Delphi Database Architecture

CHAPTER

7

IN THIS CHAPTER

- Types of Databases 298
- Database Architecture 299
- Connecting to Database Servers 299
- Working with Datasets 300
- Working with Fields 315

In this chapter, you'll learn the art and science of accessing external database files from your Delphi applications. If you're new to database programming, we do assume a bit of database knowledge, but this chapter will get you started on the road to creating high-quality database applications. If database applications are "old hat" to you, you'll benefit from the chapter's demonstration of Delphi's spin on database programming. Delphi 6 offers several mechanisms for accessing data, which we will cover in this chapter, and then in more detail in chapters to follow. This chapter discusses the architecture upon which all data access mechanisms in Delphi 6 are built.

Types of Databases

The following list is taken from Delphi's online help under "Using Databases." The references mentioned in the list are also found in the online help. We'll refer to this information here because we felt that Borland described the types of database supported by Delphi's architecture best:

- The BDE page of the Component Palette contains components that use the Borland Database Engine (BDE). The BDE defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables. However, it is also the most complicated mechanism to deploy. For more information about using the BDE components, see "Using the Borland Database Engine."
- The ADO page of the Component Palette contains components that use ActiveX Data Objects (ADO) to access database information through OLEDB. ADO is a Microsoft Standard. A broad range of ADO drivers is available for connecting to different database servers. Using ADO-based components lets you integrate your application into an ADO-based environment (for example, making use of ADO-based application servers). For more information about using the ADO components, see "Working with ADO Components."
- The dbExpress page of the Component Palette contains components that use dbExpress to access database information. dbExpress is a lightweight set of drivers that provide the fastest access to database information. In addition, dbExpress components support cross-platform development because they are also available on Linux. However, dbExpress database components also support the narrowest range of data manipulation functions. For more information about using the dbExpress components, see "Using Unidirectional Datasets."
- The InterBase page of the Component Palette contains components that access InterBase databases directly, without going through a separate engine layer. For more information about using the InterBase components, see "Getting Started with InterBase Express."

Database Architecture

Delphi's database architecture is made up of components that represent and properly encapsulate database information. Figure 7.1 represents this relationship as defined by Delphi 6's online help under "Database Architecture."

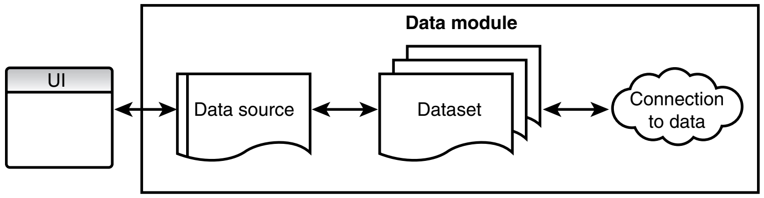


FIGURE 7.1
Delphi database architecture.

Figure 7.1 shows the database architecture in its simplest form. That is, a user interface interacts with data through a data source, which connects to the dataset that encapsulates the data. In the prior section, we discussed different types of databases with which Delphi can work. These different data repositories require different types of datasets. The dataset shown in Figure 7.1 represents an abstract dataset from which others will descend to provide access to different types of data.

Connecting to Database Servers

Okay, so you want to be a database developer. Naturally, the first thing you'll want to do is learn how to make a connection from Delphi to the database of your choice. In this section, you'll learn a number of ways Delphi enables you to make connections to servers.

Overview of Database Connectivity

Datasets must connect to database servers. This is typically done through a connection component. Connection components encapsulate the connectivity to a database server and serve as a single connection point for all datasets in the application.

Connection components are encapsulated in the `TCustomConnection` component. `TCustomConnection` is descended from to create components to encapsulate specific data repository types. Among the different types of data access components are the following for each type of data repository:

- `TDatabase` is the connection component for BDE based datasets. Such datasets are `TTable`, `TQuery`, and `TStoreproc`. BDE database connectivity is covered in Chapter 28 in the CD copy of *Delphi 5 Developer's Guide*.

- `TADOConnection` is the connection component for ADO databases such as Microsoft Access and Microsoft SQL. Such datasets are `TADODataset`, `TADOTable`, `TADOQuery`, and `TADOStoredProc`. ADO database connectivity is covered in Chapter 9, “Database Development with dbGo for ADO.”
- `TSQLConnection` is the connection component for dbExpress based datasets. DbExpress datasets are special lightweight unidirectional datasets. These are `TSQLDataset`, `TSQLTable`, `TSQLQuery` and `TSQLStoredProc`. DbExpress is covered in Chapter 8, “Database Development with dbExpress.”
- `TIBDatabase` is the connection component for Interbase Express datasets. The datasets are `TIBDataSet`, `TIBTable`, `TIBQuery`, and `TIBStoredProc`. Interbase Express isn’t covered in this book because much of the functionality mimics the other connection methods.

Each of these datasets provides the common functionality contained in the `TCustomConnection` component. This common functionality includes methods, properties, and events related to

- Connecting and disconnecting to the data repository
- Login and support for establishing secure connections
- Dataset management

Establishing a Database Connection

Although each connection component surfaces many of the same methods for database connectivity, there are some differences. The reason for this is that each connection component provides the connection functionality of its underlying data repository. Therefore, the `TADOConnection` might function slightly differently from the `TDatabase` connection. The connection methods for `TSQLConnection` and `TADOConnection` are covered in their respective chapters (Chapters 8 and 9). Connecting to a BDE based dataset is covered in Chapter 28 in the CD copy of *Delphi 5 Developer’s Guide*.

Working with Datasets

A *dataset* is a collection of rows and columns of data. Each *column* is of some homogeneous data type, and each *row* is made up of a collection of data of each column data type. Additionally, a column is also known as a *field*, and a row is sometimes called a *record*. VCL encapsulates a dataset into an abstract component called `TDataSet`. `TDataSet` introduces many of the properties and methods necessary for manipulating and navigating a dataset and serves as the component from which special types of different datasets descend.

To help keep the nomenclature clear and to cover some of the basics, the following list explains some of the common database terms that are used in this and other database-oriented chapters:

- A *dataset* is a collection of discrete data records. Each record is made up of multiple fields. Each field can contain a different type of data (integer number, string, decimal number, graphic, and so on).
- A *table* is a special type of dataset. A table is generally a file containing records that are physically stored on a disk somewhere. `TTable`, `TADOTable`, `TSQLTable`, and `TIBTable` components encapsulate this functionality.
- A *query* is also a special type of dataset. Think of queries as commands that are executed against a database server. Such commands might result in resultsets (memory tables). These resultsets are the special datasets that are encapsulated by `TQuery`, `TADOQuery`, `TSQLQuery`, and `TIBQuery` components.

NOTE

We mentioned earlier that this chapter assumes a bit of database knowledge. This chapter isn't intended to be a primer on database programming, and we expect that you're already familiar with the items in this list. If terms such as *database*, *table*, and *index* sound foreign to you, you might want to obtain an introductory text on database concepts.

Opening and Closing Datasets

Before you can do anything with a dataset, you must first open it. To open a dataset, simply call its `Open()` method, as shown in this example:

```
Table1.Open;
```

This is equivalent, by the way, to setting a dataset's `Active` property to `True`:

```
Table1.Active := True;
```

There's slightly less overhead in the latter method because the `Open()` method ends up setting the `Active` property to `True`. However, the overhead is so minimal that it's not worth worrying about.

Once the dataset has been opened, you're free to manipulate it, as you'll see in just a moment. When you finish using the dataset, you should close it by calling its `Close()` method, like this:

```
Table1.Close;
```

Alternatively, you could close it by setting its `Active` property to `False`, like this:

```
Table1.Active :=  False;
```

TIP

When you're communicating with SQL servers, a connection to the database must be established when you first open a dataset in that database. When you close the last dataset in a database, your connection is terminated. Opening and closing these connections involves a certain amount of overhead. Therefore, if you find that you open and close the connection to the database often, use a `TDatabase` component instead to maintain a connection to a SQL server's database throughout many open and close operations. The `TDatabase` component is explained in more detail in the next chapter.

To illustrate how similar it is to open and close the different type of datasets, we've provide the example shown in Listing 7.1.

LISTING 7.1 Opening and Closing Datasets

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, FMTBcd, DBXpress, IBDatabase, ADODB, DBTables, DB, SqlExpr,
  IBCustomDataSet, IBQuery, IBTable, StdCtrls;

type
  TForm1 = class(TForm)
    SQLDataSet1: TSQLDataSet;
    SQLTable1: TSQLTable;
    SQLQuery1: TSQLQuery;

    ADOTable1: TADOTable;
    ADODataSet1: TADODataSet;
    ADOQuery1: TADOQuery;

    IBTable1: TIBTable;
    IBQuery1: TIBQuery;
    IBDataSet1: TIBDataSet;

    Table1: TTable;
    Query1: TQuery;

    SQLConnection1: TSQLConnection;
    Database1: TDatabase;
    ADOConnection1: TADOConnection;
```

LISTING 7.1 Continued

```
    IBDatabase1: TIBDatabase;
    Button1: TButton;
    Label1: TLabel;
    Button2: TButton;
    IBTransaction1: TIBTransaction;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
    procedure OpenDatasets;
    procedure CloseDatasets;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  IBDatabase1.Connected    := True;
  ADOConnection1.Connected := True;
  Database1.Connected      := True;
  SQLConnection1.Connected := True;
end;

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

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  CloseDatasets;
  IBDatabase1.Connected    := false;
  ADOConnection1.Connected := false;
  Database1.Connected      := false;
  SQLConnection1.Connected := false;
end;
```

LISTING 7.1 Continued

```
procedure TForm1.CloseDatasets;
begin

  // Disconnect from dbExpress datasets
  SQLDataSet1.Close;  // or .Active := false;
  SQLTable1.Close;    // or .Active := false;
  SQLQuery1.Close;    // or .Active := false;

  // Disconnect from ADO datasets
  ADOTable1.Close;    // or .Active := false;
  ADODataSet1.Close;  // or .Active := false;
  ADOQuery1.Close;    // or .Active := false;

  // Disconnect from Interbase Express datasets
  IBTable1.Close;    // or .Active := false;
  IBQuery1.Close;    // or .Active := false;
  IBDataSet1.Close;  // or .Active := false;

  // Disconnect from BDE datasets
  Table1.Close;    // or .Active := false;
  Query1.Close;    // or .Active := false;

  Label1.Caption := 'Datasets are closed.'
end;

procedure TForm1.OpenDatasets;
begin

  // Connect to dbExpress datasets
  SQLDataSet1.Open;  // or .Active := true;
  SQLTable1.Open;    // or .Active := true;
  SQLQuery1.Open;    // or .Active := true;

  // Connect to ADO datasets
  ADOTable1.Open;    // or .Active := true;
  ADODataSet1.Open;  // or .Active := true;
  ADOQuery1.Open;    // or .Active := true;

  // Connect to Interbase Express datasets
  IBTable1.Open;    // or .Active := true;
  IBQuery1.Open;    // or .Active := true;
  IBDataSet1.Open;  // or .Active := true;

  // Connect to BDE datasets
  Table1.Open;    // or .Active := true;
```

LISTING 7.1 Continued

```
  Query1.Open;    // or .Active := true;

  Label1.Caption := 'Datasets are open.';
end;

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

end.
```

This example is provided on the CD. You might have some problems setting up the database connections because the example was created on our development machine. You'll have to set up connections based on your machine. Nevertheless, the purpose of showing you this example was to illustrate the similarities of the different datasets.

Navigating Datasets

`TDataSet` provides some simple methods for basic record navigation. The `First()` and `Last()` methods move you to the first and last records in the dataset, respectively, and the `Next()` and `Prior()` methods move you either one record forward or back in the dataset. Additionally, the `MoveBy()` method, which accepts an `Integer` parameter, moves you a specified number of records forward or back.

`BOF`, `EOF`, and Looping

`BOF` and `EOF` are Boolean properties of `TDataSet` that reveal whether the current record is the first or last record in the dataset. For example, you might need to iterate through each record in a dataset until reaching the last record. The easiest way to do so would be to employ a `while` loop to keep iterating over records until the `EOF` property returns `True`, as shown here:

```
Table1.First;                          // go to beginning of data set
while not Table1.EOF do                // iterate over table
begin
  // do some stuff with current record
  Table1.Next;                         // move to next record
end;
```

CAUTION

Be sure to call the `Next()` method inside your `while-not-EOF` loop; otherwise, your application will get caught in an endless loop.

Avoid using a `repeat..until` loop to perform actions on a dataset. The following code might look okay on the surface, but bad things might happen if you try to use it on an empty dataset because the `DoSomeStuff()` procedure will always execute at least once, regardless of whether the dataset contains records:

```
repeat
  DoSomeStuff;
  Table1.Next;
until Table1.EOF;
```

Because the `while-not-EOF` loop performs the check up front, you won't encounter such a problem with this construct.

To illustrate how similar it is to navigate among the different type of datasets, we've provided the example shown in Listing 7.2.

LISTING 7.2 Navigation with the Different Datasets

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, FMTBcd, DBXpress, IBDatabase, ADODB, DBTables, DB, SqlExpr,
  IBCustomDataSet, IBQuery, IBTable, StdCtrls, Grids, DBGrids, ExtCtrls;

type
  TForm1 = class(TForm)
    SQLTable1: TSQLTable;
    ADOTable1: TADOTable;
    IBTable1: TIBTable;
    Table1: TTable;

    SQLConnection1: TSQLConnection;
    Database1: TDatabase;
    ADOConnection1: TADOConnection;
    IBDatabase1: TIBDatabase;
    Button1: TButton;
    Label1: TLabel;
    Button2: TButton;
    IBTransaction1: TIBTransaction;
    DBGrid1: TDBGrid;
    DataSource1: TDataSource;
    RadioGroup1: TRadioGroup;
    btnFirst: TButton;
```

LISTING 7.2 Continued

```
    btnLast: TButton;
    btnNext: TButton;
    btnPrior: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure Button2Click(Sender: TObject);
    procedure RadioGroup1Click(Sender: TObject);
    procedure btnFirstClick(Sender: TObject);
    procedure btnLastClick(Sender: TObject);
    procedure btnNextClick(Sender: TObject);
    procedure btnPriorClick(Sender: TObject);
    procedure DataSource1DataChange(Sender: TObject; Field: TField);
  private
    { Private declarations }
    procedure OpenDatasets;
    procedure CloseDatasets;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.FormCreate(Sender: TObject);
begin
  IBDatabase1.Connected    := True;
  ADOConnection1.Connected := True;
  Database1.Connected      := True;
  SQLConnection1.Connected := True;

  Datasource1.DataSet := IBTable1;
  OpenDatasets;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  OpenDatasets;
end;
```

LISTING 7.2 Continued

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  CloseDatasets;
  IBDatabase1.Connected    := false;
  ADOConnection1.Connected := false;
  Database1.Connected      := false;
  SQLConnection1.Connected := false;
end;

procedure TForm1.CloseDatasets;
begin

  // Disconnect from dbExpress dataset
  SQLTable1.Close;    // or .Active := false;

  // Disconnect from ADO dataset
  ADOTable1.Close;    // or .Active := false;

  // Disconnect from Interbase Express dataset
  IBTable1.Close;    // or .Active := false;

  // Disconnect from BDE datasets
  Table1.Close;    // or .Active := false;

  Label1.Caption := 'Datasets are closed.'
end;

procedure TForm1.OpenDatasets;
begin

  // Connect to dbExpress dataset
  SQLTable1.Open;    // or .Active := true;

  // Connect to ADO dataset
  ADOTable1.Open;    // or .Active := true;

  // Connect to Interbase Express dataset
  IBTable1.Open;    // or .Active := true;

  // Connect to BDE dataset
  Table1.Open;    // or .Active := true;

  Label1.Caption := 'Datasets are open.';
end;
```

LISTING 7.2 Continued

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  CloseDatasets;
end;

procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  case RadioGroup1.ItemIndex of
    0: Datasource1.DataSet := IBTable1;
    1: Datasource1.DataSet := Table1;
    2: Datasource1.DataSet := ADOTable1;
  end; // case
end;

procedure TForm1.btnFirstClick(Sender: TObject);
begin
  DataSource1.DataSet.First;
end;

procedure TForm1.btnLastClick(Sender: TObject);
begin
  DataSource1.DataSet.Last;
end;

procedure TForm1.btnNextClick(Sender: TObject);
begin
  DataSource1.DataSet.Next;
end;

procedure TForm1.btnPriorClick(Sender: TObject);
begin
  DataSource1.DataSet.Prior;
end;

procedure TForm1.DataSource1DataChange(Sender: TObject; Field: TField);
begin
  btnLast.Enabled := not DataSource1.DataSet.Eof;
  btnNext.Enabled := not DataSource1.DataSet.Eof;
  btnFirst.Enabled := not DataSource1.DataSet.Bof;
  btnPrior.Enabled := not DataSource1.DataSet.Bof;
end;

end.
```

In this example, a `TRadioGroup` is used to allow the user to select from three of the database types. Additionally, the `OnDataChange` event handler shows how to evaluate the BOF and EOF properties to properly enable or disable the buttons when one of the two are true. You should notice that the same methods are invoked to navigate through the dataset regardless of which dataset is selected.

NOTE

You'll notice that we did not include the dbExpress component as part of this example. This is because dbExpress datasets are unidirectional datasets. That is, they can only navigate in one direction and are treated as read-only. In fact, if you attempt to connect a navigable component such as a `TDBGrid` to a dbExpress dataset, you will get an error. Navigating through unidirectional datasets requires some specific setup, which is discussed in Chapter 8.

Manipulating Datasets

A database application isn't really a database application unless you can manipulate its data. Fortunately, datasets provide methods that allow you to do this. With datasets, you are able to add, edit, and delete records from the underlying table. The methods to do this are appropriately named `Insert()`, `Edit()`, and `Delete()`.

Listing 7.3 shows a simple application illustrating how to use these methods.

LISTING 7.3 `MainFrm.pas`—Showing Simple Data Manipulation

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Mask, DBCtrls, DB, Grids, DBGrids, ADODB;

type
  TMainForm = class(TForm)
    ADOConnection1: TADOConnection;
    adodsCustomer: TADODataSet;
    dtsrcCustomer: TDataSource;
    DBGrid1: TDBGrid;
    adodsCustomerCustNo: TAutoIncField;
    adodsCustomerCompany: TWideStringField;
    adodsCustomerAddress1: TWideStringField;
```

LISTING 7.3 Continued

```
    adodsCustomerAddress2: TWideStringField;
    adodsCustomerCity: TWideStringField;
    adodsCustomerStateAbbr: TWideStringField;
    adodsCustomerZip: TWideStringField;
    adodsCustomerCountry: TWideStringField;
    adodsCustomerPhone: TWideStringField;
    adodsCustomerFax: TWideStringField;
    adodsCustomerContact: TWideStringField;
    Label1: TLabel;
    dbedtCompany: TDBEdit;
    Label2: TLabel;
    dbedtAddress1: TDBEdit;
    Label3: TLabel;
    dbedtAddress2: TDBEdit;
    Label4: TLabel;
    dbedtCity: TDBEdit;
    Label5: TLabel;
    dbedtState: TDBEdit;
    Label6: TLabel;
    dbedtZip: TDBEdit;
    Label7: TLabel;
    dbedtPhone: TDBEdit;
    Label8: TLabel;
    dbedtFax: TDBEdit;
    Label9: TLabel;
    dbedtContact: TDBEdit;
    btnAdd: TButton;
    btnEdit: TButton;
    btnSave: TButton;
    btnCancel: TButton;
    Label10: TLabel;
    dbedtCountry: TDBEdit;
    btnDelete: TButton;
    procedure btnAddClick(Sender: TObject);
    procedure btnEditClick(Sender: TObject);
    procedure btnSaveClick(Sender: TObject);
    procedure btnCancelClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure btnDeleteClick(Sender: TObject);
  private
    { Private declarations }
    procedure SetButtons;
  public
```

LISTING 7.3 Continued

```
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.dfm}

procedure TMainForm.btnAddClick(Sender: TObject);
begin
  adodsCustomer.Insert;
  SetButtons;
end;

procedure TMainForm.btnEditClick(Sender: TObject);
begin
  adodsCustomer.Edit;
  SetButtons;
end;

procedure TMainForm.btnSaveClick(Sender: TObject);
begin
  adodsCustomer.Post;
  SetButtons;
end;

procedure TMainForm.btnCancelClick(Sender: TObject);
begin
  adodsCustomer.Cancel;
  SetButtons;
end;

procedure TMainForm.SetButtons;
begin
  btnAdd.Enabled  := adodsCustomer.State = dsBrowse;
  btnEdit.Enabled := adodsCustomer.State = dsBrowse;
  btnSave.Enabled := (adodsCustomer.State = dsInsert) or
    (adodsCustomer.State = dsEdit);
  btnCancel.Enabled := (adodsCustomer.State = dsInsert) or
    (adodsCustomer.State = dsEdit);
  btnDelete.Enabled := adodsCustomer.State = dsBrowse;
end;
```

LISTING 7.3 Continued

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  adodsCustomer.Open;
  SetButtons;

end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  adodsCustomer.Close;
  ADOConnection1.Connected := False;
end;

procedure TMainForm.btnDeleteClick(Sender: TObject);
begin
  adodsCustomer.Delete;
end;

end.
```

Figure 7.2 illustrates a simple data manipulation application.

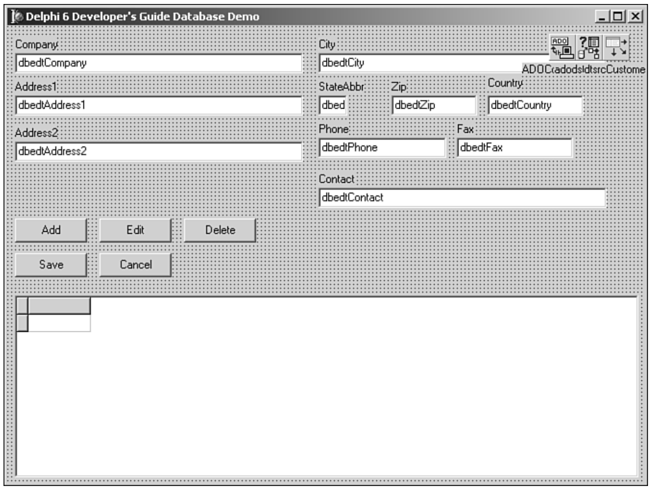


FIGURE 7.2
Main form for the data manipulation application.

This application manipulates data in the simplest form. You'll see the use of the manipulation methods listed as follows:

- `Insert()` allows the user to insert a new record.
- `Edit()` allows the user to modify the active record.
- `Post()` saves changes to a new or existing record to the table.
- `Cancel()` cancels any changes made to the record.
- `Delete()` deletes the active record from the table.

Dataset States

Listing 7.3 also shows how we referred to the `TDataSet.State` property to examine the dataset's state so that we could enable or disable our buttons appropriately. This allows us to do things such as disable our Add button when the dataset is already in Insert or Edit mode. Other states are shown in Table 7.1.

TABLE 7.1 Values for `TDataset.State`

Value	*Meaning*
`dsBrowse`	The dataset is in Browse (normal) mode.
`dsCalcFields`	The `OnCalcFields` event has been called, and a record value calculation is in progress.
`dsEdit`	The dataset is in Edit mode. This means that the `Edit()` method has been called, but the edited record hasn't yet been posted.
`dsInactive`	The dataset is closed.
`dsInsert`	The dataset is in Insert mode. This typically means that `Insert()` has been called but changes haven't been posted.
`dsSetKey`	The dataset is in SetKey mode, meaning that `SetKey()` has been called but `GotoKey()` hasn't yet been called.
`dsNewValue`	The dataset is in a temporary state where the `NewValue` property is being accessed.
`dsOldValue`	The dataset is in a temporary state where the `OldValue` property is being accessed.
`dsCurValue`	The dataset is in a temporary state where the `OldValue` property is being accessed.
`dsFilter`	The dataset is currently processing a record filter, lookup, or some other operation that requires a filter.
`dsBlockRead`	Data is being buffered en masse, so data-aware controls are not updated and events are not triggered when the cursor moves while this member is set.

TABLE 7.1 Continued

Value	*Meaning*
`dsInternalCalc`	A field value is currently being calculated for a field that has a `FieldKind` of `fkInternalCalc`.
`dsOpening`	The dataSet is in the process of opening but has not finished. This state occurs when the dataset is opened for asynchronous fetching.

Working with Fields

Delphi enables you to access the fields of any dataset through the `TField` object and its descendants. Not only can you get and set the value of a given field of the current record of a dataset, but you can also change the behavior of a field by modifying its properties. You can also modify the dataset, itself, by changing the visual order of fields, removing fields, or even creating new calculated or lookup fields.

Field Values

It's very easy to access field values from Delphi. `TDataSet` provides a default array property called `FieldValues[]` that returns the value of a particular field as a `Variant`. Because `FieldValues[]` is the default array property, you don't need to specify the property name to access the array. For example, the following piece of code assigns the value of Table1's `CustName` field to `String S`:

```
S := Table1['CustName'];
```

You could just as easily store the value of an integer field called `CustNo` in an integer variable called `I`:

```
I := Table1['CustNo'];
```

A powerful corollary to this is the capability to store the values of several fields into a `Variant` array. The only catches are that the `Variant` array index must be zero based and the `Variant` array contents should be `varVariant`. The following code demonstrates this capability:

```
const
  AStr = 'The %s is of the %s category and its length is %f in.';
var
  VarArr: Variant;
  F: Double;
begin
  VarArr := VarArrayCreate([0, 2], varVariant);
  { Assume Table1 is attached to Biolife table }
  VarArr := Table1['Common_Name;Category;Length_In'];
  F := VarArr[2];
  ShowMessage(Format(AStr, [VarArr[0], VarArr[1], F]));
end;
```

You can also use the `TDataset.Fields[]` array property or `FieldsByName()` function to access individual `TField` objects associated with the dataset. The `TField` component provides information about a specific field.

`Fields[]` is a zero-based array of `TField` objects, so `Fields[0]` returns a `TField` representing the first logical field in the record. `FieldsByName()` accepts a string parameter that corresponds to a given field name in the table; therefore, `FieldsByName('OrderNo')` would return a `TField` component representing the `OrderNo` field in the current record of the dataset.

Given a `TField` object, you can retrieve or assign the field's value using one of the `TField` properties shown in Table 7.2.

TABLE 7.2 Properties to Access `TField` Values

Property	*Return Type*
`AsBoolean`	`Boolean`
`AsFloat`	`Double`
`AsInteger`	`Longint`
`AsString`	`String`
`AsDateTime`	`TDateTime`
`Value`	`Variant`

If the first field in the current dataset is a string, you can store its value in the `String` variable `S`, like this:

```
S := Table1.Fields[0].AsString;
```

The following code sets the integral variable `I` to contain the value of the `'OrderNo'` field in the current record of the table:

```
I := Table1.FieldsByName('OrderNo').AsInteger;
```

Field Data Types

If you want to know the type of a field, look at `TField`'s `DataType` property, which indicates the data type with respect to the database table (irrespective of a corresponding Object Pascal type). The `DataType` property is of `TFieldType`, and `TFieldType` is defined as follows:

```
type
  TFieldType = (ftUnknown, ftString, ftSmallint, ftInteger, ftWord,
    ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime,
    ftBytes, ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo,
    ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor, ftFixedChar,
    ftWideString, ftLargeint, ftADT, ftArray, ftReference, ftDataSet,
    ftOraBlob, ftOraClob, ftVariant, ftInterface, ftIDispatch, ftGuid);
```

Descendants of `TField` are designed to work specifically with many of the preceding data types. These are covered a bit later in this chapter.

Field Names and Numbers

To find the name of a specified field, use the `TField.FieldName` property. For example, the following code places the name of the first field in the current table in the `String` variable `S`:

```
var
  S: String;
begin
  S := Table1.Fields[0].FieldName;
end;
```

Likewise, you can obtain the number of a field you know only by name by using the `FieldNo` property. The following code stores the number of the `OrderNo` field in the `Integer` variable `I`:

```
var
  I: integer;
begin
  I := Table1.FieldsByName('OrderNo').FieldNo;
end;
```

NOTE

To determine how many fields a dataset contains, use `TDataset`'s `FieldList` property. `FieldList` represents a flattened view of all the nested fields in a table containing fields that are abstract data types.

For backward compatibility, the `FieldCount` property still works, but it will skip over any ADT fields.

Manipulating Field Data

Here's a three-step process for editing one or more fields in the current record:

1. Call the dataset's `Edit()` method to put the dataset into Edit mode.
2. Assign new values to the fields of your choice.

3. Post the changes to the dataset either by calling the `Post()` method or by moving to a new record, which will automatically post the edit.

For instance, a typical record edit looks like this:

```
Table1.Edit;
Table1['Age'] := 23;
Table1.Post;
```

TIP

Sometimes you work with datasets that contain read-only data. Examples of this would include a table located on a CD-ROM drive or a query with a non-live resultset. Before attempting to edit data, you can determine whether the dataset contains read-only data before you try to modify it by checking the value of the `CanModify` property. If `CanModify` is `True`, you have the green light to edit the dataset.

The Fields Editor

Delphi gives you a great degree of control and flexibility when working with dataset fields through the Fields Editor. You can view the Fields Editor for a particular dataset in the Form Designer, either by double-clicking the `TTable`, `TQuery`, or `TStoredProc` or by selecting Fields Editor from the dataset's local menu. The Fields Editor window enables you to determine which of a dataset's fields you want to work with and create new calculated or lookup fields. You can use a local menu to accomplish these tasks. The Fields Editor window with its local menu deployed is shown in Figure 7.3.

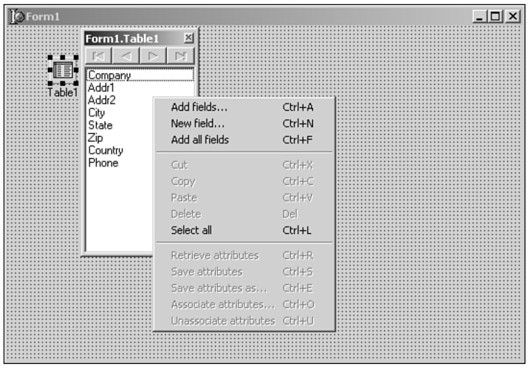


FIGURE 7.3
The Fields Editor's local menu.

To demonstrate the usage of the Fields Editor, open a new project and drop a `TTable` component onto the main form. Set the `Table1.DatabaseName` property to `DBDEMOS` (this is the alias that points to the Delphi sample tables) and set the `TableName` property to `ORDERS.DB`. To provide some visual feedback, also drop a `TDataSource` and `TDBGrid` component on the form. Hook `DataSource1` to `Table1` and then hook `DBGrid1` to `DataSource1`. Now set `Table1`'s `Active` property to `True`, and you'll see `Table1`'s data in the grid.

Adding Fields

Invoke the Fields Editor by double-clicking `Table1`, and you'll see the Fields Editor window, as shown in Figure 7.3. Let's say that you want to limit your view of the table to only a few fields. Select Add Fields from the Fields Editor local menu. This will invoke the Add Fields dialog box. Highlight the `OrderNo`, `CustNo`, and `ItemsTotal` fields in this dialog box and click OK. The three selected fields will now be visible in the Fields Editor and in the grid.

Delphi creates `TField` descendant objects, which map to the dataset fields you select in the Fields Editor. For example, for the three fields mentioned in the preceding paragraph, Delphi adds the following declarations of `TField` descendants to the source code for your form:

```
Table1OrderNo: TFloatField;
Table1CustNo: TFloatField;
Table1ItemsTotal: TCurrencyField;
```

Notice that the name of the field object is the concatenation of the `TTable` name and the field name. Because these fields are created in code, you can also access `TField` descendant properties and methods in your code rather than solely at design time.

`TField` Descendants

There are one or more different `TField` descendant objects for each field type. (Field types are described in the "Field Data Types" section, earlier in this chapter.) Many of these field types also map to Object Pascal data types. Table 7.3 shows the various classes in the `TField` hierarchy, their ancestor classes, their field types, and the Object Pascal types to which they equate.

TABLE 7.3 `TField` Descendants and Their Field Types

Field Class	*Ancestor*	*Field Type*	*Object Pascal Type*
TStringField	TField	ftString	String
TWideStringField	TStringField	ftWideString	WideString
TGuidField	TStringField	ftGuid	TGUID
TNumericField	TField	*	*
TIntegerField	TNumericField	ftInteger	Integer
TSmallIntField	TIntegerField	ftSmallInt	SmallInt

TABLE 7.3 Continued

Field Class	*Ancestor*	*Field Type*	*Object Pascal Type*
TLargeintField	TNumericField	ftLargeint	Int64
TWordField	TIntegerField	ftWord	Word
TAutoIncField	TIntegerField	ftAutoInc	Integer
TFloatField	TNumericField	ftFloat	Double
TCurrencyField	TFloatField	ftCurrency	Currency
TBCDField	TNumericField	ftBCD	Double
TBooleanField	TField	ftBoolean	Boolean
TDateTimeField	TField	ftDateTime	TDateTime
TDateField	TDateTimeField	ftDate	TDateTime
TTimeField	TDateTimeField	ftTime	TDateTime
TBinaryField	TField	*	*
TBytesField	TBinaryField	ftBytes	*none*
TVarBytesField	TBytesField	ftVarBytes	*none*
TBlobField	TField	ftBlob	*none*
TMemoField	TBlobField	ftMemo	*none*
TGraphicField	TBlobField	ftGraphic	*none*
TObjectField	TField	*	*
TADTField	TObjectField	ftADT	*none*
TArrayField	TObjectField	ftArray	*none*
TDataSetField	TObjectField	ftDataSet	TDataSet
TReferenceField	TDataSetField	ftReference	
TVariantField	TField	ftVariant	OleVariant
TInterfaceField	TField	ftInterface	IUnknown
TIDispatchField	TInterfaceField	ftIDispatch	IDispatch
TAggregateField	TField	*none*	*none*

**Denotes an abstract base class in the TField hierarchy*

As Table 7.3 shows, BLOB and Object field types are special in that they don’t map directly to native Object Pascal types. BLOB fields are discussed in more detail later in this chapter.

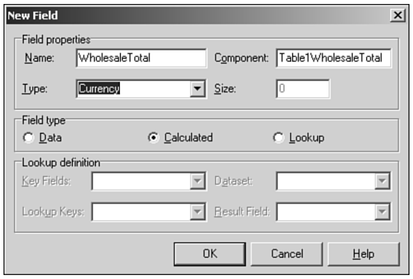
Fields and the Object Inspector

When you select a field in the Fields Editor, you can access the properties and events associated with that `TField` descendant object in the Object Inspector. This feature enables you to modify field properties such as minimum and maximum values, display formats, and whether the field is required as well as whether it's read-only. Some of these properties, such as `ReadOnly`, are obvious in their purpose, but some aren't quite as intuitive.

Switch to the Events page of the Object Inspector, and you'll see that there are also events associated with field objects. The events `OnChange`, `OnGetText`, `OnSetText`, and `OnValidate` are all well-documented in the online help. Simply click to the left of the event in the Object Inspector and press F1. Of these, `OnChange` is probably the most common to use. It enables you to perform some action whenever the contents of the field change (moving to another record or adding a record, for example).

Calculated Fields

You can also add calculated fields to a dataset using the Fields Editor. Let's say, for example, that you wanted to add a field that figures the wholesale total for each entry in the `ORDERS` table, and the wholesale total was 32% of the normal total. Select New Field from the Fields Editor local menu, and you'll be presented with the New Field dialog box, as shown in Figure 7.4. Enter the name, `WholesaleTotal`, for the new field in the Name edit control. The type of this field is Currency, so enter that in the Type edit control. Make sure that the Calculated radio button is selected in the Field Type group; then press OK. Now the new field will show up in the grid, but it won't yet contain any data.


New Field

Field properties
Name: WholesaleTotal
Component: Table1WholesaleTotal
Type: Currency
Size: 0

Field type
Data [✓ Calculated] Lookup

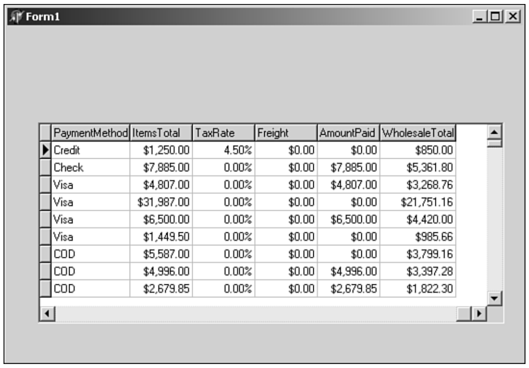
Lookup definition
Key Fields:
Dataset:
Lookup Keys:
Result Field:

FIGURE 7.4
Adding a calculated field with the New Field dialog box.

To cause the new field to become populated with data, you must assign a method to the `Table1.OnCalcFields` event. The code for this event simply assigns the value of the `WholesaleTotal` field to be 32% of the value of the existing `SalesTotal` field. This method, which handles `Table1.OnCalcFields`, is shown here:

```
procedure TForm1.Table1CalcFields(DataSet: TDataSet);
begin
  DataSet['WholesaleTotal'] := DataSet['ItemsTotal'] * 0.68;
end;
```

Figure 7.5 shows that the `WholesaleTotal` field in the grid now contains the correct data.



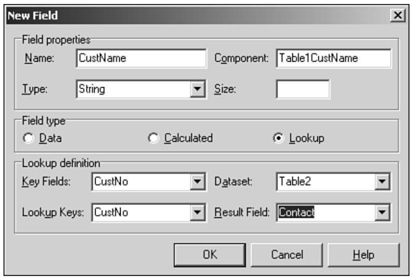
PaymentMethod	ItemsTotal	TaxRate	Freight	AmountPaid	WholesaleTotal
Credit	$1,250.00	4.50%	$0.00	$0.00	$850.00
Check	$7,885.00	0.00%	$0.00	$7,885.00	$5,361.80
Visa	$4,807.00	0.00%	$0.00	$4,807.00	$3,268.76
Visa	$31,987.00	0.00%	$0.00	$0.00	$21,751.16
Visa	$6,500.00	0.00%	$0.00	$6,500.00	$4,420.00
Visa	$1,449.50	0.00%	$0.00	$0.00	$985.66
COD	$5,587.00	0.00%	$0.00	$0.00	$3,799.16
COD	$4,996.00	0.00%	$0.00	$4,996.00	$3,397.28
COD	$2,679.85	0.00%	$0.00	$2,679.85	$1,822.30

FIGURE 7.5
The calculated field has been added to the table.

Lookup Fields

Lookup fields enable you to create fields in a dataset that actually look up their values from another dataset. To illustrate this, you'll add a lookup field to the current project. The `CustNo` field of the `ORDERS` table doesn't mean anything to someone who doesn't have all the customer numbers memorized. You can add a lookup field to `Table1` that looks into the `CUSTOMER` table and then, based on the customer number, retrieves the name of the current customer.

First, you should drop in a second `TTable` object, setting its `DatabaseName` property to `DBDEMOS` and its `TableName` property to `CUSTOMER`. This is `Table2`. Then you once again select New Field from the Fields Editor local menu to invoke the New Field dialog box. This time, you'll call the field `CustName`, and the field type will be a `String`. The size of the string is 15 characters. Don't forget to select the Lookup button in the Field Type radio group. The Dataset control in this dialog box should be set to `Table2`—the dataset you want to look into. The Key Fields and Lookup Keys controls should be set to `CustNo`—this is the common field upon which the lookup will be performed. Finally, the Result field should be set to `Contact`—this is the field you want displayed. Figure 7.6 shows the New Field dialog box for the new lookup field. The new field will now display the correct data, as shown in the completed project in Figure 7.7.



Field properties

Name: CustName
Component: Table1CustName
Type: String
Size:

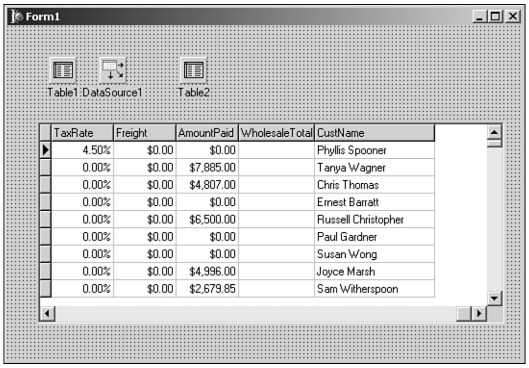
Field type

( ) Data ( ) Calculated [✓ Lookup]

Lookup definition

Key Fields: CustNo
Dataset: Table2
Lookup Keys: CustNo
Result Field: Contact

FIGURE 7.6
Adding a lookup field with the New Field dialog box.



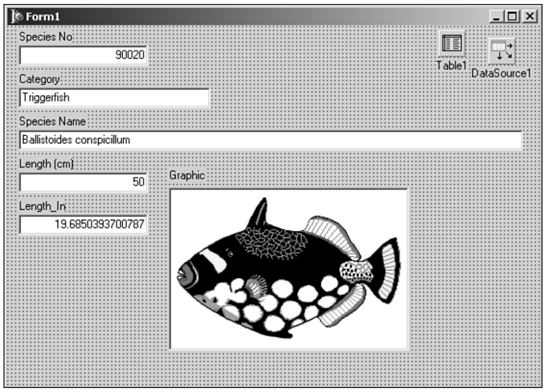
TaxRate	Freight	AmountPaid	WholesaleTotal	CustName
4.50%	$0.00	$0.00		Phyllis Spooner
0.00%	$0.00	$7,885.00		Tanya Wagner
0.00%	$0.00	$4,807.00		Chris Thomas
0.00%	$0.00	$0.00		Ernest Barratt
0.00%	$0.00	$6,500.00		Russell Christopher
0.00%	$0.00	$0.00		Paul Gardner
0.00%	$0.00	$0.00		Susan Wong
0.00%	$0.00	$4,996.00		Joyce Marsh
0.00%	$0.00	$2,679.85		Sam Witherspoon

FIGURE 7.7
Viewing the table containing a lookup field.

Drag-and-Drop Fields

Another less obvious feature of the Fields Editor is that it enables you to drag fields from its Fields list box and drop them onto your forms. We can easily demonstrate this feature by starting a new project that contains only a `TTable` on the main form. Assign `Table1.DatabaseName` to `DBDEMOS` and assign `Table1.TableName` to `BIOLIFE.DB`. Invoke the Fields Editor for this table and add all the fields in the table to the Fields Editor list box. You can now drag one or more of the fields at a time from the Fields Editor window and drop them on your main form.

You'll notice a couple of cool things happening here: First, Delphi senses what kind of field you're dropping onto your form and creates the appropriate data-aware control to display the data (that is, a `TDBEdit` is created for a string field, whereas a `TDBImage` is created for a graphic field). Second, Delphi checks to see if you have a `TDataSource` object connected to the dataset; it hooks to an existing one if available or creates one if needed. Figure 7.8 shows the result of dragging and dropping the fields of the `BIOLIFE` table onto a form.



Species No: 90020
Category: Triggerfish
Species Name: Ballistoides conspicillum
Length (cm): 50
Length_In: 19.6850393700787

FIGURE 7.8
Dragging and dropping fields on a form.

Working with BLOB Fields

A BLOB (Binary Large Object) field is a field that's designed to contain an indeterminate amount of data. A BLOB field in one record of a dataset might contain three bytes of data, whereas the same field in another record of that dataset might contain 3KB. Blobs are most useful for holding large amounts of text, graphic images, or raw data streams such as OLE objects.

`TBlobField` and Field Types

As discussed earlier, VCL includes a `TField` descendant called `TBlobField`, which encapsulates a BLOB field. `TBlobField` has a `BlobType` property of type `TBlobType`, which indicates what type of data is stored in the BLOB field. `TBlobType` is defined in the `DB` unit as follows:

```
TBlobType = ftBlob..ftOraClob;
```

All these field types and the type of data associated with these field types are listed in Table 7.4.

TABLE 7.4 `TBlobField` Field Types

Field Type	*Type of Data*
`ftBlob`	Untyped or user-defined data
`ftMemo`	Text
`ftGraphic`	Windows bitmap
`ftFmtMemo`	Paradox formatted memo
`ftParadoxOle`	Paradox OLE object
`ftDBaseOLE`	dBASE OLE object

TABLE 7.4 Continued

Field Type	*Type of Data*
`ftTypedBinary`	Raw data representation of an existing type
`ftCursor..ftDataSet`	Not valid BLOB types
`ftOraBlob`	BLOB fields in Oracle8 tables
`ftOraClob`	CLOB fields in Oracle8 tables

You'll find that most of the work you need to do in getting data in and out of `TBlobField` components can be accomplished by loading or saving the BLOB to a file or by using a `TBlobStream`. `TBlobStream` is a specialized descendant of `TStream` that uses the BLOB field inside the physical table as the stream location. To demonstrate these techniques for interacting with `TBlobField` components, you'll create a sample application.

BLOB Field Example

This project creates an application that enables the user to store WAV files in a database table and play them directly from the table. Start the project by creating a main form with the components shown in Figure 7.9. The `TTable` component can map to the `Wavez` table in the `DDGData` alias or your own table of the same structure. The structure of the table is as follows:

Field Name	*Field Type*	*Size*
`WaveTitle`	Character	25
`FileName`	Character	25
`Wave`	BLOB	

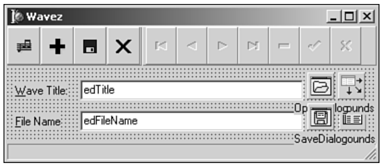


FIGURE 7.9
Main form for Wavez, the BLOB field example.

The Add button is used to load a WAV file from disk and add it to the table. The method assigned to the `OnClick` event of the Add button is shown here:

```
procedure TMainForm.sbAddClick(Sender: TObject);
begin
  if OpenDialog.Execute then
  begin
    tblSounds.Append;
    tblSounds['FileName'] := ExtractFileName(OpenDialog.FileName);
    tblSoundsWave.LoadFromFile(OpenDialog.FileName);
    edTitle.SetFocus;
  end;
end;
```

The code first attempts to execute `OpenDialog`. If it's successful, `tblSounds` is put into Append mode, the `FileName` field is assigned a value, and the `Wave` BLOB field is loaded from the file specified by `OpenDialog`. Notice that `TBlobField`'s `LoadFromFile` method is very handy here, and the code is very clean for loading a file into a BLOB field.

Similarly, the Save button saves the current WAV sound found in the `Wave` field to an external file. The code for this button is as follows:

```
procedure TMainForm.sbSaveClick(Sender: TObject);
begin
  with SaveDialog do
  begin
    FileName := tblSounds['FileName'];    // initialize file name
    if Execute then                       // execute dialog
      tblSoundsWave.SaveToFile(FileName); // save blob to file
  end;
end;
```

There's even less code here. `SaveDialog` is initialized with the value of the `FileName` field. If `SaveDialog`'s execution is successful, the `tblSoundsWave.SaveToFile()` method is called to save the contents of the BLOB field to the file.

The handler for the Play button does the work of reading the WAV data from the BLOB field and passing it to the `PlaySound()` API function to be played. The code for this handler, shown next, is a bit more complex than the code shown thus far:

```
procedure TMainForm.sbPlayClick(Sender: TObject);
var
  B: TBlobStream;
  M: TMemoryStream;
begin
  B := TBlobStream.Create(tblSoundsWave, bmRead); // create blob stream
  Screen.Cursor := crHourGlass;                   // wait hourglass
  try
    M := TMemoryStream.Create;                    // create memory stream
    try
      M.CopyFrom(B, B.Size);             // copy from blob to memory stream
      // Attempt to play sound. Raise exception if something goes wrong
      Win32Check(PlaySound(M.Memory, 0, SND_SYNC or SND_MEMORY));
    finally
      M.Free;
    end;
  finally
    Screen.Cursor := crDefault;
    B.Free;                                     // clean up
  end;
end;
```

The first thing this method does is to create an instance of `TBlobStream`, `B`, using the `tblSoundsWave` BLOB field. The first parameter passed to `TBlobStream.Create()` is the BLOB field object, and the second parameter indicates how you want to open the stream. Typically, you'll use `bmRead` for read-only access to the BLOB stream or `bmReadWrite` for read/write access.

TIP

The dataset must be in Edit, Insert, or Append mode to open a `TBlobStream` with `bmReadWrite` privilege.

An instance of `TMemoryStream`, `M`, is then created. At this point, the cursor shape is changed to an hourglass to let the user know that the operation may take a couple of seconds. The stream `B` is then copied to the stream `M`. The function used to play a WAV sound, `PlaySound()`, requires a filename or a memory pointer as its first parameter. `TBlobStream` doesn't provide pointer access to the stream data, but `TMemoryStream` does through its `Memory` property. Given that, you can successfully call `PlaySound()` to play the data pointed at by `M.Memory`. Once the function is called, it cleans up by freeing the streams and restoring the cursor. The complete code for the main unit of this project is shown in Listing 7.4.

LISTING 7.4 The Main Unit for the Wavez Project

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, DBCtrls, DB, DBTables, StdCtrls, Mask, Buttons, ComCtrls;

type
  TMainForm = class(TForm)
    tblSounds: TTable;
    dsSounds: TDataSource;
```

LISTING 7.4 Continued

```
    tblSoundsWaveTitle: TStringField;
    tblSoundsWave: TBlobField;
    edTitle: TDBEdit;
    edFileName: TDBEdit;
    Label1: TLabel;
    Label2: TLabel;
    OpenDialog: TOpenDialog;
    tblSoundsFileName: TStringField;
    SaveDialog: TSaveDialog;
    pnlToobar: TPanel;
    sbPlay: TSpeedButton;
    sbAdd: TSpeedButton;
    sbSave: TSpeedButton;
    sbExit: TSpeedButton;
    Bevel1: TBevel;
    dbnNavigator: TDBNavigator;
    stbStatus: TStatusBar;
    procedure sbPlayClick(Sender: TObject);
    procedure sbAddClick(Sender: TObject);
    procedure sbSaveClick(Sender: TObject);
    procedure sbExitClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    procedure OnAppHint(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses MMSystem;

procedure TMainForm.sbPlayClick(Sender: TObject);
var
  B: TBlobStream;
  M: TMemoryStream;
begin
  B := TBlobStream.Create(tblSoundsWave, bmRead); // create blob stream
  Screen.Cursor := crHourGlass;                   // wait hourglass
  try
```

LISTING 7.4 Continued

```
    M := TMemoryStream.Create;                    // create memory stream
    try
      M.CopyFrom(B, B.Size);             // copy from blob to memory stream
      // Attempt to play sound. Raise exception if something goes wrong
      Win32Check(PlaySound(M.Memory, 0, SND_SYNC or SND_MEMORY));
    finally
      M.Free;
    end;
  finally
    Screen.Cursor := crDefault;
    B.Free;                                       // clean up
  end;
end;

procedure TMainForm.sbAddClick(Sender: TObject);
begin
  if OpenDialog.Execute then
  begin
    tblSounds.Append;
    tblSounds['FileName'] := ExtractFileName(OpenDialog.FileName);
    tblSoundsWave.LoadFromFile(OpenDialog.FileName);
    edTitle.SetFocus;
  end;
end;

procedure TMainForm.sbSaveClick(Sender: TObject);
begin
  with SaveDialog do
  begin
    FileName := tblSounds['FileName'];    // initialize file name
    if Execute then                       // execute dialog
      tblSoundsWave.SaveToFile(FileName); // save blob to file
  end;
end;

procedure TMainForm.sbExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnHint := OnAppHint;
  tblSounds.Open;
end;
```

LISTING 7.4 Continued

```
procedure TMainForm.OnAppHint(Sender: TObject);
begin
  stbStatus.SimpleText := Application.Hint;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  tblSounds.Close;
end;

end.
```

Filtering Data

Filters enable you to do simple dataset searching or filtering using only Object Pascal code. The primary advantage of using filters is that they don't require an index or any other preparation on the datasets with which they're used. In many cases, filters can be a bit slower than index-based searching (which is covered later in this chapter), but they're still very usable in almost any type of application.

Using `TDataset's` Filtering Capabilities

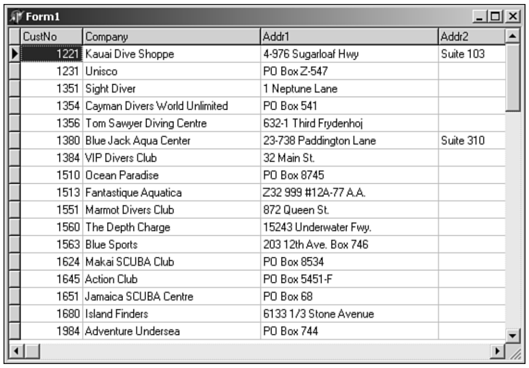
One of the more common uses of Delphi's filtering mechanism is to limit a view of a dataset to some specific records only. This is a simple two-step process:

1. Assign a procedure to the dataset's `OnFilterRecord` event. Inside of this procedure, you should write code that accepts records based on the values of one or more fields.
2. Set the dataset's `Filtered` property to `True`.

As an example, Figure 7.10 shows a form containing `TDBGrid`, which displays an unfiltered view of Delphi's `CUSTOMER` table.

In step 1, you write a handler for the table's `OnFilterRecord` event. In this case, we'll accept only records whose `Company` field starts with the letter *S*. The code for this procedure is shown here:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
var
  FieldVal: String;
begin
  FieldVal := DataSet['Company'];  // Get the value of the Company field
  Accept := FieldVal[1] = 'S';     // Accept record if field starts with 'S'
end;
```

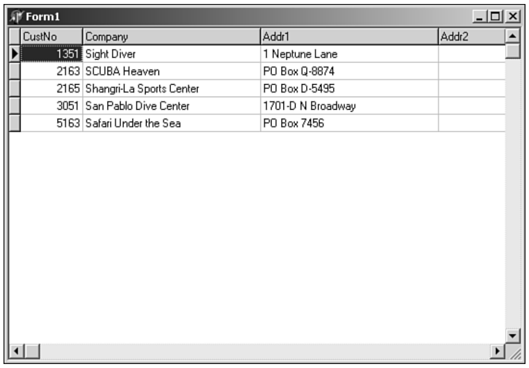


Form1

CustNo	Company	Addr1	Addr2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
1384	VIP Divers Club	32 Main St.	
1510	Ocean Paradise	PO Box 8745	
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	
1551	Marmot Divers Club	872 Queen St.	
1560	The Depth Charge	15243 Underwater Fwy.	
1563	Blue Sports	203 12th Ave. Box 746	
1624	Makai SCUBA Club	PO Box 8534	
1645	Action Club	PO Box 5451-F	
1651	Jamaica SCUBA Centre	PO Box 68	
1680	Island Finders	6133 1/3 Stone Avenue	
1984	Adventure Undersea	PO Box 744	

FIGURE 7.10
An unfiltered view of the CUSTOMER *table.*

After following step 2 and setting the table's `Filtered` property to `True`, you can see in Figure 7.11 that the grid displays only those records that meet the filter criteria.



Form1

CustNo	Company	Addr1	Addr2
1351	Sight Diver	1 Neptune Lane	
2163	SCUBA Heaven	PO Box Q-8874	
2165	Shangri-La Sports Center	PO Box D-5495	
3051	San Pablo Dive Center	1701-D N Broadway	
5163	Safari Under the Sea	PO Box 7456	

FIGURE 7.11
A filtered view of the CUSTOMER *table.*

NOTE

The `OnFilterRecord` event should only be used in cases where the filter cannot be expressed in the `Filter` property. The reason for this is that it can provide significant performance benefits. On SQL databases, for example, the `TTable` component will pass the contents of the `FILTER` property in a `WHERE` clause to the database, which is generally much faster than the record-by-record search performed in `OnFilterRecord`.

Searching Datasets

Datasets provide variations on how to search through datasets. The coverage here shows only the non-SQL type searching techniques. SQL based techniques are covered in Chapter 29 on the CD copy of *Delphi 5 Developer's Guide*.

`FindFirst()` and `FindNext()`

`TDataSet` also provides methods called `FindFirst()`, `FindNext()`, `FindPrior()`, and `FindLast()` that employ filters to find records that match a particular search criteria. All these functions work on unfiltered datasets by calling that dataset's `OnFilterRecord` event handler. Based on the search criteria in the event handler, these functions will find the first, next, previous, or last match, respectively. Each of these functions accepts no parameters and returns a Boolean, which indicates whether a match was found.

Locating a Record Using the `Locate()` Method

Not only are filters useful for defining a subset view of a particular dataset, but they can also be used to search for records within a dataset based on the value of one or more fields. For this purpose, `TDataSet` provides a method called `Locate()`. Once again, because `Locate()` employs filters to do the searching, it will work irrespective of any index applied to the dataset. The `Locate()` method is defined as follows:

```
function Locate(const KeyFields: string; const KeyValues: Variant;
  Options: TLocateOptions): Boolean;
```

The first parameter, `KeyFields`, contains the name of the field(s) on which you want to search. The second parameter, `KeyValues`, holds the field value(s) you want to locate. The third and last parameter, `Options`, allows you to customize the type of search you want to perform. This parameter is of type `TLocateOptions`, which is a set type defined in the `DB` unit as follows:

```
type
  TLocateOption = (loCaseInsensitive, loPartialKey);
  TLocateOptions = set of TLocateOption;
```

If the set includes the `loCaseInsensitive` member, a not case sensitive search of the data will be performed. If the set includes the `loPartialKey` member, the values contained in `KeyValues` will match even if they're substrings of the field value.

`Locate()` will return `True` if it finds a match. For example, to search for the first occurrence of the value `1356` in the `CustNo` field of `Table1`, use the following syntax:

```
Table1.Locate('CustNo', 1356, []);
```

TIP

You should use `Locate()` whenever possible to search for records because it will always attempt to use the fastest method possible to find the item, switching indexes temporarily if necessary. This makes your code independent of indexes. Also, if you determine that you no longer need an index on a particular field or if adding one will make your program faster, you can make that change on the data without having to recode the application.

Table Key Searching

This section describes the common properties and methods of the `TTable` component and how to use them. In particular, you learn how to search for records, filter records using ranges, and create tables. This section also contains a discussion of `TTable` events.

`TTable` Record Searching

When you need to search for records in a table, VCL provides several methods to help you out. When you're working with dBASE and Paradox tables, Delphi assumes that the fields on which you search are indexed. For SQL tables, the performance of your search will suffer if you search on non-indexed fields.

Say, for example, you have a table that's keyed on field 1, which is numeric, and on field 2, which is alphanumeric. You can search for a specific record based on those two criteria in one of two ways: using the `FindKey()` technique or the `SetKey()..GotoKey()` technique.

`FindKey()`

`TTable`'s `FindKey()` method enables you to search for a record matching one or more keyed fields in one function call. `FindKey()` accepts an `array of const` (the search criteria) as a parameter and returns `True` when it's successful. For example, the following code causes the dataset to move to the record where the first field in the index has the value `123` and the second field in the index contains the string `Hello`:

```
if not Table1.FindKey([123, 'Hello']) then MessageBeep(0);
```

If a match isn't found, `FindKey()` returns `False` and the computer beeps.

`SetKey()..GotoKey()`

Calling `TTable`'s `SetKey()` method puts the table in a mode that prepares its fields to be loaded with values representing search criteria. Once the search criteria have been established, use the

GotoKey() method to do a top-down search for a matching record. The previous example can be rewritten with SetKey()..GotoKey(), as follows:

```
with Table1 do begin
  SetKey;
  Fields[0].AsInteger := 123;
  Fields[1].AsString := 'Hello';
  if not GotoKey then MessageBeep(0);
end;
```

The Closest Match

Similarly, you can use FindNearest() or the SetKey..GotoNearest methods to search for a value in the table that's the closest match to the search criteria. To search for the first record in which the value of the first indexed field is closest to (greater than or equal to) 123, use the following code:

```
Table1.FindNearest([123]);
```

Once again, FindNearest() accepts an array of const as a parameter that contains the field values for which you want to search.

To search using the longhand technique provided by SetKey()..GotoNearest(), you can use this code:

```
with Table1 do begin
  SetKey;
  Fields[0].AsInteger := 123;
  GotoNearest;
end;
```

If the search is successful and the table's KeyExclusive property is set to False, the record pointer will be on the first matching record. If KeyExclusive is True, the current record will be the one immediately following the match.

TIP

If you want to search on the indexed fields of a table, use FindKey() and FindNearest()—rather than SetKey()..GotoX()—whenever possible because you type less code and leave less room for human error.

Which Index?

All these searching methods assume that you're searching under the table's primary index. If you want to search using a secondary index, you need to set the table's `IndexName` parameter to the desired index. For instance, if your table had a secondary index on the `Company` field called `ByCompany`, the following code would enable you to search for the company `"Unisco"`:

```
with Table1 do begin
  IndexName := 'ByCompany';
  SetKey;
  FieldValues['Company'] := 'Unisco';
  GotoKey;
end;
```

NOTE

Keep in mind that some overhead is involved in switching indexes while a table is opened. You should expect a delay of a second or more when you set the `IndexName` property to a new value.

Ranges enable you to filter a table so that it contains only records with field values that fall within a certain scope you define. Ranges work similarly to key searches, and as with searches, there are several ways to apply a range to a given table—either using the `SetRange()` method or the manual `SetRangeStart()`, `SetRangeEnd()`, and `ApplyRange()` methods.

CAUTION

If you are working with dBASE or Paradox tables, ranges only work with indexed fields. If you're working with SQL data, performance will suffer greatly if you don't have an index on the ranged field.

`SetRange()`

Like `FindKey()` and `FindNearest()`, `SetRange()` enables you to perform a fairly complex action on a table with one function call. `SetRange()` accepts two `array of const` variables as parameters: The first represents the field values for the start of the range, and the second represents the field values for the end of the range. As an example, the following code filters through only those records where the value of the first field is greater than or equal to 10 but less than or equal to 15:

```
Table1.SetRange([10], [15]);
```

ApplyRange()

To use the `ApplyRange()` method of setting a range, follow these steps:

1. Call the `SetRangeStart()` method and then modify the `Fields[]` array property of the table to establish the starting value of the keyed field(s).
2. Call the `SetRangeEnd()` method and modify the `Fields[]` array property once again to establish the ending value of the keyed field(s).
3. Call `ApplyRange()` to establish the new range filter.

The preceding range example could be rewritten using this technique:

```
with Table1 do begin
  SetRangeStart;
  Fields[0].AsInteger := 10;       // range starts at 10
  SetRangeEnd;
  Fields[0].AsInteger := 15;       // range ends at 15
  ApplyRange;
end;
```

TIP

Use `SetRange()` whenever possible to filter records—your code will be less prone to error when doing so.

To remove a range filter from a table and restore the table to the state it was in before you called `ApplyRange()` or `SetRange()`, just call `TTable`'s `CancelRange()` method.

```
Table1.CancelRange;
```

Using Data Modules

Data modules enable you to keep all your database rules and relationships in one central location to be shared across projects, groups, or enterprises. Data modules are encapsulated by VCL's `TDataModule` component. Think of `TDataModule` as an invisible form on which you can drop data-access components to be used throughout a project. Creating a `TDataModule` instance is simple: Select File, New from the main menu and then select Data Module from the Object Repository.

The simple justification for using `TDataModule` over just putting data-access components on a form is that it's easier to share the same data across multiple forms and units in your project. In a more complex situation, you would have an arrangement of multiple `TTable`, `TQuery`, and/or `TStoredProc` components. You might have relationships defined between the components and perhaps rules enforced on the field level, such as minimum/maximum values or display formats. Perhaps this assortment of data-access components models the business rules of your enterprise.

After taking great pains to set up something so impressive, you wouldn't want to have to do it again for another application, would you? Of course you wouldn't. In such cases, you would want to save your data module to the Object Repository for later use. If you work in a team environment, you might even want to keep the Object Repository on a shared network drive for the use of all the developers on your team.

In the example that follows, you'll create a simple instance of a data module so that many forms have access to the same data. In the database applications shown in several of the later chapters, you'll build more complex relationships into data modules.

The Search, Range, Filter Demo

Now it's time to create a sample application to help drive home some of the key concepts that were covered in this chapter. In particular, this application will demonstrate the proper use of filters, key searches, and range filters in your applications. This project, called SRF, contains multiple forms. The main form consists mainly of a grid for browsing a table, and other forms demonstrate the different concepts mentioned earlier. Each of these forms will be explained in turn.

The Data Module

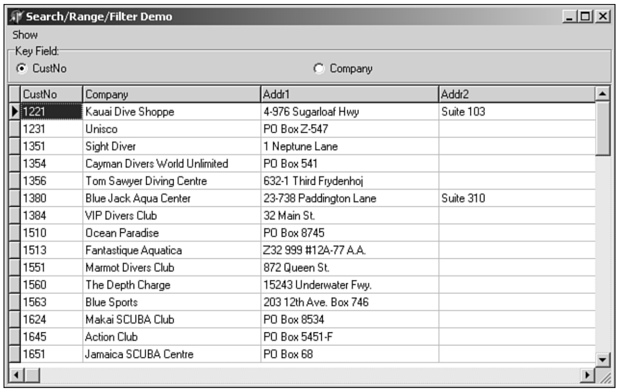
Although we're starting a bit out of order, the data module for this project will be covered first. This data module, called `DM`, contains only a `TTable` and a `TDataSource` component. The `TTable`, called `Table1`, is hooked to the `CUSTOMERS.DB` table in the `DBDEMOS` alias. The `TDataSource`, `DataSource1`, is wired to `Table1`. All the data-aware controls in this project will use `DataSource1` as their `DataSource`. `DM` is contained in a unit called `DataMod`.

The Main Form

The main form for SRF, appropriately called `MainForm`, is shown in Figure 7.12. This form is contained in a unit called `Main`. As you can see, it contains a `TDBGrid` control, `DBGrid1`, for browsing a table, and it contains a radio button that enables you to switch between different indexes on the table. `DBGrid1`, as explained earlier, is hooked to `DM.DataSource1` as its data source.

NOTE

In order for `DBGrid1` to be able to hook to `DM.DataSource1` at design time, the `DataMod` unit must be in the `uses` clause of the `Main` unit. The easiest way to do this is to bring up the `Main` unit in the Code Editor and select File, Use Unit from the main menu. You'll then be presented with a list of units in your project from which you can select `DataMod`. You must do this for each of the units from which you want to access the data contained within `DM`.



CustNo	Company	Addr1	Addr2
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103
1231	Unisco	PO Box Z-547	
1351	Sight Diver	1 Neptune Lane	
1354	Cayman Divers World Unlimited	PO Box 541	
1356	Tom Sawyer Diving Centre	632-1 Third Frydenhoj	
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310
1384	VIP Divers Club	32 Main St.	
1510	Ocean Paradise	PO Box 8745	
1513	Fantastique Aquatica	Z32 999 #12A-77 A.A.	
1551	Marmot Divers Club	872 Queen St.	
1560	The Depth Charge	15243 Underwater Fwy.	
1563	Blue Sports	203 12th Ave. Box 746	
1624	Makai SCUBA Club	PO Box 8534	
1645	Action Club	PO Box 5451-F	
1651	Jamaica SCUBA Centre	PO Box 68	

FIGURE 7.12
`MainForm` *in the SRF project.*

The radio group, called `RGKeyField`, is used to determine which of the table's two indexes is currently active. The code attached to the `OnClick` event for `RGKeyField` is shown here:

```
procedure TMainForm.RGKeyFieldClick(Sender: TObject);
begin
  case RGKeyField.ItemIndex of
    0: DM.Table1.IndexName := '';           // primary index
    1: DM.Table1.IndexName := 'ByCompany';  // secondary, by company
  end;
end;
```

`MainForm` also contains a `TMainMenu` component, `MainMenu1`, which enables you to open and close each of the other forms. The items on this menu are Key Search, Range, Filter, and Exit. The `Main` unit, in its entirety, is shown in Listing 7.5.

NOTE

In order for `DBGrid1` to be able to hook to `DM.DataSource1` at design time, the `DataMod` unit must be in the `uses` clause of the `Main` unit. The easiest way to do this is to bring up the `Main` unit in the Code Editor and select File, Use Unit from the main menu. You'll then be presented with a list of units in your project from which you can select `DataMod`. You must do this for each of the units from which you want to access the data contained within `DM`.

The radio group, called `RGKeyField`, is used to determine which of the table's two indexes is currently active. The code attached to the `OnClick` event for `RGKeyField` is shown here:

```
procedure TMainForm.RGKeyFieldClick(Sender: TObject);
begin
  case RGKeyField.ItemIndex of
    0: DM.Table1.IndexName := '';           // primary index
    1: DM.Table1.IndexName := 'ByCompany';  // secondary, by company
  end;
end;
```

MainForm also contains a TMainMenu component, MainMenu1, which enables you to open and close each of the other forms. The items on this menu are Key Search, Range, Filter, and Exit. The Main unit, in its entirety, is shown in Listing 7.5.

LISTING 7.5 Main.pas—Demonstrating Dataset Ranges

```
unit Main;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Grids, DBGrids, DB, DBTables,
  Buttons, Mask, DBCtrls, Menus, KeySrch, Rng, Fltr;

type
  TMainForm = class(TForm)
    DBGrid1: TDBGrid;
    RGKeyField: TRadioGroup;
    MainMenu1: TMainMenu;
    Forms1: TMenuItem;
    KeySearch1: TMenuItem;
    Range1: TMenuItem;
    Filter1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    procedure RGKeyFieldClick(Sender: TObject);
    procedure KeySearch1Click(Sender: TObject);
    procedure Range1Click(Sender: TObject);
    procedure Filter1Click(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

LISTING 7.5 Continued

```
var
  MainForm: TMainForm;

implementation

uses DataMod;

{$R *.DFM}

procedure TMainForm.RGKeyFieldClick(Sender: TObject);
begin
  case RGKeyField.ItemIndex of
    0: DM.Table1.IndexName := '';           // primary index
    1: DM.Table1.IndexName := 'ByCompany';  // secondary, by company
  end;
end;

procedure TMainForm.KeySearch1Click(Sender: TObject);
begin
  KeySearch1.Checked := not KeySearch1.Checked;
  KeySearchForm.Visible := KeySearch1.Checked;
end;

procedure TMainForm.Range1Click(Sender: TObject);
begin
  Range1.Checked := not Range1.Checked;
  RangeForm.Visible := Range1.Checked;
end;

procedure TMainForm.Filter1Click(Sender: TObject);
begin
  Filter1.Checked := not Filter1.Checked;
  FilterForm.Visible := Filter1.Checked;
end;

procedure TMainForm.Exit1Click(Sender: TObject);
begin
  Close;
end;

end.
```

NOTE

Pay close attention to the following line of code from the Rng unit:

```
DM.Table1.SetRange([StartEdit.Text], [EndEdit.Text]);
```

You might find it strange that although the keyed field can be of either a Numeric type or Text type, you're always passing strings to the SetRange() method. Delphi allows this because SetRange(), FindKey(), and FindNearest() will perform the conversion from String to Integer, and vice versa, automatically.

What this means to you is that you shouldn't bother calling IntToStr() or StrToInt() in these situations—it will be taken care of for you.

The Key Search Form

KeySearchForm, contained in the KeySrch unit, provides a means for the user of the application to search for a particular key value in the table. The form enables the user to search for a value in one of two ways. First, when the Normal radio button is selected, the user can search by typing text into the Search For edit control and pressing the Exact or Nearest button to find an exact match or closest match in the table. Second, when the Incremental radio button is selected, the user can perform an incremental search on the table every time he or she changes the text in the Search For edit control. The code for the KeySrch unit is shown in Listing 7.6.

LISTING 7.6 The Source Code for KeySrch.PAS

```
unit KeySrch;

interface

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

type
  TKeySearchForm = class(TForm)
    Panel1: TPanel;
    Label3: TLabel;
    SearchEdit: TEdit;
    RBNormal: TRadioButton;
    Incremental: TRadioButton;
    Label6: TLabel;
    ExactButton: TButton;
    NearestButton: TButton;
    procedure ExactButtonClick(Sender: TObject);
```

LISTING 7.6 Continued

```
    procedure NearestButtonClick(Sender: TObject);
    procedure RBNormalClick(Sender: TObject);
    procedure IncrementalClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    procedure NewSearch(Sender: TObject);
  end;

var
  KeySearchForm: TKeySearchForm;

implementation

uses DataMod, Main;

{$R *.DFM}

procedure TKeySearchForm.ExactButtonClick(Sender: TObject);
begin
  { Try to find record where key field matches SearchEdit's Text value. }
  { Notice that Delphi handles the type conversion from the string      }
  { edit control to the numeric key field value.                       }
  if not DM.Table1.FindKey([SearchEdit.Text]) then
    MessageDlg(Format('Match for "%s" not found.', [SearchEdit.Text]),
               mtInformation, [mbOk], 0);
end;

procedure TKeySearchForm.NearestButtonClick(Sender: TObject);
begin
  { Find closest match to SearchEdit's Text value. Note again the }
  { implicit type conversion.                                     }
  DM.Table1.FindNearest([SearchEdit.Text]);
end;

procedure TKeySearchForm.NewSearch(Sender: TObject);
{ This is the method which is wired to the SearchEdit's OnChange }
{ event whenever the Incremental radio is selected. }
begin
  DM.Table1.FindNearest([SearchEdit.Text]); // search for text
end;

procedure TKeySearchForm.RBNormalClick(Sender: TObject);
begin
```

LISTING 7.6 Continued

```
  ExactButton.Enabled := True;   // enable search buttons
  NearestButton.Enabled := True;
  SearchEdit.OnChange := Nil;    // unhook the OnChange event
end;

procedure TKeySearchForm.IncrementalClick(Sender: TObject);
begin
  ExactButton.Enabled := False;      // disable search buttons
  NearestButton.Enabled := False;
  SearchEdit.OnChange := NewSearch;  // hook the OnChange event
  NewSearch(Sender);                 // search current text
end;

procedure TKeySearchForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caHide;
  MainForm.KeySearch1.Checked := False;
end;

end.
```

The code for the KeySrch unit should be fairly straightforward to you. You might notice that, once again, we can safely pass text strings to the FindKey() and FindNearest() methods with the knowledge that they will do the right thing with regard to type conversion. You might also appreciate the small trick that's employed to switch to and from incremental searching on-the-fly. This is accomplished by either assigning a method to or assigning Nil to the OnChange event of the SearchEdit edit control. When assigned a handler method, the OnChange event will fire whenever the text in the control is modified. By calling FindNearest() inside that handler, an incremental search can be performed as the user types.

The Filter Form

The purpose of FilterForm, found in the Fltr unit, is two-fold. First, it enables the user to filter the view of the table to a set where the value of the State field matches that of the current record. Second, this form enables the user to search for a record where the value of any field in the table is equal to some value she has specified.

The record-filtering functionality actually involves very little code. First, the state of the check box labeled Filter on This State (called cbFiltered) determines the setting of

DM.Table1's Filtered property. This is accomplished with the following line of code attached to cbFiltered.OnClick:

```
DM.Table1.Filtered := cbFiltered.Checked;
```

When DM.Table1.Filtered is True, Table1 filters records using the following OnFilterRecord method, which is actually located in the DataMod unit:

```
procedure TDM.Table1FilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
  { Accept record as a part of the filter if the value of the State }
  { field is the same as that of DBEdit1.Text.                     }
  Accept := Table1State.Value = FilterForm.DBEdit1.Text;
end;
```

To perform the filter-based search, the Locate() method of TTable is employed:

```
DM.Table1.Locate(CBField.Text, EValue.Text, LO);
```

The field name is taken from a combo box called CBField. The contents of this combo box are generated in the OnCreate event of this form using the following code to iterate through the fields of Table1:

```
procedure TFilterForm.FormCreate(Sender: TObject);
var
  i: integer;
begin
  with DM.Table1 do begin
    for i := 0 to FieldCount - 1 do
      CBField.Items.Add(Fields[i].FieldName);
  end;
end;
```

TIP

The preceding code will only work when DM is created prior to this form. Otherwise, any attempts to access DM before it's created will probably result in an Access Violation error. To make sure that the data module, DM, is created prior to any of the child forms, we manually adjusted the creation order of the forms in the Autocreate Forms list on the Forms page of the Project Options dialog (found under Options, Project on the main menu).

The main form must, of course, be the first one created, but other than that, this little trick ensures that the data module gets created prior to any other form in the application.

The complete code for the `Fltr` unit is shown in Listing 7.7.

LISTING 7.7 The Source Code for `Fltr.pas`

```
unit Fltr;

interface

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

type
  TFilterForm = class(TForm)
    Panel1: TPanel;
    Label4: TLabel;
    DBEdit1: TDBEdit;
    cbFiltered: TCheckBox;
    Label5: TLabel;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    SpeedButton3: TSpeedButton;
    SpeedButton4: TSpeedButton;
    Panel2: TPanel;
    EValue: TEdit;
    LocateBtn: TButton;
    Label1: TLabel;
    Label2: TLabel;
    CBField: TComboBox;
    MatchGB: TGroupBox;
    RBExact: TRadioButton;
    RBClosest: TRadioButton;
    CBCaseSens: TCheckBox;
    procedure cbFilteredClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure LocateBtnClick(Sender: TObject);
    procedure SpeedButton1Click(Sender: TObject);
    procedure SpeedButton2Click(Sender: TObject);
    procedure SpeedButton3Click(Sender: TObject);
    procedure SpeedButton4Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  end;

var
  FilterForm: TFilterForm;
```

LISTING 7.7 Continued

```
implementation

uses DB, DataMod, Main;

{$R *.DFM}

procedure TFilterForm.cbFilteredClick(Sender: TObject);
begin
  { Filter table if checkbox is checked }
  DM.Table1.Filtered := cbFiltered.Checked;
end;

procedure TFilterForm.FormCreate(Sender: TObject);
var
  i: integer;
begin
  with DM.Table1 do begin
    for i := 0 to FieldCount - 1 do
      CBField.Items.Add(Fields[i].FieldName);
  end;
end;

procedure TFilterForm.LocateBtnClick(Sender: TObject);
var
  LO: TLocateOptions;
begin
  LO := [];
  if not CBCaseSens.Checked then Include(LO, loCaseInsensitive);
  if RBClosest.Checked then Include(LO, loPartialKey);
  if not DM.Table1.Locate(CBField.Text, EValue.Text, LO) then
    MessageDlg('Unable to locate match', mtInformation, [mbOk], 0);
end;

procedure TFilterForm.SpeedButton1Click(Sender: TObject);
begin
  DM.Table1.FindFirst;
end;

procedure TFilterForm.SpeedButton2Click(Sender: TObject);
begin
  DM.Table1.FindNext;
end;

procedure TFilterForm.SpeedButton3Click(Sender: TObject);
```

LISTING 7.7 Continued

```
begin
  DM.Table1.FindPrior;
end;

procedure TFilterForm.SpeedButton4Click(Sender: TObject);
begin
  DM.Table1.FindLast;
end;

procedure TFilterForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caHide;
  MainForm.Filter1.Checked := False;
end;

end.
```

Bookmarks

Bookmarks enable you to save your place in a dataset so that you can come back to the same spot at a later time. Bookmarks are very easy to use in Delphi because you only have one property to remember.

Delphi represents a bookmark as type `TBookmarkStr`. `TTable` has a property of this type called `Bookmark`. When you read from this property, you obtain a bookmark, and when you write to this property, you go to a bookmark. When you find a particularly interesting place in a dataset that you'd like to be able to get back to easily, here's the syntax to use:

```
var
  BM: TBookmarkStr;
begin
  BM := Table1.Bookmark;
```

When you want to return to the place in the dataset you marked, just do the reverse—set the `Bookmark` property to the value you obtained earlier by reading the `Bookmark` property:

```
Table1.Bookmark := BM;
```

`TBookmarkStr` is defined as an `AnsiString`, so memory is automatically managed for bookmarks (you never have to free them). If you'd like to clear an existing bookmark, just set it to an empty string:

```
BM := '';
```

Note that TBookmarkStr is an AnsiString for storage convenience. You should consider it an opaque data type and not depend on the implementation because the bookmark data is completely determined by BDE and the underlying data layers.

NOTE

Although 32-bit Delphi still supports GetBookmark(), GotoBookmark(), and FreeBookmark() from Delphi 1.0, because the 32-bit Delphi technique is a bit cleaner and less prone to error, you should use this newer technique unless you have to maintain compatibility with 16-bit projects.

You'll find an example of using bookmarks with an ADO dataset on the CD in the \Bookmark subdirectory for this chapter.

Summary

After reading this chapter, you should be ready for just about any type of database programming with Delphi. You learned the ins and outs of Delphi's TDataSet component, which is the ancestor of the different types of datasets. You also learned techniques for manipulating datasets, how to manage fields, and how to work with text tables.

In the following chapters, you will learn about dbExpress, Delphi's lightweight database development technology and about dbGo, Delphi's connectivity to ADO data in greater depth.