Fundamentals of Database Development (with Delphi) – DB/1

Chapter one of the free Delphi database online course. Delphi as the database programming tool, Data Access with Delphi...just a few words, Building a new MS Access database.

Many Delphi beginners start with projects like "My Notepad" when beginning programming with Delphi, other developers spend nights and days in writing various multimedia and graphics applications, but all of them will sunner or later realize that 90% of today's software interacts with some data stored in some way.

There's no doubt about it, Delphi has powerful and reliable data−management capabilities. Application developers building the next generation of business software are attracted to Delphi for several reasons. With Delphi we can create software that operates with just about all types of desktop databases like Paradox, dBase or MS Access. We can also use Delphi to build solutions for client−server development.

Data Access with Delphi...just a few words

Delphi ships with more than 40 prebuilt database components and provides a visual programming environment that includes an integrated code editor, Database Form wizard that speeds up steps to create a browsable data form and Data Module Designer that can be used to share data access among multiple forms. Several other database specialized tools are also provided with Delphi to help us code faster and easier.

The Data Access page of the Components Palette provides components used to connect to a data source. In the Data Controls page, data aware components are ones that (after Delphi connects to a database) can be use to retreive and send data to or from a database. The components on the ADO page use ActiveX Data Objects (ADO) to access the database information through OLEDB. The components on the InterBase page access an InterBase database directly.

Don't runaway

Database programming, of course, is not trivial. In this course we will try to bring closer some of the techniques, problems and solutions to database programming with Delphi along with all the secrets it hides from us.
Before we move on to using various data components/tools in Delphi we should first see some of the concepts of database design and try to build a simple database.

Before we start interacting with databases using Delphi, it is a good idea to get a feel what modern databases are about.
When you think of a word database you should generally think of any type of data stored inside a computer − even a SomeFile.pas file (code to some Delphi unit) is some kind of database. Another type of database is a Word document or a simple .ini file. To access an .ini file we generally use routines and techniques for typed or untyped files.

Building a modern database application requires us to think of data in a relational way. The basic idea behind the relational model is that a database consists of a series of tables (or relations) that can be manipulated using operations that return tables or so−called views. Simply put, a **database** is best described as a collection of related data. A database may include many different **tables**. Tables are like grids where columns are called **fields** and rows are called ... rows.

To fully address the concepts of database design and relational model we would need an extra online course. For a great overview check out the Fundamentals of Relational Database Design.

New...Database

Since this course will primarily focus on ADO/Access Delphi approach to database programming we will now see how to create a new .mdb database in MS Access.

If you have never built a database with MS Access, go see MS Access tutorials for a great info.

I hope you know that on this site there is a Members Area where Delphi developers can upload their free code applications and components. Each member has it's name, an email address and a possibly a web page. If we would like to keep track of every application posted to this community we could assemble a database of three tables: Applications (general information about an application), Authors (who made the application) and Types (what kind of app is it). Let's see how to do just that:

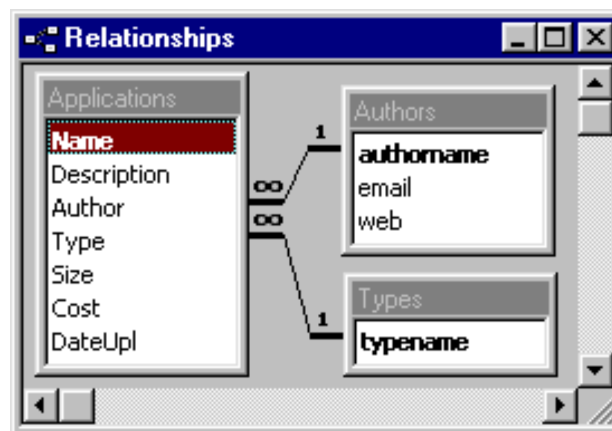
Start Access and create a blank database named aboutdelphi.mdb. Create three tables in Design view: Applications, Authors and Types.
Let's see the structure of those tables:

The Applications table contains fields that match the application description requirements: Name, Description, Author, Type, Size, Cost, DateUpl and Picture. Name, Description, Author and Type fields contain Text data, 50 characters by default. The Size filed is of a Number (Single) type – used to store the size of a file in Kb. The Cost field is a Currency field – if the app is shareware or commercial. The DateUpl field is a date/time value. The Picture is of a OLE Object type and will hold an (optional) picture for an application. Let the filed Name be the primary key.

The Authors table contains fields that match the application author requirements: AuthorName, Email and Web. All the fields contain character data (50 chars by default). Let the filed AuthorName be the primary key.

The Types table contains only one field: TypeName which is the primary key for this table. This table will be used to store the type of application (graphical, multimedia, database, ...)

We now only have to set up a relation in the relationships window and the database is ready.



Both relations should "Enforce Referential Integrity" with only "Cascade Update Related Records" check on.

Filling some data
In order to have some "dummy" data in a database fill in the Types table with the following 4 records: 'Game','Database','Internet','Graphics'. This values will be used when choosing the type of the application stored in the Applications table. Next, add one row to the Authors table: 'Delphi Guide', 'delphi.guide@about.com', 'http://delphi.about.com'. Finally let the only one row in the Applications table look like: 'Zoom', 'Zooming the Destop', 'Delphi Guide', 'Graphics', 10, 0, 02/20/2001. For the moment leave the last field (Picture) empty.

What to do with this "blank" database...I'll show you that in the following chapters of this course.

Connecting to a database. BDE? ADO? – DB/2

Chapter two of the free Delphi database online course. How to connect to an Access database – the UDL file? Looking forward: the smallest ADO example.

As shown in the previous chapter of this course, a database is a collection of one or more tables that store data in a structured format. These tables, contain data that is represented by rows and fields. When a database consists of two or more tables, these tables generally contain separate yet related data. MS Access, Interbase or SQL Server use a single file (Access, the *.mdb file) that represents the entire database. On the other hand, Paradox and dBase databases are defined with separate tables and files that represent indexes and table relations. All the files needed to describe a Paradox database are usually stored in a single directory. Delphi, of course, has means of working with both approaches.

With Delphi, we can connect to different types of databases: local or client/server (remote server) database. Local databases are stored on your local drive or on a local area network. Remote database servers usually reside on a remote machine. Types of local databases are Paradox, dBase and MS Access. Types of client/server databases are MS SQL Server, Interbase or Oracle.
Local databases are often called single–tiered databases. A single–tiered database is a database in which any changes, such as editing the data, inserting records, or deleting records – happen immediately. Single–tiered databases are limited in how much data the tables can hold and the number of users your application can support. When the database information includes complicated relationships between several tables, or when the number of clients grows, you may want to use a two–tiered or multi–tiered application. Client applications run on local machines; the application server is typically on a server, and the database itself might be on another server. The idea behind the multi–tier architecture is that client applications can be very small because the application servers do most of the work. This enables you to write what are called thin–client applications.

When we write a database application in Delphi, we need to use some *database engine* to access a data in a database. The database engine permits you to concentrate on what data you want to access, instead of how to access it. From the first version, Delphi provides database developers with the BDE (Borland Database Engine). Beside the BDE, Delphi from the fifth version supports Microsoft ADO database interface.

This course will primarily focus on *MS Access local database* producing the single–tiered application.

The BDE is a common data access layer for all of Borland's products, including Delphi and C++Builder. The BDE consists of a collection of DLLs and utilities. The beauty of the BDE is the fact that all of the data manipulation is considered "transparent" to the developer. BDE comes with a set of drivers that enables your application to talk to several different types of databases. These drivers translate high–level database commands (such as open or post) and tasks (record locking or SQL construction) into commands specific to a particular database type: Paradox, dBASE, MS Access or any ODBC data source. The BDE API (Application Programming Interface) consists of more than 200 procedures and functions, which are available through the BDE unit. Fortunately, you almost never need to call any of these routines directly. Instead, you use the BDE through the VCL's data access components, which are found on the Data Access page of Component Palette. To access the particular database the application only needs to know the Alias for the database and it will have access to all data in that database. The alias is set up in the BDE Administrator and specifies driver parameters and database locations.
The BDE ships with a collection of database drivers, allowing access to a wide variety of data sources. The standard (native) BDE drivers include Paradox, dBase, MS Access, ASCII text. Of course, any ODBC driver can also be used by the BDE through the ODBC Administrator.

Delphi applications that use the BDE to access databases require that you distribute the BDE with the application. When deploying the BDE with an application, you must use InstallShield Express or

another Borland certified installation program.

The BDE has several advantages as well as disadvantages as a database engine. It's not my intention to discuss about why and when you should (or not) use the BDE approach over some non−BDE technique.

Since this course is about ADO/MSAccess the rest of the course will focus on this non−BDE database approach.

As stated in the Introducing ADO in Delphi article, ADO is a set of COM components (DLLs) that allow you to access databases as well as e−mail and file systems. Applications built with ADO components don't require the BDE.
To access any kind of database with ADO, you'll of course need to have ADO/OLE DB libraries. Everything you need to use ADO is probably already on your computer: the files are distributed by Microsoft as a part of Windows 98/2000. If you or your client use Windows 95 or Windows NT you will probably need to distribute and install the ADO engine. Delphi 5's CD includes an installation of MDAC − Microsoft Data Access Components. You should always make sure to have the latest version, which is available from Microsoft. The Microsoft Data Access Components are the key technologies that enable Universal Data Access. They include ActiveX Data Objects (ADO), OLE DB, and Open Database Connectivity (ODBC).
Note: to install correctly on a Windows 95 computer, MDAC requires that DCOM95 be installed. MDAC installs components that rely on DLLs installed by DCOM95 in order to register correctly. Note that DCOM95 is not required on a Windows NT 4.0. In some cases, DCOM may not be installed on a Windows 98 computer. If it has not been installed, then DCOM98 should be installed prior to the installation of MDAC.

Without to much talking about OLE DB and ADO let's move on to more practical topics.

ADO Objects
The ADO programming model is built around several ADO objects that provide you with the productive means for accessing all kinds of data sources. These objects provide the functionality to connect to data sources, query and update record sets, and report errors. Delphi, through several VCL components provides wrapper components to access those objects. Let's see what are some of the Objects ADO works with:

The *Connection* object represents a connection to the data source with the connection strings. In BDE/Delphi a Connection object is a combination of the Database and Session components.

The *Command* object enables us to operate on a data source. Ir represents a command (also known as a query or statement) that can be processed to add, delete, query or update the data in a database.

The *Recordset* object is a result of a Query command. You can think of a Recordset as a Delphi Table or Query component. Each row that the Recordset returns consists of multiple *Field* objects.

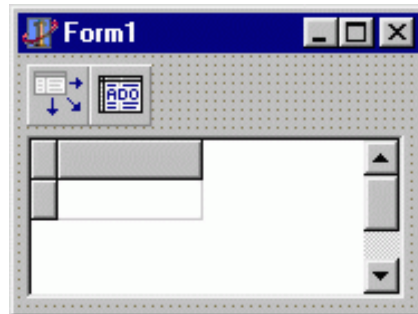
Several other objects like: the Field object, the Parameter Object and the Error object also exist in ADO model − will get back to them in the following chapters of this course.

Before going on to the brief explanation of each component in AdoExpress collection, let's first see how to connect to an Access database. Of course, we will be connecting to our AboutDelphi.mdb sample database.

Delphi (5) ADO support is concentrated in the ADOExpress components on the ADO tab of the component palette. Several other database enabled components will be used through this course. For the moment we will focus on the minimal set of components needed to access an Access database with ADO.

Start Delphi, this will open a new application with one blank form.

In order to be able to access data in an Access database with ADO and Delphi, you must add at least three data aware components to our project. First, the *DBGrid* on the DataControls component page – used to browse through the records retrieved from a table or by a query. Second, the *DataSource* (DataAccess Page) used to provide a link between a dataset and DBGrid component on a form that enable display, navigation, and editing of the data underlying the dataset. And finally the *ADOTable* (ADO page) that represents a table retrieved from an ADO data store. Drop all of them on a form. Let the names be the default one. The form should look something like:

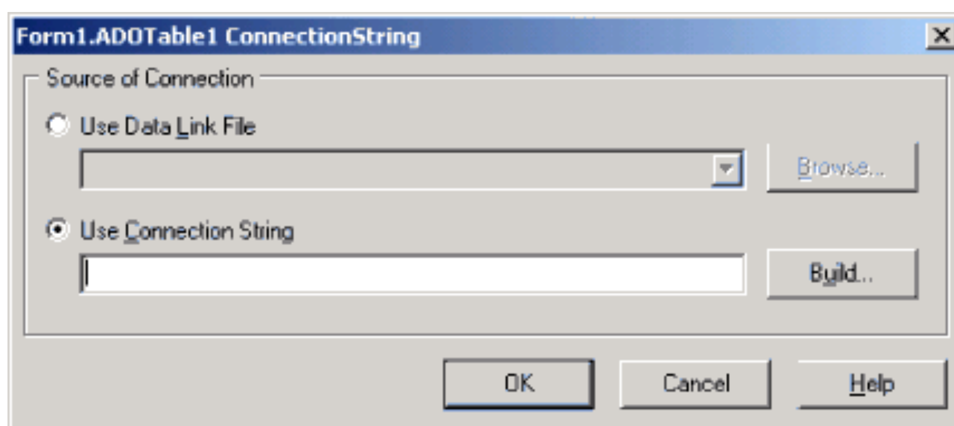


If you run the application now, nothing is displayed in a Grid – of course, we did nothing to really connect to a database. Note just one more thing: only the Grid is displayed, the rest two component are controls – unvisible to the user.

Link between components
In order to display some data from a database we have to link all three components together. Using the Object Inspector, set the following:

DBGrid1.DataSource = DataSource1
DataSource1.DataSet = ADOTable1

We have now reached the hard part, to really get the data from our database we have to build a ConnectionString. This string indicates where the database is physically stored and how we are accessing it. When you double click the ellipsis button at the ConnectionString property of the AdoTable component you get the next dialog box:



When building a connection string we have two choices: use the Data Link File (.UDL) or build a connection string by hand. Let's build it. Press the *Build* button – this pops up the Data Link Properties dialog. This dialog has 4 pages. The Provider tab allows you to specify the provider – select the Microsoft Jet 4.0 OLE DB Provider. The Next button leads us to the second page: Connection. Select the ellipsis button to browse for our database (AboutDelphi.mdb). Press the *Test Connection* button to see if the connection is successful – it should be. Leave all the other pages as they are. Finally, click on OK to close the Data Link Properties dialog, again OK to close the ConnectionString dialog – the

connection string is stored in the ConnectionString property of the ADTTable component. The connection string should look something like:

```
Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=C:\!gajba\About\aboutdelphi.mdb;
Persist Security Info=False
```

To finish, we have to set the table name that is to be accessed by the ADOTable component – again use the Object Inspector.

```
ADOTable1.TableName = Applications
```

If you want to see the data at design time use the ADOTable Active property – set it to True.

Ha! If you have done all the steps you now see the only one record (row) in the Applications table. When you start the application you can even change the data in the database. Of course, you cannot do much more – this is the simplest ADO example I could think of.

This concludes this chapter. In the next chapter we will address all the ADO component provided with Delphi and how they communicate with the rest data–aware components to crate a powerfull Delphi database application.

Pictures inside a database – DB/3

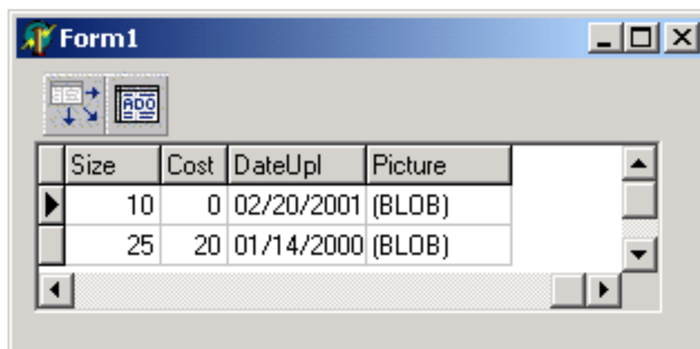
Chapter three of the free Delphi database online course. Displaying images (BMP, JPEG, ...) inside an Access database with ADO and Delphi.

These days developing database applications requires more than just operating with textual or numeric data. If you are, for example, developing an Internet/intranet or multimedia based application, frequently there is a need to display pictures along with text from a database.

In this third chapter of the Delphi database course, we'll see *how to pull out and display the graphical data (images) inside an Access database with ADO*. Don't be worried with the fact that working with images inside an Access database requires more database programming skills than this course has provided so far. Let's pretend that we know more to get more.

If you have followed this course from the beginning (specially the second chapter), you know how to connect to a database and display the Applications (from our working aboutdelphi.mdb database) table in a DBGrid. Remember, we used 3 data components: DBGrid, ADOTable and DataSource to get and display the data from the Applications table.
Back in the first chapter when we created our database, the last filed in the Applications table was left blank (after filling our database with some dummy data). The last field has the name Picture and is of the OLE object type.

If you scroll right to the last column of the DBGrid you'll see something like:



When using MS Access, we can store images (and other large data objects such as sound or video) in a field that has the OLE object type. This type of data is referred to as a Binary Large Object Bitmap (BLOB).

Naturally when working with images, several types of picture formats are available. The most commonly used include JPEG, GIF and BMP. JPEG format has proven to be widely accepted by Web masters because of the small amount of storage required (in other words JPEGs are smaller than BMPs in bytes).

Delphi, of course, has means of handling BMP, GIF and JPEG graphic formats. The rest of this article will deal with **JPEG** file formats.

Storing pictures in Access

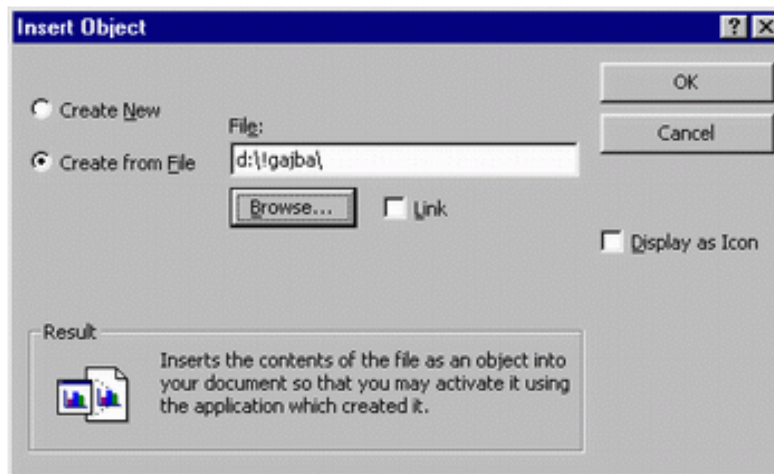
Before going on to discussion about displaying the image inside a table within a Delphi form, we need to add some graphical data to our database.

Start Access and open the aboutdelphi.mdb database. Open the Applications table (it should have one row of data) and select the Picture field.

Size	Cost	DateUpl	Picture
10	0,00 Kn	02/20/2001	
0	0,00 Kn		

To add an image do the following:

1. Select Insert|Object... this will display the Insert Object dialog box.



2. Click on the Browse button, this pops up the Browse open dialog. Note: you probably have some .jpg files on your computer, so you could use those, or if you have Win 98 and newer, MS Paint will save pictures in this format, as will many other programs. Navigate to a directory where your pictures are stored and select one.

Note: the text in the Picture field holds the name of an executable used to work with JPEG files on your computer. Of course you don't see the picture in a table grid. To actually see the graphics double click that field. This will load the image within the JPG type associated application.

Now, when we have a picture inside a database, let's see how to display it inside a Delphi form. We already have a Delphi form with data components on it from the second chapter of this course.

The DBImage – take one

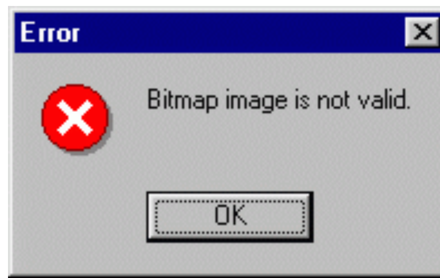
The first thing I do when trying to do something new with Delphi is to "ask" Delphi Help for help. This is what the Help system replies: TDBImage (Data Controls page on the component palette) represents a graphic image from a BLOB (binary large object) field of the current record of a dataset. Use TDBImage to represent the value of graphic fields. TDBImage allows a form to display graphical data from a dataset. The DBImage is nothing more than a TImage component with some data aware properties. The two most important ones are: *DataSource* and *Field*. The DataSource property links the image component to a dataset. We have a DataSoure component named DataSource1 on our form that represent a dataset. The Field property indicates the field (in a table) that holds the image.

All clear, put a DBImage on form and leave the DBImage1 name. To actually link a DBImage with a BLOB field in a Table we simply need to do the following assignment (using the Object Inspector):

```
DBImage1.DataSource = DataSource1
DBImage1.Field = Picture
```

This should do the trick of displaying the JPEG image stored in the Picture field of the Applications table.

To see whether this assignment will work the only thing we have to do is to set the Active property of the ADOTable1 component to True. We can do this at design time with the Object Inspector. Once you set it you'll get the following:



Now what? Why does it say "Bitmap image is not valid." We have a JPEG picture not the BMP – is this the problem? Let's go back to the Help.

After a few clicks through the Help the conclusion is: to get the JPG inside a database we need to use the TJpegImage object. To display the picture we need the simple, non–data aware, version of the Image component. Even more we'll need to use streams to load a picture from a BLOB object. The Help states that we should use TADOBlobStream to access or modify the value of a BLOB or memo field in an ADO dataset.

Pulling the Jpeg – take two!

Since we can do nothing with the DBImage – remove it from the form and place an ordinary TImage component (Additional palette) on it. Name it ADOImage. Unfortunately the Image component does not have any data–aware properties, so all the code needed to show a picture from a table inside it will require a separate procedure. The easiest thing to do is to put a Button on a form and place all the code inside it's OnClick event. Name the button 'btnShowImage'.

To use the ADOBLOBStream the Help suggests to create an instance of TADOBlobStream, use the methods of the stream to read from a graphic field in a dataset, and then free the BLOB stream. Somewhere "in the middle" we'll need to use LoadFromStream to load a Jpeg image from a TADOBlobStream object. The Image's component Picture.Graphic property will be used to actually store and display the picture.

Field object, what are those?
At this moment I'll assume that only a small amount of knowledge on Field objects will be enough for you to keep with this chapter. In Delphi database development one of the primary objects is the TField object. Field components are non–visual objects that represent fields of the dataset at run (and design) time. The TADOTable (and other TDataSet descendant) gives access to the Fields Editor at design time. The Fields Editor enables you to select the fields that you want to include in the dataset. More important, it creates a persistent lists of the field components used by the dataset in your application. To invoke the Fields Editor double click the TADOTable component. By default, the list of fields is empty. Click Add to open a dialog box listing the fields in the Applications table. By default, all fields are selected. Select OK.
When Delphi gives (default) names to Fields the following notation is used: Table name + Field name. This means that our picture field has the name: ADOTable1Picture.

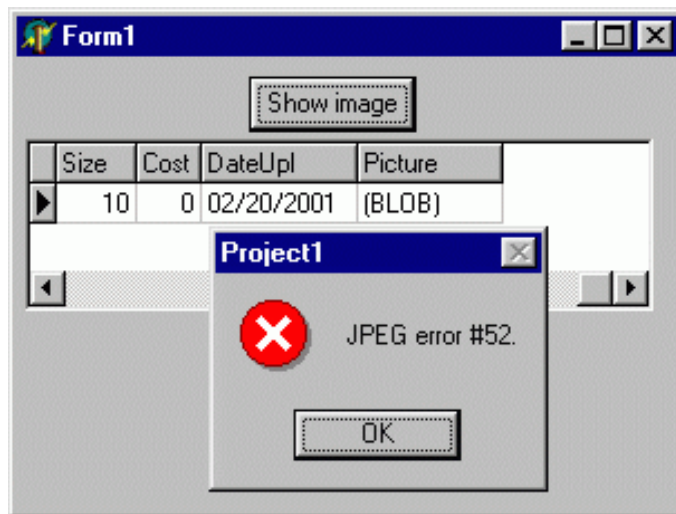
The TADOBlobStream Create method creates an instance of TADOBlobStream for reading from or writing to a specific BLOB field object, which is in our case the *ADOTable1Picture* field.

We will place the code in the OnClick event for a btnShowImage button. The code should read the picture from the Picture field of the currently selected row. This is how the code should look like:

```
uses jpeg;
...
procedure TForm1.btnShowImageClick(Sender: TObject);
var
  bS  : TADOBlobStream;
  Pic : TJpegImage;
begin
  bS := TADOBlobStream.Create
```

```
        (AdoTable1Picture, bmRead);
  try
    Pic:=TJpegImage.Create;
    try
     Pic.LoadFromStream(bS);
     ADOImage.Picture.Graphic:=Pic;
    finally
     Pic.Free;
    end;
  finally
    bS.Free
  end;
end;
```

Ok, let's run the project now. Of course set the ADOTable1.Active property to True. The form is displayed and after clicking on a button this is what we got:



Hm, what now? The code in the procedure is 100% correct but the image doesn't get displayed! Remember the "Never give up, never surrender"? Let's go down to byte level to see what's happening!

OLE object type format – take three!

All this leaves us with nothing but to store the picture to a disk (as an ordinary binary file) and see what's inside it.
One nice thing with picture files (formats) is that all have some header that uniquely identifies the image. The JPG picture file starts with the, so called, SOI marker that has the value of $FFD8 hex.

This next line of code stores the value of the Picture field to a file (BlobImage.dat) in the working directory. Assign this code in the OnCreate event for the form, start the project and remove the code.

```
ADOTable1Picture.SaveToFile('BlobImage.dat');
```

Once we have the file, we can use some Hex editor to see it's content.

```
08 09 0A 0B   0C 0D 0E 0F   0123456789ABCDEF
08 00 08 00   14 00 1C 00     $
61 67 65 00   50 61 63 6B   ````Package Pack
02 00 00 00   08 00 00 00   age
00 00 00 00   00 00 00 00   Package
6F 6D 32 2E   4A 50 47 00    ;     zoom2.JPG
41 5C 41 42   4F 55 54 2E   d:\!GAJBA\ABOUT.
55 52 53 45   5C 5A 4F 4F   COM\DBCOURSE\ZOO
00 03 00 27   00 00 00 64   M2.JPG     '   d
5C 41 42 4F   55 54 2E 43   :\!GAJBA\ABOUT.C
52 53 45 5C   5A 4F 4F 4D   OM\DBCOURSE\ZOOM
00 00 FF D8   FF E0 00 10   2.JPG  ;   Ř´ŕ
00 00 00 00   00 00 FF E1   JFIF          ´á
49 49 2A 00   08 00 00 00    ĽExif  II*
00 00 01 00   00 00 1A 01
00 00 1B 01   05 00 01 00          V
03 00 01 00   00 00 02 00      ^    (
```

Would you believe this! MS Access stores the path of a linked OLE object as part of the object's definition in the OLE object field. Because the definition of OLE object storage is not documented (!? this is straight from MS) there is no way to know what gets written before the actual image data.

Think about this twice. First: we'll need to seek to the 'FFD8' and read the image from there. Second, the 'FFD8' might not always be at the same position in the file. Conclusion: we need a function that returns the position of the SOI marker for the JPG file stored as OLE object in an Access database.

The correct way – take four!

Provided with the Blob type field our function should return the position of the 'FFD8' string inside the ADOBlobStream. The ReadBuffer reads byte by byte from the stream. Each call to ReadBuffer moves the position of the stream by one. When two bytes together (as hex values) result in SOI marker the function returns the stream position. This is the function:

```
function JpegStartsInBlob
         (PicField:TBlobField):integer;
var
 bS     : TADOBlobStream;
 buffer : Word;
 hx     : string;
begin
 Result := -1;
 bS := TADOBlobStream.Create(PicField, bmRead);
 try
  while (Result = -1) and
        (bS.Position + 1 < bS.Size) do
  begin
   bS.ReadBuffer(buffer, 1);
   hx:=IntToHex(buffer, 2);
   if hx = 'FF' then begin
     bS.ReadBuffer(buffer, 1);
     hx:=IntToHex(buffer, 2);
     if hx = 'D8' then Result := bS.Position - 2
     else if hx = 'FF' then
        bS.Position := bS.Position-1;
   end; //if
  end; //while
 finally
  bS.Free
 end;  //try
end;
```

Once we have the position of the SOI marker we use it to seek to it in the ADOBlob stream.

```
uses jpeg;
...
procedure TForm1.btnShowImageClick(Sender: TObject);
var
  bS  : TADOBlobStream;
  Pic : TJpegImage;
begin
  bS := TADOBlobStream.Create
        (AdoTable1Picture, bmRead);
  try
    bS.Seek(JpegStartsInBlob(AdoTable1Picture),
            soFromBeginning);
    Pic:=TJpegImage.Create;
    try
     Pic.LoadFromStream(bS);
     ADOImage.Picture.Graphic:=Pic;
    finally
     Pic.Free;
    end;
  finally
    bS.Free
  end;
end;
```

Run the project and voila!

Who can now say that programming isn't FUN?

Note: in real code application we would have the code to read and display the image from the current row in the *AfterScroll* event of a TDataSet (that is in the ADOTable1AfterScroll event procedure). AfterScroll occurs after an application scrolls from one record to another.

Take five!

That's it for this chapter. You can now store and display all your favorite JPG pictures. In the last page of this article I have provided you with the entire code (form1's unit); all the data assignment is placed in the OnCreate event of the form. This ensures that all three components are correctly linked – you don't need to use the Object Inspector at design–time.

I agree, the chapter was not designed for beginners, but hey the World is cruel. Another thing: did you mentioned that at the end you don't know how to change (or add some new) picture in a table! We'll, that's whole another story!

Data browsing and navigation – DB/4

Chapter four of the free Delphi Database Course for beginners. Building a data browsing form – linking data components. Navigating through a recordset.

Welcome to the fourth chapter of a free DB Delphi Course! So far, this course has provided enough information to connect to an Access database and even to display a graphical data inside a database table. In the last chapter we were discussing some *advanced* database programming techniques – let's go back to more *for beginners* level now.

This time, you will see how to build a form (the real one) that can be used to browse through the records of a database table.
All the examples presented in the previous chapters have used several data–enabled (ADOTable, DBGrid, ...) components without to much explaining what each component is designed for, and how all those data components link together.

Working together...

When developing ADO–based Delphi database applications, the components on the Data Controls page, the ADO page, and the Data Access page of the component palette allow our application to read from and write information to databases.

Every (ADO) data–aware Delphi form, in general, consist of
▪ several data–aware controls (Data Controls tab) that create a visual user interface (the look of the data form).
▪ one DataSource component (Data Access tab) that represents an interface between a dataset component and data–aware controls on a form.
▪ one or more dataset components (ADO tab) that provide access to the data residing in a database table or query.
▪ a connection component (ADO tab) that points all the dataset components to a specific data store.

Data Controls
Delphi's data–aware components are components that normally reside on a Standard palette tab but have been modified to display and manipulate the content of data in a dataset (table or query). The choice of controls is determined by how we want to present the information and how we want to let users browse (and manipulate – add or edit) through the records of datasets. DBEdit and DBMemo, for example, are used to represent an individual record from a dataset. The DBGrid, on the other hand, is generally used when representing the contents of an entire dataset. Since all the data–aware controls are counterparts to the standard Windows controls – with a few extra properties, building a functional database application should be a relatively familiar task.
All the data–aware components share one common property: Data Source.

Data Source
Simply put, the DataSource component provides a mechanism to hook dataset components to the visual data–aware components that display the data. You generally will need one datasource component for each dataset component to present a link to one or more data–aware controls.

Datasets
To create an ADO based application, Delphi provides us with four dataset components: TAdoDataSet, TAdoTable, TAdoQuery and TAdoStoredProc. All of the components are designed to retrieve, present and modify the data. All those components can connect directly (as like in the previous chapter's examples) to an ADO data store (such as data in an Access database) through it's ConnectionString property or they can chare a single connection. When connecting through a TAdoConnection the Connection specifies an ADO connection object to use to connect to an ADO data store.

ADO Connection
The ADOConnection component is used to establish a connection with an ADO data store. Although each ADO dataset component can directly connect to a database, we will typically want to use the ADOConnection component since the component provides methods and properties for activating the connection, accessing the ADO data store directly and for working with transactions. In order to connect to a specific database, we use the ConnectionString property.

Now, when we know the theory it's time to see some action. The next step is to build a data form. Before we move on, it'll be a good idea to open the database with Access and add some "dummy" data (3−4 records) to a database just to have some operational data.

There are two different ways of creating forms with access to a data from a database. The first way is to use the Database Form Expert. Unfortunately, the Database Form Expert works only with the BDE−aware set of dataset components. The second way is to place and connect all the data components by hand.

Defining the User Interface

We'll build our data browsing form in three steps. First step is to define the user interface for the form. Next, the data access components are added and configured. In the third and final step, the data−aware controls are added.

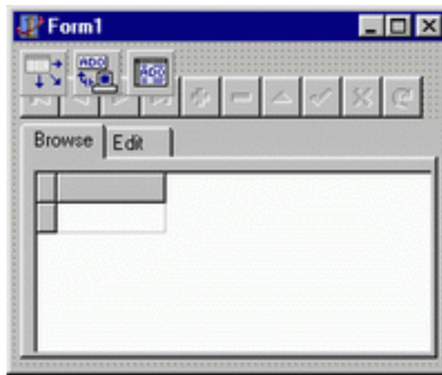
Before you start, close any open projects. Then use the following steps:

- Select File|New Application. This creates a new project containing an empty form, a unit, and a project file.
- Add one one *PageControl* on the form. The PageControl can be found on the Win32 tab on the component palette. Let it have the default name, PageControl.
- Add two *TabSheets* on the PageControl. Set the Caption of the first TabSheet1 to "Browse". Set the Caption of the second TabSheet1 to "Edit".

- Place a DataSource (DataAccess tab), an ADOTable and an ADOConnection (ADO tab) component on the form. Leave all the components with their default names.
- Select the first page of the PageControl and place a DBGrid (Data Controls tab) component on the Browse tabsheet.
- place a DBNavigator component (Data Controls tab). The navigator buttons are used to move through the records in a table.
- By using the Object Inspector set the link between components like:

```
DBNavigator1.DataSource = DataSource1
DBGrid1.DataSource = DataSource1
DataSource1.DataSet = ADOTable1
ADOTable1.Connection = ADOConnection1
ADOConnection1.ConnectionString = ...
ADOConnection1.LoginPrompt = False
ADOTable1.Table = 'Applications'
```

Note: as discussed in the second chapter, the ConnectionString property indicates where the data is physically stored and how we are accessing it. You can use the same connection string as in the second chapter, or you can build one by invoking the connection string editor.
Setting the LoginPrompt property of the ADOConnection component to False suppresses the database login from showing. Since we have not set any password for our database we don't need the login prompt.



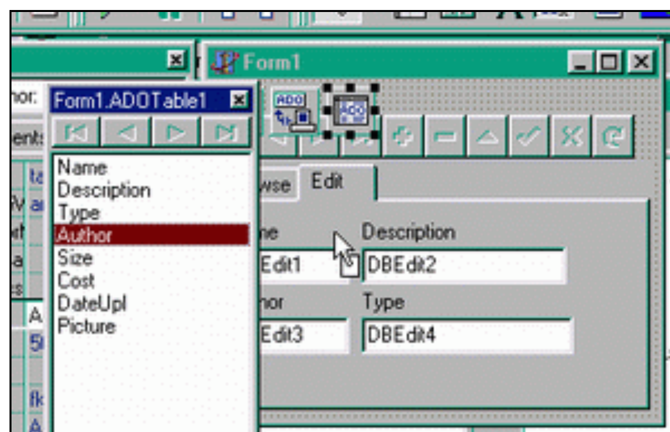
Fields

The DBGrid component is generally used when we want to present the entire recordset to the user (data in a table). Even though we can use the DBGird to let the user add, edit and delete records in a table – better approach is to use Field objects for all the fields in a table. Field objects are mostly used to control the display and editing of data in your applications. By using the Fields Editor we can set the list of persistent field object for every column in a table. The Field Editor is invoked by double clicking the DataSet component (ADOTable1). To add fields to the list of persistent fields for a dataset right–click the list and choose Add Fields.

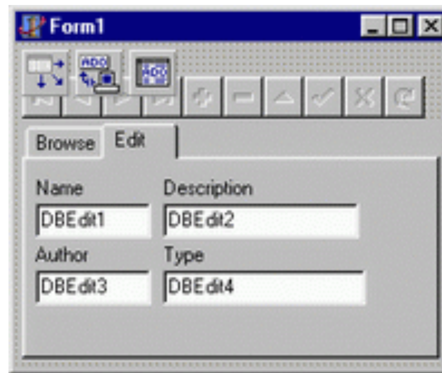
Rather than presenting all the data in a table to the user (within the DBGrid), we might want to use field–oriented data aware components, such as edit boxes. The DBEdit component for example, is a data–aware version of the TEdit class. DBEdit is the building block of any data entry application.

The easiest way to place a DBEdit on the form and connect it with a field in a table is to use the following:

1. Invoke the Fields editor by double clicking on the ADOTable component.
2. Select the Name field, for example. Let the second tab of the Page control be the selected one.
3. Drag the Name field on the form.



When you drop the Name field on the tabsheet, Delphi places one Label and one DBEdit component on it. The Caption of the Label component is the same as the *DisplayLabel* property of the dragged field. The DBEdit component is linked to the dataset's datasource through it's *DataSource* property.
If you select more than one filed from the Fields Editor and drag it on the form, Delphi will set as much Label/DBEdit components as you have dropped on the form.



It's alive

Ok, all set up. Everything we have to do now is to activate the connection and scroll through the records. The *Active* property of a dataset component (ADOTable) indicates whether we have a live connection to a table or not. Setting the Active to True or calling the *Open* method sets the Connected property of the ADOConnection to True – and displays the data in related data–aware controls.

First, Move by, Last,...

Now, we are finally ready for some action. The next step is to see how to walk through the recordset. The DBNavigator component gives a simple and friendly tool for navigating through the recordset. In addition to it's navigational abilities, the DBNavigator provides a means for manipulating the data with actions like Insert, Delete or Cancel the changes. For example, if the we click the Delete button, the correct record is deleted from the recordset. Each button is optional and you can mix and match at will.

Using the button set we are able to skip to the last record or move to the previous one. For example, clicking on the Last button sets the current record to the last record in the recordset and disables the Last and Next buttons. Clicking the Last button is functionally the same as calling the Last method of a dataset.
Note that one of the navigational operations that the DBNavigator cannot process is moving forward or backward in a recordset by a number or records. The *MoveBy* method of a dataset is used to position the cursor on a record relative to the active record in the recordset.

That's it for this chapter. We are now ready to move on to topics like editing and searching the recordset, expect to learn about that in the following chapters of this course...

Behind data in datasets – DB/5

Chapter five of the free Delphi Database Course for beginners. What is the state of data? Iterating through a recordset, bookmarking and reading the data from a database table.

When developing database applications with Delphi and ADO, most of the work is done with the help of dataset components. To create an ADO based application, Delphi provides us with several dataset components. TAdoTable, TAdoQuery and others are all designed to retrieve, present and modify the data inside a database table or query.
In this fifth chapter of the free database course we'll see exactly how to present, navigate and read the data – by looking at some of the most interesting datasets properties, events and methods.

Pick, set, connect and get
Since this is the fifth chapter, you should be familiar with the steps needed to create a database form. Back in the fourth chapter we have created, by hand, a simple data browsing form. The same form can be used to follow the discussion in this chapter.

The only (ADO) dataset component we used, by now, was TAdoTable. It's important to know that both TADOQuery and TADODatSet (as dataset components) share the same set of common methods and events.

Open Sesame ; Close Sesame

One of the great features of Delphi database development is that Delphi enables us to work directly with the data while in design–mode. If you recall – in the previous chapters we used the *Active* property at design time to open the live connection with the data.
It's understandable, that prior to working with the data in a table, an application must first open a dataset. Delphi has two methods of performing this function. As we already saw, the Active property can be set to True at design or run time. We can also call the *Open* method at run time. For example, add the following code to the form's OnCreate event handler to get the data from the ADOTable1 component:

```
ADOTable1.Open;
```

Note: Every ADO dataset can acess data in a database through its own *ConnectionString* property or through an ADOConnection component (and it's *ConnectionString*). If the ADOTable1 component is connected to ADOConnection1 component (preferable) than opening the ADOTable will result in activating the corresponding ADOConection component. The ADOConnection provides two events that will be executed: *OnWillConnect* and *OnConnectComplete*.

The Open method sets the Active property to True and activates the connection. When we are done with using the connection we can close it by setting the Active property to False or by calling the *Close* method. Generally you will place the call to Close in the form's OnClose event handler:

```
ADOTable1.Close;
```

Before moving on, it's crucial to know that working with dataset's methods and properties relies on knowing the current state of the data. Simply put, the State property of a dataset determines what actions can and cannot occur at any moment on a dataset.

How are you doing?

If the dataset is closed the State of the data indicates an Inactive connection. No operations or actions or methods can be done on the data while the connection is closed. The first time we open the connection the dataset is placed in the default Browse state. You should always be aware of the state "your" data is in. For example, when we connect a dataset to a DBGrid, the user is able to see the underlying dataset (or recordset), but to be able to change some of the data the State must be

changed to Edit.

It's important to know that the dataset state constantly changes as an application processes data. If, for example, while browsing the data in a DBGrid (Browse state) the user starts editing the records the state will automatically change to Edit. Of course, this is the default behaviour of the data–aware controls (DBGrid, DBEdit) with their *AutoEdit* property set to True.

But, how do we get the state? The ADOTable (nor any other dataset component) doesn't have an event that triggers when the State changes.
Ok, let's see: for each dataset component we generally use one datasource component to present a link to one or more data–aware controls. That's it.

Every datasource component has an *OnStateChange* event that fires whenever the state of the underlying dataset changes. Placing the following code for the OnStateChange event handler causes the caption of the form to indicate the current state of the ADOTable1 dataset component:

```
procedure TForm1.DataSource1StateChange
  (Sender: TObject);
var ds: string;
begin
 case ADOTable1.State of
  dsInactive: ds:='Closed';
  dsBrowse  : ds:='Browsing';
  dsEdit    : ds:='Editing';
  dsInsert  : ds:='New record inserting';
 else
  ds:='Other states'
 end;
 Caption:='ADOTable1 state: ' + ds;
end;
```

In the last chapter we used the DBNavigator component to navigate through the dataset. This component presents a visual tool for navigating through a dataset. As stated, the DBNavigator has buttons that the user can click to move among dataset's records at run–time.

Moving on from BOF to EOF

To iterate through a recordset and to sum some values we'll need to use methods of a dataset component. Take a look at the following code:

```
...
 ADOTable1.DisableControls;
 try
  ADOTable1.First;
  while not ADOTable1.EOF do begin;
    Do_Summing_Calculation;
    ADOTable1.Next;
  end;
 finally
  ADOTable1.EnableControls;
 end;
...
```

The *First* method is used to set the current row in the dataset to the first one; the *Next* moves to the next row in a dataset. The *EOF* (and *BOF*) property indicates whether the dataset is at the last (first) row.
In most cases, the dataset is connected to one or more data–aware controls. When long iterations take place it's quite interesting to "disconnect" those data–aware controls from the dataset – to prevent data–aware controls from updating every time the active record changes. The *DisableControls* and *EnableControls* are used to disable or enable data display in controls associated with the dataset.
The error catching (try–finally) part simply ensures that all data–aware controls remain connected to

the dataset if some exception occurs.
The *Do_Summing_Calculation* should obviously sum values represented by fields in a dataset.

Bookmarking
Prior to calling the above code the dataset was probably at some *middle* position – the user was browsing a dataset with a DBGrid. The code moves the *current* row to the end (EOF) causing the program to loose the previous position. It would be much better (and user friendly) if we could store the current position and make it the current one (again) when the iteration completes. Of course, Delphi has that option. The *Bookmark* property of the ADOTable (and any other TDataset decedent) can be used to store and set the current record's position. Bookmarks are used like:

```
var Bok : TBookmarkStr
...
Bok := ADOTable1.Bookmark;
{iteration code}
ADOTable1.Bookmark := Bok;
```

The value of data
In the previous code the *Do_Summing_Calculation* part was left. Most likely that part should get the value of some field (column) in a dataset and sum it.
When we talk about record values in datasets we talk about values of data fields. As we have seen in the previous chapters the fields of a dataset are represented with unvisible Field components. In the examples from previous chapters we used the Object Inspector to set up a list of persistent fields for a dataset.

When data–aware controls are connected to a dataset and the user moves through a recordset the corresponding field values are presented in those controls. When we want to use the same values directly in code we need to know how to read them.
By default, when Delphi gives names to field objects the following notation is used: Table name + Field name. This means that if we have the Type field in table the filed object connected to that, hm, field will have the name: ADOTable1Type.

To access the data value from a field we can use several notations.

ADOTable1Type.Value
ADOTable1.Fields[x].Value
ADOTable1.FieldByName('Type').Value

Note: All fields of a dataset are stored in the Fields array. x represents the position of the field in the fields array.

The *Value* property for a field object holds the data value. Since Value is a varian type it's preferable to cast fields value to a type that we currently need. In other words an application should use the *AsString* property to convert a value (date, integer, currency, ...) in a field to a string when the string representation of the fields value is needed.

Now we can write the entire code to iterate through a recordset and count how many 'database' applications are in a table (of course we are talking about Applications table in our AboutDelphi.mdb Access database).

```
var Bok : TBookmarkStr
    ict : Integer;
begin
 ict:=0;
 Bok:=ADOTable1.Bookmark;
 try
  ADOTable1.DisableControls;
  try
   ADOTable1.First;
```

```
   while not ADOTable1.EOF do begin;
    if ADOTable1.FieldByName('Type').AsString
       = 'database' then Inc(ict);
    ADOTable1.Next;
   end;
  finally
   ADOTable1.EnableControls;
  end;
 finally
  ADOTable1.Bookmark:=Bok;
 end;
 ShowMessage('Number of database
              apps: ' + IntToStr(ict));
end;
```

I agree with you! We should use ADOQuery for such purposes!

That's it for the fifth chapter. Next time we'll see how to add, delete and insert recordset to a database table.

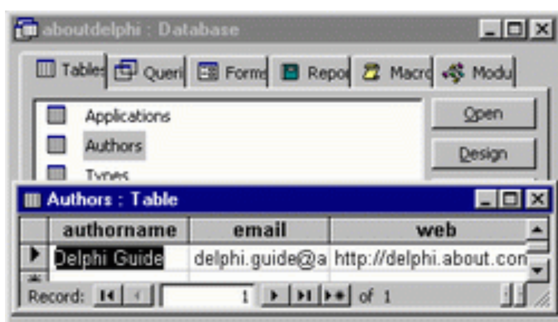
Data modifications – DB/6

Chapter six of the free Delphi Database Course for beginners. Learn how to add, insert and delete records from a database table.

The main goal of developing database applications is to provide a means of modifying the data. In the first five chapters this DB Course has shown how to connect to an Access database, how to display the data from a database table and how to navigate through the records in a table.

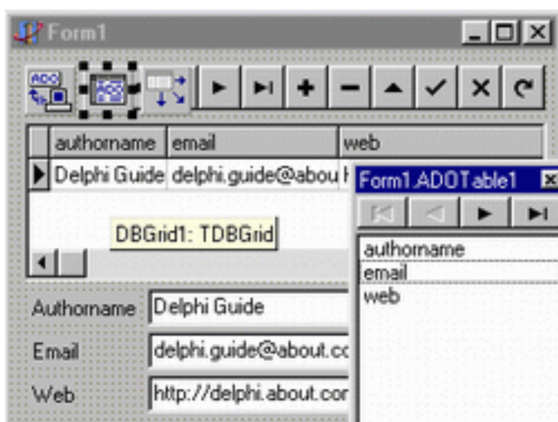
In this sixth chapter of the free database course we'll see exactly how to add, edit and delete the data – by looking at some of the most interesting properties, events and methods of the db–aware/enabled components and objects.

To follow this article you'll need to create a data form similar to ones we were creating in the previous chapters. Use the standard set (DataSource, ADOTable and ADOConnection) of components to connect to our Access database. This time we will be exploring the *Authors* table. Recall that the Authors table has three fields (columns): AuthorName, Email and Web. All three are text fields, in the first chapter we added one "dummy" record.



Start a new Delphi project and on the default new form place all the data access components and a DBGrid and a DBNavigator. Use the Object Inspector to set the link between all those components. Set the *Table* name of the *ADOTable1* component to point to the *Authors* table. You should already be familiar with the steps to achieve the connection. Use the *Active* property of the ADOTable to activate the connection at design time. Or use the OnCreate/OnClose pair of event handlers for the form to Open and Close the dataset at run–time.

One of the great advantages of database development with Delphi is in the existance of the TField object. As already stated, in the previous chapters, database fields can be persistent or created dynamically. It is recommended to set the persistent list of fields for a (known) dataset. By using the Object Inspector add all three fields to the list. Use dragging and dropping (as explained in the 5th chapter) to link a data–aware DBEdits to fields in a database table.



Posting

When linking components in a way that the DBNavigator is used with the data–aware components like the DBGrid operations like editing, deleting and inserting new records to a table are done semi–automatically. For example, suppose you are browsing the recordset with the DBGrid. When you start retyping the text in some cell (editing the value of the underlying field) the data is not modified until the *Post* method is called. The Post method (of a dataset) plays the central role in a Delphi database application.
When the dataset is in the Edit state, a call to Post modifies the current record. The DBNavigator has the Post button (the one with the check mark on it) that calls the Post method when clicked. You should be aware of the fact that Post is called implicitly (for example) when you move to the next record – just by pressing the down key while editing in a DBGrid.

When an application calls the Post method (implicitly or explicitly) several events happen that can be handeled by Delphi. For example the *BeforePost* event (of a dataset) is triggered before the "modified" record is actually modified and updated with the new values. Your application might use the *OnBeforePost* to perform validity checks on data changes before posting them to the database. This is a place where so–called record–based validation should be done. Record–based validation is used when other fields are involved in determining if a value entered for a field is valid. To check for the validity of one field at a time you could use the *OnValidate* event handler for that specific field. The OnValidate event handler is created from the Object Inspector when the Fields editor is invoked and the appropriate field is selected.

Editing a record

To be able to edit the data returned by a dataset the dataset must be in the Edit state. The default behaviour of the data–aware controls (DBGrid, DBEdit) with their *AutoEdit* property set to True is that once the user starts editing the values in DBEdit controls the state changes (from Browse) to Edit. No error occurs if we try to put a dataset in the Edit state while the dataset is already in the Edit state.

Programmatically editing and posting could look like:

```
ADOTable1.Edit;
ADOTable1AuthorName.AsString := 'Delphi Guide';
ADOTable1.Post;
```

The first line simply puts the dataset in the Edit state. The last one Posts the data to the database. The second one assigns the string value 'Delphi Guide' to the AuthorName field.

Take a look at (some of the) events that were triggered by the previous (simple) call.

ADOTable1**BeforeEdit**
DataSource1**StateChange**
DataSource1**DataChange**
ADOTable1**AfterEdit**
ADOTable1*AuthorName***Validate**
ADOTable1*AuthorName***Change**
DataSource1**DataChange**
DataSource1**StateChange**
ADOTable1**BeforePost**
DataSource1**StateChange**
ADOTable1**AfterPost**

Note: the DBGrid and the appropriate DBEdit component are refreshed to show the new value for the *AuthorName* field.

Adding a new record

The simplest way to add a new record to a table is to click on the DBNavigators Insert button (the one

with the plus sign on it). The *Insert* method called adds/opens a new – empty record in a table. The DBGrid display one empty row with the asterisk sign in the first column. All three DBEdit components are empty and ready for the user to enter values for the new record.
The call to Insert results in calling series related events, too.

Programmatically inserting and posting could look like:

```
with ADOTable1 do begin
 Insert;
 FieldByName('AuthorName').Value := 'Zarko Gajic';
 FieldByName('Email').Value := 'gzarko@sf.hr';
 FieldByName('Web').Value := 'http://sf.hr';
 Post;
end;
```

Note: the ADOTable component has the *InsertRecord* method that can be used to create a new, empty record at in the dataset, fill fields with values, and post the values to the database – all that with just one line of code. The previous example could look like:

```
ADOTable1.InsertRecord('Zarko Gajic',
                       'gzarko@sf.hr',
                       'http://sf.hr')
```

"Undo" changes

While in the Edit (the user is changing the data) or in the Insert state (a new record is to be added), the application can call the *Cancel* method. The DBNavigator has the X sign on the appropriate button. If the record is being edited the call to Cancel returns the original values to connected data–aware components. If the insertion was canceled the empty row is "deleted". Cancel returns dataset to Browse state.

Deleting a record

The button with the minus sign on the DBNavigator calls the *Delete* method for the dataset. There is no need to call the Post method after Delete. You can use the *BeforeDelete* event to attempt to prevent the user from deleting the record from table. Note that the DBNavigator has the *ConfirmDelete* property to help prevent the user from accidentally deleting a record from the dataset. If you don't have the DBNavigator connected to a dataset – pressing Ctrl+Delete in a DBGrid calls the Delete method. If while executing the Delete method an error occurs the *OnDeleteError* is triggered.

Queries with ADO − DB/7

Chapter seven of the free Delphi Database Course for beginners. Take a look at how you can take advantage of the TADOQuery component to boost your ADO−Delphi productivity.

In this chapter of the free database course for Delphi beginners − focus on ADO, we'll look at how you can take advantage of the *TADOQuery* component to boost your ADO−Delphi productivity.

SQL with TADOQuery

The TADOQuery component provides Delphi developers the ability to fetch data from one or multiple tables from an ADO database using SQL.

These SQL statements can either be DDL (Data Definition Language) statements such as CREATE TABLE, ALTER INDEX, and so forth, or they can be DML (Data Manipulation Language) statements, such as SELECT, UPDATE, and DELETE. The most common statement, however, is the SELECT statement, which produces a view similar to that available using a Table component.

Note: even though executing commands using the ADOQuery component is possible, the *ADOCommand* component is more appropriate for this purpose. It is most often used to execute DDL commands or to execute a stored procedure (even though you should use the *TADOStoredProc* for such tasks) that does not return a result set.

The SQL used in a ADOQuery component must be acceptable to the ADO driver in use. In other words you should be familiar with the SQL writing differences between, for example, MS Access and MS SQL.

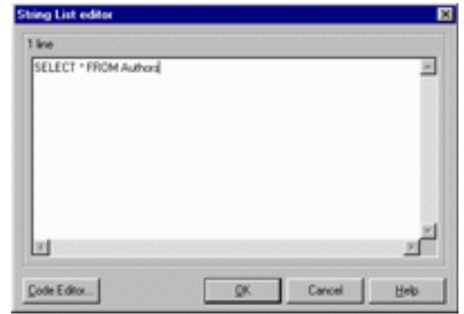
As when working with the ADOTable component, the data in a database is accessed using a data store connection established by the ADOQuery component using its *ConnectionString* property or through a separate ADOConnection component specified in the *Connection* property.

To make a Delphi form capable of retrieving the data from an Access database with the ADOQuery component simply drop all the related data−access and data−aware components on it and make a link as described in the previous chapters of this course. The data−access components: DataSource, ADOConnection along with ADOQuery (instead of the ADOTable) and one data−aware component like DBGrid is all we need.
As already explained, by using the Object Inspector set the link between those components as follows:

```
DBGrid1.DataSource = DataSource1
DataSource1.DataSet = ADOQuery1
ADOQuery1.Connection = ADOConnection1
//build the ConnectionString as described in the second chapter.
ADOConnection1.ConnectionString = ...
ADOConnection1.LoginPrompt = False
```

Doing a SQL query

The TADOQuery component doesn't have a *TableName* property as the TADOTable does. TADOQuery has a property (TStrings) called *SQL* which is used to store the SQL statement. You can set the SQL property's value with the Object Inspector at design time or through code at runtime.



At design–time, invoke the property editor for the SQL property by clicking the ellipsis button in the Object Inspector.
Type the following SQL statement: "SELECT * FROM Authors".

The SQL statement can be executed in one of two ways, depending on the type of the statement. The Data Definition Language statements are generally executed with the *ExecSQL* method. For example to delete a specific record from a specific table you could write a DELETE DDL statement and run the query with the ExecSQL method.
The (ordinary) SQL statements are executed by setting the *TADOQuery.Active* property to *True* or by calling the *Open* method (essentialy the same). This approach is similar to retrieving a table data with the TADOTable component.

At run–time, the SQL statement in the SQL property can be used as any StringList object:

```
with ADOQuery1 do begin
  Close;
  SQL.Clear;
  SQL.Add:='SELECT * FROM Authors '
  SQL.Add:='ORDER BY authorname DESC'
  Open;
end;
```

The above code, at run–time, closes the dataset, empties the SQL string in the SQL property, assigns a new SQL command and activates the dataset by calling the Open method.

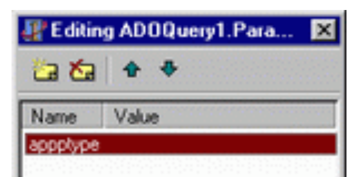
Note that obviously creating a persistent list of field objects for an ADOQuery component does not make sense. The next time you call the Open method the SQL can be so different that the whole set of filed names (and types) may change. Of course, this is not the case if we are using ADOQuery to fetch the rows from just one table with the constant set of fields – and the resulting set depends on the WHERE part of the SQL statement.

Dynamic queries

One of the great properties of the TADOQuery components is the *Params* property. A parameterized query is one that permits flexible row/column selection using a parameter in the WHERE clause of a SQL statement. The Params property allows replacable parameters in the predefined SQL statement. A parameter is a placeholder for a value in the WHERE clause, defined just before the query is opened. To specify a parameter in a query, use a colon (:) preceding a parameter name.

At design–time use the Object Inspector to set the SQL property as follows:

ADOQuery1.SQL := 'SELECT * FROM Applications WHERE type =: **apptype**'



When you close the SQL editor window open the Parameters window by clicking the ellipsis button in the Object Inspector.

The parameter in the preceding SQL statement is named *apptype*. We can set the values of the parameters in the Params collection at design time via the Parameters dialog box, but most of the time we will be changing the parameters at runtime. The Parameters dialog can be used to specify the datatypes and default values of parameters used in a query.

At run–time, the parameters can be changed and the query re–executed to refresh the data. In order to execute a parameterized query, it is necessary to supply a value for each parameter prior to the execution of the query. To modify the parameter value, we use either the Params property or ParamByName method. For example, given the SQL statement as above, at run–time we could use the following code:

```
with ADOQuery1 do begin
  Close;
  SQL.Clear;
  SQL.Add('SELECT * FROM Applications WHERE type =: apptype');
  ParamByName('apptype').Value:='multimedia';
  Open;
end;
```

Navigating and editing the query

As like when working with the ADOTable component the ADOQuery returns a set or records from a table (or two or more). Navigating through a dataset is done with the same set of methods as described in the "Behind data in datasets" chapter.

In general ADOQuery component should not be used when editing takes place. The SQL based queries are mostly used for reporting purposes. If your query returns a result set, it is sometimes possible to edit the returned dataset. The result set must contain records from a single table and it must not use any SQL aggregate functions. Editing of a dataset returned by the ADOQuery is the same as editing the ADOTAble's dataset.

An example

To see some ADOQuery action we'll code a small example. Let's make a query that can be used to fetch the rows from various tables in a database. To show the list of all the tables in a database we can use the *GetTableNames* method of the *ADOConnection* component. The GetTableNames in the OnCreate event of the form fills the ComboBox with the table names and the Button is used to close the query and to recreate it to retrieve the records from a picked table. The () event handlers should look like:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 ADOConnection1.GetTableNames(ComboBox1.Items);
end;

procedure TForm1.Button1Click(Sender: TObject);
var  tblname : string;
begin
if ComboBox1.ItemIndex < 0 then Exit;
tblname := ComboBox1.Items[ComboBox1.ItemIndex];
with ADOQuery1 do begin
  Close;
  SQL.Text := 'SELECT * FROM ' + tblname;
  Open;
end;
end;
```

Note that all this can be done by using the ADOTable and it's TableName property – much easily.

Data filtering – DB/8

Chapter eight of the free Delphi Database Course for beginners. Using Filters to narow the scope of data that is presented to the user.

As stated in one of the previous chapters, both TADOQuery and TADODatSet (as dataset components) share the same set of common methods and events. On of the features exposed by those datasets is the ability to narrow the scope of data that is presented to the user.

Consider that you might have a database table with thousands of records, but your users are interested in seeing or working on only a small subset of the table data.

To follow the article, set up the data form with the core components (data–access and data–aware) as described in the previous chapters. The next code examples will assume that you are working with the ADOTable component pointing to the Applications table in our working Access database.

Filtering

Filtering is the method by which some data from the dataset is excluded from view by displaying only those records that meet specific criteria. Filtering permits you to present varying views of the data stored in a dataset without actually affecting that data. This criteria is set through the *Filter* property of the dataset component (TADOTable or TADOQuery), it can be set at both design and run time. Filter property represents a string that defines the filter criteria.
For example, if you want to limit the displayed data (from the Applications table) to freeware applications (cost $0.00), a filter such as the following will only display records that meet the condition:

```
ADOTable1.Filter := 'Cost = 0';
```

You can also add a value for Filter based on the text entered in a control. If the filtered dataset should only display free applications and you want to enable users to supply the type of the applications, a filter could be set as follows:

```
ADOTable1.Filter :=
 'Cost = 0 AND
  Type = ' + QuotedStr(Edit1.Text);
```

By using combinations of the following operators, you can create quite sophisticated filters.

Operator	***Meaning***
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to
AND	Tests two statements are both *True*
NOT	Tests that the following statement is not *True*
OR	Tests that at least one of two statements is *True*

Filtered, FilterOptions, FilterGroup, OnFilterRecord
The *Filtered* property is a Boolean value (True or False) that determines if the string in the Filter property is used to filter the dataset. When Filtered is False, the filtering is ignored and the complete dataset is available to the application.

The *FilterOptions* is a set of two values – both used when filtering string fields. If the *foCaseInsensitive* is included in the FilterOptions, comparison between the literal in the Filter property string and the field values are case–insensitive. The *foNoPartialCompare* forces Delphi to treat the asterisks (*) as a literal character rather than as wildcard. By default, FilterOptions is set to an empty set.

The *OnFilterRecord* event fires each time the filtering is enabled and a new record becomes the current one. You will generally use this event to filter records using a criterion that can't be (easily) implemented using the Filter property.

```
procedure TForm1.ADOTable1FilterRecord
  (DataSet: TDataSet; var Accept: Boolean);
var AppZipSize : Single;
begin
  AppZipSize := ADOTable1.FieldByName('size').Value;
  Accept := (AppZipSize < 10);
end;
```

The key element here is the *Accept* parameter. You set the Accept parameter to True for any rows that you want to show. The preceding code sets Accept to True for any rows in which the Size field contains a value that is less than 10 (all apps whose download size is less than 10 Kb).

The *FilterGroup* set property allows you to filter records depending on their status.

To filter or not to filter

Note that

- the *Filter* property behaves much like a WHERE clause in a *SQL* statement.
- you can have multiple conditions, specified in the *Filter* property, separated by *AND* and *OR* operators.
- generally one should avoid Filters unless the fetched recordset is small. A filter is done on the fly, and may or may not use the current index (filters are applied to every record retrieved in a dataset).
- filters are rarely used with client/server databases, a SQL query (*TADOQuery*) should be used to achieve the same effect that filters have on local databases.
- you should generally not use filtering with datasets on data modules. In a specific situation when filtering a table that is never viewed from any other form, or a table that makes use of a range, or sort order that is not used anywhere else in the application – data modules *should* be avoided.
- to search a filtered dataset, you can use the *FindFirst*, *FindNext*, *FindPrior*, and *FindLast* methods. These methods are the best way to search a filtered dataset because the filter is reapplied each time one of these methods is called. Therefore, if records that previously did not match the filter have been modified so that they now match the filter, they will be included in the dataset before the search is performed.

Searching for data – DB/9

Chapter nine of the free Delphi Database Course for beginners. Walking through various methods of data seeking and locating while developing ADO based Delphi database applications.

A very common task for a database application is to search for a specific record based on some criteria. In Delphi, ADOExpress components implement record searching methods analogous to those found in the BDE approach. This chapter will walk you through various methods of data seeking and locating while developing ADO based Delphi database applications.

Note: the rest of this chapter deals with the aboutdelphi.mdb MS Access database that was introduced in the first chapter of this course. To use the code examples presented in this chapter, set up the data form with the core components (data–access and data–aware) as described in the previous chapters. The following code examples will assume that you are working with the ADOTable component pointing to the Applications table in our database.

When you think of it, a searching algorithm could look like: start at the top of the table, examine the field in each row – to see if it matches the criteria, stop the loop on the selected record or at the bottom row – whichever comes first.
Hopefully, Delphi hides those *steps* from us. There are several ways to locate a record in a dataset retrieved by an ADODataset (Table or Query) component.

Locate

This generic search method sets the current record to be the first row matching a specified set of search criteria. By using the Locate method we can look for values of one or more fields, passed in a variant array.
The next code puts the Locate method to work finding the first record that contains the string 'Zoom' in the Name field. If the call to Locate returns True – the record is found and is set to be the current one.

```
AdoTable1.Locate('Name','Zoom',[]);

{...or...}

var ffield, fvalue: string;
    opts : TLocateOptions;

ffield := 'Name';
fvalue := 'zoom';
opts := [loCaseInsensitive];

if not AdoTable1.Locate(ffield, fvalue, opts) then
  ShowMessage(fvalue + ' not found in ' + ffield);
```

Lookup

Lookup does not move the cursor to the matching row, it only returns values from it. Lookup returns a variant array containing the values from the fields specified in a semicolon–delimited list of field names whose values should be returned from the matching row. If there are no matching records, Lookup returns a Null variant.

The following code fills in a LookupRes variant array

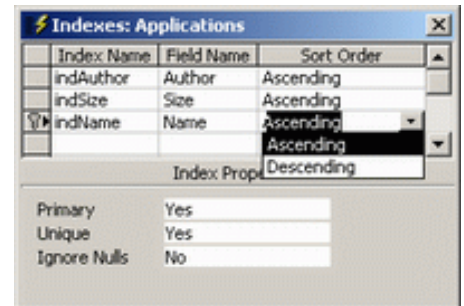
```
var LookupRes: Variant;

LookupRes := ADOTable1.Lookup
  ('Name', 'Zoom', 'Author; Description');

if not VarIsNull(LookupRes) then
 ShowMessage(VarToStr(LookupRes[0])) //author name
```

One advantage of the Locate and Lookup methods is that they *don't* require the table to be indexed. However, the Locate function will use the fastest method available to search the table; if a table is indexed, Locate will use the index.

Indexing



An index helps *find and sort* records faster. You can create indexes based on a single field or on multiple fields. Multiple–field indexes enable you to distinguish between records in which the first field may have the same value. In most cases you'll want to index fields you search/resort frequently. For example, if you search for specific application type in a Type field, you can create an index for this field to speed up the search for a specific type.
The primary key of a table is automatically indexed, and you can't index a field whose data type is OLE Object. Note that if many of the values in the field are the same, the index may not significantly speed up data retrieval.
The main drawbacks are that indexes consume additional disk space, and inserting, deleting and updating of data takes longer takes longer on indexed columns than on non indexed columns.

When working with a Table component and the BDE (not ADO) Delhi provides us with a number of functions that will search for values in a database table. Some of these are Goto, GoToKey, GoToNearest, Find, FindKey, Find Nearest, etc. For a complete reference see Delphi's help, topic: Searching for records based on indexed fields. The ADO approach *does not support* those methods, instead it introduces a *Seek* method.

Seek

The ADO datasets Seek method uses an index when performing a search. If you don't specify an index and you are working with an Access database, the database engine will use the primary key index. Seek is used to find a record with a specified value (or values) in the field (or fields) on which the current index is based. If Seek does not find the desired row, no error occurs, and the row is positioned at the end of the dataset. Seek returns a boolean value reflecting the success of the search: True if a record was found or False if no matching record was found.

The *GetIndexNames* method of a TADOTable component retrieves a list (for example: items of a combo box) of available indexes for a table.

```
ADOTable1.GetIndexNames(ComboBox1.Items);
```

The same list is available at design–time in the *IndexName* property of a TADOTable component. The *IndexFieldNames* property can be used as an alternative method of specifying the index to use for a table. In IndexFieldNames, specify the name of each field to use as an index for a table.

The Seek method has the following declaration:

```
function Seek(const KeyValues: Variant; SeekOption: TSeekOption = soFirstEQ): Boolean;
```

· *KeyValues* is an array of Variant values. An index consists of one or more columns and the array contains a value to compare against each corresponding column.

· *SeekOption* specifies the type of comparison to be made between the columns of the index and the corresponding KeyValues.

SeekOption	***Meaning***
soFirstEQ	Record pointer positioned at the first matching record, if one is found, or at the end of the dataset if one is not found
soLastEQ	Record pointer positioned at the last matching record, if one is found, or at the end of the dataset if one is not found.
soAfterEQ	Record pointer positioned at matching record, if found, or just after where that matching record would have been found.
soAfter	Record pointer positioned just after where a matching record would have been found.
soBeforeEQ	Record pointer positioned at matching record, if found, or just before where that matching record would have been found.
soBefore	Record pointer positioned just before where a matching record would have been found.

Note 1: the Seek method is supported only with server−side cursors. Seek is not supported when the dataset's *CursorLocation* property value is *clUseClient*. Use the Supports method to determine whether the underlying provider supports Seek.

Note 2: when you use the Seek method on multiple fields, the Seek fields must be in the same order as the fields in the underlying table. If they are not, the Seek method fails.

Note 3: you cannot use the Seek method on TADOQuery component.

To determine whether a matching record was found, we use the BOF or EOF property (depending on the search direction). The next code uses the index specified in the ComboBox to look for a value typed in the Edit1 edit box.

```
var strIndex: string;

strIndex := ComboBox1.Text; //from the code above

if ADOTable1.Supports(coSeek) then begin
 with ADOTable1 do begin
   Close;
   IndexName := strIndex;
   CursorLocation := clUseServer;
   Open;
   Seek (Edit1.Text, soFirstEQ);
  end;
  if ADOTable1.EOF then
   ShowMessage ('Record value NOT found');
end
```

ADO Cursors – DB/10

Chapter ten of the free Delphi Database Course for beginners. How ADO uses cursors as a storage and access mechanism, and what you should do to choose the best cursor for your Delphi ADO application.

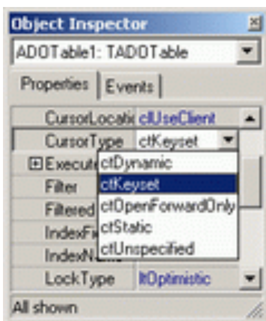
Welcome to the chapter ten of the free Delphi ADO Database Course for beginners. In the past nine chapters you were presented with some of the basic techniques when developing ADO–based Delphi applications. We've seen how several data–access components are used to connect and retrieve data from an Access database. One thing is for sure: ADOExpress components fit quite nicely into the Delphi data access model and map very closely to the basic data objects that ADO uses to access data.

However, the way you use ADOExpress components is quite different from the traditional Delphi programming with the BDE based TTable, and TQuery components. If you're accustomed to working with the BDE dataset components, there are a number of things you'll find different when you use ADO. The available properties are different, and so should be the programming style.

At the heart of ADO is the Recordset object. The Recordset object (aka Dataset) is a result of a Query command (SELECT statement of a TADOQuery component, for example). When an ADO–based application retrieves rows from a database, an ADO Recordset object encapsulates the data and the operations allowed on that data. ADO uses *cursors* to contain the logical set of rows maintained for a recordset. The cursor also provide the current position in the recordset. In development we use cursors to create a recordset to scroll forward or backward in, or to have the recordset pick up another user's changes.

Cursor?!

The simplest definition would be: a query result set where browsing is enabled and the current position is known.



Within ADO, cursors have three functions. First, the cursor type determines movement within the cursor and whether the recordset will reflect users changes. Second, the cursor location determines where to store the recordset while the cursor is open. Third, the cursor's locking type specifies how an ADO datastore will lock the rows when you want to make changes.

The understanding of cursors is extremely important. For example, in order to get our recordset to do the things we want, we need to open certain ADO recordsets with specific types of cursors. To RecordCount property, for example, is NOT supported with forward–only cursors.

In the ADOExpress set, TCustomADODataSet encapsulates a set of properties, events, and methods for working with data accessed through an ADO datastore. All the TCustomADODataSet descendant classes (TADODataSet, TADOTable, TADOQuery, and TADOStoredProc) share several common properties. Use the *CursorType*, *CursorLocation*, and *LockType* properties to create the most efficient recordset.

CursorType

Choosing the correct cursor has a direct impact on the success of your Delphi ADO–based application.

ADO provides four cursor options: dynamic, keyset, forward−only and static. Since each cursor type behaves differently, you will greatly benefit from understanding the capabilities of each one.

The CursorType property specifies how you move through the recordset and whether changes made on the database are visible to the recordset after you retrieve it. Delphi wraps ADO cursor types in the TCursorType.

ctDynamic
Allows you to view additions, changes and deletions by other users, and allows all types of movement through the Recordset that don't rely on bookmarks; allows bookmarks if the provider supports them. The Supports method of an ADODataset indicates whether a recordset supports certain types of operations. The following statement can be used to check if the provider supports bookmarks:
```
if ADOTable1.Supports(coBookmark) then ...
```
Choose dynamic cursors if multiple users insert, update, and delete rows in the database at the same time.

ctKeyset
Behaves like a dynamic cursor, except that it prevents you from seeing records that other users add, and prevents access to records that other users delete. Data change by other users will still be visible. It always supports bookmarks and therefore allows all types of movement through the Recordset.

ctStatic
Provides a static copy of a set of records for you to use to find data or generate reports. Always allows bookmarks and therefore allows all types of movement through the Recordset. Additions, changes, or deletions by other users will not be visible. A static cursor behaves like the result set from a BDE Query component with its RequestLive property set to False.

ctForward−only
Behaves identically to a dynamic cursor except that it allows you to scroll only forward through records. This improves performance in situations where you need to make only a single pass through a Recordset.

Note: only ctStatic is supported if the CursorLocation property of the ADO dataset component is set to clUseClient.

Note: if the requested cursor type is not supported by the provider, the provider may return another cursor type. That is, if you try to set CursorLocation to clUseServer and CursorType to ctDynamic, on an Access database, Delphi will change the CursorType to ctKeyset.

CursorLocation
The CursorLocation property defines where the recordset is created when it's opened — on the client or the server.

The data in a *client−side* cursor is "inherently disconnected" from the database. ADO retrieves the results of the selection query (all rows) and copies the data to the client before you start using it (into the ADO cursor). After you make changes to your Recordset, the ADO translates those changes into an action query and submits that query to your database through the OLE DB provider. The client−side cursor behaves like a local cache.
In most cases, a client−side cursor is preferred, because scrolling and updates are faster and more efficient, although returning data to the client increases network traffic.

Using the *server−side* cursor means retrieving only the required records, requesting more from the server as the user browses the data. Server−side cursors are useful when inserting, updating, or deleting records. This type of cursor can sometimes provide better performance than the client−side cursor, especially in situations where excessive network traffic is a problem.

You should consider a number of factors when choosing a cursor type: whether you're doing more data updates or just retrieving data, whether you'll be using ADO in a desktop application or in an Internet−based application, the size of your resultset, and factors determined by your data store and environment. Other factors might restrict you as well. For example, the MS Access doesn't support dynamic cursors; it uses keyset instead. Some data providers automatically scale the CursorType and CursorLocation properties, while others generate an error if you use an unsupported CursorType or CursorLocation.

LockType

The LockType property tells the provider what type of locks should be placed on records during editing. Locking can prevent one user from reading data that is being changed by another user, and it can prevent a user from changing data that is about to be changed by another user.

Modifying a record in an Access database locks some neighboring records. This is because Access uses, so called, page locking strategy. This means that if a user is editing a record, some other user won't be allowed to modify that record, or even to modify the next few records after or before it.

In Delphi, the TADOLockType specifies the types of locks that can be used. You can control row and page locking by setting the appropriate cursor lock option. To use a specific locking scheme, the provider and database type must support that locking scheme.

ltOptimistic
Optimistic locking locks the record only when it's physically updated. This type of locking is useful in conditions where there is only a small chance that a second user may update a row in the interval between when a cursor is opened and the row is finally updated. The current values in the row are compared with the values retrieved when the row was last fetched.

ltPessimistic
Pessimistic locking locks each record while it's being edited. This option tells ADO to get an exclusive lock on the row when the user makes any change to any column in the record. The ADOExpress components don't directly support pessimistic record locking because ADO itself does not have any way to arbitrarily lock a given record and still support navigating to other records.

ltReadOnly
Read only locking simply does not allow data editing. This lock is useful in conditions where your application must temporarily prevent data changes, but still can allow unrestricted reading. Read only locking with CursorType set to ctForwardOnly is ideal for reporting purposes.

ltBatchOptimistic
BatchOptimistic locking is used with disconnected recordsets. These recordsets are updated locally and all modifications are sent back to the database in a batch.

From Paradox to Access with ADO and Delphi – DB Course/Chapter 11

Chapter eleven of the free Delphi Database Course for beginners. Focusing on the TADOCommand components and using the SQL DDL language to help porting your BDE/Paradox data to ADO/Access.

Chapter 5 of this course (Free Delphi Database Course for Beginners – focus on ADO techniques) featured the ADOQuery component designed to enable Delphi developers to use the SQL language with ADO. The SQL statements can either be DDL (Data Definition Language) statements such as CREATE TABLE, ALTER INDEX, and so forth, or they can be DML (Data Manipulation Language) statements, such as SELECT, UPDATE, and DELETE.

In this chapter, I'll focus on the TADOCommand components and using the SQL DDL language to help **porting your BDE/Paradox data to ADO/Access.**

Data definition language

Creating a database programmatically isn't something most developers do every day – we all use some kind of visual tool, like MS Access for maintaining a MDB file. Unfortunately, sometimes you'll need to create and destroy databases and database objects from code. The most basic technique in use today is Structured Query Language Data Definition Language (SQL DDL). Data definition language (DDL) statements are SQL statements that support the definition or declaration of database objects (for example, CREATE TABLE, DROP TABLE, CREATE INDEX and similar).
My intention here is not to teach you the DDL language, if you are familiar with the SQL DML then DDL should be no barrier for you. Note that working with DDL can be quite tricky, every database vendor generally provides its own extensions to SQL.

Let's quickly take a look at a simple CREATE TABLE statement:

```
CREATE TABLE PhoneBook(
 Name TEXT(50)
 Tel TEXT(50)
);
```

This DDL statemnt (for MS Access), when executed, will create a new table named PhoneBook. The PhoneBook table will have two fields, Name and Tel. Both fields are of the string (TEXT) type and the size of the fields is 50 characters.

TFieldDef.DataType

Obviously, data type that represents a string in Access is TEXT. In Paradox it's STRING. In order to port Paradox tables to Access we'll have to know what data types are available and what are their names. When working with the BDE and Paradox tables, the *TFieldDef.DataType* determines the type of a physical field in a (dataset) table. To successfully migrate Paradox tables to Access you need to have a function that "transforms" a Paradox field type to an Access type.
The next function checks the type of the field (fd) and returns the corresponding Access type along with a field size when needed for a CREATE TABLE DDL statement.

```
function AccessType(fd:TFieldDef):string;
begin
 case fd.DataType of
  ftString: Result:='TEXT('+IntToStr(fd.Size)+')';
  ftSmallint: Result:='SMALLINT';
  ftInteger: Result:='INTEGER';
  ftWord: Result:='WORD';
  ftBoolean: Result:='YESNO';
  ftFloat : Result:='FLOAT';
  ftCurrency: Result := 'CURRENCY';
```

```
  ftDate, ftTime, ftDateTime: Result := 'DATETIME';
  ftAutoInc: Result := 'COUNTER';
  ftBlob, ftGraphic: Result := 'LONGBINARY';
  ftMemo, ftFmtMemo: Result := 'MEMO';
 else
  Result:='MEMO';
 end;
end;
```

ADOX

ADO Extensions for Data Definition Language and Security (ADOX) is an extension to the ADO objects and programming model. ADOX gives developers a rich set of tools for gaining access to the structure, security model, and procedures stored in a database.

To use ADOX in Delphi, you should establish a reference to the ADOX type library.
1. Select Project | Import Type Library
3. Choose "Microsoft ADO Ext 2.x for DDL and Security (Version 2.x)"
4. Change "TTable" to "TADOXTable"
5. Change "TColumn" to "TADOXColumn"
6 .Change "TIndex" to "TADOXIndex"
7. Press Install button (rebuilding packages)
8. Press OK once and Yes twice
9. File | Close All | Yes

The top–level object in the ADOX object model is the Catalog object. It provides access to the Tables, Views, and Procedures collections, which are used to work with the structure of the database, and also provides the Users and Groups collections, which are used to work with security. Each Catalog object is associated with only one Connection to an underlying data source.

We'll leave ADOX (at least for now) and stick to ADOExpress.

TADOCommand

In ADOExpress, the *TADOCommand* component is the VCL representation of the ADO Command object. The Command object represents a command (a query or statement) that can be processed by the data source. Commands can then be executed using the ADOCommand's *Execute* method. TADOCommand is most often used for executing data definition language (DDL) SQL commands. The *CommandText* property specifies the command to execute. The *CommandType* property determines how the CommandText property is interpreted. The *cmdText* type is used to specify the DDL statement. Although it makes no sense to use the ADOCommand component to retrieve a dataset from a table, query, or stored procedure, you can do so.

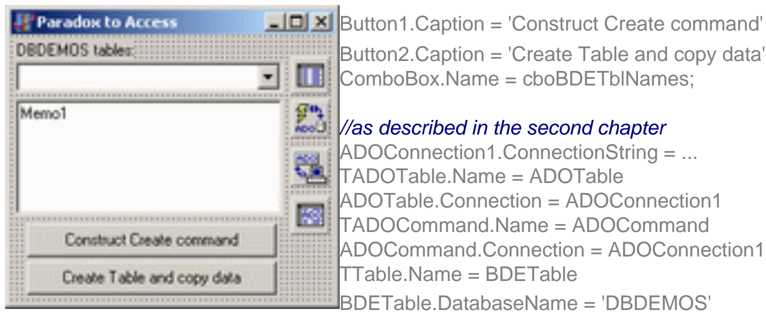
It's time for some real code...
The following project will demonstrate how to:

- get the list of all tables in a BDE alias
- use TFieldDefs in order to retrieve the definition (name, data type, size, etc.) of fields in a table.
- create a CREATE TABLE statement
- copy data from BDE/Paradox table to ADO/Access table.

Basically what we want to do is to copy several tables from DBDemos to our aboutdelphi.mdb Access database. The structure of the aboutdelphi.mdb is discussed in the first chapter.

Let's do it step by step:
GUI
Start Delphi – this creates a new project with one blank form. Add two Buttons, one ComboBox and one Memo component. Next, add a TTable, TADOTable, TADOConnection and a TADOCommand component. Use the Object Inspector to set the following properties (leave all the other properties as

they are – for example the Memo should have the default name: Memo1):



Code

To retrieve a list of the tables associated with a given database (DBDEMOS) we use the next code (OnCreate for the form):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 Session.GetTableNames('DBDEMOS',
                       '*.db',False, False,
                       cboBDETblNames.Items);
end;
```

When you start the project the ComboBox has all the (Paradox) table names in the DBDEMOS alias directory. In the code that follows, we'll pick the Country table.

The next task is to create a CREATE TABLE DDL statement. This gets done in the 'Construct Create command' button's OnClick procedure:

```
procedure TForm1.Button1Click(Sender: TObject);
//'Construct Create command' button
var i:integer;
    s:string;
begin
 BDETable.TableName:=cboBDETblNames.Text;
 BDETable.FieldDefs.Update;

 s:='CREATE TABLE ' + BDETable.TableName + ' (';
 with BDETable.FieldDefs do begin
  for i:=0 to Count-1 do begin
   s:=s + ' ' + Items[i].Name;
   s:=s + ' ' + AccessType(Items[i]);
   s:=s + ',';
  end; //for
  s[Length(s)]:=')';
 end;//with

 Memo1.Clear;
 Memo1.lines.Add (s);
end;
```

The above code simply parses the field definitions for the selected table (cboBDETblNames) and generates a string that will be used by the CommandText property of the TADOCommand component.

For example, when you select the Country table the Memo gets filled with the next string:

```
CREATE TABLE country (
 Name TEXT(24),
 Capital TEXT(24),
 Continent TEXT(24),
```

```
  Area FLOAT,
  Population FLOAT
)
```

And finally, the code for the 'Create Table and copy data' button drops a table (DROP..EXECUTE), creates a table (CREATE..EXECUTE), and then copies data into the new table (INSERT...POST). Some error handling code is provided, but the code will fail if, for example, the (new) table does not already exist (since it first gets dropped).

```
procedure TForm1.Button2Click(Sender: TObject);
//'Create Table and copy data' button
var i:integer;
    tblName:string;
begin
 tblName:=cboBDETblNames.Text;

//refresh
 Button1Click(Sender);

//drop & create table
 ADOCommand.CommandText:='DROP TABLE ' + tblName;
 ADOCommand.Execute;

 ADOCommand.CommandText:=Memo1.Text;
 ADOCommand.Execute;

 ADOTable.TableName:=tblName;

//copy data
 BDETable.Open;
 ADOTable.Open;
 try
  while not BDETable.Eof do begin
   ADOTable.Insert;
   for i:=0 to BDETable.Fields.Count-1 do begin
    ADOTable.FieldByName
   (BDETable.FieldDefs[i].Name).Value :=
      BDETable.Fields[i].Value;
   end;//for
   ADOTable.Post;
   BDETable.Next
  end;//while
 finally
  BDETable.Close;
  ADOTable.Close;
 end;//try
end;
```

That's it. Check out your Access database now...voila there is a Country table with all the data from DBDEMOS.

Now you can port all your Paradox tables to Access. Few questions, however, stay unanswered. The first one is: how to add index definitions (CREATE INDEX ON ...) to tables. The second one is: how to create an empty Access database. I'll leave those (and others you can think of) for the Forum or for some future article.

Master detail relationships – DB/12

Chapter twelve of the free Delphi Database Course for beginners. How to use parent−child database relationships to deal effectively with the problem of joining two database tables to present information.

Master−detail data relationships are a fact of life for every Delphi database developer; just as data relationships are a fundamental feature of relational databases.

In the previous chapters of this course, we've invariably used only one table from our "demo" aboutdelphi.mdb MS Access database. In real time database programming, the data in one table is related to the data in other tables. In general, tables can be related in one of three different ways: one−to−one, one−to−many or many−to−many. This chapter will show you how to use one−to−many database relationships to deal effectively with the problem of joining two database tables to present information.

A one−to−many relationship, often referred to as a "master−detail" or "parent−child" relationship, is the most usual relationship between two tables in a database.

Common scenarios include customer/purchase data, patient/medical−record data, and student/course−result data. For example, each customer is associated with at least one order record. Valued customers have many order records involving significant sums and often a user needs to view one in connection with the other. In a one−to−many relationship, a record in Table A can have (none or one or) more than one matching record in Table B, but for every record in Table B there is exactly one record in Table A.

A typical master−detail data browsing form displays the results of a one−to−many relationship, where one DBGrid displays (or set of data enabled controls) the results of the first or master table. It then tracks a selection in the first DBGrid to filter the results of a second table used to display the details of the selection in the second DBGrid.

When working with the BDE and Delphi, the simplest way to assemble a master−detail form is to use the Database Form Wizard. Wizard simplifies the steps needed to create a tabular or data−entry form by use of an existing database, unfortunately it is designed to use the BDE versions of TTable and TQuery components. Everything the wizard does, we can do by hand.

Since, through this course, we are working with the ADOExpress set of Delphi components, we'll need to set all the components step by step. Firstly we have to make sure that we have two tables in a master−detail relationship.

MS Access relationships

Our focus will be on the following two tables: Customers and Orders. Both tables are a part of the DBDEMOS database that comes with Delphi. Since both tables are Paradox tables, we'll use the code from the previous article to port them to our working aboutdelphi.mdb MS Access database.

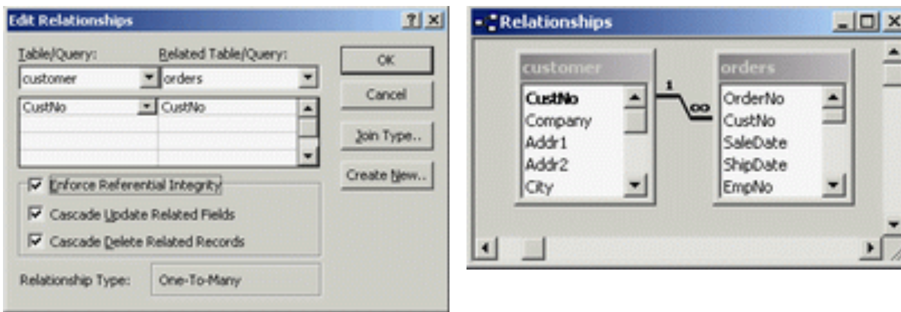
Notice that when you port those tables to Access both of them have no index or primary key nor are they linked in any way in Access.

The power in a relational database management system such as MS Access comes from its ability to quickly find and bring together information stored in separate tables. In order for MS Access to work most efficiently, each table in your database should include a field or set of fields that uniquely identifies each individual record stored in the table. If two tables are liked in a relation (of any kind) we should set that relation with the MS Access.

Customers−Orders relation

To set up the relationship, you add the field or fields that make up the primary key on the "one" side of

the relationship to the table on the "many" side of the relationship. In our case, you would add the CustNo field from the Customers table to the Orders table, because one customer has many orders. Note that you have to set the CustNo in Customers to be the primary key for the table.



When creating a relation between two tables MS Access provides us with the Referential Integrity feature. This feature prevents adding records to a detail table for which there is no matching record in the master table. It will also cause the key fields in the detail table to be changed when the corresponding key fields in the master are changed – this is commonly referred to as a cascading update. The second options is to enable cascading deletes. This causes the deletion of all the related records in a detail table when the corresponding record in the master table gets deleted. *These events occur automatically, **requiring no intervention** by a Delphi application using these tables.*

Now, when we have all the relations set up, we simply link few data components to create a master–detail data browsing Delphi form.

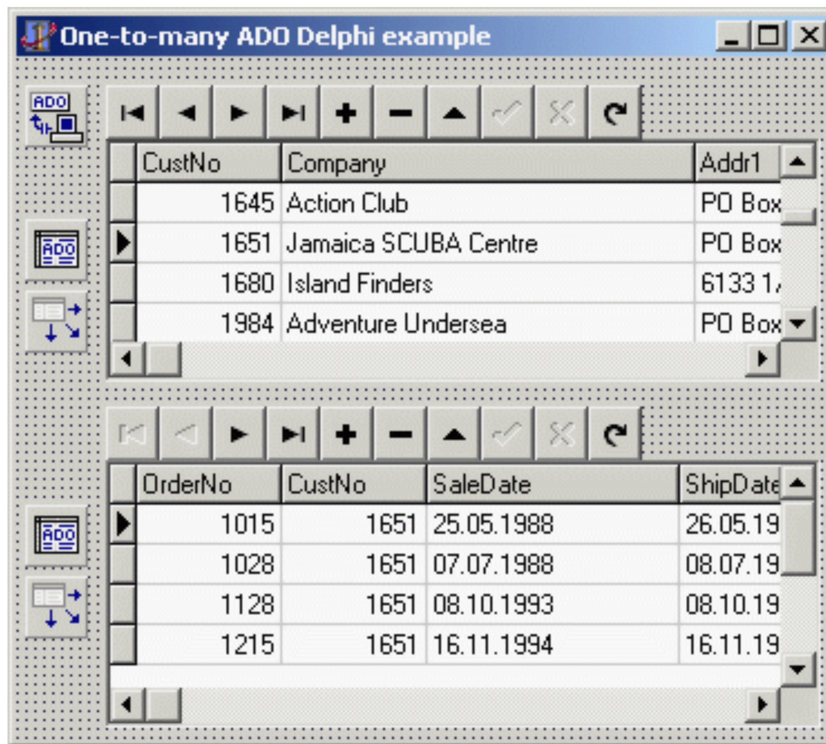
Setting up Master/Detail with ADOExpress

Creating a master–detail data form is not to much complicated. Have an empty Delphi form, and just follow the steps:

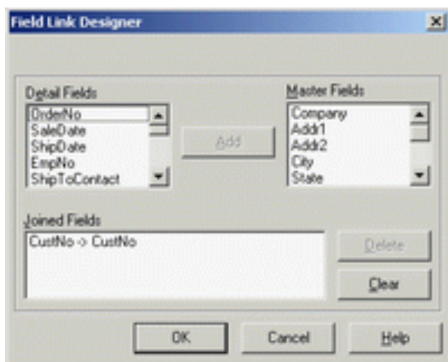
1. Select the ADO page on the Component palette. Add two TADOTable components and one TADOConnection to a form.
2. Select the Data Access page on the Component palette. Add two TDataSource components to a form.
3. Select Data Controls page on the Component palette. Place two TDbGrid components on a form. Add two DBNavigator components, too.
4. Use the ADOConnection, the ConnectionString property, to link to the aboutdelphi.bdb MS Access database, as explained in the first chapter of this course.
5. Connect DBGrid1 with DataSource1, and DataSource1 with ADOTable1. This will be the master table. Connect DBNavigator1 with DataSource1.
6. Connect DBGrid2 with DataSource2, and DataSource2 with ADOTable2. This will be the detail table. Connect DBNavigator2 with DataSource2.
7. Set ADOTable1.TableName to point to the Customers table (master).
8. Set ADOTable2.TableName to point to the Orders table (detail).

If you, at this moment, set the Active property of both ADOTable components to true, you'll notice that the entire Orders table is displayed – this is because we haven't set up the master–detail relationship yet.

Your form should look something like:



MasterSource and MasterFields
The MasterSource and MasterFields properties of the TADOTable component define master–detail relationships in Delphi/ADO database applications.



To create a master–detail relationships with Delphi, you simply need to set the detail table's MasterSource property to the DataSource of the master table and its MasterFields property to the chosen key field in the master table.

In our case, first, set ADOTable2.MasterSource to be DataSource1. Second, activate the Field Link Designer window to set the MasterFields property: in the Detail Fields list box and the Master Fields list box select the CustNo field. Click Add and OK.

These properties keep both tables in synchronization, so as you move through the Customers table, the Orders table will only move to records which match the key field (CustNo) in the Customers table.

Each time you highlight a row and select a new customer, the second grid displays only the orders pertaining to that customer.

When you delete a record in a master table – all the corresponding record in the detail table are deleted. When you change a linked field in a record in a master table – the corresponding field in the detail table gets changed to (in as many records as needed).

Simple as that!

Stop. Note that creating a master–detail form with Delphi is not enough to support referential integrity features on two tables. Even though we can use methods described here to display two tables in a parent–child relation; if those two tables are not linked (one–to–many) within MS Access – cascading

updates and deletes won't take place if you try to delete or update the "master" record.

ADO Shaping

Shaped recordsets are an alternative to master−detail relationships. Beginning with ADO 2.0, this method is available. Shaped recordsets allow the developer to retrieve data in a hierarchical fashion. The shaped recordset adds a special "field" that is actually a recordset unto itself. Essentially, data shaping gives you the ability to build hierarchical recordsets. For instance, a typical hierarchical recordset might consist of a parent recordset with several fields, one of which might be another recordset.

For an example of the SHAPE command take a look at the shapedemo project that shiped with Delphi (in the Demos\Ado directory). You must specify the shaping provider in your connection string, by adding Provider=MSDataShape; to the beginning.

```
SHAPE {select * from customer}
APPEND ({select * from orders} AS Orders
RELATE CustNo TO CustNo)
```

Although it takes some time to master the SHAPE command that's used to create these queries, it can result in significantly smaller resultsets. Data shaping reduces the amount of traffic crossing a network, provides more flexibility when using aggregate functions, and reduces overhead when interfacing with leading−edge tools like XML.

New...Access Database from Delphi – DB Course/Chapter 13

Chapter thirteen of the free Delphi Database Course for beginners. How to create an MS Access database without the MS Access. How to create a table, add an index to an existing table, how to join two tables and set up referential integrity. No MS Access, only Pure Delphi code.

Chapter 11 of this course (Free Delphi Database Course for Beginners – focus on ADO techniques) featured the ADOCommand component which is most often used for executing data definition language (DDL) SQL commands. We've presented a way of porting your existing Paradox/BDE tables to MS Access. However few questions have stayed unanswered: how to create an empty Access database, how to add an index to an existing table, how to join two tables with referential integrity, ...

Many of the attendees of this course have complained that they do not have MS Access installed on their computer – and are unable to create a sample database (aboutdelphi.mdb) that is presented in the first chapter and used through this course.

In this chapter, we'll again focus on the TADOCommand and the ADOX to see how **set up an empty MS Access database from "nothing"**.

ADOX

As stated in the mentioned chapter, ADO Extensions for Data Definition Language and Security is an extension to the ADO objects and programming model. ADOX gives developers a rich set of tools for gaining access to the structure, security model, and procedures stored in a database.

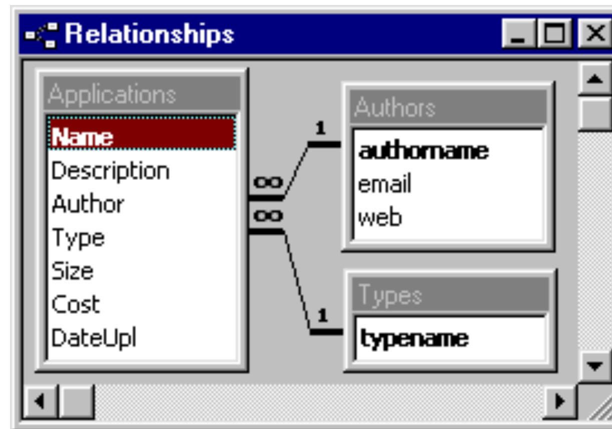
Even though ADOX is part of ADO, Delphi does not wrap it inside ADOExpress. To use ADOX with Delphi, you should establish a reference to the ADOX type library. The description of the ADOX library is "Microsoft ADO Ext. for DDL and Security." The ADOX library file name is Msadox.dll. You'll have to import the ADOX library into the IDE.

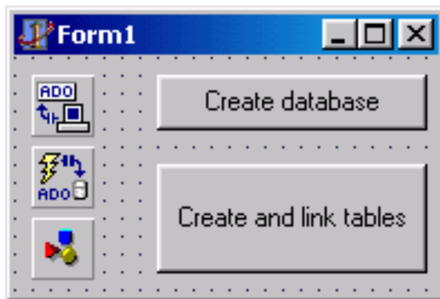
To use ADOX in Delphi, you should establish a reference to the ADOX type library.
1. Select Project | Import Type Library
2. Choose "Microsoft ADO Ext 2.x for DDL and Security (Version 2.x)"
3a. Change "TTable" to "TADOXTable"
3b. Change "TColumn" to "TADOXColumn"
3c. Change "TIndex" to "TADOXIndex"
3d. Change "TKey" to "TADOXKey"
3e. Change "TGroup" to "TADOXGroup"
3f. Change "TUser" to "TADOXUser"
3g. Change "TCatalog" to "TADOXCatalog"
4. Press Install button (rebuilding packages)
5. Press OK once and Yes twice
6. File | Close All | Yes

This process creates a ADOX_TLB.pas unit and places its name in the uses clause of the current project. The process also places 7 new components on the ActiveX page of the component palette. It is very important to change the class names as described in step 3. If you omit that part Delphi will complain that class TTable is already defined – this is because the VCL already has the BDE version of TTable component.

aboutdelphi.mdb

Before we move on, you should recall that our sample aboutdelphi.mdb database has three tables: Application, Authors and Types. Both Authors and Types are child tables to Applications. Both Authors and Types have a primary index.





The Delphi Project
Our task is to have Delphi do all the work. We want to create a new database from code, add all three tables from code, add indexes from code and even set up a referential integrity between those tables – again from code.

As usual, have an empty Delphi form. Add two button component. Add a TADOConnection, TADOCommand. We'll use TADOCommand with a DDL language to create and link tables. Add a TADOXCatalog component (ActiveX page). The TADOXCatalog will do the trick of creating a new database. Let the name of the first button be *btnNewDatabase* (caption: 'Create database'), the second one should be called *btnAddTables* (caption: 'Create and link tables'). All the other components should have the default name.
In this chapter we'll link those components from code. Therefore, you do not need to set up a ConnectionString for the ADOConnection component and the Connection property for the ADOCOmmand component now.

New...Database
Before we move on to creating tables and linking them we have to create a new (empty) database. This is the code:

```
procedure TForm1.btnNewDatabaseClick(Sender: TObject);
var
 DataSource : string;
 dbName     : string;
begin
 dbName:='c:\aboutdelphi.mdb';

 DataSource :=
    'Provider=Microsoft.Jet.OLEDB.4.0' +
    ';Data Source=' + dbName +
    ';Jet OLEDB:Engine Type=4';

  ADOXCatalog1.Create1(DataSource);
end;
```

Would you belive – simple as that. Obviously the ADOXCatalog has a method called Create1 that creates a new database. Pretty unusual since we've accustomed that Create methods are used to create an object from a class. The ADOXCatalog really has a Create method which has nothing in common to Create1.

The variable DataSource looks pretty much like a standard connection string for the TADOConnection component. There is only one addition, the *Jet OLEDB:Engine Type=X* part. Engine type 4 is used to

create an MS Access 97 database, type 5 is for MS Access 2000.

Note that the above code does not check for the existance of the c:\aboutdelphi.mdb database. If you run this code twice it will complain that the databae already exists.

Add table, create index, set referential integrity

The next step is to create all tables (three of them), add indexes, and create referential integrity. Even though we could use ADOX, that is, TADOXTable, TADOXKey, etc. I'm somehow more familiar with the (standard) DDL language and the TADOCommand component. Back in the chapter 11 of this course we discussed database tables porting issues. This time we'll create tables from nothing.
The following peaces of code are to be placed inside the button's btnAddTables OnClick even handler, I'll slice the code and add some explanations in between.

First, we need to connect to the newly created database with the TADOConnection component. Since we've left the ADOCommand unattached to ADOConnection – we'll link them from code (this should be obvious by now):

```
procedure TForm1.btnAddTablesClick(Sender: TObject);
var
  DataSource : string;
  cs         : string;
begin
 DataSource :=
     'Provider=Microsoft.Jet.OLEDB.4.0'+
     ';Data Source=c:\aboutdelphi.mdb'+
     ';Persist Security Info=False';

 ADOConnection1.ConnectionString := DataSource;
 ADOConnection1.LoginPrompt := False;
 ADOCommand1.Connection := ADOConnection1;
 ...
```

Second, we create both Types and Authors tables, the structures are given in the first chapter. To build a new table with DDL by using the Jet SQL, we use the CREATE TABLE statement by providing it the name the table, name the fields, and fiedl type definitions. Then, the Execute method of the ADOCommand component is used.

```
...
 cs:='CREATE TABLE Types (typename TEXT(50))';
 ADOCommand1.CommandText := cs;
 ADOCommand1.Execute;


 cs:='CREATE TABLE Authors (' +
       'authorname TEXT(50),' +
       'email TEXT(50),' +
       'web TEXT(50))';
 ADOCommand1.CommandText := cs;
 ADOCommand1.Execute;
...
```

Next, we add indexes to those two tables. When you apply an index to a table, you are specifying a certain arrangement of the data so that it can be accessed more quickly. To build an index on a table, you must name the index, name the table to build the index on, name the field or fields within the table to use, and name the options you want to use. You use the CREATE INDEX statement to build the index. There are four main options that you can use with an index: PRIMARY, DISALLOW NULL, IGNORE NULL, and UNIQUE. The PRIMARY option designates the index as the primary key for the table.

```
...
```

```
 cs:='CREATE INDEX idxPrimary '+
     'ON Types (typename) WITH PRIMARY';
 ADOCommand1.CommandText := cs;
 ADOCommand1.Execute;

 cs:='CREATE INDEX idxPrimary '+
     'ON Authors (authorname) WITH PRIMARY';
 ADOCommand1.CommandText := cs;
 ADOCommand1.Execute;
...
```

Finally, we add the last table. Applications table is linked with both Types and Authors in a master detail relationship. Back in the last chapter we were discussing one–to–many relationships that define the following: for every record in the master table, there are one or more related records in the child table. In our case, one Author (Authors table) can post more Applications; and the Application can be of some type.
When defining the relationships between tables, we use the CONSTRAINT declarations at the field level. This means that the constraints are defined within a CREATE TABLE statement.

```
...
 cs:='CREATE TABLE Applications ('+
     '  Name TEXT(50),'+
     '  Description TEXT(50),'+
     '  Author TEXT(50) CONSTRAINT idxauthor '+
          'REFERENCES Authors (authorname),'+
     '  Type TEXT(50) CONSTRAINT idxtype '+
          'REFERENCES Types (typename),'+
     '  Size FLOAT,'+
     '  Cost CURRENCY,'+
     '  DateUpl DATETIME,'+
     '  Picture LONGBINARY)';

 ADOCommand1.CommandText := cs;
 ADOCommand1.Execute;

end;//btnAddTablesClick
```

That's it. Now run the project, click the btnNewDatabase button, click the btnAddTables button and you have a new (empty) aboutdelphi.mdb database in the root of the C disk. If you have MS Access installed on your system you can open this database with it and check that all thje tables are here and in the Relationships window all the tables are linked.

Charting with Databases – DB Course/Chapter 14

Chapter fourteen of the free Delphi Database Course for beginners. Introducing the TDBChart component by integrating some basic charts into a Delphi ADO based application to quickly make graphs directly for the data in recordsets without requiring any code.

In most modern database applications some kind of graphical data representation is preferable or even required. For such purposes Delphi includes several data aware components: DBImage, DBChart, DecisionChart, etc. The DBImage is an extension to an Image component that displays a picture inside a BLOB field. Chapter 3 of this database course discussed displaying images (BMP, JPEG, etc.) inside an Access database with ADO and Delphi. The DBChart is a data aware graphic version of the TChart component.

Our goal in this chapter is to introduce the TDBChart by showing you how to integrate some basic charts into your Delphi ADO based application.

TeeChart
The DBChart component is a powerful tool for creating database charts and graphs. It is not only powerful, but also complex. We wont be exploring all of its properties and methods, so you'll have to experiment with it to discover all that it is capable of and how it can best suite your needs. By using the DBChart with the TeeChart charting engine you can quickly make graphs directly for the data in datasets without requiring any code. TDBChart connects to any Delphi DataSource. ADO recordsets are natively supported. No additional code is required – or just a little as you'll see. The Chart editor will guide you through the steps to connect to your data – you don't even need to go to the Object Inspector.
Runtime TeeChart libraries are included as part of Delphi Professional and Enterprise versions. TChart is also integrated with QuickReport with a custom TChart component on the QuickReport palette. Delphi Enterprise includes a DecisionChart control in the Decision Cube page of the Component palette.

Let's chart! – Prepare

Our task will be to create a simple Delphi form with a chart filled with values from a database query. To follow along, create a Delphi form as follows:

1. Start a a new Delphi Application – one blank form is created by default.
2. Place the next set of components on the form: ADOConnection, ADOQuery, DataSource, DBGrid and a DBChart.
3. Use the Object Inspector to connect ADOQuery with ADOConnection, DBGrid with DataSource with ADOQuery.
4. Set up a link with our demo database (aboutdelphi.mdb) by using the ConnectionString of the ADOConnection component.
5. Select the ADOQuery component and assign the next string to the SQL property:

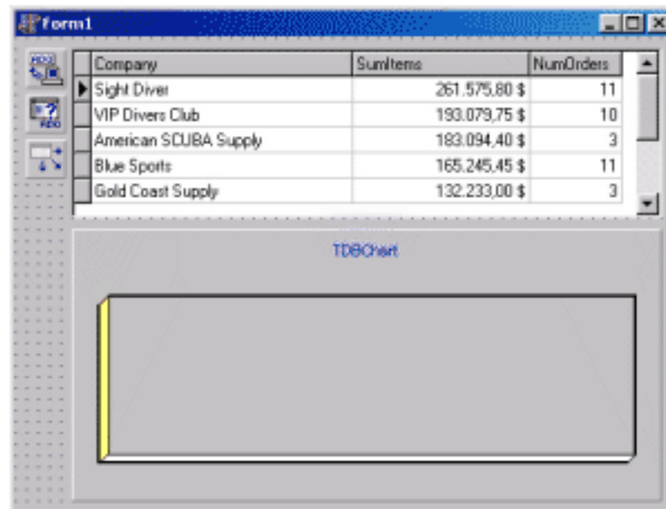
```
SELECT TOP 5 customer.Company,
SUM(orders.itemstotal) AS SumItems,
COUNT(orders.orderno) AS NumOrders
FROM customer, orders
WHERE customer.custno = orders.custno
GROUP BY customer.Company
ORDER BY SUM(orders.itemstotal) DESC
```

This query uses two tables: orders and customer. Both tables were imported from the (BDE/Paradox) DBDemos database to our demo (MS Access) database back in the chapter 11. This query results in a recordset with only 5 records. The first field is the Company name, the second (SumItems) is a sum of all the orders made by the company and the third field (NumOrders) represents the number of orders

that were made by the company. Note that those two tables are linked in a master–detail relationship.

6. Create a persistent list of database fields. (To invoke the Fields Editor double click the ADOQuery component. By default, the list of fields is empty. Click Add to open a dialog box listing the fields retrieved by the query (Company, NumOrders, SumItems). By default, all fields are selected. Select OK.) Even though you don't need a persistent set of fields to work with a DBChart component – we'll create it now. The reasons will be explained later.
7. Set ADOQuery.Active to True in the Object Inspector to see the resulting set at design time.

The form should look something like:



Note that the DBChart is "empty". We have not connected the recordset with the chart, yet.

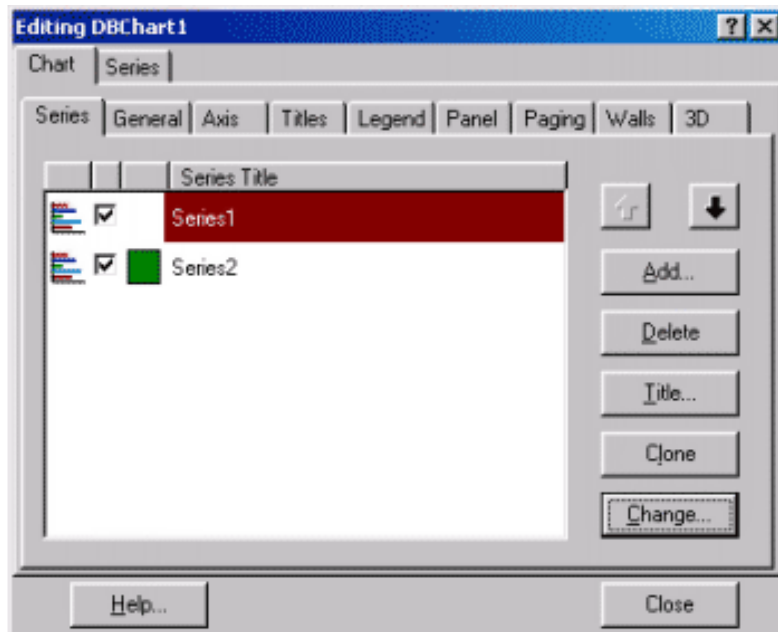
Let's chart! – Connect

In most cases, connecting a recordset with a DBChart and setting various properties can be done using the specific Chart editor.

Using the fields of a recordset for the source, the chart becomes a dynamic display that is updated as the recordset if modified. You have a wide variety of chart type options to choose from and all of the parameters of the display can be controlled within an application.

In general, you fill a chart with data designated in one or more series; each series consists of categories and values. For example, to fill a chart with data about customer orders, you could add a series that charts customer names as the categories and order amounts as the values.

The Chart editor is a one stop shop for all Chart and Series specific parameters. Once you have the DBChart component on a form, you should create one or more series. To accomplish this, open the Chart Component Editor: select the component, right–click it, and choose the Edit Chart menu item. Use the Chart tab to define all you general chart parameters. Select the Series tab and you may choose from your list of series to modify series specific features.



We'll add two chart series. One that presents the sum of all the orders per company and second that will plot the total number of orders.

Chart tab

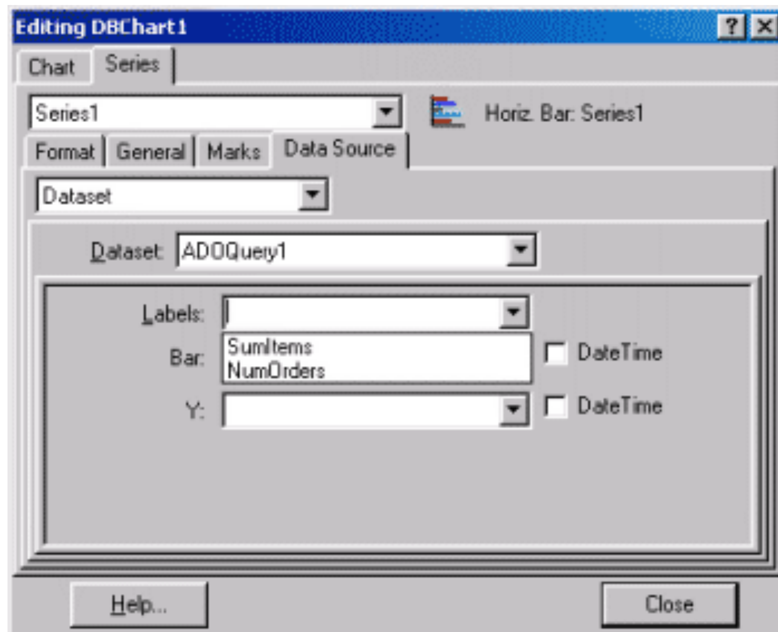
To add a data series press the Add button in the Series tab section of the Chart page. Now simply choose the graph you want to add from Gallery – we'll use the Horizontal Bar. Once we've added the Series to the Chart some random values are displayed, in the Chart at design–time, to easily follow any changes you are making.

Note that a chart can have multiple series; if they are all of the same type they will integrate better, as in the case of multiple bars.

Now add a second series – again use the Horizontal Bar.

Series tab

Selecting the Series tab (or double clicking the desired Series) allows you to edit your Series. Select Series1. Select the DataSource tab. Select "Dataset" from the drop down list. Select AdoQuery1 for the dataset. The next set of choices determine the way how the data form the datasource will appear on the graph (different kinds of charts have different properties and methods). The exact contents of the page will change depending on the Series type you have chosen. The Bar property determines what value each bar is representing – pick SumItems from the list. The Labels property is used to draw labels for each of the bars in the graph. Note that Lables drop down does not list the Company field from the recordset!



BUG! When working with ADO recordset the Chart Editor does not list fields of the WideString type (for the XLabels property). This bug is present in TeeChart v4.xx STANDARD that ships with Delphi 5 and Delphi 6. This is what I got as a reply from the authors of the TeeChart:
"Well, all Delphi versions use the same TeeChart version (v4 STANDARD, I think D4–D6 and CB4–CB5 use the same version). We discovered this bug only when Delphi 5 was released. We fixed it, but not for TeeChart v4 STANDARD version (only for new v5 PRO version).
The good news is user can still connect ADOQuery to specific series at runtime (via code):"

```
Series1.DataSource := ADOQuery1;
Series1.XLabelsSource := ADOQuery1WideStringField1.FieldName;
Series1.XValues.ValueSource := ADOQuery1FloatField1.FieldName;
Series1.YValues.ValueSource := ''; { no YValues }
Series1.CheckDataSource;
```

I believe we fixed it in TeeChart v5.01 (v5.00) release...
Upgrading to the 5.01 release is not free.

Let's chart! – Code

Ok, let's see what we can do about this bug. It seems that all the properties of the chart can be set with the Chart Editor except those related to recordset. We'll simply (as suggested) set all from code. Add the next code to the form's OnCreate even handler:

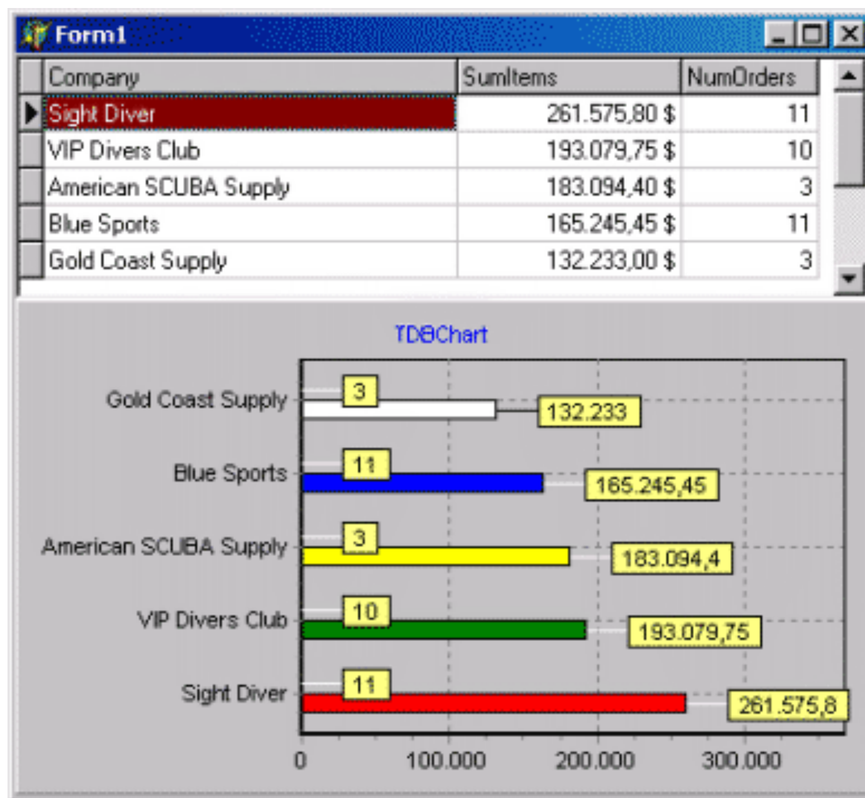
```
ADOQuery1.Close;
DBChart1.Legend.Visible:=False;
with DBChart1.SeriesList.Series[0] do begin
 DataSource := ADOQuery1;
 XLabelsSource := ADOQuery1Company.FieldName;
 XValues.ValueSource := ADOQuery1SumItems.FieldName;
 YValues.ValueSource := '';
 Marks.Style := smsXValue;
 CheckDataSource;
end; //with
with DBChart1.SeriesList.Series[1] do begin
 DataSource := ADOQuery1;
 XLabelsSource := '';
 XValues.ValueSource := ADOQuery1NumOrders.FieldName;
 YValues.ValueSource := '';
 CheckDataSource;
end; //with
```

```
ADOQuery1.Open;
```

Of course, I suggest you to browse through the Help system in order to find out about the commands, properties and methods used in this code.

Minor adjustments and notes

If you run the application, you'll notice that the chart is actually three–dimensional – set View3D to False – to display in two dimensions. You can prevent a legend from appearing in your chart by setting its Legend.Visible to False. The default value for Title is 'TDBChart' – change it as you like. Chart uses the current recordset order (ORDER BY keyword in the SQL statement) to populate Series points. This order can be overridden setting the Series values Order property.



That's it. It's not to easy, it's not to hard. Explore the Project's Code to find and learn more.

Lookup! – DB Course/Chapter 15

Chapter fifteen of the free Delphi Database Course for beginners. See how to use lookup fields in Delphi to achieve faster, better and safer data editing. Also, find how to create a new field for a dataset and discuss some of the key lookup properties. Plus, take a look at how to place a combo box inside a DBGrid.

Back in the master–detail relationships chapter of this course, we've seen that in most cases one database table is related to the data in one or more other tables.

Consider the next example. In our sample aboutdelphi.mdb database, Applications table lists Delphi applications uploaded by the visitors to this site. Among other fields, the Author and Type fields are linked with the corresponding fields in the Authors and Types tables. Only values from the AuthorName field in the Authors table can appear in the Author field of the Applications table. The same rule is applied to the TypeName field (Types table) and the Type field (Applications table).
Another situation: consider an application data entry form whose fields are connected to the Applications table. Let's say that this table has only one information related to the author of the application, an AuthorNo field corresponding to authors unique number. The Authors table, on the other hand, contains an AuthorNo field corresponding to authors UN, and also contains additional information, such as the authors name, email and a web page. It would be convenient if the data entry form enabled a user to select the author by its name (and email) instead by its UN.

Lookup!
If you have a Delphi form with data controls designed to allow editing the Applications table, you have to make sure that only TypeName values from the Types table can be applied to the Types field of the Application table. You also have to make sure that only AuthorName values from the Authors table can be applied to the Author field of the Application table. The best way to make this sure is to provide users with a string list to select the values from rather than having them enter values manually.

Both TDBLookupListBox and TDBLookupComboBox are data–aware controls that enable us to choose a value from another table or from a predefined list of values. This pair of components are so similar that it makes sense to discuss only one of them. Both components are designed to list items from either a dataset or from a secondary datasource. We'll be looking at the TDBLookupComboBox component.

In this chapter, we'll see how to use lookup fields in Delphi to achieve faster, better and safer data editing. We'll also see how to create a new field for a dataset and discuss some of the key lookup properties. Plus, take a look at how to place a combo box inside a DBGrid.

There are three ways you can set a lookup field in Delphi.

1. If you have a data form where all editings are done in a DBGrid, the best is to *create* a new (lookup) field for the dataset.
2. If a form hosts a set of data controls (DBEdit, DBComboBox, etc.), it makes sense to just use DBLookupComboBox without creating a new field.
3. The last one is to use columns of a DBGrid with its PickList property, again without creating a new field. This approach is not a true lookup approach but you'll see that it can act as one. The PickList list values that the user can select for the field value – just like a standard DBComboBox – but inside a DBGrid. We'll make it to list values from another dataset, thus defining sa lookup.

We'll discuss each of them, but we first need to build a data entry form.

Creating a data entry form

Creating a data editing form by hand is not to much complicated, as we already know. When developing database applications with Delphi (and ADO), most of the work is done inside the IDE by simply connecting various components together, thus having to write no code. A typical data

browsing/editing form presents a database table inside a DBGrid. Another way is to add several data aware controls to a form and link them to the data source. We'll place both a DBGrid and several data aware controls.

Have a new Delphi project with an empty form, then add the next set of components: one AdoConnection, one DataSource and two ADOTables.
Use the Object Inspector and connect all those components in the following way:
First set the name of the ADOConnection component to be ADOConnection. Use ConnectionString property to link to our aboutdelphi.mdb database (LoginPrompt = False). Set ADOTable1.Name = ApplicationTable, ADOTable2.Name = AuthorsTable. Set DataSource1.Name = ApplicationsSource. Set the Connection property of all ADOTable components to point to ADOConnection component. Set ApplicationSource.DataSet = ApplicationsTable. Finally, set ApplicationTable.Table = Applications, AuthorsTable.Table = Authors. Finally, add a DBGrid (DBGrid1) to a form and Connect it with ApplicationsSource.

Here is a list of relevant values, as can be seen in the dfm file for a form.

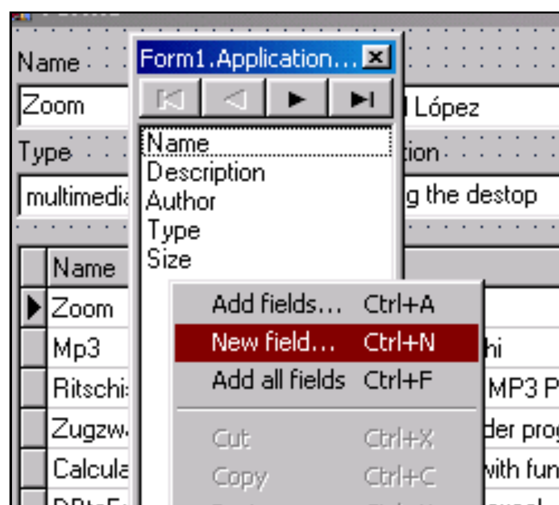
```
object ApplicationsTable: TADOTable
    Connection = ADOConnection
    TableName = 'Applications'
  end
  object AuthorsTable: TADOTable
    Connection = ADOConnection
    TableName = 'Authors'
  end
  object ApplicationsSource: TDataSource
    DataSet = ApplicationsTable
  end
  object DBGrid1: TDBGrid
    DataSource = ApplicationsSource
  end
```

Double click the Applications table (Fields editor) and create a persistent set of field objects – pick Name, Description, Type, Size and Author.

All set up (at least for now). It's time to see the first approach to lookup fields.

New ... lookup field

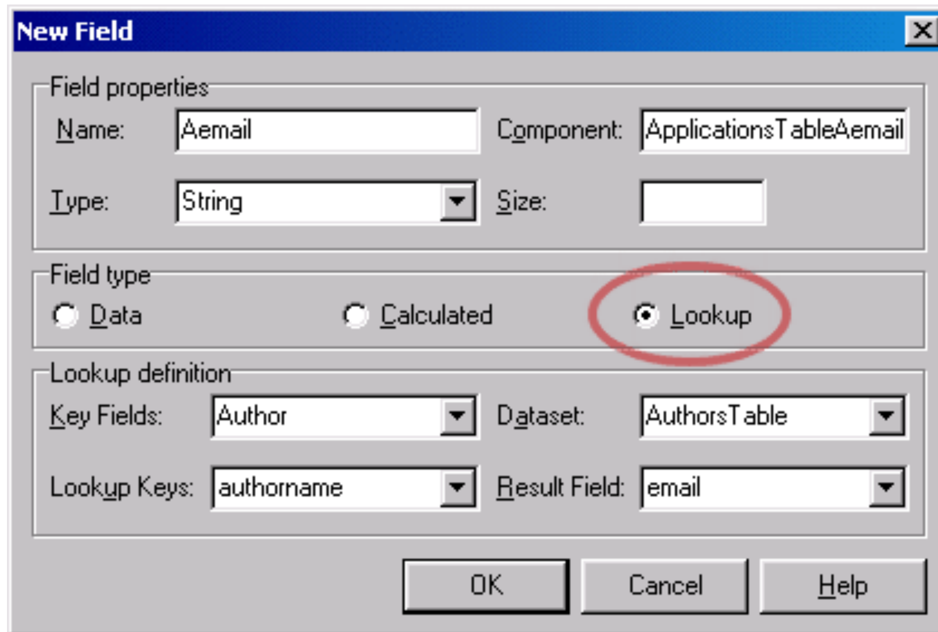
To crete a new field object you have to invoke the Fields editor for a dataset. Double click the ApplicationsTable and right click it to have a pop up menu displayed. Pick "New field..."



The idea is to set the value for the Author field by picking it's email. I know this does not make much sense, but it will bring the lookup idea closer. In the New field form fill the boxes as on the picture

below. We create a new AEmail field for the ApplicationTable dataset. As you can see from the picture, there are few very important properties you have to understand prior to working with lookup fields.

.*Field Name* is the name of the lookup field. Set it to AEmail.
.*Dataset* identifies the dataset that contains lookup values (Authors).
.*KeyFields* is a field in the Applications dataset we are setting through a lookup.
.*Lookup Keys* is a field in the lookup dataset (Authors) to match against the KeyFields.
.*ResultField* is a field in the lookup dataset to return as the value of the lookup field you are creating.



That's it. No code required. Start your project (or set Active property for both tables to True) and you can see that DBGrid, in the AName fields column, has a combo box with author emails. As you pick an email the corresponding author field changes. We are looking up an authors email to change the author. Note that AEMail is a read–only field.

Author	Aemail
Manuel López	morsa@virtualia.com.
Delphi Guide	delphi.guide@about.c
Delphi Guide	delphi.guide@about.c
Manuel López	oracle2025@gmx.de
Cevahir Parlak	morsa@virtualia.com.r
Cihan Sokmen	cevahirparlak@hotmai
	csokmen@yaysat.com.t

Note: the combo box in the cell of a DBGrid has nothing with the PickList property of Grid's column. This will be explained later in this chapter.

Lookup with DBLookupComboBox

As stated above, when your data entry form is made of more data controls (DBEdit, DBComboBox, etc.) it makes sense to just use DBLookupComboBox without creating a new field.

For the start, use dragging from the Fields editor to add data controls to a form. Drag Name, Author, Type and Description. This will add 4 DBEdit components and 4 Label components. At this point, remove the DBEdit connected with the Author field of the Applications table and replace it with a DBLookupComboBox. Name it ApplicationsAuthorLookup. We'll also need another DataSource component. Drop one on the form, change the name to AuthorsSource and link it to AuthorsTable. Finally, we need to set the lookup combo box to work properly. Use Object Inspector and set the

following:

```
object ApplicationsAuthorLookup: TDBLookupComboBox
  DataSource = ApplicationsSource
  DataField = 'Author'
  ListSource = AuthorsSource
  KeyField = 'authorname'
  ListField = 'authorname;email'
end
```

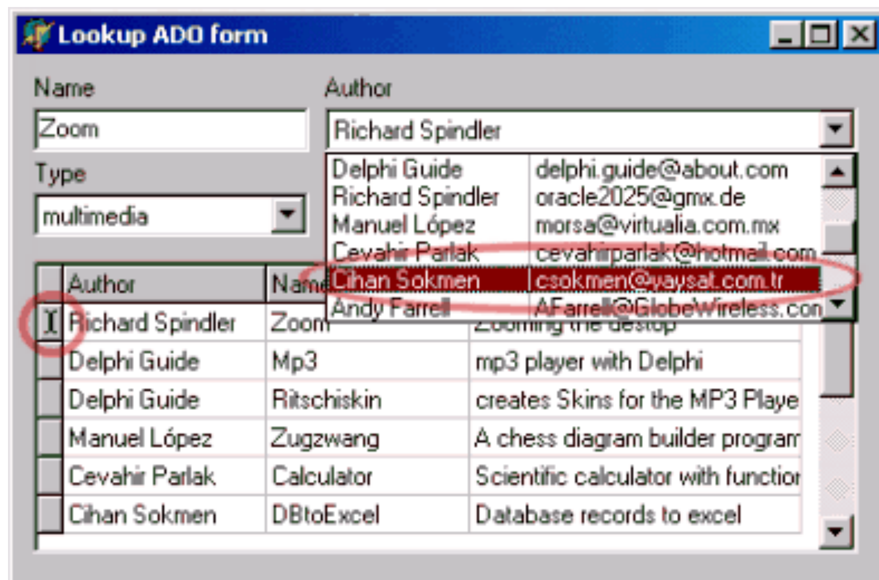
These properties are key to the lookup connection:

. *DataSource* and *DataField* determine the main connection. The DatField is a field into which we insert the looked–up values.
. *ListSource* is the source of the lookup dataset.
. *KeyField* identifies the field in the ListSource that must match the value of the DataField field.
. *ListFields* is the field (one or more) of the lookup dataset that are actually displayed in the combo. ListField can show more than one field. Multiple field names should be separated by semicolons. You have to set large enough value for the DropDownWidth (of a ComboBox) to really see multiple columns of data.

Just for practice, replace the Type's field DBEdit with DBLookComboBox and set a lookup relation with the Types table. Note that you need one more DataSource and one more ADO Table pointing to the Types table.
Set the DataSource and DataField properties to the dataset and field where the selection will be written (Applications; Type). Set the ListSource, KeyField and ListField properties to the lookup field from which the list should be populated (Types; TypeName, TypeName).

Graphically, connections between tables, sources and ADOConnections can now be presented as:

That's it. Again, no code required. When the user selects an item from the combo box, an appropriate value of the KeyField field changes the value of the DataField. At run time, the form looks like:



Note 1: When you want to display more than one field in a LookupComboBox, as in the example above, you have to make sure that all columns are visible. This is done by setting the DropDownWidth property. However, you'll see that initially you have to set this to a very large value which results in dropped list being too wide in most cases. One workaround is to set the DisplayWidth of a particular Field shown in a drop down list. The next code snippet, placed inside the OnCreate event for the form, ensures that both author name and it's email are displayed inside the drop down list.

```
AuthorsTable.FieldByName('authorname').DisplayWidth:=15;
AuthorsTable.FieldByName('email').DisplayWidth:=20;
```

Note 2 : If you drag AEmail field from a Field Editor to a form, Delphi will connect it with a DBLookupComboBox automatically. Of course, key lookup properties will look different since we don't need another data source for this link to work – lookup field is already defined "inside" the ApplicationsTable/Source.

Lookup inside a PickList of a DBGrid Column

The last approach to having a lookup values displayed inside a DBGrid is to use the PickList property of a DBGrid Column object. You'll usually add Columns to a DBGird when you want to define how a column appears and how the data in the column is displayed. A customized grid enables you to configure multiple columns to present different views of the same dataset (different column orders, different field choices, and different column colors and fonts, for example).

I will not discuss this topic briefly here. Let's just see what are the steps to add columns to a DBGrid.
1. Right–click the DBGgrid component. This pops up the Columns Editor.
2. Right–click the Columns Editor to invoke the context menu and choose Add All Fields. This creates one column for each field in a dataset and connects them – the FieldName property determines the connection.

While you have the Columns Editor displayed, see that each column has a PickList property. We'll use this String List to fill a list of lookup values. Take a look at the AEMail column, its FieldName points to a lookup field – therefore displaying a combo box inside a DBGrid as we saw at the beginning of this article. However its PickList is empty. What I want to show you here is how to fill that String List with values from another dataset at run time – without creating a new lookup field. In general, a pick list column looks and works like a lookup list, except that the drop–down list is populated with the list of values in the column's PickList property instead of from a lookup table. The reason to show how to use PickList to mimic the lookup list is just to show that the same task can be done in several ways when you have a great toll like Delphi.

What we want to do is to fill a PickList with the values from another dataset at run time – this time using Pascal code. We'll use the Form's OnCreate event.

```
procedure TForm1.FormCreate(Sender: TObject);
var AuthorsList:TStringList;
    i:integer;
begin
  AuthorsList:=TStringList.Create;
  try
    AuthorsTable.First;
    AuthorsTable.DisableControls;
    while not AuthorsTable.EOF do begin
      AuthorsList.Add(
             AuthorsTable.FieldByName(
    'authorname').AsString);
      AuthorsTable.Next;
    end; //while
  finally
    //place the list it the correct column
    for i:=0 to DBGrid1.Columns.Count-1 do begin
      if DBGrid1.Columns[i].FieldName = 'Author'
      then begin
        DBGrid1.Columns[i].PickList:=AuthorsList;
        Break;
      end;//if
    end; //for
    AuthorsTable.EnableControls;
    AuthorsList.Free;
  end; //try
end;
```

The code simply fills the AuthorsList with the values of AuthorName, by iterating through the Authors table. It then assigns the contents of the AuthorsList to the PickList whose FiledName point to the Author field. Since we can change the order of columns at design and run time you have to make sure that the correct PickList get's the values.

That's all for this chapter. Make sure you download code for this project.

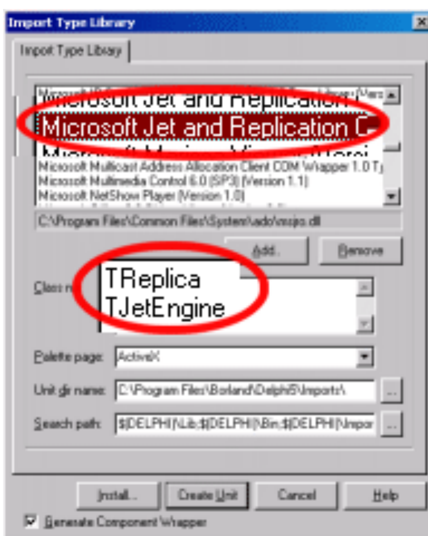
Compacting an Access database with ADO and Delphi – DB Course/Chapter 16

Chapter sixteen of the free Delphi Database Course for beginners. While working in a database application you change data in a database, the database becomes fragmented and uses more disk space than is necessary. Periodically, you can compact your database to defragment the database file. This article shows how to use JRO from Delphi in order to compact an Access database from code.

Why compacting

While you add and delete records from database tables, your database becomes more and more fragmented and uses disk space inefficiently. Compacting a database makes a copy of the database, rearranging how the database file is stored on disk. The compacted database is usually smaller and often runs faster.

This chapter of the free database course for Delphi beginners shows how to use JRO from Delphi in order to compact an Access database from code.



JRO TLB

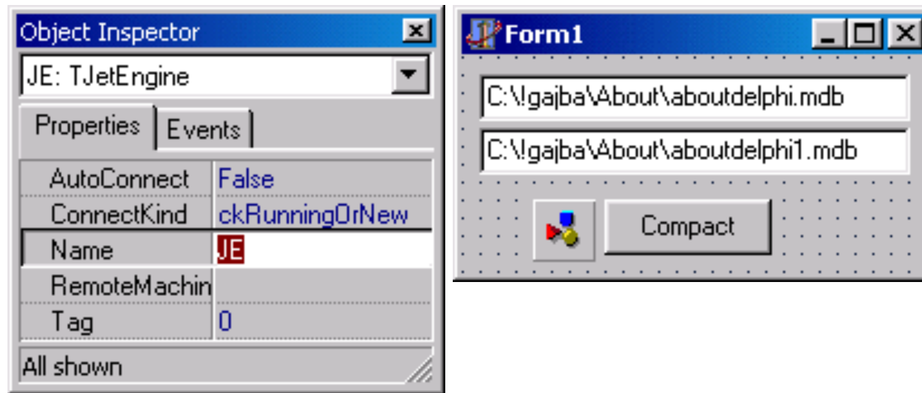
ADO does not directly expose a method for compacting a database. By using Jet and Replication Objects (JRO), you can compact databases, refresh data from the cache, and create and maintain replicated databases. The JRO exposes two objects, the JetEngine object and the Replica object. The Replica object is used to manipulate replicated databases. We will not deal with database replications in this chapter. By using the Jet Engine object we can programmatically control compacting and refreshing data from the memory cache.

As with ADOX, the JRO library must be imported in Delphi, since it is not a part of the ADOExpress (or dbGo in D6). The description of the ADOX library is "Microsoft Jet and Replication Objects 2.x Library (Version 2.x)". The JRO library file name is MSJRO.dll. We've already seen the steps needed to import a type library in Delphi (ADOX). The same process should be repeated in this case. To import JRO in Delphi you need to open a new project and Select *Project | Import Type Library*. In the dialog box choose "Microsoft Jet and Replication Objects 2.x Library (Version 2.x)". Note that it will add two new classes, the TReplica and TJetEngine. Press *Install* button to add JRO to a package or press *Create unit* to just create a single interface unit. If you click *Install*, two new icons will appear on the *ActiveX* tab (if you have left the default Palette page on the Dialog).

Note: Delphi 6 users will not succeed in importing JRO type library. If you have Delphi 6, while trying to install the library in a package, an error will pop up indicating that ActiveConnection in the JRO_TLB file doesn't exist (along with some other errors). The problem lies in Delphi 6 TLB importer. There are two options to overcome the problem: 1. Use Delphi 5 to import JRO an then install it in Delphi 6. 2. Manually declare the missing ActiveConnection property and change property declarations to make them writeable.

Compact Delphi Project

It's time to see some code. Create a new Delphi application with one form. Add two Edit controls and a Button. From the ActiveX component page pick JetEngine. The first Edit should be renamed to edSource, the second one to edDest. The button should be renamed to btnComapct. The JetEngine should be renamed to JE. It should all look like:



The TJetEngine class has a CompactDatabase method. The method takes two parameters: the ADO connection string for the source as well for the destination database. CompactDatabase method compacts a database and gives you the option of changing its version, password, collating order and encryption.
Encrypting a database makes it indecipherable by a utility program or word processor. Encrypted databases can still be opened by Access or through Delphi code. The proper way to protect a database is to set a password for it. Collation order is used for string comparison in the database. Changing a database version gives you the way to "upgrade" it.

In our form, the edSource is used to specify the database we want to compact. The edDest specifies the destination database. Within the connection strings, you specify various connection properties to determine how the source database is opened and how the destination database is compacted. At a minimum, you must use the Data Source property in each connection string to specify the path and name of the database.
When you use the CompactDatabase method, you can't save the compacted database to the same name as the original database. CompactDatabase also requires that the destination database does not exist.

The next code (btnCompact OnClick event handler) is an example of the CompactDatabase method:

```
procedure TForm1.btnCompactClick(Sender: TObject);
var
  dbSrc  : WideString;
  dbDest : WideString;
const
  SProvider = 'Provider=Microsoft.Jet.OLEDB.4.0;
              Data Source=';
begin
  dbSrc  := SProvider + edSource.Text;
  dbDest := SProvider + edDest.Text;

  if FileExists(edDest.Text) then
     DeleteFile(edDest.Text);

  JE.CompactDatabase(dbSrc,dbDest);
end;
```

Note that the above code presumes an Access 2000 database. Microsoft Jet OLEDB 4.0 is the default data engine for Access 2000.

In many cases you'll want to have the same database name after the compact operation. Since edSource and edDest can't be the same your code should replace the original file with the compacted version. The next function takes only one parameter – the name of the database you want to compact:

```
function DatabaseCompact
  (const sdbName: WideString) : boolean;
var
```

```
 JE          : TJetEngine; //Jet Engine
 sdbTemp     : WideString; //TEMP database
 sdbTempConn : WideString; //Connection string
const
 SProvider = 'Provider=Microsoft.Jet.OLEDB.4.0;
             Data Source=';
begin
 Result:=False;
 sdbTemp := ExtractFileDir(sdbName) +
            'TEMP' +
            ExtractFileName(sdbName);
 sdbTempConn := SProvider + sdbtemp;
 if FileExists(sdbTemp) then
    DeleteFile(sdbTemp);
 JE:= TJetEngine.Create(Application);
 try
  try
    JE.CompactDatabase(SProvider + sdbName, sdbTempConn);
    DeleteFile(sdbName);
    RenameFile(sdbTemp, sdbName);
  except
    on E:Exception do
           ShowMessage(E.Message);
  end;
 finally
   JE.FreeOnRelease;
   Result:=True;
 end;
end;
```

The DatabaseCompact receives a sdbName string parameter with the full name of the database you want to compact. The function returns True if compact is successful False otherwise. The sdbName is compacted in sdbTemp, the sdbName is then deleted and sdbTemp renamed to sdbName. The DatabaseCompact could be called as:

```
  DatabaseCompact('C:\ADP\aboutdelphi.mdb');
```

The DatabaseCompact function is ideal to be called from within your Delphi ADO application as an external application. It could also be written as a console mode application that takes one command line parameter (or more) since it requires no GUI.

Database reports with Delphi and ADO – DB Course/Chapter 17

Chapter seventeen of the free Delphi Database Course for beginners. How to use QuickReport set of components to create database reports with Delphi. See how to produce database output with text, images, charts and memos – quickly and easily.

In programming (database) terminology *reports* are printed documents containing data from a database. To generate reports with Delphi we generally use special reporting tools. In most cases such tools are available from third–party developers. One set of components, QuickReport from QuSoft, comes with Delphi as a set of VCL components.

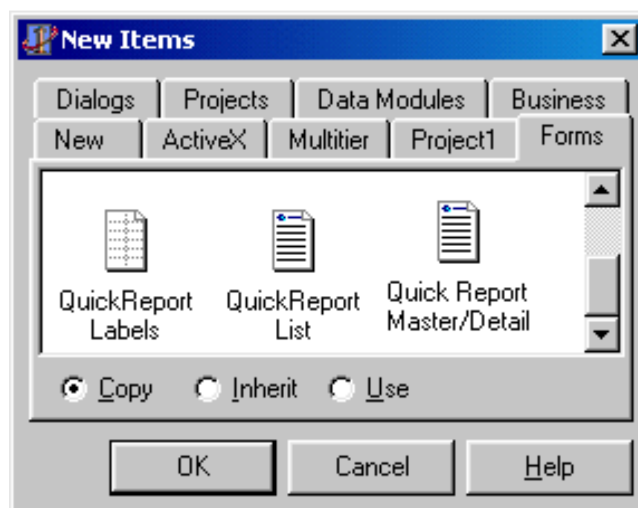
In this chapter of the free database course, you'll see how to use QuickReport set of components to create database reports with Delphi; how to produce database output with text, images, charts and memos – quickly and easily.

QuickReport is designed to let us produce database reports as well as reports based on text files and string lists. QuickReport lets us create print previews where the user can check the result of a printout without wasting paper, and export data to other file formats, such as plain ASCII, comma separated values (CSV), MS Excel XLS, Rich Text RTF and HTML. By using the components on the *QReport page of the Component palette*, you can visually build banded reports to present and summarize the information in your database tables and queries. You can add summaries to group headers or footers to analyze the data based on grouping criteria.

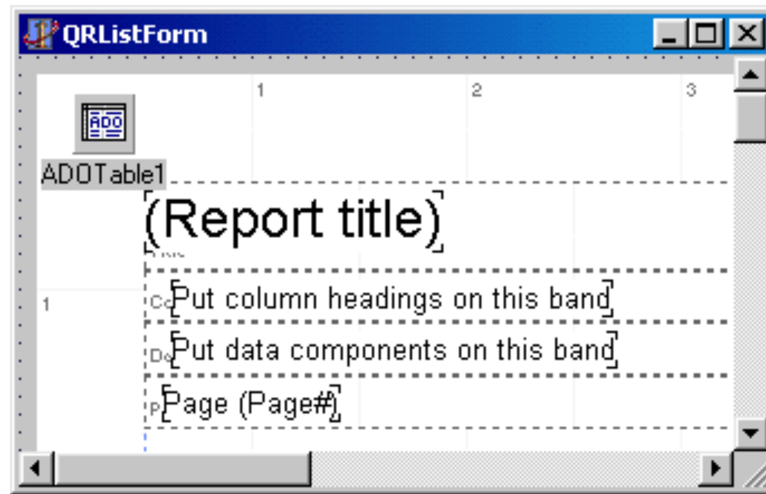
Report Types

Among many different types of database reports, three are most common:

- *List reports* look like a table (or a DBGrid), each row containg data from a row in a recordset.
- *Label reports* present data inside rectangular areas. This type of report is mostly used when printing envelopes.
- *Master–detail* reports are used when report contains data from two linked recordsets. In general, several detail records follow the appropriate master record. This type of report requires parent–child relationship between two recordsets.



The simplest way to create a report is to use the QuickReport Wizard located on the Forms page (Delphi 5) of the New Items dialog. We'll pick the QuickReport List – this creates a new form with the main reporting component TQuickRep. This component acts as a container for several TQrBand components containing TQrLabels. There is also a TTable component on that form.

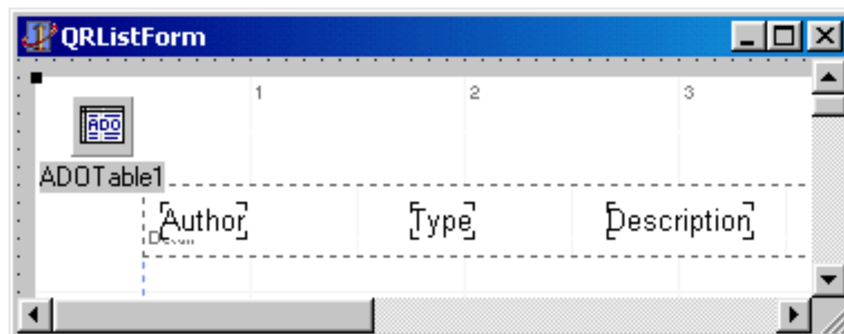


We are not going into details about QuickReport here, since there is a **comprehensive tutorial on using the QuickReport** set of components with Delphi available on this site.

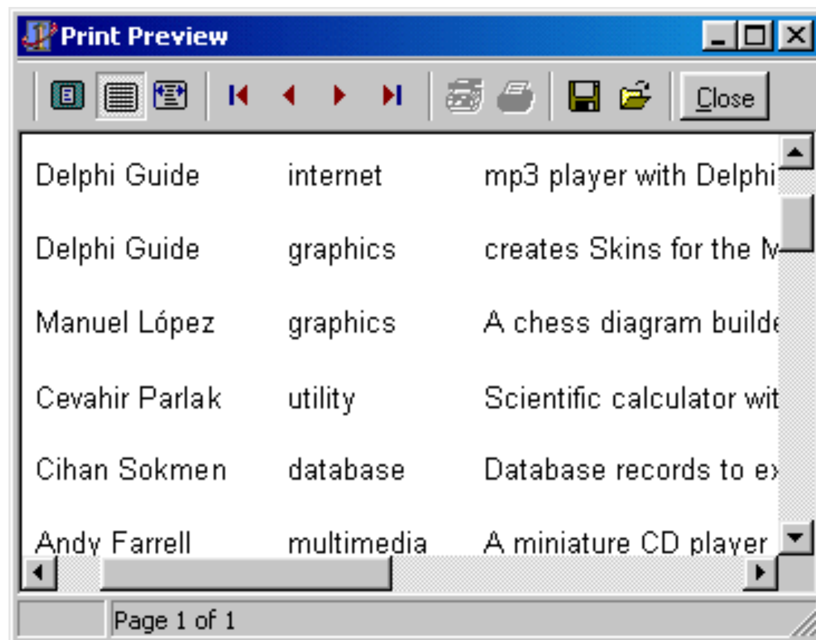
Quick ADO Delphi report

Unfortunately the Wizard crates a reporting template for BDE based database application, by placing the TTable component on a form along with DB and DBTables units in the uses clause. To prepare the form for our aboutdelphi.mdb MS Access database and ADO you need to delete the TTable from the form as well as the DB and DBTables units from the forms uses clause.

We'll now create a simple list report containing data from the Applications table. First make sure the newly created form is the default for the project (Project | Options | Forms). Second add the TADOTable component on a form, set it's Connection property to point to the aboutdelphi.mdb database (this time were not going to use ADOConnection) and set the Table property to Application. Third, remove the TitleBand1 (TQrBand), PageFooterBand1 (TQrBand) and the ColumnHeaderBand1 (TQrBand). Also, remove the QRLabel2 from the DetailBand1. This leaves us with only ADOTable1, QuickRep1 and DetailBand1. To link QuickReport1 with ADOTable1 set it's Dataset property to point to ADOTable1.



The next step is to place several TQRDBText components on DetailBand1, one for each field we want to print, and set their Dataset property to ADOTable1. Add three QRDBText components, let them point to Author, Type and Description fields of the Applications table.
To see the report at design time set ADOTable1.Active to True, right click the QuickRep1 component and select Preview. This is something similar to what you should see:



To show this preview window at run time, we need to call the Preview method of the TQuickRep component.

```
procedure TQRListForm.FormCreate(Sender: TObject);
begin
 QRListForm.QuickRep1.Preview;
end;
```

Note that when previewing the report, Delphi (QuickReport) uses the standard QuickReport preview form. If you want to change the appearance or behaviour of this form you can create your own preview form with the TQRPreview component.

To send this report to a printer use the *Print* method.

Charts and Images

In many cases you'll need to create reports that consist of other elements like charts or images. Creating a report with chart is simple. Just pick the TQRChart component and use it as explained it the Charting with Databases chapter. To show a picture stored inside an Access database you should use the TQRImage (not TQRDBImage) component. Since there are some "problems" with displaying pictures inside Access you should consider the Pictures inside a database chapter.

To the next chapter

That's it. I just wanted to inform you about printing options with ADO Delphi based solutions. You should be aware that beside QuickReport components there are other reporting tools available to a Delphi developer. You can even use Automation to control MS Word and use it as a reporting tool. If you need any kind of help so far, please post to the Delphi Programming Forum where *all the questions are answered and beginners are treated as experts.*

Data Modules – DB Course/Chapter 18

How to use the TDataModule class – central location for collecting and encapsulating data access objects, their properties, events and code.

When developing simple database applications (with one or two forms) it is quite usual to place all the data–access components on a form with data–aware components. But if you plan on reusing groups of database and system objects, or if you want to isolate the parts of your application that handle database connectivity and business rules, then Delphi's Data Modules provide a convenient organizational tool.

This chapter of the free database course for Delphi beginners shows how to use Data Modules in Delphi (ADO) database development to partition an application into user interface, application logic and data.

Data modules provide a formal mechanism for collecting and encapsulating DataSet and DataSource objects, their attributes, events and code (business rules) in one central location. Data modules can contain only nonvisual components and are generally used in database and Web development. They provide Delphi developers a visual way to manipulate and code non–visual components of an application.

Generally, a TDataModule class is used for some of the purposes:

– *Sharing data access components and code*
Use a TDataModule to provide a location for centralized handling of nonvisual components. Typically these are data access components (TADOConnection, TADOTable, TADOQuery, TADOCommand, etc), but they can also be other nonvisual components. This is convenient when an application has multiple forms that share the same data access provider – if your application operates on only one database file (one mdb) then you'll need only one TADOConnection component for all the dataset components.

– *Design time visual organization*
At design time, the Diagram page in the Code editor provides visual tools for setting up logical relationships among the components on the data module. You can drag a property connector from one component to another to hook them up. Drag a master–detail connector between tables to join them. You reach the persistent datasets fields much faster than when using the fields editor.

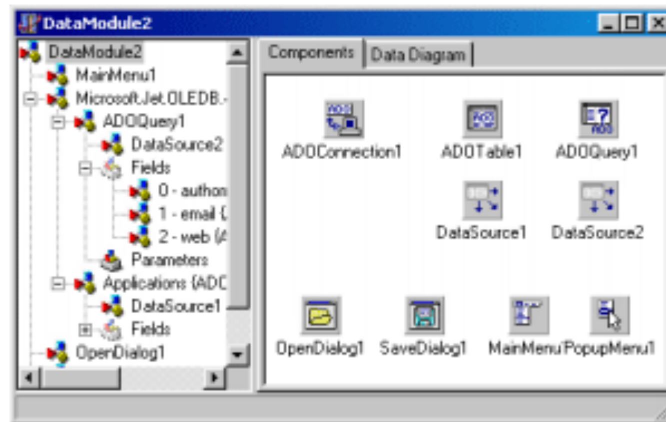
– *Business rules centralization*
In the unit file for the data module a Delphi developer may also place any business rules that are to be applied to the application. These are the controls that prevent invalid data begin entered into a database and ensuring that valid data is maintained within the database.

– *Code maintenance*
Data modules make code maintenance easier since you only have to change code in one place instead of in every form – especially when having common db functions located in a data module or if we create an application with several forms serving the same database tables and/or queries.

New ... Data Module

To create a data module at design time, choose File | New | Data Module. At design time, a data module looks like a standard Delphi form with a white background and no alignment grid. At run time data module exists only in memory. DataModule has only two properties, Name and Tag, and two events, OnCreate and OnDestroy. Use the Name property when referring to module's objects from other units.

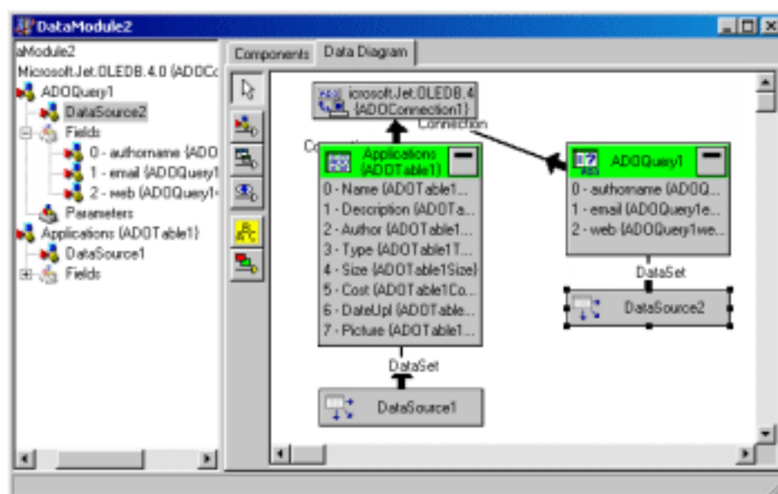
DataModules are not limited to data access components, they can also contain other nonvisual components, such as TMainMenu, TTimer, TSaveDialog or TPopUpMenu).



You place all the components in the right pane ("Components") of data module's window. The tree view on the left is for navigation purposes.

Note: in Delphi versions prior to version 6, the TDataModule class was defined in the Forms unit. Delphi 6 moves TDataModule class to the Classes unit to allow smaller GUI–less applications to be written and to separate OS independent classes. In Delphi 6, the Tree diagram and the Data Diagram are separated. The Diagram page on the Code editor provides visual tools for setting relationships among non–visual (and visual) components.

When you select the Data Diagram page (for the first time), you are presented with an empty container, in which you can drag any object from the Tree View and then link those objects. Relationships include parent/child, datasource/dataset and similar. For example, after moving a TADOTable and TADOConnection to the Diagram view, you select the Property connector icon, click the ADOTable and drag to the ADOConnection. The connection will be made: TADOTable.Connection property will point to ADOConnection component – as can be seen in the Object Inspector.



To make the data module available to another unit in the application, select that unit, then choose File|Use Unit to add the data module to the uses clause for the unit. If you have several forms that refer to the same data module, make sure the data module is created before it is referenced from those form – or you'll get an access violation error.
When refering to a dataset like ADOTable1 (on a data module, named uDM) from some data browsing form (let's say form1) your code will look like:

```
// some procedure in form1
  uDM.ADOTable1.Open;
```

Handling database errors – DB Course/Chapter 19

Introducing error handling techniques in Delphi ADO database application development. Find out about global exception handling and dataset specific error events. See how to write an error logging procedure.

One of the most important elements of database programming is handling database errors. Any robust (database) application must contain error trapping mechanism to prevent an inexperienced (or careless) user to, for example, input a bad value into a data field or delete a record that should not be deleted. Another error might pop up while trying to connect to a database that does not exist.

Our goal in this chapter of the free database course is to introduce error handling techniques in Delphi ADO database application. You can learn about errors and exceptions (general, not database specific) in the article entitled Errors and Exceptions.

There are several approaches we can use to handle database related errors. One approach is to let Delphi show the exception message, but it is much more user friendly to try to correct the error and show some more details on the error:

- **Global exception handling** allows you to create a single procedure to handle any exceptions that may occur in a program. You write a code for the OnException event of the global Application object by using the ApplicationEvents component. This code is then used to handle not just database related errors but any error that may appear while the application is running.
- Handling **dataset's specific events** is an another approach. Events like OnPostError, OnEditError, OnInsertError and similar occur when an attempt to post a record, edit or insert fails. All those events are dataset specific.
- Surrounding risky database operations with **try–except** and **try–finally blocks**, such as a call to the Execute statement of the ADOQuery object is another way to trap a possible exception.

EDatabaseError, EADOError, OLE exceptions

When developing Delphi ADO based database solutions you should be aware of the several types of exceptions. In general, information that describes the conditions of a database error can be obtained for use by an application through the use of an EDatabaseError exception. The more ADO specific exception object is the ADOError object raised when an application detects errors generated by ADO datasets.
Finally, since ADO is based on OLE, OLE exceptions (EOLEError, EOLEException and EOLESysError) might occur, in a Delphi application, during an attempt to invoke a method or property of an OLE automation object – ADO Express (dbGO in D6) components that implement ADO in Delphi are built as wrappers around the ADO objects. When we use ADO extensions like ADOX or JRO libraries OLE exceptions may occur, too.

Error information and management

All the types of exceptions that may occur in a Delphi application are deriveed from the Exception class, which provides a set of properties and methods that can be used to handle error conditions in a graceful manner.

In situations where an ADO dataset is used without any data aware components (reporting, looping through a recordset, etc) we can use exception handling blocks:

```
try
   ADOTable.Open;
 except
   on E:Exception do
   begin
     MessageDlg('Error Opening Table ' + E.ClassName,
```

```
                mtError, [mbOK], 0);
      LogError(E);
    end;
  end;
```

In the except block, you use a series of on/do statements to check for different exceptions. When using the Exception for E your code will handle all exceptions that occur. In the MessageDlg we display the error message and the exception class (E.ClassName). For example, if you try to open a table that is exclusively opened by some other user (Table design in MS Access) you'll get an EOLEError.

When handling dataset's specific events (for example OnPostError for ADOTable dataset) we write code for the appropriate events.

```
procedure TForm1.ADOTablePostError(
  DataSet: TDataSet;
  E: EDatabaseError;
  var Action: TDataAction);
begin
  LogError (E); //custom error loggin procedure
  {
  Show some message to the user
  about the failure of the the post operation..
  }
  Action := daAbort;
end;
```

The Action parameter indicates how an application should respond to a database error condition. If you are able to determine the cause of the error and fix it, you use the daRetry action – Delphi will try to re–execute the Post method. The daAbort action should be specified when you handle the error (show some meaningful message) and there is no need for the exception to "go" the the global exception handler (Application.OnException). The daFial is the default action.

If your application provides the event handler for the OnException event of the ApplicationEvents component, all the exceptions can be handled in one place:

```
procedure TForm1.ApplicationEventsException
        (Sender: TObjException);
begin
  LogError (E); //custom error logging procedure
end;
```

The message property (of the Exception object) can be used to show a message on the screen, save the message in some log file, or combine some specific message information with our own custom message.

Error logging procedure

In most situations it is a good idea to have some error logging procedure that writes every error to a text file. The LogError procedure below uses the TStringList object to store information of the current error in a file named error.log (in the application folder).

```
procedure LogError(E:Exception);
var sFileName  : string;
    errLogList : TStringList;
begin
 sFileName := ExtractFilePath(Application.EXEName) + 'error.log';

 errLogList := TStringList.Create;
 try
  if FileExists(sFileName) then
     errLogList.LoadFromFile(sFileName);
```

```
  with errLogList do
  begin
   Add('Error Time Stamp: ' +
      FormatDateTime('hh:nn am/pm', Now) +
      ' on ' +
      FormatDateTime('mm/dd/yy', Now));
   Add('Error Class: ' + E.ClassName);
   Add('Error Message: ' + E.Message);
   SaveToFile(sFileName);
  end; //with
 finally
  errLogList.Free;
 end;
end;
```

When you open the error.log file with some text editor, you'll get something like:

...
Error Time Stamp: 02:49 pm on 10.30.01
Error Class: EOleException
Error Message: Table 'djelatnici' is exclusively locked by user...
...

ADO Errors collection

Any operation involving ADOExpress components can generate one or more errors. As each error occurs, one or more Error objects are placed in the Errors collection of the ADOConnection component. You must note that error objects represent individual errors from the provider and are not ADO–specific, this means that the "same" error will be reported differently by MS Access and differently by MS SQL Server. When an error occurs, the provider is responsible for passing an error text to ADO. In Delphi, using the Errors Collection object directly is not recommended unless you are familiar with connection object operations.
The Error property of TADOConnection component represents the Errors object and has several properties. The Description property contains the text of the error. The Number property contains the long value of the error constant. The Count property indicates the number of Error objects currently stored in the collection. To get the Description of the last error in the Errors object you can use the next statement:
ADOConnection1.Errors.Item[ADOConnection1.Errors.Count–1].Description;

From ADO Query to HTML – DB Course/Chapter 20

How to export your data to HTML using Delphi and ADO. This is the first step in publishing your database on the Internet – see how to create a static HTML page from an ADO query.

In this chapter, of the free database Delphi ADO course, you are going to see how to easily create HTML pages based on database information. In particular, you'll see how to open a query from an MS Access database with Delphi and loop through the contents generating an HTML page for each row in a recordset.

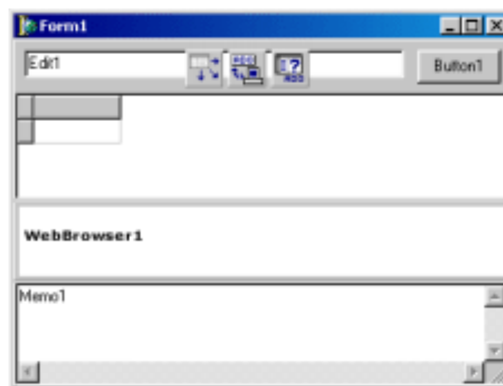
Start a new application – this creates an empty Delphi form.
Add the following data components :

- TAdoConnection (name: AdoConnection1)
- TAdoQuery (name: AdoQuery1)
- TDataSource (name: DataSource1)
- TDBGrid (name: DBGrid1)

To display the generated HTML you'll need several more components to be dropped on a form.

- TWebBrowser (name: WebBrowser1) – used to display the generated HTML.
- TMemo (name: Memo1) – used to display the text of the generate HTML.
- TButton (name: Button1) – used to open a query, create the HTML code and display it.

This is how your form should look at design time (in the middle of the form there is a WebBrowser component)



The first thing you need to do when working with a database in Delphi and ADO, is to set relations between data access components then open a database connection. Just to test your knowledge we'll link all DB related components from code (no need to use the Object Inspector). You place the code in the OnCreate event for the form (plus some extra code to pre–set other components):

```
procedure TForm1.FormCreate(Sender: TObject);
var ConStr: widestring;
begin
ConStr := 'Provider=Microsoft.Jet.OLEDB.4.0;'+
          'Data Source=C:\!gajba\About\aboutdelphi.mdb;'+
          'Persist Security Info=False';

DBGrid1.DataSource := DataSource1;
DataSource1.DataSet := ADOQuery1;
ADOQuery1.Connection := ADOConnection1;
ADOConnection1.ConnectionString := ConStr;
ADOConnection1.LoginPrompt:=False;
```

```
Edit1.Text:='SELECT * FROM [tablename]';
Memo1.Text:='';
end;
```

The next step is to get the recordset by opening a query so we can get access to the data. The query text is in the Edit1 components Text property. The code should be placed in the OnClick event of a Button1 (as described below).

```
  ADOQuery1.SQL.Text:=Edit1.Text;
  ADOQuery1.Open;
```

HTML code

HTML is the most widely spread format for content on the Web. If your knowledge on HTML is low, please consider visiting About's HTML site. Here's a really quick info on html: in general, a HTML page consists of text, plain ASCII text. An HTML file can contain many, so called, tags which determine the style of the font, formating of the paragraph, ... You use special tags to create a table with rows and columns.

After we've opened up a recordset, we loop through the records creating a table for each entry. Prior to generating a HTML table, we create the HTML header, title and body. We first loop through the field names to create a table header, then loop through the recordset to create and fill table rows.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  shtml      : widestring;
  htmlfile   : TextFile;
  i          : integer;
  AvailableFields: set of  TFieldType;
begin
AvailableFields:=[ftWideString, ftDate,  ftFloat];
//open query
ADOQuery1.SQL.Text:=Edit1.Text;
ADOQuery1.Open;

// --> create a html page


//html header shtml:= '<html> <head> <title>'; shtml:= shtml + Edit1.Text; shtml:= shtml +
'</title></head>' + #13#10; shtml:= shtml + '<body>' + #13#10; shtml:= shtml + 'Table created from
query: <i>' + Edit1.Text + '</i>' + #13#10; //table header shtml:= shtml + '<table border="1"
width="100%">' + #13#10; shtml:= shtml + '<tr>' + #13#10; for i:=0 to AdoQuery1.FieldCount–1 do
begin if ADOQuery1.Fields[i].DataType in AvailableFields then begin shtml:= shtml + '<td>'; shtml:=
shtml + '<b>' + ADOQuery1.Fields[i].DisplayName + '</b>'; shtml:= shtml + '</td>' + #13#10; end;
end;{for} shtml:= shtml + '</tr>' + #13#10; //table body while not adoquery1.Eof do begin shtml:=
shtml + '<tr>' + #13#10; for i:=0 to AdoQuery1.FieldCount–1 do begin if
ADOQuery1.Fields[i].DataType in AvailableFields then begin shtml:= shtml + '<td>'; shtml:= shtml +
ADOQuery1.Fields[i].AsString; shtml:= shtml + '</td>' + #13#10; end; end;{for} shtml:= shtml + '</tr>'
+ #13#10; ADOQuery1.Next; end;{while} shtml:= shtml + '</table>' + #13#10; shtml:= shtml +
'</body></html>';
```

Note: for the sake of simplicity, we'll allow only string, number and date fields to be converted and inserted in a html table. The AvailableFields variable is a set of TFieldTypes declared as: `AvailableFields: set of TFieldType`; in the `VAR` section and assigned like: `AvailableFields:=[ftWideString, ftDate, ftFloat];`

Once you have the *shtml* string variable filled with the HTML, we simply show it in the Memo, save the html string to the disk (in the text file that has the applications name, extension is set to htm), and

finally navigate to the saved file in a WebBrowser.

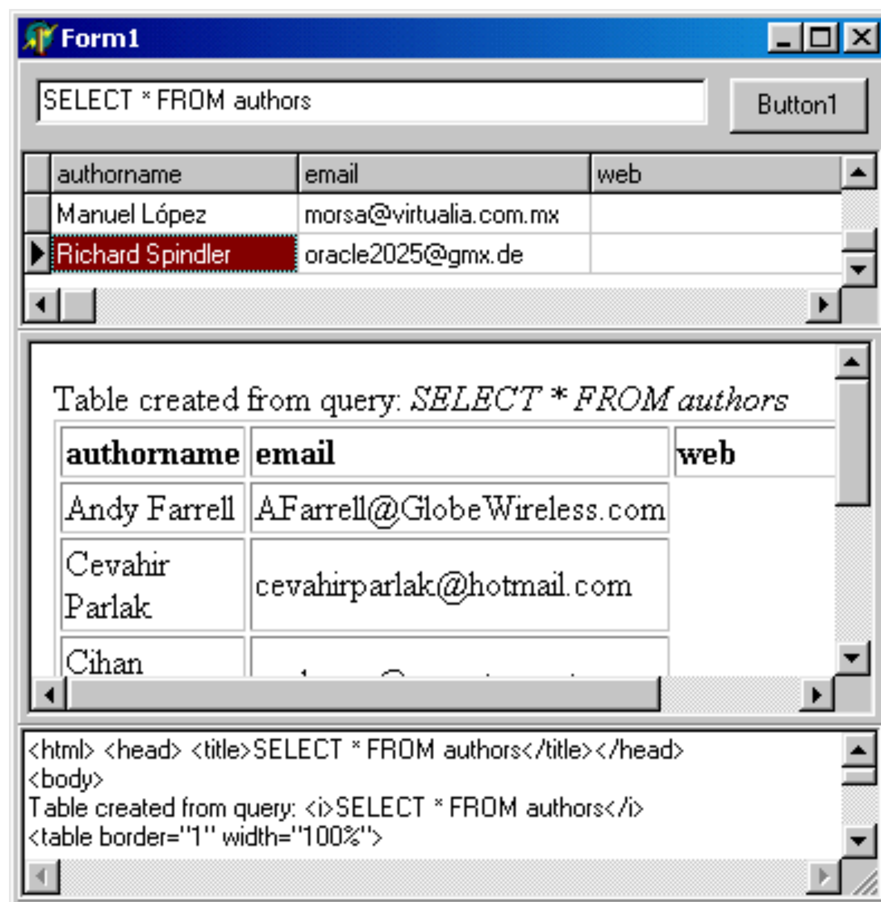
```
// --> assign to memo
  Memo1.Text := shtml;

// --> save in a htm file
 AssignFile(htmlfile,
            ChangeFileExt(
            Application.ExeName,'.htm'));
 Rewrite(htmlfile);
 WriteLn(htmlfile, shtml);
 CloseFile(htmlfile);


// --> browse to the file
 WebBrowser1.Navigate(ChangeFileExt(
     Application.ExeName,'.htm'));

end;{Button1.OnClick}
```

Now start the project, write some valid query string in the Edit component, like SELECT * FROM authors, click the Button1, and this is what you should get:



It is clear that you can easily configure this code to determine which fields to include and which not, how to sort the database, which filters to apply (you can use WHERE in your SQL statement), and set the HTML file look (Colors, Fonts, etc). Download the projects code and start exporting your data to HTML.

The TCustomContentProducer components

Another way of converting database query results to html pages is to use some of the descendants of the TCustomContentProducer component. The components (PageProducer, DataSetTableProducer, QueryTableProducer, ...) are located on the Internet page of the component palette – if your Delphi

version includes them. We'll be looking at some of those components in the next chapter.

Using ADO in Delphi 3 and 4 (before AdoExpress) – DB Course/Chapter 21

How to import Active Data Objects (ADO) type–libraries in Delphi 3 and 4 to create a wrapper around components that encapsulate the functionality of ADO objects, properties and methods.

Article submitted by: Richard Sutcliffe.

Chapter 21 of the free Delphi ADO DB Couse for beginners.

With the advent of Delphi 5 Enterprise, Borland introduced ADO Express, the first time Delphi has ever provided database access without the BDE. The reason behind this whas simple, the BDE had its day. COM development was becoming the norm and the BDE was unfortunately incompatible.

Unfortunately, the ADO Express components aren't without their flaws aither. One of the main (now fixed) bugs was the dreaded 'either BOF or EOF is true or the record has been deleted error. On top of this, the components were uncomfortably shoe–horned into BDE–style component interfaces.

Which leaves us with a quandry over what to do if we want to use Active Data Objects (ADO) in our Delphi applications; after all there are many benefits – ADO is faster, with native support for some of the more common Microsoft database engines, and you don't have to live with the old BDE distribution problem as all recent versions of Windows have support built right in.

Delphi does provide an answer, however, in its amazing support for type–libraries and all things COM, by allowing you to import the type–library and automatically creating a wrapper.

MSADOxx.TLB – Ado Type Library

To import the ADO type library go to the File menu and select Open... From the Files of type combo select Type Library, and browse to the ADO folder (usually found in \PROGRAM FILES\COMMON FILES\SYSTEM\). The ADO type–libraries are named MSADOxx.TLB. The number on the end will depend on the version of Windows you have installed (ADO is also installed by Office and many other programs). The latest version as of writing is 2.7, which you can download from Microsoft.

Once Delphi has finished processing the MSADOxx.TLB file, it will present you with an editor showing all the types, interfaces, etc it has read from the type–library. Pressing F12 at this point will display the Pascal wrapper unit ADODB_TLB.PAS (if the file–path isn't the BORLAND\DELPHIx\IMPORTS\ folder you may wish to move this so subsequent projects can find it).

You can at this point use the ADODB_TLB unit by adding it into your projects USES clause, however there are some problems to be overcome.

Microsoft's VARIANT type

The first is how to handle Microsoft's VARIANT type. Delphi's equivalent is OLEVARIANT, though most programmers will be used to the native STRING, INTEGER and the like. Again this is something Delphi handles with ease with an array of functions such as VarCast. Below is an example of a function that allows us to cast variants into something easier to swallow:

```
function oleGetStr(value: oleVariant): string;
var
  index,
  lowVal,
  highVal : integer;
  oleArray: PSafeArray;
  oleObj : oleVariant;
begin
  result := '';
```

```
  try
    case VarType(value) of
    varEmpty,
    varNull: result := '';
    varSmallint,
    varInteger,
    varByte,
    varError: result := IntToStr(value);
    varSingle,
    varDouble,
    varCurrency: result := FloatToStr(value);
    varDate: result := DateTimeToStr(value);
    varOleStr,
    varStrArg,
    varString: result := value;
    varBoolean:
       if value then
          result := 'True'
       else result := 'False';
    varDispatch, // do not remove IDispatch!
    varVariant,
    varUnknown,
    varTypeMask:
    begin
      VarAsType(value, varOleStr);
      result := value;
    end;
    else
      if VarIsArray(value) then
      begin
        VarArrayLock(value);
        index := VarArrayDimCount(value);
        lowVal := VarArrayLowBound(value, index);
        highVal := VarArrayHighBound(value, index);
        oleArray := TVariantArg(value).pArray;

        for index := lowVal to highVal do
        begin
          SafeArrayGetElement(oleArray, index, oleObj);
          result := result + oleGetStr(oleObj) + #13#10;
        end;

        VarArrayUnlock(value);
        Delete(result, length(result) - 1, 2);
      end
      else
        result := ''; //varAny, varByRef
    end;
  except
    // do nothing, just capture
  end;
end;
```

Once you know how to import type–libraries & handle variant types, Microsoft's MSDN site will become your best friend. There is no way Borland could keep up with the new type–libraries and technologies churned out by Microsoft, however by utilizing MSDN you can find out exactly what each method expects and returns and the wrapper unit will allow you to see exactly how this converts into Object Pascal.

An example of this can be found in the function below:

```
function adoConnect(
  connectionStr: string;
  userName: string = '';
```

```
  password: string = ''
 ): oleVariant;
begin
  result := CreateOleObject('ADODB.Connection');

  if (VarType(result) = varDispatch) then
    if (userName <> '') then
      result.Open(connectionStr, userName, password)
    else
      result.Open(connectionStr);
end;
```

This function allows us to open up an ADO connection by passing in a connection string (a collection of parameters parsed by ADO describing the database & provider), and optionally the required username and password.

The example above uses what is termed late–binding, the disadvantage of this approach being that Delphi cannot interpret what is being called until runtime, however it does allow you better support as the automatic type–library conversion process cannot convert every method available. The alternative, early–binding, utilizes the classes & methods as defined in the Delphi wrapper unit and as such allow you to use features such as code–completion, although is much more restrictive. An example of early–binding is shown below:

```
function adoGetDisconnectedRecordset(
    adoCon: oleVariant;
    adoSQL: string
): oleVariant;
var
  data: RecordSet;
begin
  data := CoRecordset.Create;

  try
    data.CursorLocation := adUseClient;

    data.Open(
      adoSQL,
      adoCon.ConnectionString,
      adOpenKeyset,
      adLockBatchOptimistic,
      adCmdText
    );

    data.Set_ActiveConnection(nil);

    result := data;
  finally
    data := nil;
  end;
end;
```

Now that you know how to interpret type–libraries you can apply your knowledge to other Microsoft and third–party technologies, such as XML and MTS.

Transactions in Delphi ADO database development – DB Course/Chapter 22

How many times have you wanted to insert, delete or update a lot of records collectively wanting that either all of them get executed or if there is an error then none is executed at all? This article will show you how to post or undo a series of changes made to the source data in a single call.

The general idea behind a transaction is that several steps can be performed in series, with the capability to undo all of the steps at once if needed. In addition, the transaction should happen inside an isolated world where other transactions cannot change data while your transaction is running.

For example, to transfer money between two bank accounts, you subtract an amount from one account and add the same amount to the other. If either update fails, the accounts no longer balance. Treating these changes as a single event ensures either all or none of the changes apply.

In this chapter of the free Delphi ADO database course we'll see how to **enable transaction processing in Delphi ADO development**.

BeginTrans, CommitTrans, RollBackTrans

Database transactions are a means to allow a user to do many operations on a recordset or not to do any of them. There is no such thing in a transaction that one task is done and other is not. Transactions are always executed as a *whole*. By using transactions, you ensure that the database is not left in an inconsistent state when a problem occurs completing one of the actions that make up the transaction.

In Delphi ADO's transaction processing, 3 methods are used with the TADOConnection object to save or cancel changes made to the data source.

Once you call the *BeginTrans* method, the provider will no longer instantaneously commit any changes you make until you call *CommitTrans* or *RollbackTrans* to end the transaction.

Transaction Level

The *IsolationLevel* property is the level of transaction isolation for a TADOConnection object. The purpose of the isolation level is to define how other transactions can interact with your transactions, when they work with the same tables. For example, can you see changes in other transactions before or after they are committed? This property only goes into effect after you make a BeginTrans method call.

Transaction processing

To start a transaction call the BeginTrans method of the TADOConnection object. BeginTrans returns the nesting level of the new transaction. A return value of "1" indicates you have opened a top–level transaction (that is, the transaction is not nested within another transaction), "2" indicates that you have opened a second–level transaction (a transaction nested within a top–level transaction), and so forth. Once the BeginTrans is executed, the *OnBeginTransComplete* event is triggered and the *InTransaction* property to True.

Note: Since transactions can be nested, all lower–level transactions must be resolved before you can resolve higher–level transactions.

Once you have started a transaction, a call to commit the transaction is usually attempted in a try...except block. If the transaction cannot commit successfully, you can use the except block to handle the error and retry the operation or to roll back the transaction.

```
var Level: integer;
begin
  Level:=ADOConnection1.BeginTrans;
```

```
  try
    //do some database
    //updating, deleting or inserting
    ADOConnection1.CommitTrans;
  except
    on E:Exception do ADOConnection1.RollbackTrans;
  end;//try
end;
```

As you can see, using transactions in Delphi ADO is rather simple. When you call CommitTrans, the current transaction ends and, if possible, all changes are saved. However, if the database is unable to save any one of the changes, then none of them are saved. In this latter case when a saving problem occurs, the CommitTrans method throws an exception, we catch it in the except part and call the RollbackTrans to cancel any changes made during the current transaction.

Even though it is possible, it is NOT advisable to start, commit or rollback a transaction in different event handlers (button clicks, for example). Windows are event driven, if a user starts a transaction within a button click procedure, you must be sure he will try to commit it. Thus, the longer a transaction is active, the higher is the probability that transaction will conflict with another when you attempt to commit any changes.

Deploying Delphi ADO database applications – DB Course/Chapter 23

It is time to make your Delphi ADO database application available for others to run. Once you have created a Delphi ADO based solution, the final step is to successfully deploy it to the user's computer.

Once you have created a Delphi ADO based solution, the final step is to successfully deploy it to the user's computer. This is the topic will cover in this chapter of the free Delphi ADO database course.

What is required by a given application varies, depending on the type of application. Each project may contain several executable files, and a number of supporting files, such as DLLs, and package files. In most cases you'll only have one executable to install on a target computer – your application's executable file – Delphi produces applications wrapped in compact exe files, so called standalone Windows application.

Applications that access databases involve special installation considerations beyond copying the application's executable file onto the target computer.

When deploying database applications that use ADO, you need to be sure that MDAC version 2.1 or later is installed on the system where you plan to run the application. MDAC is automatically installed with software such as Windows 98 / 2000 and Internet Explorer version 5 or later. No other deployment steps are required.

ADO on Windows 95
If your client's computer has Windows 95 you may get a warning indicating that you need DCOM95 to install MDAC, since there is no support for ADO in Windows 95. Once you install DCOM95, you'll can proceed and install MDAC. Here's a tip how to find the OS version.

MS Data Version Checker
One of the tools that might be handy is the Microsoft's Component Checker. The Component Checker tool is designed to help you determine installed version information and diagnose installation issues with the Microsoft Data Access Components (MDAC).

The database file
The data files (in our case an MS Access database), must be made available to the application. You make it available by simply copying (an "empty") the MDB file to the client computer. The process must be done before the user first start the application. Each time you send an updated exe file, if the structure of the database has not changed, you do not need to redeploy the MDB file.

The connection string
Obviously, when creating database applications that are to be run on various machines, the connection to the data source should not be hard–coded in the executable. In other words, the database file may be located anywhere on the user's computer – the connection string used in the TADOConnection object must be created at run time. One of the suggested places to store the path to the database is the Windows Registry. In general to create a connection string at run time you have to a) place the Full Path to the database in Registry; and b) each time you start your application, read the Registry, "create" the ConnectionString and Open the ADOConnection. Here's a sample code.

Is that all? This seams to easy
Yes, when described by words it seems very easy, just copy the exe file, the mdb file and be sure the client computer has MDAC. Of course, the first time you try to deploy your state–of–the–art database solution you'll bump into a wall. How to copy a file on another computer to a folder like '\Program Files\YOUR COMPANY\PROJECT NAME\ApplicationName.exe'? There is an answer that might solve most of the problems, the installation tools. The Install Sheild Express that comes with some Delphi

versions can be used to create an installation application. This tool gives your users the ability to select where to install your application, let's the user decide what parts of the application would be installed (exe, help, mdb, ...) and makes the process of creating a Project group in the Start menu easy. Creating insulation applications with IE Express is not hard nor is it trivial. I'll leave this for some future article.

TOP ADO programming TIPS – DB Course/Chapter 25

Collection of frequently asked questions, answers, tips and tricks about ADO programming.

This chapter of free Delphi/ADO DB course offers specific recommendations to help improve and speed up the development of your database applications made with Delphi and ADO.

Our intention is to update this chapter dynamically with new tips, and code suggestions. If you have a tip (or a question) on (Delphi) ADO programming feel free to add it to this page.
Note that some of the questions you might ask are most likely already answered through the chapters of this Course.

Access, ADOExpress, dbGo

My clients do not have Access on their machines, will my Delphi / ADO application work?
In general: what ever database you create (Paradox or Access) the users of your software do not need to have MS Access or Paradox on their machines. When using Access with ADO, MS provides MDAC (components to access an Access database).

I have Delphi 5 Professional, where is ADOExpress?
Either you'll need to buy at least Delphi 6 Professional, or try with some third party ADO VCL components

Connecting to data stores

How do I connect to a MS Access 2000 database?
ADOConnection.ConnectionString :=
'Provider=Microsoft.Jet.OLEDB.4.0;DataSource=C:\MyDatabase.mdb;Persist Security Info=False';

How do I connect to a password protected MS Access 2000 database?
ADOConnection.ConnectionString := 'Provider=Microsoft.Jet.OLEDB.4.0;Jet OLEDB:Database Password=XXXXXX;DataSource=C:\MyDatabase.mdb;Persist Security Info=False';

What provider should I use for MS Access
For MS Access 97 use Microsoft.Jet.OLEDB.3.51
For MS Access 2000 use Microsoft.Jet.OLEDB.4.0

How do I connect to a dBase database?
ADOConnection.ConnectionString :=
'Provider=Microsoft.Jet.OLEDB.4.0;DataSource=C:\MyDatabase.mdb;Extended Properties="dBase 5.0;"';

How do I connect to a Paradox database?
ADOConnection.ConnectionString :=
'Provider=Microsoft.Jet.OLEDB.4.0;DataSource=C:\MyDatabase.mdb;Extended Properties="Paradox 7.X;"';

How do I connect to a MS Access database on a CD (read only) drive?
ADOConnection.Mode := cmShareExclusive;

Data retrieving and manipulation

How do I use multiword table / field names (spaces in Table or Field name)?
Enclose multiword names in [ ] brackets:
ADOQuery1.SQL.Text := 'SELECT [Last Name], [First Name] FROM [Address Book]';

How do I use constant fields in an SQL query?
ADOQuery1.SQL.Text := 'SELECT ''2002'', [First Name], Salary FROM Employess';

How do I delete all records in a table?
ADOQuery1.SQL.Text := 'DELETE * FROM TableName';

Why do I keep getting a "−1" for the RecordCount property
If you need the RecordCount to be correct, set the CursorType to something other than ctOpenForwardOnly.

I'm using AutoNumber for Primary Key field to make every record unique. If I want to read or Edit some ADOTable record after one was appended (and Post−ed) I get en error: "The specified row could not be located for updating. Some values may have been changed since it was last read". Why?
After every new record you should use:

```
var bok: TBookmarkStr;
begin
bok:=adotable1.Bookmark;
adotable1.Requery();
adotable1.Bookmark:=bok;
end;
```

How do I create a disconnected ADO recordset? I want to run a query, pick the data and delete some records but not physically.
In order to create a disconnected ADO recordset, you must first set the ADODataSets CursorLocation property to "clUseClient". Then open the Recordset. Then set the ADODatasets Connection to Nil. Do not close the ADODataset.

How do I retrieve a system information, for example, list of tables, fields (columns), indexes from the database?
TADOConnection object has an OpenSchema method that retrieves system information like list of tables, list of columns, list of data types and so on. The following example shows how to fill an ADODataSet (DS) with a list of all indexes on a table (TableName):

```
var DS:TADODataSet;
...
ADOConnection.OpenSchema(siIndexes, VarArrayOf([Unassigned, Unassigned, Unassigned,
Unassigned, TableName]), EmptyParam, DS);
```

How can I improve the performance of my Ado application (like speed up query data retrieval)?
. Avoid returning too many fields. ADO performance suffers as a larger number of fields are returned. For example using "SELECT * FROM TableName" when TableName has 40 fields, and you really need only 2 or 3 fields
. Choose your cursor location, cursor type, and lock type with care. There is no single cursor type you should always use. Your choice of cursor type would depend on the functionality you want like updatability, cursor membership, visibility and scrollability. Opening a keyset cursor may take time for building the key information if you have a lot of rows in the table whereas opening a dynamic cursor is much faster.
. Release your dynamically created ADO objects ASAP.
. Check your SQL expression: when joining tables with Where t1.f1 = t2.f1 and t2.f2 = t2.f2 it is important that f1 and f2 as fields are set to be indexed.