Native UltraLite™ for Java User's Guide

Part number: DC36294-01-0902-01
Last modified: October 2004

Contents

About This Manual

Subject	This manual describes Native UltraLite for Java. With Native UltraLite for Java you can develop and deploy database applications to Windows CE devices running the Jeode or CrEme VMs.
Audience	This manual is intended for Java 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 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 SNMP Extension Agent User's Guide** This book describes how to configure the Adaptive Server Anywhere SNMP Extension Agent for use with SNMP management applications to manage Adaptive Server Anywhere databases.
- **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 Administration 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 Clients** This book describes how to set up and synchronize Adaptive Server Anywhere and UltraLite remote databases.
- **MobiLink Server-Initiated Synchronization User's Guide** This book describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization from the consolidated database.
- **MobiLink Tutorials** This book provides several tutorials that walk you through how to set up and run MobiLink applications.
- **QAnywhere User's Guide** This manual describes MobiLink QAnywhere, a messaging platform that enables the development and deployment of messaging applications for mobile and wireless clients, as well as traditional desktop and laptop clients.
- **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.

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

  The PDF books are accessible from the online books, or from the Windows Start menu.

- **Printed books** The complete set of books is available from Sybase sales or from eShop, the Sybase online store, at *http://eshop.sybase.com/eshop/documentation*.

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.

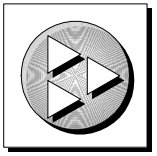
  [**QUOTES** { **ON** | **OFF** }]

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

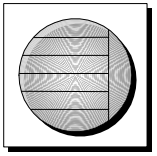
Graphic icons

The following icons are used in this documentation.

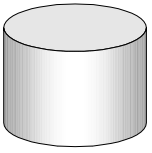
- A client application.

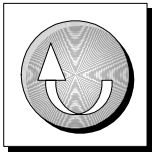
- A database server, such as Sybase Adaptive Server Anywhere.

- 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.

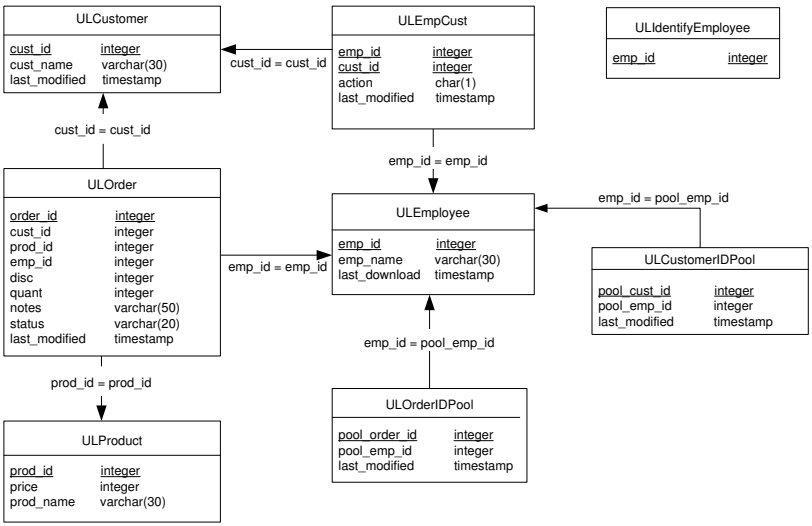
The CustDB sample database

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 diagram shows the tables in the CustDB database and how they are related to each other.



Finding out more and providing feedback

Finding out more

Additional information and resources, including a code exchange, are available at the iAnywhere Developer Network at *http://www.ianywhere.com/developer/*.

If you have questions or need help, you can post messages to the iAnywhere Solutions newsgroups listed below.

When you write to one of these newsgroups, always provide detailed information about your problem, including the build number of your version of SQL Anywhere Studio. You can find this information by typing **dbeng9 -v** at a command prompt.

The newsgroups are located 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
- ianywhere.public.sqlanywhere.qanywhere

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 ensure 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.

Feedback

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

You can e-mail comments and suggestions to the SQL Anywhere documentation team at iasdoc@ianywhere.com. Although we do not reply to e-mails sent to that address, we read all suggestions with interest.

In addition, you can provide feedback on the documentation and the software through the newsgroups listed above.

CHAPTER 1

Introduction to Native UltraLite for Java

About this chapter

This chapter introduces you to Native UltraLite for Java. It assumes that you are familiar with the features of UltraLite, as described in "Welcome to UltraLite" [*UltraLite Database User's Guide*, page 3].

☞ For more information about creating applications using Native UltraLite for Java, see "Understanding Native UltraLite for Java Development" on page 33.

☞ For a hands-on tutorial introducing Native UltraLite for Java, see "Tutorial: Build a Native UltraLite for Java Application" on page 5 or "Tutorial: The CustDB Sample Application" on page 23.

Contents

UltraLite and Java

Java developers wishing to take advantage of UltraLite database features have two options:

- The Static Java API is a pure Java UltraLite technology described in *UltraLite Static Java User's Guide*.
- Native UltraLite for Java (documented in the current book) provides a Native UltraLite for Java package, together with a native (C++) UltraLite runtime library. This combines the benefits of Java development with the performance of native applications and access to operating system-specific features such as ActiveSync synchronization.

Native UltraLite for Java differs from UltraLite for Java in the following ways:

- **Native components** The UltraLite runtime for Native UltraLite for Java uses native methods. That is, it is not written in Java but in C++, and compiled into binary form specific to the underlying CPU and operating system. In contrast, the UltraLite runtime for the Static Java API is a pure Java implementation.
- **Component API** Native UltraLite for Java shares API features and structure with the other members of the UltraLite Component Suite. The Static Java API uses JDBC as the programming interface.
- **Windows CE deployment** Native UltraLite for Java offers Windows CE as a deployment target. It supports the Jeode VM provided with many Windows CE devices and the CrEme VM. It also supports ActiveSync synchronization.

  For details of platform support, see "UltraLite target platforms" [*Introducing SQL Anywhere Studio*, page 109].

☞ For more information about the features of UltraLite databases, see "UltraLite Databases" [*UltraLite Database User's Guide*, page 27].

System requirements and supported platforms

Development platforms

To develop applications using Native UltraLite for Java, you require the following.

- Microsoft Windows NT/2000/XP.
- JDK 1.1.8 or Personal Java 1.2 is recommended for Native UltraLite for Java application development. Borland JBuilder 7 and 8 integration is provided.

Target platforms

Native UltraLite for Java supports the following target platforms:

- Windows CE 3.0 or higher on devices using the ARM processor, including the Compaq iPaq and NEC MobilePro P300, and a Jeode Personal Java 1.2 compatible VM.
- Windows NT/2000/XP with JRE 1.1.8 or later.

☞ For more information, see "UltraLite development platforms" [*Introducing SQL Anywhere Studio*, page 99] and "UltraLite target platforms" [*Introducing SQL Anywhere Studio*, page 109].

Native UltraLite for Java architecture

The Native UltraLite for Java package is named ianywhere.native_ultralite.

The following list describes some of the more commonly-used high level classes.

- **DatabaseManager** You create one DatabaseManager object for each application.
- **Connection** Each Connection object represents a connection to an UltraLite database. You can create one or more Connection objects.
- **Table** The Table object provides access to the data in the database.
- **PreparedStatement, ResultSet, and ResultSetSchema** These dynamic SQL objects allow you to create Dynamic SQL statements, make queries and execute INSERT, UPDATE and DELETE statements, and attain programmatic control over database result sets.
- **SyncParms** You use the SyncParms object to synchronize your UltraLite database with a MobiLink synchronization server.

The API Reference is supplied in Javadoc format in the *UltraLite\NativeUltraLiteForJava\doc* subdirectory of your SQL Anywhere installation. For more information about accessing the API reference, see "Native UltraLite for Java API Reference" on page 63.

CHAPTER 2

Tutorial: Build a Native UltraLite for Java Application

About this chapter

This chapter provides a tutorial to guide you through the process of building a sample Native UltraLite for Java application.

Contents

Introduction

This tutorial guides you through the process of building a Native UltraLite for Java application.

This tutorial uses a text editor to edit the Java files. You can also use Native UltraLite for Java in the Borland JBuilder development environment.

☞ For more information, see "Developing applications with Borland JBuilder" on page 60.

Timing

This tutorial takes about 30 minutes if you copy and paste the code. If you enter the code yourself, it takes significantly longer.

Competencies and experience

This tutorial assumes:

- you are familiar with the Java programming language
- you have JDK 1.1.8 or Personal Java 1.2 installed on your computer
- you know how to create an UltraLite schema using the UltraLite Schema Painter

  ☞ For more information, see "Lesson 1: Create an UltraLite database schema" [*UltraLite Database User's Guide*, page 130].

> **Note**
> The synchronization portion of this tutorial requires SQL Anywhere Studio.

Goals

The goals for this tutorial are to gain competence and familiarity with the process of developing a Native UltraLite for Java application.

Lesson 1: Connect to the database

In the first procedure, you create a database schema. You then write, compile, and run a Java application that creates a database using the schema you have created, or connects to an existing database.

❖ To create a database schema

1. Create a directory to hold the files you create in this tutorial.

   This tutorial assumes the directory is *c:\tutorial\java*. If you create a directory with a different name, use that directory instead of *c:\tutorial\java* throughout the tutorial.

2. Create a database schema with the following characteristics using the UltraLite Schema Painter.

   ☞ For more information, see the "Lesson 1: Create an UltraLite database schema" [*UltraLite Database User's Guide*, page 130].

   Schema file name: **tutcustomer.usm**

   Table name: **Customer**

   Columns in Customer:

Column Name	Data Type (Size)	Column allows NULL values?	Default value
ID	integer	No	autoincrement
FName	char(15)	No	None
LName	char(20)	No	None
City	char(20)	Yes	None
Phone	char(12)	Yes	555-1234

   Primary key: ascending **ID**

❖ To connect to an UltraLite database

1. In your tutorial directory, create a file named *Customer.java*.

2. Copy the code below into *Customer.java*.

   This code carries out the following tasks:

   - Imports the UltraLite library and the JDBC SQLException class.
   - Declares a class named Customer.
   - Declares a static variable to hold the database connection object. This object will be shared among several methods later in the tutorial.

- Invokes the class constructor.
- If an error occurs, prints the error code and a stack trace.

  ☞ For more information about the error code, see *ASA Error Messages*.

```
import ianywhere.native_ultralite.*;
import java.sql.SQLException;

public class Customer {
  static Connection conn;
  public static void main( String args[] ) {
    try {
      Customer cust = new Customer();
      // Clean up
      conn.close();
    } catch( SQLException e ) {
      System.out.println(
        "Exception: " + e.getMessage() +
        " sqlcode=" + e.getErrorCode()
          );
      e.printStackTrace();
    }
  }
}
```

3. Add a constructor to the class.

   Copy the code below into your file. This code carries out the following tasks.

   - Instantiates a new DatabaseManager object. All UltraLite objects are created from the DatabaseManager object.
   - 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 and the schema file on the desktop.

     In this example, the locations are hard coded for convenience. In a real application, they would not be hard coded. In addition, these parameters are sufficient only for connections in the development environment; additional parameters are needed for the application to run on a Windows CE device. These additional parameters are described later in the tutorial.
   - 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.

```
public Customer() throws SQLException
{
  // Connect
  DatabaseManager dbMgr = new DatabaseManager();
  CreateParms parms = new CreateParms();
  parms.databaseOnDesktop = "c:\\tutorial\\java\\
         tutcustomer.udb";
  parms.schema.schemaOnDesktop = "c:\\tutorial\\java\\
         tutcustomer.usm";
  try {
    conn = dbMgr.openConnection( parms );
    System.out.println(
           "Connected to an existing database." );
  } catch( SQLException econn ) {
    if( econn.getErrorCode() ==
        SQLCode.SQLE_ULTRALITE_DATABASE_NOT_FOUND ){
      conn = dbMgr.createDatabase( parms );
      System.out.println(
           "Connected to a new database." );
    } else {
      throw econn;
    }
  }
}
```

4. Compile the Customer class.

   It is recommended that you use Sun JDK 1.1.8 or Personal Java 1.2 to compile the class. In addition, you must add the UltraLite library *jul9.jar* to your classpath. This library is in the *ultralite\NativeUltraLiteForJava* subdirectory of your SQL Anywhere installation.

   Enter the following command on a single line at a command prompt.

   ```
   javac -classpath
   "%ASANY9%\ultralite\NativeUltraLiteForJava\
           jul9.jar;%classpath%"
   Customer.java
   ```

5. Run the application.

   The classpath must include the UltraLite library *jul9.jar*, as in the previous step.

   The application must also be able to load the DLL that holds UltraLite native methods. The DLL is *jul9.dll* in the *ultralite\NativeUltraLiteForJava\win32* subdirectory of your SQL Anywhere 9 installation. This DLL can be in your system path or you can specify it on the java command line, as follows:

   ```
   java -classpath
   ".;%ASANY9%\ultralite\NativeUltraLiteForJava\jul9.jar"
   -Djul.library.dir=
   "%ASANY9%\ultralite\NativeUltraLiteForJava\win32" Customer
   ```

This command must be all on one line, with no spaces inside the individual arguments.

Lesson 2: Insert data into the database

The following procedure demonstrates how to add data to the database.

❖ To add rows to your database

1. Add the method below to the *Customer.java* file.

   This method carries out the following tasks:

   - Opens the table. You must open a Table object to carry out any operations on the table. To obtain a Table object, use the **connection.getTable()** method.
   - Obtains identifiers for some of the columns of the table. The other columns in the table can accept NULL values or have a default value. In this tutorial only required values are specified.
   - If the table is empty, adds two rows. To insert each row, the code sets the mode to insert mode using InsertBegin, sets values for each required column, and executes an insert to add the rows to the database.

     The commit method is not strictly needed here, as the default mode for applications is to commit operations after each insert. It has been added to the code to emphasize that if you turn off autocommit behavior you must commit a change for it to be permanent.
   - If the table is not empty, reports the number of rows in the table.
   - Closes the Table object.

```
private void insert() throws SQLException
{
  // Open the Customer table
  Table t = conn.getTable( "Customer" );
  t.open();

  short id = t.schema.getColumnID( "ID" );
  short fname = t.schema.getColumnID( "FName" );
  short lname = t.schema.getColumnID( "LName" );

  // Insert two rows if the table is empty
  if( t.getRowCount() == 0 ) {

      t.insertBegin();
      t.setString( fname, "Gene" );
      t.setString( lname, "Poole" );
      t.insert();

      t.insertBegin();
      t.setString( fname, "Penny" );
      t.setString( lname, "Stamp" );
      t.insert();

      conn.commit();
      System.out.println( "Two rows inserted." );
  } else {
      System.out.println( "The table has " +
      t.getRowCount() + " rows." );
  }
  t.close();
}
```

2. Call the insert method you have created.

   Add the following line to the `main()` method, immediately after the call to the Customer constructor.

   ```
   cust.insert();
   ```

3. Compile and run your application, as in “Lesson 1: Connect to the database” on page 7.

Lesson 3: Select the rows from the table

The following procedure demonstrates how to loop over the rows of a table. It retrieves rows from the table, and prints them on the command line.

❖ To list the rows in the table

1. Add the following method to the *Customer.java* file.

   This method carries out the following tasks.

   - Opens the Table object.
   - Retrieves the column identifiers.
   - Sets the current position before the first row of the table.

     Any operations on a table are carried out at the current position. The position may be before the first row, on one of the rows of the table, or after the last row. By default, as in this case, the rows are ordered by their primary key value (id). To order rows in a different way, you can add a new index to the database and open a table using that index.
   - For each row, the id and name are written out. The loop carries on until **moveNext** returns false, which happens after the final row.
   - Closes the Table object.

   ```
   private void select() throws SQLException
   {
     // Fetch rows
     // Open the Customer table
     Table t = conn.getTable( "Customer" );
     t.open();
     short id = t.schema.getColumnID( "ID" );
     short fname = t.schema.getColumnID( "FName" );
     short lname = t.schema.getColumnID( "LName" );
     t.moveBeforeFirst();
     while( t.moveNext() ) {
         System.out.println(
           "id= " + t.getInt( id )
           + ", name= " + t.getString( fname )
           + " " + t.getString( lname )
           );
     }
     t.close();
   }
   ```

2. Call the select function you have created.

   Add the following line to the main() method, immediately after the call to the insert method.

   ```
   cust.select();
   ```

3. Compile and run your application, as in “Lesson 1: Connect to the database” on page 7.

Lesson 4: Deploy your application to a Windows CE device

Running the sample application on Windows CE requires that the a supported Java VM be installed on your device.

The Jeode Runtime (Java VM) is packaged with some devices, including the HP/Compaq iPAQ and NEC MobilePro P300. CrEme Plus is also available and includes CrEme bundled with Symbol value-add capabilities. Both VMs can be purchased for other devices.

❖ To confirm that the Jeode VM is installed on your Windows CE device

1. Choose Start ➤ Programs.

   If the Jeode VM is installed, a Jeode folder appears in the Programs folder.

❖ To prepare your application to run on a Windows CE device

1. Specify the location of your database file and schema file by specifying the schemaOnCE and databaseOnCE parameters.

   In the Customer constructor, alter the CreateParms object to read as follows.

   ```
   CreateParms parms = new CreateParms();
   parms.databaseOnDesktop = "c:\\tutorial\\java\\
            tutcustomer.udb";
   parms.databaseOnCE = "\\UltraLite\\tutorial\\
            tutcustomer.udb";
   parms.schema.schemaOnDesktop = "c:\\tutorial\\java\\
            tutcustomer.usm";
   parms.schema.schemaOnCE = "\\UltraLite\\tutorial\\
            tutcustomer.usm"";
   ```

2. Compile your application as in “Lesson 1: Build the CustDB application” on page 25.

3. Run your application to check that no errors were introduced. The schemaOnCE and databaseOnCE parameters have no effect when running in a desktop environment.

4. Prepare a shortcut to run the application.

   A shortcut is a text file with *.lnk* extension, which contains a command line to run the application. In the next procedure, you copy this shortcut file to a location on the device.

Using a text editor, create a file named *tutorial.lnk* in your tutorial directory with the following content, which should all be on a single line.

```
18#"\Windows\evm.exe"
-Djeode.evm.console.local.keep=TRUE
-Djeode.evm.console.local.paging=TRUE
-Djul.library.dir=\UltraLite\lib
-cp \UltraLite\tutorial;\UltraLite\lib\jul9.jar
Customer
```

The meaning of the elements in this command are as follows.

- The first line starts the Jeode VM executable.
- The `-Djeode` options control the display of the text console that is used for output from the application.
- The `-Djul.library.dir` option directs the VM to the UltraLite native interface runtime library (*jul9.dll*).
- The `-cp` option provides the classpath for the VM. It indicates the location of the application and the UltraLite runtime library.
- The final argument is the class, which in this case is `Customer`.

The following procedure copies the files to your device. You must copy both UltraLite runtime files and application-specific files.

❖ To deploy the UltraLite runtime to the Windows CE device

1. Start Windows Explorer on your Windows CE device.
   - Ensure that your device is connected to your desktop computer.
   - In the ActiveSync window, click Explore. An Explorer window opens.
2. Create directories to hold the UltraLite runtime and application.
   - In the Explorer window, click My Pocket PC to access the root directory.
   - Create a directory named *UltraLite*.
   - Open the UltraLite directory and create subdirectories named *lib* and *tutorial*. The directory path *\UltraLite\lib* is the location for the UltraLite runtime files, and the path *\UltraLite\tutorial* is the location for the application. These directories match the options in the shortcut file described above.
3. Copy the UltraLite runtime files to the Windows CE device.
   - Start a separate Explorer window and navigate to the SQL Anywhere installation directory on our desktop machine.
   - Drag the following files from the desktop to the device

Desktop location relative to your SQL Anywhere directory	Windows CE location
UltraLite\NativeUltraLiteForJava\jul9.jar	\UltraLite\lib\jul9.jar
UltraLite\NativeUltraLiteForJava \ce\arm\jul9.dll	\UltraLite\lib\jul9.dll

As an alternative, you can deploy the UltraLite engine, an executable that permits concurrent multi-process access to UltraLite databases. If you use the UltraLite engine, you must deploy the appropriate version of the component (*julclient9.dll*) and you must specify the Runtime Type property in the Database Manager.

4. Copy the application files to the Windows CE device.
   - In your Explorer window, navigate to the tutorial directory.
   - Drag the following files from the desktop to the device

Desktop location	Windows CE location
Customer.class	\UltraLite\tutorial
tutcustomer.usm	\UltraLite\tutorial

In this tutorial, you do not copy the database file to the Windows CE device. Instead, you copy a schema file to the device and the application creates a database based on the schema file.

5. If you are running the CrEme VM and wish to use ActiveSync, add *\Windows\CrEme\lib\crmex.jar* to your classpath.

6. Copy the shortcut file to the Windows CE device.
   - Drag the following file from the desktop to the device.

Desktop location	Windows CE location
tutorial.lnk	\Windows\Start Menu

The following procedure runs the application

❖ To run the application

1. On the Windows CE device, choose Start ➤ tutorial.

   This shortcut is the *tutorial.lnk* file that you copied onto the device.
   - The Jeode VM loads and the console is displayed.
   - The following messages are printed onto the console.

```
Connected to a new database.
Two rows inserted.
id= 1, name= Gene Poole
id= 2, name= Penny Stamp
Application finished. Please close console
```

- Close the console.

2. On the Windows CE device, choose Start ➤ tutorial again.

   This time the first two messages are as follows.

   ```
   Connected to an existing database.
   The table has 2 rows.
   ```

3. Close the console.

Lesson 5: Add synchronization to your application

The following procedures add synchronization code to your application, starts the MobiLink synchronization server, and runs your application to synchronize.

> **Note**
> This lesson uses MobiLink synchronization, which is part of SQL Anywhere Studio. You must have SQL Anywhere Studio installed on your computer to carry out this lesson.

Synchronization is carried out with the ASA 9.0 Sample database. The ASA 9.0 Sample database has a Customer table whose columns match those in the customer table of your UltraLite database. During synchronization the data in that table is downloaded to your UltraLite application.

This lesson assumes some familiarity with MobiLink synchronization.

❖ To add synchronization to your application

1. Add the method below to *Customer.java.*

   This code carries out the following tasks.

   - Sets the synchronization stream to TCP/IP. Synchronization can also be carried out over HTTP, ActiveSync, or HTTPS. HTTPS synchronization requires that you obtain the separately licensable SQL Anywhere Studio security option.

     ☞ For more information, see **ianywhere.native_ultralite.StreamType** and **ianywhere.native_ultralite.Connection** in the "Native UltraLite for Java API Reference" on page 63.

     The syncParms field of the Connection object provides convenient access to a SyncParms object. For more information, see **ianywhere.native_ultralite.SyncParms** in the "Native UltraLite for Java API Reference" on page 63.

   - MobiLink synchronization is controlled by scripts at the MobiLink synchronization server. The script version identifies which set of scripts to use. The setSendColumnNames method together with an option on the MobiLink synchronization server causes the scripts to be generated automatically.

     ☞ For more information, see "Generating scripts automatically" [*MobiLink Administration Guide*, page 230].

   - Sets the MobiLink user name. This value is used for authentication at the MobiLink synchronization server, and is different from the

UltraLite database user ID, although in some applications you may wish to make the values the same.

- Sets the synchronization to only download data. By default, MobiLink synchronization is two-way. Here, we use download-only synchronization so that the rows in your table do not get uploaded to the sample database.

```
private void sync() throws SQLException {
  conn.syncParms.setStream( StreamType.TCPIP );
  conn.syncParms.setVersion( "ul_default" );
  conn.syncParms.setUserName( "sample" );
  conn.syncParms.setSendColumnNames( true );
  conn.syncParms.setDownloadOnly( true );
  conn.synchronize();
}
```

2. Call your sync method.

   Add the following line to the main() method, immediately after the call to the insert method and before the call to the select method.

```
cust.sync();
```

3. Compile your application, as in "Lesson 1: Connect to the database" on page 7. Do not run the application.

❖ To synchronize your data

1. Start the MobiLink synchronization server.

   From a command prompt, run the MobiLink following command line.

```
dbmlsrv9 -c "dsn=ASA 9.0 Sample" -v+ -zu+ -za
```

   The ASA 9.0 Sample database has a Customer table that matches the columns in the UltraLite database you have created. You can synchronize your UltraLite application with the ASA 9.0 Sample database.

   The `-zu+` and `-za` command line options provide automatic addition of users and generation of synchronization scripts.

   ☞ For more information about these options, see the "MobiLink Synchronization Server Options" [*MobiLink Administration Guide*, page 189].

2. Run your application, as in "Lesson 1: Connect to the database" on page 7.

   The MobiLink synchronization server window displays status messages indicating the synchronization progress. The final message displays `Synchronization complete`.

Summary

Learning accomplishments

During this tutorial, you:

- created a database schema
- created a Native UltraLite for Java application
- synchronized an UltraLite remote database with an Adaptive Server Anywhere consolidated database
- gained competence with the process of developing a Native UltraLite for Java application

Samples

- For more code samples, see *Samples\NativeUltraLiteForJava\Simple\Simple.java*.

CHAPTER 3

Tutorial: The CustDB Sample Application

About this chapter

This chapter provides a tutorial to guide you through the process of building and deploying a multi-table application using Native UltraLite for Java.

This tutorial also demonstrates synchronization between an UltraLite remote database and a consolidated database. The synchronization portion of this tutorial requires SQL Anywhere Studio.

Contents

Introduction

This tutorial guides you through the process of compiling, running, and deploying the CustDB sample application. It demonstrates how to implement several common tasks using Native UltraLite for Java.

Source code for CustDB is supplied in the *Samples\NativeUltraLiteForJava\CustDB* subdirectory of your SQL Anywhere installation.

Timing

The tutorial takes about 30 minutes if you copy and paste the code. If you enter the code yourself, it takes significantly longer.

Competencies and experience

This tutorial assumes:

- you have access a Windows CE device running the Jeode VM
- you have a Java development kit installed on your computer. JDK 1.1.8 or Personal Java 1.2 is recommended, as it is compatible with the Jeode VM used for deployment
- you know how to create an UltraLite schema using the UltraLite Schema Painter

  ☞ For more information, see "Lesson 1: Create an UltraLite database schema" [*UltraLite Database User's Guide*, page 130].

Note
The synchronization portion of this tutorial requires SQL Anywhere Studio.

Goals

The goals for this tutorial are to gain competence and familiarity with developing Native UltraLite for Java applications.

Lesson 1: Build the CustDB application

The following procedure uses the *build.bat* script, located in the *Samples\NativeUltraLiteForJava\CustDB* subdirectory of your installation, to build the CustDB sample from the source java files.

A *clean.bat* script is also provided for removing the results of the build.

❖ To build the sample

1. Open a command prompt and navigate to the *Samples\NativeUltraLiteForJava\CustDB* subdirectory of your SQL Anywhere installation.

   You should leave this command prompt open for the entire tutorial.

2. Set the required environment variables.

   Ensure the environment variables ASANY9 and JAVA_HOME are properly defined.

   - ASANY9 is set by the setup program to the location of your SQL Anywhere installation.
   - Set the JAVA_HOME environment variable to the location of a JDK on your computer. JDK 1.1.8 or Personal Java 1.2 is recommended, as it is compatible with the Jeode VM used for deployment.

     For example, if the JDK is version 1.1.8 in *c:\jdk1.1.8* run the following command.

     ```
     set JAVA_HOME=c:\jdk1.1.8
     ```

3. Build the application.

   At the command prompt, run the following command:

   ```
   build
   ```

   The *build* batch file carries out the following operations.

   - Compiles the CustDB sample code.

     The compiled classes are stored in the *builddir* subdirectory.
   - Puts the compiled classes into a JAR file.

     The JAR file is stored in the *CustDB* directory.
   - Creates a database schema file using the *ulinit* command line utility.

     The schema file is created from the Adaptive Server Anywhere UltraLite 9.0 Sample database.

     ☞ For more information about the ulinit tool, see "The ulinit utility" [*UltraLite Database User's Guide*, page 112].

The sample code

The sample code files are given below together with a brief description.

File	Description
Application.java	The main user interface frame, event processing, and ActiveSync support.
CustDB.java	The main interface to the database. Most of the database processing is in this file.
DialogDelOrder.java	The delete confirmation dialog.
DialogNewOrder.java	The new order entry form which shows populating a list box with data from the database.
Dialogs.java	The common base class for dialogs.
DialogSyncHost.java	This prompts for synchronization host name.
DialogUserID.java	This prompts for the Employee ID.
GetOrder.java	This is a class representing order data. It shows how to do a key join.
GetOrderData.java	This computes min and max order ID. Equivalent to SELECT max(order_id), min(order_id) FROM ULOrder.

Lesson 2: Run the CustDB application

The following procedure demonstrates how to run the CustDB sample on Windows. Deployment to a Windows CE device is described in "Lesson 3: Deploy CustDB to a Windows CE device" on page 28. This lesson uses MobiLink synchronization, and so requires SQL Anywhere Studio.

❖ To run the sample on Windows

1. From the same command prompt as in the previous lesson, type `win32\run.bat`.

   This command carries out the following operations.

   - Starts the MobiLink synchronization server.

     The server connects to the UltraLite 9.0 Sample database, which is used as a consolidated database for the CustDB application.

   - Starts the CustDB sample.

     The first time you run it, the application starts with no UltraLite database (*.udb* file) and so it creates this file from the UltraLite schema.

   - Displays a logon window.

     The window appears in the middle of the screen, but may be behind other windows. You may have to move other windows to locate the logon window.

2. Logon to the application.

   Click OK to logon as a user with MobiLink user name **50**. The application synchronizes and synchronization progress information is displayed. After a pause, a window with a single order is displayed.

3. Explore the application.

   You can move through the rows in the database, approve and deny orders, and synchronize. When you quit the application, the batch file shuts down the MobiLink synchronization server.

Lesson 3: Deploy CustDB to a Windows CE device

The following procedure deploys your application to a Windows CE device. Deployment requires the Jeode Runtime (Java VM).

❖ To deploy the application to a Windows CE device

1. Start Windows Explorer on your Windows CE device.
   - Ensure that your device is connected to your desktop computer.
   - In the ActiveSync window, click Explore.

     An Explorer window opens.
2. Create directories to hold the UltraLite runtime and application.

   If you have completed the previous tutorial, some of these directories may already exist.
   - In the Explorer window, click My Pocket PC. This is the root directory, and has a path of \.
   - Create a directory named *UltraLite*.
   - Open the UltraLite directory and create subdirectories named *lib* and *CustDB*.

     \UltraLite\lib is the location for the UltraLite runtime files, and *\UltraLite\CustDB* is the location for the application. These directories match the options in the shortcut file provided in the *ce* subdirectory.
3. Copy the UltraLite runtime files to the Windows CE device.

   If you have completed the previous tutorial, you may have already carried out this operation. You do not need to repeat the step.
   - Start a separate Explorer window and navigate to the SQL Anywhere installation directory on your desktop machine.
   - Drag the following files from the desktop to the device.

Desktop location relative to your SQL Anywhere directory	Windows CE location
UltraLite\NativeUltraLiteForJava\jul9.jar	*\UltraLite\lib*
UltraLite\NativeUltraLiteForJava\ce\arm\jul9.dll	*\UltraLite\lib*

4. Copy the application files to the Windows CE device.
   - In your Explorer window, navigate to the *Samples\NativeUltraLiteForJava\CustDB* directory.
   - Drag the following files from the desktop to the device.

Desktop location	Windows CE location
custdb.jar	*\UltraLite\CustDB*
ul_custapi.usm	*\UltraLite\CustDB*

In this tutorial, you do not copy the database file to the Windows CE device. Instead, you deploy a schema file to the device and the application creates a database.

5. Copy the shortcut file to the Windows CE device.
   - Drag the following file from the desktop to the device.

Desktop location relative to your SQL Anywhere directory	Windows CE location
ce\CustDB.lnk	*\Windows\Start Menu*

6. Install the ActiveSync provider.

   The ActiveSync provider is required for synchronization.

   ☞ For instructions, see "ActiveSync provider installation utility" [*MobiLink Clients*, page 28].

7. Register the application for use with ActiveSync.

   When registering CustDB, use the following properties.

Property	Value
Name	JULCustDB
Class Name	JULCustDB
File Location	*\Windows\evm.exe*
Arguments	-Djeode.evm.console.local.keep=TRUE -Djul.library.-dir=\UltraLite\lib -cp \UltraLite\CustDB\custdb.-jar;\UltraLite\lib\jul9.jar custdb.Application ACTIVE_SYNC_LAUNCH

   You can run the batch file *Samples\NativeUltraLiteForJava\CustDB\ce\as_register.bat* to carry out this operation.

SQL Anywhere Studio users
For the setup steps for ActiveSync synchronization, see "ActiveSync provider installation utility" [*MobiLink Clients*, page 28].

❖ To run the application

1. Start the MobiLink synchronization server.

   Run the *startdb.bat* batch file, located in the *Samples\NativeUltraLiteForJava\CustDB\ce* subdirectory of your SQL Anywhere installation.

2. On the CE device, run the *CustDB* application from the Start menu.

Summary

During this tutorial, you:

- Built and ran the CustDB sample on your desktop computer.
- Deployed a Native UltraLite for Java application to a Windows CE device.

CHAPTER 4

Understanding Native UltraLite for Java Development

About this chapter

This chapter explains how to develop applications using Native UltraLite for Java.

☞ For hands-on tutorials, see "Tutorial: Build a Native UltraLite for Java Application" on page 5 or "Tutorial: The CustDB Sample Application" on page 23.

Contents

UltraLite database schemas

The database schema is a description of the database. It is the collection of tables, indexes, keys, and publications within the database, and all the relationships between them.

You do not alter the schema of an UltraLite database directly. Instead, you create a schema (*.usm*) file and upgrade the database schema from that file by calling a built-in UltraLite function in your application.

A schema file is also used in the initial creation of a database to specify the structure of the database.

Creating UltraLite database schema files

You can create an UltraLite schema file using the UltraLite Schema Painter or the *ulinit* utility.

- **UltraLite Schema Painter** The UltraLite Schema Painter is a graphical utility for creating and editing UltraLite schema files.

  To start the Schema painter, choose Start ➤ Programs ➤ SQL Anywhere 9 ➤ UltraLite ➤ UltraLite Schema Painter, or double-click a schema (*.usm*) file in Windows Explorer.

  ☞ For more information about using the UltraLite Schema Painter, see "Lesson 1: Create an UltraLite database schema" [*UltraLite Database User's Guide*, page 130].

- **The ulinit utility** If you have the Adaptive Server Anywhere database management system, you can generate an UltraLite schema file using the *ulinit* command line utility.

  ☞ For more information about using the *ulinit* utility, see "The ulinit utility" [*UltraLite Database User's Guide*, page 112].

Upgrading your database schema

To modify your existing database structure, use the DatabaseSchema.applyFile method. In most cases there will be no data loss, however, data loss can occur if columns are deleted or if the data type for a column is changed to an incompatible type.

☞ For more information, see DatabaseSchema.ApplyFile in the "Native UltraLite for Java API Reference" on page 63.

Example

The following code applies a new schema file using the ApplyFile method. It assumes that you have already connected to the database as shown in "Connecting to a database" on page 36.

```
DatabaseSchema schema = conn.schema;
SchemaParms parms = new SchemaParms();
parms.schemaOnDesktop = "c:\tutorial\tutcustomer.usm";
try {
  schema.applyFile(parms );
} catch {
  // handle any errors
}
```

Connecting to a database

UltraLite applications must connect to a database before carrying out operations on the data in it. This section describes how to connect to an UltraLite database.

Using the Connection object

The following properties of the Connection object govern global application behavior.

- **Commit behavior** By default, Native UltraLite for Java 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 51.

- **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" on page 54.

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

  ☞ For more information, see "Synchronizing UltraLite applications" on page 55.

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

  ☞ For more information, see "Accessing and manipulating data with the Table API" on page 45.

- **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 "Accessing and manipulating data using dynamic SQL" on page 40.

☞ For more information, see **ianywhere.native_ultralite.Connection in the** "Native UltraLite for Java API Reference" on page 63

Multi-threaded applications

Each Connection and all objects created from it should be used on a single thread. If your application requires multiple threads accessing the UltraLite database, each thread requires a separate connection.

❖ To connect to an UltraLite database

1. Create and initialize a DatabaseManager object.

   The DatabaseManager object is at the root of the object hierarchy. You create only one DatabaseManager object per application. It is often best to declare the DatabaseManager object as global to the application.

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

   ☞ For more information, see **ianywhere.native_ultralite.DatabaseManager** in the "Native UltraLite for Java API Reference" on page 63.

2. Declare a Connection object.

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

   ```
   static Connection conn;
   ```

3. Open a connection to an existing database, or create a new database if the specified database file does not exist.

   Most UltraLite applications deploy a schema file rather than a database file, and let UltraLite create the database file on the first connection attempt. Thus, the following code attempts to connect to an existing database. If the database file does not exist, the application creates a database file.

   ```
   ConnectionParms parms = new ConnectionParms();
   // specify the location of the database file
   parms.databaseOnDesktop = "mydata.udb";
   try {
     conn = dbMgr.openConnection( parms );
   } catch( SQLException econn ) {
     if( econn.getErrorCode() ==
       SQLCode.SQLE_ULTRALITE_DATABASE_NOT_FOUND ){
       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*. If the database does not exist, it is created using the UltraLite schema file named *mydata.usm*.

☞ For more information, see the code in *Samples\NativeUltraLiteForJava\Simple\Simple.java*.

```
CreateParms parms = new CreateParms();
parms.databaseOnDesktop = "mydata.udb";
parms.schema.schemaOnDesktop = "mydata.usm";
try {
  conn = dbMgr.openConnection( parms );
  System.out.println(
    "Connected to an existing database." );
}
catch( SQLException econn ) {
  if( econn.getErrorCode() ==
      SQLCode.SQLE_ULTRALITE_DATABASE_NOT_FOUND ){
    conn = dbMgr.createDatabase( parms );
    System.out.println(
    "Connected to a new database." );
  } else {
    throw econn;
  }
}
```

Encryption and obfuscation

You can encrypt or obfuscate your UltraLite database using Native UltraLite for Java.

Encryption

To create a database with encryption, specify an encryption key using the **ianywhere.native_ultralite.ConnectionParms.EncryptionKey** property, then call the **createDatabase** method. When you call the **createDatabase** method, the database is created and encrypted with the specified key.

☞ For more information, see "Encryption Key connection parameter " [*UltraLite Database User's Guide*, page 75].

You can change the encryption key by specifying the new encryption key with the **ianywhere.native_ultralite.Connection.changeEncryptionKey** method.

☞ For more information, see **ianywhere.native_ultralite.Connection** in the "Native UltraLite for Java API Reference" on page 63.

After the database is encrypted, connections to the database must specify the correct encryption key. Otherwise, the connection fails.

Obfuscation

To obfuscate the database, specify obfuscate=1 as a creation parameter.

☞ For more information about database encryption, see "Encrypting UltraLite databases" [*UltraLite Database User's Guide*, page 36].

Accessing and manipulating data using dynamic SQL

UltraLite applications can access table data using dynamic SQL or the Table API. This section describes data access using dynamic SQL.

☞ For information about using the Table API, see "Accessing and manipulating data with the Table API" on page 45.

This section explains how to perform the following tasks using dynamic SQL.

- 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 159].

☞ For an overview of the sequence of operations required for any SQL operation, see "Using dynamic SQL" [*UltraLite Database User's Guide*, page 161].

Data manipulation: INSERT, UPDATE and DELETE

With UltraLite, you can perform SQL Data Manipulation Language operations. These operations are performed using the PreparedStatement.executeStatement method.

☞ For more information, see **ianywhere.native_ultralite.PreparedStatement** in the "Native UltraLite for Java API Reference" on page 63.

Using parameters in your prepared statements

Placeholders for parameters in SQL statements are indicated the ? character. For any INSERT, UPDATE or DELETE, each ? is referenced according to its ordinal position in the prepared statement. For example, the first ? is referred to as 1, and the second as 2.

❖ To INSERT a row

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 disabled autoCommit, commit the change.

```
conn.commit();
```

❖ To UPDATE a row

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 disabled autoCommit, commit the change.

```
conn.commit();
```

❖ To DELETE a row

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 disabled autoCommit, commit the change.

```
conn.commit();
```

Data retrieval: SELECT

The SELECT statement allows you to retrieve information from the database. This section describes how to execute a SELECT statement and how to handle the result set it returns.

❖ To execute a SELECT statement

1. Declare a PreparedStatement object, which holds the query.

```
PreparedStatement prepStmt;
```

2. Assign a statement to the 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.out.print(
        customerNames.getString( i ) + " "
        );

  }
  System.out.println();
}
```

Navigating dynamic SQL result sets

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

The result set object provides you with the following methods to navigate a 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, but allows you to repopulate the row buffer.

Result set schema description

The ResultSet.schema property allows you to retrieve information about a result set, such as column names, total number of columns, column scales, column sizes and column SQL types.

Example

The following example demonstrates how to use the ResultSet.schema property to display schema information in a console window.

```
for ( int i = 1;
      i <= MyResultSet.schema.getColumnCount();
      i++ ) {
   System.out.println(
      MyResultSet.schema.getColumnName(i) + " " +
      MyResultSet.schema.getColumnSQLType(i)
   );
}
```

Accessing and manipulating data with the Table API

UltraLite applications can access table data using dynamic SQL or the Table API. This section describes data access using the Table API.

☞ For information about dynamic SQL, see "Accessing and manipulating data using dynamic SQL" on page 40.

This section explains how to perform the following tasks using the Table API.

- 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.

Navigating the rows of a table

Native UltraLite for Java provides you with a number of methods to navigate a table in order to perform a wide range of navigation tasks.

The table object provides you with the following methods to navigate a table.

- **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 table, relative to the current position of the cursor in the table, and negative offset values move backward in the table. An offset value of zero does not move the cursor, but allows you to repopulate the row buffer.

Example

The following code opens the MyTable table and displays 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.out.println( t.getString( colID ) );
}
```

You expose the rows of the table to the application when you open the table object. By default, the rows are ordered by primary key value, but you can specify an index when opening a table to access the rows in a particular order.

Example

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.native_ultralite.Table** and **ianywhere.native_ultralite.TableSchema** in the "Native UltraLite for Java API Reference" on page 63.

Using UltraLite modes

UltraLite mode determines the purpose for which the values in the buffer will be used. UltraLite has the following four modes of operation, in addition to a default mode.

- **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** Used to locate a row whose value exactly matches the data in the buffer when one of the find methods is called.
- **Lookup mode** Used to locate a row whose value matches or is greater than the data in the buffer when one of the lookup methods is called.

Accessing the values of the current row

A Table object is always located 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 retrieve or modify the value of each column.

Retrieving column values

The Table object provides a set of methods for retrieving column values. These methods take the column ID as argument.

Example

The following code retrieves the value of the lname column, which is a character string.

```
short lname = t.schema.getColumnID( "lname" );
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 );
```

Modifying column values

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

Example

For example, the following code sets the value of the lname column to Kaminski.

```
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
tCustomer.moveBeforeFirst();
id = tCustomer.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 rows with find and lookup

UltraLite has several modes of operation for working with data. Two of these modes, the find and lookup modes, are used for searching. The Table object has methods corresponding to these modes for locating particular rows in a table.

Note
The columns searched using Find and Lookup methods must be in the index used to open the table.

- **Find 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, the application is positioned before the first or after the last row.
- **Lookup 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.

❖ To search for a row

1. Enter find or lookup mode.

   The mode is entered by calling a method on the table object. For example, the following code enters find mode.

   ```
   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, the following code sets the value in the buffer to Kaminski.

   ```
   short lname = t.schema.getColumnID( "lname" );
   t.setString( lname, "Kaminski" );
   ```

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.

   For multi-column indexes, a value for the first column is always used, but you can omit the other columns.

   ```
   tCustomer.findFirst();
   ```

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 **ianywhere.native_ultralite.Table** in the "Native UltraLite for Java API Reference" on page 63.

Updating rows

The following procedure updates a row.

❖ To update a row

1. Move to the row you wish to update.

   You can move to a row by scrolling through the table or by searching the table 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, the following instruction sets the id column in the buffer to 3.

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

4. Execute the Update.

   ```
   t.update();
   ```

After the update operation the current row is the row that has been 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, Native UltraLite for Java 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 51.

> ***Caution***
> *You cannot 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 a row in the table before carrying out the insert operation. The order of row insertion into the table has no significance.

Example

The following code 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, one of the following entries is used.

- 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.

For update operations, an insert is applied to the database in permanent storage 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 following procedure deletes a row.

❖ To delete a row

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

```
t.delete();
```

Transaction processing in UltraLite

UltraLite provides transaction processing to ensure the integrity of the data in your database. A transaction is a logical unit of work. Either an entire transaction is executed or none of the statements in the transaction are executed.

By default, Native UltraLite for Java operates in autocommit mode, so that each insert, update, or delete is executed as a separate transaction. Once the operation is complete, 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 statement 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.native_ultralite.Connection** class in the "Native UltraLite for Java API Reference" on page 63.

Accessing schema information

The 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.

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

☞ For information about modifying the schema, see "Upgrading your database schema" on page 34.

You can access the following schema objects and information.

- **DatabaseSchema** exposes 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.

  ☞ For more information, see **ianywhere.native_ultralite.Connection** in the "Native UltraLite for Java API Reference" on page 63.

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

  To obtain a TableSchema object, access Table.schema.

- **IndexSchema** Information about the column in the index. As an index has no data directly associated with it there is no separate Index class, just an IndexSchema class.

  To obtain an IndexSchema object, call the TableSchema.getIndex, the TableSchema.getOptimalIndex, or the TableSchema.getPrimaryKey method.

- **PublicationSchema** A list of the tables and columns contained in a publication. Publications are also comprised of schema only, and so there is no Publication object.

  To obtain a PublicationSchema object, call the DatabaseSchema.getPublicationSchema method.

☞ For more information, see **ianywhere.native_ultralite.TableSchema** in the "Native UltraLite for Java API Reference" on page 63.

Error handling

You can use the standard Java error-handling features to handle errors. Most methods throw java.sql.SQLException errors. You can use SQLException.getErrorCode() to retrieve the SQLCode value assigned to this error. SQLCode errors are negative numbers indicating the error type.

☞ For a list of error codes, see *ASA Error Messages*.

After synchronization, you can use the SyncResult property of the connection to obtain more detailed error information.

☞ For more information, see the following:

- **ianywhere.native_ultralite.SyncResult** in the "Native UltraLite for Java API Reference" on page 63

User authentication

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.

You cannot change a user ID. Instead, you add a user and delete an existing user. A maximum of four user IDs are permitted for each UltraLite database.

☞ For more information, see "User authentication in UltraLite" [*UltraLite Database User's Guide*, page 40].

❖ To add a user or change a password for an existing user

1. Connect to the database as a user with DBA authority.
2. Grant the user access to the database with the desired password using the Connection.grantConnectTo method.

   This procedure is the same whether you are adding a new user or changing the password of an existing user.

   ☞ For more information, see **ianywhere.native_ultralite.Connection** in the "Native UltraLite for Java API Reference" on page 63.

❖ To delete an existing user

1. Connect to the database as a user with DBA authority.
2. Revoke the user's connection authority using the Connection.revokeConnectFrom method.

Synchronizing UltraLite applications

You synchronize UltraLite applications with a central consolidated database. Synchronization requires the MobiLink synchronization software included with SQL Anywhere Studio.

This section provides a brief introduction to synchronization and describes some features of particular interest to users of Native UltraLite for Java.

☞ For a more detailed explanation of synchronization, see "UltraLite Clients" [*MobiLink Clients*, page 277].

You can also find a working example of synchronization in the CustDB sample application. For more information, see the *Samples\Ultralite\NativeUltraLiteForJava\CustDB* subdirectory of your SQL Anywhere 9 installation.

Native UltraLite for Java supports TCP/IP, HTTP, and HTTPS synchronization. Synchronization is initiated by the UltraLite application. In all cases, you use properties of the SyncParms object to control synchronization.

Note

To synchronize using HTTPS you must obtain the separately-licensable security option. To order this option, see the card in your SQL Anywhere Studio package or see *http://www.sybase.com/detail?id=1015780*.

☞ For more information, see "Welcome to SQL Anywhere Studio" [*Introducing SQL Anywhere Studio*, page 4].

Adding ActiveSync synchronization to your application

This section explains how to add ActiveSync to a Native UltraLite for Java application, and how to register your application for use with ActiveSync on your end users' computers. ActiveSync is available on Windows CE devices through the Jeode and CrEme Java VMs.

If you are using the CrEme VM and wish to use ActiveSync, you must add *\Windows\CrEme\lib\crmex.jar* to your classpath.

Synchronization requires SQL Anywhere Studio. For general information about setting up ActiveSync synchronization, see "Deploying applications that use ActiveSync" [*MobiLink Clients*, page 312]. For general information on adding synchronization to an application, see "UltraLite Clients" [*MobiLink Clients*, page 277].

ActiveSync synchronization can only be initiated by ActiveSync.

ActiveSync initiates 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 the setActiveSyncListener method, where MyAppClassName is a unique Windows class name for the application..

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

☞ For more information, see **anywhere.native_ultralite.DatabaseManager** in the "Native UltraLite for Java API Reference" on page 63.

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 Java **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 syncInfo 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.
- **Path** The path to the Jeode VM *(\Windows\evm.exe*).
- **Arguments** Includes the classpath (`-cp`) and other Jeode command line arguments, the application name and applications arguments.

  If you specify unique arguments to indicate ActiveSync activation, your application can carry out a special startup sequence knowing that it is to close upon the completion of ActiveSync synchronization.

CustDB and ActiveSync

The Native UltraLite for Java version of the CustDB sample provides examples of synchronization through an application menu using a socket and through ActiveSync.

You can find source code for this sample in *Samples\NativeUltraLiteForJava\CustDB\Application.java*, located under your SQL Anywhere directory. This section describes the code in that sample.

- CustDB parses its arguments to check for a special flag used to indicate it was launched by the MobiLink provider for ActiveSync. This allows it to streamline initialization (such as avoiding form population), since applications launched for ActiveSync are expected to shut down once they have synchronized.

```
// Normal versus Active sync launch
boolean isNormalLaunch = true;
int alen = args.length;
if( alen > 0 ) {
  String asflag = args[ alen - 1 ].toUpperCase();
  if( asflag.compareTo( "ACTIVE_SYNC_LAUNCH" ) == 0) {
    isNormalLaunch = false;
    --alen;
  }
```

- For normal launches (that is, non-ActiveSync launches), CustDB performs the connection initialization and determines the employee ID. It then initializes for ActiveSync by specifying a listener and loads its main form. For ActiveSync launches, CustDB performs the ActiveSync synchronization then shuts down.

```
if( isNormalLaunch ) {
  db.initActiveSync( "JULCustDB", main );
  db.getOrder( 1 );
} else {
  // ActiveSync launch
  db.activeSync( false );
  main.quit();
}
public void initActiveSync(
   String appName, ActiveSyncListener listener )
{
  DEBUG( "initActiveSync" );
  _conn.setActiveSyncListener( appName, listener );
}
public void activeSync( boolean useDialog )
{
  try {
    // Change stream
    _conn.syncInfo.setStream(
                      StreamType.ACTIVE_SYNC );
    // since if "stream=" not in parms,
    //it defaults to tcpip, no
    // need to change stream parms
    _conn.synchronize( useDialog );
    freeLists();
    allocateLists();
    skipToValidOrder();
  } catch( SQLException e ) {
    System.out.println(
    "Can't synchronize, sql code=" +
    e.getErrorCode()
      );
  }
}
```

- The Application class implements the ActiveSyncListener interface so the running application can be notified to perform an ActiveSync synchronization.

```
public class Application
extends Frame
implements ActionListener,
// ActiveSyncListener functional only on CE devices
ActiveSyncListener
```

- When activeSyncInvoked() is invoked, it posts a message to the UI thread.

```
static final int ACTIVE_SYNC_EVENT_MASK =
   AWTEvent.RESERVED_ID_MAX + 1;
static class ActiveSyncEvent extends AWTEvent {
   ActiveSyncEvent( Object source ) {
      super( source, ACTIVE_SYNC_EVENT_MASK );
   }

/** Process ActiveSync message
*/
public void activeSyncInvoked(
        boolean launchedByProvider ) {
  // This method is invoked on a special thread.
  // Post an event so that active sync
  // takes places on the same thread
  // as the rest of the application.
  DEBUG( "activeSyncInvoked()" );
  getToolkit().getSystemEventQueue().postEvent(
            new ActiveSyncEvent( this )
  );
  DEBUG( "ActiveSync Event posted" );
}
```

- The UI thread catches the message by overriding processEvent

```
/** Intercept my special action events
 *    for ActiveSync
 */
protected void processEvent( AWTEvent e ) {
  if( e instanceof ActiveSyncEvent ) {
    _db.activeSync( true );
    refresh();
  } else {
    super.processEvent( e );
  }
}
```

However, for the application to receive the event, it must be enabled. This is done in Application's constructor.

```
// ActiveSync support
enableEvents( ACTIVE_SYNC_EVENT_MASK );
```

Developing applications with Borland JBuilder

Borland JBuilder is a development environment for Java applications. Native UltraLite for Java includes integration with JBuilder 7 and JBuilder 8. This section describes how to use Native UltraLite for Java within the JBuilder environment.

Preparing to use Native UltraLite for Java with JBuilder

If JBuilder is installed, the UltraLite setup program enables JBuilder integration. If JBuilder is installed after the UltraLite, re-run the UltraLite setup program to enable JBuilder integration.

Setting the JDK

You should use JDK 1.1.8 or Personal Java 1.2 when developing Native UltraLite for Java applications, for compatibility with the Jeode VM on Windows CE devices.

JBuilder SE and Enterprise fully support JDK switching, while JBuilder Personal allows you to edit a single JDK.

❖ **To set the JDK version**

1. In JBuilder Personal, click Tools ➤ Configure JDKs and edit the JDK.
2. In JBuilder SE and Enterprise, right-click the project file in the project pane and choose Properties. On the Paths tab, click JDK and browse to your desired JDK location.

Using the Native UltraLite for Java setup

The Native UltraLite for Java setup can be used with existing projects or with new projects.

❖ **To add UltraLite features to a JBuilder project**

1. Open a JBuilder project.
2. Add the Native UltraLite for Java setup.
   - Choose File ➤ New. The Object Gallery appears.
   - On the General tab, select Native UltraLite Setup and click OK.
   - Choose a database name and deployment directories on the Windows CE device. Click Next.
   - Add names for the Windows CE shortcut (which will be displayed on the Start menu), the JAR name, and the main class. Click Finish.

   A link file is added to your project. This link file is used to run the application on the Windows CE device.

The Native UltraLite for Java setup makes the following changes to your JBuilder project.

- Adds Native UltraLite for Java in the list of available libraries.
- Adds project properties for code insight templates.
- Modifies the runtime configurations to locate *jul9.dll* when you run the application from within the development environment.

Using Native UltraLite for Java templates

During development, you can use Native UltraLite for Java code templates for some of the standard parts of your code. To use a template, type the template name at the appropriate place in your .java file and type CTRL+J to expand the template.

The following templates are provided.

- **julimp** Adds a line to import the Native UltraLite for Java package. Use this template in the imports section of your files.
- **juldb** Adds code to declare a DatabaseManager object.
- **julconn** Adds code to connect to to a database.
- **julskel** Adds both the juldb and julconn code, as a `main` method.

Accessing Native UltraLite for Java utilities from JBuilder

You can access the following Native UltraLite for Java utilities from the JBuilder interface.

- **Schema Painter** The UltraLite schema painter is a tool for creating and editing database definitions.

  To open the schema painter, choose Tools ➤ UltraLite Schema Painter.

- **Online help** This documentation is available from the interface.

  To open the online help, choose Help ➤ Native UltraLite Reference.

CHAPTER 5

Native UltraLite for Java API Reference

About this chapter

The Native UltraLite for Java package is named ianywhere.native_ultralite. The Native UltraLite for Java API reference is supplied in Javadoc format in the *docs\javadocs\NativeUltraLiteForJava* subdirectory of your SQL Anywhere 9 installation.

For more information, see the following classes:

- **ianywhere.native_ultralite.ActiveSyncListener**
- **ianywhere.native_ultralite.AuthStatusCode**
- **ianywhere.native_ultralite.Connection**
- **ianywhere.native_ultralite.ConnectionParms**
- **ianywhere.native_ultralite.CreateParms**
- **ianywhere.native_ultralite.Cursor**
- **ianywhere.native_ultralite.CursorSchema**
- **ianywhere.native_ultralite.DatabaseManager**
- **ianywhere.native_ultralite.DatabaseNameParms**
- **ianywhere.native_ultralite.DatabaseSchema**
- **ianywhere.native_ultralite.IndexSchema**
- **ianywhere.native_ultralite.PreparedStatement**
- **ianywhere.native_ultralite.ResultSet**
- **ianywhere.native_ultralite.ResultSetSchema**
- **ianywhere.native_ultralite.SchemaParms**
- **ianywhere.native_ultralite.SchemaUpgradeData**
- **ianywhere.native_ultralite.SchemaUpgradeListener**
- **ianywhere.native_ultralite.SQLCode**
- **ianywhere.native_ultralite.SQLType**
- **ianywhere.native_ultralite.StreamErrorCode**
- **ianywhere.native_ultralite.StreamErrorID**
- **ianywhere.native_ultralite.StreamType**
- **ianywhere.native_ultralite.SyncParms**
- **ianywhere.native_ultralite.SyncProgressData**
- **ianywhere.native_ultralite.SyncProgressDialog**
- **ianywhere.native_ultralite.SyncProgressListener**
- **ianywhere.native_ultralite.SyncResult**
- **ianywhere.native_ultralite.Table**
- **ianywhere.native_ultralite.TableSchema**

- **ianywhere.native_ultralite.UUID**

Index

E

F

G

I

J

L

M

N

O

P

R

S

T

U

V