemos en una ficha nueva un componente *Table*, un *DataSource* y un *dbCtrlGrid*. A continuación enlazamos la fuente de datos con la tabla y ésta con el archivo. La grilla de controles se enlaza con la fuente de datos, igual que la grilla común, a través de la propiedad *DataSource*.

La grilla de controles tiene un solo panel activo (los demás aparecen rayados). Este es el panel que nosotros debemos armar con los controles de acceso a datos; una vez en funcionamiento, se repetirán los controles en los demás paneles -mostrando un registro diferente en cada uno, claro.

Ponemos entonces los controles para acceder a los datos en el panel activo. La ventana queda como en la fig. 13. A continuación, activamos la tabla. ¿Se ven los datos en los controles? ¿Se activan los paneles restantes?

Cuando ejecutamos el programa, vemos que los demás paneles se activan y cada uno muestra un registro diferente.

Algunas de las propiedades que nos servirán para manejar la grilla son las siguientes:

- C **AllowDelete**: cuando está en verdadero (True) se permite el borrado de registros con <Ctrl>+
- C **AllowInsert**: como se habrán imaginado, habilita o no la posibilidad de insertar registros utilizando <Alt>+<Ins>
- C **ColCount, RowCount**: cantidad de columnas y de filas, respectivamente. El tamaño total del control se divide en partes iguales para cada celda de la matriz resultante. Por lo tanto, si la altura de un panel no nos alcanza, no tenemos otra opción más que agrandar el control completo o disminuir la cantidad de filas.
- C **Orientation**: determina el orden de aparición de los registros en la grilla, según la siguiente figura:

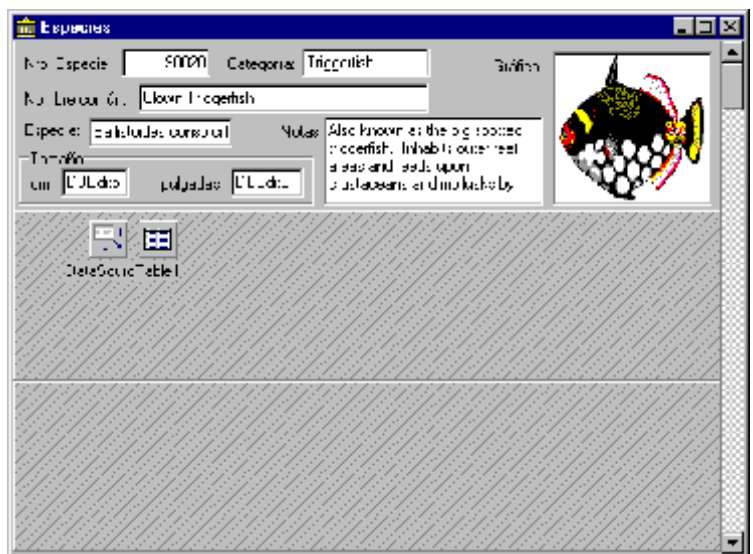


Figura 13: grilla de controles en tiempo de diseño

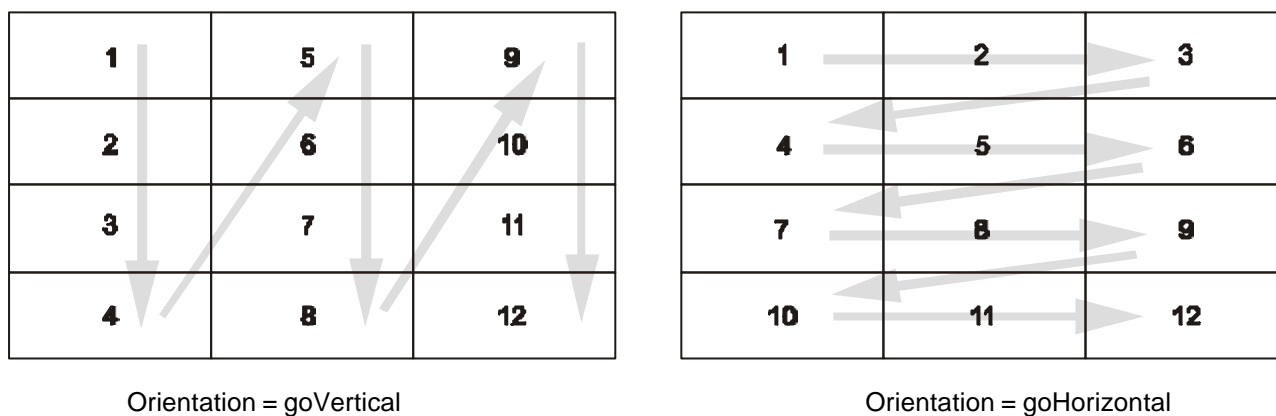


Figura 14: distribución de los registros en la grilla según el valor de la propiedad **Orientation**

- C **SelectedIndex:** indica el nro. de panel seleccionado (el que tiene el foco de edición). Asignando un valor a esta propiedad estamos efectivamente cambiando la posición del cursor en la tabla. Ver el ejemplo más abajo. *Esta propiedad no es visible en tiempo de diseño.*
- C **PanelCount:** número de paneles en el control. *No visible en tiempo de diseño.*

Ejemplo: vamos a hacer un programa que mueva el cursor de la tabla al registro que corresponda al panel sobre el que pasamos con el ratón. Entonces, si paso con el puntero sobre la celda marcada 1, el registro actual de la tabla deberá ser el registro que se muestra en ese panel.

Pondremos también una grilla común en la ventana para ver el efecto del cambio de registro. La ficha principal queda como en la figura 15.

Notemos primero el comportamiento por defecto de la grilla de controles; al ejecutar el programa vemos que se selecciona el primer panel (y en la grilla inferior vemos que el cursor se posiciona en el registro correspondiente). Para lograr nuestro cometido tenemos que responder al movimiento del ratón (evento **OnMouseMove**), y seleccionar entonces el panel sobre el que esté el puntero.

Para calcular el nro. de panel debemos tomar en cuenta la anchura y la altura de cada panel.

El procedimiento queda como se ve en el siguiente listado:

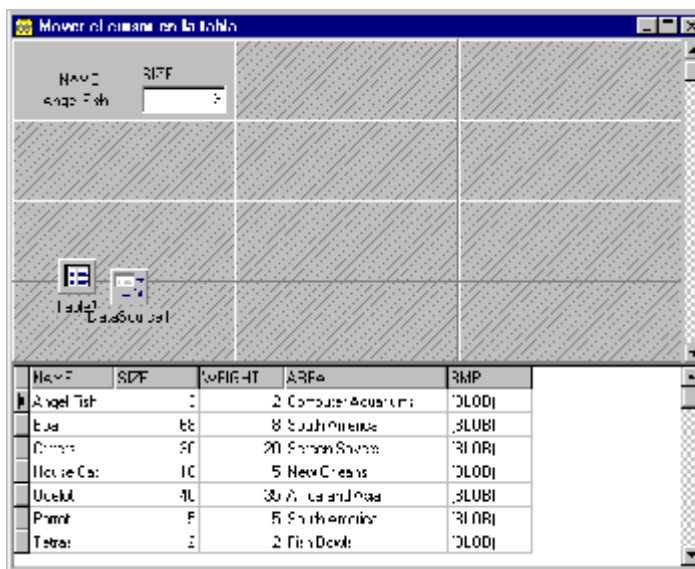


Figura 15: ficha del ejemplo

```

procedure TForm1.DBCtrlGrid1MouseMove(Sender: TObject; Shift: TShiftState;
  X, Y: Integer);
begin
  Caption:= IntToStr(x)+' ', '+IntToStr(y);
  if dbCtrlGrid1.Orientation=goHorizontal then

```

```
dbCtrlGrid1.PanelIndex:= y div dbCtrlGrid1.PanelHeight+
    (x div dbCtrlGrid1.PanelWidth)*dbCtrlGrid1.RowCount
else
    dbCtrlGrid1.PanelIndex:= (y div dbCtrlGrid1.PanelHeight)*dbCtrlGrid1.ColCount+
        x div dbCtrlGrid1.PanelWidth;
end;
```

Ahora si, cuando ejecutamos esta aplicación vemos que el cursor de la tabla se desplaza inmediatamente al registro correspondiente al panel sobre el que está el puntero del ratón. Podemos ver también que se *selecciona* este panel, lo que implica que será el que responda al teclado.

Acceso por programa a los datos de un campo

¿Cómo podemos cambiar *por programa* el contenido de un campo? ¿Puedo leer el contenido de un campo y colocarlo por ejemplo en el título de la ventana?

Uds. se estarán haciendo estas preguntas y otras por el estilo. Si la única manera de acceder a los datos de una tabla es con los controles de datos, entonces la utilidad de su existencia se vuelve una restricción. Por supuesto, éste no es el caso.

El acceso a los datos implica la utilización de componentes especiales diseñados para representar un campo. Estos componentes son descendientes de una clase abstracta llamada Tfield, y hay una clase especializada para cada tipo de dato diferente que pueda tener un campo. Les llamaremos en forma general *componentes de campo*.

Estos componentes proveen los métodos y propiedades necesarios para

- C Recuperar el valor de un campo de una tabla
- C Cambiar el valor de un campo de una tabla
- C Convertir el valor de un campo de un tipo de datos a otro
- C Validar el dato ingresado
- C Determinar el aspecto del campo cuando se mira y cuando se edita
- C Mostrar el contenido del campo como el resultado de algún cálculo (campos calculados)
- C Buscar el contenido del campo desde otra tabla (campos *lookup*)

Veremos todas estas características en las siguientes secciones.

En la siguiente tabla se pueden ver las distintas clases, junto con el tipo de dato correspondiente en Paradox y Delphi:

Clase	Tipo de campo en Paradox	Tipo de datos
tField		
tStringField	Alpha (A)	String
tIntegerField	LongInteger (I)	Entero entre -2.147.483.648 y 2.147.483.647 (32 bits - longint)
tSmallIntField	Short (S)	Entero entre -32.768 y 32.767 (16 bits - integer)
tWordField	Puede ser I ó S	Entero entre 0 y 65.535 (16 bits - word)
tFloatField	Number (N)	Número real entre $5.0 * 10^{-324}$ y $1.7 * 10^{308}$
tAutoincField	Autoincrement (+)	Número entero secuencial, automático

Clase	Tipo de campo en Paradox	Tipo de datos
tCurrencyField	Money (\$)	Número real. Internamente se trabaja en 6 dígitos, independientemente de la cantidad de decimales mostrados. Por defecto se muestra con 2 decimales y el signo monetario.
tBCDField	BCD (#)	Número en formato BCD (Binary Coded Decimal). Sirve para compatibilidad con otras aplicaciones que usen este formato ⁶ .
tDateTimeField	Timestamp (@)	Fecha y hora, juntos en un tipo tDateTime.
tDateField	Date (D)	Fecha.
tTimeField	Time (T)	Hora.
tBooleanField	Logical (L)	Valor True o False.
tBytesField	Bytes (Y)	Campo de bytes, que puede contener hasta 255 bytes con cualquier información.
tVarBytesField		Un conjunto de hasta 65535 bytes con datos arbitrarios. Los primeros dos indican la longitud.
tBlobField	Binary (B) Formatted Memo (F) OLE (O)	Un conjunto de bytes de longitud arbitraria, por ejemplo para contener un sonido o un objeto OLE.
tMemoField	Memo (M)	Igual que el tipo tBlobField, pero los bytes deberían contener texto.
tGraphicField	Graphic (G)	Igual que el tipo tBlobField, pero los bytes deberían contener un gráfico.
TADTField		Tipo de datos definido por el usuario (Abstract Data Type). No disponible en Paradox.
TarrayField		Vector de datos, ya sea comunes o definidos por el usuario. No disponible en Paradox.
TreferenceField		Referencia a otra tabla. No disponible en Paradox.
TaggregateField		Funciones de agregación sobre grupos de registros; no disponible en Paradox.

En la clase tField tenemos definidas algunas propiedades que nos permiten acceder al dato del campo

⁶ Internamente Delphi usa el tipo Currency para los campos BCD, por lo que la precisión se restringe a 20 dígitos significativos con 4 decimales

correspondiente haciendo las conversiones de tipo necesarias automáticamente. Estas propiedades nos permiten acceder al contenido del campo al que está enlazado el componente, aunque el tipo de dato que pedimos no sea el correcto (mientras se pueda hacer la conversión). Por ejemplo, podemos pedir una fecha de un campo tipo Date como String, o ingresar un valor numérico como un string. Las conversiones necesarias se realizan automáticamente, y en caso de no ser posibles se genera una excepción.

Ahora bien, ¿Cómo se crean estos componentes? Hay dos maneras: llamando al método **FieldByName** que poseen los Datasets (los componentes Table o Query, entre otros), o creando componentes en tiempo de diseño, de los cuales podremos ver sus propiedades y eventos en el Inspector de Objetos. Cada método tiene sus ventajas y desventajas, que trataremos a su debido tiempo.

Veamos un ejemplo concreto. Haremos que al presionar un botón se tome el contenido del campo “Nombre” de la tabla NOMBRES.DB que creamos anteriormente, y se muestre ese valor en el título del form. Para esto crearemos un form con un componente Ttable, y enlazaremos este componente al archivo en disco. Coloquemos también un DataSource, una grilla y un navegador para poder ver los datos y movernos por los registros. Enlazamos todo como ya sabemos, y agregamos un botón para tomar el dato y ponerlo en la propiedad *Caption* del form.

El evento **OnClick** del botón deberá ser algo como esto:

```
procedure TForm1.Button1Click(sender:TObject);
begin
    caption:= Table1.FieldByName('Nombre').AsString;
end;
```

Como vemos, utilizamos el método FieldByName pasándole como parámetro el nombre del campo. Esta función nos devuelve un componente de campo (un descendiente de Tfield, en el caso de este campo será un TStringField porque el campo es de tipo Alfanumérico). Ahora bien, estos componentes tienen propiedades que nos permiten acceder al dato que han extraído del archivo: todas estas propiedades tienen nombres que comienzan con “As...”, de tal manera que nos indican qué tipo de datos aceptan (si se les asigna un valor) o devuelven (si se leen). En el ejemplo anterior necesitábamos el dato como un string, por lo que lo pedimos leyendo la propiedad **AsString** del componente creado por la llamada a **FieldByName**. Dado que en este caso el campo era de tipo alfanumérico, no fue necesaria ninguna conversión.

Agreguemos ahora otro botón para poner en la barra de título no el campo “Nombre” sino el campo “Edad”. Este campo es de tipo Entero, por lo que el componente creado será una instancia de TIntegerField.

```
procedure TForm1.Button2Click(sender:TObject);
begin
    caption:= Table1.FieldByName('Edad').AsString;
end;
```

Cuando corremos el programa, podemos ver que se muestra la edad del registro actual correctamente en la barra de título del form. Esta vez, fue necesaria una conversión de número entero a string que fue realizada internamente.

De la misma manera, podemos por ejemplo asignar un nuevo valor al campo “Edad” del registro actual tomándolo directamente de un editor (conversión de string a entero).

Ejercicio 2-8

Agregar al form del ejemplo anterior un editor y otro botón. Al presionar este nuevo botón se debe tomar el número escrito en el editor y ponerlo en el campo “Edad”.

Como siempre, antes de modificar el valor de un registro debemos poner la tabla en modo de edición; y después tenemos que aceptar (Post) o cancelar (Cancel) los cambios realizados.

Las conversiones posibles se resumen en el siguiente cuadro:

Tipo	AsString	AsInteger	AsFloat / AsCurrency	AsDateTime	AsBoolean
tStringField		Convierte a integer si es posible	Convierte a Float si es posible	Convierte a DateTime si es posible	Convierte a Boolean si es posible
tIntegerField tSmallIntField tWordField	Convierte a String		Convierte a Float	No permitido	No permitido
tFloatField tCurrencyField tBCDField	Convierte a String	Redondea al entero más cercano		No permitido	No permitido
tDateTimeField tDateField tTimeField	Convierte a String	No permitido	Convierte la fecha a la cantidad de días desde el 01/01/0001. Convierte la hora a una fracción de 24 hs.	Si no se especifica la hora o la fecha valen cero.	No permitido
tBooleanField	Convierte a string “True” o “False”	No permitido	No permitido	No permitido	
tBytesField tVarBytesField tBlobField tMemoField tGraphicField	Convierte a string (generalmente sólo tiene sentido para los Memo)	No permitido	No permitido	No permitido	No permitido

NOTAS:

- ⊞ También hay una propiedad llamada **AsVariant**, que funciona para todos los tipos de datos listados; como su nombre lo indica, trabaja con variables tipo **Variant**.
- ⊞ El tipo **Currency** es igual al **Float** para la conversión, pero de precisión fija.

La función **FieldByName** crea un componente de campo y devuelve un puntero a él como resultado. No obstante, hay otra forma de crear los componentes de campo: como propiedades de la ficha en donde está la tabla.

Creación de campos persistentes

Para crear los componentes de campo en tiempo de diseño (que llamaremos desde ahora “campos persistentes” o “componentes de campo persistentes”) y agregarlos como propiedades de la ficha que contiene la tabla, debemos hacer doble click en el icono de la tabla. Aparece la ventana del **Editor de campos**, que se ve en la figura 16.

El navegador del editor de campos nos permite navegar por la tabla en tiempo de diseño.



Figura 16: El editor de campos

¿Y dónde está el piloto... es decir, los comandos para crear los componentes de campo? Ocultos en un *menú contextual*, que traemos a la vista como siempre con el botón derecho del ratón. Tenemos un comando para crear un componente para cada campo: **Add all fields**. Al invocarlo se crean todos los componentes como propiedades de la ficha actual.

Cada componente de campo tiene propiedades y eventos, que ahora podemos ver en el Inspector de Objetos cuando seleccionamos uno en el editor. Notemos en especial la propiedad **Name**, que como de costumbre nos indica el nombre de la variable (propiedad, en este caso) creada para referenciar al componente. Por defecto se asignan nombres compuestos por el nombre de la tabla (por ejemplo, **Table1**) seguidos del nombre del campo sin espacios ni caracteres especiales. Por ejemplo, el campo “Nro factura” se referencia con el componente **Table1NroFactura**.

Luego veremos otras propiedades de estos componentes; por ahora, presentamos algunos ejemplos para aclarar un poco las tormentosas nubes que nos tapan el sol de la mañana.

Ejemplos

- ⊞ Queremos poner el valor “01/04/1996” en el campo “Fecha De Nacimiento” del registro actual.

Suponiendo que creamos el objeto correspondiente al campo Fecha de Nacimiento y se llama **Table1FechaDeNacimiento**, podemos hacer

```
Table1FechaDeNacimiento.AsString:= '01/04/1996';
```

- ⊞ De la misma manera, para tomar el valor de la fecha y ponerlo en un string **s**, hacemos

```
s:= Table1FechaDeNacimiento.AsString;
```

C Si queremos por ejemplo colocar la fecha actual en el campo Fecha de la factura:

```
Table1Fecha.AsDateTime:= Now;
```

Notemos que no hay propiedades **AsDate** o **AsTime**; simplemente, los componentes se limitan a tomar en cuenta la parte que les interesa.

C Para asignar un valor a un campo memo podemos usar la propiedad AsString:

```
Table1Comentarios.AsString:= 'Esto es un comentario';
```

o también asignarle las líneas de un memo común:

```
Table1Comentarios.Assign(Mem1.Lines);
```

Los campos de tipo “Blob” (Binary Large Objects) como ser el memo, graphic o binary poseen los métodos “LoadFromFile” y “SaveToFile” para trabajar con archivos. También poseen un método “Clear” para limpiar el campo correspondiente.

Por ejemplo⁷, para cargar un gráfico en un campo de tipo Graphic, podemos una vez creado el componente de campo correspondiente utilizar sus métodos. Recordemos en el ejemplo “BiblioDB” anterior, que trabajamos con el método *LoadFromFile* pero del **DBImage.Picture**. Es decir, si tuviéramos que cargar un gráfico en un campo pero sin mostrarlo en pantalla, tendríamos que usar igualmente un componente DBImage pero no visible. Una solución más sencilla y “natural” es utilizar un componente de campo para hacer la tarea. Comparemos las dos soluciones:

En BiblioDB:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then begin
    Table1.Edit;
    DBImage1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
  end;
end;
```

Con componentes de campo:

⁷ El ejemplo completo (Agenda Multimedia) se lista al final.


```

procedure TForm2.Button5Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then begin
    Table1.Edit;
    Table1Foto.LoadFromFile(OpenPictureDialog1.FileName);
  end;
end;

```

Vemos que ahora el método llamado al presionar el botón de “Cargar” corresponde al componente de campo (Table1Foto) en lugar del DBImage1.

Lo mismo sucede con el botón de borrar la imagen. Lo único necesario es llamar al método “Clear” del componente de campo:

```

procedure TForm2.Button6Click(Sender: TObject);
begin
  Table1.Edit;
  Table1Foto.Clear;
end;

```

El ejemplo completo de la Agenda Multimedia es muy largo y complejo para mostrarlo aquí; por lo tanto posponemos el listado completo al final del capítulo. En este programa se ve además la utilización por programa del componente MediaPlayer, que tiene también sus “vueltas”.

La propiedad Fields

Los campos de la tabla se mantienen en una propiedad de tTable llamada **Fields**. Es un array de objetos tipo tField o descendientes. De aquí que podemos acceder a un determinado campo con el índice en el array, por ejemplo para leer el contenido del tercer campo como un string haríamos

```
Table1.Fields[2].AsString:= 'Algun valor';
```

Notemos que el orden de los campos está determinado en la estructura de la tabla; si luego cambiamos la estructura desde el Database Desktop por ejemplo, la sentencia anterior no cambia y puede modificar un campo no deseado.

Determinar el aspecto de los datos

Los componentes de campo tienen propiedades que afectan al aspecto de los datos al ser presentados en pantalla. Las características de presentación que podemos controlar dependen del tipo de datos que contiene el campo.

Damos a continuación una tabla de las propiedades utilizadas para controlar la presentación de los datos,

indicando cuando sea necesario si son específicas para un tipo de dato:

Propiedad	Tipo de campo	Se utiliza para	Ejemplos
Alignment	Todos	Alineación horizontal	Para los números usualmente se utiliza alRight (a la derecha)
DisplayFormat	Numéricos, fecha, hora	Formato de presentación cuando no está en edición	#.00 muestra un dígito (si existe) a la izquierda del punto; si no existe un dígito no pone nada, y dos dígitos a la derecha del punto, completando con ceros. dd/mm/yyyy dos dígitos para el día/dos dígitos para el mes/cuatro dígitos para el año hh:mm hora con dos dígitos, dos puntos, minutos con dos dígitos
DisplayLabel	Todos	Título de columna en grillas*	
DisplayWidth	Todos	Ancho de columna en grillas*	
EditMask	Fecha, hora, texto	Máscara para edición	99/99/0000 formato para fecha, dos números opcionales / dos números opcionales / cuatro números requeridos
EditFormat	Numéricos	Formato para edición	##;"("#)";0!" Los nros. Positivos se muestran normalmente, los negativos entre paréntesis y el cero con un signo de admiración detrás
Visible	Todos	Determina si el campo se ve o no en una grilla*	

* Si se definen columnas para la grilla, estas propiedades no tienen efecto. No obstante, si los componentes de campo se crean *antes* que las columnas, éstas toman los valores de las propiedades de los componentes.

Los símbolos usados para las máscaras son diferentes para cada propiedad, desgraciadamente; y también son diferentes de las máscaras de edición de Paradox. Se puede ver la lista completa en la ayuda.

Campos virtuales

Todos los componentes de campo con los que hemos trabajado hasta el momento se refieren a un campo real de una tabla física. Pero tenemos otras posibilidades: podemos crear *campos virtuales*, que no se corresponden con un campo de la tabla. Estos campos pueden representar por ejemplo un cálculo entre otros campos o mostrar valores de una tabla mientras trabajamos con otra.

Los *campos calculados* son aquellos en los que mostramos el resultado de un cálculo. Por lo tanto, *no son editables*. En los otros aspectos, son iguales a los campos reales: tienen las mismas propiedades, métodos y eventos. Su contenido debe ser calculado en el programa, en un evento especial *de la tabla* que se denomina **OnCalcFields**.

Este evento se produce cada vez que es necesario cambiar el contenido de los campos calculados, *siempre que la propiedad AutoCalcFields de la tabla esté en TRUE*. Caso contrario, tendremos que activarlo nosotros mismos. Esto puede servir cuando hay muchos cambios a los datos, ya que por cada cambio se llama automáticamente este evento.

En este evento debemos calcular *todos* los valores necesarios, con cuidado de no modificar los campos comunes: esto provocaría que se llame nuevamente al evento, entrando en un bucle recursivo con consecuencias generalmente desastrosas para la aplicación.

Vamos ya a un ejemplo. El señor Desparicio Temprano, dueño de la empresa Desparicio-nes Co. quiere facturar con la computadora, que para eso se la ha comprado pues. El programa debe cumplir los siguientes requerimientos:

- C Se desea saber el Nro. de factura, el Tipo (A, B o C), la fecha y el cliente
- C Cada factura no puede tener más de cuatro ítems (líneas)
- C Por cada ítem hay que guardar la Cantidad, Descripción y Precio Unitario.
- C Por cada ítem hay que calcular el subtotal, y por cada factura el total general. Estos cálculos deben ser dinámicos, es decir, actualizarse automáticamente cuando el usuario cambia algún valor.

Dados estos requerimientos, podemos plantear la siguiente estructura para la tabla FACTURAS1.DB:

Nombre	Tipo	Longitud	Comentarios
Nro factura	I		Clave primaria
Tipo	A	1	Mínimo 'A', máximo 'C', por defecto 'C'
Fecha	D		Por defecto: fecha de creación de la factura
Cliente	A	30	
Cant1	I		Por defecto: 1
Desc1	A	40	
PU1	\$		
Cant2	I		
Desc2	A	40	
PU2	\$		
Cant3	I		
Desc3	A	40	
PU3	\$		
Cant4	I		

Nombre	Tipo	Longitud	Comentarios
Desc4	A	40	
PU4	\$		

Ahora creamos en Delphi el formulario de Entrada, Modificación y Baja de datos (ABM), como el de la figura 17.

En esta ficha podemos ver varios componentes de campo. Hay un editor para cada campo, no creo que tengan problemas en localizar cada uno. Ahora bien, ¿a qué campo están enlazados los componentes DBText que aparecen a la derecha?

Estos componentes están ahí para mostrar el subtotal de cada línea. Como habrán adivinado, para cada uno vamos a definir un campo calculado.

Para definir los campos calculados debemos entrar al editor de campos, haciendo doble click en el icono de la tabla. A continuación invocamos el menú contextual y seleccionamos la opción “New Field”, que nos trae a la pantalla el diálogo de creación de nuevo campo (fig. 18). En esta ventana definimos las características de nuestro campo virtual.

En la Fig. 18 vemos los datos correspondientes al primero de nuestros campos calculados, el Subtotal para la línea 1. Lo llamamos **ST1**⁸ (el componente se llama **Table1ST1**) y especificamos el tipo de datos que trabaja (Currency) y el tipo de campo (Calculated, calculado). Los demás datos no se aplican a este caso.

Bueno, pues manos a la tecla! La creación de los otros tres campos calculados ST2, ST3 y ST4 queda para Uds. Es indispensable que lo hagan antes de continuar.

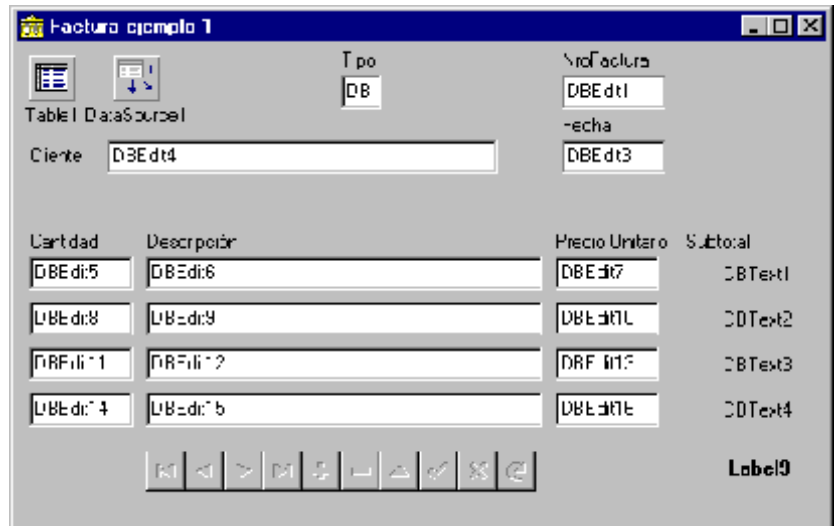


Figura 17: El formulario de ABM de la tabla Facturas1.db

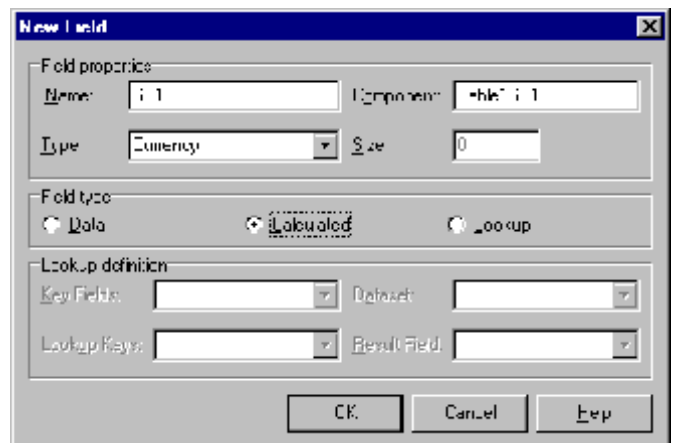


Figura 18: Cuadro de diálogo para definir un nuevo campo calculado

Ahora si, vamos a dar valor a estos campos. Para ello, disponemos como dijimos arriba de un evento en el componente Ttable llamado **OnCalcFields**. En él debemos asignar un valor a cada campo calculado - inclusive podemos asignar valores a otros componentes o hacer otros cálculos, siempre que no modifiquemos

⁸ Un alarde de imaginación y creatividad!

un componente no calculado.

En nuestro caso, se puede calcular el valor correspondiente a ST1 como el valor del campo Cant1 multiplicado por el valor del campo PU1:

```
Table1ST1.AsCurrency:= Table1Cant1.AsInteger * Table1PU1.AsCurrency;
```

Lo mismo para los demás, por supuesto. Más aún: después de calcular todos los subtotales, podemos aquí mismo calcular el total general sumando todo. El objetivo del **Label9** que veíamos en la ficha es justamente mostrar ese total general. La última línea entonces del evento OnCalcFields es

```
Label9.Caption:= FloatToStr(Table1ST1.AsCurrency + Table1ST2.AsCurrency +  
    Table1ST3.AsCurrency + Table1ST4.AsCurrency);
```

Queda para el afanoso lector la tarea de completar este evento y probar la aplicación.

Campos y componentes de búsqueda (Lookup)

Ahora bien, el señor Desparicio Temprano no es capaz de escribir un nombre dos veces igual; así, nos encontramos con varias facturas con clientes como “Perez”, “Pérez”, “Peres”, etc. Todos distintos a la hora de buscar por cliente. Por lo tanto, decidimos guardar los datos de los clientes en otra tabla (y aprovecharemos para guardar algo más que el nombre) como nuestra ya famosa agenda. Y en la factura, únicamente dejar al usuario que elija uno de los nombres que ya están ingresados. Para esto tenemos disponible un tipo de campo especial: el tipo *Lookup* o campo de búsqueda.

Básicamente, los campos de búsqueda son como los políticos: muestran algo que no es lo que contienen. Se enlazan con *dos* tablas, una de donde sacan los datos para mostrar (en una lista) y la otra es donde dejan el valor correspondiente al dato elegido por el usuario.

Estos campos son principalmente para su uso con una grilla. Cuando entramos a editar un campo de este tipo en una grilla la celda se transforma en un ComboBox, que muestra los datos de la tabla secundaria.

Cuando no usamos una grilla, tenemos dos componentes especiales para la misma tarea: el DBLookupComboBox y el DBLookupListBox. Los llamaremos genéricamente *Controles de búsqueda*.

En todos los casos, necesitamos dos tablas. Una de ellas será la depositaria de la elección del usuario, mientras que la otra proveerá los datos para mostrar como opciones entre las cuales elegir. Luego, el funcionamiento es sencillo: cada vez que el valor del campo principal cambia, el componente busca el nuevo valor en la tabla secundaria (por un campo declarado como clave para la búsqueda) y muestra el valor de otro campo de la tabla secundaria en su lugar. Así por ejemplo podemos tener en la tabla principal los números de DNI y ver en cambio los nombres.

Utilizaremos en el ejemplo de la factura un componente DBLookupComboBox para traer el nombre del cliente desde la tabla de agenda creada anteriormente.

Hay varias propiedades a tener en cuenta cuando usamos un componente DBLookupComboBox:

- C DataSource, DataField: como siempre, indican el destino de los datos que ingresemos (tabla primaria).
- C ListSource: el nombre de la fuente de datos (componente DataSource) que proveerá el contenido de la lista de opciones (tabla secundaria).

- C **KeyField**: el nombre del campo de la tabla secundaria (aquella referenciada por la fuente de datos *ListSource*) en donde se buscará una coincidencia con el campo principal (*DataField*). El valor de este campo es el que se guarda en la tabla principal.
- C **ListField**: el campo de la tabla secundaria que aparecerá en la lista. Se puede especificar una lista de campos (en la lista aparecerán varias columnas), que entonces debemos escribir a mano separando cada uno con ; (punto y coma).
- C **ListFieldIndex**: en caso que utilicemos más de un campo en la propiedad *ListField*, debemos seleccionar *uno* para que aparezca en la parte de texto del *ComboBox*. Además, en las búsquedas incrementales se utilizará este campo. La propiedad *ListFieldIndex* es el número de campo de los listados en *ListField*.

En nuestro ejemplo, vamos a mostrar en la lista del *ComboBox* los campos “DNI”, “Apellido y Nombre” y “Dirección” para tener una buena idea de quién es nuestro cliente. De estos campos, mostraremos el apellido y nombre. El campo de la tabla principal es el campo “Cliente”, que enlazaremos con el campo “Apellido y Nombre” de la tabla secundaria. En este caso, los campos de búsqueda y de resultado son del mismo tipo (ambos son cadenas de caracteres), pero no es obligatorio que sea así.

Bueno, basta de cháchara. Las propiedades que tenemos que especificar en el componente *DBLookupComboBox* son entonces:

DataSource: la fuente de datos principal (enlazada con la tabla Facturas1.db)

DataField: Cliente

ListSource: la fuente de datos secundaria, enlazada con la tabla Agenda.db.

KeyField: Nombre y Apellido

ListField: DNI;Apellido y nombre;Dirección

ListFieldIndex: 1

Ahora al correr el programa, el usuario no puede escribir en el *ComboBox*, sólo seleccionar un dato de la lista (Fig. 19)

NOTA: es posible que la lista sea muy angosta para mostrar todas las columnas; por defecto toma el ancho del *ComboBox*. Podemos especificar el tamaño de la lista con las propiedades *DropDownRows* (cantidad de líneas a mostrar en la lista) y *DropDownWidth* (ancho total de la lista).

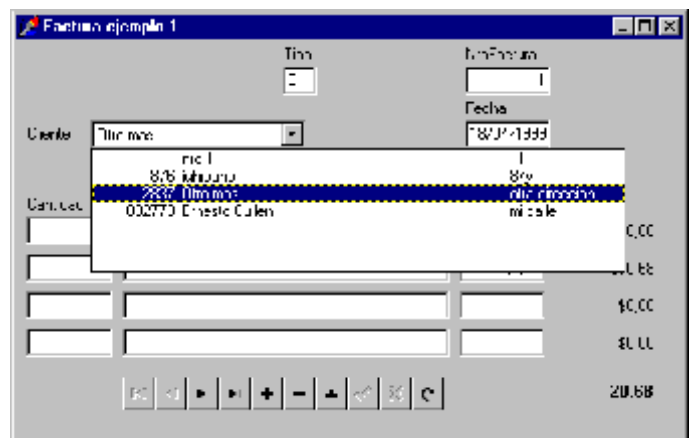


Figura 19: El usuario no puede escribir el nombre del cliente directamente, debe seleccionarlo en la lista

Claro que el no poder escribir un nuevo nombre puede ser un poco engorroso; normalmente pondremos también un botón para permitir el ingreso de los datos de un nuevo cliente. En el programa *Facturas1* completo se puede ver la técnica.

Para definir un *campo* de búsqueda, debemos seguir los mismos pasos que para definir un campo calculado: esto es, en el Editor de Campos pedimos la creación de un nuevo campo con el menú contextual, le damos nombre, tipo de dato y longitud. Pero esta vez será un campo de tipo *Lookup*, con lo que se nos habilitan los controles de la parte inferior del cuadro de diálogo. Estas son las propiedades equivalentes a aquellas que tuvimos que cambiar para el *DBLookupComboBox*:

	Control de búsqueda	Campo de búsqueda
Tabla principal	DataSource	El dataset sobre el que está definido
Campo principal	DataField	El mismo
Tabla secundaria	ListSource	LookupDataSet
Campo de búsqueda	KeyField	LookupKeyFields
Campo de resultado	ListField	LookupResultFields

La definición del campo BusCliente quedaría como se ve en la figura 20.

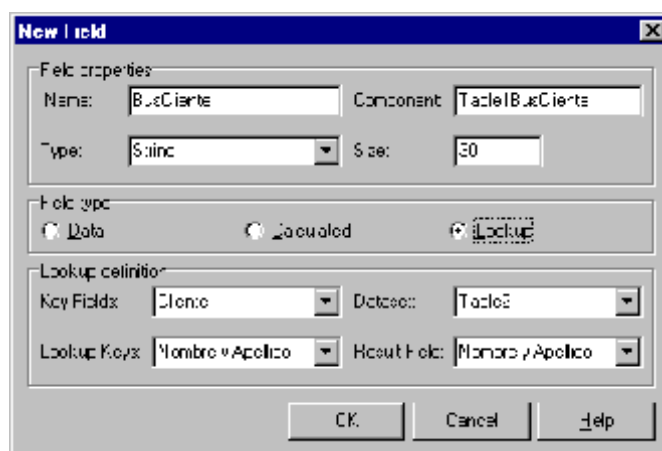


Figura 20: definición de un campo de búsqueda

NOTA: por más que el campo de búsqueda acepta la especificación de múltiples campos en la lista, no tenemos una forma de seleccionar *cuál* de ellos se devolverá como resultado (lo que en los controles de campo era función de la propiedad *ListFieldIndex*). Por lo tanto, si especificamos múltiples campos como *LookupResultField* nos encontraremos con un bonito mensaje de error.

<<<Modificar el ejemplo para que también utilice el campo Lookup en una grilla>>>

Haremos un ejemplo completo con campos calculados y campos de búsqueda después de ver tablas relacionadas.

Cambiar el índice activo

Las tablas utilizan por defecto el índice principal, formado por los campos de la clave principal. Pero muchas veces será necesario utilizar otro índice distinto, lo que Paradox llama *índice secundario*. Recordemos que el índice determina el orden en que se muestran los registros, y también permiten una búsqueda rápida.

Sólo un índice puede estar activo a la vez. El nombre de este índice aparece en la propiedad **IndexName** del componente de tabla (los componentes Query no tienen esta propiedad porque trabajan con una tabla virtual). Si esta propiedad está en blanco, se utiliza el índice principal.

Supongamos que queremos ver las facturas que hemos ingresado, pero ordenadas por tipo para que aparezcan todas las de tipo 'A', luego las 'B' y luego... ¿sabrá el lector dilucidar la letra que viene después?

Creamos entonces un índice secundario sobre la tabla Facturas1, con el campo Tipo como único elemento. Lo guardamos como "iTipo". Ahora en Delphi, creamos una nueva aplicación y colocamos los consabidos Table, DataSource y DBGrid. Enlacemos todo y activemos la tabla, para ver inmediatamente los datos. ¿Cuál es el orden de los registros? ¿Por qué? ¿Cuál es el índice activo?

Exacto, el índice activo es el principal; por lo tanto, los registros aparecen ordenados por "Nro factura". Para cambiar el índice podemos abrir la lista de la propiedad **IndexName** del componente Table (Fig. 21). Aparecerá el nuevo índice que hemos definido: "iTipo". Seleccionamos este, y ¡voilà! los datos se reordenan automáticamente en la grilla. Fácil, no?

No todo es color de rosa... si no, sería muy aburrida la vida! Recordemos que el uso de los índices secundarios se debe pensar con cuidado.

Para volver al índice principal, simplemente asignamos la cadena vacía a la propiedad **IndexName** (o borramos la casilla correspondiente en el Inspector de Objetos).



Figura 21: Selección de un nuevo índice activo

¡Busque, Sultán, busque!

Una de las operaciones básicas a realizar sobre cualquier tabla es la búsqueda. Delphi nos ofrece varios caminos posibles:

C El método **FindKey** (componente TTable)

Busca la primera ocurrencia exacta de los valores buscados. Los campos sobre los que se trabaja deben ser los primeros en el índice activo. Es una función que devuelve un valor lógico *verdadero* si se encuentra un registro coincidente (y el cursor en la tabla queda posicionado sobre ese registro) o *falso* en otro caso (y el cursor no se mueve).

Esta función espera como parámetro un *array* de valores. Esto es, una lista con los valores a buscar. Estos valores deben estar especificados en el orden correspondiente según el índice activo.

Por ejemplo, si el índice está formado por los campos "Fecha" y "NumCliente", podemos buscar un registro particular con la siguiente instrucción:

```
FindKey(['10/10/1998',8])
```

que nos devuelve un valor lógico verdadero si lo encuentra -y posiciona la tabla en ese registro- o bien

devuelve falso y deja la tabla en la posición que estaba.

Notemos que al momento de especificar los valores, si es necesaria alguna conversión Delphi la hace automáticamente.

C El método **FindNearest** (tTable)

Es similar al anterior, sólo que no devuelve nada (es un procedimiento). Si el registro buscado se encuentra, se posiciona el cursor de la tabla sobre el mismo. Caso contrario, se ubica el cursor en el lugar *donde debería estar* el registro si existiera.

Este procedimiento espera como parámetro también una lista de valores, igual que **FindKey**.

Ambos métodos trabajan con el índice activo; sólo se puede buscar con estos métodos en los campos del índice. Y en el orden que aparecen estos campos en el índice: por ejemplo, si en el índice tenemos primero el campo "NumCliente" y luego el campo "Fecha" entonces los valores se deben listar en ese orden en la llamada al método.

Esto indica la necesidad de definir índices secundarios por cada campo o combinación de campos que vayamos a utilizar para búsqueda; teniendo en cuenta que los índices secundarios deben ser automantenidos y que son generalmente muy grandes -a veces más que la tabla misma, vemos que hay que buscar una solución de compromiso entre la sobrecarga del sistema con muchos índices y las posibilidades de búsqueda.

También hay que tener en cuenta que con estos métodos se encuentra *el primer* registro que cumpla los requisitos; si hay más de uno, debemos usar otro método.

Estas consideraciones nos llevan a otros métodos de búsqueda, tal vez más flexibles pero más lentos, que trataremos más adelante.

Veamos algunos ejemplos.

Ejemplo 1

Buscar en la tabla de nombres un registro con el nombre 'Eduardo Pérez'

```
if Table1.FindKey(['Eduardo Pérez']) then
    ShowMessage('Encontrado!')
else
    ShowMessage('No encontrado');
```

Ejemplo 2

Buscar en la tabla de nombres un registro que empiece con 'E' y mostrar el nombre encontrado.

```
Table1.FindNearest(['E']);
```

Si no se encuentra un registro que empiece con 'E', el cursor se ubicará en la posición donde debería estar tal registro.

Ejemplo 3

Buscar en la tabla de nombres una persona de nombre 'Eduardo Pérez' y dirección 'Almafuerte 9287'. La tabla se llama TableNombres y los objetos de campo están creados con los nombres TableNombresNombreYApellido y TableNombresDireccion respectivamente.

```
If Table1.FindKey(['Eduardo Pérez','Almafuerte 9287']) then
  ShowMessage('Encontrado!')
else
  ShowMessage('No encontrado');
```

Ejemplo 4

Buscar en la tabla de nombres el primer registro que tenga fecha de nacimiento posterior al 1/1/1960. La tabla se llama Table1 y el índice secundario por fecha de nacimiento es 'iFecha';

```
Table1.IndexName:= 'iFecha';
table1.FindNearest(['1/1/1960']);
```

Ahora la tabla queda posicionada en el registro buscado o en el lugar donde debería estar.

Ejemplo 5

Hacer un programa con búsqueda secuencial; es decir, a medida que se va escribiendo un nombre en un editor se debe ir buscando en la tabla. Para ello escribimos en el evento OnChange del editor lo siguiente:

```
procedure TForm1.Edit1Change(sender:TObject);
begin
  Table1.FindNearest([edit1.text]);
end;
```

De esta manera, cada vez que cambia el contenido del editor se busca nuevamente una coincidencia (no exacta) en la tabla. Esto nos permite encontrar rápidamente lo que buscamos, escribiendo generalmente sólo las primeras letras.

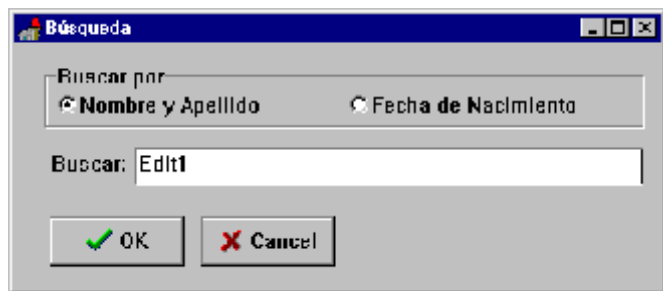
¿Qué pasaría si usamos **FindKey** en lugar de **FindNearest** en el ejemplo 5?

Ejercicio 2-9

Crear una aplicación que muestre los datos de la tabla de nombres; agregar un form que pida un nombre al usuario. Después de ingresarlo se debe buscar en la tabla y mostrar los datos encontrados o un mensaje indicando que la búsqueda fue infructuosa.

Agregar una nueva opción en el programa anterior, para realizar búsquedas inexactas por nombre o fecha de nacimiento, de la siguiente manera:

Si se selecciona buscar por Nombre y Apellido, se toma el contenido del editor como nombre; caso contrario, como fecha de nacimiento. Después de realizar la búsqueda hay que mostrar los datos encontrados o un mensaje indicando el fallo de la operación.



C El método Locate

El método Locate es un nuevo procedimiento de búsqueda agregado en las últimas versiones de Delphi. En breve, este método determina cuál es la forma más rápida y eficiente de cumplir con nuestro pedido de búsqueda de datos, y lo ejecuta; si existe un índice se utiliza y si no, la búsqueda será un poco más lenta pero se realizará igual.

C Filtros

El problema con los métodos de búsqueda que vimos antes es que si tenemos más de un registro que cumpla las especificaciones (por ejemplo, buscamos el apellido “Pérez”, y resulta que hay 12 “Pérez” en la tabla), sólo se muestra el primero de ellos. Para ver los demás, tenemos que avanzar en la tabla. Dado que es prácticamente obligatorio usar un índice, sabemos que los registros con nuestro dato estarán todos juntos (todos los “Pérez” están uno a continuación del otro) por lo que sin mucho problema podemos avanzar y recorrer todos.

Pero no es muy elegante que habiendo buscado “Pérez”, de golpe el usuario se encuentre con un “Zapata”.

Podemos evitar eso poniendo límites a los registros que verá el usuario: solamente tendrá acceso a los registros que cumplan con las condiciones de búsqueda. Este proceso se denomina *Filtrado* de la tabla.

El filtrado consiste en especificar una condición que tienen que cumplir los registros para poder ser vistos. Los registros que no cumplan con esa condición no podrán ser accedidos hasta que no retiremos la restricción. Incluso la función “RecordCount” nos devuelve la cantidad de registros *que cumplen la condición*, como si sólo éstos existieran en la tabla.

Hay por supuesto una desventaja: *el acceso a una tabla filtrada es mucho más lento, en general, que el acceso a una tabla sin filtrar*. Entonces como regla práctica se prefiere no utilizar los filtros cuando la tabla es muy grande o está relacionada con otra -ya veremos este tema.

C Consultas SQL (tQuery)

El SQL (Structured Query Language) es un lenguaje de consulta (si, otro lenguaje *distinto* del Pascal, del Basic, del C...) creado especialmente para responder a la necesidad de trabajar en forma eficiente con los datos en el modelo relacional de Bases de Datos, que es el que utilizamos aquí. Este tema será tratado con extensión en otro lugar. Notemos por ahora que este lenguaje permite hacer una pregunta a una tabla, y la respuesta será *otra tabla* conteniendo los registros que cumplen con la consulta. No es necesario tener definido un índice en los campos de la consulta, aunque si los tenemos la BDE los utilizará automáticamente para acelerar el proceso. La consulta puede obtener datos de más de una tabla a la vez -estableciendo relaciones entre ellas. Es un método de búsqueda muy poderoso.

Validación de datos

La validación de datos es el proceso de controlar los valores: por ejemplo, controlar que la fecha de nacimiento de una persona no sea mayor que la fecha de muerte (salvo que descubran una manera de hacer que el tiempo vaya para atrás...) O bien que se cumplan las reglas o políticas de negocio (business rules) de la aplicación particular, por ejemplo si en un campo debemos ingresar el nombre de la tarjeta de crédito con la que va a trabajar el cliente y en nuestra empresa no se acepta otra que no sea Mastercard o Visa, entonces hay que controlar que el nombre ingresado sea uno de estos y rechazarlo en caso contrario.

Ya hemos visto algunos de estos controles en la definición de las *reglas de validación (Validity Checks)* en la estructura de la tabla. Estas reglas son restricciones impuestas a los valores de los distintos campos, y son controladas por el motor de base de datos (no por nuestro programa). También se pueden hacer los mismos controles (y muchos más) en el programa; la diferencia, pros y contras de cada estrategia se discutirán al final cuando hablemos del diseño cliente/servidor.

Por ahora nos atendremos a nuestra línea de estudio: veremos qué reglas se pueden imponer desde Delphi, cómo imponerlas y cuándo.

Las restricciones son particulares de cada *campo* -no de un registro- por lo que el lugar lógico donde las encontraremos es... los componentes de campo.

Los componentes de campo tienen una serie de propiedades que permiten la implementación de algunas reglas de validación:

- C **ImportedConstraint** (restricción importada): conviene no tocarla, contiene una expresión traída directamente de la base de datos con la que trabaja. Esta propiedad **no funciona** con las tablas de Paradox o Dbase. Cuando no se cumple esta restricción, se muestra el mensaje dado en la propiedad **ConstraintErrorMessage**.
- C **CustomConstraint** (restricción personalizada): una condición escrita como una expresión del tipo de las utilizadas en SQL. Cuando se intenta introducir un nuevo valor en este campo, se comprueba esta restricción. Por ejemplo: `ElCampo>10 and ElCampo<100`. El nombre del campo no necesita ser el verdadero, puede ser cualquier cadena de caracteres mientras no sea una palabra reservada de SQL. Cuando la condición no se cumple, se muestra el mensaje de la propiedad **ConstraintErrorMessage**.
- C **ConstraintErrorMessage**: el mensaje que se mostrará en una ventana cuando se produzca una

violación de las restricciones impuestas en las propiedades **ImportedConstraint** o **CustomConstraint**.

- C **DefaultExpression**: valor por defecto que se asigna al campo cuando se agrega un registro. *Esta propiedad no se importa desde la estructura de las tablas de Paradox o Dbase*. Si se utilizan valores no numéricos, hay que indicarlos entre comillas simples, por ejemplo: 'El cliente' o '13:30'. Los valores especificados en esta propiedad tienen prioridad sobre los indicados en la estructura de la tabla.
- C En los campos numéricos tenemos las propiedades **MinValue** y **MaxValue**, que nos permiten especificar un valor mínimo y un valor máximo para el campo correspondiente. Estos valores tampoco se traen automáticamente de la definición de la estructura de la tabla.
- C **Required**: cuando está en Verdadero, no se puede dejar vacío este campo.
- C **EditMask**: especifica el formato de edición para los campos no numéricos. Por ejemplo, una máscara común para los campos de fecha es la cadena "90/90/0000" (escrita sin comillas).
- C **EditFormat**: máscara de entrada (igual que **EditMask**) pero para campos numéricos.

Las dos últimas propiedades nos permiten una validación *caracter a caracter*; a medida que vamos escribiendo en un control (por ejemplo un editor), sólo se permite la entrada del carácter escrito si corresponde con la máscara.

¿Y qué sucederá si escribimos un valor en el campo *por programa*, es decir sin utilizar un control? ¿Se controla la correspondencia con la máscara de entrada?

La respuesta es... NO! Podemos burlar ese control de esa manera. Hay otro modo de controlar la validez de los datos, que siempre es invocado: el evento **OnValidate** de los componentes de campo.

Cada vez que se escribe un valor en un campo, se realizan los siguientes pasos:

- Se genera el evento OnValidate
- Se escriben los datos en el registro temporal de edición
- Se genera el evento OnChange

Por tanto, tenemos la oportunidad de detener el ingreso de un valor en el evento OnValidate. Para detener el proceso descrito anteriormente, hay que provocar una excepción: por ejemplo, llamando al procedimiento **Abort** (no se muestra un mensaje de error).

Hay otras consideraciones a tener en cuenta en la implementación de restricciones a los campos, sobre todo cuando se trabaja en red.

Por ejemplo, podríamos preguntarnos ¿qué ocurre si se implementan todas las restricciones en el programa y un usuario ingresa a la tabla a través del Database Desktop? Bien, definamos entonces las restricciones en la estructura de la tabla. Pero entonces, cuando ingresamos un valor en un campo éste no es comprobado y recién podremos saber si es correcto cuando enviamos el registro completo al servidor... y éste nos devuelve un mensaje de error, lo cual implica que nuestro programa tendrá que esperar la confirmación del servidor y responder a los errores en caso que los haya.

Estas y otras consideraciones hacen que el diseño de Bases de Datos se complique un poco. Hablaremos más de los problemas y soluciones cuando veamos el modelo Cliente/Servidor.

Para cerrar el tema de la validación, existe otro lugar donde podemos controlar si los datos ingresados son correctos, pero a nivel de registro (cuando se termina de modificar todo el registro, no un solo campo): antes

de escribir los datos al archivo. Los componentes de tipo Dataset tienen un evento llamado **BeforePost** que se genera inmediatamente antes de enviar el registro al archivo. Nuevamente, si vemos que algo anda mal, debemos detener el proceso provocando una excepción.

Resumiendo. Tenemos varias opciones para la validación de los datos:

- C En la aplicación cliente
 - C caracter por caracter, con las propiedades **EditMask** o **EditFormat** de los componentes de campo
 - C cuando el campo está completo, con las propiedades **ImportedConstraint** y **CustomConstraint** de los componentes de campo
 - C cuando el campo está completo, en el evento **OnValidate** de los componentes de campo
 - C cuando el registro está completo, en el evento **BeforePost** del componente Dataset (Table, Query, etc)
- C En el servidor
 - C Implementación de las restricciones en la estructura de cada tabla. Son aplicadas por el servidor y validan el registro completo.

Veamos un poco más en profundidad los métodos y eventos que nos permitirán trabajar por programa con los datos.

Trabajar con las tablas: métodos y eventos

Los componentes de tabla tienen métodos que nos permiten trabajar con los registros de una tabla; podemos buscar, agregar, borrar registros llamando a los métodos del componente. Ya trabajamos con algunos de estos métodos al crear nuestro propio navegador (métodos First, Last, Post, Cancel, etc) y cuando hablamos de la búsqueda (FindKey, FindNearest, Locate, etc). Veamos ahora una lista un poco más completa.

Métodos

Abrir/Cerrar una tabla

L **Open**: abre la tabla. Equivale a poner la propiedad **Active** en True.

L **Close**: cierra la tabla. Equivale a poner la propiedad **Active** en False.

Grabar/Descartar modificaciones en un registro

L **Post**: confirma las modificaciones y graba los cambios en el archivo de tabla.

L **Cancel**: descarta los cambios; carga el buffer de memoria con los valores del registro actual de la tabla.

Agregar/Borrar/Actualizar registros

NOTA: siempre que se agregue un registro, éste se ordenará según el índice activo. No importa si mientras lo estamos ingresando aparece por ejemplo al final de una grilla, o en el medio.

L **Append**: agrega un registro en blanco.

L **AppendRecord**: agrega un registro con los valores iniciales pasados como parámetros.

L **Insert**: inserta un registro en blanco en la posición actual.

L **InsertRecord**: inserta un registro con los valores iniciales pasados como parámetros.

L **Delete**: borra el registro actual.

Mover el puntero

NOTA: no se puede indicar directamente un número de registro para posicionarse; la numeración es poco segura cuando se trabaja en red. Supongamos que podemos hacerlo, y queremos ir al registro nro. 32 porque vimos que allí estaban los datos que tenemos que modificar. Pero resulta que mientras escribíamos el número 32 para saltar allí, otro usuario hizo un **Post** que inserta un registro nuevo en la posición 24. ¡Ahora nuestro registro estará en la posición 33! Es por esto que en lugar de números absolutos de registro utilizamos *marcadores* (Bookmarks) para indicar un registro particular. Pero sí tenemos métodos que nos permiten movernos en forma relativa:

L **First**: nos ubica en el primer registro de la tabla.

L **Prior**: va al registro anterior.

L **Next**: va al registro siguiente

L **Last**: va al último registro

L **MoveBy**: mueve el puntero en la cantidad de registros pedida; si el valor es negativo se mueve para atrás, si no para adelante.

Marcadores (Bookmarks)

L **GetBookmark**: crea un marcador a la posición actual del cursor en la tabla.

L **GotoBookmark**: mueve el cursor al marcador dado como parámetro.

L **FreeBookmark**: libera los recursos ocupados por el bookmark dado como parámetro.

Filtrado de datos

L **SetRange**: pone el comienzo y el fin del rango de filtrado en los valores pasados como parámetros.

L **ApplyRange**: aplica el rango, filtrando efectivamente los datos. A partir de esta llamada, sólo podemos ver los datos que entran en el rango.

L CancelRange: cancela el rango aplicado. Después de esta llamada se pueden acceder todos los registros.

Eventos

Además de los métodos, la tabla tiene eventos que se producen cuando realizamos ciertas acciones sobre la tabla. Aquí podemos incluir cualquier operación, lo que nos da un gran nivel de control sobre el trabajo con los datos.

L BeforeCancel: se produce antes de cancelar una operación; es la primera acción que toma el método Cancel.

L AfterCancel: es la última acción que hace el método Cancel.

Si el componente de datos (tTable o tQuery) no está en modo de edición o no hay cambios pendientes, no se producen estos eventos.

L BeforeClose / L AfterClose

Estos eventos se llaman al cerrar la tabla, ya sea con una llamada directa al método Close o poniendo la propiedad Active a false.

L BeforeDelete / AfterDelete

Estos eventos se llaman al borrar un registro.

L BeforeEdit / AfterEdit

BeforeEdit se produce antes de entrar a modo edición (estado dsEdit); AfterEdit se produce después de entrar al modo edición. Ojo- no al *salir* de modo edición, sino al terminar de entrar a ese modo.

L BeforeInsert / AfterInsert

Lo mismo que para BeforeEdit y AfterEdit, pero para el estado dsInsert.

L BeforeOpen / AfterOpen

Se producen antes y después de abrir la tabla, ya sea con el método Open o poniendo Active=true.

L BeforePost / AfterPost

Se producen antes y después de hacer un post, ya sea explícito -llamando al método post- o implícito, por ejemplo al movernos en la tabla.

L Before/AfterScroll

Se generan antes y después de mover la posición del cursor en la tabla.

L OnCalcFields

Este evento se usa para poner los valores de los campos calculados. Se llama cuando se lee un registro de una tabla. Además, si la propiedad `AutoCalcFields` es `true`, se llama este evento cuando se modifica un registro no calculado mientras el componente de datos está en estado de edición o inserción.

Este evento debe mantenerse corto, ya que será llamado muchas veces. Además, se debe evitar modificar otros registros en él ya que esto podría llevar a una recursión.

Cuando se entra en este evento la tabla se pone en estado `CalcFields`. En este estado, no se pueden modificar otros campos que no sean calculados. Al salir del evento se vuelve la tabla a su estado inicial.

L OnNewRecord

Este evento se produce después de `BeforeInsert` y antes de `AfterInsert`. Cualquier valor colocado en el registro en este evento no activará el flag de `Modificado (Modified)`, por lo que podemos inicializar los campos en un registro nuevo sin que se tome como una modificación.

L OnFilterRecord

Se genera por cada registro leído de la tabla, cuando la propiedad **Filtered** es verdadera. Aquí podemos implementar el filtrado de los datos: si ponemos el parámetro *Accept* en `TRUE` el registro se muestra.

L OnUpdateRecord

Se genera cuando se aplican *actualizaciones pendientes (cached updates)* a un registro. Se puede usar para hacer actualizaciones en cascada o para tratar casos especiales.

L OnDeleteError

Ocurre cuando se genera una excepción al querer borrar un registro.

L OnEditError

Ocurre cuando se genera una excepción al editar un registro.

L OnPostError

Ocurre cuando se genera una excepción al intentar guardar un registro con **Post**.

L OnUpdateError

Ocurre cuando se produce una excepción al aplicar actualizaciones pendientes (`Cached Updates`).

El modelo relacional. Normalización

El modelo relacional de Bases de Datos fue concebido por E. F. Codd en 1969. Está basado en una teoría matemática -en las disciplinas de teoría de conjuntos y lógica de predicados. La idea básica es la siguiente:

Una base de datos consiste de una serie de tablas desordenadas (llamadas relaciones) que pueden ser manipuladas usando operaciones no procedurales que devuelven tablas.

Cuando se diseña una base de datos, hay que tomar decisiones sobre cómo es mejor modelado un sistema del mundo real. Este proceso consiste en decidir las tablas a crear, las columnas que contendrán, y las relaciones entre las tablas. Este proceso toma tiempo y esfuerzo, y es más cercano a un arte que a una ciencia.

La definición de la estructura de una base de datos puede decidir el éxito de una aplicación

Esta afirmación, aunque un poco categórica, no deja de ser verdad; en todas las aplicaciones será necesario efectuar cambios y ajustes sobre la marcha, y es mucho más fácil si está bien diseñada. En caso contrario, puede incluso tornar *imposible* la tarea de agregar alguna funcionalidad.

Algunos de los beneficios de una base de datos organizada de acuerdo al modelo relacional son los siguientes:

- Eficiencia en las entradas, actualizaciones, y el borrado de elementos
- Eficiencia en la recuperación, sumarización y reporte de los datos
- Se puede predecir el comportamiento de la base de datos, dado que sigue un modelo bien formulado
- Es muy simple hacer cambios en el esquema de la base de datos (definición de las tablas)

Las tablas en el modelo relacional son usadas para representar cosas (entidades) del mundo real, que pueden ser objetos (un paciente, una factura) o sucesos (una visita del paciente, una llamada de teléfono). *Cada tabla debería representar sólo un tipo de entidad.*

Clave primaria

Las tablas se componen de filas y columnas. El modelo relacional dicta que *las filas en una tabla deben ser únicas*. Caso contrario, no habría forma de seleccionar una fila unívocamente. La unicidad de las filas de una tabla se garantiza definiendo una *clave primaria* (primary key).

Una clave primaria se compone de una o varias columnas (clave simple o compuesta, respectivamente). Estas columnas contienen valores que no se repiten, es decir, permiten determinar unívocamente una fila de la tabla.

Por ejemplo, en una tabla que contenga datos de personas, no podríamos usar la columna de "Nombre" para la clave primaria, dado que puede haber más de una persona con el mismo nombre. Lo mismo sucede con el apellido, e incluso con una clave compuesta por las dos columnas, "Nombre + Apellido". Hay muchas

familias en las cuales los hijos tienen el mismo nombre que los padres o abuelos. ¿Y si agregamos a esta clave compuesta la columna “Dirección”?

En el ejemplo anterior, una buena elección sería una columna con el Nro. de Documento de Identidad. Este número, por definición, es único para cada persona y por lo tanto cumple con los requisitos.

Clave externa

Las claves externas son columnas de una tabla cuyos valores referencian la clave primaria de otra tabla. Estas claves permiten relacionar las tablas.

Las claves primarias y externas que se relacionen deben estar en el mismo *dominio* o conjunto de valores. Es decir: deben ser del mismo tipo de datos y deben tener el mismo rango de valores posibles. Por ejemplo, si la clave primaria de la tabla de empleados es un campo numérico entero con valores entre 1 y 10000, y queremos relacionar con ésta otra tabla que contenga datos de trabajos realizados, la clave externa de la segunda tabla debe ser también entera y con valores de 1 a 10000.

Los gestores de bases de datos como Access o Paradox no controlan esta restricción totalmente; sólo toman en cuenta el tipo de datos. Así por ejemplo, sería posible relacionar una clave “Años trabajados” con una “Edad” si son del mismo tipo pero provienen de diferentes dominios.

Relaciones

Las relaciones entre tablas indican que las filas de una están ligadas a las de otra de alguna manera, de la misma manera que las relaciones entre las entidades del mundo real (como por ejemplo, la relación “es hijo de” entre un conjunto de padres y un conjunto de chicos).

Las relaciones se hacen entre dos tablas, y pueden ser de tres tipos: uno-a-uno, uno-a-muchos, y muchos-a-muchos.

Relaciones uno-a-uno

Dos tablas están relacionadas de manera uno-a-uno si, para cada fila de la primera tabla, hay *como máximo* una fila de la segunda. Estas relaciones se dan raramente en el mundo real, y son utilizadas principalmente cuando se desea dividir una tabla en dos (por ejemplo, por razones de seguridad).

Relaciones uno-a-muchos

Dos tablas están relacionadas de manera uno-a-muchos si para cada fila de la primera tabla hay cero, una, o muchas filas de la segunda tabla; pero para cada fila de la segunda tabla hay *sólo una* de la primera. Esta relación también se conoce con el nombre de *relación padre-hijo* (*parent-child*) o *relación principal-detalle* (*master-detail*).

Es el tipo de relaciones más utilizado. El ejemplo típico es una factura u orden de pedido: normalmente pondremos los datos de la orden o factura en una tabla, y los datos de los ítems pedidos en otra. De esta manera se pueden pedir tantos ítems como se quiera para una orden, pero cada ítem individual corresponde a una sola orden.

Otra aplicación muy común se da en las tablas de búsqueda (lookup tables), cuando el valor para un campo está restringido a los valores existentes en otra tabla; por ejemplo, los ítems de una factura sólo pueden ser los que están en la tabla de stock.

Relaciones muchos-a-muchos

Dos tablas están relacionadas de manera muchos-a-muchos cuando para cada fila de la primera tabla puede haber muchas en la segunda, y para cada fila de la segunda puede haber muchas en la primera. Este tipo de relaciones se da por ejemplo en el caso de una tabla con datos de empresas de seguros y otra con datos de asegurados. Cada empresa puede tener más de un asegurado, y cada asegurado puede a su vez estar en más de una empresa.

Los sistemas de bases de datos relacionales no pueden modelar las relaciones muchos-a-muchos directamente. Hay que dividir la relación en dos de tipo uno-a-muchos creando una tabla intermedia. En el ejemplo anterior tendríamos que crear una tabla de enlace con una clave externa a cada una de las dos tablas relacionadas, por ejemplo con campos ID_Empresa e ID_Asegurado.

Normalización

El proceso de normalización tiene por objetivo alcanzar la estructura óptima en una base de datos. La teoría nos da el concepto de *formas normales* como reglas para alcanzar progresivamente un diseño mejor y más eficiente. El proceso de normalización es el proceso de reestructurar la base de datos para organizar las tablas de acuerdo a las formas normales.

Un par de notas antes de continuar:

- En el proceso de normalización de tablas *no se pierde información*; sólo se estructura de manera diferente.
- Las tablas se descomponen de tal manera que siempre sea posible volver a reunir las utilizando consultas; en la práctica, esto significa que tiene que haber relaciones entre las nuevas tablas resultantes de la descomposición.
- La normalización *no es obligatoria*, sólo recomendable. Hay veces que convendrá quebrantar alguna de las reglas; no obstante, hay que estar bien seguro de las razones para hacer esto, ya que estas recomendaciones son resultado de largos estudios sobre optimización del proceso de los datos.

Daremos aquí sólo las definiciones de las primeras tres formas normales y algunos ejemplos.

NOTA: los campos marcados con un asterisco (*) forman la clave principal de cada tabla

1^{ra} forma normal (1NF): los valores de todas las columnas deben ser atómicos, esto es, no divisibles. Además, no debe haber campos repetidos.
--

Por ejemplo si tuviéramos en una tabla de facturas registros como los siguientes, estaríamos violando la primera forma normal:

NroFactura (*)	Cliente	Items
0001-000001	Aquiles Meo de la Torre	1 mesa, 6 sillas, 1 aparador
0001-000002	Elba Gallo	2 estanterías, 1 mesa ratona

La segunda parte de la 1ra forma prohíbe una reestructuración como la siguiente:

NroFactura (*)	Cliente	Item1	Item2	Item3
0001-000001	Aquiles Meo de la Torre	1 mesa	6 sillas	1 aparador
0001-000002	Elba Gallo	2 estanterías	1 mesa ratona	

Aquí se ve el problema de poner varios campos para el mismo tipo de información; si alguien compra más de tres ítems es un problema; lo mismo si queremos averiguar por ejemplo cuantas sillas se han vendido habría que rastrear las tres columnas buscando sillas. El diseño no es bueno.

Para que esta tabla cumpla los requisitos de la 1ra forma normal, podríamos estructurarla como sigue:

NroFactura (*)	NroItem (*)	Cliente	Cantidad	Item
0001-000001	1	Aquiles Meo de la Torre	1	Mesa
0001-000001	2	Aquiles Meo de la Torre	6	Sillas
0001-000001	3	Aquiles Meo de la Torre	1	Aparador
0001-000002	1	Elba Gallo	2	Estanterías
0001-000002	2	Elba Gallo	1	Mesa ratona

En este caso formaríamos una clave primaria con el Nro de Factura y el Nro de Item juntos. Con esta construcción, sería muy fácil consultar por la cantidad de sillas vendidas.

2^{da} forma normal (2NF): se dice que una tabla está en segunda forma normal cuando está en 1NF y cada columna que no es clave es dependiente completamente de la clave primaria entera.

En otras palabras, la clave primaria debe implicar el valor de cada otra columna en el registro. Esto generalmente viene solo como resultado de *estructurar la tabla con datos de distintas entidades*. Por ejemplo, si en la tabla de datos vendidos anterior agregamos un campo "Fecha" para llevar la fecha de emisión de la factura, tendríamos una situación como la siguiente:

NroFactura	NroItem	Fecha	Cliente	Cantidad	Item
0001-000001	1	4/8/98	Aquiles Meo de la Torre	1	Mesa
0001-000001	2	4/8/98	Aquiles Meo de la Torre	6	Sillas
0001-000001	3	4/8/98	Aquiles Meo de la Torre	1	Aparador
0001-000002	1	5/8/98	Elba Gallo	2	Estanterías
0001-000002	2	5/8/98	Elba Gallo	1	Mesa ratona

El problema aquí está en que el campo Fecha depende sólo del nro. de factura, no del nro. de ítem. Es decir,

no depende de la clave principal completa sino sólo de una parte. En otras palabras, cada valor de la clave primaria completa *no define* los demás campos del registro; hay un campo que se define sin necesidad de conocer la clave completa.

Para alcanzar la 2NF, se puede dividir la tabla anterior en dos tablas diferentes, una con los datos de las facturas y la otra con los datos de los ítems.

Esto se aplica en general, como ya hemos visto más arriba: *cada tabla debería representar sólo un tipo de entidad*. Aplicando esta regla práctica casi siempre aseguramos que se cumpla la 2NF (y por lo tanto la 1NF).

3^{ra} Forma Normal (3NF): se dice que una tabla está en tercera forma normal si está en 2NF y *todas las columnas que no son clave son mutuamente independientes*.

Esto elimina inmediatamente las columnas con valores calculados, así como las columnas que muestran la misma información de diferentes maneras. Por ejemplo, en el caso de la factura anterior podríamos agregar campos a la tabla de ítems para llevar cuenta del precio unitario de cada uno, y calcular el subtotal de cada línea multiplicando este valor por la cantidad de artículos dada en el campo *cantidad*; pero para cumplir con 3NF, este cálculo no se almacenará en un nuevo campo de la tabla. Para esto se definen los **campos calculados**, que no tienen existencia en la tabla física sino que se crean “al vuelo” en el momento de mostrar los datos. Ya hablamos de estos campos al trabajar con los campos virtuales; también es posible crearlos en una consulta SQL.

Implementación en Delphi

Veamos ahora cómo implementan Delphi y la BDE este modelo relacional.

La BDE nos permite relacionar tablas a través de campos clave, de manera que al seleccionar un registro de la tabla principal (master) se filtre automáticamente la tabla secundaria (detail). Asimismo, cuando agregamos un registro en la tabla detalle los campos relacionados toman los valores correspondientes al registro actual de la tabla principal.

En Delphi, la relación se hace a través de las propiedades `MasterSource`, `MasterFields` e `IndexFields`.

MasterSource especifica el `DataSource` utilizado por la tabla principal.

MasterFields indica los campos de la tabla maestra a usar para mantener el enlace.

IndexFields indica los campos de la tabla detalle que se relacionan con los de la principal.

En estas dos últimas propiedades, si se utiliza más de un campo para la relación se separan con punto y coma.

Vamos a resumir los pasos necesarios para enlazar dos tablas, principal y secundaria, a través de un ejemplo utilizando tablas de demostración de Delphi:

- 1) Poner un componente `Table` y enlazar con el archivo de la tabla principal (**Table1** - `DatabaseName=DBDEMOS, TableName=Orders.db`)
- 2) Colocar otro componente `Table` y enlazar con el archivo de la tabla secundaria (**Table2** - `DatabaseName=DBDEMOS, TableName=Items.db`)

- 3) Colocar dos fuentes de datos (Datasource) y enlazar cada uno con una tabla (**DataSource1 - Dataset=Table1; DataSource2 - Dataset=Table2**)
- 4) En la tabla secundaria, modificar la propiedad **MasterSource** para que señale a la Fuente de datos de la tabla principal (**Table2 - MasterSource=DataSource1**)
- 5) Indicar en la tabla secundaria los campos que determinan la relación (clave externa -propiedad **IndexFieldNames**), y a qué campos referencian de la tabla principal (propiedad **MasterFields**). *Debe haber un índice definido sobre los campos de la tabla secundaria que se enlazan.*

En nuestro ejemplo, se enlazarán las dos tablas mediante los campos OrderNo de cada tabla. No es necesario que los campos que enlazan las tablas se llamen igual, solamente que tengan el mismo tipo de datos.

Actualmente, en Delphi utilizaremos un editor especial de relaciones para enlazar las tablas. Presionando el botón con tres puntos de la propiedad **MasterFields** (una vez seleccionado el **MasterSource**) entramos al editor de relaciones (Fig. 23).

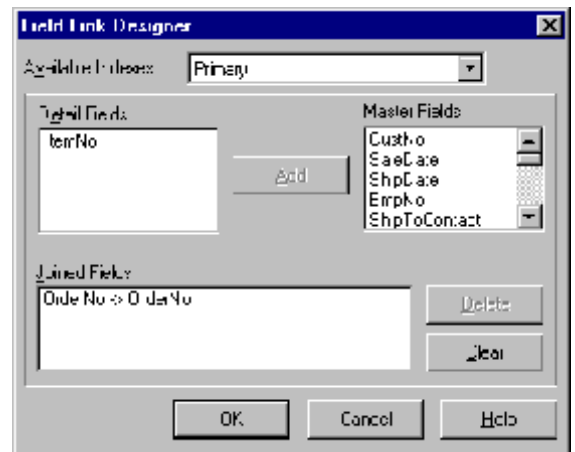


Figura 23: enlace de campos entre tablas relacionadas

En el *combobox* de arriba podemos elegir uno de los índices de la tabla detalle (el valor elegido aquí se coloca automáticamente en la propiedad **IndexName**).

Dependiendo del índice escogido, nos aparecen listados a la izquierda los campos disponibles de la tabla detalle. En la lista de la derecha vemos los campos de la tabla principal.

Para crear una relación, seleccionamos un campo de cada lista y presionamos el botón **Add**. La relación se muestra en la lista de abajo, con el título 'Joined Fields' (campos enlazados) y los campos utilizados se eliminan de las listas de arriba. Los campos de la tabla detalle se ponen en la propiedad **IndexFieldNames**, mientras que los de la principal se colocan en la propiedad **MasterFields**.

Una vez establecida la relación, veremos que la tabla secundaria se *filtra* automáticamente mostrando solo los registros que tienen en el campo OrderNo el mismo valor que el campo OrderNo del registro seleccionado en la tabla principal.

En el ejemplo colocamos dos grillas -cada una mostrando una tabla- para ver el efecto de la relación. La ficha queda como se ve en la figura 24.

Para comprobar el filtrado automático de la tabla secundaria, ejecutemos el programa y movamos el cursor de un registro a otro. Veremos que la grilla inferior cambia por cada registro, mostrando sólo aquellos registros cuyo campo **OrderNo** coincide con el campo homónimo del registro seleccionado en la tabla Principal.

La BDE mantiene la integridad de los datos si fue definida en la estructura de la tabla. Además, cuando se agrega un registro a la tabla secundaria se coloca automáticamente el valor correcto en el campo enlazado, en el ejemplo el campo **OrderNo**.



Figura 24: tablas relacionadas en acción

Hemos creado una relación de tipo uno-a-muchos.

Ejercicio 2-10

Modificar la aplicación del ejemplo incluyendo otra grilla mostrando una tercera tabla: Parts.db, de la misma Base de Datos. Tomar esta tercera tabla como tabla secundaria en relación con la tabla de los items, Table2, enlazadas por el campo **PartNo**.

Al movernos en la segunda tabla, vemos que la tercera automáticamente se acomoda para mostrar el registro que tiene el mismo número de parte.

Ejercicio 2-11

Modificar la aplicación del ejemplo para hacer que la segunda tabla muestre al lado del número de parte la descripción correspondiente. (Ayudita: recordar los campos de búsqueda y las columnas de la grilla). ¿Se viola la 3ra. Forma Normal?

Ahora podemos terminar el ejemplo de la factura, aplicando las recomendaciones de normalización.

Ejercicio 2-12

Crear una aplicación que simule una factura. Los datos de la factura son Nro, Tipo, Fecha, Cliente. Los items deben ir en otra tabla con los siguientes campos:

IDItem, NroFactura, TipoFac, Cantidad, Descripcion, Precio Unitario.

El enlace se hace entre los campos NroFactura-TipoFac (tabla secundaria) y Nro-Tipo (tabla principal).

Ejercicio 2-13

Agregar en el ejemplo de la factura un campo virtual *Subtotal* que muestre para cada línea de detalle el subtotal correspondiente (Cantidad * Precio Unitario). ¿En cuál de las dos tablas lo agregaría? ¿Se viola aquí la 3ra. Forma Normal?

Integridad referencial, o cómo eliminar los orfanatos

Hemos visto anteriormente al estudiar el motor de Paradox, el concepto de *Integridad Referencial*. Lo recordamos aquí:

Integridad Referencial (Referential Integrity): es una regla que dice que para cada registro de una tabla de detalle que esté en relación con otra principal *debe existir un valor de la clave primaria (en la tabla principal) para cada valor de la clave externa (en la tabla secundaria)*. Esto es, no pueden quedar registros “huérfanos” en el detalle, que no se correspondan con ningún registro de la tabla principal.

Consideremos los problemas que se nos pueden presentar para cumplir con esta regla.

1) En la tabla secundaria

- C Al agregar un registro en la tabla secundaria (detalle): el valor de los campos que forman la clave externa deben tener valores que se encuentren en los campos referenciados de la tabla principal. Por ejemplo, en la factura no se puede agregar un registro de detalle con un número de factura que no exista.
- C Al modificar un registro en la tabla secundaria: igual que en el caso anterior, debemos controlar que el nuevo valor de los campos de la clave externa referencie a un registro existente en la tabla principal.
- C Al borrar un registro en la tabla secundaria: no pasa nada, no es necesario controlar nada.

2) En la tabla principal

- C Al agregar un registro: no hay problemas.
- C Al borrar un registro: debemos controlar que no queden registros de detalle referenciando a este registro. Las opciones son borrarlos a todos los secundarios junto con el principal (borrado en cascada), o impedir el borrado del principal mientras haya alguno en la tabla secundaria. Como dijimos antes, Paradox *prohíbe* directamente el borrado del registro principal. Veremos aquí como hacer el borrado en cascada, por programa.
- C Al modificar un registro: las mismas consideraciones que en el caso anterior, pero en este caso Paradox permite las dos opciones, modificación en cascada o prohibición total.

Veamos entonces cómo hacer el caso que Paradox no contempla: Borrado en cascada de registros.

Verifiquemos primero la restricción: en la aplicación del ejemplo con dos tablas (Orders.db e Items.db) trate de borrar un registro de la tabla principal. Paradox lo detendrá con un mensaje que indica que existen registros relacionados en la tabla de detalle. No se puede borrar un registro de la tabla principal si tiene registros relacionados en la secundaria porque éstos quedarían huérfanos.

El proceso a grandes rasgos sería el siguiente: detectar cuando se quiere borrar un registro de la tabla principal y antes de hacerlo, borrar todos los registros de la tabla secundaria que lo referencian. Nada más...

Pasemos del dicho al hecho. ¿Cómo hacemos para detectar cuando se quiere borrar un registro de la tabla principal? Pues con el evento BeforeDelete, claro. En el momento de producirse este evento debemos filtrar la tabla secundaria y eliminar todos los registros que queden visibles.

En realidad, la tabla secundaria *ya está filtrada* porque está en relación con la principal. Ergo, sólo nos queda borrar todo lo que veamos en la tabla secundaria. Podemos lograrlo con un bucle como el siguiente:

```
Table2.First;  
while not Table2.Eof do  
    Table2.Delete;
```

Al borrar el registro actual, el cursor se posiciona en el siguiente.

¡La parte divertida queda para Uds!

Ejercicio 2-14

Modificar la aplicación de la factura creada en los ejercicios anteriores para que mantenga la Integridad Referencial en todas las situaciones.

)

SQL - 1ra parte

El Lenguaje de Manipulación de Datos (DML)

Los ejemplos de esta sección utilizan Access 97

Veremos aquí una introducción al Lenguaje Estructurado de Consulta o SQL. Dividimos el estudio en dos grandes áreas: los comandos para manipulación de datos (que llamaremos DML, Data Manipulation Language) y los comandos para definición de datos (que llamaremos DDL, Data Definition Language). Para estar seguros que todos hablamos de lo mismo, comenzaremos con algunas definiciones.

SQL

SQL son las iniciales de “Structured Query Language” o Lenguaje Estructurado de Consulta. Es un lenguaje no procedural inventado en IBM a principios de los años 70, para implementar el modelo relacional de Codd. Inicialmente se le llamó SEQUEL (Structured English Query Language), luego pasó a ser SEQUEL/2 y finalmente SQL (esperemos que no siga la tendencia y en unos años se termine llamando S). Hoy en día es el lenguaje de consulta más utilizado por los gestores de Bases de Datos de todos los tamaños, desde Dbase o Paradox pasando por Oracle, Informix o SQL Server, hasta los gestores de datos utilizados en supercomputadoras.

Hay definidos tres estándares oficiales por el Instituto Nacional de Estándares Americano (American National Standards Institute, ANSI): SQL-86, SQL-89 y SQL-92.

La mayoría de los gestores de consultas implementan “a su manera” las recomendaciones del estándar. Estudiaremos los comandos más comunes, que se aplican a los motores de consulta más utilizados, y algunas particularidades de los formatos locales (Paradox, Access).

QBE

La creación de consultas utilizando ejemplos (Query By Example) es una forma fácil y gráfica de realizar una pregunta a la Base de Datos. Access en particular contiene un editor QBE muy bueno, el cual utilizaremos para aprender a realizar consultas básicas.

Crear consultas con el editor QBE

Para trabajar con consultas, debemos primero seleccionar la página correspondiente en la ventana de base de datos (Fig. ?). Aquí se nos muestra una lista de las consultas ya definidas, que forman parte de la Base de Datos.

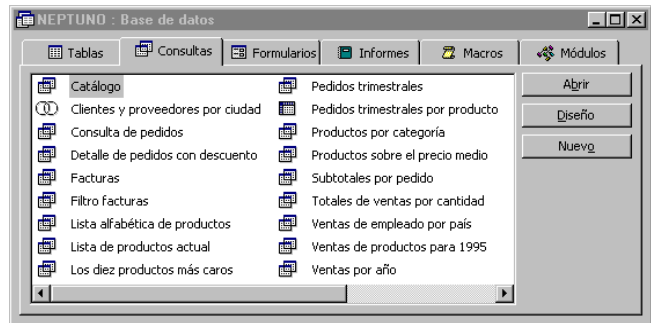



Figura 1: Página “Consultas” de la base de datos Neptune

En esta página, presionamos el botón “Nuevo” para indicar a Access que queremos hacer una nueva consulta.

El sistema nos ofrecerá una serie de opciones para crear la nueva consulta (Fig. 2). La mayoría son asistentes que nos llevarán paso a paso por la creación de determinados tipos de consultas; aquí veremos cómo definir una consulta en *modo diseño*, sin asistente. Por lo tanto vamos a seleccionar la opción “Vista diseño” para ver el **Editor QBE** (Fig. 3).

El primer paso para crear una consulta es seleccionar las tablas que intervendrán en la misma¹. Para ello Access nos muestra una lista de las tablas y consultas disponibles (Fig. ?). La primera vez que abrimos el editor QBE esta ventana aparece sola, en otro momento debemos indicar que queremos agregar una tabla con el comando  **Mostrar Tabla** del menú **Consultas**.

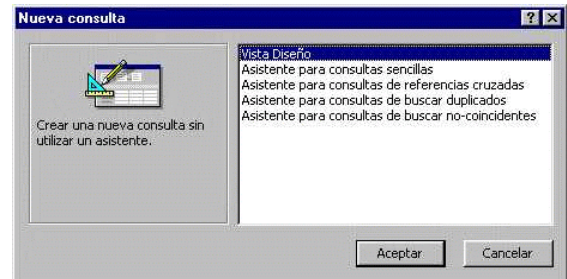


Figura 2: Opciones para crear una nueva consulta

Vamos a ilustrar los pasos básicos a seguir para crear una consulta haciendo una muy simple sobre la Base de Datos Neptune: una consulta que nos de un listado de los clientes de la compañía.

Mostramos entonces la tabla de Clientes. En el editor QBE se representan las tablas como una ventana que contiene una lista de los campos disponibles, con un agregado: una línea que contiene el carácter “*”. Este carácter es especial y significa “todos los campos”; si lo agregamos a la lista de campos a mostrar, veremos lo mismo que si abrimos la tabla desde la ventana Base de Datos.

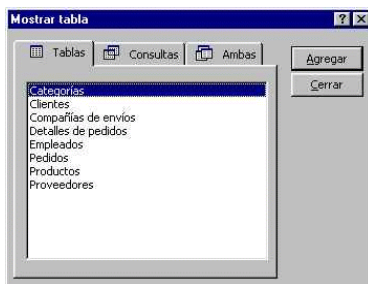


Figura 4: Ventana de selección de tablas para la consulta

Indicamos los campos que queremos ver arrastrándolos desde la

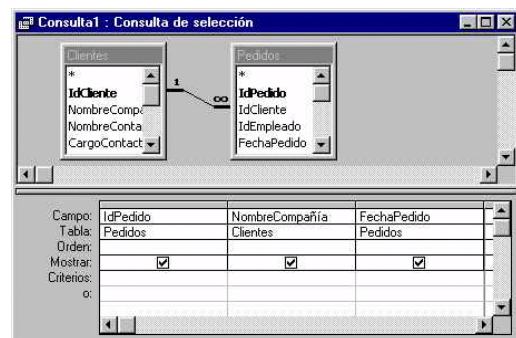



Figura 3: Editor QBE con dos tablas relacionadas

¹ Nota: podemos utilizar una tabla resultado de una consulta previa

ventana que representa la tabla hasta una columna de la grilla inferior (también se puede hacer una doble pulsación sobre el nombre del campo). Para ver todos los campos de la tabla arrastramos la línea que tiene el asterisco.

El caracter * en una consulta indica que se seleccionan *todos* los campos.

Ahora ejecutamos la consulta presionando el botón con un signo de admiración () o seleccionando la opción **Ejecutar** desde el menú **Consulta**. El resultado, como habíamos anticipado, es la tabla completa como si la hubiéramos abierto desde la ventana de Bases de Datos.

Todas las consultas en última instancia deben ser traducidas al lenguaje SQL para que el motor de acceso a los datos pueda procesarlas. El editor QBE de Access traduce la consulta a este lenguaje, y el resultado se puede ver con la opción **Vista SQL** del menú **Ver**². La sentencia SQL generada por nuestra consulta se puede ver en el cuadro 2

Podemos ver en estas líneas la consulta más simple posible. En lenguaje normal, esta consulta dice más o menos lo siguiente: “**SELECCIONAR** todos los campos *DE* la tabla **Cientes**”. Es por eso que estas consultas se denominan **Consultas de selección**.

```
SELECT Cientes.*  
FROM Cientes;
```

Listado 2: Consulta que devuelve todos los clientes

La sentencia **SELECT** indica que queremos seleccionar una serie de campos desde una o varias tablas o consultas. La sintaxis mínima es la siguiente:

```
SELECT <lista de campos>  
FROM <lista de tablas o consultas>
```

El punto y coma (;) final es opcional.

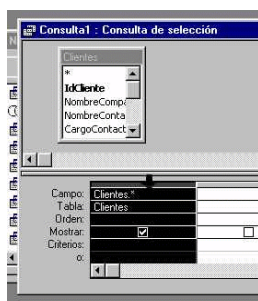


Figura 5: Seleccionar una columna

Modifiquemos entonces nuestra consulta en el editor QBE para ver los cambios en el SQL. Refinaremos nuestro pedido seleccionando ciertos campos de la tabla **Cientes**.

Debemos primero borrar la columna en la que está el signo * (ya que ahora no queremos todos los campos). Para ello basta con borrar el signo mismo, o bien seleccionar la columna completa haciendo click en la barra gris que separa la misma de la parte superior de la grilla (Fig. 5) y presionando **Delete**.

² También podemos escribir la consulta directamente en SQL y ver el equivalente en QBE seleccionando **Vista Diseño** del menú **Ver**.

Ahora seleccionamos los campos que deseamos que aparezcan en la tabla resultado. Para el ejemplo elegimos los campos *NombreCompañía*, *NombreContacto*, *CargoContacto*, *País* y *Ciudad*. La grilla QBE queda como en la Fig. 6

¿Cuál será el resultado de esta consulta? Trata de preverlo mentalmente antes de ejecutarla.

Tal cual esperábamos, el resultado es una tabla reducida con sólo las columnas que indicamos pero *todos* los registros. Enseguida veremos como restringir la cantidad de registros.

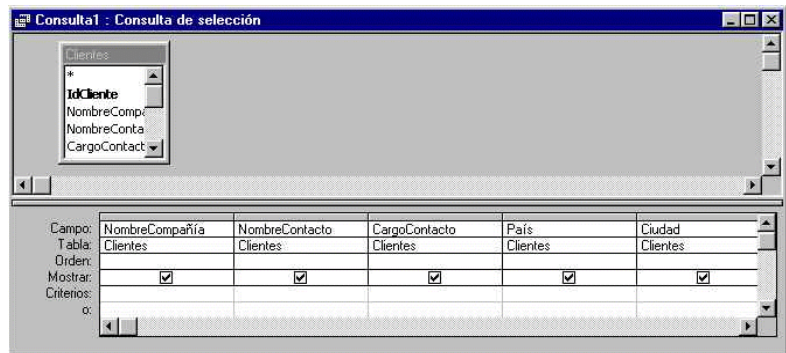


Figura 6: Consulta clientes modificada

Ahora, trata de deducir la sentencia SQL que ha generado Access.

Como podíamos imaginarnos, sólo cambia la *lista de campos* de la misma (Listado 3). En realidad, no siempre es necesario indicar para cada campo el nombre de la tabla separado con un punto; Access lo hace por simplicidad, ya que no afecta al resultado y a veces es indispensable (por ejemplo, cuando seleccionamos datos de dos tablas diferentes pero con el mismo nombre de campo).

```
SELECT Clientes.NombreCompañía,
Clientes.NombreContacto,
Clientes.CargoContacto, Clientes.País,
Clientes.Ciudad
FROM Clientes;
```

Listado 3: Consulta clientes modificada

Prueba a borrar el último campo de la selección directamente en el SQL; pasa luego a la vista diseño para ver el cambio en el editor QBE.

Vuelve a poner el campo Ciudad en la consulta, escribiéndolo sin el nombre de la tabla.

Campos calculados. Alias

También podemos *crear* nuevos campos que existirán solo en la tabla resultado de la consulta (normalmente serán columnas calculadas tomando valores de las otras columnas). Para ello nos posicionamos en una columna vacía de la grilla QBE y escribimos la expresión que queremos sea evaluada para cada registro. Por ejemplo, para crear una columna que contenga la ciudad y a continuación el código postal, del estilo “Madrid (28023)” escribimos la siguiente expresión en una

columna vacía en la línea que contiene los nombres de los campos:

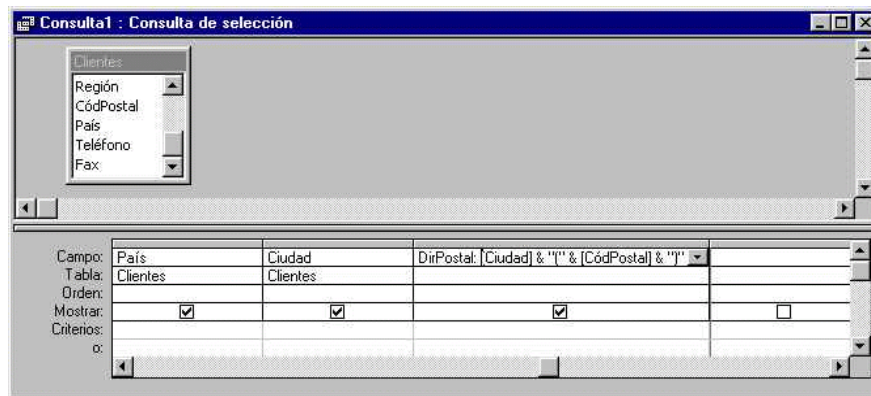


Figura 7: Creación de un campo calculado

Veamos el SQL generado por Access (Listado 5). Notamos enseguida la utilización de otra palabra clave (en mayúsculas): “AS”. Esta partícula se utiliza en SQL para asignar un nombre nuevo a una columna -un *alias*. En este caso la columna está formada por una combinación de campos y cadenas de caracteres. Los campos se referencian normalmente entre corchetes ([]), aunque sólo es estrictamente necesario cuando el nombre de un campo contiene caracteres especiales tales como espacios o ciertos símbolos.

```
SELECT Clientes.NombreCompañía,
Clientes.NombreContacto,
Clientes.CargoContacto, Clientes.País,
Clientes.Ciudad, [Ciudad] & "(" & [CódPostal]
& ")" AS DirPostal
FROM Clientes;
```

Listado 4: Consulta con un campo calculado

Ejercicio 1-1

Antes de continuar, crea una nueva consulta con la tabla “Detalles de Pedidos” en la que se deben ver los siguientes campos: *IdPedido*, *IdProducto*, *PrecioUnidad*, *Cantidad*, *Descuento*, *PrecioUnidad*Cantidad*(1-Descuento)*. Llama a la última columna *Subtotal*. Llama a la consulta “Detalle con subtotal”.

Según las reglas de normalización, las columnas que se pueden calcular en base a los valores de otras columnas no deberían incluirse en la tabla; deberían calcularse “al vuelo” creando una columna calculada

como la anterior en una consulta³.

Ordenando la salida

Para conseguir que la tabla resultado esté ordenada por algún campo (puede ser más de uno) debemos especificar en la grilla QBE el tipo de orden (ascendente o descendente) en los campos deseados. Ver Fig. 8.

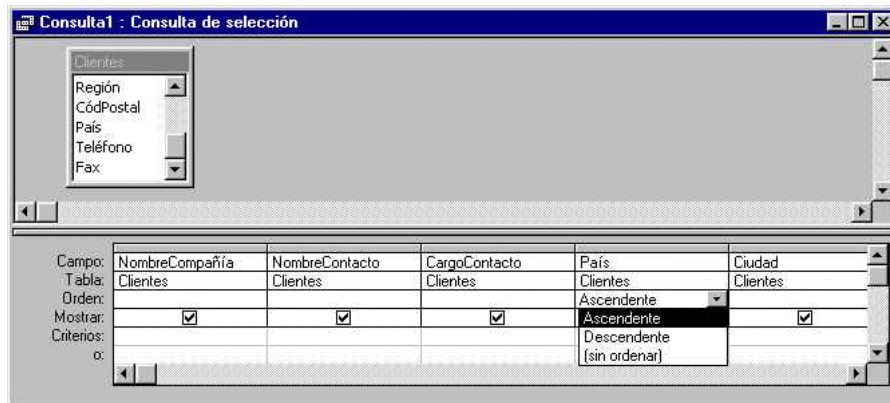


Figura 8: Ordenar la salida por un campo

En SQL, especificamos el orden con la cláusula ORDER BY:

```
SELECT Clientes.NombreCompañía, Clientes.NombreContacto, Clientes.CargoContacto,  
Clientes.País, Clientes.Ciudad, [Ciudad] & "(" & [CódPostal] & ")" AS DirPostal
```

```
FROM Clientes
```

```
ORDER BY Clientes.País;
```

En la cláusula ORDER BY se puede especificar más de un campo, tomándose los mismos de izquierda a derecha.

Ejercicio 1-2

Crear una consulta que muestre los datos de los clientes ordenados por País y Ciudad. Estos campos deben ser los primeros de la tabla resultado.

³ Claro que habrá casos en que será mejor incluir la columna calculada en la tabla física; las formas normales son sólo sugerencias, aunque en general conviene seguirlas.

Unión de tablas (combinaciones)

Siguiendo las sugerencias de la normalización, los datos de los pedidos están divididos en dos tablas: una con los datos generales de cada pedido y la otra con los datos de cada ítem de cada pedido (las líneas de un pedido escrito, cada línea sería un registro en la tabla de detalles).

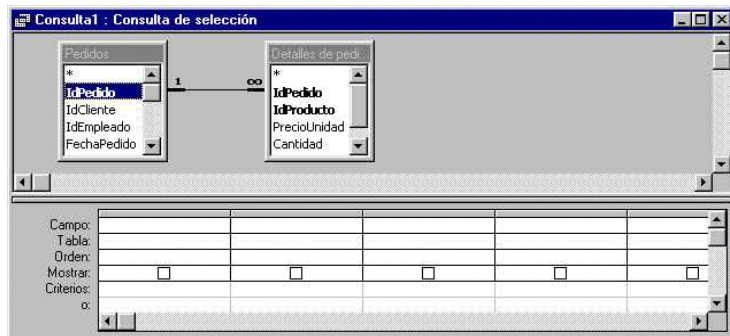


Figura 9: Consulta con dos tablas relacionadas

Deseamos ahora ver todos los datos de los pedidos, tanto los generales como el detalle de cada uno. Para ello agregaremos primero a la consulta la tabla “Pedidos” (opción **Mostrar tabla** del menú **Consulta**).

Notamos inmediatamente que se dibuja una línea entre las dos tablas, conectando los campos IdPedido de las mismas (Fig. 9).

Esta línea indica que hay una relación ya establecida entre las dos tablas -es decir, que los registros de una están relacionados con los de la otra de cierta manera. En este caso

la relación indica que cada registro de la tabla de detalle corresponde a uno de la tabla de pedidos.

Tipos de relaciones

En cada extremo de la línea de conexión Access dibuja símbolos indicando el tipo de relación. El “1” indica que el valor del campo correspondiente es único para cada registro de la tabla; el “4” indica que puede haber cero, uno o más registros con el mismo valor en esa tabla. Esta característica está dada por el tipo de índice que establecemos al definir las tablas: si indicamos que queremos un índice que no admita duplicados en un registro, nos aseguramos que cada registro tendrá un valor diferente en ese campo, y en las relaciones siempre le corresponderá un “1”. En cambio, si el campo forma parte de un índice que admite duplicados o un índice compuesto, le corresponderá el “4”.

Además, podemos especificar la manera en que se mostrarán los registros con campos combinados de las dos tablas: si pulsamos el ratón en la línea de relación ésta se pone más gruesa indicando que está seleccionada y con el botón derecho del ratón

Estos símbolos *no indican la cardinalidad* de la relación; únicamente indican la manera de crear los registros de la tabla resultado.

accedemos a un menú contextual que nos permite cambiar las características de la relación.

Las opciones de que disponemos están indicadas claramente en la caja de diálogo de propiedades de la combinación (Fig. 11)

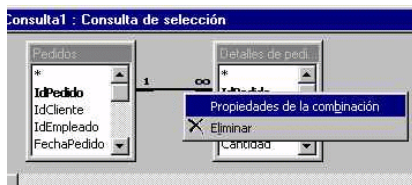


Figura 10: Menú contextual para cambiar el tipo de relación

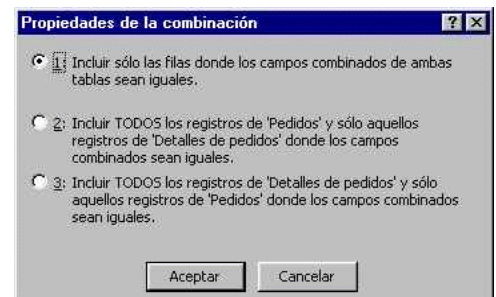


Figura 11: tipos de combinaciones

Podemos ver el efecto del tipo de combinación en la selección agregando un registro “huérfano” en la tabla de Pedidos; es decir, un registro con un

valor de IdPedido que no exista en la tabla de Detalles de Pedidos. Con los registros establecidos de esta manera, si seleccionamos la opción 1 (mostrar sólo los registros que coincidan en ambas tablas) no aparecerán los datos del pedido nuevo porque no hay datos correspondientes en la otra tabla. De la misma manera, la opción 3 (mostrar todos los registros de la tabla “Detalles de Pedidos” y sólo los que coincidan de la tabla de “Pedidos”) tampoco mostrará los datos nuevos porque no hay ninguno en la tabla de Detalles. La única opción que nos mostrará los datos del pedido -con los campos de detalle en blanco- es la opción 2 (mostrar todos los registros de la tabla “Pedidos” y sólo aquellos que coincidan de la tabla “Detalles de Pedidos”).

Cuando vemos el SQL generado por la consulta anterior (listado 6) encontramos la sintaxis de relaciones entre tablas estandarizada en ANSI SQL 92 con el uso de la partícula “JOIN” en la cláusula FROM.

```
SELECT Pedidos.IdPedido, Pedidos.FechaPedido, [Detalles de pedidos].IdProducto, [Detalles de
pedidos].PrecioUnidad, [Detalles de pedidos].Cantidad, [Detalles de pedidos].Descuento
FROM Pedidos INNER JOIN [Detalles de pedidos] ON Pedidos.IdPedido = [Detalles de
pedidos].IdPedido;
```

Listado 6: Relación entre dos tablas, estilo SQL-92

La sintaxis general es como sigue (los corchetes indican que esa parte es opcional, mientras que las diferentes opciones posibles se muestran entre llaves separadas por una barra vertical):

```
FROM tabla1 {INNER | LEFT [OUTER] | RIGHT [OUTER]} JOIN tabla2 ON tabla1.columna1 =
tabla2.columna2
```

Hay tres tipos de relaciones, coincidentes con las tres opciones que tenemos en el cuadro de diálogo de propiedades de la combinación:

INNER JOIN: sólo entran en el resultado los registros en los cuales coinciden los campos de la unión (deben existir en las dos tablas, es equivalente a la opción 1 de las propiedades de la combinación)

LEFT OUTER JOIN: se muestran todos los registros de la tabla de la izquierda (tabla1); si hay valores coincidentes en los campos de la unión de la segunda tabla se muestran, caso contrario se dejan en blanco. La tabla a la izquierda de la palabra JOIN se denomina *tabla preservada* y la de la derecha *no preservada*.

RIGHT OUTER JOIN: se muestran todos los registros de la tabla de la derecha (tabla2); si hay valores coincidentes en los campos de la unión de la primera tabla se muestran, caso contrario se dejan en blanco. La tabla a la derecha de la palabra JOIN se denomina *tabla preservada* y la de la izquierda *no preservada*.

Las dos últimas formas de combinación tienen también su equivalente en el cuadro de Propiedades de la combinación, pero el nro (2 o 3) depende de cómo fue hecha la combinación. Por esta razón en el cuadro

está explicado utilizando los nombres de las tablas.

Hay otra manera de especificar una combinación entre tablas: la que se utilizaba antes de la especificación ANSI 92, indicando la condición de combinación en la cláusula WHERE. Veamos el equivalente a la consulta anterior (Listado 7).

```
SELECT Pedidos.IdPedido, Pedidos.FechaPedido, [Detalles de pedidos].IdProducto, [Detalles de pedidos].PrecioUnidad, [Detalles de pedidos].Cantidad, [Detalles de pedidos].Descuento  
FROM Pedidos, [Detalles de pedidos]  
WHERE Pedidos.IdPedido = [Detalles de pedidos].IdPedido;
```

Listado 7: Relación entre dos tablas, estilo SQL-89

Esta forma de especificar una combinación es normalmente más simple de escribir y leer, pero no provee la capacidad de *combinaciones externas* que provee el SQL-92; además, las consultas escritas siguiendo el SQL89 son siempre sólo de lectura -restricción importante que no se aplica a la nueva sintaxis.

El editor QBE de Access *siempre* genera SQL con la sintaxis ANSI 92, aunque soporta el viejo formato también. La mayoría de los servidores de bases de datos de la actualidad se encuentran en la misma situación.

Combinaciones múltiples

Hay ocasiones en que se hace necesario utilizar más de dos tablas en una consulta. Por ejemplo, si queremos extender la consulta anterior de los detalles de los pedidos para mostrar también el Nombre del producto junto con el precio unitario y la cantidad, debemos agregar a la consulta la tabla de *Productos*. Esta tabla está enlazada con la de *Detalles de Pedidos* a través del campo **IdProducto**, como podemos ver en el editor QBE (Fig. 12)

Si miramos el SQL generado por Access, veremos la forma de especificar una combinación múltiple en SQL (Listado 8):



Figura 12: combinación múltiple

```
SELECT Pedidos.IdPedido, Pedidos.FechaPedido, [Detalles de pedidos].IdProducto, [Detalles de pedidos].PrecioUnidad, [Detalles de pedidos].Cantidad, [Detalles de pedidos].Descuento  
FROM Productos INNER JOIN (Pedidos INNER JOIN [Detalles de pedidos] ON Pedidos.IdPedido = [Detalles de pedidos].IdPedido) ON Productos.IdProducto = [Detalles de pedidos].IdProducto;
```

Listado 8: Combinación múltiple

Asusta un poco, ¿no? Las combinaciones se *anidan* en la cláusula FROM. La sintaxis general es la siguiente:

```
FROM (...(tabla1 JOIN tabla2 ON condicion1) JOIN tabla3 ON condicion3) JOIN...)
```

Claro que hay que reemplazar la palabra JOIN por la forma completa, utilizando INNER, LEFT o RIGHT.

En las combinaciones externas *el orden tiene importancia*. Con la sintaxis anterior especificamos claramente el orden en que se deben formar las combinaciones... en la mayoría de los servidores de bases de datos, *pero no en Access*. Esto es muy particular, ya que el procesador de consultas de Access reordena las combinaciones y las ejecuta en el orden que más le convenga. Debido a esto, hay un par de reglas que hay que tener en cuenta cuando escribimos combinaciones múltiples en Access:

- 1) La tabla no preservada de una combinación externa no puede participar en una combinación interna.
- 2) la tabla no preservada de una combinación externa no puede ser la tabla no preservada en otra combinación externa.

Veremos a continuación una serie de ejemplos. Recomendamos leer el enunciado de cada problema y tratar de escribir la consulta solos, antes de ver la respuesta.

Ejemplos:

Los siguientes ejemplos utilizan las tablas de la Base de Datos Neptuno que viene con Access.

1. mostrar los items de las ordenes de pedido.

SQL 89:

```
select *
from pedidos, [detalles de pedidos]
where pedidos.idpedido=[detalles de pedidos].idpedido
```

SQL 92:

```
select *
from pedidos inner join [detalles de pedidos] on pedidos.idpedido = [detalles de
pedidos].idpedido
```



Figura 13: Ejemplo 1

¿Qué ocurrirá si tenemos un registro en la tabla de Pedidos que no se corresponda con ninguno de la tabla de Detalles de Pedidos (si por ejemplo, alguien inicia un pedido pero antes de ingresar el detalle tiene que hacer una pausa)? Los datos de ese pedido inconcluso no aparecerán, porque la unión anterior -en las dos formas- es una unión *interna*. Esto implica que sólo aparecerán los datos de los pedidos para los cuales existan registros en la tabla de Detalles.

Para lograr que aparezcan los datos de ese pedido sin el detalle, debemos hacer una unión externa -y para esto utilizar la sintaxis SQL-92:

```
select *
from pedidos left join [detalles de pedidos] on [detalles de pedidos].idpedido =
pedidos.idpedido
```

NOTA: para cambiar el tipo de combinación en el editor QBE debemos invocar las propiedades de la combinación; no obstante, no veremos cambios en la representación gráfica.

< Catálogo de productos, mostrando datos del producto y de la categoría a que pertenece.

```
SELECT Categorías.NombreCategoría, Categorías.Descripción, Productos.IdProducto,
Productos.NombreProducto, Productos.CantidadPorUnidad, Productos.PrecioUnidad
FROM Categorías INNER JOIN Productos ON Categorías.IdCategoría = Productos.IdCategoría
```

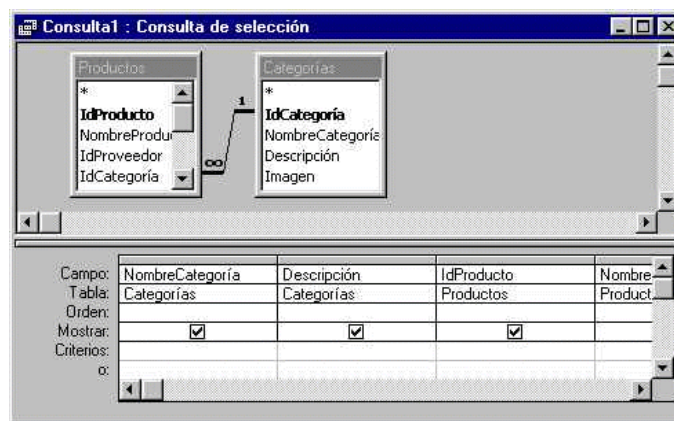


Figura 14: Ejemplo 2

< Detalle de pedidos con subtotales (campo calculado)

```
select pedidos.idpedido,pedidos.fechapedido as [Fecha de pedido], cantidad, preciounidad,
descuento, cantidad*preciounidad*(1-descuento) as subtotal from pedidos left join
[detalles de pedidos] on pedidos.idpedido=[detalles de pedidos].idpedido
```

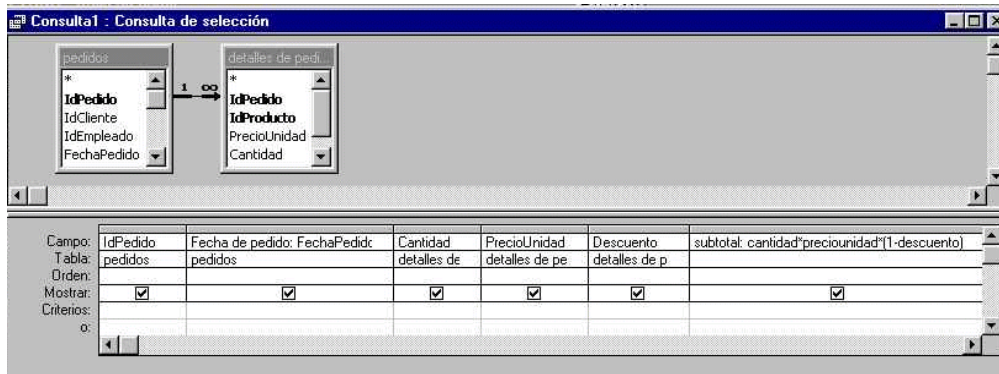


Figura 15: Ejemplo 3

< Agregar una restricción de fechas (entre el 4 de Agosto de 1994 y el 20 del mismo mes) a la consulta anterior

```
SELECT pedidos.idpedido,pedidos.fechapedido AS [Fecha de pedido], cantidad, preciounidad, descuento, cantidad*preciounidad*(1-descuento) AS subtotal
FROM pedidos LEFT JOIN [detalles de pedidos] ON pedidos.idpedido = [detalles de pedidos].idpedido
WHERE fechapedido BETWEEN #4/8/1994# AND #20/8/1994#
```

NOTA: el formato de las fechas no depende de la configuración del sistema operativo, sino de la configuración del servidor de datos. En particular en Access 97 podemos cambiarlo con el menú **Herramientas|Opciones|General|Nuevo orden de la base de datos.**

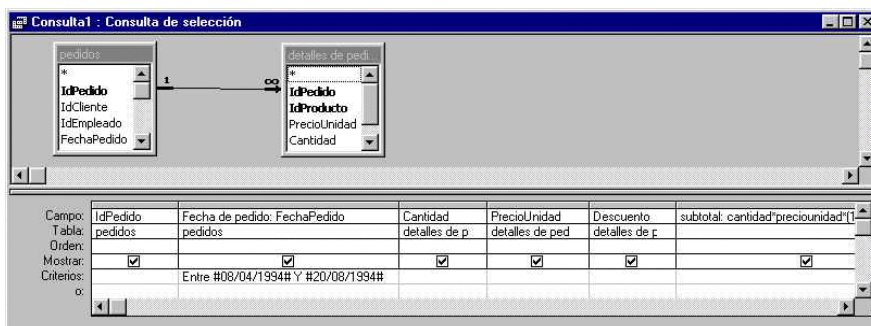


Figura 16: Ejemplo 4

< Join anidado. Crear una consulta que muestre el detalle de los pedidos con subtotales y descripción de productos. Necesitamos tres tablas: Pedidos, Detalles de Pedidos y Productos. Las dos últimas se enlazan con la primera por los correspondientes campos de ID.

Este tipo de enlaces múltiples se indican como *combinaciones anidadas*. En Access, estas combinaciones se ejecutan en el orden que más convenga, elegido por el motor de consultas. Por lo tanto, si cambiamos el orden de las combinaciones en la instrucción siguiente obtenemos el mismo resultado.

```

SELECT Pedidos.IdPedido, Pedidos.FechaPedido, [Detalles de pedidos].IdProducto,
Productos.NombreProducto, [Detalles de pedidos].PrecioUnidad, [Detalles de
pedidos].Cantidad, [Detalles de pedidos].Descuento, CCur([Detalles de
pedidos].[PrecioUnidad]*[Cantidad]*(1-[Descuento])/100)*100 AS PrecioConDescuento
FROM Productos INNER JOIN (Pedidos INNER JOIN [Detalles de pedidos] ON Pedidos.IdPedido =
[Detalles de pedidos].IdPedido) ON Productos.IdProducto = [Detalles de
pedidos].IdProducto ORDER BY Pedidos.IdPedido;

```

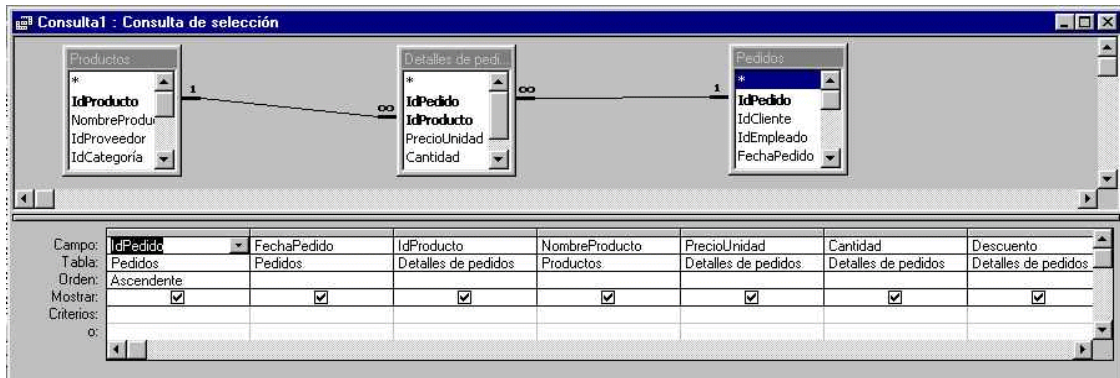


Figura 17: Ejemplo 5

Consultas de agregación

Las consultas de agregación construyen una o varias columnas aplicando funciones a los datos existentes. Por ejemplo, una función típica es la *suma* de los valores de una columna; podemos crear una consulta que nos devuelva la suma de los subtotales de todos los pedidos de la siguiente manera:

```

SELECT SUM(cantidad*preciounidad*(1-descuento)) AS Total
FROM [detalles de pedidos]

```

Obtenemos entonces un solo número como resultado, que representaría el total de dinero que recaudamos (Fig. 18)

Total
1265793.03828717

Figura 18: total de dinero de todos los pedidos

Otra posibilidad que nos da SQL es la de calcular totales *por grupos*. Por ejemplo, la consulta anterior podría agruparse por pedido, obteniendo el total de cada pedido, de la siguiente manera:

```

SELECT IdPedido, SUM(cantidad*preciounidad*(1-descuento)) AS
Total
FROM [detalles de pedidos]
GROUP BY IdPedido

```

Obtenemos un registro para cada pedido, con el ID y el total de dinero (Fig. 19).

IdPedido	Total
10248	440
10249	1863,4
10250	1552,59998965263
10251	654,059999750555
10252	3597,89999803156
10253	1444,8
10254	556,619997274876

Figura 19: Totales por pedido

NOTA: Cada campo especificado en el Select que no esté afectado por una función de agregación debe figurar en el GROUP BY. Cada grupo va desde un valor de *todos* los campos especificados en GROUP BY hasta que *alguno* de estos campos cambia. Por lo tanto, si tenemos por ejemplo

```
SELECT IdPedido,Cantidad, SUM(cantidad*preciounidad*(1-
descuento)) AS Total
FROM [detalles de pedidos]
GROUP BY IdPedido,Cantidad
```

IdPedido	Cantidad	Total
10248	5	174
10249	10	98
10249	12	168
10249	9	167.4
10249	40	1696
10250	10	77
10250	15	214.199998497963

Figura 20: totales por pedido Y cantidad

obtendremos un registro por cada *pedido* y *cantidad* diferentes. (Fig. 20)

Por lo tanto, hay que tener mucho cuidado con las columnas que incluimos en la consulta.

Se pueden aplicar criterios a la selección, utilizando WHERE igual que antes. Estos criterios se aplican a los registros *antes* de agrupar (se agrupan solo los registros que cumplen con los criterios). Por ejemplo, para agrupar sólo los pedidos entre el 10249 y el 10270 podemos hacer:

```
SELECT IdPedido, SUM(cantidad*preciounidad*(1-descuento)) AS Total
FROM [detalles de pedidos]
WHERE IdPedido BETWEEN 10249 AND 10270
GROUP BY IdPedido
```

Las funciones de agregación soportadas por Access son las siguientes:

(Todas las funciones toman una columna como parámetro entre paréntesis, salvo la indicada)

Función	Propósito
AVG (col)	Valor medio de los valores no nulos de la columna
COUNT (col)	Cuenta del número de valores no nulos de la columna
COUNT (*)	Cuenta el total de filas de la tabla, incluyendo filas con valores nulos
SUM (col)	Suma de los valores no nulos de la columna
MIN (col)	Valor más pequeño (no nulo) de la columna
MAX (col)	Valor más grande (no nulo) de la columna
FIRST (col)	Valor de la columna en el primer registro de la tabla resultado. Puede ser nulo
LAST (col)	Valor de la columna en el último registro de la tabla resultado. Puede ser nulo
STDEV (col)	Desviación estándar de muestra (n-1) para la columna, excluyendo nulos
STDEVP (col)	Desviación estándar de población (n), excluyendo nulos.

Función	Propósito
VAR (col)	Varianza de muestra para la columna, excluyendo nulos. VAR=STDEV ²
VARP (col)	Varianza de población, excluyendo nulos

Seleccionar los registros ya agrupados: la sentencia HAVING

Podemos agregar un refinamiento más a nuestra consulta de agrupación seleccionando sólo aquellos grupos que cumplan determinado requisito *después de la agrupación*. Es decir, sería como una condición WHERE pero que se aplicaría a los grupos, no a los registros individuales.

Por ejemplo, en la última consulta podemos pedir sólo aquellos pedidos en los cuales el total sea mayor que una determinada cantidad:

```
SELECT IdPedido, SUM(cantidad*preciounidad*(1-descuento)) AS Total
FROM [detalles de pedidos]
WHERE IdPedido BETWEEN 10249 AND 10270
GROUP BY IdPedido
HAVING SUM(cantidad*preciounidad*(1-descuento))>200
```

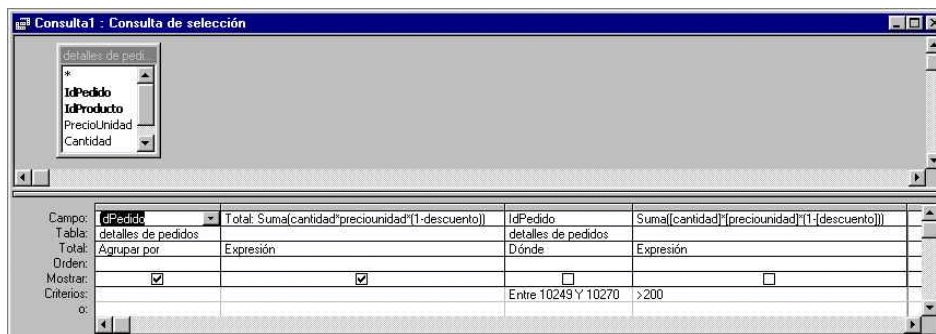


Figura 21: Aplicar criterios a los grupos

Como podemos ver en el editor QBE, el criterio está escrito en una columna extra donde se repite la expresión del campo calculado. Esta columna tiene desactivada la opción de *Mostrar*, ya que sólo se utiliza para especificar un criterio. Lo mismo sucede con la columna donde se expresa el criterio de selección de los pedidos (en el SQL, este criterio se escribe en la cláusula WHERE).

Aunque la cláusula HAVING nos da más posibilidades, conviene no utilizarla a menos que no haya otra opción porque hace que la operación se ponga muy lenta.

Consultas de acción

Las consultas que hemos realizado hasta ahora no modifican la Base de Datos; únicamente crean tablas virtuales -que se destruyen cuando cerramos la consulta.

Ahora veremos las instrucciones del lenguaje SQL que nos permiten modificar la base de datos: crear tablas e índices, agregar, borrar y modificar registros, etc.

En Access las consultas son *de Selección* por defecto. Podemos transformar cualquier consulta de un tipo a otro, y eventualmente Access nos pedirá algún otro dato que sea necesario (por ejemplo, el nombre de la tabla destino). Los comandos para cambiar el tipo de consulta están en el menú **Consulta**.

Modificar registros existentes (consulta de actualización)

La sentencia *UPDATE* nos permite modificar un grupo de registros existente, cambiando el valor de alguno de los campos. La sintaxis general es la siguiente:

```
UPDATE <tabla>
SET columna1=expresion1 [, columna2=expresion2,...]
[WHERE criterios]
```



Figura 22: Consulta de actualización

Por ejemplo, si queremos aumentar el precio de los productos de la categoría 1 en un 10% podríamos hacer:

```
UPDATE productos
SET PrecioUnidad=PrecioUnidad*1.1
WHERE IDCategoría=1
```

Ahora supongamos que cambiamos de proveedor para ciertos productos (por ejemplo, los del proveedor 1 ahora los obtenemos del 11), porque los precios que conseguimos son un 10% más baratos⁴:

```
UPDATE productos
SET PrecioUnidad=PrecioUnidad*0.90, IdProveedor=11
WHERE IdProveedor=1
```



Figura 23: cambiar dos campos a la vez

⁴ No, nadie dijo nada de la calidad!

donde estamos cambiando dos columnas a la vez.

Eliminar registros (Consulta de eliminación)

Para eliminar registros utilizamos la sentencia DELETE:

```
DELETE [table.*]
FROM tabla
[WHERE criterios]
```

Notemos que es *opcional* la especificación de los campos luego de la palabra DELETE; siempre se borran registros completos -nunca campos aislados- por lo que no importa lo que indiquemos. Las órdenes siguientes son equivalentes en el resultado:

```
DELETE IdCategoría FROM productos WHERE IdCategoría=1
```

```
DELETE IdProveedor FROM productos WHERE IdCategoría=1
```

```
DELETE FROM productos WHERE IdCategoría=1
```

```
DELETE * FROM productos WHERE IdCategoría=1
```

Pregunta: ¿Qué sucederá si no especificamos un criterio con WHERE?

Insertar nuevos registros (Consulta de datos anexados)

Para agregar valores a una tabla tenemos dos opciones: agregar valores obtenidos a través de una consulta, o bien agregar valores conocidos de antemano -por ejemplo, ingresados por el usuario a través del teclado. Las dos son variaciones de la sentencia *INSERT INTO*.

La primera forma toma los datos resultado de una consulta de selección y los agrega en la tabla destino:

```
INSERT INTO tabla
<orden select>
```

Por ejemplo, asumiendo que tenemos una tabla con la misma estructura que Productos llamada Productos2,

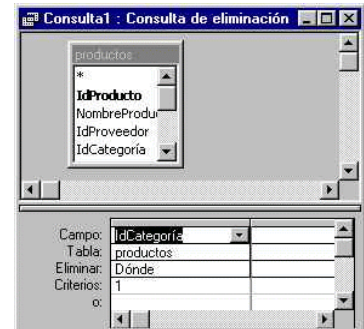


Figura 24: consulta de eliminación

podemos copiar todos los productos de la categoría 2 de Productos a Productos2 con la siguiente sentencia:

```
INSERT INTO productos2
SELECT IdProducto, NombreProducto, IdProveedor, IdCategoría
FROM productos WHERE IdCategoría=2
```

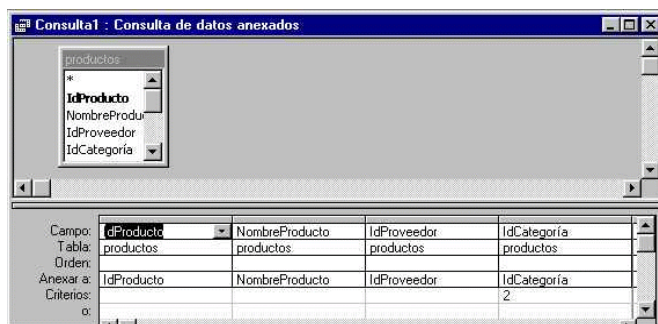


Figura 25: Consulta de datos anexados

Notemos que no estamos copiando todos los campos, sólo algunos. Los demás toman los valores por defecto que se les asignaron al crear la tabla Productos2.

La segunda forma de la sentencia INSERT INTO permite ingresar valores específicos a la tabla:

```
INSERT INTO tabla [(lista de columnas)]
VALUES (lista de valores)
```

Por ejemplo, si queremos agregar un cliente con algunos datos ingresados por el usuario, la sentencia a utilizar podría ser algo como lo siguiente:

```
INSERT INTO Clientes
VALUES ("abcde", "Compañía nueva", "Contacto principal",
"el cargo del contacto", "la dirección de la cía.",
"la ciudad", "la región", "el CP", "el país", "el telefono",
"el fax")
```

Notemos que tenemos que dar valores a *todos* los campos porque no hemos especificado cuáles son los que queremos modificar. Si queremos dar valor sólo a algunos de los campos, podemos hacer

```
INSERT INTO Clientes (IdCliente, NombreCompañía, País)
VALUES ("nadie", "Otra Compañía nueva", "China")
```

Los demás campos toman los valores por defecto. Tenemos que tener en cuenta las posibles restricciones al

contenido de los campos, por ejemplo debemos dar un valor al campo o campos de la clave principal que además debe ser único. Asimismo, los tipos de datos deben coincidir o se generará un error.

Esta forma de insertar registros no está soportada por el editor QBE, por lo que Access nos muestra el siguiente mensaje:

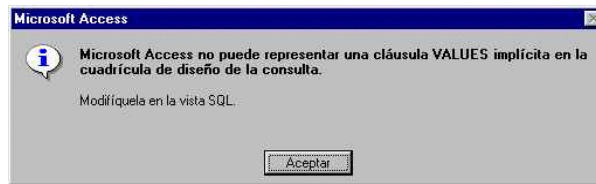


Figura 26: No se puede representar una sentencia INSERT INTO explícita en el editor QBE

No obstante, podemos ejecutarla igualmente desde la vista SQL.

Hasta aquí con los comandos que trabajan con los datos; esta categoría de sentencias SQL se denomina Lenguaje de Manipulación de Datos (Data Manipulation Language, DML). La otra gran categoría de sentencias se denomina Lenguaje de Definición de Datos (Data Definition Language, DDL) y consta de instrucciones SQL para crear y modificar objetos de la base de datos tales como tablas, índices, vistas, procedimientos, etc.

2 - SQL - 2da. Parte

Lenguaje de Definición de Datos (DDL)

El Lenguaje de Definición de Datos es un subconjunto de SQL con instrucciones para crear, modificar y borrar los distintos objetos que componen la Base de Datos como ser tablas, índices, dominios, etc. Incluso permiten la definición de Bases de Datos completas.

Nota: en todos los listados de sintaxis que figuran en esta sección, se utilizan algunos símbolos con significados especiales (no se deben incluir en la sentencia):

- ! Corchetes “[]” : sección opcional. No es indispensable para la correcta declaración de la sentencia
 - ! Barra vertical “|” : separa opciones mutuamente excluyentes. Se debe utilizar una sola de ellas.
 - ! Llaves “{ }” : agrupa opciones excluyentes.
-

Bases de Datos

Para crear una base de datos se utiliza la sentencia CREATE DATABASE. En Interbase 5, la sintaxis es:

```
CREATE DATABASE 'nombre' [USER 'usuario' [PASSWORD 'clave_de_ingreso']]
```

En la utilidad Windows Interactive SQL de Interbase, la creación de Bases de Datos se hace a través de la opción de menú “Create Database”. Se crea un archivo físico con el nombre indicado, que puede contener el camino completo.

Para eliminar una base de datos (eliminando el archivo físico) se utiliza

```
DROP DATABASE  
sin parámetros.
```

Se puede también alterar la definición de la Base de Datos; consultar la documentación del manejador particular que se utilice.

Tablas

La creación de tablas se hace con la sentencia

```
CREATE TABLE
```

, que en general tiene la siguiente sintaxis:

```
CREATE TABLE nombre_tabla  
( <def_col> [, <def_col> | <restricción_de_tabla> ...] );
```

La sentencia espera dos parámetros al menos: el nombre de la tabla (nombre_tabla

) y la lista de los campos o columnas (<def_col>

) entre paréntesis, separadas con comas. Se debe especificar al menos una columna.

Las definiciones de las columnas tienen la sintaxis general siguiente en Interbase 5:

```
<def_col> = nombre_columna {< tipo_de_dato> | COMPUTED [BY] (< expr>) | domain}  
[DEFAULT { literal | NULL | USER}]  
[NOT NULL]  
[ <restriccion_de_columna>]
```

donde nombre_columna es el nombre de la columna, que debe ir seguido de una de las tres opciones

1. tipo_de_dato: indica el tipo de dato que almacenará la columna (listados abajo)
2. COMPUTED BY <expresion>: define que la columna tiene un valor que se calcula evaluando la expresión cada vez. La expresión puede involucrar otros campos de la tabla; un ejemplo típico es el de los ítems de una factura donde tenemos los campos Cantidad y PrecioUnitario con los que podemos calcular el Subtotal. Para este ejemplo la definición

del campo Subtotal sería la siguiente:

Subtotal COMPUTED BY (Cantidad*PrecioUnitario)

3. domain: tipo de dato definido por el usuario. La creación de estos tipos de datos o dominios se verá luego.

Hasta aquí la parte obligatoria de la definición de un campo o columna. A continuación se pueden especificar datos opcionales:

4. Valor por defecto. Agregando la partícula

DEFAULT

seguida por una de las opciones:

- a. literal: un valor literal que se tomará como valor por defecto del campo. Por ejemplo

DEFAULT 0.

- b. NULL: valor por defecto nulo.

- c. USER: valor por defecto = nombre del usuario. Utilizado con fines administrativos.

5. Campo requerido. Con la partícula NOT NULL indicamos que el campo no puede tener valor nulo, es decir, que se requiere un valor concreto en el campo para que sea aceptado el registro.

6. Restricciones de columna. Podemos indicar directamente algunas restricciones para los valores de esta columna, como por ejemplo que referencia a otra columna de otra tabla (integridad referencial) o que el valor debe estar en un rango dado, etc. La sintaxis se verá luego.

Tipos de datos

Los tipos de datos indican el *dominio* de la columna, es decir, si en la misma se espera un número entero o real, una cadena de caracteres, etc. Los tipos de datos soportados por Interbase 5 son los siguientes:

Numéricos¹

SMALLINT

: entero pequeño con signo

INTEGER

: entero con signo

¹ El tamaño de cada tipo así como la precisión es dependiente de la plataforma. En las plataformas Intel, por ejemplo, el tipo

SMALLINT

es de dos bytes y el

INTEGER

de cuatro.

FLOAT

: real de precisión simple

DOUBLE PRECISION

: real de doble precisión

{DECIMAL | NUMERIC} [(precision [, scale])]

: real de precisión variable. Se almacenan

precision

dígitos, con

scale

decimales. Por ejemplo,

DECIMAL (5,2)

serían números de la forma

ppp.ss

Fecha/Hora

DATE

: fecha y hora

Caracteres²

{CHAR | CHARACTER} (long)

: cadena de caracteres de longitud fija. Entre paréntesis se especifica la cantidad de caracteres máxima que aceptará el campo.

{VARCHAR | CHARACTER VARYING | CHAR VARYING} (LONG)

: cadena de caracteres de longitud variable. Se indica entre paréntesis la longitud máxima que puede tomar el campo.

Restricciones

Las restricciones son condiciones que imponemos a los valores que pueden ser almacenados en una columna. Tienen la siguiente sintaxis general en Interbase 5:

```
<restriccion_de_columna> = [CONSTRAINT nombre_restr] <def_restr>
```

Es decir: la palabra reservada

CONSTRAINT

seguida por un nombre que asignamos a la condición (que debe ser único en la tabla) -todo esto opcional- y la definición de la restricción propiamente dicha.

² En Interbase 5, la longitud máxima un campo de caracteres es de 32767 bytes (la cantidad actual de caracteres depende del juego de caracteres particular utilizado). Para campos más grandes, Interbase permite el tipo no estándar

BLOB

.

Cuando asignamos condiciones a las columnas de la tabla podemos obviar la partícula CONSTRAINT

y el nombre; Interbase asignará un nombre único automáticamente.

Las restricciones que podemos especificar para las columnas son las siguientes:

```
<def_restr> = {UNIQUE | PRIMARY KEY  
| CHECK (<condicion>)  
| REFERENCES otra_tabla [( otra_col [, otra_col ...])]  
  [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]  
  [ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
```

1.

UNIQUE

especifica que el valor de la columna para cada registro debe ser único

2.

PRIMARY KEY

indica que la columna forma la clave primaria de la tabla (debe indicarse también

NOT NULL

)

3.

CHECK

(condicion) hace que se compruebe si el valor ingresado cumple con la condición impuesta; si no lo hace, no se permite la entrada del registro a la tabla.

4.

REFERENCES

indica que esta tabla (tabla secundaria o *detail*) está relacionada con otra (tabla principal o *master*). El campo en cuestión forma una clave externa que referencia un campo de la tabla principal, de tal manera que no puede haber un valor en el campo de la tabla detalle que no exista en el campo correspondiente de la tabla principal. Las demás especificaciones (opcionales) indican qué debe hacerse con los valores de la tabla secundaria si cambian o se borran los valores de la tabla principal:

a.

NO ACTION

: no se permite la modificación o borrado del valor en la tabla principal

b.

CASCADE

: se modifica el valor de la tabla detalle dejándolo igual al de la principal (para actualizaciones). Se borran los registros de la tabla detalle que contengan el valor del registro borrado en la tabla principal (para eliminaciones).

c.

SET DEFAULT

: se coloca el valor por defecto del campo en la tabla detalle.

d.

SET NULL

: se coloca el campo de la tabla detalle en

NULL

.

Las dos últimas opciones no son recomendables, ya que llevan a perder el enlace entre los registros de la tabla detalle y los de la principal.

Condición para CHECK

La condición de la restricción CHECK puede tomar muchas formas distintas, aunque normalmente será la comprobación de que la columna esté en un rango de valores. Veremos a continuación las opciones más utilizadas; son válidos aquí los siguientes símbolos:

1. <valor> puede ser el nombre de una columna, una expresión o una variable
2. el símbolo “%” se utiliza como comodín en la expresión LIKE para indicar “cualquier cosa”

3.
ALL
= todos,
SOME
= algunos,
ANY
= cualquiera

<valor> <operador> { <valor> | (<lista de valores>)}
por ej.,
CHECK (edad>0)

<valor> [NOT] BETWEEN <valor> AND <valor>

por ej.,
CHECK (edad BETWEEN 0 AND 100)

<valor> [NOT] LIKE <valor>

por ej.,
CHECK (titulo LIKE "part%")

<valor> [NOT] IN (lista_de_valores)

por ej.,
CHECK (tarjeta IN ("MASTERCARD", "VISA"))

<valor> IS [NOT] NULL

<valor> {[NOT] {= | < | >} | >= | <=} {ALL | SOME | ANY} (<lista_de_valores>)

<valor> [NOT] CONTAINING <valor>

<valor> [NOT] STARTING [WITH] <valor>

también se pueden encadenar condiciones utilizando operadores lógicos.

Restricciones de tabla

Además de las restricciones de columna, podemos especificar *restricciones de tabla*. Se definen igual que las columnas, al final de la declaración de la tabla. La sintaxis general es muy similar a la declaración de restricciones de columna:

<restricción_de_tabla> = CONSTRAINT nombre <def_de_restricción>

```
<def_de_restricción> = {{PRIMARY KEY | UNIQUE} ( columna [, columna ...])  
| FOREIGN KEY ( col [, col ...]) REFERENCES otra_tabla  
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]  
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]  
| CHECK (condicion)}
```

Notemos que esta es la única manera en que podemos definir una clave primaria compuesta por más de una columna, por ejemplo:

```
CONSTRAINT clavePrimaria PRIMARY KEY (Nombre,Apellido,Direccion)
```

Ejemplos

Veamos ahora algunos ejemplos completos de definición de tablas en Interbase 5.

1) Tabla de facturas

Queremos definir una tabla que guardará datos de facturas de un comercio hipotético, con la siguiente estructura:

Nombre de campo	Tipo de dato	Longitud
Nro_Factura	Entero	
Fecha	Fecha	
Cliente	Caracter	30
Dir_Cliente	Caracter	30
Forma_Pago	Caracter	10

Para hacerlo en SQL, ejecutamos la siguiente instrucción (ya sea en WISQL, el SQL Explorer, o un query):

```
CREATE TABLE Facturas  
(Nro_Factura INT NOT NULL PRIMARY KEY,  
Fecha DATE DEFAULT "TODAY",  
Cliente CHAR(30),  
Dir_Cliente CHAR(30),  
Forma_Pago CHAR(10));
```

Notemos la especificación de restricciones de columna: indicamos que el número de factura formará la clave primaria de la tabla (y por lo tanto no puede ser nulo), y que el campo “Fecha”

tomará como valor por defecto la fecha actual.

Ahora queremos crear otra tabla con los ítems de las facturas. La llamaremos items:

```
CREATE TABLE items
(ID_Item INT NOT NULL,
Factura INT NOT NULL,
Cantidad INT DEFAULT 1 NOT NULL,
Precio_Unitario NUMERIC(10,2),
Subtotal COMPUTED BY (Cantidad*Precio_Unitario),
CONSTRAINT ClavePrimaria PRIMARY KEY (ID_Item,Factura),
CONSTRAINT Relacion FOREIGN KEY (Factura) REFERENCES Facturas (Nro_Factura)
ON DELETE CASCADE
ON UPDATE CASCADE);
```

Ahora las cosas se empiezan a poner interesantes! Tenemos en esta tabla unas cuantas cosas para recalcar:

1. los campos ID_Item y Factura deben ser no nulos para que puedan formar parte de la clave primaria
2. la cantidad toma un valor por defecto 1 y a la vez nunca puede ser nula
3. El subtotal es un campo calculado como (cantidad*Precio unitario) de cada producto
4. La clave primaria se compone de dos campos, por lo tanto es necesario especificarla como una restricción de tabla que en este caso se llamará “ClavePrimaria”
5. El campo “Factura” referencia a la tabla de facturas, indica a qué factura pertenece el ítem. Con la segunda restricción de tabla creamos una relación entre las dos tablas indicando también las acciones para mantener la Integridad Referencial. En este caso, si el nro de la factura a que se refiere el ítem se cambia, también cambiará automáticamente el campo de todos los ítems que pertenecen a esa factura; y si se borra la factura, se borrarán automáticamente todos los ítems de la misma.

Todavía podemos refinar un poco esta tabla; específicamente, modificaremos el campo ID_Item para que tome un valor automáticamente -para asegurarnos que no se repita ninguno.

Restructurar tablas

Podemos alterar la definición de una tabla con el comando ALTER TABLE. La sintaxis general es la siguiente:

```
ALTER TABLE nombre_tabla {ADD {<def_col> | <restricción_de_tabla>} |
DROP {columna | CONSTRAINT nombre_restricción}}
```

Con

ADD

se agrega una columna o restricción de tabla; con

DROP

se eliminan de la estructura. Se pueden hacer varias operaciones a la vez listandolas separadas

por comas.

La sintaxis de definición de columnas o restricción de tabla es igual a la que se utiliza para el comando

```
CREATE TABLE
```

.

Notemos que para modificar la estructura de una columna hay que descartar primero la columna inicial y luego agregar la nueva.

Eliminar tablas

Por último, podemos eliminar una tabla completa de la Base de Datos utilizando DROP:

```
DROP TABLE nombre
```

Los datos se eliminan permanentemente. También se eliminan los índices y los triggers que correspondan a la tabla.

Utilización de las tablas desde Delphi

Las tablas se pueden utilizar desde Delphi en la forma habitual, es decir, con los componentes Ttable y Tquery. Preferentemente al trabajar con servidores SQL como Interbase se utilizarán componentes Tquery.

Dominios

Los dominios son especificaciones de valores posibles para los campos; algo así como los tipos de datos definidos por el usuario de Pascal.

```
CREATE DOMAIN nombre [AS] <tipo_de_datos>
[DEFAULT { literal | NULL | USER}]
[NOT NULL] [CHECK (<condición>)]
```

Donde

<tipo de datos>

es la especificación de un tipo de datos existente o más generalmente un subrango del mismo.

Veamos algunos ejemplos:

```
CREATE DOMAIN nroCliente
AS INTEGER
CHECK (VALUE > 1000);
```

esta sentencia crea un dominio llamado *nroCliente*, que referencia datos de tipo entero y mayores

que 1000. La cláusula

VALUE

se substituye por el valor del campo que utiliza este dominio.

Una vez definido un dominio podemos utilizarlo como un tipo de dato, por ejemplo en la definición de una tabla:

```
CREATE TABLE clientes
(nombre CHAR(30) NOT NULL,
numero nroCliente NOT NULL,
direccion CHAR(40));
```

De esta manera, el campo “numero” de la tabla “clientes” es de tipo *integer*, no puede contener valores menores o iguales a 1000 y no puede ser nulo.

En el siguiente ejemplo se crea un dominio que reemplazaría al tipo *boolean* de Pascal:

```
CREATE DOMAIN logico
AS INTEGER
CHECK (VALUE=0 OR VALUE=1)
```

Una gran ventaja de la utilización de dominios es que si cambian las especificaciones del dominio cambian automáticamente todos los campos que lo utilicen.

Se puede modificar la definición de un dominio con

ALTER DOMAIN

:

```
ALTER DOMAIN name {
[SET DEFAULT {literal | NULL | USER}]
| [DROP DEFAULT]
| [ADD [CONSTRAINT] CHECK (<condición>)]
| [DROP CONSTRAINT]
};
```

A esta altura ya deben ser capaces de interpretar por si mismos esta definición, ¿verdad?. En breve, se puede

6. asignar un nuevo valor por defecto (

SET DEFAULT

)

7. eliminar el actual valor por defecto (

DROP DEFAULT

)

8. agregar una restricción (

ADD CHECK

)

9. eliminar la restricción (
DROP CONSTRAINT
)

Por ejemplo, para poner un valor por defecto al dominio nroCliente creado anteriormente podríamos hacer:

```
ALTER DOMAIN nroCliente SET DEFAULT 9999;
```

Notemos que no podemos cambiar un dominio en el que se especifica la cláusula NOT NULL ; debemos eliminar el dominio y crearlo nuevamente.

¿Y cómo eliminamos un dominio? Claro, con DROP:

```
DROP DOMAIN nombre
```

Indices

Los índices son como las aspirinas: no son indispensables, pero... si se los tiene, mejor. Son objetos que optimizan el acceso a los datos por parte del manejador de la base de datos. De hecho, muchas veces el mismo manejador creará índices temporalmente para optimizar algunas acciones.

Para crear un índice permanente utilizamos... ¡SI!

```
CREATE INDEX
```

:

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]]  
INDEX nombre_indice ON nombre_tabla ( col [, col ...]);
```

Podemos crear índices ordenados en forma ascendente (

```
ASC
```

) o descendente (

```
DESC
```

) compuestos de de una o varias columnas, listadas entre paréntesis. El índice estará ligado a la tabla, de manera que si se elimina la misma (con

```
DROP TABLE
```

) también se eliminará cualquier índice asociado.

Los índices se utilizan automáticamente cuando se realizan consultas que involucren los campos del índice. Se puede definir un índice ascendente y otro descendente en la misma columna, y se

utilizará el que corresponda con la cláusula

`ORDER BY`

en las consultas.

Se puede especificar que los valores de las columnas que componen el índice sean únicos para cada registro, con la cláusula

`UNIQUE`

. Por supuesto, la definición de las columnas debe contemplar también el mismo caso o se producirá un error al crear el índice.

Lo único que podemos alterar de un índice es su disponibilidad; se puede desactivar un índice para ahorrar recursos del sistema si no lo vamos a utilizar, y luego volver a activarlo con

`ALTER INDEX`

:

```
ALTER INDEX nombre_indice {ACTIVE | INACTIVE};
```

Al reactivar un índice que estaba inactivo, el mismo se reconstruye y se balancea. Por lo tanto, se puede utilizar este procedimiento para rebalancear un índice. También es conveniente desactivarlo antes de hacer un gran número de modificaciones a los datos de la tabla, para acelerar el proceso; al reactivarlo se balanceará automáticamente.

NOTA: en Interbase 5, si un índice está en uso no tendrá efecto la sentencia

`ALTER`

hasta que se deje de utilizar.

Como habrán imaginado, para eliminar un índice podemos utilizar la sentencia

`DROP`

:

```
DROP INDEX nombre_indice;
```

Vistas (Views)

Las vistas son tablas virtuales creadas por sentencias

`SELECT`

precompiladas (más rápidas para ejecutarse); el usuario las tratará como si fueran tablas comunes. Para crearlas utilizamos

`CREATE VIEW`

:

```
CREATE VIEW nombre_vista [(columna [, columna ...])]
AS <select> [WITH CHECK OPTION];
```

podemos especificar los nombres de las columnas de la nueva tabla especificándolos entre

paréntesis; esto es obligatorio si las columnas son calculadas. Si no se indican los nombres explícitamente, se utilizan los de la consulta que la genera.

La sentencia

`<select>`

es una consulta, con la única restricción que no se puede utilizar la cláusula

`ORDER BY`

; la tabla resultado de la misma compone la vista.

Las vistas pueden ser modificables o no (sólo lectura). Una vista es modificable (se pueden agregar, modificar o borrar registros) si se cumplen los siguientes requisitos:

10. es un subconjunto de una sola tabla o de otra vista modificable

11. todas las columnas de la tabla que no entran en la vista permiten valores nulos

12. la sentencia

`SELECT`

no contiene subconsultas, funciones de agregación, el predicado

`DISTINCT`

, la cláusula

`HAVING`

, funciones definidas por el usuario o procedimientos almacenados.

Si la vista es modificable, se puede especificar la cláusula

`WITH CHECK OPTION`

, que previene la modificación de los datos de la vista si esta modificación no cumple con la cláusula

`WHERE`

de la consulta de selección -es decir, los registros modificados no entrarían en la vista. Esta opción no se utiliza por supuesto si la vista no es modificable.

No se puede modificar la definición de una vista una vez creada; únicamente podemos eliminarla completamente con

`DROP VIEW`

:

```
DROP VIEW nombre_vista;
```

Utilización desde Delphi

Las vistas son tablas virtuales, y se ven en Delphi como tablas comunes. Por lo tanto, podemos accederlas con los componentes Ttable o Tquery.

Triggers

Los triggers o gatillos son procedimientos que se asocian a determinadas acciones que se pueden realizar sobre una tabla; se ejecutan automáticamente *antes* (before) o *después* (after) de las acciones correspondientes.

Estos procedimientos se pueden asociar a las acciones INSERT, UPDATE y DELETE. Para crear

un trigger utilizamos CREATE TRIGGER:

```
CREATE TRIGGER nombre FOR nombre_tabla
[ACTIVE | INACTIVE]
{BEFORE | AFTER}
{DELETE | INSERT | UPDATE}
[POSITION numero_de_orden]
AS
[lista de variables, separadas por ";"]
BEGIN
  <cuerpo>
END <terminador
>
```

Como podemos ver, el trigger consta de un cuerpo entre BEGIN y END, como los procedimientos de pascal; sólo que aquí tenemos un lenguaje particular de cada gestor de Bases de Datos.

De particular interés para los triggers de inserción es la variable NEW: en un trigger de antes de insertar (BEFORE INSERT) representa al registro nuevo. Podemos acceder a los campos indicándolos separados por un punto, por ejemplo para poner valores por defecto:

```
NEW.Fecha = TODAY;
```

Esta sentencia, puesta en un gatillo Before Insert, pondrá en el campo Fecha de cada registro nuevo la fecha actual.

El terminador es un caracter especialmente designado como terminador; no podemos utilizar el punto y coma (terminador por defecto) porque se usa en la lista de variables y para terminar cada línea del cuerpo del trigger. Por lo tanto debemos especificar un nuevo terminador antes de definir el gatillo, con la sentencia

```
SET TERM
:
SET TERM terminador_nuevo terminador_viejo
```

Por ejemplo, para asignar el papel de terminador al caracter “!” podemos hacer

```
SET TERM !;
```

Y para volver al antiguo “;”, haríamos

```
SET TERM ;!
```

El trigger puede crearse inmediatamente activo o inicialmente inactivo. Además, podemos asignar más de un procedimiento a cada evento (entre 0 y 32767) en cuyo caso se ejecutarán en el

orden especificado por la cláusula

`POSITION`

(los más bajos primero). Por defecto se asigna la posición cero. Si hay más de un trigger con el mismo nro. de posición, se ejecutan en orden aleatorio.

Podemos alterar un trigger con la sentencia

`ALTER TRIGGER`

, que tiene la misma sintaxis que

`CREATE TRIGGER`

pero con el cuerpo opcional (por ejemplo, podríamos sólo cambiar la disponibilidad activando el trigger, o cambiar el número de posición).

Para eliminar un trigger utilizamos

`DROP TRIGGER nombre_trigger`

Los triggers son una forma muy utilizada de hacer cumplir ciertas reglas de negocio (business rules) directamente en la Base de Datos, cualquiera sea el método utilizado para acceder a los datos.

Utilización desde Delphi

Los triggers no se acceden desde Delphi; justamente, son procedimientos que corren en la Base de Datos sin intervención del programa cliente. El equivalente en Delphi serían los eventos de los componentes DataSet (Table, Query), por ejemplo BeforeInsert y AfterInsert. La diferencia está en que estos procedimientos se ejecutan en el cliente. Veremos las ventajas y desventajas de cada esquema cuando hablemos del modelo Cliente/Servidor.

Procedimientos Almacenados (Stored Procedures)

Los procedimientos almacenados son pequeños programas que se escriben en un lenguaje nativo de la Base de Datos. Permiten hacer muchas operaciones en una sola llamada, y al igual que los triggers se utilizan principalmente para implementar reglas de negocio en el servidor. Además, por el hecho de estar en la base de datos, logramos dos ventajas:

13. Lo único que pasamos a través de la red es el nombre del procedimiento, no los datos.
14. El proceso de los datos se hace en el servidor, que generalmente está mejor preparado para eso.

En Interbase, se utiliza el mismo lenguaje para los Procedimientos Almacenados que para los Triggers.

La definición de un Procedimiento se hace con... Create!

```

CREATE PROCEDURE nombre
[( parametro <tipo_dato> [, parametro <tipo_dato> ...])]
[RETURNS <tipo_dato> [, parametro <tipo_dato> ...])]
AS
[<lista_de_variiables_separadas_por_>]
BEGIN
    <cuerpo>
END [terminator]

```

Notas:

- ! El nombre debe ser único entre las tablas, procedimientos y vistas de la Base de Datos
- ! Se pueden especificar parámetros de entrada listándolos junto con sus tipos entre paréntesis luego del nombre.
- ! El procedimiento puede devolver uno o más valores, listados después de la cláusula RETURNS
- ! Para la especificación del terminador hay que seguir las mismas reglas que en los Triggers.

Para modificar un procedimiento, utilizamos como siempre ALTER. La sintaxis es exactamente la misma que para CREATE PROCEDURE, sólo cambiamos la palabra CREATE por ALTER.

Para eliminar un procedimiento almacenado de la Base de Datos utilizamos DROP:

```
DROP PROCEDURE nombre;
```

donde especificamos sólo el nombre del procedimiento.

Se puede utilizar un Procedimiento almacenado en una sentencia

```
SELECT
```

. El procedimiento debe devolver una fila (un registro) cada vez que es llamado, en variables de salida. La sentencia

```
SUSPEND
```

se utiliza dentro del procedimiento para devolver las variables por cada registro. Por ejemplo, escribiremos un procedimiento que nos devuelva todos los registros de la tabla

```
COUNTRY
```

de la base de datos

```
EMPLOYEE.GDB
```

. Este procedimiento nos devuelve tres parámetros, que serán los campos de una tabla virtual: "PAIS"

```
'
"MONEDA"
```

```
y
"CALIF"
```

. En las columnas "PAIS" y "MONEDA" vendrán los valores de la tabla, mientras que en la columna "CALIF" pondremos "EL MEJOR" si el país era "Argentina" y "NADA" en caso

contrario. Este procedimiento está pensado para ser llamado con una sentencia

```
SELECT
```

.

Para ingresar el procedimiento en la Base de Datos, debemos escribir un *script*. Es decir, todas las sentencias SQL que componen el procedimiento en un archivo de texto que será ejecutado a través del WISQL con la opción “Run an ISQL script” del menú “File”. He aquí el script:

```
CONNECT 'c:\archivos de programa\archivos comunes\borland shared\data\employee.gdb'  
user 'SYSDBA' password 'masterkey';
```

```
SET TERM !;
```

```
CREATE PROCEDURE MONEDAS  
RETURNS (PAIS CHAR(30),MONEDA CHAR(10),CALIF CHAR(10))  
AS  
BEGIN  
  FOR SELECT COUNTRY,CURRENCY FROM COUNTRY  
  INTO :PAIS, :MONEDA  
  DO BEGIN  
    IF (:PAIS='Argentina') then calif='EL MEJOR';  
    ELSE calif='NADA';  
    SUSPEND;  
  END  
END!
```

```
SET TERM ;!
```

Notemos que es necesario conectar explícitamente con la Base de Datos usando la sentencia

```
CONNECT
```

.

Al ejecutar este programa se crea en la Base de Datos un Procedimiento Almacenado llamado “MONEDAS”. Luego podemos hacer, ya sea directamente en WISQL o en un Query de Delphi:

```
SELECT * FROM monedas
```

y veremos todos los países con su moneda y calificación. Si existe un país llamado “Argentina”, se verá la calificación “EL MEJOR”.

Utilización desde Delphi

Podemos ejecutar un procedimiento almacenado desde Delphi utilizando un componente

StoredProc

. Sólo es necesario especificar el nombre de la Base de Datos en la propiedad

DatabaseName

, como siempre, y el nombre del procedimiento en la propiedad

StoredProcName

(que muestra una lista de los procedimientos disponibles). Para ejecutarlo, hay que llamar al método

ExecProc

del componente.

Nota: un procedimiento que use

```
SUSPEND
```

(como el que creamos en el ejemplo anterior) no debería ejecutarse de esta manera, ya que sólo veríamos el primer registro del resultado. En su lugar tenemos que llamar al procedimiento como si fuera una tabla, dentro de una sentencia

```
SELECT
```

.

Generadores

Los generadores son provistos por la Base de Datos para permitir la generación de números secuenciales únicos (enteros). Al crear un generador éste se pone automáticamente en cero.

Luego, cada vez que llamemos a la función

```
GEN_ID( )
```

, se incrementará en la cantidad especificada.

Se utilizan principalmente para asegurar valores únicos en la clave primaria de una tabla, insertándolos en un campo con un trigger. Corresponden a los tipos de datos Autoincrementales de Paradox o Access.

La sintaxis es muy simple:

```
CREATE GENERATOR nombre;
```

Para utilizarlo, como dijimos antes, utilizamos la función

```
GEN_ID( )
```

. En la llamada, especificamos el nombre del generador y la cantidad que queremos que se incremente para la próxima llamada:

```
GEN_ID(nombre_generador, incremento);
```

Veamos un ejemplo completo. Crearemos un generador para identificar unívocamente cada ítem de una factura, y un trigger para utilizarlo:

```
CREATE GENERATOR id_item;
```

```
COMMIT;
```

```
SET TERM !;
```

```
CREATE TRIGGER NuevoItem FOR items
```

```
BEFORE INSERT
```

```
POSITION 0
```

```
AS
```

```
BEGIN
```

```
    New.identItem = GEN_ID(id_item,1);
```

```
END!
```

```
SET TERM ;!
```

Los generadores son globales a la Base de Datos, es decir que se pueden utilizar desde cualquier procedimiento y deben tener un nombre único en todo el ámbito de la base.

Se puede modificar un generador especificando un nuevo valor, con

```
SET GENERATOR
```

```
:
```

```
SET GENERATOR nombre TO valor;
```

El valor debe ser un número entero entre -2^{31} y $2^{31} - 1$.

No existe la orden

```
DROP GENERATOR
```

; para borrar un generador debemos eliminarlo de una tabla especial del sistema:

```
DELETE FROM RDB$GENERATORS WHERE RDB$GENERATOR_NAME = '<nombre>';
```

Totalmente intuitivo, ¿no?.

El modelo relacional. Normalización

El modelo relacional de Bases de Datos fue concebido por E. F. Codd en 1969. Está basado en una teoría matemática -en las disciplinas de teoría de conjuntos y lógica de predicados. La idea básica es la siguiente:

Una base de datos consiste de una serie de tablas desordenadas (llamadas relaciones) que pueden ser manipuladas usando operaciones no procedurales que devuelven tablas.

Cuando se diseña una base de datos, hay que tomar decisiones sobre cómo es mejor modelado un sistema del mundo real. Este proceso consiste en decidir las tablas a crear, las columnas que contendrán, y las relaciones entre las tablas. Este proceso toma tiempo y esfuerzo, y es más cercano a un arte que a una ciencia.

La definición de la estructura de una base de datos puede decidir el éxito de una aplicación
--

Esta afirmación, aunque un poco categórica, no deja de ser verdad; en todas las aplicaciones será necesario efectuar cambios y ajustes sobre la marcha, y es mucho más fácil si está bien diseñada. En caso contrario, se puede incluso tornar *imposible* la tarea de agregar alguna funcionalidad.

Algunos de los beneficios de una base de datos organizada de acuerdo al modelo relacional son los siguientes:

- Eficiencia en las entradas, actualizaciones, y el borrado de elementos
- Eficiencia en la recuperación, sumariación y reporte de los datos
- Se puede predecir el comportamiento de la base de datos, dado que sigue un modelo bien formulado
- Es muy simple hacer cambios en el esquema de la base de datos (definición de las tablas)

Las tablas en el modelo relacional son usadas para representar cosas (entidades) del mundo real, que pueden ser objetos (un paciente, una factura) o sucesos (una visita del paciente, una llamada de teléfono). *Cada tabla debería representar sólo un tipo de entidad.*

Clave primaria

Las tablas se componen de filas y columnas. El modelo relacional dicta que *las filas en una tabla deben ser únicas*. Caso contrario, no habría forma de seleccionar una fila unívocamente. La unicidad de las filas de una tabla se garantiza definiendo una *clave primaria* (primary key).

Una clave primaria se compone de una o varias columnas (clave simple o compuesta, respectivamente). Estas columnas contienen valores que no se repiten, es decir, permiten determinar unívocamente una fila de la tabla.

Por ejemplo, en una tabla que contenga datos de personas, no podríamos usar la columna de “Nombre” para la clave primaria, dado que puede haber más de una persona con el mismo nombre. Lo mismo sucede con el apellido, e incluso con una clave compuesta por las dos columnas, “Nombre + Apellido”. Hay muchas familias en las cuales los hijos tienen el mismo nombre que los padres o abuelos.

¿Y si agregamos a esta clave compuesta la columna “Dirección”?

En el ejemplo anterior, una buena elección sería una columna con el Nro. de Documento de Identidad. Este número, por definición, es único para cada persona y por lo tanto cumple con los requisitos.

Clave externa

Las claves externas son columnas de una tabla cuyos valores referencian la clave primaria de otra tabla. Estas claves permiten relacionar las tablas.

Las claves primarias y externas que se relacionen deben estar en el mismo *dominio* o conjunto de valores. Es decir: deben ser del mismo tipo de datos y deben tener el mismo rango de valores posibles. Por ejemplo, si la clave primaria de la tabla de empleados es un campo numérico entero con valores entre 1 y 10000, y queremos relacionar con ésta otra tabla que contenga datos de trabajos realizados, la clave externa de la segunda tabla debe ser también entera y con valores de 1 a 10000.

Los gestores de bases de datos como Access o Paradox no controlan esta restricción totalmente; sólo toman en cuenta el tipo de datos. Así por ejemplo, sería posible relacionar una clave “Años trabajados” con una “Edad” si son del mismo tipo pero provienen de diferentes dominios.

Relaciones

Las relaciones entre tablas indican que las filas de una están ligadas a las de otra de alguna manera, de la misma manera que las relaciones entre las entidades del mundo real (como por ejemplo, la relación “es hijo de” entre un conjunto de padres y un conjunto de chicos).

Las relaciones se hacen entre dos tablas, y pueden ser de tres tipos: uno-a-uno, uno-a-muchos, y muchos-a-muchos.

Relaciones uno-a-uno

Dos tablas están relacionadas de manera uno-a-uno si, para cada fila de la primera tabla, hay *como máximo* una fila de la segunda. Estas relaciones se dan raramente en el mundo real, y son utilizadas principalmente cuando se desea dividir una tabla en dos (por ejemplo, por razones de seguridad).

Relaciones uno-a-muchos

Dos tablas están relacionadas de manera uno-a-muchos si para cada fila de la primera tabla hay cero, una, o muchas filas de la segunda tabla; pero para cada fila de la segunda tabla hay *sólo una* de la primera. Esta relación también se conoce con el nombre de *relación padre-hijo* (*parent-child*) o *relación principal-detalle* (*master-detail*).

Es el tipo de relaciones más utilizado. El ejemplo típico es una factura u orden de pedido: normalmente pondremos los datos de la orden o factura en una tabla, y los datos de los ítems pedidos en otra. De esta manera se pueden pedir tantos ítems como se quiera para una orden, pero cada ítem individual corresponde a una sola orden.

Otra aplicación muy común se da en las tablas de búsqueda (lookup tables), cuando el valor para un campo está restringido a los valores existentes en otra tabla; por ejemplo, los ítems de una factura sólo pueden ser los que están en la tabla de stock.

Relaciones muchos-a-muchos

Dos tablas están relacionadas de manera muchos-a-muchos cuando para cada fila de la primera tabla puede haber muchas en la segunda, y para cada fila de la segunda puede haber muchas en la primera. Este tipo de relaciones se da por ejemplo en el caso de una tabla con datos de empresas de seguros y otra con datos de asegurados. Cada empresa puede tener más de un asegurado, y cada asegurado puede a su vez estar en más de una empresa.

Los sistemas de bases de datos relacionales no pueden modelar las relaciones muchos-a-muchos directamente. Hay que dividir la relación en dos de tipo uno-a-muchos creando una tabla intermedia. En el ejemplo anterior tendríamos que crear una tabla de enlace con una clave externa a cada una de las dos tablas relacionadas, por ejemplo con campos ID_Empresa e ID_Asegurado.

Normalización

El proceso de normalización tiene por objetivo alcanzar la estructura óptima en una base de datos. La teoría nos da el concepto de *formas normales* como reglas para alcanzar progresivamente un diseño mejor y más eficiente. El proceso de normalización es el proceso de reestructurar la base de datos para organizar las tablas de acuerdo a las formas normales.

Un par de notas antes de continuar:

- En el proceso de normalización de tablas *no se pierde información*; sólo se estructura de manera diferente.
- Las tablas se descomponen de tal manera que siempre sea posible volver a reunirlos utilizando consultas; en la práctica, esto significa que tiene que haber relaciones entre las nuevas tablas resultantes de la descomposición.
- La normalización *no es obligatoria*, sólo recomendable. Hay veces que convendrá quebrantar alguna de las reglas; no obstante, hay que estar bien seguro de las razones para hacer esto, ya que estas recomendaciones son resultado de largos estudios sobre optimización del proceso de los datos.

Daremos aquí sólo las definiciones de las primeras tres formas normales y algunos ejemplos.

NOTA: los campos marcados con un asterisco (*) forman la clave principal de cada tabla

1^{ra} forma normal (1NF): los valores de todas las columnas deben ser atómicos, esto es, no divisibles. Además, no debe haber campos repetidos.

Por ejemplo si tuviéramos en una tabla de facturas registros como los siguientes, estaríamos violando la primera forma normal:

NroFactura (*)	Cliente	Items
0001-000001	Aquiles Meo de la Torre	1 mesa, 6 sillas, 1 aparador
0001-000002	Elba Gallo	2 estanterías, 1 mesa ratona

La segunda parte de la 1ra forma prohíbe una reestructuración como la siguiente:

NroFactura (*)	Cliente	Item1	Item2	Item3
0001-000001	Aquiles Meo de la Torre	1 mesa	6 sillas	1 aparador
0001-000002	Elba Gallo	2 estanterías	1 mesa ratona	

Aquí se ve el problema de poner varios campos para el mismo tipo de información; si alguien compra más de tres ítems es un problema; lo mismo si queremos averiguar por ejemplo cuantas sillas se han vendido habría que rastrear las tres columnas buscando sillas. El diseño no es bueno.

Para que esta tabla cumpla los requisitos de la 1ra forma normal, podríamos estructurarla como sigue:

NroFactura (*)	NroItem (*)	Cliente	Cantidad	Item
0001-000001	1	Aquiles Meo de la Torre	1	Mesa
0001-000001	2	Aquiles Meo de la Torre	6	Sillas

0001-000001	3	Aquiles Meo de la Torre	1	Aparador
0001-000002	1	Elba Gallo	2	Estanterías
0001-000002	2	Elba Gallo	1	Mesa ratona

En este caso formaríamos una clave primaria con el Nro de Factura y el Nro de Item juntos. Con esta construcción, sería muy fácil consultar por la cantidad de sillas vendidas.

2^{da} forma normal (2NF):	se dice que una tabla está en segunda forma normal cuando está en 1NF y cada columna que no es clave es dependiente completamente de la clave primaria entera.
---	--

En otras palabras, la clave primaria debe implicar el valor de cada otra columna en el registro. Esto generalmente viene solo como resultado de *estructurar la tabla con datos de distintas entidades*. Por ejemplo, si en la tabla de datos vendidos anterior agregamos un campo "Fecha" para llevar la fecha de emisión de la factura, tendríamos una situación como la siguiente:

NroFactura	NroItem	Fecha	Cliente	Cantidad	Item
0001-000001	1	4/8/98	Aquiles Meo de la Torre	1	Mesa
0001-000001	2	4/8/98	Aquiles Meo de la Torre	6	Sillas
0001-000001	3	4/8/98	Aquiles Meo de la Torre	1	Aparador
0001-000002	1	5/8/98	Elba Gallo	2	Estanterías
0001-000002	2	5/8/98	Elba Gallo	1	Mesa ratona

El problema aquí está en que el campo Fecha depende sólo del nro. de factura, no del nro. de ítem. Es decir, no depende de la clave principal completa sino sólo de una parte. En otras palabras, cada valor de la clave primaria completa *no define* los demás campos del registro; hay un campo que se define sin necesidad de conocer la clave completa. Lo mismo sucede con el campo Cliente.

Para alcanzar la 2NF, se puede dividir la tabla anterior en dos tablas diferentes, una con los datos de las facturas y la otra con los datos de los ítems.

Esto se aplica en general, como ya hemos visto más arriba: *cada tabla debería representar sólo un tipo de entidad*. Aplicando esta regla práctica casi siempre aseguramos que se cumpla la 2NF (y por lo tanto la 1NF).

3^{ra} Forma Normal (3NF):	se dice que una tabla está en tercera forma normal si está en 2NF y todas las columnas que no son clave son mutuamente independientes.
---	--

Esto elimina inmediatamente las columnas con valores calculados, así como las columnas que muestran la misma información de diferentes maneras. Por ejemplo, en el caso de la factura anterior podríamos agregar campos a la tabla de ítems para llevar cuenta del precio unitario de cada uno, y calcular el

subtotal de cada línea multiplicando este valor por la cantidad de artículos dada en el campo *cantidad*; pero para cumplir con 3NF, este cálculo no se almacenará en un nuevo campo de la tabla. Para esto se definen los **campos calculados**, que no tienen existencia en la tabla física sino que se crean “al vuelo” en el momento de mostrar los datos. Ya hablamos de estos campos al trabajar con los campos virtuales; también es posible crearlos en una consulta SQL.

Esta regla también se aplica al caso de tener en una tabla de ítems un campo para el código de producto y otro para la descripción; no son independientes entre sí, si uno cambia hay que cambiar al otro.

Ejemplo

Veamos otro ejemplo para clarificar un poco las cosas. El almacén del Sr. Nolo (Ma-Nolo Groceries Co.) desea implementar un sistema para llevar las cuentas corrientes de sus clientes. Después de un arduo trabajo para lograr entrevistar al ocupado señor Nolo (y un más arduo trabajo para descifrar lo que éste quiso decir), el programador llega a la conclusión que sólo hará falta una tabla:

LaUnica

IdCliente

Nombre y Apellido

Direccion

Telefono

CUIT

NroCuenta

NroFactura

Saldo

Comentarios

Vencimiento Pago 1

Monto Adeudado Pago 1

Fecha Pago 1

Monto Pagado 1

Vencimiento Pago 2

Monto Adeudado Pago 2

Fecha Pago 2

Monto Pagado 2

Vencimiento Pago 3

Monto Adeudado Pago 3

Fecha Pago 3

Monto Pagado 3

Cuando el jefe de programadores (que había estudiado en la UTN) vio el diseño de la Base de Datos... bueno, era jueves.

Si bien a primera vista parece que esta estructura de datos puede resolver el problema, si pensamos un poco en las posibles consultas y operaciones a realizar llegamos a las siguientes conclusiones:

? No será fácil saber cuántas cuotas tiene la cuenta.

? Tampoco será fácil determinar cuántas faltan por pagar.

? Si queremos saber lo que se recaudó en una fecha determinada... y ni hablar de un rango de fechas!

? Si el día de mañana Nolo decide dar más de tres cuotas, hay que armar todo de nuevo.

? Cuando se abre más de una cuenta corriente (se pueden comprar varios productos en cuenta, y por cada uno se abre una nueva) habrá que repetir los datos del cliente.

- ? Si se hacen menos de tres cuotas, estamos desperdiciando espacio en disco.
- ? Cada vez que se modifica un campo de pago hay que recalcular el saldo y colocar en el campo correspondiente. Si hay algún problema entre el momento de la modificación y el recálculo, los valores no coincidirán.

Después de leer un poco sobre la normalización, nuestro programador llega a la siguiente estructura:

<u>Cientes</u>	<u>Cuentas</u>	<u>Cuotas</u>
IdCliente (*)	NroCuenta (*)	Cuenta (*)
Nombre y Apellido	Cliente	NroCuota (*)
Direccion	NroFactura	Vencimiento
Telefono	Comentarios	Monto Adeudado
CUIT		Fecha Pago
		Monto Pagado

Claro que estas tablas están relacionadas:

- ? el campo *Cliente* de la tabla **Cuentas** es una clave externa al campo *IdCliente* de la tabla **Cientes** (relación uno-a-muchos, un cliente con varias cuentas)
- ? el campo *Cuenta* de la tabla de **Cuotas** es una clave externa que referencia al campo *NroCuenta* de la tabla **Cuentas** (también una relación uno-a-muchos)

Ahora si, la estructura parece mucho mejor. Podemos contestar a cada una de las objeciones anteriores con facilidad:

- ! No será fácil saber cuántas cuotas tiene la cuenta.
 - " Ahora es simple: ubicamos la cuenta por el número de cliente y el número de factura, luego contamos los registros de la tabla de Cuotas que correspondan al número de cuenta.
- ! Tampoco será fácil determinar cuántas faltan por pagar.
 - " Contamos igual que antes, pero ahora sólo las que tienen en blanco la fecha de pago o el monto en cero.
- ! Si queremos saber lo que se recaudó en una fecha determinada... y ni hablar de un rango de fechas!
 - " Seleccionamos la cuenta igual que antes, pero ahora sumamos los Montos Pagados de todos los registros que cumplan el requisito de las fechas.
- ! Si el día de mañana Nolo decide dar más de tres cuotas, hay que armar todo de nuevo.
 - " No será necesario, simplemente se agregarán más registros en la tabla de Cuotas.
- ! Cuando se abre más de una cuenta corriente (se pueden comprar varios productos en cuenta, y por cada uno se abre una nueva) habrá que repetir los datos del cliente.
 - " Por cada nueva cuenta, agregamos un registro a la tabla de cuentas y los que sean necesarios para las cuotas.
- ! Si se hacen menos de tres cuotas, estamos desperdiciando espacio en disco.
 - " Solo se utilizará el espacio necesario.
- ! Cada vez que se modifica un campo de pago hay que recalcular el saldo y colocar en el campo correspondiente. Si hay algún problema entre el momento de la modificación y el recálculo, los valores no coincidirán.
 - " Este campo será calculado cada vez que se necesite, como resultado de restar la suma de los montos a pagar de todas las cuotas menos la suma de los montos pagados.

Como vemos, la nueva estructura nos soluciona los problemas que planteamos al principio. El resultado que obtuvimos sale automáticamente aplicando las reglas de normalización a la primera tabla que mostramos.

No obstante, hay veces que es conveniente quebrar algunas de las reglas. Por ejemplo, el cálculo del saldo de una cuenta es bastante laborioso. Si tenemos al cliente en el teléfono preguntando cuánto debe, y la Base de Datos tiene muchos registros... sólo tenemos que cobrar por minuto de teléfono y terminaremos en las playas del Caribe disfrutando de vacaciones.

En este caso, convendría tener un campo en la tabla de Cuentas con el saldo de la misma; este campo se recalcularía cada vez que cambia algún registro de la tabla de Cuotas. De esta manera la consulta se agiliza mucho, y nos permitirá hacer rápidamente algunas otras preguntas del estilo “¿cuáles son las cuentas que deben más de \$1000?”. Seguimos teniendo el problema anterior, de la modificación unilateral por ejemplo si una “mano negra” ingresa a la Base de Datos desde afuera del programa y modifica los valores de uno de los pagos pero no el saldo total. Sólo que ahora tenemos la posibilidad de controlar el valor del saldo cada vez que lo deseemos, mientras tengamos el tiempo suficiente.

En definitiva: las reglas de normalización son *consejos*, resultados de estudios prolongados, que en general llevan a un diseño más eficiente de la Base de Datos. Pero **no son obligatorios**.

Funciones y procedimientos útiles de Delphi

No todo en Delphi son objetos. Además de la librería de componentes existe una librería de funciones y procedimientos de utilidad. Son las mismas que conformaban la batería de funciones de las versiones anteriores de Turbo Pascal, con algunos agregados muy prácticos.

Repasaremos ahora algunas de las más utilizadas. La referencia completa, como siempre, se puede encontrar en la ayuda del producto.

IntToStr

```
function IntToStr(Value: Longint): string;
```

Esta función toma un valor numérico entero y lo transforma a string.

Para transformar un valor real, use la función FloatToStr..

StrToInt

```
function StrToInt(const S: string): Longint;
```

Esta función toma una cadena compuesta por números y la transforma al valor entero correspondiente. Si no se puede convertir, se provoca una excepción EConvertError.

Para convertir un valor real, use la función StrToFloat o StrToFloatF

StrToIntDef

```
function StrToIntDef(const s: string; Default: integer): string;
```

Esta función toma una cadena compuesta por números y la transforma al valor entero correspondiente, al igual que StrToInt; pero si la cadena **s** no contiene un valor entero válido, devuelve el valor dado en **Default** en lugar de generar un error.

FloatToStr

```
function FloatToStr(Value: Extended): string;
```

Esta función toma un valor numérico real y lo transforma a string. No se puede controlar el formato del string resultante; para esto, utilice la función FloatToStrF.

Para transformar un valor entero, use la función IntToStr..

StrToFloat

```
function StrToFloat(const S: string): Extended;
```

Esta función toma una cadena y la transforma al valor real correspondiente. Si no se puede convertir, se provoca una excepción EConvertError.

Son válidos en el string los símbolos "+", "-", "E" (que indica notación científica y quiere decir "por diez a la...") y el separador decimal. Este separador (normalmente el punto o la coma) se define en el Panel de Control de Windows. No obstante, es posible cambiarlo localmente, solamente para nuestro programa, asignando el valor deseado a la variable **DecimalSeparator**.

Veamos un par de ejemplos:

- ! el String "5,32E2" se convierte al número 532.
- ! el String "-43.23" se convierte en el número -43.23
- ! para utilizar siempre la "coma" como separador decimal, al margen de lo que esté definido en el Panel de Control, podemos hacer

```
DecimalSeparator:= ', ';
```

al principio de la aplicación, por ejemplo en el evento `OnCreate` del form principal.

Para convertir un valor entero, use la función `StrToInt`.

FloatToStrF

```
function FloatToStrF(Value: Extended; Format: TFloatFormat; Precision,  
  Digits: Integer): string;
```

Esta función convierte un número real a su representación en caracteres (string) con el formato especificado. Los parámetros son los siguientes:

- ! **Value:** el número real a convertir
- ! **Format:** especifica el formato del string resultante. Puede ser alguna de las constantes siguientes:
 - " `ffGeneral`: el valor es convertido a la forma más corta posible, usando notación científica si es necesario.
 - " `ffExponent`: el valor es convertido a notación científica
 - " `ffFixed`: notación normal, con al menos un dígito antes de la coma decimal y la cantidad de dígitos decimales después de la coma dada por **Digits**.
 - " `ffNumber`: el mismo formato que para `ffFixed` pero con separadores de miles.
 - " `ffCurrency`: un valor monetario
- ! **Precision:** especifica la precisión del resultado; debe ser 7 o menos para variables de tipo *single*, 15 o menos para tipo *double* y 18 o menos para *extended*.
- ! **Digits:** indica la cantidad de dígitos decimales cuando no se usa la notación científica, o la cantidad de dígitos del exponente cuando se la utiliza.

StrToDate

```
function StrToDate(const S: string): TDateTime;
```

Convierte una cadena a Fecha. El formato se especifica en el Panel de Control de Windows, y dentro de la aplicación se puede consultar o modificar con las variables `DateSeparator` (separador: normalmente la barra inclinada "/") y `ShortDateFormat` (indica el orden de los componentes; por ejemplo 'd/m/y' para día/mes/año o 'm/d/y' para mes/día/año).

DateToStr

```
function DateToStr(Date: TDateTime): string;
```

Convierte una fecha a cadena. El formato de la cadena resultante está dado por la variable `ShortDateFormat`, como se explica en la función `StrToDate` más arriba.

De la misma manera se definen dos funciones `StrToTime` y `TimeToStr` que convierten entre cadenas y variables de tipo `TDateTime` pero considerando la parte de la hora, no la fecha. El carácter que se utiliza aquí para separar las distintas partes de la especificación está almacenado en la variable `TimeSeparator` (usualmente ":").

Format

```
function Format(const Format: string; const Args: array of const): string;
```

Esta función nos permite crear un string con datos de distintos tipos fácilmente. La cadena que queremos de resultado contiene caracteres especiales que indican dónde insertar los valores dados en el argumento **Args**, y con algún control sobre el formato de la conversión.

Los parámetros son los siguientes:

Args: un array de valores. Es una lista de valores a ser introducidos en la expresión final, separados por comas y encerrados entre corchetes. El reemplazo se hace en orden de aparición, es decir que el primer valor reemplaza al primer código, el segundo al segundo y así sucesivamente.

Format: el string a ser formateado, con códigos especiales que indican el tipo de dato que debe reemplazar. El string tiene el siguiente formato:

"%" [índice ":"] ["-"] [ancho] ["." precisión] tipo

comienza con un signo de porcentaje "%".

El único especificador obligatorio es el de tipo, que puede ser una letra de las siguientes:

- d Decimal. El argumento correspondiente debe ser un valor entero, que se convierte a un string de dígitos decimales. Si la cadena de formato tiene una especificación de precisión, indica que el string resultante debe contener al menos la cantidad especificada de dígitos; si tiene menos, se rellena con espacios a la izquierda.
- e Notación científica. El argumento debe ser un número en punto flotante. El valor se convierte a un string de la forma "d.ddd...E+ddd". Siempre hay un dígito precediendo al punto decimal, y si el número es negativo se coloca un signo menos delante. El número total de caracteres de la cadena resultante -incluyendo el punto decimal- es dado por el especificador de precisión de la cadena de formato. Si no se da este especificador, se toma una precisión de 15. La parte que sigue a la letra "E" (que indica 10 elevado a la cantidad que sigue) tiene siempre tres dígitos y un signo, "+" o "-".
- f Precisión fija. El argumento debe ser un número de punto flotante, que es convertido a un string de la forma "ddd.ddd...". Si el número es negativo se agrega el signo menos delante. El número de dígitos se especifica con la precisión, tomándose un valor por defecto de 2.
- g General. El argumento debe ser un número de punto flotante, que se convierte a la cadena decimal más corta posible, usando formato fijo o científico. El número de dígitos se especifica con la precisión, tomándose 15 por defecto.
- n Número. El argumento debe ser un número en punto flotante, que se convierte a un string de la forma "d.ddd.ddd.ddd,ddd...". Es igual que el formato con "f" pero con separadores de miles.
- m Monetario. El argumento debe ser un número en punto flotante, que se convierte a un string representando un valor monetario. El formato de la cadena es controlado por las variables globales CurrencyString, CurrencyFormat, NegCurrFormat, ThousandSeparator, DecimalSeparator, y CurrencyDecimals que se inicializan con los valores del panel de control de Windows, sección Internacional. Si se especifica un valor de precisión, tiene prioridad sobre el valor de la variable CurrencyDecimals.
- p Puntero. El argumento debe ser un valor de puntero, que es convertido a la forma segmento:offset "XXXX:YYYY" con dígitos hexadecimales.
- s String. El argumento debe ser un carácter, un string o un valor pChar. La cadena se inserta en el lugar del identificador de formato. Si se especifica un valor de precisión, se toma como longitud máxima de la cadena resultante (si se pasa de esta longitud, se trunca).
- x Hexadecimal. El argumento debe ser un valor entero, que se convierte a una cadena de dígitos hexadecimales. Si se especifica una precisión, se toma como la cantidad mínima de dígitos que debe contener la cadena resultante, llenándose por la izquierda con ceros.

Los especificadores de índice, ancho y precisión pueden ser dados directamente con dígitos decimales, o a través de parámetros indicando en la cadena de formato un asterisco "*". Por ejemplo,

```
format( '%*.*f', [ 8, 2, 123.456 ] ) es lo mismo que  
format( '%8.2f', [ 123.456 ] )
```

El especificador de ancho indica el mínimo ancho de un campo para la conversión. Si la cadena resultante es más corta, se llena con espacios a la izquierda; salvo que se especifique también el indicador de justificación a la izquierda (un signo menos antes del ancho) en cuyo caso los espacios se agregan a la derecha.

El indicador de índice nos permite reutilizar los valores del array de parámetros. Este número especifica el índice del parámetro a usar en el próximo reemplazo. Por ejemplo,

```
format( "%d %d %1:d", [ 1, 2, 3 ] ) da como resultado la cadena '1 2 2'.
```

DateToStr

```
function DateToStr(Date: TDateTime): string;
```

Convierte una fecha dada en formato TdateTime a un string.

StrToDate

```
function StrToDate(const S: string): TDateTime;
```

Convierte un string en TdateTime. Toma la hora como las 0:00. Si no puede convertir (el string contiene una fecha errónea) se genera una excepción EconvertError.

Si se ingresa el año con sólo dos dígitos, se determina el siglo a que pertenece utilizando la variable global TwoDigitYearCenturyWindow. Los detalles se pueden ver en la ayuda, con ejemplos.

IncMonth

```
function IncMonth(const Date: TDateTime; NumberOfMonths: Integer): TDateTime;
```

Incrementa la fecha pasada como parámetro (Date) en el número de meses dado por **NumberOfMonths**, tomando en cuenta el cambio de año y la diferencia de días entre los meses, si es necesario. El número de meses a incrementar puede ser negativo, dando como resultado una fecha anterior a la original.

Mensajes y notificaciones

Muchas veces nos encontraremos con que tenemos que mostrar mensajes al usuario de manera de atraer su atención. Para cumplir este objetivo, es generalmente deseable que el mensaje sea presentado en una ventana secundaria que se mostrará sobre las demás y tendrá que ser cerrada para poder continuar trabajando en el mismo programa (ventana Modal).

Microsoft ha especificado algunos criterios para este tipo de ventanas en sus Windows Interface Guidelines for Software Design (*Lineamientos para el diseño de interfaces en programas de Windows*). Viene al caso comentar algunos de estos criterios.

- Siempre es preferible evitar las situaciones que demandan la presentación de un mensaje. Si por ejemplo una determinada tarea necesita algunos recursos especiales para ser realizada, es mejor controlar la existencia de los mismos y deshabilitar el comando antes que permitir al usuario intentar la operación y mostrar luego un mensaje indicando la falta de recursos.
- El título del mensaje debe ser claro sobre quién produjo el mensaje. Mucho más tomando en cuenta que varias aplicaciones pueden estar funcionando simultáneamente; incluso dentro de una misma aplicación podemos tener un documento compuesto con objetos OLE. Por lo tanto es importante indicar en el título la aplicación (y eventualmente el documento) que genera el mensaje.
- *Nunca utilice en el título de una ventana de mensaje la palabra **Error***. Intente evitar la palabra *Error* en general: en lugar de “Error en el nombre de archivo” utilice “No se encuentra el archivo”.
- Utilice frases simples y entendibles para el usuario común. Evite mencionar características técnicas.
- Expresé el problema, su causa si es posible, y las acciones que el usuario puede tomar para corregirlo. En lugar de “No hay espacio en disco” utilice “El documento XXXX no pudo grabarse por falta de espacio en disco. Seleccione otro disco o libere un poco de espacio y vuelva a intentarlo”
- Trate de ofrecer la solución como una opción. Por ejemplo, en el ejemplo anterior de falta de espacio en disco se podría ofrecer al usuario vaciar la papelera.
- Limite el mensaje a dos o tres líneas. Si es necesaria más información, ofrezca un botón “Ayuda” que muestre la ventana de ayuda general en el tópico correspondiente.

Veamos ahora las funciones de Pascal que hacen posible mostrar una ventana de notificación en forma simple y rápida. Recuerde que siempre es posible sortear las limitaciones de estas funciones (por ejemplo, para agregar un botón no considerado o una animación) creando un cuadro de diálogo propio.

ShowMessage

```
procedure ShowMessage(const Msg: string);
```

Presenta en pantalla una caja de diálogo con el mensaje pasado como parámetro centrado sobre un botón OK. El título de la ventana es el nombre del ejecutable y no se puede cambiar. Es sólo para presentar mensajes en una forma fácil y rápida, generalmente usada por los programadores durante el desarrollo.

Otra función relacionada que nos permite más control sobre los botones y el título es **MessageDlg**

MessageDlg

```
function MessageDlg(const Msg: string; AType: TMsgDlgType; AButtons: TMsgDlgButtons;
    HelpCtx: Longint): Word;
```

Esta función muestra una caja de diálogo con un mensaje nuestro, permitiendo además especificar el título y los botones, así como un icono característico del tipo de mensaje.

Los parámetros son los siguientes:

- **Msg**: un string con el mensaje que debe presentar centrado en la caja de diálogo.
- **aType**: una constante que identifica el tipo de mensaje (y el título de la caja de diálogo). Los valores posibles son los siguientes:
 - mtWarning: caja de diálogo con un signo de exclamación amarillo. Título: **'Warning'**.
 - mtError: caja de diálogo con un signo de Stop rojo. Título: **'Error'**.
 - mtInformation: contiene un icono con una letra 'i' azul. Título: **'Information'**.
 - mtConfirmation: contiene un icono con un signo de pregunta verde. Título: **'Confirmation'**.
 - mtCustom: no contiene icono. El título es el nombre de la aplicación.
- **aButtons**: un set de constantes que especifican los botones que debe mostrar la caja. Los valores posibles son los siguientes:
 - mbYes, mbNo, mbOK, mbCancel, mbHelp, mbAbort, mbRetry (Reintentar), mbIgnore (Ignorar), mbAll (todo)

Además de estos valores (a ser usados entre corchetes, es decir formando un set) están definidos algunos conjuntos de uso común (no se ponen entre corchetes, ya están definidos como conjuntos):

- mbYesNoCancel: botones Yes, No, Cancel.
- mbOkCancel: botones Ok, Cancel.
- mbAbortRetryIgnore: botones Abort, Retry, Ignore.

HelpCtx: determina la página de ayuda a usar para la caja de diálogo.

Cuando se presiona un botón se cierra la caja de diálogo y la función MessageDlg devuelve un valor de tipo Word que indica el botón que fue presionado. Estos valores son constantes predefinidas:

- mrNone, mrOk, mrCancel, mrAbort, mrRetry, mrIgnore, mrYes, mrNo, mrAll

Ejemplos

La línea siguiente pide confirmación al usuario:

```
MessageDlg('Esta operación no se puede deshacer. ¿Está seguro de querer borrar el
    registro?', mtConfirmation, [mbYes, mbNo, mbCancel], 0);
```

el mismo resultado se logra utilizando uno de los conjuntos de botones predefinidos:

```
MessageDlg('Esta operación no se puede deshacer. ¿Está seguro de querer borrar el
    registro?', mtConfirmation, mbYesNoCancel, 0);
```

la línea siguiente presenta un mensaje preguntando al usuario si desea grabar los datos; según la respuesta, se ejecuta una parte u otra de la aplicación

```
if MessageDlg('¿Desea grabar los datos?', mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    {Grabar los datos}
else
    {no graba los datos}
```

Existe todavía otra forma de presentar un mensaje al usuario, un poco más configurable: la función MessageBox.

MessageBox

Actualmente, existen dos versiones de esta función con el mismo nombre: una en la API de Windows y un método de la clase `tApplication`. La única diferencia entre ellas es que el segundo provee automáticamente el manejador de ventana requerido por la función API. Veamos las declaraciones de las dos:

Función API:

```
function MessageBox(hWnd: HWND; lpText, lpCaption: PChar; uType: UINT): Integer;
```

Método de `tApplication`:

```
function MessageBox(Text, Caption: PChar; Flags: Longint): Integer;
```

Los parámetros son los mismos (salvo algunos nombres diferentes, que no alteran el resultado). Dado que en cualquier lugar del programa tenemos acceso a la instancia de `tApplication` que corresponde al mismo, discutiremos aquí la segunda versión (que se accede como `Application.MessageBox`).

- **Text:** en este parámetro especificamos el mensaje que queremos mostrar
- **Caption:** el título de la ventana
- **Flags:** especifica los botones y el comportamiento del cuadro de diálogo. Veremos aquí los valores más utilizados; para una referencia completa, vea la ayuda de la función en la API de Windows. Los valores se pueden combinar sumándolos.
 - **MB_ABORTRETRYIGNORE:** botones Abort, Retry, Ignore (Cancelar, Reintentar, Ignorar)
 - **MB_OK:** botón OK
 - **MB_OKCANCEL:** botones OK y Cancel
 - **MB_RETRYCANCEL:** botones Retry (Reintentar) y Cancel
 - **MB_YESNO:** botones Yes (Si) y No
 - **MB_YESNOCANCEL:** botones Yes, No y Cancel

El primer botón es marcado como botón por defecto (presionar ENTER en el teclado tiene el mismo efecto que presionar este botón) a menos que se especifique alguna de las siguientes constantes en el parámetro **Flags**:

- **MB_DEFBUTTON2:** el segundo botón es el botón por defecto
- **MB_DEFBUTTON3:** ídem para el tercer botón
- **MB_DEFBUTTON4:** ídem para el cuarto botón

También es posible indicar que queremos un dibujo explicativo (icono) en la ventana, utilizando las siguientes constantes:



MB_ICONASTERISK, MB_ICONINFORMATION

Este icono indica que el mensaje es una información únicamente; no se deberían ofrecer opciones al usuario, sólo el botón "Cerrar" o "OK".



MB_ICONHAND, MB_ICONERROR*, MB_ICONSTOP

Este tipo de mensaje notifica al usuario de alguna anomalía seria que requiere su intervención (la impresora no está lista, no hay disquete en la unidad, etc.). Normalmente se ofrecerá al usuario la opción de reintentar o cancelar la operación.



MB_ICONEXCLAMATION, MB_ICONWARNING*

Informa de una situación que requiere que el usuario tome una decisión antes de proseguir, acciones que pueden ser destructivas o irreversibles como reemplazar un bloque de texto demasiado extenso, borrar un registro en una tabla o reemplazar un archivo ya existente. Se dan a usuario las opciones necesarias a través de los botones.



MB_ICONQUESTION

De acuerdo con los lineamientos de diseño para Windows 95 y NT, este símbolo no debería utilizarse y

Las constantes marcadas con un asterisco (*) sólo están definidas en Windows 95 (Delphi 2 o superior).