UltraLite.NET™ User's Guide

Part number: DC50043-01-0900-01

Last modified: June 2003

Contents

About This Manual

Subject	This manual describes UltraLite.NET, which is part of the UltraLite Component Suite. With UltraLite.NET you can develop and deploy database applications to desktop computers, or handheld, mobile, or embedded devices.
Audience	This manual is intended for .NET application developers who wish to take advantage of the performance, resource efficiency, robustness, and security of an UltraLite relational database for data storage and synchronization.

SQL Anywhere Studio documentation

This book is part of the SQL Anywhere documentation set. This section describes the books in the documentation set and how you can use them.

The SQL Anywhere Studio documentation

The SQL Anywhere Studio documentation is available in a variety of forms: in an online form that combines all books in one large help file; as separate PDF files for each book; and as printed books that you can purchase. The documentation consists of the following books:

- **Introducing SQL Anywhere Studio** This book provides an overview of the SQL Anywhere Studio database management and synchronization technologies. It includes tutorials to introduce you to each of the pieces that make up SQL Anywhere Studio.
- **What's New in SQL Anywhere Studio** This book is for users of previous versions of the software. It lists new features in this and previous releases of the product and describes upgrade procedures.
- **Adaptive Server Anywhere Getting Started** This book is for people new to relational databases or new to Adaptive Server Anywhere. It provides a quick start to using the Adaptive Server Anywhere database-management system and introductory material on designing, building, and working with databases.
- **Adaptive Server Anywhere Database Administration Guide** This book covers material related to running, managing, and configuring databases and database servers.
- **Adaptive Server Anywhere SQL User's Guide** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- **Adaptive Server Anywhere SQL Reference Manual** This book provides a complete reference for the SQL language used by Adaptive Server Anywhere. It also describes the Adaptive Server Anywhere system tables and procedures.
- **Adaptive Server Anywhere Programming Guide** This book describes how to build and deploy database applications using the C, C++, and Java programming languages. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by those tools. It also describes the Adaptive Server Anywhere ADO.NET data provider.

- **Adaptive Server Anywhere Error Messages** This book provides a complete listing of Adaptive Server Anywhere error messages together with diagnostic information.
- **SQL Anywhere Studio Security Guide** This book provides information about security features in Adaptive Server Anywhere databases. Adaptive Server Anywhere 7.0 was awarded a TCSEC (Trusted Computer System Evaluation Criteria) C2 security rating from the U.S. Government. This book may be of interest to those who wish to run the current version of Adaptive Server Anywhere in a manner equivalent to the C2-certified environment.
- **MobiLink Synchronization User's Guide** This book describes how to use the MobiLink data synchronization system for mobile computing, which enables sharing of data between a single Oracle, Sybase, Microsoft or IBM database and many Adaptive Server Anywhere or UltraLite databases.
- **MobiLink Synchronization Reference** This book is a reference guide to MobiLink command line options, synchronization scripts, SQL statements, stored procedures, utilities, system tables, and error messages.
- **iAnywhere Solutions ODBC Drivers** This book describes how to set up ODBC drivers to access consolidated databases other than Adaptive Server Anywhere from the MobiLink synchronization server and from Adaptive Server Anywhere remote data access.
- **SQL Remote User's Guide** This book describes all aspects of the SQL Remote data replication system for mobile computing, which enables sharing of data between a single Adaptive Server Anywhere or Adaptive Server Enterprise database and many Adaptive Server Anywhere databases using an indirect link such as e-mail or file transfer.
- **SQL Anywhere Studio Help** This book includes the context-sensitive help for Sybase Central, Interactive SQL, and other graphical tools. It is not included in the printed documentation set.
- **UltraLite Database User's Guide** This book is intended for all UltraLite developers. It introduces the UltraLite database system and provides information common to all UltraLite programming interfaces.
- **UltraLite Interface Guides** A separate book is provided for each UltraLite programming interface. Some of these interfaces are provided as UltraLite components for rapid application development, and others are provided as static interfaces for C, C++, and Java development.

In addition to this documentation set, PowerDesigner and InfoMaker include their own online documentation.

Documentation formats

SQL Anywhere Studio provides documentation in the following formats:

- **Online documentation** The online documentation contains the complete SQL Anywhere Studio documentation, including both the books and the context-sensitive help for SQL Anywhere tools. The online documentation is updated with each maintenance release of the product, and is the most complete and up-to-date source of documentation.

  To access the online documentation on Windows operating systems, choose Start ➤ Programs ➤ SQL Anywhere 9 ➤ Online Books. You can navigate the online documentation using the HTML Help table of contents, index, and search facility in the left pane, as well as using the links and menus in the right pane.

  To access the online documentation on UNIX operating systems, see the HTML documentation under your SQL Anywhere installation.

- **Printable books** The SQL Anywhere books are provided as a set of PDF files, viewable with Adobe Acrobat Reader.

  The PDF files are available on the CD ROM in the *pdf_docs* directory. You can choose to install them when running the setup program.

- **Printed books** The complete set of books is available from Sybase sales or from eShop, the Sybase online store. You can access eShop by clicking How to Buy ➤ eShop at *http://www.ianywhere.com*.

Documentation conventions

This section lists the typographic and graphical conventions used in this documentation.

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords** All SQL keywords appear in upper case, like the words ALTER TABLE in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Placeholders** Items that must be replaced with appropriate identifiers or expressions are shown like the words *owner* and *table-name* in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Repeating items** Lists of repeating items are shown with an element of the list followed by an ellipsis (three dots), like *column-constraint* in the following example:

  ADD *column-definition* [*column-constraint*, . . .]

  One or more list elements are allowed. In this example, if more than one is specified, they must be separated by commas.

- **Optional portions** Optional portions of a statement are enclosed by square brackets.

  RELEASE SAVEPOINT [*savepoint-name*]

  These square brackets indicate that the *savepoint-name* is optional. The square brackets should not be typed.

- **Options** When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

  [**ASC** | **DESC**]

  For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

- **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces and a bar is used to separate the options.

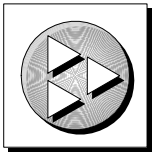
  [**QUOTES** { **ON** | **OFF** }]

  If the QUOTES option is used, one of ON or OFF must be provided. The brackets and braces should not be typed.

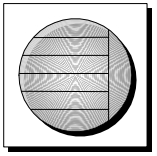
Graphic icons

The following icons are used in this documentation.

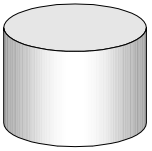
- A client application.



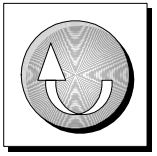
- A database server, such as Sybase Adaptive Server Anywhere.



- A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



- Replication or synchronization middleware. These assist in sharing data among databases. Examples are the MobiLink Synchronization Server and the SQL Remote Message Agent.



- A programming interface.

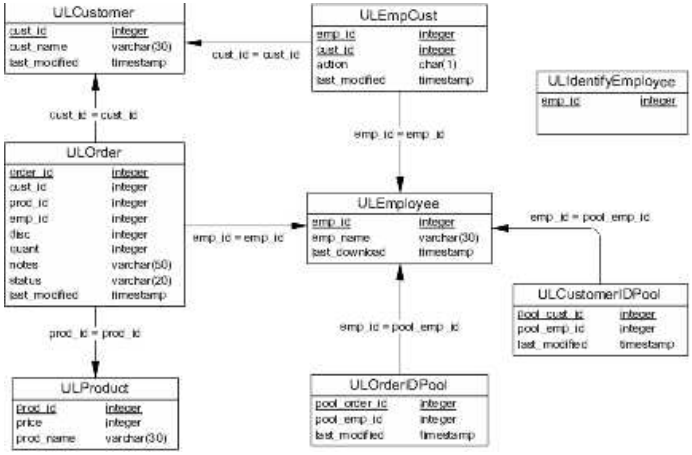
The CustDB sample database

Many of the examples in the MobiLink and UltraLite documentation use the UltraLite sample database.

The reference database for the UltraLite sample database is held in a file named *custdb.db*, and is located in the *Samples\UltraLite\CustDB* subdirectory of your SQL Anywhere directory. A complete application built on this database is also supplied.

The sample database is a sales-status database for a hardware supplier. It holds customer, product, and sales force information for the supplier.

The following figure shows the tables in the CustDB database and how they are related to each other.



Finding out more and providing feedback

We would like to receive your opinions, suggestions, and feedback on this documentation.

You can provide feedback on this documentation and on the software through newsgroups set up to discuss SQL Anywhere technologies. These newsgroups can be found on the *forums.sybase.com* news server.

The newsgroups include the following:

- sybase.public.sqlanywhere.general.
- sybase.public.sqlanywhere.linux.
- sybase.public.sqlanywhere.mobilink.
- sybase.public.sqlanywhere.product_futures_discussion.
- sybase.public.sqlanywhere.replication.
- sybase.public.sqlanywhere.ultralite.

Newsgroup disclaimer

iAnywhere Solutions has no obligation to provide solutions, information or ideas on its newsgroups, nor is iAnywhere Solutions obliged to provide anything other than a systems operator to monitor the service and insure its operation and availability.

iAnywhere Solutions Technical Advisors as well as other staff assist on the newsgroup service when they have time available. They offer their help on a volunteer basis and may not be available on a regular basis to provide solutions and information. Their ability to help is based on their workload.

CHAPTER 1

Introduction to UltraLite.NET

About this chapter

This chapter introduces you to UltraLite.NET. It assumes that you are familiar with the features of UltraLite, as described in “Welcome to UltraLite” [*UltraLite Database User’s Guide*, page 3].

Contents

UltraLite.NET features

UltraLite.NET provides the following benefits for developers targeting small devices:

- a robust relational database store
- .NET programming ease-of-use
- deployment on the Windows CE and Windows XP platforms

☞ For more information on the features and benefits of the UltraLite, see "Introduction" [*UltraLite Database User's Guide*, page 4].

UltraLite and .NET

The .NET Compact Framework is the Microsoft .NET runtime component for Windows CE. It must be deployed on any Windows CE device running a .NET application.

Developers deploying UltraLite applications using the .NET Compact Framework have the option of programming their applications in either Visual Basic .NET or the C# programming language, both of which are supported by UltraLite.NET. UltraLite.NET provides an iAnywhere.UltraLite namespace together with an UltraLite runtime library. This combination provides the benefits of .NET development together with access to operating-system specific features such as ActiveSync synchronization.

UltraLite.NET provides you:

- **Component API** UltraLite.NET shares API features and structure with the other UltraLite components.
- **Windows CE deployment** UltraLite.NET has been developed with Windows CE as a deployment target. It also supports ActiveSync synchronization.

System requirements and supported platforms

Deployment

Deployment requires one of the following:

- Microsoft .NET Compact Framework 1.0.3705 or later on Windows CE 3.0 and higher, with Pocket PC on Arm or MIPS processors. The Windows CE emulator is also supported.
- Microsoft .NET Compact Framework version 1.0.5000 or later on Windows XP.

In addition to your application code and the .NET Compact Framework, you must deploy the following files to your Windows CE device or computer running Windows XP:

- **iAnywhere.UltraLite.dll** This file contains the UltraLite.NET namespace.
- **iAnywhere.UltraLite.resources.dll** The resources needed by UltraLite.NET.
- **ulnet9.dll** The UltraLite runtime. A separate version of this runtime is provided for each target platform.
- **UltraLite schema file** The UltraLite runtime library uses the information in the schema file to set the database schema. Once a database file is created, the schema file is no longer required.

☞ For more information, see “UltraLite host platforms” [*Introducing SQL Anywhere Studio*, page 126], and “UltraLite target platforms” [*Introducing SQL Anywhere Studio*, page 136]

UltraLite.NET architecture

The UltraLite.NET namespace is named **iAnywhere.UltraLite**. It includes a set of classes for data access and synchronization: Some of the more commonly-used high level classes are:

- **DatabaseManager** You create one DatabaseManager object for each UltraLite.NET application.

  ☞ For more information, see **iAnywhere.UltraLite.DatabaseManager** in the API Reference.

- **Connection** Each Connection object represents a connection to the UltraLite database. You can create a number of Connection objects.

  ☞ For more information, see **iAnywhere.UltraLite.Connection** in the API Reference.

- **Dynamic SQL objects - PreparedStatement, ResultSet, and ResultSetSchema** objects allow you to create Dynamic SQL statements, make queries and execute INSERT, UPDATE and DELETE statements, and attain programmatic control over database resultsets.

  ☞ For more information on the PreparedStatement, Resultset, and ResultSetSchema objects, see **ianywhere.UltraLite.PreparedStatement**, **ianywhere.UltraLite.ResultSet** and **ianywhere.UltraLite.ResultSetSchema**.

- **Table** The Table object provides access to the data in the database.

  ☞ For more information, see **iAnywhere.UltraLite.Table** in the API Reference.

- **SyncParms** You use the SyncParms object to add synchronization to your application.

  ☞ For more information, see **iAnywhere.UltraLite.SyncParms** in the API Reference.

The API Reference is supplied in the *docs* subdirectory of your SQL Anywhere installation and is accessible from the front page of the SQL Anywhere Studio online books under the title iAnywhere.UltraLite.

CHAPTER 2

Tutorial: Visual Studio Application

About this chapter

This chapter walks you through all the steps of building an UltraLite.NET application in Visual Studio.

Contents

Introduction

This tutorial walks you through building an UltraLite.NET application in Visual Studio.

Timing

The tutorial takes about thirty minutes if you cut and paste the code. It takes significantly longer if you type the code yourself.

Competencies and experience

This tutorial assumes:

- you are familiar with the C# programming language or the Visual Basic .NET programming language
- you have Microsoft Visual Studio .NET 2003 installed on your machine
- you know how to create an UltraLite schema using either ulinit or the UltraLite Schema Painter.

  ☞ For more information, see “UltraLite Schema Painter Tutorial” [*UltraLite Database User’s Guide*, page 83].

Goals

The goals for the tutorial are to gain competence and familiarity with the process of developing UltraLite.NET applications in the Visual Studio environment.

This tutorial contains code for both a Visual Basic .NET application and a Visual C# application.

Lesson 1: Create a Visual Studio project

The first step is to set up a new Visual Studio application. You also need to create a schema from which to generate your database.

❖ Create a new project in Visual Studio

1. Create a Visual Studio project to hold your work:
   (a) From the Visual Studio .NET 2003 File menu, choose New ➤ Project to create a new project. The New Project window appears.
   (b) Select either a Visual Basic Smart Device Application or a Visual C# Smart Device Application and name your project **VBapp** or **CSapp**, depending on whether you've selected a Visual Basic or C# project.
   (c) Enter a Location of *c:\tutorial\uldotnet* and click OK. The Smart Device Application Wizard appears.
   (d) Choose Pocket PC as the target platform, and select Windows Application as the project type. Click OK. A Design workspace appears, displaying a form.
2. Add references to your project:
   (a) From the Project menu, choose Add References. The Add Reference window appears.
   (b) Select iAnywhere.UltraLite from the components listed. Click Select to add it to the list of selected components and click OK.

   If UltraLite.NET does not appear in the list, click Browse and locate it in the *ultralite\UltraLite.NET\ce* subdirectory of your SQL Anywhere installation. Select *iAnywhere.UltraLite.dll* and click Open.

   Alternatively, open a command prompt and browse to the *ultralite\UltraLite.NET\ce* subdirectory of your SQL Anywhere installation. Run *install.btm*. For usage, type the following command:

   ```
   install ?
   ```

   (c) Click Project ➤ Add Existing Item and browse to the *ultralite\UltraLite.NET\ce\en* subdirectory of your SQL Anywhere installation. Select the file *iAnywhere.UltraLite.resources.dll*. Click the arrow on the Open button and select Link File to link it to your project.
   (d) Click Project ➤ Add Existing Item and browse to the *ultralite\UltraLite.NET\ce* subdirectory of your SQL Anywhere installation.

   Open the folder corresponding to the processor of the CE device you are using (for the Pocket PC emulator, open the *x86* folder) and select

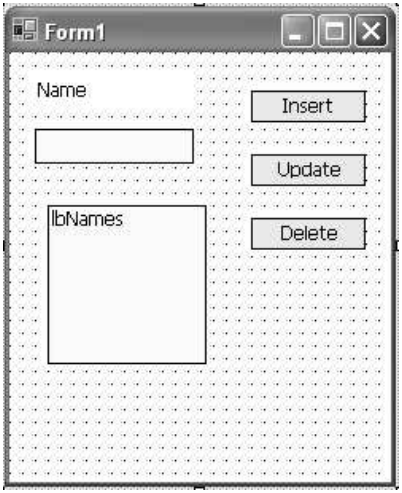
ulnet9.dll. Click on the arrow on the Open button and select Link File to link it to your project.

(e) Create a form for your application.

Open Form1, and add the following components to it:

Type	Name	Text
Button	btnInsert	Insert
Button	btnUpdate	Update
Button	btnDelete	Delete
TextBox	txtName	
ListBox	lbNames	
Label	laName	Name:

Your form should look like this:



You are now ready to move to the next lesson.

Lesson 2: Create an UltraLite schema file

This section assumes some familiarity with the UltraLite Schema Painter. For more information, see the "UltraLite Schema Painter Tutorial" [*UltraLite Database User's Guide*, page 83].

❖ Create a schema file

1. Use the UltraLite Schema Painter to create a database schema in the same directory as your application with the following characteristics:
   - **Schema file name** tutorial.usm
   - **Character set** Default
   - **Case sensitivity** Leave unchecked.
   - **Table name** Names
   - **Columns** Create columns in the Names table with the following attributes:

Column Name	Data Type (Size)	Allow NULL?	Default value
id	integer	No	autoincrement
name	char(30)	No	None

   - **Primary key** Ascending **id**

2. Link the schema file to your application:
   (a) From the Visual Studio .NET 2003 interface, choose Project ➤ Add Existing Item.
   (b) Browse to your tutorial directory and select tutorial.usm.
   (c) Click the down arrow on the Open button and choose Link File to add the file to your application.
   (d) Once the tutorial schema file has been linked to your application, right click it in the Solution Explorer and select Properties. In the properties pane, set the Build Action property to Content.

You have now added the schema file to your application.

Lesson 3: Connect to the database

In this lesson you add code to your .NET application that connects to a database using the schema you have created.

UltraLite database files have an extension of *.udb*. If an application attempts to connect to a database and the specified database file does not exist, UltraLite uses the schema file to create the database.

This tutorial assumes that if you are designing a C# application, your files are in the directory *c:\tutorial\uldotnet\CSapp* and that if you are designing a Visual Basic application, your files are in the directory *c:\tutorial\uldotnet\VBapp*. If you create a directory with a different name, use that directory instead of *c:\tutorial\uldotnet\CSApp* or *c:\tutorial\uldotnet\VBApp* throughout the tutorial.

❖ To connect to an UltraLite database

1. Right click on your form and select View Code. For a Visual C# application, add the following statement at the beginning of the file:

   ```
   using iAnywhere.UltraLite;
   ```

   This references the iAnywhere.UltraLite.NET libraries. Visual Basic projects do not need this explicit declaration.

   Add the following global variables to the form declaration of your Visual C# project, after the code describing the form components.

   ```
   private DatabaseManager DbMgr = new DatabaseManager();
   private Connection Conn;
   private PreparedStatement PrepStmt;
   private int[] ids;
   ```

   For a Visual Basic project, add the following code after the declaration of the form's components:

   ```
   Dim DbMgr As New iAnywhere.UltraLite.DatabaseManager
   Dim Conn As iAnywhere.UltraLite.Connection
   Dim PrepStmt As iAnywhere.UltraLite.PreparedStatement
   Dim ids[] As Integer
   ```

2. Double click on a blank area of your form to create a Form1_Load method. If developing a C# application, add the following code to the method:

```
try
{
   CreateParms parms = new CreateParms();
   parms.DatabaseOnCE =
      "\\Program Files\\CSApp\\tutorial.udb";
   parms.Schema.SchemaOnCE =
      "\\Program Files\\CSApp\\tutorial.usm";
   try
   {
      Conn = DbMgr.OpenConnection( parms );
      MessageBox.Show(
         "Connected to an existing database." );
   }
   catch( SQLException err )
   {
      if( err.ErrorCode ==
        SQLCode.SQLE_ULTRALITE_DATABASE_NOT_FOUND )
      {
         Conn = DbMgr.CreateDatabase( parms );
         MessageBox.Show(
            "Connected to a new database.");
      }
      else
      {
         throw err;
      }
   RefreshLB();
   }
}
catch( SQLException err )
{
   MessageBox.Show("Exception: " + err.Message +
            " sqlcode=" + err.ErrorCode);
}
catch ( System.Exception t )
{
   MessageBox.Show( "Exception: " + t.Message);
}
```

For a Visual Basic application, use this code:

```
Try
   Dim parms As New iAnywhere.UltraLite.CreateParms
   parms.DatabaseOnCE = _
      "\\Program Files\\VBApp\\tutorial.udb"
   parms.Schema.SchemaOnCE = _
      "\\Program Files\\VBApp\\tutorial.usm"
   Try
      Conn = DbMgr.OpenConnection(parms)
      MessageBox.Show( _
        "Connected to an existing database.")
   Catch err As iAnywhere.UltraLite.SQLException
      If err.ErrorCode = _
        SQLCode.SQLE_ULTRALITE_DATABASE_NOT_FOUND _
      Then
         Conn = DbMgr.CreateDatabase(parms)
         MessageBox.Show("Connected to a new database.")
      Else
         Throw err
      End If
   End Try
RefreshLB()
Catch
   MsgBox("Exception: " + err.Description)
End Try
```

This code carries out the following tasks:

- Instantiates and defines a new CreateParms object. CreateParms stores the parameters necessary to connect to or create a database. Here, the parameters are the location of the database file on the device, and the location of the schema file on the device to use if the database does not exist.
- Opens a connection to the database using the CreateParms object.

  If the database file does not exist, a SQLException is thrown. The code that catches this exception uses the schema file to create a new database and establish a connection to it.

  If the database file does exist, a connection is established.
- Calls the RefreshLB method, which you will define later in this tutorial.
- If an error occurs, prints the error message. For SQL errors, the code also prints the error code. For more information on the error code, you can look it up in the Adaptive Server Anywhere Error Messages book that is part of this documentation set.

3. Build the project.

   Choose Build ➤ Build Solution. At this stage, you should get a single error reported, which is that the name RefreshLB does not exist.

If you get other errors, check the code until they are fixed. Check for common errors, such as case differences (for example, UltraLite, DbMgr must match case exactly).

Once the project builds with a single error, you can move to the next lesson.

Lesson 4: Insert, update, and delete data

This lesson shows you how to modify your database. It uses dynamic SQL to modify the data. You can also use a table-based API.

☞ For more information, see "Data access in UltraLite" [*UltraLite Database User's Guide*, page 11].

The first step in this section is to create a supporting method to maintain the list box. Once that is complete, you can add code to modify data.

❖ Add code to maintain the listbox

1. Right click on the form and select View Code. Add the following method to update and populate the listbox; it will be used by other methods in your form.

   The following code is for a C# application:

```
private void RefreshLB()
{
   try
   {
      ResultSet DisplayAll;
      long NumRows;
      ListBoxNames.Items.Clear();
      PrepStmt = Conn.PrepareStatement(
          "SELECT id, name FROM Names");
      DisplayAll = PrepStmt.ExecuteQuery();
      DisplayAll.MoveBeforeFirst();
      NumRows = DisplayAll.RowCount;
      ids = new int[ NumRows ];
      while (DisplayAll.MoveNext())
      {
        lbNames.Items.Add(
        new LBItem(DisplayAll.GetString(2),
                   DisplayAll.GetInt(1))
        );
       ids[ DisplayAll.GetInt(1) - 1 ] =
            DisplayAll.GetInt(1);
      }
```

```
        DisplayAll.Close();
        PrepStmt.Close();
           }
           catch( SQLException err )
           {
        MessageBox.Show(
           "Exception: " + err.Message +
           " sqlcode=" + err.ErrorCode);
           }
           catch ( System.Exception t )
           {
        MessageBox.Show(
           "Exception: " + t.Message );
           }
        }
```

For a Visual Basic application, use this code:

```
Private Sub RefreshLB()
   Try
      Dim DisplayAll As iAnywhere.UltraLite.ResultSet
      Dim NumRows As Long

      lbNames.Items.Clear()
      PrepStmt = Conn.PrepareStatement( _
          "SELECT id, name FROM Names")
      DisplayAll = PrepStmt.ExecuteQuery
      DisplayAll.MoveBeforeFirst()
      NumRows = DisplayAll.RowCount

      While (DisplayAll.MoveNext)
         ListBoxNames.Items.Add( _
             New LBItem(DisplayAll.GetString(2), _
             DisplayAll.GetInt(1)) )
             Ids[ DisplayAll.GetInt(1) - 1 ] =
            DisplayAll.GetInt(1)
      End While

      DisplayAll.Close()
      PrepStmt.Close()
   Catch
      MessageBox.Show(
        "Exception: " + Err.Description
      )
   End Try
End Sub
```

This code carries out the following tasks:

- Clears the listbox.
- Instantiates a PreparedStatement and assigns it a SELECT query that returns the Names table in the database.
- Instantiates an integer array with length equal to the number of rows in the Names table.

- Populates the listbox with the names stored in the database, returned by the PreparedStatement, and populates the integer array with the ids returned by the PreparedStatement.
- Closes the resultset and prepared statement.
- If an error occurs, prints the error message. For SQL errors, the code also prints the error code. For more information on the error code, you can look it up in the Adaptive Server Anywhere Error Messages book that is part of this documentation set.

2. Add the following method to create a new listbox item class to store the name and id.

   The following code is for a C# application:

```
public class LBItem
{
   public String ItemName;
   public int ItemID;

   public LBItem(String name, int id)
   {
      ItemName = name;
      ItemID = id;
   }

   public override string ToString()
   {
      return ItemName;
   }
}
```

   For a Visual Basic application, add the following code after your Form1 class:

```
Public Class LBItem
   Public ItemName As String
   Public Itemid As Integer

   Public Sub New(ByVal name As String, ByVal id As Integer)
      ItemName = name
      Itemid = id
   End Sub

   Public Overrides Function ToString() As String
      ToString = ItemName
   End Function
End Class
```

3. Add the following method to the Form1 class to get the ID of a selected listbox item.

   The following code is for a C# application:

```
private int GetSelectedID()
{
   int curSel;
   LBItem item;
   curSel = lbNames.SelectedIndex;
   if (curSel < 0)
   {
      return -1;
   }
   else
   {
      item = (LBItem)lbNames.Items[ curSel ];
      return item.ItemID;
   }
}
```

For a Visual Basic application, add the following code just before the end of your Form1 class:

```
Private Function GetSelectedid() As Integer
   Dim curSel As Integer
   Dim item As LBItem

   curSel = lbNames.SelectedIndex
   If curSel < 0 Then
      GetSelectedid = -1
      Exit Function
   End If

   item = lbNames.Items(curSel)
   GetSelectedid = item.Itemid
End Function
```

This code carries out the following tasks:

- Gets the SelectedIndex property of the listbox.
- If the SelectedIndex does not correspond to a selection, returns -1.
- If the SelectedIndex does correspond to a selection, returns the ID for that item.

4. Build the project.

   You should be able to build the solution with no errors.

❖ Add data modification methods

1. Double click on the Insert button to create a btnInsert_Click method. If developing a C# application, add the following code to the method:

```
try
{
   long RowsInserted;

   PrepStmt = Conn.PrepareStatement(
      "INSERT INTO Names(name) VALUES (?)");
   PrepStmt.SetStringParameter(1, txtName.Text);
   RowsInserted = PrepStmt.ExecuteStatement();
   PrepStmt.Close();
   RefreshLB();
}
catch( SQLException err )
{
   MessageBox.Show("Exception: " + err.Message +
      " sqlcode=" + err.ErrorCode);
}
catch ( System.Exception t )
{
   MessageBox.Show( "Exception: " + t.Message);
}
```

   For a Visual Basic application, use the following code:

```
Try
   Dim RowsInserted As Long

   PrepStmt = Conn.PrepareStatement( _
       "INSERT INTO Names(name) VALUES (?)")
   PrepStmt.SetStringParameter(1, txtName.Text)
   RowsInserted = PrepStmt.ExecuteStatement()
   PrepStmt.Close()
   RefreshLB()
   Catch
      MessageBox.Show("Exception: " + Err.Description)
   End Try
```

   This code carries out the following tasks:

   - Instantiates a PreparedStatement and assigns it an INSERT statement that inserts the value in the textbox into the database.
   - Executes the statement.
   - Closes the statement.
   - Refreshes the listbox.
   - If an error occurs, prints the error message. For SQL errors, the code also prints the error code. For more information on the error code, you can look it up in the Adaptive Server Anywhere Error Messages book that is part of this documentation set.

2. Double click on the Update button to create a btnUpdate_Click method. If developing a C# application, add the following code to the method:

```
try
{
   long RowsUpdated;
   int updtid = ids[ lbNames.SelectedIndex ];

   PrepStmt = Conn.PrepareStatement(
      "UPDATE Names SET name = ? WHERE id = ?" );
   PrepStmt.SetStringParameter(1, txtName.Text);
   PrepStmt.SetIntParameter(2, updtid);
   RowsUpdated = PrepStmt.ExecuteStatement();
   PrepStmt.Close();
   RefreshLB();
}
catch( SQLException err )
{
   MessageBox.Show(
      "Exception: " + err.Message +
      " sqlcode=" + err.ErrorCode);
}
catch ( System.Exception t )
{
   MessageBox.Show( "Exception: " + t.Message);
}
```

For a Visual Basic application, use the following code:

```
Try
   Dim RowsUpdated As Long
   Dim updtid As Integer = ids(lbNames.SelectedIndex)

   PrepStmt = Conn.PrepareStatement( _
      "UPDATE Names SET name = ? WHERE id = ?")
   PrepStmt.SetStringParameter(1, txtName.Text)
   PrepStmt.SetIntParameter(2, updtid)
   RowsUpdated = PrepStmt.ExecuteStatement()
   PrepStmt.Close()
   RefreshLB()
Catch
   MessageBox.Show("Exception: " + Err.Description)
End Try
```

This code carries out the following tasks:

- Instantiates a PreparedStatement and assigns it an UPDATE statement that inserts the value in the textbox into the database based on the associated id (saved in the integer array ids).
- Executes the statement.
- Refreshes the listbox.
- Closes the prepared statement.

- If an error occurs, prints the error message. For SQL errors, the code also prints the error code. For more information on the error code, you can look it up in the Adaptive Server Anywhere Error Messages book that is part of this documentation set.

3. Double click on the Delete button to create a btnDelete_Click method. If developing a C# application, add the following code to the method:

```
try
{
   int delid = ids[lbNames.SelectedIndex];
   long RowsDeleted;

   PrepStmt = Conn.PrepareStatement(
      "DELETE From Names WHERE id = ?" );
   PrepStmt.SetIntParameter(1, delid);
   RowsDeleted = PrepStmt.ExecuteStatement();
   PrepStmt.Close();
   RefreshLB();
}
catch( SQLException err )
{
   MessageBox.Show("Exception: " + err.Message +
      " sqlcode=" + err.ErrorCode);
}
catch ( System.Exception t )
{
   MessageBox.Show( "Exception: " + t.Message);
}
```

For a Visual Basic application, use the following code:

```
Try
   Dim delid As Integer = ids(lbNames.SelectedIndex)
   Dim RowsDeleted As Long

   PrepStmt = Conn.PrepareStatement( _
      "DELETE From Names WHERE id = ?")
   PrepStmt.SetIntParameter(1, delid)
   RowsDeleted = PrepStmt.ExecuteStatement()
   PrepStmt.Close()
   RefreshLB()
Catch
   MessageBox.Show("Exception: " + err.description)
End Try
```

This code carries out the following tasks:

- Instantiates a PreparedStatement and assigns it a DELETE statement that deletes the selected row from the database, based on the associated id from the integer array ids.
- Executes the statement.
- Refreshes the listbox.

- Closes the prepared statement.
- If an error occurs, prints the error message. For SQL errors, the code also prints the error code. For more information on the error code, you can look it up in the Adaptive Server Anywhere Error Messages book that is part of this documentation set.

Lesson 5: Build and deploy your application

❖ Deploy and run your application

1. Build the solution.

   Ensure that your application builds without errors.

2. Select Debug ➤ Start.

   This builds an executable file containing your application and deploys it to the Pocket PC emulator. The process may take some time, especially if it must deploy the .NET Compact Framework before running the application.

3. The first time you run the application, it should display the following text in a message box:

   ```
   Connected to a new database.
   ```

   Subsequent times, it displays the following text in a message box:

   ```
   Connected to an existing database.
   ```

Deployment checklist

If errrors are reported, you may wish to check that your deployment was completed successfully. For example:

- Confirm that the application is deployed into \Program Files*appname*, where *appname* is the name you gave your application in Lesson 1 (**CSApp** or **VBApp**).
- Confirm that the schema file is deployed into the same directory as the application.
- Confirm that the path to the schema file in your application code is correct (see “Lesson 3: Connect to the database” on page 10).
- Confirm that you chose Link File when adding the schema file to the project and that you set the Build Action to Content. If you did not, the files will not be deployed to the device.
- Ensure that you added a reference to the correct version of *ulnet9.dll* for your target platform, or ran the ce installer. If you switch between the emulator and a real device, you must change the version of this library that you use. See “Lesson 1: Create a Visual Studio project” on page 7.

❖ Test your application

1. Insert data into the database.

   Enter a name in the text box and click Insert. The name should now appear in the listbox.

2. Update data in the database.

   Select a name from the listbox. Enter a new name in the text box. Click Update. The new name should now appear in place of the old name in the listbox.

3. Delete data from the database.

You have now written, built, and tested an UltraLite.NET application.

This completes the tutorial.

CHAPTER 3

Understanding UltraLite Development

About this chapter

This chapter describes how to develop applications with UltraLite.NET.

Contents

Connecting to a database

Any UltraLite application must connect to a database before it can carry out any operation on the data. This section describes how to write code to connect to an UltraLite database.

> **Note**
> The following code samples are in Microsoft C#. If you are using one of the other supported development tools, modify the instructions to fit your tool.

❖ To connect to an UltraLite database

1. Create a DatabaseManager object.

   You can create only one DatabaseManager object per application. This object is at the root of the object hierarchy. For this reason, it is often best to declare the DatabaseManager object global to the application.

   The following code creates a DatabaseManager object named dbMgr.

   ```
   DatabaseManager dbMgr = new DatabaseManager();
   ```

2. Declare a Connection object.

   Most applications use a single connection to an UltraLite database and keep the connection open all the time. Multiple connections are only required for multi-threaded data access. For this reason, it is often best to declare the Connection object global to the application.

   ```
   Connection conn;
   ```

3. Attempt to open a connection to an existing database.
   - The following code establishes a connection to an existing database held in the *mydata.udb* file on Windows.

     ```
     ConnectionParms parms = new ConnectionParms();
     parms.DatabaseOnDesktop = "mydata.udb";
     try {
       conn = dbMgr.OpenConnection( parms );
       // more actions here
     }
     ```

   - The method establishes a connection to an existing UltraLite database file and returns that open connection as a Connection object.
   - It is common to deploy a schema file rather than a database file, and to let UltraLite create the database file on the first connection attempt. If no database file exists, you should check for the error and create a database file.

The following code illustrates how to catch the error when the database file does not exist:

```
catch( SQLException econn ) {
  if( econn.GetErrorCode() ==
      SQLCode.SQLE_ULTRALITE_DATABASE_NOT_FOUND ){
      // action here
}
```

4. If no database exists, create a database and establish a connection to it.
   - The following code carries out this task on Windows, using a schema file of *mydata.usm*.

```
CreateParms parms = new CreateParms();
parms.DatabaseOnDesktop = "mydata.udb";
parms.Schema.SchemaOnDesktop = "mydata.usm";
try {
   conn = dbMgr.CreateDatabase( parms );
}
```

Example

The following code opens a connection to an UltraLite database named *mydata.udb*.

```
CreateParms parms = new CreateParms();
parms.DatabaseOnDesktop = "mydata.udb";
parms.Schema.SchemaOnDesktop = "mydata.usm";
try {
  conn = dbMgr.OpenConnection( parms );
  System.Console.WriteLine(
    "Connected to an existing database." );
 }
catch( SQLException econn ) {
  if( econn.GetErrorCode() ==
      SQLCode.SQLE_ULTRALITE_DATABASE_NOT_FOUND ){
    conn = dbMgr.CreateDatabase( parms );
    System.Console.WriteLine(
      "Connected to a new database." );
  } else {
    throw econn;
  }
}
```

In general, you will want to specify a more complete path to the file.

Using the Connection object

Properties or methods of the Connection object govern global application behavior, including the following:

- **Commit behavior** By default, UltraLite applications are in autocommit mode. Each insert, update, or delete statement is committed to the database immediately. You can also set Connection.AutoCommit to false to build transactions into your application.

☞ For more information, see "Transaction processing in UltraLite" on page 40.

- **User authentication** You can change the user ID and password for the application from the default values of DBA and SQL by using methods to Grant and Revoke connection permissions. Each application can have a maximum of four user IDs.

  ☞ For more information, see "User authentication" [*UltraLite Database User's Guide*, page 38].

- **Synchronization** A set of objects governing synchronization are accessed from the Connection object.

  ☞ For more information, see the API Reference in the online documentation.

- **Tables** UltraLite tables are accessed using methods of the Connection application.

- **Prepared statements** A set of objects is provided to handle the execution of dynamic SQL statements and to navigate result sets.

☞ For more information, see Connection.SyncParms and Connection.SyncResult in the API Reference in the online documentation.

Multi-threaded applications

Each Connection and all objects created from it should be used on a single thread. If you need to have multiple threads accessing the UltraLite database, then each thread should have its own connection.

Accessing and manipulating data with dynamic SQL

UltraLite supports dynamic SQL for accessing data in tables. This section describes the programming interface to access dynamic SQL features, including the following topics:

- Inserting, deleting, and updating rows.
- Retrieving rows to a result set.
- Scrolling through the rows of a result set.

☞ This section does not describe the SQL language itself. For information about dynamic SQL features, see "Dynamic SQL" [*UltraLite Database User's Guide*, page 125].

☞ The sequence of operations required is similar for any SQL operation. For an overview, see "Using dynamic SQL" [*UltraLite Database User's Guide*, page 126].

Data manipulation: INSERT, UPDATE and DELETE

To perform SQL Data Manipulation Language operations (INSERT, UPDATE, and DELETE), you carry out the following sequence of operations:

1. Prepare the statement.

   You can indicate parameters in the statement using the ? character.

2. Assign values for parameters in the statement.

   For any INSERT, UPDATE or DELETE, each ? is referred to by its ordinal position in the prepared statement.

3. Execute the statement.

4. Repeat steps 2 and 3 as required.

❖ To perform INSERT operations using ExecuteStatement:

1. Declare a PreparedStatement.

```
PreparedStatement prepStmt;
```

2. Assign a SQL statement to the PreparedStatement object.

```
prepStmt = conn.PrepareStatement(
   "INSERT INTO MyTable(MyColumn) values (?)");
```

3. Assign input parameter values for the statement.

   The following code shows a string parameter.

```
String newValue;
// assign value
prepStmt.SetStringParameter(1, newValue);
```

4. Execute the statement.

   The return value indicates the number of rows affected by the statement.

```
long rowsInserted = prepStmt.ExecuteStatement();
```

5. If you have set AutoCommit to off, commit the change.

```
conn.Commit();
```

❖ To perform UPDATE operations using ExecuteStatement:

1. Declare a PreparedStatement.

```
PreparedStatement prepStmt;
```

2. Assign a statement to the PreparedStatement object.

```
prepStmt = conn.PrepareStatement(
   "UPDATE MyTable SET MyColumn1 = ? WHERE MyColumn2 = ?");
```

3. Assign input parameter values for the statement.

```
String newValue;
String oldValue;
// assign values
prepStmt.SetStringParameter( 1, newValue );
prepStmt.SetStringParameter( 2, oldValue );
```

4. Execute the statement.

```
long rowsUpdated = prepStmt.ExecuteStatement();
```

5. If you have set AutoCommit to off, commit the change.

```
conn.Commit();
```

❖ **To perform DELETE operations using ExecuteStatement:**

1. Declare a PreparedStatement.

```
PreparedStatement prepStmt;
```

2. Assign a statement to the PreparedStatement object.

```
prepStmt = conn.PrepareStatement(
   "DELETE FROM MyTable WHERE MyColumn = ?");
```

3. Assign input parameter values for the statement.

```
String deleteValue;
prepStmt.SetStringParameter(1, deleteValue);
```

4. Execute the statement.

```
long rowsDeleted = prepStmt.ExecuteStatement();
```

5. If you have set AutoCommit to off, commit the change.

```
conn.Commit();
```

Data retrieval: SELECT

Use the SELECT statement to retrieve information from the database.

❖ **To execute a SELECT query using ExecuteQuery:**

1. Create a new prepared statement and result set.

```
PreparedStatement prepStmt;
```

2. Assign a prepared statement to your newly created PreparedStatement object.

```
prepStmt = conn.PrepareStatement(
   "SELECT MyColumn FROM MyTable");
```

3. Execute the statement.

   In the following code, the result of the SELECT query contain a string, which is output to a command prompt.

```
ResultSet customerNames = prepStmt.ExecuteQuery();
customerNames.MoveBeforeFirst();
while( customerNames.MoveNext() ) {
   for ( int i = 1;
         i <= customerNames.Schema.GetColumnCount();
         i++ ) {
        System.Console.Write(
        customerNames.GetString( i ) + " "
        );
        System.Console.WriteLine();
  }
}
```

Navigating through dynamic SQL result sets

You can navigate through a result set using methods associated with the ResultSet object.

Moving through a result set

The result set object provides you with a number of methods to navigate a result set.

The following methods allow you to navigate your result set:

- MoveAfterLast() moves to a position after the last row.
- MoveBeforeFirst() moves to a position before the first row.
- MoveFirst() moves to the first row.
- MoveLast() moves to the last row.
- MoveNext() moves to the next row.
- MovePrevious() moves to the previous row.
- MoveRelative(offset) moves a certain number of rows relative to the current row, as specified by the offset. Positive offset values move forward in the result set, relative to the current position of the cursor in the result set, and negative offset values move backward in the result set. An offset value of zero does not move the cursor. Zero is useful if you want to repopulate a row buffer.

Result set schema description

The ResultSet.Schema method allows you to retrieve information about a result set, such as column names, total number of columns, column

precisions, column scales, column sizes and column SQL types. The following example shows how you can use ResultSet.Schema to display schema information in a console window.

```
for ( int i = 1;
      i <= MyResultSet.Schema.GetColumnCount();
      i++ ) {
   System.Console.WriteLine(
      MyResultSet.Schema.GetColumnName(i) + " " +
      MyResultSet.Schema.GetColumnSQLType(i)
   );
}
```

Accessing and manipulating data with the Table API

UltraLite applications can access data in tables in a row-by-row fashion. This section covers the following topics:

- Scrolling through the rows of a table.
- Accessing the values of the current row.
- Using find and lookup methods to locate rows in a table.
- Inserting, deleting, and updating rows.

The section also provides a lower-level description of the way that UltraLite operates on the underlying data to help you understand how it handles transactions, and how changes are made to the data in your database.

Data manipulation internals

UltraLite exposes the rows in a table to your application one at a time. The Table object has a current position, which may be on a row, before the first row, or after the last row of the table.

When your application changes its row (by using one of the navigation methods on the Table object) UltraLite makes a copy of the row in a buffer. Any operations to get or set values affect only the copy of data in this buffer. They do not affect the data in the database. For example, the following statement changes the value of the ID column in the buffer to 3.

```
short id = t.Schema.GetColumnID( "ID" );
t.SetInt( id, 3 );
```

Using UltraLite modes

UltraLite uses the values in the buffer for a variety of purposes, depending on the kind of operation you are carrying out. UltraLite has four different modes of operation, in addition to a default mode, and in each mode the buffer is used for a different purpose.

- **Insert mode** The data in the buffer is added to the table as a new row when the insert method is called.
- **Update mode** The data in the buffer replaces the current row when the update method is called.
- **Find mode** The data in the buffer is used to locate rows when one of the find methods is called.
- **Lookup mode** The data in the buffer is used to locate rows when one of the lookup methods is called.

Whichever mode you are using, there is a similar sequence of operations:

1. Enter the mode.

   The Table methods set UltraLite into the mode.

2. Set the values in the buffer.

   Use the set methods to set values in the buffer.

3. Carry out the operation.

   Use a Table method to carry out an operation such as insert, update, or find using the values in the buffer. The UltraLite mode is set back to the default method and you must enter a new mode before performing another data manipulation or searching operation.

Scrolling through the rows of a table

The following code opens the MyTable table and scrolls through its rows, displaying the value of the MyColumn column for each row.

```
Table t = conn.GetTable( "MyTable" );
short colID = t.Schema.GetColumnID( "MyColumn" );
t.Open();
t.MoveBeforeFirst();
while ( t.MoveNext() ){
   System.Console.WriteLine( t.GetString( colID ) );
}
```

You expose the rows of the table to the application when you open the table object. By default, the rows are exposed in order by primary key value, but you can specify an index to access the rows in a particular order. The following code moves to the first row of the MyTable table as ordered by the ix_col index.

```
Table t = conn.GetTable("MyTable");
t.Open( "ix_col" );
t.MoveFirst();
```

☞ For more information, see iAnywhere.UltraLite.Table iAnywhere.UltraLite.TableSchema and iAnywhere.UltraLite.Connection in the online API Reference.

Accessing the values of the current row

At any time, a Table object is positioned at one of the following positions:

- Before the first row of the table.
- On a row of the table.

- After the last row of the table.

If the Table object is positioned on a row, you can use one of a set of methods appropriate for the data type to access the value of each column. These methods take the column ID as argument. For example, the following code retrieves the value of the lname column, which is a character string:

```
short lname = t.Schema.GetColumnID( "lname" );
System.String lastname = t.GetString( lname );
```

The following code retrieves the value of the cust_id column, which is an integer:

```
short cust_id = t.Schema.GetColumnID( "cust_id" );
int id = t.GetInt( cust_id );
```

In addition to the methods for retrieving values, there are methods for setting values. These take the column ID and the value as arguments. For example:

```
t.SetString( lname, "Kaminski" );
```

By assigning values to these properties you do not alter the value of the data in the database. You can assign values to the properties even if you are before the first row or after the last row of the table, but it is an error to try to access data when the current row is at one of these positions, for example by assigning the property to a variable.

```
// This code is incorrect
t.MoveBeforeFirst();
id = t.GetInt( cust_id );
```

Casting values

The method you choose must match the data type you wish to assign. UltraLite automatically casts database data types where they are compatible, so that you could use the getString method to fetch an integer value into a string variable, and so on.

Searching for rows with find and lookup

UltraLite has several modes of operation when working with data. The Table object has two sets of methods for locating particular rows in a table:

- **find methods** These methods move to the first row that exactly matches specified search values, under the sort order specified when the Table object was opened. If the search values cannot be found you are positioned before the first or after the last row.
- **lookup methods** These methods move to the first row that matches or is greater than a specified search value, under the sort order specified when the Table object was opened.

Both sets are used in a similar manner:

1. Enter find or lookup mode.

   The mode is entered by calling a method on the table object. For example.

   ```
   t.FindBegin();
   ```

2. Set the search values.

   You do this by setting values in the current row. Setting these values affects the buffer holding the current row only, not the database. For example:

   ```
   short lname = t.Schema.GetColumnID( "lname" );
   t.SetString( lname, "Kaminski" );
   ```

   Only values in the columns of the index are relevant to the search.

3. Search for the row.

   Use the appropriate method to carry out the search. For example, the following instruction looks for the first row that exactly matches the specified value in the current index:

   ```
   tCustomer.FindFirst();
   ```

   For multi-column indexes, a value for the first column is always used, but you can omit the other columns and you can specify the number of columns as a parameter to the find method.

4. Search for the next instance of the row.

   Use the appropriate method to carry out the search. For a find operation, FindNext() locates the next instance of the parameters in the index. For a lookup, MoveNext() locates the next instance.

☞ For more information, see the following classes in the API Reference:

- **iAnywhere.UltraLite.Table**

Inserting updating, and deleting rows

To update a row in a table, use the following sequence of instructions:

1. Move to the row you wish to update.

   You can move to a row by scrolling through the table or by searching, using find and lookup methods.

2. Enter update mode.

   For example, the following instruction enters update mode on t:

   ```
   t.BeginUpdate();
   ```

3. Set the new values for the row to be updated. For example:

```
t.SetInt( id , 3);
```

4. Execute the Update.

```
t.Update();
```

After the update operation the current row is the row that was just updated. If you changed the value of a column in the index specified when the Table object was opened, the current row is undefined.

By default, UltraLite.NET operates in autocommit mode, so that the update is immediately applied to the row in permanent storage. If you have disabled autocommit mode, the update is not applied until you execute a commit operation. For more information, see "Transaction processing in UltraLite" on page 40.

> ***Caution***
> *You can not update the primary key of a row: delete the row and add a new row instead.*

Inserting rows

The steps to insert a row are very similar to those for updating rows, except that there is no need to locate any particular row in the table before carrying out the insert operation. The order of row insertion into the table has no significance.

For example, the following sequence of instructions inserts a new row:

```
t.InsertBegin();
t.SetInt( id, 3 );
t.SetString( lname, "Carlo" );
t.Insert();
```

If you do not set a value for one of the columns, and that column has a default, the default value is used. If the column has no default, the following entries are added:

- For nullable columns, NULL.
- For numeric columns that disallow NULL, zero.
- For character columns that disallow NULL, an empty string.
- To explicitly set a value to NULL, use the setNull method.

As for update operations, an insert is applied to the database in permanent storage itself when a commit is carried out. In AutoCommit mode, a commit is carried out as part of the insert method.

Deleting rows

The steps to delete a row are simpler than to insert or update rows. There is no delete mode corresponding to the insert or update modes. The steps are as follows:

1. Move to the row you wish to delete.
2. Execute the Table.Delete() method.

Transaction processing in UltraLite

UltraLite provides transaction processing to ensure the correctness of the data in your database. A transaction is a logical unit of work: it is either all executed or none of it is executed.

By default, UltraLite.NET operates in autocommit mode, so that each insert, update, or delete is executed as a separate transaction. Once the operation is completed, the change is made to the database. If you set the Connection.AutoCommit property to false, you can use multi-statement transactions. For example, if your application transfers money between two accounts, either both the deduction from the source account and the addition to the destination account must be completed, or neither must be completed.

If AutoCommit is set to false, you must execute a Connection.Commit() statement to complete a transaction and make changes to your database permanent, or you must execute a Connection.Rollback() statement to cancel all the operations of a transaction.

☞ For more information, see the **ianywhere.Ultralite.Connection** class in the API Reference, in the online books.

Accessing schema information

Objects in the API represent tables, columns, indexes, and synchronization publications. Each object has a Schema property that provides access to information about the structure of that object.

Here is a summary of the information you can access through the schema objects.

- **DatabaseSchema** The number and names of the tables in the database, as well as global properties such as the format of dates and times.

  To obtain a DatabaseSchema object, access Connection.Schema. See the API Reference in the online books.

- **TableSchema** The number and names of the columns and indexes for this table.

  To obtain a TableSchema object, access Table.Schema. See the API Reference in the online books.

- **IndexSchema** Information about the column in the index. As an index has no data directly associated with it (only the type which is in the columns of the index) there is no separate Index class, just a IndexSchema class.

  To obtain a IndexSchema object, call the TableSchema.GetIndex, the TableSchema.GetOptimalIndex, or the TableSchema.GetPrimaryKey method. See the API Reference in the online books.

- **PublicationSchema** Tables and columns contained in a publication. Publications are also comprised of schema only, and so there is only a PublicationSchema object and not a Publication object.

  To obtain a PublicationSchema object, call the DatabaseSchema.TableSchema.GetPublicationSchema method. See the API Reference in the online books.

You cannot modify the schema through the API. You can only retrieve information about the schema.

Error handling

You can use the standard .NET error-handling features to handle errors. Most methods throw SQLException errors. You can use SQLException.GetErrorCode() to retrieve the SQLCode value assigned to this error. SQLCode errors are negative numbers indicating the particular kind of error.

☞ For more information, see the following enumeration in the API Reference, in the online books:

- **iAnywhere.UltraLite.StreamErrorID**

After Synchronization, you can use the SyncResult object to obtain more detailed error information.

☞ For more information, see the following properties and enumerations in the API Reference, in the online books:

- **ActiveSyncListener.AuthStatusCode Enumeration**
- **SyncResult.StreamErrorID**
- **SyncResult.StreamErrorCode**
- **SyncResult.StreamErrorContext**

User authentication

There is a common sequence of events to managing user IDs and passwords.

1. New users have to be added from an existing connection. As all UltraLite databases are created with a default user ID and password of DBA and SQL, respectively, you must first connect as this initial user and implement user management only upon successful connection.
2. You cannot change a user ID: you add a user and delete an existing user. A maximum of four user IDs are permitted for each UltraLite database.
3. To change the password for an existing user ID, use the Connection.GrantConnectTo method.

☞ For more information, see the following classes in the API Reference:

- **iAnywhere.UltraLite.Connection**

Adding ActiveSync synchronization to your application

This section describes special steps that you must take to add ActiveSync to your application, and how to register your application for use with ActiveSync on your end users' machines.

Synchronization requires SQL Anywhere Studio. For general information on setting up ActiveSync synchronization, see "Deploying applications that use ActiveSync" [*MobiLink Synchronization User's Guide*, page 225] in the MobiLink Synchronization User's Guide. For general information on adding synchronization to an application, see "Synchronizing UltraLite applications" [*UltraLite Database User's Guide*, page 144].

ActiveSync synchronization can be initiated only by ActiveSync itself. ActiveSync can automatically initiate a synchronization when the device is placed in the cradle or when the Synchronization command is selected from the ActiveSync window.

When ActiveSync initiates synchronization, the MobiLink ActiveSync provider starts the UltraLite application, if it is not already running, and sends a message to it. Your application must implement an ActiveSyncListener to receive and process messages from the MobiLink provider. Your application must specify the listener object using:

```
dbMgr.SetActiveSyncListener(
   "MyAppClassName", listener );
```

where MyAppClassName is a unique Windows class name for the application. For more information, see in the API Reference.

When UltraLite receives an ActiveSync message, it invokes the specified listener's ActiveSyncInvoked(boolean) method on a different thread. To avoid multi-threading issues, your ActiveSyncInvoked(boolean) method should post an event to the user interface.

If your application is multi-threaded, use a separate connection and use the **synchronized** keyword to access any objects shared with the rest of the application. The ActiveSyncInvoked() method should specify a StreamType.ACTIVE_SYNC for its connection's SyncParms.Stream and then call Connection.Synchronize().

When registering your application, set the following parameters:

- **Class Name** The same class name the application used with the Connection.SetActiveSyncListener method.

Index

L

M

N

O

P

R

S

T

U

V