Chapter 12
ADO.NET Programming

This chapter introduces ADO.NET, a set of Compact Framework classes used for database programming in managed code. Like its counterpart on the desktop, the ADO.NET classes in Compact Framework provide a convenient, flexible model for operating on memory-resident databases. Three options are presented for the persistent storage of ADO.NET data sets: XML files, Microsoft SQL Server CE databases on a Windows CE device, and Microsoft SQL Server databases (version 7.0 or 2000) on a desktop or server version of SQL Server. [Comment 12cs.1]

Introducing ADO.NET 4
- A Tiered Approach 5
- The ADO.NET Classes 5
  - Namespace References and assembly references 7
  - Functionality – Super Sets and Sub Sets 8
- ADO.NET Error Handling 8
- Utility Routines 9

Working with Data Sets 10
- Creating and Accessing Datasets, Data Tables, and Data Views 11
  - Understanding Data Tables. 12
  - Working with DataRows 13
  - Introducing DataViews 14
- Data Binding 14
  - Binding to Multi-item controls 16
  - Binding to Single-item controls 17
  - Designating the row to be displayed 18
  - Positioning the controls to a row 21
  - Updating the bound data table 21
- Reading and Writing a Dataset as XML 22

Microsoft SQL Server CE 24
- SQL Server CE Files 24
- SQL Server CE Syntax 25
- SQL Server CE Query Analyzer 28
- Creating a SQL Server CE Database 30
- Populating a SQL Server CE Database 31
  - The Connection and Command Classes 31
- Retrieving and Displaying Data 33
  - The Data Reader Class 33
- Updating a SQL Server CE Database 41
- The SqlCeDataAdapter class 42
  - Using the Data Adapter to Retrieve Data 42
  - Using the Data Adapter to Update a Database 43
- Querying Schema Information 47
- ADO.NET Programing for SQL Server CE – Conclusion 51

Microsoft SQL Server 51
- Connecting to SQL Server 52
- Creating Command Objects 59
- Using SQL Server Stored Procedures 59

Copyright © 2003 Paul Yao & David Durant

Using Stored Procedures with DataSets 65
Datasets and Concurrency 68
ADO.NET Programing for SQL Server – Conclusion 68
Web Services 69
XML, XSD and SOAP 69
Web Classes 70
A Web Service Application 71
A Web Services Client Application 79
Web Services Summary 82
Summary 82

A Compact Framework program that manages a significant volume of data is probably going to use the ADO.NET classes to manage that data. ADO.NET provides managed code support for memory-resident data. The various ADO.NET classes support concepts familiar to database programmers, like rows and columns, that are the fundamental elements of a table. Some ADO.NET classes provide the ability to create data tables, and programmatically add columns. Other classes provide the ability to add, modify and delete rows, providing a location for a program's in-memory data. Central to the design of these classes is the independence of the classes from any particular database product or storage format. This means that an ADO.NET table might be constructed with data from two or more data sources. An ADO.NET table can also be dynamically created, manipulated in memory, and then discarded without writing to any persistent storage. We expect, however, that most programs that use the ADO.NET classes will store the results of their work in a more persistent form. [Comment 12cs.2]

Table 12-1 summarizes all of the persistent data storage options available to Compact Framework programmers. As indicated in this table, ADO.NET supports three different storage types. Two storage types reside in the file system of a Windows CE-powered device: SQL Server CE databases and XML files. These are appropriate for a mobile device, such as a Pocket PC or SmartPhone, which needs to run in a self-contained, disconnected mode. Using the SQL Server 2000 data provider, a Compact Framework program can directly access a third type of storage on a connected server, a SQL Server 2000 database. This requires a live network connection between the Windows CE client system and a server running SQL Server 2000. [Comment 12cs.3]

Supported ADO.NET Data Providers

As of this writing, we know of 3 ADO.NET data providers which are available for the Compact Framework: Microsoft SQL Server CE, Microsoft SQL Server (version 7.0 and 2000), and one for Pocket Access (available from In The Hand, http://www.inthehand.com). [Comment 12cs.4]

The following data providers are available for the desktop .NET Framework, but as of this writing are not available for Compact Framework programs: OLE DB, ODBC, Oracle, MySQL, DB2, SyBase SQL. [Comment 12cs.5]

There are two types of ADO.NET classes. One set of classes provides database programming support, but are not tied to a specific database file format. Instead, the classes operate on in-memory objects which model relational data tables. [Comment 12cs.6]

A second set of classes, the *data provider classes*, provide the connection between in-memory objects and physical databases that reside in permanent storage. Alternatively,

Copyright © 2003 Paul Yao & David Durant

you can choose not to use a formal database, and instead allow the ADO.NET classes to store your data in XML files in the Window CE file system. [Comment 12cs.7]

Table 12-1 Persistent storage options available on Windows CE-powered devices. [Comment 12cs.8]

Type of Storage	Managed API	Native API
SQL Server CE Database	ADO.NET with SQL Server CE Data Provider	ADOCE OLEDBCE
SQL Server 2000 Database	ADO.NET with SQL Server 2000 Data Provider	ADOCE OLEDBCE
XML Files	ADO.NET	DOM provided by msxml.dll
Raw Data Files	`System.IO` classes (see chapter 11)	Win32 API file access functions
CE Property Databases	n/a	CE-specific Win32 API functions including `CeCreateDataBaseEx`, etc.
System registry	n/a	Win32 API functions required (see chapter 11)

We expect that many databases on Windows CE-powered devices will need to be synchronized with databases on desktop and server systems. Table 12-2 summarizes the available options for synchronizing data with desktop and server systems. Two of these methods are appropriate for synchronizing a device-based SQL Server CE database with a SQL Server 2000 database: Merge Replication and Remote Data Access (RDA). We discuss these in detail in chapter 13, *Synchronizing Mobile Data*. [Comment 12cs.9]

Table 12-2 Data synchronization options on Windows CE-powered devices. [Comment 12cs.10]

Synchronization Method	Description
Merge Replication	Publish and subscribe model for moving data between a SQL Server CE database and a SQL Server 2000 database. Described in detail in chapter 13, *Synchronizing Mobile Data*.
Remote Data Access (RDA)	An SQL-driven mechanism for moving data between a SQL Server CE database, and a SQL Server 6.5, SQL Server 7, or SQL Server 2000 database. Described in detail in chapter 13, *Synchronizing Mobile Data*.
XML Web services	General purpose remote procedure call (RPC) support and document delivery mechanism supported by .NET Framework and .NET Compact Framework. Built on SOAP, an XML-based industry standard remoting protocol. Described in detail in chapter 14, *Building XML Web Service Clients*.
ActiveSync	General purpose mechanism for synchronizing desktop and smart-device data. Relies on Service Providers, which currently must be implemented in native DLLs which export a specific set of COM interfaces. Both desktop and device-side service providers are supported. SQL Server CE does not use ActiveSync for any of its synchronization. However, ActiveSync does support synchronizing a Pocket Access (*.cdb) file with a desktop Microsoft Access (*.mdb) file. Pocket Access is a discontinued product, but third-party support[1] exists for the Pocket Access file format (in the form of an ADO.NET-compatible data provider), and also third-party

[1] For details on the ADOCE .NET Wrapper available from In The Hand, visit their web site: http://www.inthehand.com.

Copyright © 2003 Paul Yao & David Durant

Synchronization Method	Description
	development tools.
Remote API (RAPI)	Part of the ActiveSync support libraries. These provide a general purpose mechanism for accessing the object store and installable file systems of a smart device from an ActiveSync-connected desktop system. This mechanism can be used to install or backup anything on a CE device. However there is no specific ADO.NET support or support for SQL databases. RAPI is described in detail in chapter 15, *The Remote API (RAPI)*.

Introducing ADO.NET

ADO.NET is a set of classes for accessing and managing in-memory data. These classes provide the primary means for bringing data from a database into a managed code program. On the desktop, ADO.NET support is provided for accessing data in the following types of databases: SQL Server, Sybase SQL, MySQL, Oracle, DB2, and those with OLEDB and ODBC providers. On Windows CE, support from Microsoft is limited to two types of databases: SQL Server CE and SQL Server (version 7.0 and later). The ADO.NET classes provide in-memory data support in a variety of environments for a wide range of application types, including desktop Windows Forms applications, Web Forms applications, and also in XML Web services and in mobile device applications. [Comment 12cs.11]

The ADO.NET classes were designed to work with data that is drawn from a variety of sources, or in the terminology of ADO.NET, from a variety of *data providers*. Whatever the source of the data, the ADO.NET classes allow data to be managed and presented in a standard fashion. To make this happen, some of the ADO.NET classes are provider-specific, meaning capable of working with only one data provider, such as the `SqlCeConnection` class or the `ODBCConnection` classe. [Comment 12cs.12]

Others classes, such as the `DataSet`, `DataTable` and `DataView` are provider-*independent*. The data contained within them has been converted to the ADO.NET data types; and the structure of the data supported by these classes is independent of the database engine that provides the data. This creates the intriguing possibility that an ADO.NET data set might contain data tables with data drawn from multiple sources; for example, a data set could contain some tables that have been read from a Microsoft SQL Server CE database and other tables that are read from a Microsoft SQL Server 2000 database. [Comment 12cs.13]

Limitations of ADO.NET in the Compact Framework

Strongly-typed data sets are not supported in the current version of the Compact Framework. Strongly-typed data sets are bound to the schema of a database, which means we can use database field names instead of column numbers. [Comment 12cs.14]

Early-binding makes it easier to write code, makes it easier to read code, and so makes it easier to maintain the code. There are run-time benefits as well, because early-bound, strongly-typed data sets allow the compiler to create code that executes more efficiently; which it can do because the data set schema is known at compile-time. This capability may become

Copyright © 2003 Paul Yao & David Durant

available in future versions of Compact Framework, but it is not available in the current version. [Comment 12cs.15]

The ADO.NET classes treat data as provider-agnostic, meaning there is no easy way – and, hopefully, no need to –determine which data came from what data source. You can even build and populate data sets and data tables from scratch without reference to a particular database. As part of its provider-independent design, the ADO.NET classes can read and write their data to XML files. **[Comment 12cs.16]**

A Tiered Approach

Because of the capabilities of the ADO.NET classes, you can use either a two-tier or three-tier approach to data management. [Comment 12cs.17]

In a two-tier approach, you submit SQL queries to the database and retrieve the data into program variables and control properties. You then build and submit SQL statements to make the user requested changes to the data and submit them to the database. See figure 12-1. This is often referred to as the *connected* approach, as the application must be connected to the database while transferring data. Do not confuse this meaning of the term *connected* with being connected to a network. Throughout this chapter, when we use the term *connected* we mean connected to the SQL Server or SQL Server CE database. [Comment 12cs.18]

Figure 12-1 – A two tiered approach
[Hand drawing – to be delivered. David.]

In a three-tier approach, you bring the data into a memory-resident dataset, and then bind some or all data to controls on your form. Changes made by the user to the data in the form are automatically propagated back to the dataset; class methods then propagate changes from a dataset to the database. See figure 12-2. This is often referred to as the *disconnected* approach, meaning that the application is not connected to the database while it is accessing the dataset data. [Comment 12cs.19]

Figure 12-2 – A three tiered approach
[Hand drawing – to be delivered. David.]

The ADO.NET Classes

Table 12-3 presents an overview of the classes which can be used to access SQL Server CE databases. We present most of these classes in detail later in this chapter. They fall into four general categories: [Comment 12cs.20]

1. File system classes [Comment 12cs.21]
2. Provider specific ADO.NET classes [Comment 12cs.22]
3. Provider independent ADO.NET classes [Comment 12cs.23]
4. Data binding classes [Comment 12cs.24]

We discuss the file system classes in detail in chapter 11, *Storage*. And we cover controls used for data binding in chapter 8, *Data Binding*. [Comment 12cs.25]

Copyright © 2003 Paul Yao & David Durant

Table 12-3 – Classes for accessing, maintaining and displaying data [Comment 12cs.26]

Category	Class	Purpose / Comments
Filesystem[2]	`Directory`	Create, delete, move, and detect directories.
	`File`	Create, delete, move, copy, and detect files.
Provider-Specific classes for SQL Server CE[3] and SQL Server[4]	`SqlCeEngine`	Create a SQL Server CE database. Compact a SQL Server CE database.
	`SqlCeConnection`, `SqlConnection`	Open a connection to a database. Only one open connection per SQL Server CE database at a time.
	`SqlCeCommand`, `SqlCommand`	Submit a SQL statement to the database. Contains the connection object.
	`SqlCeDataReader`, `SqlDataReader`	Retrieve the rows returned by a command. Can only be created by the command's `ExecuteReader` method. Only one open data reader per SQL Server connection at a time.
	`SqlCeDataAdapter`, `SqlDataAdapter`	Moves data between a database and a `DataTable`. Contains up to four command objects.
	`SqlCeCommandBuilder`, `SqlCommandBuilder`	Reverse engineers a simple `SELECT` statement to create the `INSERT`, `UPDATE`, and `DELETE` statements used by an adapter to move modified data from a `DataTable` back to the database.
Provider-independent ADO.NET Classes[5]	`DataSet`	Memory resident database. Contains the provider-independent classes listed below.
	`DataTable`	Memory resident table.
	`DataRow`	A row within a `DataTable`. Contains the data. Has methods to access other rows in other tables based on `DataRelations`, such as `GetChildRows`.
	`DataView`	Sorts and filters the rows of a `DataTable`. Primarily for use with data binding.
	`DataRelation`	Specification of parent-child relationship between two `DataTables`.
	`UniqueConstraint`	Constraint on a `DataTable`.
	`ForeignKeyConstraint`	Constraint relating two `DataTables`.
Data Binding Support[6]	Multi-item controls – `DataGrid`, `ListBox`, `ComboBox`	When data bound, display one or all columns of a `DataTable` or `DataView`. Are non-updateable.
	Single-item controls – `TextBox`, `Label`, `RadioButton`, `CheckBox`, `Button`, etc.	When data bound, display one column of one row of a `DataTable` or `DataView`. Automatically propagate data changes

[2] In the `System.IO` namespace.
[3] In the `System.Data.SqlServerCe` namespace.
[4] In the `System.Data.SqlClient` namespace.
[5] In the `System.Data` namespace.
[6] In the `System.Windows.Forms` namespace.

Copyright © 2003 Paul Yao & David Durant

	back to the bound `DataTable`. A `CurrencyManager` (see below) manages row positioning and data updating.
`DataBindings` Collection	A property of a single-item control. Used to specify data binding between a `DataTable` or `DataView` and the single-item control.
`BindingContext`	A property of a form. Contains a collection of `CurrencyManager` objects. Used by the form to manage all data binding of all contained controls. Created by the .NET runtime.
`CurrencyManager`	Manages the automatic updating of bound `DataTables`. Can be used to override the automatic updating of bound `DataTables` by canceling the update or forcing the immediate execution of the update. Contains a read-write indexer property that identifies the "current" row of the bound `DataTable` or `DataView`. Created by the .NET runtime. Can be obtained by calling the form's `BindingContext`(DataTable/DataView name) indexer property.

Note from table 12-3 that SQL Server permits multiple open connections but only one open data reader per connection; while SQL Server CE is just the opposite, permitting only one open connection but allowing multiple open readers on that connection. [Comment 12cs.27]

Also note that Compact Framework multi-item controls do permit users to enter data. As mentioned in chapter 8, *Data Binding*, the data grid control is display only, and the combo box control does not permit direct entry of a value, allowing the user to select from the list only. [Comment 12cs.28]

Namespace References and assembly references

The SQL Server CE specific ADO.NET classes are located in the `System.Data.SqlServerCe` namespace. There is a similar set of classes in the `System.Data.SqlClient` namespace for use when accessing SQL Server. Other namespaces hold classes for accessing data via Oracle, OLEDB, and ODBC on the desktop, but at the present these data providers are not supported in the Compact Framework. No doubt, additional providers will become available in the future. [Comment 12cs.29]

Because the classes are located in specific namespaces, you must set some references in your project before writing any code. You need to set a reference to the `System.Data.SqlServerCe` namespace because it holds the classes that you must use to access a SQL Server CE database; and you need to set a reference to the `System.Data.SqlClient when accessing SQL Server`. You also need to set a reference to the `System.Data.Common` namespace because it contains private classes used by the data adapter class. And, finally, you need a reference to `System.Data` because it contains the provider-independent classes listed in table 12-3. Among the namespace references and the assembly reference that you may need are the following: [Comment 12cs.30]

Copyright © 2003 Paul Yao & David Durant

Namespace	Located in Assembly
System.Data	System.Data.dll
System.Data.Common	System.Data.Common.dll
System.Data.SqlServerCe	System.Data.SqlServerCe.dll
System.Data.SqlClient	System.Data.SqlClient.dll

Functionality – Super Sets and Sub Sets

The minimum set of provider-specific classes, methods, properties and events that must be supported by a provider has been specified by Microsoft and is the same for all providers. As you move between SQL Server CE and SQL Server 2000, you use the same properties and methods to access the data regardless of provider. For example, we're about to use the `SqlCeDataReader` class, which has a `NextResult` method. The `SqlDataReader` class and the `OleDbDataReader` class also have this same method, as does the `AdoceDataReader` class in the Pocket Access data provider from In The Hand. They have to, for the standard requires it. [Comment 12cs.31]

The advantage of this standards-driven approach is that the same code can be used to access data from a variety of providers, be it SQL Server, Pocket Access, or SQL Server CE; yet the underlying classes can be optimized to take advantage of the capabilities of the individual data provider. The disadvantage of this approach, especially for us SQL Server CE programmers, is that functionality which is inappropriate for SQL Server CE exists in our classes simply because it may be appropriate for SQL Server or Oracle or some other data provider, and therefore has been included as part of the standard. [Comment 12cs.32]

Continuing on our example above, the `SqlCeDataReader` class' `NextResult` method provides for the processing of commands that contain multiple `SELECT` statements. But the SQL Server CE database does not support multi-`SELECT` commands. For this reason, the `NextResult` method is not supported and so is completely inappropriate when accessing a SQL Server CE database. [Comment 12cs.33]

Thus, learning the object model in the `System.Data.SqlServerCe` namespace requires more than just learning the properties and methods, it also involves learning which of those properties and methods are appropriate for SQL Server CE (or whatever ADO.NET data provider you choose to use). One of the goals of these two chapters is to focus on the classes, methods and properties that are important for some typical SQL Server CE processing scenario. [Comment 12cs.34]

Just as the SQL Server CE SQL syntax is a subset of that found in SQL Server, ADO.NET for Windows CE is also a subset of the desktop version. For instance, only three of the eight `DataSet.ReadXML` overloads are implemented. In general, however, ADO.NET for Windows CE has most of the functionality that the desktop has; and you should not feel constrained by it. [Comment 12cs.35]

ADO.NET Error Handling

Before diving into SQL Server CE programming in general, we take a moment to examine error handling. SQL Server CE error handling uses the standard `try-catch-finally` technique used in any .NET application. If an error occurs, a `SqlCeException` is thrown. There is only one `SqlCeClient` exception class, `SqlCeException`, not one derived class for each possible error or category of error. Since SQL Server CE is written in unmanaged (native) code, it is written for the world of error numbers and

Copyright © 2003 Paul Yao & David Durant

HResults, not exceptions. Thus each error represented by a SqlCeException contains its own error information. [Comment 12cs.36]

The SqlCeException class has an Errors collection property, consisting of one or more SqlCeErrors, each containing the following properties: Source, Message, NativeError and HResult; plus three string parameters and three numeric parameters. The first four of these properties are also properties of the SqlCeException class. In the SqlCeException class, Source, NativeError and HResult are equal to those of SqlCeException.Errors[0], while Message is equal to String.Empty. The SqlCeException class also has an InnerException property, which is often empty. So, your best plan when handling a SQL Server CE exception is to examine its Errors collection. For instance, if a SELECT statement contains a misspelled column name, you receive the following information via the SqlCeException.Errors[0] object: [Comment 12cs.37]

Source – "Microsoft SQL Server 2000 Windows CE Edition".
Message – "The column name is not valid".
NativeError – 25503.
HResult – -214721900.
ErrorParameters[0]– Empty.
ErrorParameters[1] – The name of the misspelled column
ErrorParameters[2] – Empty.
NumericErrorParameters[0]– 0.
NumericErrorParameters[1] – 0.
NumericErrorParameters[2] - 0

You should always use error handling. If nothing else, error handling allows your application to present the most meaningful message possible to the user and to exit gracefully, as shown in the code below. [Comment 12cs.38]

```
try
{
   dbEngine.CreateDatabase();
}
catch( SqlCeException exSQL )
{
   MessageBox.Show("Unable to create database at " +
                   strFile +
                   ". Reason:  " +
                   exSQL.Errors[0].Message );
}
```

We mention error handling because it is a necessary part of production code. Without it, your code is brittle to the touch of all but the most casual user. With it, your code becomes robust enough to withstand the most brutal attack. That said, you may notice that our sample code has only a minimal amount of error handling (and sometimes none at all). We do this to keep the focus on the subject at hand, and to help make the samples easier to understand. [Comment 12cs.39]

Utility Routines

In any programming environment it is advantageous to have a set of utility routines for performing generic tasks. This is certainly true in the ADO.NET environment where data in repetitively moved from one tier to another or converted from one format to

Copyright © 2003 Paul Yao & David Durant

another. For instance, several ADO.NET methods return an array of data rows, arrays that you might like to convert into a data table so that you could bind it to a control. This convert-from-row-array-to-data-table task is a good candidate for being handled by a generic routine. We have written some generic routines for use in this chapter and placed them in the `UtilData` class located in the `UtilData` directory of this chapter. [Comment 12cs.40]

Working with Data Sets

The ADO.NET `DataSet` class lies at the center of the three tiered approach and at the heart of this chapter. Understanding the data set and its contained objects is a key to successful data management in managed code. To recap what has already been said about data sets (including table 12-3): [Comment 12cs.41]

1. A `DataSet` object is a memory-resident database. [Comment 12cs.42]
2. A `DataSet` object can contain multiple `DataTable` objects. [Comment 12cs.43]
3. `DataTable` objects can have constraints defined for them, such as primary key, foreign key, and unique. [Comment 12cs.44]
4. Parent/child relationships can be specified between the data tables of a `DataSet`. [Comment 12cs.45]
5. A `DataTable` object contains an instance of a `DataView`, which sorts and filters the rows of the data table. [Comment 12cs.46]

[Hand drawing – to be delivered. David.]

Remember that the `DataSet` class and associated classes are located in the `System.Data` namespace, and therefore are not provider specific. That is, a single dataset object can be used to hold the data regardless of the provider from which the data was obtained and regardless of the environment in which it is being used. The `DataSet` class is used today in web services, web applications, and Windows applications, as well as mobile applications. [Comment 12cs.47]

As you might expect, some of the dataset's capabilities are more applicable to the non-mobile environments. For example, a web service might, for performance reasons, need to pre-gather data from various data sources and stage it in memory to handle a large number of requests from a wide variety of users. The `DataSet` class is a great help in providing this capability. But our mobile applications have a small amount of available memory and are only serving one user. The `DataSet` class is the cornerstone class of ADO.NET, but it is undoubtedly least beneficial in the mobile application environment. Having said this, let us now state that there are good reasons for using the `DataSet` class in mobile applications, two of which we demonstrate here; data binding and automatic updating. [Comment 12cs.48]

Data Set Limitations

Datasets do not have an ad-hoc query capability. Instead, the data in a data set is accessed programmatically, not through a `SELECT` statement. Properties and methods of the data objects are used to access the data. Thus you access all rows of a data table by iterating through its `Rows`

Copyright © 2003 Paul Yao & David Durant

collection with a `For Each` statement; you access all the rows of a data table that meet a certain criteria through the table's `Select` method; and you access the rows of a relationship by using the parent row's `GetChildRows` method. [Comment 12cs.49]

Creating and Accessing Datasets, Data Tables, and Data Views

Creating a dataset is easy, you simply "new" it, with the option of specifying a name, as follows: [Comment 12cs.50]

```
dsetDB = new DataSet("AnyName");
```

Within a dataset, a data table can be created and populated from scratch, as shown in listing 12-1.

Listing 12-1 – Building a DataTable from Scratch [Comment 12cs.51]

```
private void CreateTable() {
      //  Create empty table.
      DataTable  dtabCustomers = new DataTable("Customers");

      //  Create three columns.
      DataColumn  dcolID = new DataColumn("ID");
      dcolID.DataType = typeof(int);
      dcolID.AutoIncrement = true;

      DataColumn  dcolName = new DataColumn("Name");
      dcolName.DataType = typeof(string );

      DataColumn  dcolAddress = new DataColumn("Address");
      dcolAddress.DataType = typeof(string );

      //  Add columns to table.
      dtabCustomers.Columns.Add(dcolID);
      dtabCustomers.Columns.Add(dcolName);
      dtabCustomers.Columns.Add(dcolAddress);

      //  Add a primary key constraint.
      dtabCustomers.Constraints.Add("PKCust", dcolID, true);

      //  Add two rows to the table
      DataRow drowCust;
      drowCust = dtabCustomers.NewRow();
      drowCust["ID"] = 1;
      drowCust["Name"] = "Amalgamated United";
      drowCust["Address"] = "PO Box 123, 98765";
      dtabCustomers.Rows.Add(drowCust);
      drowCust = dtabCustomers.NewRow();
      drowCust["ID"] = 2;
      drowCust["Name"] = "United Amalgamated";
      drowCust["Address"] = "PO Box 987, 12345";
      dtabCustomers.Rows.Add(drowCust);
```


Copyright © 2003 Paul Yao & David Durant

```
        }
    }
```

However, the most common way to create and populate a data table is to execute a `SELECT` statement against a database and place the results into the data table. These `SELECT` statements can be as simple as "`SELECT * FROM Products`", or as complex as a multi-table join with an aggregate function and grouping. Different data tables in the same dataset can be drawn from different databases. We cover the details of moving data between the dataset and a data base in the *SQL Server CE* and *SQL Server* sections of this chapter. For now, we are only interested in accessing the data as it resides in the dataset and in the movement of data between the dataset and the user. [Comment 12cs.52]

Understanding Data Tables.

A data table consists of a collection of columns and a collection of rows. `DataRow` objects are a collection of `Items`, which are the fields of the row. To access a value in a data table you must specify the row number and column identifier, either column name or number, as shown in the code below, which sets the `CategoryName` column of the second row in the `dtabCategories` data table to "Muzzy". [Comment 12cs.53]

```
dtabCategories.Rows[1]["CategoryName"] = "Muzzy";
```

You can also use the data table's `Select` method to access rows by value. The following code returns an array containing one row, the row whose `CategoryName` is "Widgets". [Comment 12cs.54]

```
dsetDB.Tables["Categories"].Select("CategoryName = 'Widgets'");
```

To save the reference to that one row, you would code:

```
drowX = dsetDB.Tables["Categories"].Select(
        "CategoryName = 'Widgets'")[0];
```

(Remember arrays are zero based; the first row of the above array is row 0.)

We often think of a data table as a two dimensional object, but we should think of it as a three dimensional object. Each data element, that is each field, in a data table can have up to four values, referred to as "versions"; `Current`, `Original`, `Proposed` and `Default`. [Comment 12cs.55]

Having four possible values for each field does not necessarily make the data table four times as large, for values only exist when necessary. The `Default`, for instance, is applicable at the column level, not the individual field level. There is at most one default value per column, not for one for each field. [Comment 12cs.56]

In addition to multiple possible field versions, individual rows have a status value, such as `Unchanged`, `Modified`, `Added`, or `Deleted`. Understanding the meaning of the versions and status codes, and the relationship between them, is essential when programming for data sets. [Comment 12cs.57]

To illustrate this we focus on two field versions, `Original` and `Current`, and two row statuses, `Unchanged` and `Modified`. [Comment 12cs.58]

Copyright © 2003 Paul Yao & David Durant

When new rows are added to a data table, the original and current values are equal to each other and the status is set to `Unchanged`. When field values change in a data table – either because of changes done by your code, or from user actions on a data bound control –the new value becomes the `Current`, the old value remains the `Original` and the state is set to `Modified`. [Comment 12cs.59]

At some point in the execution of the program, you decide that you have "completed" your changes, that new values are no longer "new." That is, you want the Original value thrown away and replaced with the `Current` value and the row status to be set back to `Unchanged`. Normally this desire to "shuffle" versions and statuses occurs because you have pushed the changes back to the underlying database, but it could occur for a variety of other reasons as well. [Comment 12cs.60]

You accomplish this synchronization of versions and changing of states by calling the data row's `AcceptChanges` method. The data set and data table classes also have `AcceptChanges` methods, which operate by cascading the `AcceptChanges` call down to the individual rows. Never call `AcceptChanges` just prior to updating the database. The classes used for transferring data from a data table to the database examine the rows' status first. Data in rows that have their status marked as `Unchanged` do not cause updates to the database. (See *Updating a Database*, later in this chapter.) [Comment 12cs.61]

Having mentioned the `Current`, `Original` and `Default` versions of a data row, we need to say a word about the fourth version, the `Proposed` version. The `Proposed` version is used to provide transaction commit and rollback capability, and only exists during the life of a transaction. [Comment 12cs.62]

Working with DataRows

As mentioned earlier, data rows contain the actual data, which can be access or modified by specifying the row and the column within the row. Columns can be specified by either name or index. [Comment 12cs.63]

Data rows have methods for accessing other rows based on data relations that have been defined in the dataset. Two of the most common methods are `GetChildRows` and `GetParentRow`. For example, the following code retrieves an array of all rows containing all products that are `Widgets`, assuming that the `dsetDB` dataset already contains the `Categories` and `Products` tables. [Comment 12cs.64]

```
//  Define the relationship
//     between Products and Categories.

dsetDB.Relations.Add(
   "FKProdCat",
   .Tables["Categories"].Columns["CategoryID"],
   .Tables["Products"].Columns["CategoryID"],
   true);

//  Select the Widget row in the Categories data table.
//    Use it to retrieve all Widget rows from the
//    Products data table.

dsetDB.Tables["Categories"].
   Select("CategoryName = 'Widgets'")[0].
      GetChildRows("FKProdCat");
End With
```


Copyright © 2003 Paul Yao & David Durant

Introducing DataViews

A `DataView` is always a view of a single data table, and it only provides two capabilities; sorting and filtering rows. It cannot be used to do joins, nor to evaluate aggregate functions, nor to filter out columns. It is not intended to behave like a SQL Server view. Its purpose, instead, has to do with data binding. [Comment 12cs.65]

Every data table has a `DefaultView` property. When this property is first accessed, the data view is created, and its `Filter` and `Sort` properties are set to `String.empty.` So its contents are, on first access, identical to the table's contents. Thereafter, you can set the sort and filter properties to anything that you desire. [Comment 12cs.66]

Having discussed the dataset and its contained objects, it's time to cover moving data between the dataset and the presentation tier; and that means that it is time to cover *data binding*. [Comment 12cs.67]

Data Binding

Data binding is the ability to associate an object that holds data with a control that displays and updates that data. We covered the subject in chapter 8, *Data Binding*, but now we need to apply it to the data table and data view classes specifically. The following list summarizes what was covered in chapter 8. [Comment 12cs.68]

- The `DataGrid` control can display the entire contents of a bound data object.
- Binding to a `DataGrid` is done by setting its `DataSource` property.
- Unlike the desktop `DataGrid`, the Compact Framework version is read-only. A work around for this limitation was presented in chapter 8, *Data Binding*.
- The `ListBox` and `ComboBox` can display one column of a data object, and can use one other column for an identifying key.
- Binding to a `ListBox` or `ComboBox` is done by setting its `DataSource`, `DisplayMember`, and `ValueMember` properties.
- Unlike the desktop `ComboBox`, the Compact Framework version is read-only. That is, the user can select from the list of choices, but they cannot enter text into the combo box.
- Single-item controls, such as the `TextBox`, `Label`, `RadioButton`, and `CheckBox`, can bind to a single data element of a data object.
- Binding to a Single-item control is done by adding entries to its `DataBindings` collection.
- The `DataBindings` collection specifies the column to be bound to, but not the row. A data object's "current" row is always used for binding.
- Data binding is a two way street. Changes to data made in the control are automatically pushed back to the bound data table.

As a lead in to our upcoming discussion of data binding, we add the following to our list: [Comment 12cs.69]

- The only `DataSet` classes that can be data bound are the `DataTable` and `DataView` classes.
- Every `DataTable` has an associated `DataView` that can be used for data binding.
- `DataTables` do not really have a "current" row. Instead, the data tables's `CurrencyManager`, located within the forms' `BindingContext`, must be

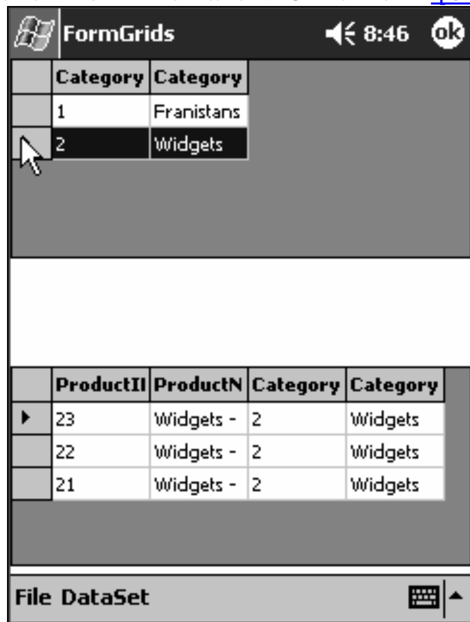
Copyright © 2003 Paul Yao & David Durant

used to position to a row.

We begin by writing a very simple application to illustrate the benefits, and issues, of data binding. We have a very simple SQL Server CE database that we use to populate the dataset. Again, we avoid comment on the code in this chapter, because moving data between a dataset and a database is the subject of later sections in this chapter. The tables in the dataset are the `Categories` and `Products` tables; the relationship between them has been defined within the data set by adding a data relation object named "FKProdCat" to the dataset. Once the dataset has been populated, we use data binding to display the information in the tables to the user. [Comment 12cs.70]

The application, located in the `DataBinding` directory for this chapter, consists of two forms. The first form, shown in figure 12-3, consists of two `DataGrid` controls, one for displaying the `Categories` rows and one for displaying the `Product` rows of whichever `Category` the user selects. In other words, we want to reflect the parent-child relationship between categories and products on the form. [Comment 12cs.71]

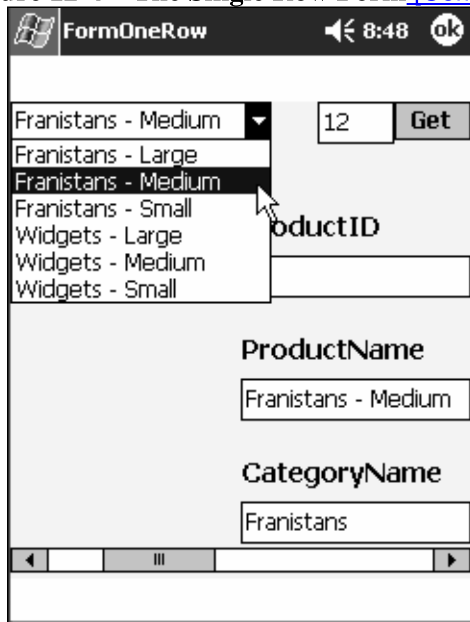
Figure 12-3 – The Parent-Child Form [Comment 12cs.72]



The second form, shown in figure 12-4, displays products, one product at a time, in textboxes, and provides the user with a variety of ways for specifying the desired product. It also allows the user to update the product. [Comment 12cs.73]

Copyright © 2003 Paul Yao & David Durant

Figure 12-4 – The Single Row Form [Comment 12cs.74]



The two forms are not related in a business sense; we are simply using one to illustrate multi-item data binding and the other to illustrate single-item data binding. [Comment 12cs.75]

The first form is the simpler to program, as we shall see, for it binds data tables to multi-item controls. [Comment 12cs.76]

Binding to Multi-item controls

The first form declares a private variable to hold the dataset, named `dsetDB`; then creates a new dataset, storing the reference in `dsetDB`; and loads `dsetDB` with data from the SQL Server CE database. At this point, we have a dataset containing two data tables and the relationship between them. [Comment 12cs.77]

To reflect that relationship on the form, we bind the upper data grid control to the `Categories` data table, so that the entire table is displayed within the data grid. We bind the lower data table to the default view of the `Products` table, as we can easily update the view when ever the user selects a new category. [Comment 12cs.78]

```
private void mitemDisplayDS_Click(object sender, EventArgs e)
{
   //  Display the Categories and Products tables
   //      in the parent and child DataGrids.
   dgridParent.DataSource =
      dsetDB.Tables["Categories"];
   dgridChild.DataSource =
      dsetDB.Tables["Products"].DefaultView;
}
```

Whenever the user selects a new category, the application reacts to the `CurrentCellChanged` event by setting the row filter for the `Products` view to select only products of that category, thusly. [Comment 12cs.79]

```
private void dgridParent_CurrentCellChanged(object sender,
```


Copyright © 2003 Paul Yao & David Durant

```
                                            EventArgs e)
      {
         DataTable dtabParent = (DataTable)dgridParent.DataSource;
         DataView dviewChild = (DataView)dgridChild.DataSource;
         dviewChild.RowFilter =
            "CategoryID = " +
            dtabParent.Rows[dgridParent.CurrentRowIndex]["CategoryID"];
      }
```

Thus, the data view class is the key piece in reflecting the parent-child relationship to the user. [Comment 12cs.80]

Binding to Single-item controls

When asked to display the second form, the first form stores a reference to the dataset into an `Internal` variable so that the second form can retrieve it, and then displays the second form. The second form displays one product row at a time, using labels and textboxes. This makes it inherently more complex than the first form, for binding to single-item controls is more difficult than binding to multi-item controls for three reasons. [Comment 12cs.81]

1. The user must be given a mechanism to specify which product should be displayed. [Comment 12cs.82]
2. The textboxes must display the specified row. [Comment 12cs.83]
3. Textboxes can be used to update data as well as display it. [Comment 12cs.84]

Data binding to single valued controls, such as textboxes, requires some additional design decisions; and making good decisions requires an understanding of data binding internals. So, let's look at the decisions that must be made and the impacts of each. [Comment 12cs.85]

The decisions are: [Comment 12cs.86]

1. How does the user designate which row is to be displayed? [Comment 12cs.87]
   a. By matching binding. Bind the data object to a multi-item control, such as a combo box or data grid, as well as to the single-row controls. The user selects the desired row from the multi-item control. [Comment 12cs.88]
   b. By indexing. Provide a scrollbar, or some other numeric position control. The user indicates the relative position of the desired row within the data object. The application positions to that row. [Comment 12cs.89]
   c. By search key. Provide a textbox. The user enters a value. The application searches the data object for the row containing the entered value. [Comment 12cs.90]
2. How should that row be assigned to the control? [Comment 12cs.91]
   a. By current row. The application designates the desired row as the current row of the data object. [Comment 12cs.92]
   b. By data view. The application binds the single-item controls to the data table's DefaultView, and then sets the view's `Filter` property to designate the desired row. [Comment 12cs.93]

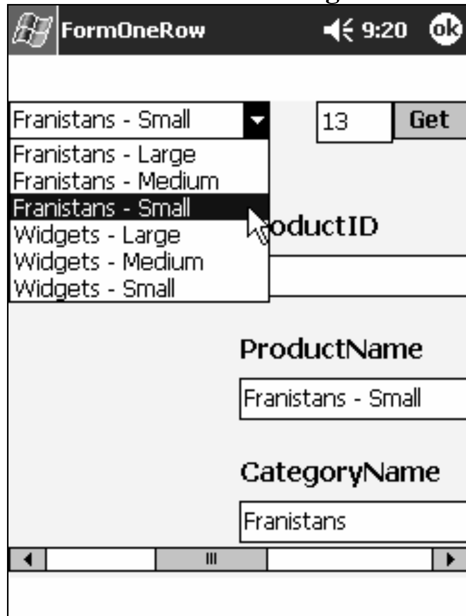
Copyright © 2003 Paul Yao & David Durant

3. When should the data table be updated with the values from the single-item controls?[7] [Comment 12cs.94]
   a. *When the user moves to a new field?* No, it is not necessary to update the data object if the user is continuing to modify other fields of the same row; wait until the user moves to a new row. [Comment 12cs.95]
   b. *When the user positions to a new row?* No, data binding causes the old row values to be updated whenever the user moves to a new row. [Comment 12cs.96]
   c. *When the user indicates that they are finished with the edit (be it through a cancel button, an update button, closing the form, etc.)?* Yes, you need to use data table's currency manager to complete or cancel the update of the current row if the user exits the operation without moving to a new row. [Comment 12cs.97]

Designating the row to be displayed

Figure 12-5 shows the second form providing all three methods. We use this form to cover the issues being addressed here. It's not the most beautiful form we ever designed, but it does illustrate the functionality we want to cover. [Comment 12cs.98]

Figure 12-5 – The Three Designation Methods [Comment 12cs.99]



Matching Binding

Of the three designation methods, the easiest to program is 1a, matching binding. Once you bind the single-item controls and the multi-item controls to the same data object, decision #2 goes away, as the users selection is automatically reflected in the single-item controls. In our example, the binding is done in the form's Load event handler, as shown here [Comment 12cs.100]

```
//  Bind the ComboBox with the Product names.
comboProductIDs.DataSource = dtabProducts;
comboProductIDs.DisplayMember = strPKDesc;
```

[7] The question of when-and-how the data should be propagated from the data table down to the database is covered later in the *SQL Server CE* and *SQL Server* sections of this chapter.

Copyright © 2003 Paul Yao & David Durant

```
         comboProductIDs.ValueMember = strPKName;
         comboProductIDs.SelectedIndex = 0;

         //  Bind the DataTable's columns to the textboxes.
         textProductID.DataBindings.Add
            ("Text", dtabProducts, strPKName);
         textProductName.DataBindings.Add
            ("Text", dtabProducts, strPKDesc);
         textCategoryName.DataBindings.Add
            ("Text", dtabProducts, strFKDesc);
```

When the user taps on an entry in the combo box, that row is displayed in the text boxes. If the user enters new values in the textboxes, those values are updated to the dataset as soon as the user selects a new row; and the validation events for the textbox fire before the row is updated. What more could we want? [Comment 12cs.101]

Set the `DisplayMember` property prior to setting the `DataSource` property. Setting the `DataSource` property causes things to happen, some of them visible to the user. Therefore, you want everything in place before setting the `DataSource` property. To illustrate this, you can take the sample code shown above and reposition the two calls such that the `DataSource` property is set prior to the `DisplayMember` property. When you run the code you may notice a slight flicker in the list box. [Comment 12cs.102]

Indexing

Decision 1b is easy to implement, but not as easy as 1a. And, although it is easy, it is not obvious. When the user positions the scrollbar to a value of “n”, the application knows that it needs to make row “n” the current row of the data table. Unfortunately, data tables do not have a current row. They do not have a current row property because the concept of a current row is only meaningful within data binding. To maintain a current row context when not data bound would be wasted overhead. [Comment 12cs.103]

Since data binding involves controls and since controls reside within forms, the `Form` class assumes the responsibility for managing the bindings and for maintaining the concept of current row. When the first binding in the code shown above that involved the `dtabProducts` data table executed, the form created an instance of the `BindingContext` class for managing `dtabProducts` bindings. As the subsequent binding code executed, that object was updated with information about the additional bindings. [Comment 12cs.104]

When the user tapped on the nth entry in the combo box, the form reacted to the combo box’s `SelectedIndexChanged` event by noting that the combo box was bound to `dtabProducts`, and that the text boxes were also bound to `dtabProducts`. So it notified the text boxes to obtain their `Text` property values from row “n” of the data table, and it updated the binding context object for `dtabProducts` to have a `Position` property value of “n”. [Comment 12cs.105]

To control the “current row” of a bound data table, you need to tap into the form’s binding management capability. Specifically, you need to obtain the `CurrencyManager` object for the data table (or data view) and set its `Position` property to “n”. The `CurrencyManager` for a data object is contained in the default collection within a form’s `BindingContext` object. In our application this means handling the scrollbar’s `ValueChanged` event, like so, (assuming the scrollbar is named `hsbRows` and its min and max values are 0 and the-number-of-rows-in the-data-object-minus-1, respectively). [Comment 12cs.106]

```
   this.BindingContext[dtabProducts].Position = hsbRows.Value;
```


Copyright © 2003 Paul Yao & David Durant

This causes the textboxes to update their `Text` property values with the values from the "`hsbRows.Value`"th row of the `dtabProducts` table. [Comment 12cs.107]

It seems like we are saying that 1b implies 2a. You might ask, why not use 2b, that is, why not bind the textboxes to `dtabProducts.DefaultView`, rather than to the data table itself, and then react to scrollbar's `ValueChanged` event by setting the view's `Filter` property with the primary key of the nth row, like so. [Comment 12cs.108]

```
         dtabProducts.DefaultView.RowFilter =
                  "ProductID = " +
                  dtabProducts.Rows[hsbRows.Value]["ProductID"];
```

Now the text boxes are bound to a view that contains only one row, which is the row that they display, and the concept of current row becomes meaningless. [Comment 12cs.109]

You can do it this way, and it works fine until you try to update the fields by entering new values into the textboxes. When you enter data into the textboxes and then reposition the scrollbar thumb, two things go wrong. First, tapping in the scrollbar does not cause the textbox to lose focus and thus the validation events do not fire. Second, you do not move from one row to another row within the bound view, rather you replace the contents of the bound view with new contents, thus the values in the underlying data table are not updated. [Comment 12cs.110]

So, although 1b does not imply 2a, we recommend implementing 2a whenever implementing 1b. As you saw in the code above, implementing 2a can be done in one line of code. [Comment 12cs.111]

Search Value

The third method for the user to use when designating the desired row, 1c, which has the user enter a search value, fits nicely with 2b, which is value based rather than position based. If the single-item controls are bound to the data table's default view, as in [Comment 12cs.112]

```
         //  Bind the DataTable's columns to the textboxes.
         textProductID.DataBindings.Add
            ("Text", dtabProducts, strPKName);
         textProductName.DataBindings.Add
            ("Text", dtabProducts, strPKDesc);
         textCategoryName.DataBindings.Add
            ("Text", dtabProducts, strFKDesc);
```

then the controls can be positioned to the requested row by extracting the user entered key value and specifying it when setting the default view's Filter property, like so. [Comment 12cs.113]

```
      dtabProducts.DefaultView.RowFilter =
         "ProductID = " + textGet.Text;
```

Since `ProductID` is the primary key, this limits the view to, at most, one row. [Comment 12cs.114]

Using variation 2b to assign the row to the textboxes in this situation does not have the data set updating problems that it had when used in conjunction with a scroll bar.

Copyright © 2003 Paul Yao & David Durant

When the user begins to enter a new key value after having entered new field values in the old row, the focus does shift, the validation events are called, and the underlying data table is updated. [Comment 12cs.115]

Conversely, variation 2a, which is position based rather than value based, does not fit well with 1c. There is no single property or method of the data table, data view, or data row classes that translates a primary key value into a row number. If you need this type of translation, you need to write your own routine to provide it. [Comment 12cs.116]

Positioning the controls to a row

Because, as we have just seen, this subject is so tightly tied to the issue of designating the row to be displayed, we have already covered it. To summarize: [Comment 12cs.117]

1. If the user is designating the desired row by selecting from a multi-item control bound to the same data object as the single-item controls, do nothing. [Comment 12cs.118]
2. If the user is designating the desired row by entering an index value, bind your single-item controls to the data table, obtain the binding context for the data table from the form, and set its Position property to the index value. [Comment 12cs.119]
3. If the user is designating the desired row by entering a search value, bind your single-item controls to the data table's default view and use its filter property to accept only rows containing the search value. [Comment 12cs.120]

Updating the bound data table

We said earlier that data binding is a two way street; changes made in the control propagate back to the dataset. However, bound controls do not update the underlying row until they are repositioned to a new row. Normally this is the behavior you want, for the row is probably out-of-synch with itself as its individual fields are being entered / updated, and it is best to wait for the automatic update that occurs when the user moves to a new row. [Comment 12cs.121]

It is always possible for the user to complete the editing of a row and not move to a new row. This is why forms tend to have specific Update / Cancel buttons on them, and why we handle not only the `Click` events of those buttons but also form's `Closing` and `Deactivate` events. If the user positions to a row, makes a change to the contents through a bound control, and then closes the form; the change that they made is not persisted, it is lost. This is because the application never repositioned to a new row after the change was made, and therefore, the data table was not modified. [Comment 12cs.122]

To programmatically complete or cancel an update; that is, to force or prevent the transfer of data from the single-item controls to the data table; use the currency manager's `EndCurrentEdit` or `CancelCurrentEdit` method, respectively. This is the same currency manager that was used earlier in this chapter to specify the "current row" of a data object. For instance, the code below reacts to the form's `Closing` event by completing the edit of the current row, thus causing the data in the controls to propagate to the row. [Comment 12cs.123]

```
using System.ComponentModel;
   :
   :
   private void FormUpdate_Closing(object sender,
                                   CancelEventArgs e)
```


Copyright © 2003 Paul Yao & David Durant

```
{
   //  Force the current modification to complete.
   this.BindingContext[dtabCategories].EndCurrentEdit ();
}
```

The currency manager has both an `EndCurrentEdit` method to force the update to occur, and a `CancelCurrentEdit` method to prevent the update from occurring; as well as the `Position` property that is used to specify the current row. [Comment 12cs.124]

So, the key points to consider when using data binding to update data set data is how to let the user position the controls to a specific row and how to prevent lost or unintended updates to that row. [Comment 12cs.125]

This concludes our discussion of moving data between the data set and the presentation layer. For the rest of this chapter we examine moving it between the dataset and persistent storage. [Comment 12cs.126]

Reading and Writing a Dataset as XML

A database is not the only source for, or destination of, dataset data. XML files can also be the storage media for the data. Unlike database access, which requires a separate provider specific class to perform the transfer, XML file I/O is done by the `DataSet` class itself. After all, an XML file is a text file of information in a standard format; therefore access to it is provided in a standard manner by derived classes of the `Stream` class. [Comment 12cs.127]

The `DataSet` class has four methods for performing XML I/O: [Comment 12cs.128]

1. `WriteXML` – Writes the contents of a dataset as XML to a file. [Comment 12cs.129]
2. `WriteXMLSchema` – Writes the structure of the dataset as XML schema to a file. [Comment 12cs.130]
3. `ReadXML` – Reads the contents of an XML file into a dataset. [Comment 12cs.131]
4. `ReadXMLSchema` – Reads the contents of an XML schema file and builds the dataset structure from it. [Comment 12cs.132]

The XML schema is translated to and from `Constraint` and `DataRelation` objects in the data set, as well as the `DataTable` objects. Unlike relational data, which is flat in structure, XML is hierarchical. The data relation classes are the mechanism for expressing the hierarchy within the data set. For instance, the following code, which we have used before, creates a data relation object that specifies the parent child relationship between the `Categories` and `Products` tables. [Comment 12cs.133]

```
//  Add relation to DataSet.
dsetDB.Relations.Add(
   "FKProdCat",
   dsetDB.Tables["Categories"].Columns["CategoryID"],
   dsetDB.Tables["Products"].Columns["CategoryID"],
   true);
```

This relationship has been in our data set ever since we first build the data set earlier in this chapter. If we now execute the `WriteXML` method, like so, [Comment 12cs.134]

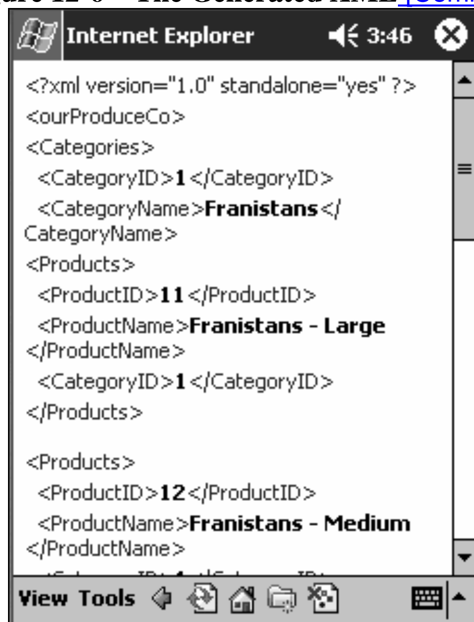
```
   //  The XML file.
```


Copyright © 2003 Paul Yao & David Durant

```
private string strXMLFile =
                        @"My Documents\ourProduceCo.xml";
         :
         :
private void mitemWriteXML_Click(object sender,
                                   EventArgs e)
{
   dsetDB.WriteXml(strXMLFile);
}
```

and then view the contents of the file, as shown in figure 12-6, we see the XML that was generated. [Comment 12cs.135]

Figure 12-6 – The Generated XML [Comment 12cs.136]



The XML that we see here not only contains the data but also reflects the relationship between the tables; the first category element is the first entry in the file, and it contains the product elements for all its products, then the next category element, and within it all its product elements, etc. This is a nested relationship, and it is the reason why the data relation class has a `Nested` property. [Comment 12cs.137]

When we added the `FKProdCat` data relationship to our dataset in the code shown above, it was added with a `Nested` property value of false. To ensure that your XML, and XML schema, reflects the hierarchical nature of your dataset, set the `Nested` property of your data relationships to true, like so. [Comment 12cs.138]

```
   //  Make each relationship a nested relationship
   foreach( DataRelation drelForXML in dsetDB.Relations )
   {
      drelForXML.Nested = true;
   }
```

Or, for a single data relation, [Comment 12cs.139]

```
//  Make the FKProdCat relationship nested.
```


Copyright © 2003 Paul Yao & David Durant

```
dsetDB.Relations["FKProdCat"].Nested = true;
```

It is also possible to obtain nested XML by having matching primary key and foreign keys defined on the respective data tables, but nested relationships are usually easier to work with, as all the information about the relationship is contained within a single data relation object rather than being spread across two constraint objects. [Comment 12cs.140]

Thus, .NET Compact Framework datasets give you a convenient way to convert relational data to and from XML format. This, in turn, gives you a way to save dataset data that your application has captured to a file; without incurring the overhead of a using a database to do so. [Comment 12cs.141]

Microsoft SQL Server CE

There are a variety of databases available for mobile devices, but the one we use throughout these chapters is *SQL Server 2000 Windows CE Edition*, a slimmed-down version of Microsoft's desktop database, *SQL Server 2000* (in the interest of brevity, we refer to the CE version as "SQL Server CE"). It is the mobile database of choice for .NET Compact Framework programming, being fully supported in the Compact Framework runtime environment and the Compact Framework development environment. [Comment 12cs.142]

Most applications written for SQL Server CE have two primary database tasks: [Comment 12cs.143]

1. Manipulating the data while disconnected, and [Comment 12cs.144]
2. Transferring the data to SQL Server when the machine is connected. [Comment 12cs.145]

The first database task involves using the generic ADO.NET class which we discuss earlier in this chapter, and the provider-specific classes which we discuss shortly. For programmers who have worked with ADO.NET on desktop systems, all of the generic classes on the Compact Framework implementation of ADO.NET are going to be very familiar. [Comment 12cs.146]

The second major database task, the transfer of data between SQL Server and SQL Server CE, involves not only the participating database engines but also Microsoft Internet Information Services (IIS). There are two different mechanisms available to help with this: Merge Replication and Remote Data Access (RDA). These two topics are the subject of chapter 13 (Synchronizing Mobile Data). [Comment 12cs.147]

Accessing SQL Server CE Databases from Native Code

ADO.NET classes are only available in managed code. But you do not need managed code to access the data in a SQL Server CE database. There are two separate native APIs available for manipulating the data in a SQL Server CE database. One involves a COM object (ADOCE), and the other is an API (OLEDBCE). These components match up with their desktop counterparts, ADO and OLE DB. [Comment 12cs.148]

SQL Server CE Files

Each SQL Server CE database is stored as a single file on your CE device. The recommended naming convention for the file extension is ".sdf", and the name of the file is the name of the database. Thus, you open a connection to "mydb.sdf", not to "mydb".

Copyright © 2003 Paul Yao & David Durant

You may associate a password with the database and may also encrypt the physical file using 128bit RSA encryption. Since each database is one file, they are visible in the file Explorer. Figure 12-7 shows a database file selected in the Explorer. [Comment 12cs.149]

Since SQL Server CE databases are files in the Windows CE file system, they can be accessed using the `File` and `Directory` classes that we discuss in Chapter 11, *Storage*. You can, for example, copy a database file, delete a database file, or test for the existence of specifically-named database file. Your ability to operate on database files is subject to the same restrictions as other files in the file system; for example, you can only delete a database file if the file is currently not in use. [Comment 12cs.150]

Figure 12-7 – A Database File Viewed in File Explorer [Comment 12cs.151]

SQL Server CE only reclaims space within a database file when you ask it to; it's not a capability available through the data manipulation language, nor is it provided as a background process. To reclaim space within a SQL Server CE database you must use the `Compact` method of the `SqlCeEngine`[8] class. [Comment 12cs.152]

SQL Server CE Syntax

SQL Server CE programming is simpler than SQL Server programming because the SQL language supported by SQL Server CE is a subset of the SQL that is available with SQL Server. For example, since a SQL Server CE database is a single user database whose security is controlled by a file password, there is no need for `GRANT`, `DENY`, `REVOKE`. [Comment 12cs.153]

Also missing are SQL Server's Transact-SQL extensions; no stored procedures, triggers, multi-statement batches, or `DECLARE`, `SET`, `IF`, `WHILE` statements. Even some standard SQL had to be left out, for example views are not supported. Not all the SQL Server data types could be included, but most of the missing ones can be converted to ones that do exist. For example, Windows CE itself only supports Unicode, and so Unicode is the only character data type (`nchar`, `nvarchar`, `ntext`). [Comment 12cs.154]

[8] Fully-qualified name: `System.Data.SqlServerCe.SqlCeEngine`.

Copyright © 2003 Paul Yao & David Durant

What SQL Server CE does provide is support for tables, indexes, defaults, and referential integrity. It also has the ability to add, modify, and delete rows in those tables using standard SQL Data Manipulation Language (DML). Therefore, your application can manipulates data in SQL Server CE by connecting to the database file and submitting `INSERT`, `UPDATE`, and `DELETE` statements to the database. You can write these DML statements yourself, or you can have ADO.NET write them for you as one step in the synchronizing of data set updates to the database. Your SQL Server CE-based application may be more elementary than what you've come to expect from a typical SQL Server application, but it should provide enough functionality to be useful in mobile situations. The lack of the complete Transact-SQL syntax is an inevitable limitation given the available memory of a typical Windows CE-powered device. And yet in spite of this size limitation, SQL Server CE provides a more than adequate set of SQL commands for just about any Windows CE-based application that you are likely to need. [Comment 12cs.155]

The SQL syntax uses single quotes to delineate literals, which is very convenient when you are quoting that SQL within your code, for it eliminates the need to quote quotes. Table 12-4 and Table 12-5 compare available SQL Server CE functionality with that of SQL Server. [Comment 12cs.156]

ANSI Standard SQL

This specification for the SQL syntax is a standard established by an ANSI (American National Standards Institute) standards committee. Both SQL Server and SQL Server CE are highly compliant with the ANSI 92 standard. SQL Server 2000 has some options for choosing between ANSI compliance and compliance with older versions of SQL Server. These options are not supported in SQL Server CE; ANSI compliance is the only choice in SQL Server CE. [Comment 12cs.157]

Table 12-4 –Unsupported SQL Server functionality in SQL Server CE [Comment 12cs.158]

Functionality	Comments
DCL[9] GRANT, REVOKE, DENY	Not needed in a single user database
DDL[10] Views, Triggers, Stored Procedures, User defined functions, User defined data types.	Most are SQL Server Transact-SQL extensions to the ANSI functionality.
DML[11] IF-ELSE, WHILE, DECLARE, SET.	Most are SQL Server Transact-SQL extensions to the ANSI functionality.
INFORMATION_SCHEMA TABLES	Replaced by `MSysObjects` and `MSysConstrains` tables.

Table 12-5 –Supported functionality in SQL Server CE [Comment 12cs.159]

Functionality	Comments
DDL Databases, Tables, Data types, Indexes, Constraints.	Only Unicode character types are supported.

[9] DCL = “Data Control Language”.
[10] DDL = “Data Definition Language”
[11] DML = “Data Manipulation Language”

Copyright © 2003 Paul Yao & David Durant

DML SELECT, INSERT, UPDATE, DELETE	
Functions Aggregate, Math, DateTime, String, System.	
Transactions	Transaction isolation level is always READ COMMITTED. The limit on nesting is 5. Exclusive lock granularity is table level, and is held for the duration of the transaction. Single phase commit only.

SQL Server CE is a little fussier about syntax than SQL Server. Having a smaller footprint means having less code that can "deduce" what you meant. Thus the following SQL, which is missing the comma between the last column definition and the constraint definition that is required by ANSI 92 specification, executes as intended in SQL Server 2000, but produces a syntax error in SQL Server CE. [Comment 12cs.160]

```
CREATE TABLE Products
           ( ProductID integer not null primary key
           , ProductName nchar(20) not null
           , CategoryID integer not null
             CONSTRAINT FKProductCategory foreign key (CategoryID)
                                 references Categories(CategoryID)
           )
```

Constraints

There are two primary constraints for databases; Primary key and Foreign key. The Primary key constraint specifies the column or columns of a table in which duplicate values are prohibited, and a non-null value is required. The product ID in a product table and the order number in an order table are both examples of primary keys. Foreign keys are values that exist as primary keys elsewhere in the database. For instance, the customer number column in an order table is an example of a Foreign key, for its value must exist in the customer table also. [Comment 12cs.161]

Column names can be aliased within a `SELECT` statement. These aliases propagate into the data objects and on to the bound controls by default. Thus the following `SELECT` statement, used to fill a `DataTable` that is bound to a `DataGrid`, would provide the results shown in Figure 12-8. [Comment 12cs.162]

```
SELECT P.ProductID as ID
     , P.ProductName as Name
     , C.CategoryName as Category
  FROM Products P
  JOIN Categories C on C.CategoryID = P.CategoryID
```


Copyright © 2003 Paul Yao & David Durant

Figure 12-8 - A DataGrid with Aliased Column Names [Comment 12cs.163]

FormMain 11:14

ID	Name	Category
11	Franistans - Large	Franistans
12	Franistans - Medium	Franistans
13	Franistans - Small	Franistans
21	Widgets - Large	Widgets
22	Widgets - Medium	Widgets
23	Widgets - Small	Widgets

Test

So, if you do not want the existing column names presented to the user, alias them within the `SELECT` statement. [Comment 12cs.164]

SQL Server CE Query Analyzer

The primary utility tool for viewing and querying a SQL Server CE database is the *SQL Server CE Query Analyzer*. Like its counterpart for SQL Server 2000, the SQL Server CE Query Analyzer provides a convenient way to create and submit ad-hoc queries. [Comment 12cs.165]

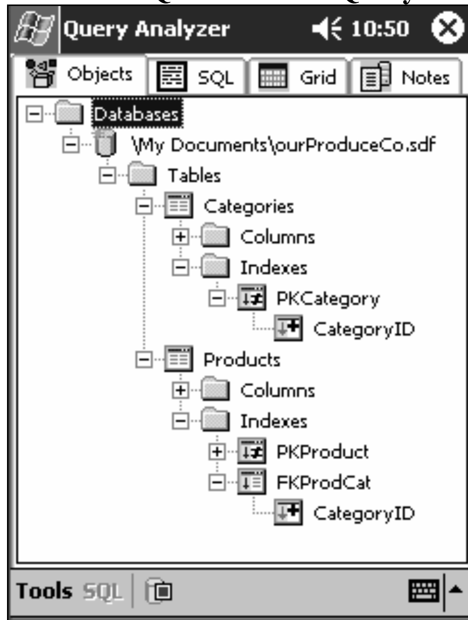
The installation of SQL Server CE Query Analyzer depends on the development environment onto which SQL Server CE is installed. When you install SQL Server CE, SQL Server CE Query Analyzer is not installed by default on a device[12]. If Query Analyzer is installed on the device then you can run the executable file, `Isqlw20.exe`, directly from the **Start** menu or from the directory in which it is installed. Clicking on a database file from within File Explorer also opens Query Analyzer. [Comment 12cs.166]

Query Analyzer allows you to see structural information about the database, as shown in figure 12-9, and submit queries to that database, as shown in figure 12-10. [Comment 12cs.167]

[12] For details, see SQL Server CE Books Online.

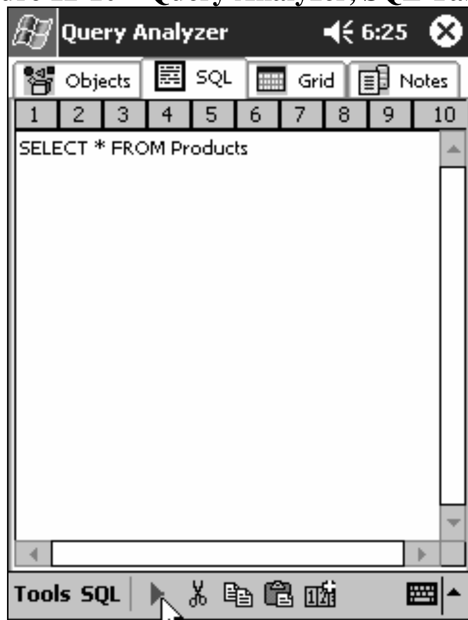
Copyright © 2003 Paul Yao & David Durant

Figure 12-9 – SQL Server CE Query Analyzer, Objects Tab [Comment 12cs.168]



The Query Analyzer Form displays a `minimize box`, not a `close box` (a topic we discuss more fully in chapter 5, *Creating Forms*). If you use the Query Analyzer to examine your database and then click on the minimize box, it disappears but it does not close, nor does it close the database that it is displaying. If another application tries to access the database, it can not do so until you close Query Analyzer. To close Query Analyzer, select ***Tools->Exit*** . [Comment 12cs.169]

Figure 12-10 – Query Analyzer, SQL Tab [Comment 12cs.170]



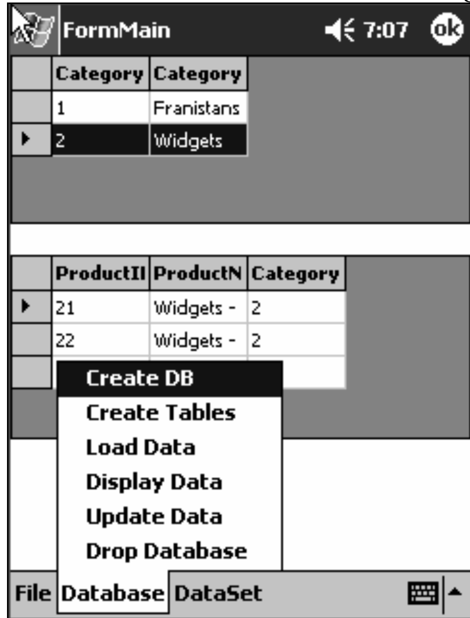
To help explain the ADO.NET classes, we write a series of sample applications beginning with database creation and proceeding through table and index creation; retrieving, displaying, and updating data using the two-tier approach; and then into the three-tier approach of creating and using data sets, and data binding. [Comment 12cs.171]

Copyright © 2003 Paul Yao & David Durant

Creating a SQL Server CE Database

Our first sample application for accessing SQL Server CE is located in the `CreateDatabase` directory for this chapter. It is used to create and populate a database, and is shown in figure 12-11. [Comment 12cs.172]

Figure 12-11 – The Create Database Program [Comment 12cs.173]



To create a SQL Server CE database, use the `Engine` class located in the `System.Data.SqlServerCe` namespace. Since each SQL Server CE database is a single file, all that you need to tell the engine object is the path and name of that file. You can do this either in the engine class constructor or by setting its `LocalConnectionString` property. In either case you must specify the file name prior to executing the `CreateDatabase` method of the engine class, for this method does not take any parameters. [Comment 12cs.174]

Thus, the code to create a SQL Server CE database looks like this. [Comment 12cs.175]

```
      private string  strFile = @"My Documents\ourProduceCo.sdf";
      private string  strConn = "Data Source=" +
                                @"My Documents\ourProduceCo.sdf";


      private void mitemCreateDB_Click(object sender, EventArgs e)
      {
         if ( File.Exists(strFile) ) { File.Delete(strFile); }

         SqlCeEngine dbEngine = new SqlCeEngine();
         dbEngine.LocalConnectionString = strConn;
         try
         {
            dbEngine.CreateDatabase();
         }
         catch( SqlCeException exSQL )
         {
```


Copyright © 2003 Paul Yao & David Durant

```
                MessageBox.Show("Unable to create database at " +
                                strFile +
                                ". Reason:  " +
                                exSQL.Errors[0].Message );
            }
        }
```

Although the engine has a `CreateDatabase` method, it does not have a `DropDatabase` method. Dropping a database must be done by submitting a `DROP DATABASE` statement to the SQL Server CE database, or using the file class to delete the file. [Comment 12cs.176]

Populating a SQL Server CE Database

Now that we have our SQL Server CE Database created, we can populate it with tables, indexes and data. You might think that you use the engine class to do this, but you do not. The engine class only operates on an entire database, not on the individual components within the database, such as tables. In the next chapter we look at populating a SQL Server CE database from a connected SQL Server database, but in this chapter we are focus on the application's creating and maintaining the database. Therefore, to populate our SQL Server CE database, we use standard SQL DDL statements, such as `CREATE TABLE` or `CREATE INDEX`, and the standard SQL DML language statements `INSERT`, `UPDATE`, and `DELETE`. [Comment 12cs.177]

Submitting SQL statements to a database requires two classes, one to open the connection and one to submit the statement. [Comment 12cs.178]

The Connection and Command Classes

The `SqlCeConnection` class opens a connection to a database, and so needs the name of the database file. The `SqlCeCommand` class submits one SQL statement at a time to the database using an `Execute` method, and needs to know the connection object to use and the SQL statement to be submitted. The `SqlCeConnection` object and the DML statement are properties of the `SqlCeCommand` object. They must be set, and the connection must be opened, before the command's execute method can be called. There are three possible execute methods, summarized in table 12-6. [Comment 12cs.179]

Table 12-6 – The Execute Methods of the SqlCeCommand class [Comment 12cs.180]

Method	Function
`ExecuteNonQuery`	Execute a SQL statement that returns no rows, such as INSERT or CREATE.
`ExecuteScalar`	Execute a SQL statement that returns just one value, such as SELECT SUM(Value) FROM Orders WHERE CustomerID = "ABCD".
`ExecuteReader`	Execute a SQL statement that returns multiple columns or multiple rows.

Thus the code to create two simple tables is: [Comment 12cs.181]

```
        private void mitemCreateTables_Click(object sender, EventArgs e)
        {
            SqlCeConnection  connDB = new SqlCeConnection();
```


Copyright © 2003 Paul Yao & David Durant

Thank You

Thank you for taking the time to read this preview chapter. We hope it has provided you insights and tips to help with your Compact Framework programming project. You can help us create a better book by clicking the comment link at the end of each paragraph, and sending your comments and suggestions on our review web site.

Preview Chapter Text

Our public review site provides the complete table of contents for the Compact Framework book at this link: http://www.paulyao.com/cfbook.htm. That table of contents contains links to all the preview chapters. The preview chapter provides the complete outline of topics covered in a chapter, and also the first section or two from each chapter.

Complete Chapter Text

You can get the complete text for each chapter, available to readers who register at our web site. Registration is simply and easy – we only ask for an email address. To register, click on this link: http://www.paulyao.com/ReaderFeedback/Logon.aspx.

When you register, you can download the available chapters from the full-text Table of Contents, available at this link: http://www.paulyao.com/ReaderFeedback/default.aspx. We notify registered readers of new chapters – and chapter updates – as they become available.