CHAPTER 5

Caching Data with SQL Server CE

Executive Summary

Chapter 3 focused on how applications written with the Compact Framework could manipulate data locally on a mobile device. Although this capability is important, many application scenarios require more robust data caching. These scenarios comprise many applications of the occasionally connected type, including sales-force automation, field-service automation, real estate, and home-visit medical applications, among others. For these applications a local relational database that features data integrity, built-in synchronization, access from multiple development environments, and strong security is a must.

Shortly before VS .NET 2003 and the Compact Framework were released, Microsoft shipped SQL Server 2000 Windows CE Edition 2.0 (SQLCE 2.0),[1] which fulfills these requirements. This local database engine consisting of a storage engine and query processor is implemented as an OLE DB provider and runs in-process with Compact Framework applications. SQLCE 2.0 includes a number of new features, including parameterized queries, index seeks, and the `UNION` clause.

To provide access to SQLCE, Compact Framework developers can use the SqlServerCe .NET Data Provider. This provider is implemented using a common set of interfaces and classes and therefore allows developers to leverage their existing ADO.NET knowledge and begin writing applications for SQLCE. The classes in the provider, for example, `SqlCeEngine`, can be used to create databases, tables, and indexes programmatically, in addition to compacting databases and querying and modifying data in disconnected

[1] Some prefer to use the acronym SSCE, but we prefer SQLCE because we believe it is easier to understand.

Fox/Box.book Page 154 Tuesday, September 30, 2003 9:10 PM

and connected scenarios. In particular, the ability of SqlServerCe to support index seeks directly on tables can greatly speed the performance of an application because it eliminates the overhead of the query processor. A common approach to manipulating data in a SQLCE database is to write data-access utility classes that encapsulate calls to the provider and distribute the classes to other developers in an organization in an assembly.

Because the Compact Framework supports accessing both a remote SQL Server through the SqlClient .NET Data Provider and local SQLCE databases, developers can use the Abstract Factory design pattern to create factory classes that can be utilized to abstract which provider is used at runtime. This is particularly effective for occasionally connected applications that require access to a remote SQL Server when connected to the network through a WLAN, for example.

SQLCE also offers a high level of security by supporting both password protection and encryption of the database on the device. This is important because the device on which SQLCE is running is inherently mobile and can easily fall into the wrong hands. The encryption algorithm is based on the password; therefore, passwords of eight or more characters are recommended.

SQLCE is included with VS .NET 2003 and automatically installs on the developer workstations. It is deployed automatically when a developer references the SqlServerCe .NET Data Provider in his or her application. It is also possible, and sometimes a preferred strategy, to prebuild SQLCE databases that will be included in RAM or on a storage card and that are large or will be deployed to a large number of devices.

The Role of SQLCE

In Chapter 3 the discussion focused on how to handle file, XML, and relational data locally on a smart device. Part of that discussion showed how data could be persisted on the device and later reretrieved and displayed to the user. Although these techniques are acceptable for some applications, many applications require more sophisticated local storage. To address this need, this chapter focuses on the third essential architectural concept: robust data caching.

History of SQLCE

Along with the release of the Pocket PC 2000 platform, Microsoft anticipated the need to extend the data-management capabilities of enterpriseto-mobile devices. As a result, they shipped SQL Server 2000 Windows CE Edition 1.0 (SQLCE 1.0), code-named “Pegasus,” in late 2000 to coincide with the release of SQL Server 2000.[2] SQLCE provides a local relational database of multiple tables running on a device that can be queried with a subset of the Transact-SQL syntax supported in SQL Server 2000. In addition, the first version supported referential integrity, transactions, and even accessing data remotely from SQL Server 6.5, as well as merge replication with SQL Server 2000. Developers could access SQLCE 1.0 using the ActiveX Data Objects for Windows CE 3.1 (ADOCE 3.1) library that shipped with the product.[3] This library, analogous to the ADO 2.x components used with VB 6.0 to access server-based relational data, allowed eVB developers to manipulate SQLCE databases on the device, while eVC++ developers could make OLE DB calls directly using the OLE DB provider for SQL Server CE (OLE DB CE).

Although SQLCE was well received, in order to access remote servers in version 1.0, the device had to be connected to the network via a modem or network card. Microsoft released version 1.1 in June of 2001 and added the SQL Server CE Relay product, which allowed the device to access remote servers when cradled using ActiveSync 3.1. In addition, version 1.1 was included in Microsoft Platform Builder 3.0 as a component and could be deployed as part of an embedded device, as described in Chapter 1.

With the planned release of the .NET Compact Framework in early 2003, it became essential that Compact Framework applications be able to take advantage of SQLCE easily. In September of 2002 Microsoft officially released SQL Server 2000 Windows CE Edition 2.0 (SQLCE 2.0, or from here on out simply SQLCE), which added integration with the Compact Framework through the `System.Data.SqlServerCe` namespace and also added important new functionality in the query processor and storage engine. As a result, applications built on the Compact Framework can use SQLCE for robust data caching directly from managed code and needn’t use ADOCE or the OLE DB CE API.

[2] Prior to this, developers had to rely on proprietary schemes, Pocket Access, or the Windows CE database (CEDB) format accessible through Windows CE APIs or through ADOCE, which, to say the least, lacked many of the features developers expect in a relational database system.

[3] In addition, Microsoft released the ActiveX Data Object Extensions for Data Definition Language and Security (ADOXCE) to extend ADOCE for creating, deleting, and modifying schema objects.

NOTE: As you might imagine, because SQLCE and the Compact Framework are separate products, they do not always overlap. For example, SQLCE can be run on Handheld PC 2000 (H/PC 2K) devices, including the HP Jornada 720 and Intermec 6651, and embedded devices built with the Platform Builder 3.0, such as the Intermec 5020. For using SQLCE on these devices, developers will need to use eVB and eVC++.

Robust Data Caching

So what does the term "robust data caching" actually mean? This concept addresses several key elements, including the following:

- *Local relational database access:* SQLCE supports a programming model on a smart device, which most developers are familiar with on the desktop. This allows developers to leverage their skills when creating applications and organizations to extend their data-management capabilities to devices. This is particularly the case for SQL Server developers because SQLCE supports familiar Transact-SQL syntax. Desktop Framework developers will also be able to get up to speed quickly because access to SQLCE is provided through a .NET Data Provider, using the standard classes and interfaces of ADO.NET.
- *Disconnected data integrity:* SQLCE provides a robust cache for storing data on the device for use when the device is disconnected from the network. Because SQLCE supports relational database features such as unique indexes and foreign keys, the data can be manipulated on the device while it is disconnected and still maintain its integrity.
- *Built-in synchronization:* SQLCE includes two different synchronization mechanisms that are built in, thereby saving developers from having to write complex infrastructure code for occasionally connected applications. These techniques will be addressed in Chapter 7.
- *Managed or unmanaged access:* SQLCE is architected as an OLE DB CE provider and, therefore, can be accessed using either the native SqlServerCe .NET Data Provider in the Compact Framework or ADOCE; therefore it can be used with both VS .NET 2003 and eVB/eVC++.
- *Security:* A key part of being a robust data cache is securing the data. As will be discussed later in this chapter, SQLCE supports password protection and data encryption so that data stored on the device is secure.

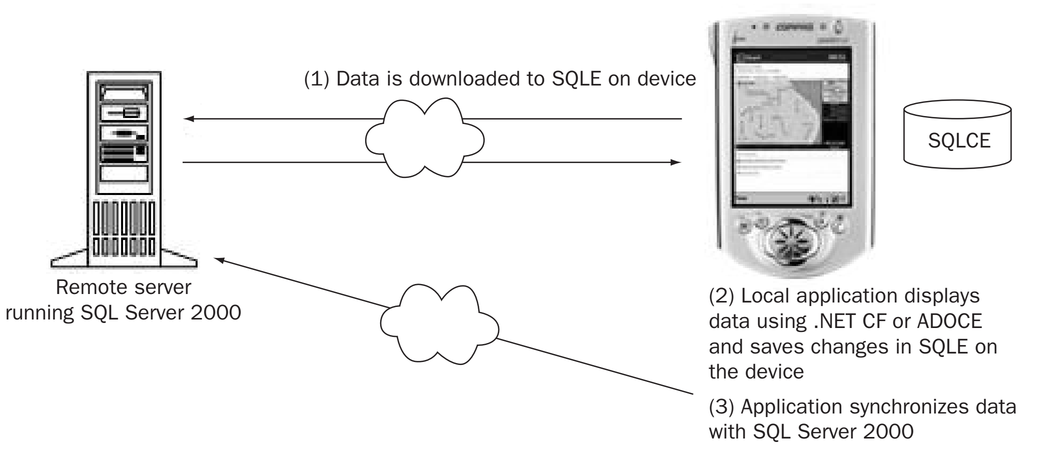
(1) Data is downloaded to SQLE on device
SQLCE
Remote server running SQL Server 2000
(2) Local application displays data using .NET CF or ADOCE and saves changes in SQLE on the device
(3) Application synchronizes data with SQL Server 2000

Figure 5–1 *The Role of SQLCE*. This diagram shows the role of SQLCE in storing local copies of data that are then modified on the device and later synchronized with the remote server.

Obviously, applications that can utilize these features run the gamut, but typically they fall into the occasionally connected scenario where data is downloaded to the device, stored in SQLCE, accessed from SQLCE and updated on the device, and then later synchronized with a back-end data store such as SQL Server 2000, as shown in Figure 5-1.

KEY POINT

As you can imagine, an architecture like that shown in Figure 5-1 is useful in a variety of application scenarios, including sales-force automation, where mobile users are downloading and updating customer information; field-service automation, where delivery drivers and maintenance workers download and process deliveries and work orders; real estate, where agents download MLS listings; and medical applications, where doctors and nurses download and update patient and prescription information. Many of these solution scenarios have been implemented and documented as cases studies on the SQL Server CE Web site referenced in the "Related Reading" section at the end of the chapter.

Differences with Local Data Handling

As discussed in Chapter 3, Compact Framework developers can use the ADO.NET `DataSet` object to persist data on the device.[4] As covered in

[4] Although not mentioned in Chapter 3, the Compact Framework does not support typed `DataSet` objects, which are classes derived from the `DataSet` class and generated through a visual designer and code generator in VS .NET.

Chapter 4, the `DataSet` can be populated from a remote server by calling an XML Web Service or by connecting to a remote SQL Server directly, using the SqlClient .NET Data Provider. While these techniques are well suited to some applications, there are several important advantages that a robust data-caching product such as SQLCE provides that ADO.NET cannot:

- *Secure and efficient data storage:* As mentioned previously, SQLCE supports encryption and database passwords. Data stored in `DataSet` objects is persisted to XML and has no such protection. In addition, XML is a verbose format and consumes more memory on the device than does data stored natively in SQLCE.
- *Query access to multiple tables:* Although the `DataSet` object can include multiple `DataTable` objects and can even include primary and foreign keys, it does not provide the ability to query those tables using `SQL` and `JOIN` clauses. SQLCE, as a relational database, makes it easy to query multiple tables through joins.
- *Query performance for large data sets:* As a corollary to the previous point, `DataSet` objects must be programmatically manipulated and, therefore, are much more processor-intensive to query. SQLCE is well suited to querying large amounts of data using a data reader, whereas `DataSet` objects are useful when manipulating between 10 and 100 rows.
- *Performance when populating:* Populating a `DataSet` from an XML Web Service does not offer any support for compression during the transmission of the data, whereas SQLCE does compress data when using its synchronization techniques. As a result, a larger amount of bandwidth is required for accessing the same amount of data with a `DataSet` and XML Web Service.

SQLCE Architecture

Once you understand the role that SQLCE plays in a solution, it is important to understand its architecture and features. The diagram shown in Figure 5-2 illustrates the architecture of SQLCE and the various software components that make up a solution that uses it.

As you can see from Figure 5-2, SQLCE itself is implemented as a DLL and an OLE DB provider (OLE DB CE) that can be accessed from both managed code using the .NET Data Provider for SQL Server CE (SqlServerCe is discussed in the following section) and eVB using ADOCE and directly

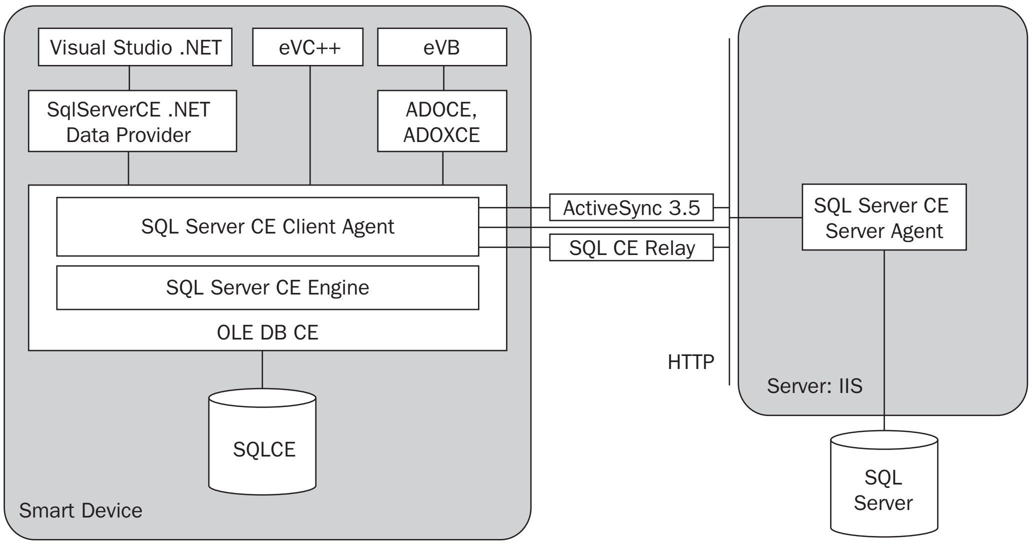

Figure 5–2 *SQLCE Architecture*. This diagram shows all the software components that make up an application that uses SQLCE. Note that both client and server components are required and that SQLCE can be accessed from managed and unmanaged code.

using eVC++. This provider encapsulates the SQL Server Client Agent that is responsible for replication and RDA, discussed further in Chapter 7, and the SQL Server CE Engine.

SQL Server CE Engine

KEY POINT

Unlike the server version, the SQLCE database engine is implemented in a DLL for performance reasons, even though the engine will be loaded in each process, using SQLCE on the device. However, because typically only one application using SQLCE will be active at any one time and because SQLCE supports only one concurrent connection, it is not a significant issue.

The database engine consists of two components: the storage engine that manages the data stored on the device in 4K pages and the query processor that processes (compiles, optimizes, and generates query plans) queries sent from applications. Together these two components support the following features, among others:

- Query processor supports SQL, including `SELECT`, `MAX`, `MIN`, `COUNT`, `SUM`, `AVG`, `INNER/OUTER JOIN`, `GROUP BY/HAVING`, `ORDER BY`, `UNION`, and operators including `ALL`, `AND`, `ANY`, `BETWEEN`, `EXISTS`, `NOT`, `SOME`, `OR`, `LIKE`, `IN`, also Transact-SQL including `DATEADD`, `DATEDIFF`, `GETDATE`, `COALESCE`, `SUBSTRING`, and `@@IDENTITY`[5] among others

- 249 indexes per table, multicolumn indexes
- Databases of up to 2GB
- BLOBs of up 1GB
- 255 columns per table
- 128 character identifiers
- Unlimited nested subqueries
- Nested transactions
- Support for `NULL` values
- Parameterized queries
- Data Manipulation Language (DML): `INSERT`, `UPDATE`, and `DELETE`
- Data Definition Language (DDL): `CREATE`, `DROP`, `ALTER` on databases, tables, and indexes
- 17 data types, including Unicode (`nchar`, `ntext`) and GUID (`uniqueidentifier`)
- `PRIMARY KEY`, `UNIQUE`, and `FOREIGN KEY` constraints
- Replication tracking capability that tracks changed data on the device as discussed in Chapter 7

Query Analyzer

Although not shown in Figure 5-2, SQLCE also ships with an improved Query Analyzer that runs on the device and can be used to create databases, tables, and indexes; query data; insert and delete rows; compact and repair a database. It is generally used by developers to ensure that their local database is accessible.

When using the Compact Framework, Query Analyzer (`Isqlw20.exe`) is deployed to the device automatically using a .cab file if the application references the SqlServerCe provider, and a shortcut is placed in the Start menu on the device.[6]

To use the Query Analyzer, a developer need simply navigate to the database file (typically with an .sdf extension) and tap the green arrow on

[5] Both `IDENTITY` columns and `uniqueidentifier` can be used to create system-assigned primary key values in a SQLCE table. However, if all rows in the table are deleted and the database compacted, the identity counter is reset to its original value. This behavior can affect applications that need to synchronize with a back-end database.

[6] eVB and eVC++ developers must manually copy the Query Analyzer to the device, along with the appropriate supporting files, as noted in the documentation.

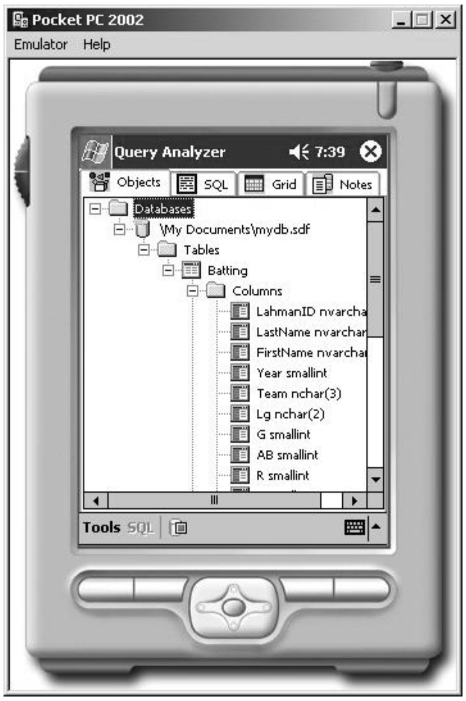

Figure 5–3 *Query Analyzer*. This screen shot shows navigating tables and columns in a local SQLCE database using the Query Analyzer.

the bottom of the screen. The tables and their structures can then be navigated, as shown in Figure 5-3. Developers can then query the data in a table by tapping on the arrow or tapping the SQL pane and writing the SQL directly.

Accessing SQLCE

To access SQLCE Compact Framework programmatically, developers can use the managed .NET Data Provider referred to as SqlServerCe. In this section we'll explore the data provider and how it can be used to connect to, query, and update SQLCE, and we'll also show a technique for writing provider-independent code when an application must access both a remote SQL Server and SQLCE.

SqlServerCe Provider Architecture

KEY POINT

The SqlServerCe provider was implemented using the same pattern as the SqlClient .NET Data Provider used to access remote SQL Servers, as discussed in the previous chapter; therefore, it consists of the same basic classes. This parity with the desktop provider allows developers to leverage their existing ADO.NET knowledge and begin writing applications for SQLCE.

Unlike SqlClient, however, the SqlServerCe provider is shipped in an assembly separate from `System.Data.dll` and, so, must be explicitly referenced by the developer in his or her SDP. Figure 5-4 shows a diagram of the architecture of the provider, all of whose classes are found in the `System.Data.SqlServerCe` namespace.[7]

You'll notice in Figure 5-4 that the layout of the classes is similar to that found in Figure 4-4. For example, SqlServerCe supports both the disconnected programming model using the `DataSet` via the `SqlCeDataAdapter`

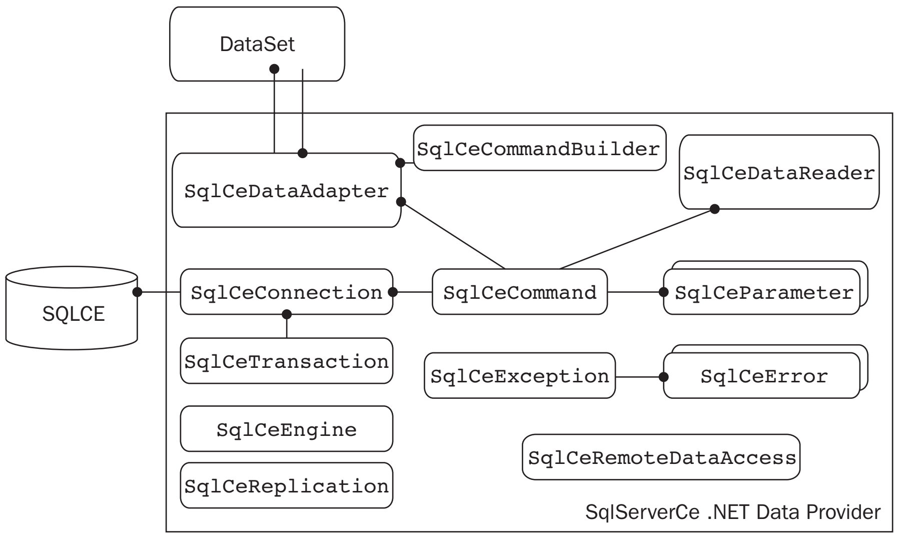

Figure 5–4 *SqlServerCe Architecture*. This diagram shows the primary classes found in the `System.Data.SqlServerCe` namespace in the SqlServerCe .NET data provider. Not shown are the `collection` and `events` classes, delegates, and enumerations.

[7] All the listings and code snippets in this chapter assume that the `System.Data.SqlServerCe` and `System.Data` namespaces have been imported (using C#).

object and the connected model using SqlCeDataReader. SQL commands are encapsulated with the SqlCommand class and can use parameters represented by SqlCeParameter. The SqlCeConnection and SqlCeTransaction objects also support local transactions, while database engine errors are captured in SqlCeError objects and thrown using a SqlCeException object. In fact, SqlServerCe even supports the SqlCeCommandBuilder class that can be used to create the INSERT, UPDATE, and DELETE statements automatically for synchronizing data in a data set with SQLCE. However, you'll also notice that SqlServerCe includes the additional classes shown in Table 5-1. These classes are found only in the SqlServerCe provider and have no analogs in SqlClient.

Although SqlCeEngine will be discussed in the following section, both SqlCeReplication and SqlCeRemoteDataAccess used for synchronization will be covered in detail in Chapter 7.

Manipulating Data with SqlServerCe

Once the SqlServerCe provider is referenced in an SDP, it can be used to manipulate SQLCE on the device. In this section we'll look at the common tasks developers will need to perform against SQLCE.

Creating Databases and Objects

Although a database with the appropriate structure and data can be deployed with the application, it is sometimes necessary for developers to create databases and objects on the fly. This can be accomplished using the CreateDatabase method of the SqlCeEngine object. In fact, a good strategy is to encapsulate the creation in a utility class and expose the functionality through shared methods like that shown in Listing 5-1.

Table 5–1 Additional SqlServerCe Classes

Namespace	Use
`SqlCeEngine`	Includes the methods and properties used to manipulate the SQL Server CE engine directly.
`SqlCeReplication`	Allows developers to use merge replication with SQL Server 2000; discussed fully in Chapter 7.
`SqlCeRemoteDataAccess`	Allows developers to access a data store remotely and synchronize its data with SQLCE; discussed fully in Chapter 7.

TIP: If you or your developers do elect to create a utility class to encapsulate common database functionality, you should consider marking the class as sealed (`NotInheritable` in VB) and giving it a private constructor. In this way, other developers can neither derive from the class nor create public instances of it. All the listings in this section can be thought of as methods in such a data-access utility class.

Listing 5–1 *Creating a SQLCE Database.* This method shows how to create a SQLCE database on the device using the `SqlCeEngine` class.

```
Public Shared Function CreateDb(ByVal filePath As String) As Boolean
   ' Delete and create the database

   Try
      If File.Exists(filePath) Then
          File.Delete(filePath)
      End If
   Catch e As Exception
      _lastException = e
      MsgBox("Could not delete the existing " & filePath, _
        MsgBoxStyle.Critical)
      Return False
   End Try

   Dim eng As SqlCeEngine
   Try
       eng = New SqlCeEngine("Data Source=" & filePath)
       eng.CreateDatabase()
       Return True
   Catch e As SqlCeException
       _lastException = e
       LogSqlError("CreateDb",e)
       MsgBox("Could not create the database at " & filePath, _
         MsgBoxStyle.Critical)
       Return False
   Finally
       eng.Dispose()
   End Try
End Function
```

In this case you'll notice that the `CreateDb` method first attempts to delete the database if it exists; it then passes the path to the database to the constructor of `SqlCeEngine` before calling the `CreateDatabase` method. The connection string need only consist of the `Data Source` attribute, and the `Provider` attribute will be defaulted to `Microsoft.SQLSERVER.OLEDB.CE.2.0`.[8] Other attributes may also be used, as discussed later in the chapter.

If an exception is found, the exception is placed in a private variable called `_lastException` that is exposed as a read-only shared property of the class. In this way the caller can optionally access full information about the exception that occurred. The database error is also logged using a custom method. To use this method the calling code would look like the following (assuming the method was placed in the `Atomic.SqlCeUtils` class):

```
If Atomic.SqlCeUtils.CreateDatabase(FileSystem.DocumentsFolder & _
   "\Personal\mydb.sdf") Then
    ' Go ahead and create some tables
End If
```

Note that the calling code uses the `FileSystem` class shown in Listing 3-5 to retrieve the My Documents folder on the device and then creates the database in the Personal folder.

NOTE: Databases may also be created using the `CREATE DATBASE` DDL statement when already connected to a different database. This statement also supports password protecting and encrypting the database, as discussed later in the chapter.

To create objects within a database, the application must first create a connection with the `SqlCeConnection` object. This is easily accomplished by passing the same connection string used to initialize the `SqlCeEngine` object in Listing 5-1 to the constructor of `SqlCeConnection` and calling the `Open` method as follows:

```
Dim cnCE As New SqlCeConnection(dbConnect)
cnCE.Open()
```

[8] This differs from ADOCE used in eVB, where omitting the `Provider` attribute assumes the `CEDB` provider and not `SQLCE`.

As you would expect, the previous snippet may throw a `SqlCeException` on either line if the connection string is malformed or the database is already open or does not exist. For this reason the opening of a connection should also be wrapped in a `Try-Catch` block.

KEY POINT

More important, as SQLCE supports only one concurrent connection (unlike SQL Server 2000) because Windows CE is a single-user operating system, the connection object is usually obtained early in the run of the application and persisted in a variable until the application closes. It is therefore important to ensure that the connection eventually gets closed so that other applications (for example, the Query Analyzer) may connect to the database.

After creating a connection, DDL statements can be executed against the connection to create the appropriate tables and indexes. Each DDL statement must be encapsulated in a `SqlCeCommand` object and executed with the `ExecuteNonQuery` method. However, if the application requires that multiple statements be executed (to create several tables and their indexes, for example), it is possible to create a utility function to read the SQL from a resource file deployed with the application. This is accomplished by adding a text file to SDP and setting its Build Action property in the Properties window to Embedded Resource. Then the resource file can be populated with `CREATE` and `ALTER` statements, like those shown below, to create a table to hold batting statistics and add a primary key and an index.

```
CREATE TABLE Batting (Id int NOT NULL, LastName nvarchar(50),
 FirstName nvarchar(50),Year smallint NOT NULL,Team nchar(3),
 G smallint NULL,AB smallint NULL,R smallint NULL ,
 H smallint NULL,"2B" smallint NULL,"3B" smallint NULL ,
 HR smallint NULL,RBI smallint NULL);
ALTER TABLE Batting ADD CONSTRAINT pk_batting PRIMARY KEY (Id, Year);
CREATE INDEX idx_bat_team ON Batting (Year, Team ASC);
```

When the project is built, the file will then be compiled as a resource in the assembly and deployed to the device.

To read the resource script and execute its DDL, a method like that shown in Listing 5-2 can be written.

Listing 5–2 *Running a SQL Script*. This method reads from a resource file and executes all the commands found therein. Note that none of the commands may use parameters.

```
Public Shared Function RunScript(ByVal scriptName As String, _
  ByVal cn As SqlCeConnection) As Boolean
```

```
   ' Perform a simple execute non query
   Dim closeIt As Boolean = False
   Dim resource As Stream

   Try
       Resource = _
        [Assembly].GetExecutingAssembly().GetManifestResourceStream( _
        scriptName))
       Dim sr As New StreamReader(resource)
       Dim script As String = sr.ReadToEnd()
       Dim commands() As String
       commands = script.Split(";"c)

       ' Open the connection if closed
       If cn.State = ConnectionState.Closed Then
          cn.Open()
          closeIt = True
       End If

       Dim cm As New SqlCeCommand()
       cm.Connection = cn
       Dim s As String
       For Each s In commands
          If s <> "" Then
              cm.CommandText = s
              cm.ExecuteNonQuery()
          End If
       Next

   ' Clean up
   Catch e As SqlCeException
        _lastException = e
        LogSqlError("RunScript",e)
        MsgBox("Could not run script " & scriptName, _
         MsgBoxStyle.Critical)
       Return False
   Catch e As Exception
        _lastException = e
        MsgBox("Could not run script " & scriptName, _
         MsgBoxStyle.Critical)
       Return False
   Finally
      If closeIt Then cn.Close()
   End Try
   Return True
End Sub
```

In Listing 5-2 you'll notice that the `GetManifestResourceStream` method of the `System.Reflection.Assembly` class is used to read the resource file into a `Stream` object. The `Stream` object is then read by a `StreamReader` and placed into a string variable. In this scenario, the method is expecting strings delimited with a semicolon and, therefore, creates an array of strings using the `Split` method. This is required in order to execute multiple statements, because SQLCE does not support batch SQL as SQL Server does. In other words, SQLCE can execute only one statement per `SqlCeCommand` object.

The method then proceeds to open the connection object if it is closed and create a `SqlCeCommand` object. The command object is then populated repeatedly in a loop, and each statement is executed using `ExecuteNon-Query`. You'll notice in the `Finally` block that the connection is closed only if it were opened by the method. The advantage to this technique is that it allows for looser coupling between the script and the code that executes it, so that the script can be changed without changing any code and the project recompiled and deployed. To put it all together, an application could use code like the following in its main form's `Load` event to create the database, connect to it, and create tables and indexes:

```
If Atomic.SqlCeUtils.CreateDatabase(FileSystem.DocumentsFolder & _
   "\Personal\mydb.sdf") Then
    ' Connect (cnCE is global)
   cnCE = New SqlCeConnection(dbConnect)
   cnCE.Open()
   ' Go ahead and create some tables
   If Atomic.SqlCeUtils.RunScript("firstrun.sql", cnCE) Then
      ' All is well and the database is ready
   End If
End If
```

Querying Data

As mentioned previously, SqlServerCe supports both the disconnected and connected programming models using the `DataSet` and data reader that were discussed in Chapter 4. Unfortunately, unlike in SQL Server 2000, SQLCE does not support stored procedures. As a result, developers will need to formulate SQL within the application and submit it to the database engine (although SQLCE does support parameterized queries, as will be discussed later). Also, as mentioned previously, SQLCE does not support batch SQL, and so, multiple `SELECT` statements cannot be executed and their results cannot be automatically populated in multiple `DataTable` objects in a `DataSet` or through multiple result sets using the `NextResult` property of the `SqlCeDataReader`. However, developers can still create data-access helper methods that reduce the amount of code required by the caller. For example, the method in Listing 5-3 adds data to a `DataSet` based on the `SQL` passed to the method.

Listing 5–3 *Populating a Data Set.* This method adds data to a data set given the SQL statement and the connection object to use.

```
Public Shared Sub FillSimpleDataSet(ByVal ds As DataSet, _
   ByVal sql As String, ByVal cn As SqlCeConnection, _
   ByVal acceptChanges As Boolean)

   Try
       Dim cm As New SqlCeCommand(sql, cn)
       Dim da As New SqlCeDataAdapter(cm)
       da.AcceptChangesDuringFill = acceptChanges

       da.MissingMappingAction = MissingMappingAction.Passthrough
       da.MissingSchemaAction = MissingSchemaAction.AddWithKey

       da.Fill(ds)
   Catch e As SqlCeException
       LogSqlError("FillSimpleDataSet",e)
       Throw New SqlCEUtilException( _
         "Could not fill dataset for: " & sql, e)
   End Try
End Function
```

You'll notice that in Listing 5-3 an existing connection object is used and that the caller determines whether `AcceptChangesDuringFill` is set to True or False to determine if the newly added rows are treated as new rows (with their `RowState` property set to `Added`) or as unmodified rows. In this case the connection object needn't be opened explicitly because the `SqlCe-DataAdapter` will open it if it is not already open. The `MissingMapping-Action` and `MissingSchemaAction` properties are also set to allow the data adapter to create any missing tables or columns in the `DataSet` and to add primary key information if available. Obviously, this method would not be useful if more sophisticated table mappings were required.[9] If any errors

[9] See Chapter 12 of *Teach Yourself ADO.NET in 21 Days,* by Dan Fox, for a complete explanation of how data adapters use table and column mappings.

occur, a custom exception of type `SqlCeUtilException` inherited from `ApplicationException` is thrown.

NOTE: Creating custom exception classes like `SqlCeUtilException` in Listing 5-3 that can be used to encapsulate application-specific messages and custom methods and properties is a good strategy. The original exception can then be chained to the custom exception using the `InnerException` property. This technique of exception wrapping, or chaining, allows the application to add specific messages at multiple levels in the call stack.

Data readers can similarly be created to stream through the results from a table as shown in Listing 5-4.

Listing 5–4 *Creating a Data Reader.* This method creates and returns a `SqlCeDataReader` given a SQL statement and a connection object.

```
Public Shared Function ExecDataReader(ByVal sql As String, _
   ByVal cn As SqlCeConnection) As SqlCeDataReader

    Try
         ' Create the command
         Dim cm As New SqlCeCommand(sql, cn)
         If cn.State = ConnectionState.Closed Then
             cn.Open()
         End If

         ' Execute data reader
         Dim dr As SqlCeDataReader
         dr = cm.ExecuteReader()
         Return dr
    Catch e As SqlCeException
         LogSqlError("ExecDataReader",e)
         Throw New SqlCEUtilException( _
           "Could not execute data reader for :" & sql, e)
    End Try
End Function
```

In Listing 5-4 the method creates a command object and associates it with the connection passed into the method. In this case the method must also open the connection if it is not already open before executing the data reader and returning it. Note that although the `ExecuteReader` method supports the `CloseConnection` and other command behaviors, it is not used because typically a single global database connection remains open for the lifetime of the application.

A caller would then use the method as follows:

```
Dim dr As SqlCeDataReader
dr = SqlCeUtils.ExecDataReader( _
  "SELECT * FROM Batting WHERE Id = 660", cnCE)

Do While dr.Read()
   ' Process the data
Loop
dr.Close()
```

Although not shown in this listing, it is also interesting to note that unlike the SqlClient provider, the SqlServerCe provider does support multiple data readers on the same open connection object. In other words, developers needn't close the `SqlCeDataReader` before using the connection to execute another command. Again, this is the case because SQLCE supports only a single concurrent connection.

One of the most interesting new features of SQLCE is the inclusion of parameterized queries. Using parameterized queries, developers can simply populate `SqlCeParameter` objects associated with a `SqlCeCommand`, rather than having to manually concatenate parameters into a single string. In addition, parameterized queries are recommended for performance reasons. However, unlike SqlClient, SQLCE supports only positional parameters, and the parameters must be defined in the SQL statement using a question mark. In other words, developers must declare a `SqlCeParameter` object for each question mark in the SQL statement so that the `SqlCeCommand` object can perform the substitution at runtime. For example, in order to execute the query shown above as a parameterized query, a developer could do the following:

```
Dim dr As SqlCeDataReader
Dim cm As New SqlCeCommand("SELECT * FROM Batting WHERE Id = ?", cnCE)
cm.Parameters.Add(New SqlCeParameter("@Id", SqlDbType.Int))
cm.Parameters(0).Value = 660

dr = SqlCeUtils.ExecDataReader()
```

In this case, although the parameter was referenced by its ordinal, it could alternatively have been referenced by its name (`@Id`).

In a helper or utility class, the creation of parameter objects and their association with a command object can be handled by a structure and private method like that shown in Listing 5-5.

Listing 5–5 Listing 5-5: *Automating Parameterized Queries.* This structure and method can be used to create and attach parameters automatically to a `SqlCeCommand` object.

```
Public Structure ParmData
    Public Name As String
    Public Value As Object
    Public DataType As SqlDbType

    Public Sub New(ByVal name As String, ByVal dataType As SqlDbType, _
      ByVal value As Object)
        Me.Name = name
        Me.DataType = dataType
        Me.Value = value
    End Sub
End Structure

Private Shared Function PopulateCommand(ByVal sql As String, _
   ByVal parms As ArrayList, ByVal cn As SqlCeConnection) _
   As SqlCeCommand

    Dim cm As New SqlCeCommand(sql, cn)
    cm.CommandType = CommandType.Text

    ' Populate parameters
    Dim p As Object
    For Each p In parms
        Dim p1 As ParmData = CType(p, ParmData)
        cm.Parameters.Add( _
          New SqlCeParameter(p1.Name, p1.DataType, p1.Value))
    Next

    Return cm

End Function
```

In Listing 5-5 you'll notice that the private PopulateCommand method accepts an ArrayList of ParmData objects as a parameter and uses it to populate a SqlCeCommand created from the SQL statements and SqlCe-Connection object passed in as well.[10] With this technique an overloaded version of the method in Listing 5-4 can be created to accept parameterized SQL, as shown in Listing 5-6.

Listing 5–6 *Creating a Data Reader with Parameters.* This method creates and returns a SqlCeDataReader given a SQL statement, parameters, and a connection object.

```
Public Shared Function ExecDataReader(ByVal sql As String, _
      ByVal cn As SqlCeConnection, _
      ByVal parms As ArrayList) As SqlCeDataReader

    Try
         ' Create the command
         Dim cm As SqlCeCommand = Me.PopulateCommand(sql, parms, cn)
         If cn.State = ConnectionState.Closed Then
             cn.Open()
         End If

         ' Execute data reader
         Dim dr As SqlCeDataReader
         dr = cm.ExecuteReader()
         Return dr
    Catch e As SqlCeException
         LogSqlError("ExecDataReader",e)
         Throw New SqlCEUtilException( _
           "Could not execute data reader for :" & sql, e)
    End Try
End Function
```

At this point the caller need create only the ParmData objects, specifying the appropriate data type, and place them in an ArrayList before passing them to ExecDataReader, as shown in this snippet:

[10] Since SQLCE does not support stored procedures, the StoredProcedure CommandType is also not supported.

```
Dim dr As SqlCeDataReader
Dim sql As String = "SELECT * FROM Batting WHERE Id = ?"
Dim parms As New ArrayList()

parms.Add(New ParmData("id", SqlDbType.Int, 660))
dr = SqlCeUtils.ExecDataReader(sql, cnCE, parms)
```

Using Indexes

KEY POINT

Perhaps the biggest difference between the SqlClient provider and the SqlServerCe provider is the inclusion of index seeks using data readers in SqlServerCe. Using this technique allows developers to write code that performs better than issuing `SELECT` statements with `WHERE` clauses. This is the case because the SQLCE query processor must compile, optimize, and generate a query plan for each query, while performing the index seek directly avoids these costly steps. The caveat is that this works only against single tables, and the table must of course have an index. As a result, for complex queries developers will likely want to rely on the query processor.

NOTE: In one example documented on Microsoft's SQLCE Web site and referenced in the "Related Reading" section, using an index seek versus the query processor improved performance by a factor of 20 or greater.

For example, consider the scenario where a developer wanted to retrieve the statistics for a specific team and year from the batting table created in Listing 5-2, and it is known that the year will be in the range from 1980 to 1989. The batting table has a composite index on the Year and Team columns, and so a method like that shown in Listing 5-7 can be written to return a `SqlCeDataReader` positioned on the correct row.

Listing 5–7 *Seeking a Row Using an Index.* This method creates and returns a `SqlCeDataReader` positioned on the appropriate row for a given set of index values.

```
public static SqlCeDataReader ExecTeamReader(SqlCeConnection cn,
  string team, int year)
{
   SqlCeCommand cmd = new SqlCeCommand("Batting",cn);
   cmd.CommandType  = CommandType.TableDirect;
   if (cn.State == ConnectionState.Closed)
   {
      cn.Open();
   }

   // Index contains Year and Team
   cmd.IndexName = "idx_bat_team";

   object[] start = {1980, 1989};
   object[] end = {null, null};
   cmd.SetRange(DbRangeOptions.InclusiveStart |
     DbRangeOptions.InclusiveEnd, start, end);

   Try
   {
      SqlCeDataReader rdr = cmd.ExecuteReader();
      rdr.Seek(DbSeekOptions.AfterEqual, year, team);
      return rdr;
   }
   Catch (SqlCeException e)
   {
     LogSqlError("ExecTeamReader",e);
     // Throw a custom exception
     return null;
   }
}
```

You'll notice in Listing 5-7 that the `SqlCeCommand` must have its `CommandText` property set to the name of the table to search and that the `CommandType` must be set to `TableDirect`. The name of the index is then set using the `IndexName` property. Although it is not required, this listing also shows that the range of values searched can be restricted by passing arrays of start and end values to the `SetRange` method. The `DbRangeOptions` enumeration determines how the `Seek` method uses the start and end values. After opening the data reader using `ExecuteReader`, its `Seek` method is then called with a value from the `DbSeekOptions` enumeration. This value specifies which row if any is to be returned. In this case, `AfterEqual` is used and if a row is not found, the first row after the index range will be the one pointed to by the data reader. Alternatively, if `FirstEqual` is used, the `Seek` method will throw a `SqlCeException` if a row cannot be located.

A caller can then invoke the method to position a data reader at the statistics for the 1984 Chicago Cubs as follows:

```
SqlCeDataReader dr = SqlCeUtils.ExecTeamReader(cnCE,"CHN",1984);
```

Modifying Data

Inserting, updating, and deleting data in SQLCE are not handled any differently than they are using the SqlClient provider, with the exception, of course, that SQLCE does not support stored procedures. In other words developers may use the `SqlCeDataAdapter` to modify data in an underlying base table utilizing the table and column mappings collections and then invoking the `Update` method of the data adapter. Developers may also execute command objects directly. In either case parameterized queries are used and, in fact, are required for use with the `SqlCeDataAdapter`.

For example, to insert a new row into the Batting Table, the method shown in Listing 5-8 could be written to return the command object used in either scenario.

Listing 5–8 *Inserting Data with a Command.* This method creates and returns a `SqlCeCommand` to insert new rows into the Batting Table.

```
Public Shared Function GetBattingCmd(cnCE As SqlCeConnection, _
   trans As SqlCeTransaction) As SqlCeCommand

   Dim sql As String = "INSERT INTO Batting (Id, LastName, " & _
     "FirstName, Year, Team, G, AB, R, H, ""2B"", ""3B"", " & _
     "HR, RBI) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)"

   battingCmd = New SqlCeCommand(sql)
   battingCmd.CommandType = CommandType.Text
   If Not trans Is Nothing Then
       battingCmd.Transaction = trans
   End If

   battingCmd.Parameters.Add(New SqlCeParameter("Id", _
     SqlDbType.NVarChar, 9, "Id"))
   battingCmd.Parameters.Add(New SqlCeParameter("LastName", _
     SqlDbType.NVarChar, 50, "LastName"))
   battingCmd.Parameters.Add(New SqlCeParameter("FirstName", _
     SqlDbType.NVarChar, 50, "FirstName"))
   battingCmd.Parameters.Add(New SqlCeParameter("Year", _
     SqlDbType.SmallInt, 4, "Year"))
```

```
    battingCmd.Parameters.Add(New SqlCeParameter("Team", _
      SqlDbType.NVarChar, 3, "Team"))
    battingCmd.Parameters.Add(New SqlCeParameter("G", _
      SqlDbType.SmallInt, 4, "G"))
    battingCmd.Parameters.Add(New SqlCeParameter("AB", _
      SqlDbType.SmallInt, 4, "AB"))
    battingCmd.Parameters.Add(New SqlCeParameter("R", _
      SqlDbType.SmallInt, 4, "R"))
    battingCmd.Parameters.Add(New SqlCeParameter("H", _
      SqlDbType.SmallInt, 4, "H"))
    battingCmd.Parameters.Add(New SqlCeParameter("2B", _
      SqlDbType.SmallInt, 4, "2B"))
    battingCmd.Parameters.Add(New SqlCeParameter("3B", _
      SqlDbType.SmallInt, 4, "3B"))
    battingCmd.Parameters.Add(New SqlCeParameter("HR", _
      SqlDbType.SmallInt, 4, "HR"))
    battingCmd.Parameters.Add(New SqlCeParameter("RBI", _
     SqlDbType.SmallInt, 4, "RBI"))

    return battingCmd
End Function
```

The `GetBattingCmd` static method could then be used by a caller to retrieve the appropriate command before populating the parameters with values manually through code or by setting it to the `InsertCommand` property of the `SqlCeDataAdapter`.

Although not typically recommended for production scenarios with the SqlClient provider,[11] the command builder included with SqlServerCe (`SqlCeCommandBuilder`) can be used in place of code like that shown in Listing 5-7. This is due to the fact that SQLCE is single user and runs in process with the application; therefore, an extra round trip isn't as costly in terms of performance. In any case, it can be used simply by passing the `SqlCeDataAdapter` to the constructor of the command builder:

```
Dim cb as New SqlCeCommandBuilder(da)
```

When needed,[12] the command builder will then build the insert, update, and delete commands based on the `SELECT` statement exposed in

[11] Using the `SqlCommandBuilder` object always engenders one extra trip to the database server so that the command builder can determine column names and data types.
[12] When the RowUpdating events fire on the `SqlCeDataAdapter` object.

the `CommandText` property of the `SelectCommand`. Note that just as with the SqlClient provider, the `SELECT` statement used by the data adapter mustn't be complex (contain aggregates columns and joins) and must return at least one primary key or unique column, or an exception will be thrown.

NOTE: If the `SELECT` statement changes or the connection or transaction associated with the command changes, a developer can call the `RefreshSchema` method of the `SqlCeCommandBuilder` to regenerate the insert, update, and delete commands.

Handling Transactions

Just like SqlClient, the SqlServerCe provider supports transactions, or the ability to group a series of data modifications in an atomic operation. This is useful if an application needs to update two tables, with the requirement that if one of the updates fails, they both fail (a parent/child relationship, for example).

This is accomplished through the `BeginTransaction` method of the `SqlCeConnection` object, which spawns a `SqlCeTransaction` object used to control the outcome (`Commit` or `Rollback`) of the transaction. For example, the following code snippet uses the `GetBattingCmd` method shown in Listing 5-8 and the `GetPitchingCmd` method (not shown) to execute two commands in a single transaction:

```
SqlCeTransaction trans = null;
Try
{
    trans = cnCE.BeginTransaction();
    SqlCeCommand bat = SqlCeUtils.GetBattingCmd(cnCE, trans);
    SqlCeCommand pitch = SqlCeUtils.GetPitchingCmd(cnCE, trans);

    // populate the commands with the new values

    bat.ExecuteNonQuery();
    pitch.ExecuteNonQuery();
    trans.Commit();
}
catch (SqlCeException e)
{
  if (trans != null) {trans.Rollback();}
  LogSqlError("MyMethod",e);
  // most likely throw a custom exception
}
```

You'll notice here that the `GetBattingCmd` and `GetPitchingCmd` methods accept a transaction as the second argument. Referring to Listing 5-8, this transaction, if instantiated, is associated with the command object using its `Transaction` property.

However, the transactional behavior of SQLCE differs from SQL Server 2000, and so developers must be aware of four differences. First, SQLCE only supports an isolation level of `ReadCommitted`, which exclusively locks data being modified in a transaction. As a result, the `IsolationLevel` property of `SqlCeTransaction` can only be set to the `ReadCommitted` value of the `IsolationLevel` enumeration. Second, SQLCE supports nested transactions, but only up to five levels. Third, SQLCE holds an exclusive lock on any table that has been modified in a transaction.[13] This means that any attempt to access any data from the table outside the transaction, while it is pending, will result in an exception. Fourth, if a data reader is opened within a transaction, the data reader will automatically be closed if the transaction is rolled back. If the transaction commits, the data reader can still be used.

Abstracting .NET Data Providers

As discussed in Chapter 4, applications written with VS .NET 2003 and the Compact Framework can access a SQL Server 2000 server remotely using the SqlClient .NET Data Provider. And, as discussed in this chapter, applications can store data locally in SQLCE using the SqlServerCe .NET Data Provider. In some scenarios, an application may wish to do both, for example, by accessing the remote SQL Server when connected to a corporate LAN via a direct connection, WLAN, or WAN and accessing SQLCE when disconnected.

In these instances, developers can take advantage of the object-oriented nature of the Compact Framework to write code that can be used with either provider. Doing so allows a greater level of code reuse and easier porting of code from the desktop Framework to the Compact Framework. Abstracting data providers is possible since, as mentioned in Chapter 4, all .NET Data Providers are implemented using the same underlying base classes and interfaces. These include the interfaces `IDbConnection`, `IDbCommand`, `IDataRecord`, `IDataParameter`, `IDbDataParameter`, `IDataParameterCollection`, `IDataReader`, `IDataAdapter`, `IDbDataAdapter`, and `IDbTransaction`, along with the `DataAdapter` and `DbDataAdapter` classes,

[13] As opposed to row- and page-level locks used by SQL Server 2000.

Interfaces or Base Classes?

The Compact Framework relies on both interfaces and base classes to allow code reuse through inheritance and polymorphism. Simply put, interfaces (typically prefixed with an "I") enable interface inheritance by allowing a class to implement a set of method signatures defined in the interface. When using interface inheritance, the class implementing the interface must include all of the method signatures from the interface but must implement the functionality of the methods itself. Using a base class, a class may use implementation inheritance to inherit both the method signatures and the implementation (the code) in the base class. The derived class may then override the methods of the base class to augment or replace the base class code.

Both techniques are useful, and, as you would imagine, interface inheritance is used when a variety of different classes needs to implement the same behavior (methods) in different ways, while implementation inheritance is used when classes form a natural hierarchy represented with an "is a" relationship (Employee is a Person). Both can be used together in the same class, although in the Compact Framework, implementation inheritance is restricted to a single inheritance, meaning that each class may inherit only from one base class.

Using both techniques, developers can write polymorphic (literally "multiform") code by targeting the reference variables in their code at the interfaces and base classes, rather than at the class inheriting from the interface or base class (often called the concrete class). In this way, at runtime the reference variables may actually refer to instances of any of the concrete classes in the inheritance relationship, thereby allowing the code to work in a variety of scenarios.

among others, found in the `System.Data` and `System.Data.Common` namespaces.

One technique for abstracting the data provider used is to implement the Abstract Factory design pattern documented in the book *Design Patterns*, as noted in the "Related Reading" section at the end of the chapter. This design pattern allows code to create families of related classes without specifying their concrete classes at design time. In this case, the family of related classes comprises the classes that make up a data provider, including connection, command, data adapter, and parameter.

Although it is possible to use the Abstract Factory pattern as documented in *Design Patterns*, a slight variant of the pattern, shown in Listing 5-9, is flexible because it allows the data provider to be specified in a shared method of the Abstract Factory class rather than having to be hard-coded at the creation of the class at runtime.

Listing 5–9 *Implementing the Abstract Factory Pattern.* This listing shows the code necessary to implement the Abstract Factory pattern so that polymorphic code can be written to use either of the data providers that ships with the Compact Framework. Note that the `SqlClientFactory` class is not shown.

```
Public Enum ProviderType
    SqlClient = 0
    SqlServerCe = 1
End Enum

Public MustInherit Class ProviderFactory

    Public Shared Function CreateFactory( _
      ByVal provider As ProviderType) As ProviderFactory
        If provider = ProviderType.SqlClient Then
            Return New SqlClientFactory
        Else
            Return New SqlServerCeFactory
        End If
    End Function

    Public MustOverride Function CreateConnection( _
      ByVal connect As String) As IDbConnection
    Public MustOverride Overloads Function CreateDataAdapter( _
      ByVal cmdText As String, _
      ByVal connection As IDbConnection) As IDataAdapter
    Public MustOverride Overloads Function CreateDataAdapter( _
      ByVal command As IDbCommand) As IDataAdapter
    Public MustOverride Overloads Function CreateParameter( _
      ByVal paramName As String, _
      ByVal paramType As DbType) As IDataParameter
    Public MustOverride Overloads Function CreateParameter( _
      ByVal paramName As String, _
      ByVal paramType As DbType, _
      ByVal value As Object) As IDataParameter
    Public MustOverride Function CreateCommand( _
       ByVal cmdText As String, _
       ByVal connection As IDbConnection) As IDbCommand

End Class

Public NotInheritable Class SqlServerCeFactory
   Inherits ProviderFactory
```

```
    Public Overrides Function CreateConnection( _
     ByVal connect As String) As IDbConnection
        Return New SqlCeConnection(connect)
    End Function

    Public Overloads Overrides Function CreateDataAdapter( _
      ByVal cmdText As String, _
      ByVal connection As IDbConnection) As IDataAdapter
        Return New SqlCeDataAdapter(cmdText, _
         CType(connection, SqlCeConnection))
    End Function

    Public Overloads Overrides Function CreateDataAdapter( _
      ByVal command As IDbCommand) As IDataAdapter
        Return New SqlCeDataAdapter(CType(command, SqlCeCommand))
    End Function

    Public Overloads Overrides Function CreateParameter( _
      ByVal paramName As String, _
      ByVal paramType As DbType) As IDataParameter
        Return New SqlCeParameter(paramName, paramType)
    End Function

    Public Overloads Overrides Function CreateParameter( _
      ByVal paramName As String, _
      ByVal paramType As DbType, _
      ByVal value As Object) As IDataParameter
        Dim parm As New SqlCeParameter(paramName, paramType)
        parm.Value = value
        Return parm
    End Function

    Public Overrides Function CreateCommand(ByVal cmdText As String, _
      ByVal connection As IDbConnection) As IDbCommand
        Return New SqlCeCommand(cmdText, _
         CType(connection, SqlCeConnection))
    End Function

End Class
```

As you'll notice in Listing 5-9, the `ProviderType` enumeration identifies which factory classes are available. The heart of the listing is the abstract (marked as `MustInherit` in VB and `abstract` in C#) `ProviderFactory` class. This class implements a shared method to create an instance of a concrete `ProviderFactory` class, along with a set of method signatures marked with the `MustOverride` keyword. This keyword ensures that the class inheriting from `ProviderFactory` will override the methods to provide an implementation. The `SqlServerCeFactory` class inherits from `ProviderFactory`, overriding the base class methods and returning instances of the appropriate SqlServerCe objects (`SqlCeConnection`, `SqlCeDataAdapter`, and so forth). Note that the methods of the `ProviderFactory` class return references to the interfaces implemented by data providers discussed earlier. This is the key to enabling the writing of polymorphic code. Although not shown in the listing due to space constraints, there would, of course, be a corresponding factory class for the SqlClient provider that also inherits from `ProviderFactory`.

NOTE: To extend the `ProviderFactory` to support new providers (for example, one for Sybase SQL Anywhere Studio), a developer need only create a factory class that inherits from `ProviderFactory`. He or she would also likely want to extend the `ProviderType` enumeration and the `CreateFactory` method.

To use the `ProviderFactory` class, a caller need only instantiate the correct class using the shared method, as follows:

```
Dim pf As ProviderFactory
If CheckForNetworkConn() Then
   ' Go remote
    pf = ProviderFactory.CreateFactory(ProviderType.SqlClient)
Else
   ' Go local
    pf = ProviderFactory.CreateFactory(ProviderType.SqlServerCe)
End If
```

In this snippet the `CheckForNetworkConn` method shown in Chapter 4 is used first to determine if a network connection is available; if so, it uses SqlClient and if not, SqlServerCe. Of course, the value for the `ProviderType` enumeration could also easily be read from a configuration file or passed into the method as a variable to allow for flexibility.

Once the concrete `ProviderFactory` has been created, it can be passed into methods like those shown in the listings in this chapter so that the methods can be used against either provider. For example, the `ExecDataReader` method shown in Listing 5-4 could then be rewritten as shown in Listing 5-10.

Listing 5–10 *Using the Abstract Factory Pattern.* This method shows the `ExecDataReader` method rewritten to use an instance of the `ProviderFactory` class to enable provider-independent database access.

```
Public Shared Function ExecDataReader(ByVal pf As ProviderFactory, _
   ByVal sql As String, ByVal cn As IDbConnection) As IDataReader

    Try
         ' Create the command
         Dim cm As IDbCommand = pf.CreateCommand(sql, cn)
         If cn.State = ConnectionState.Closed Then
             cn.Open()
         End If

         ' Execute data reader
         Dim dr As IDataReader
         dr = cm.ExecuteReader()
         Return dr
    Catch e As Exception
         LogSqlError("ExecDataReader",e)
         Throw New Exception( _
           "Could not execute data reader for :" & sql, e)
    End Try
End Function
```

Note that because the `ExecDataReader` method can now be used with either provider, it returns an object that implements the `IDataReader` interface and accepts an `IDbConnection` object, rather than the concrete types for SqlServerCe. In addition, the creation of the `SqlCeCommand` object has been replaced with a call to the `CreateCommand` method of the `ProviderFactory`, and the reference to the `SqlCeException` object in the `Catch` block has been replaced with the generic `Exception` object.[14]

[14] An alternative and more dynamic approach to creating an abstract factory class using the runtime type creation methods of the desktop Framework and Compact Framework can be found in Chapter 18 of *Teach Yourself ADO.NET in 21 Days*.

Administering SQLCE

Because SQLCE is separate from the Compact Framework, it must be administered separately. The administration tasks take the form of security administration, database maintenance, and installation and deployment.

Security

KEY POINT

As with any database, it is important that the data in SQLCE be secure. This is particularly the case because the device on which SQLCE is running is inherently mobile and can easily fall into the hands of someone who is not the intended user. As a result, it is important that Compact Framework applications be able to present an authentication dialog to users before providing access to the data and that the data itself can be encrypted on the device.

NOTE: Keep in mind that because Windows CE is a single-user operating system, there is no support in SQLCE for individual user authentication or permissions; and, in fact, the `syslogins`, `sysprotects`, and `sysusers` system tables present in SQL Server 2000 to support these functions are not included in SQLCE. Any user who can open the database has full permissions. Along the same lines, the Windows CE file system does not support permissions; so, there is no inherent protection for the .sdf file.

SQLCE supports these requirements by offering both password protection for the entire database file and encryption for the entire file using a 128-bit key.

Password Protection

Password protecting a SQLCE database can be done only when the database is created or compacted (as discussed in the next section) and can be done with either the `CreateDatabase` method of the `SqlCeEngine` object or the `CREATE DATABASE` DDL statement.

When using the `CreateDatabase` method, the password attribute is simply appended to the connection string passed into the constructor of the `SqlCeEngine` class. As a result, the `CreateDb` method shown in Listing 5-1 could be altered as shown in the following snippet to accept a password of up to 40 characters to use when creating the database.

```
Public Shared Function CreateDb(ByVal filePath As String, _
  ByVal pwd As String) As Boolean

   ' Code ommitted for brevity

   Dim eng As SqlCeEngine
   Try
       eng = New SqlCeEngine("Data Source=" & filePath & _
        ";password= & pwd)
       eng.CreateDatabase()
       Return True
   Catch e As SqlCeException
       ' Code ommitted for brevity
   End Try
End Function
```

Once the password has been created, there is no way to recover it; however, the password can be changed by compacting the database, as will be discussed later in this section.

If the application is executing DDL to create a database, a `CREATE DATABASE` statement like the following can be issued:

```
CREATE DATABASE 'mydb.sdf' DATABASEPASSWORD 'sdfg53$h'
```

Encryption

Just as with password protection, encrypting a SQLCE database can be accomplished with the `CreateDatabase` method, the process of compacting, or the `CREATE DATABASE` DDL statement.

To encrypt using `CreateDatabase`, the `encrypt database` attribute needs to be added to the connection string in addition to the password, as shown in the following snippet, where the `CreateDb` method from Listing 5-1 is once again modified to support an argument to determine if the database should be encrypted. Note, however, that the attribute needn't be provided when the database is opened.

```
Public Shared Function CreateDb(ByVal filePath As String, _
  ByVal pwd As String, ByVal encrypt As Boolean) As Boolean

   ' Code ommitted for brevity

   Dim eng As SqlCeEngine
```

```
    Try
        Dim connect = "Data Source=" & filePath & _
         ";password= & pwd
        If encrypt Then
            connect &= ";encrypt database=TRUE"
        End If
        eng = New SqlCeEngine(connect)
        eng.CreateDatabase()
        Return True
    Catch e As SqlCeException
        ' Code ommitted for brevity
    End Try
End Function
```

KEY POINT

The password attribute must be included because SQLCE uses the MD5[15] hashing algorithm to create the 128-bit key required by the RC4[16] algorithm used to encrypt the database. For this reason it is important that the password chosen be of a reasonable length to avoid easy cracking by hackers.[17] Although it would be cumbersome to force users to input 40-character passwords, passwords of at least 8 characters (including letters, numbers, and at least once special character) should suffice to offer a reasonable amount of protection. Changing passwords periodically via compaction is also a good strategy because it moves the target for any potential hacker.

To encrypt the database file using the `CREATE DATABASE` statement, the `ENCRYPTION ON` clause is used as follows:

```
CREATE DATABASE 'mydb.sdf' DATABASEPASSWORD 'sdfg53$h' ENCRYPTION ON
```

Database Maintenance

KEY POINT

As alluded to earlier, the `SqlCeEngine` class also supports the `Compact-Database` method, which can be used to compact and reclaim wasted space that collects in the database as data and objects are deleted and tables are

[15] A message-digest algorithm developed in 1991 by RSA Security.

[16] A symmetric encryption algorithm designed by RSA Security in 1987 and used in Secure Sockets Layer (SSL) and other commercial applications.

[17] Hackers can extract the hash value from the .sdf file and then run either a dictionary or a brute-force attack to discover the password. Longer passwords are recommended because the effort required in using a brute-force method increases exponentially. For example, two-character passwords take seconds to break, while eight-character passwords can require years.

reindexed. It is recommended that SQLCE databases be periodically compacted because this also leads to improved query performance through index reordering and the refreshing of statistics used by the query processor to generate execution plans.

Compacting a database can also be used to change the collating order,[18] encryption, or password for the database, as mentioned previously in this section. This method creates a new database and requires that the source database be closed and that the destination file not exist. It is also important to remember that because a copy is created, the device will need to have enough room to make the copy or an error will result.

Once again, it makes sense to wrap the `CompactDatabase` functionality in a method that checks for the existence of the source database and then automatically copies the destination back to the source when completed, as shown in Listing 5-11, which takes advantage of the `FileSystem` class in Listing 3-5 to create the temporary destination that is ultimately moved back to the original file name.

Listing 5–11 *Compacting a SQLCE Database.* This method compacts a database, reclaiming wasted space, and copies the newly created database back to the old name.

```
Public Shared Function CompactDb(ByVal filePath As String) As Boolean

   If Not File.Exists(filePath) Then
      MsgBox("Source database does not exist = " & filePath, _
        MsgBoxStyle.Critical)
      Return False
   End If

   Dim eng As SqlCeEngine
   Try
       eng = New SqlCeEngine("Data Source=" & filePath)
       eng.Compact("Data Source=" & _
        FileSystem.GetSpecialFolderPath(ceFolders.PERSONAL) & _
        "\temp000.sdf")
       File.Delete(filePath)
```

[18] If not specified in the `CREATE DATABASE` statement or the destination database connection string, the default collation assigned is `Latin1_General`. This collation uses Latin 1 General dictionary sorting rules, code page 1,252, and is case-insensitive and accent-insensitive. All databases in SQLCE are always case-sensitive and accent-insensitive. To see the available collations, see the Books Online for SQLCE.

```
        File.Move(FileSystem.GetSpecialFolderPath( _
          ceFolders.PERSONAL) & "\temp000.sdf", filePath)
    Catch e As Exception
        _lastException = e
        MsgBox("Could not compact the database at " & filePath, _
         MsgBoxStyle.Critical)
        Return False
    Finally
        eng.Dispose()
    End Try

    Return True

End Function
```

It should also be noted that SQLCE creates a temporary file each time the database engine is initialized and attempts to delete it when the engine terminates normally. This file is used for storing pages that exceed the SQLCE buffer cache, as well as interim results and tables used in queries. By default, the file is created in the Temp directory on the device, although its location can be specified using the `temp file directory` attribute of the connection string as shown here:

```
Dim connect = "Data Source=\mydb.sdf;temp file directory=\StorageCard"
Dim eng As New SqlCeEngine(connect)
```

This may be required if the need to store the temporary file on a storage card, rather than in RAM, arises. The file will grow the most when transactions and large `UPDATE` and `DELETE` statements are executed. However, keep in mind that accessing storage cards is typically slower than accessing RAM; so, query performance may suffer as a result.

Installation and Deployment

To use SQLCE in a solution, components must be installed both on the development machines as well on the device. Fortunately for Compact Framework developers, all the required SQLCE components are installed and configured with VS .NET 2003. This allows a developer to reference the `System.Data.SqlServerCe.dll` assembly from any SDP and begin coding against SQLCE.

When an SDP that accesses SQLCE is deployed to either an emulator or an actual device from VS .NET using the Build menu, two .cab files are automatically copied to the device and extracted. Which .cab files are deployed is determined by the processor type and version of Windows CE running on the device. They include a development-only time .cab (`Sqlce.dev.platform.`*`processor`*`.cab`) that contains Query Analyzer and error string files, as well as the .cab file that contains the SQLCE database engine (`Sqlce.platform.`*`processor`*`.cab`).

When an application is ready for final deployment, the SQLCE .cab file must be added to the deployment and extracted on the device, as discussed in Chapter 10. The amount of space required on the device varies with the platform and processor, but it ranges from 1 to 3MB.

NOTE: In order to use SQLCE to connect to SQL Server 2000 using RDA or replication, additional configuration steps must be undertaken on the server machine as discussed in Chapter 7.

Deploying a SQLCE Database

Finally, it's important to note that in many instances it is more efficient and reduces load on the database server to prebuild a SQLCE database and deploy it to the device, rather than forcing clients to perform an initial synchronization using RDA or replication, as discussed in Chapter 7. This benefit only increases as the number of deployed devices in a solution increases. For example, a field service solution could be initially deployed with parts lists and geographic data.

To prebuild a SQLCE database, a developer can write an administrative application that creates the database on the device or the emulator and pulls in the appropriate data using RDA. The database can then be copied back to the development machine using ActiveSync and included in a VS .NET project as a content file using the Properties window. In this way, the database will be deployed with the application, as discussed in Chapter 10. Although it would be a welcome addition, at this time there is no desktop- or server-based utility to allow developers to create and populate SQLCE databases.

Alternatively, and especially if the database is large, the database file can be distributed on CompactFlash memory and CompactFlash disk drives, both of which are supported by SQLCE.

What's Ahead

This chapter has discussed the need for robust data caching and how that need is addressed using SQL Server CE. However, mobile applications that cache data locally using XML or SQLCE also typically need to synchronize their data with back-end systems. This final essential architectural concept will be addressed in the following two chapters, which look at both primitive synchronization using ActiveSync and more complex synchronization using RDA and merge replication.

Related Reading

Microsoft SQL Serve CE 2.0 Web site, at www.microsoft.com/sql/CE/default.asp.

SQL Server CE case studies, at www.microsoft.com/sql/ce/productinfo/casestudies.asp. Many of these case studies involve using RDA or merge replication or both.

Xue, Song. "SQL Server 2000 Windows CE Edition 2.0 Query Processor Overview and Performance Tuning Approaches." *Microsoft TechNet* (October 2002), at www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/sql/maintain/Optimize/SSCEQPOP.asp.

Yao, Paul, and David Durant. "SQL Server CE: New Version Lets You Store and Update Data on Handheld Devices." *MSDN Magazine* (June 2001), at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsqlce/html/sqlce_secmodelscen20.asp.

Fox, Dan. *Teach Yourself ADO.NET in 21 Days*. Sams, 2002. ISBN 0-672-32386-9. See especially Chapter 18.

Gamma, Erich, et al. *Design Patterns*. Addison-Wesley, 1995. ISBN 0-201-63361-2. See p. 87 and following for a discussion of the Abstract Factory pattern.

Download the page for the Sybase Anywhere Studio .NET Data Provider from www.sybase.com.

Fox, Dan. "Protect Private Data with the Cryptography Namespaces of the desktop Framework." *MSDN Magazine* (June 2002), at http://msdn.microsoft.com/msdnmag/issues/02/06/Crypto/default.aspx.