
# UltraLite™ for Java User's Guide

Last modified: October 2002
Part Number: 36294-01-0802-01

Last modified October 2002. Part number 36294-01-0802-01.

# Contents

# About This Manual

| | |
|---|---|
| Subject | This manual describes Native UltraLite for Java, which is part of the UltraLite Component Suite. With Native UltraLite for Java you can develop and deploy database applications to handheld, mobile, or embedded devices running a Java VM. In particular, you can deploy applications to Windows CE devices running the Jeode VM. |
| 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. |

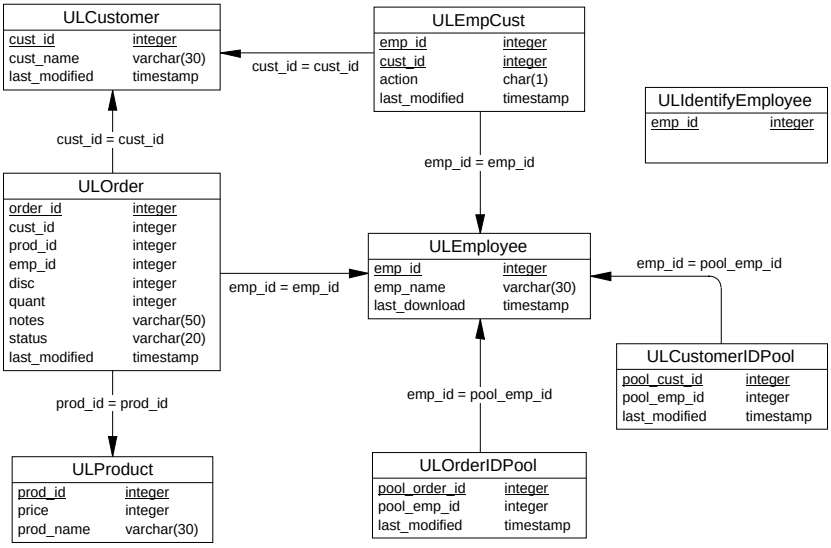
# The UltraLite sample database

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

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 as *Samples\NativeUltraLiteForJava\CustDB*.

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 beta program.

You can provide feedback on this documentation and on the software through a newsgroup and via e-mail. The newsgroup can be found on the *forums.sybase.com* news server as news://forums.sybase.com/ianywhere.private.ultralitetools.beta. The e-mail address is ulbeta@ianywhere.com.

**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 Native UltraLite for Java

About this chapter

This chapter introduces you to Native UltraLite for Java. It assumes that you are familiar with the UltraLite Component Suite, as described in "Introduction to the UltraLite Component Suite" on page 1 of the book *UltraLite Foundations*.

Contents

# Native UltraLite for Java features

Native UltraLite for Java is a member of the UltraLite Component Suite. It provides the following benefits for developers targeting small devices:

- a robust relational database store.
- Java programming ease-of-use with native performance
- deployment on the Windows CE platform

For more information on the features and benefits of the UltraLite Component Suite, see "Introduction to the UltraLite Component Suite" on page 2 of the book *UltraLite Foundations*.

## UltraLite and Java

Java developers wishing to take advantage of UltraLite database features have two options. UltraLite for Java is an existing UltraLite technology described in the *UltraLite 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 combination provides the benefits of Java development together 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 underying CPU and operating system. In contrast, the UltraLite runtime for UltraLite Java described in the *UltraLite User's Guide* 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. UltraLite Java described in the *UltraLite User's Guide* uses JDBC as the programming interface.
- **Windows CE deployment** Native UltraLite for Java has been developed with Windows CE as a deployment target. It supports the Jeode VM provided with many Windows CE devices. It also supports ActiveSync synchronization.

# Native UltraLite for Java architecture

The Native UltraLite for Java package is named **ianywhere.native_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 Native UltraLite for Java application.

  For more information, see **ianywhere.native_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.native_ultralite.Connection** in the API Reference.

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

  For more information, see **ianywhere.native_ultralite.Table** in the API Reference.

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

  For more information, see **ianywhere.native_ultralite.SyncParms** in the API Reference.

The API Reference is supplied in Javadoc format in the *UltraLite\NativeUltraLiteForJava\doc* subdirectory of your SQL Anywhere installation and is accessible from the front page of the UltraLite Component Suite online books.

## Deployment and supported platforms

Native UltraLite for Java supports the Jeode VM on Windows CE/ARM devices including the Compaq iPaq and NEC MobilePro P300, which come supplied with the Jeode VM. Windows operating systems other than Windows CE are supported for testing and development purposes only.

During development, it is recommended that you use JDK 1.1.8 or PersonalJava 1.2, as these are compatible with the Jeode VM.

In addition to your application code and the Jeode VM, you must deploy the following files to your Windows CE device:

- **jul8.jar** This JAR file contains the Native UltraLite for Java package and a utility package.
- **jul8.dll** The UltraLite runtime library.

- **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 "Lesson 5: Deploy your application to a Windows CE device" on page 16.

CHAPTER 2

# Tutorial: A Native UltraLite for Java Application

About this chapter

This chapter walks you through all the steps of building a simple Native UltraLite for Java application.

Contents

# Introduction

This tutorial walks you through building a Native UltraLite for Java application.

**Timing**

The tutorial takes about 45 minutes.

**Competencies and experience**

This tutorial assumes:

- you are familiar with the Java programming language
- you have JDK 1.1.8 installed on your machine

**Goals**

The goals for the tutorial are to gain competence and familiarity with the process of developing an 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 51.

# Lesson 1: Create a database schema

A **schema** is a database definitions without the data. You create an UltraLite schema file as a necessary first step to making an UltraLite database.

When creating UltraLite schemas, the following information is necessary:

Two utilities allow you to create schemas: *ulinit* allows you to generate an UltraLite database schema from an Adaptive Server Anywhere database, while the UltraLite Schema Painter allows you to design an UltraLite database schema from scratch. In this tutorial you use the UltraLite schema painter.

## Create your schema file using the UltraLite Schema Painter

❖ **To create the schema file using the UltraLite Schema Painter:**

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

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

2. Start the UltraLite Schema Painter:

   Click Start➤Programs➤Sybase SQL Anywhere 8➤UltraLite ➤ UltraLite Schema Painter.

3. Create a schema file.

   - Open the Tools folder and double-click Create UltraLite schema.
   - In the file dialog box, type **c:\tutorial\tutcustomer.usm** or Browse to the folder and enter the name **tutcustomer**.
   - Choose the collation sequence for your database. A collation sequence is a character set and sorting order. For languages based on the Roman alphabet, such as most European languages, choose **1252LATIN1 - Code Page 122, Windows Latin 1, Western**.
   - Leave the checkbox unchecked, so that the database is case insensitive.
   - Click OK to create the schema.

4. Create a table called *customer*.

   - Expand the tutorial item in the left pane of the UltraLite Schema Painter and select the *Tables* folder.

- Open the Tables folder and double-click Add Table. The New Table dialog appears.
- Enter the name *customer*.
- In the New Table dialog, add columns with the following properties.

| Column name | Data type (Size) | Column Allows NULL values? | Default value |
|---|---|---|---|
| id | integer | No | autoincrement |
| fnamee | char (15) | No | None |
| lnamee | char (20) | No | None |
| city | char (20) | Yes | None |
| phone | char (12) | Yes | 555-1234 |

- Set *id* as the primary key: Click Primary Key and add *id* to the index, marking it as ascending.
- Check your work and click OK to complete the table definition and dismiss the New Table dialog.

5 Click File ➤Save to save the *tutcustomer.usm* file.

You have now defined the schema of your UltraLite database. Although this database contains only a single table, you can use many tables in UltraLite databases.

# Lesson 2: Connect to the database

In this lesson you write, compile, and run a Java application that connects to a database using the schema you just 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.

❖ **To connect to an UltraLite database:**

1 In your tutorial directory, create a file named Customer.java that has the following content:

```
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();
    }
  }
}
```

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, which is described in the following step.

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

The class constructor establishes a connection to the database.

```
public Customer() throws SQLException
{
  // Connect
  DatabaseManager dbMgr = new DatabaseManager();
  String parms = "uid=DBA"
    + ";pwd=SQL"
    + ";file_name=c:\\tutorial\\tutcustomer.udb"
    + ";schema_file=c:\\tutorial\\tutcustomer.usm";
  try {
    conn = dbMgr.openConnection( parms );
    System.out.println(
            "Connected to an existing database." );
  } catch( SQLException econn ) {
    if( econn.getErrorCode() ==
        SQLCode.SQLE_DATABASE_NOT_FOUND ){
      conn = dbMgr.createDatabase( parms );
      System.out.println(
            "Connected to a new database." );
    } else {
      throw econn;
    }
  }
}
```

This code carries out the following tasks:

- Instantiates a new DatabaseManager object. All UltraLite objects are created from the DatabaseManager object.
- Defines connection parameters to connect to the database. Here, the parameters are a user ID and password (which have the default values for UltraLite databases), the location of the database file, and the location of a schema file to use if the database does not exist.

  Here, the locations are hardwired for convenience. In a real application, they would not be. In addition, these connection 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.

3 Compile the Customer class.

It is recommended that you use Sun JDK 1.1.8 to compile the class. In addition, you must add the UltraLite library *jul8.jar* to your classpath. This library is in the *ultralite\NativeUltraLiteForJava* subdirectory of your SQL Anywhere or UltraLite Component Suite installation.

The following command compiles the application. It should be entered on a single line at a command prompt:

```
javac -classpath
"%ASANY8%\ultralite\NativeUltraLiteForJava\jul8.jar"
Customer.java
```

4 Run the application.

The classpath must include the UltraLite library jul8.jar, as in the previous step.

The application must also be able to load the DLL that holds UltraLite native methods. The DLL is *jul8.dll* in the *ultralite\NativeUltraLiteForJava\win32* subdirectory of your SQL Anywhere or UltraLite Component Suite installation. This DLL can be in your system path or you can specify it on the java command line, as follows:

```
java -classpath
".;%ASANY8%\ultralite\NativeUltraLiteForJava\jul8.jar"
-Djul.library.dir=
"%ASANY8%\ultralite\NativeUltraLiteForJava\win32"
Customer
```

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

The first time you run the application, it should write the following text to the command line:

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

Subsequent times, it writes the following text to the command line:

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

# Lesson 3: Insert data into the database

This lesson shows you how to add data to the database.

❖ **To add rows to your database:**

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

```
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();
}
```

This code 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 (for better performance, or for multi-operation transactions) 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.

2 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 2: Connect to the database" on page 9.

The first time you run the application, it prints the following messages:

```
Connected to an existing database.
Two rows inserted.
```

Subsequent times, it prints the following messages:

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

# Lesson 4: Select the rows from the table

This lesson retrieves rows from the table, and prints them on the command line. It shows how to loop over the rows of a table.

❖ **To list the rows in the table:**

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

```
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();
}
```

This code carries out the following tasks:

- Opens the Table object, as in the previous lesson.
- Retrieves the column identifiers, as in the previous lesson.
- 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 (alphabetically by name, for example), you can add an index to an UltraLite 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.

2 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 2: Connect to the database" on page 9.

The application prints the following message:

```
Connected to an existing database.
The table has 2 rows.
id= 1, name= Gene Poole
id= 2, name= Penny Stamp
```

# Lesson 5: Deploy your application to a Windows CE device

Running the application on Windows CE requires that the Jeode Runtime Java VM be installed on your device. The Jeode Runtime is available for ARM-based devices such as the Compaq iPaq, and can be found on the CD that comes with the device.

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

- Choose Start ➤Programs.

  A Jeode folder is listed 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.

   In the Customer constructor, alter the connection parameters to read as follows:

   ```
   String parms = "uid=DBA"
   + ";pwd=SQL"
   + ";file_name=c:\\tutorial\\tutcustomer.udb"
   + ";schema_file=c:\\tutorial\\tutcustomer.usm"
   + ";ce_file=\\UltraLite\\tutorial\\tutcustomer.udb"
   + ";ce_schema=\\UltraLite\\tutorial\\tutcustomer.usm";
   ```

   The new parameters are `ce_file` and `ce_schema`. They indicate where on the device the schema and database are to be deployed.

2. Compile your application again.

3. Run your application to check that no errors were introduced. The ce_file and ce_schema 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\jul8.jar
Customer
```

The content is displayed on multiple lines for legibility. 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 (*jul8.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`.

You are now ready to copy 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. 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 *tutorial*. *\UltraLite\lib* is the location for the UltraLite runtime files, and *\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\jul8.jar | \UltraLite\lib\jul8.jar |
| UltraLite\NativeUltraLiteForJava\ce\arm\jul8.dll | \UltraLite\lib\jul8.dll |

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, do not copy the database file to the Windows CE device.

5 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 |

You are now ready to run the application on your Windows CE device.

❖ **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.
```

- Close the console.

You have now written an application, tested it on a desktop computer, and deployed it to a Windows CE device.

# Lesson 6: Add synchronization to your application

This lesson uses MobiLink synchronization, which is part of SQL Anywhere Studio. You must have the SQL Anywhere Studio installed to carry out this lesson.

The steps involved in this lesson are to add synchronization code to your application, to start the MobiLink synchronization server, and to run your application to synchronize.

The synchronization is carried out with the UltraLite 8.0 Sample database. This is an Adaptive Server Anywhere database that holds several tables. The *ULCustomer* table has a *cust_id* column and a *cust_name* column. During synchronization the data in that table is downloaded to your UltraLite application, and the two rows in your application's ULCustomer table are uploaded to the *UltraLite 8.0 Sample* database.

This lesson assumes some familiarity with MobiLink synchronization.

❖ **To add synchronization to your application:**

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

```
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();
}
```

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 API Reference.

  The syncParms field of the Connection object provides convenient access to a SyncParms object. For more information, see **ianywhere.native_ultralite.SyncParms** in the API Reference.

- 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 generates those scripts automatically.

  ☞ For more information, see the *MobiLink Synchronization User's Guide*.

- Sets the MobiLink user ID. This 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 them 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.

2 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.synchronize();
```

3 Compile your application, as in "Lesson 2: Connect to the database" on page 9. Do not run the application yet.

### ❖ To synchronize your data:

1 Start the MobiLink synchronization server.

From a command prompt, start the MobiLink synchronization server with the following command line:

```
dbmlsrv8 -c "dsn=ASA 8.0 Sample" -v+ -zu+ -za
```

The ASA 8.0 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 8.0 Sample database.

The `-zu+` and `-za` command line options provide automatic addition of users and generation of synchronization scripts. For more information on these options, see the *MobiLink Synchronization User's Guide*.

2 Run your application, as in "Lesson 2: Connect to the database" on page 9.

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

The data downloaded from the sample database are listed at the command prompt window, confirming that the synchronization succeeded:

```
Connected to an existing database.
The table has 128 rows.
id= 1, name= Gene Poole
id= 2, name= Penny Stamp
id= 101, name= Michaels Devlin
id= 102, name= Beth Reiser
id= 103, name= Erin Niedringhaus
id= 104, name= Meghan Mason
...
```

This completes the tutorial.

## Samples

For more code samples, see *Samples\Native UltraLite for Java\simple\Simple.java*

CHAPTER 3

# Tutorial: The CustDB Sample Application

About this chapter

This chapter walks you through all the steps of building and deploying a multi-table application that demonstrates UltraLite's ability to synchronize with a consolidated database. Some steps of this tutorial require SQL Anywhere Studio.

Contents

# Introduction

The previous tutorial, "Tutorial: A Native UltraLite for Java Application" on page 5, describes a very simple application.

This tutorial walks you through compiling, running and deploying CustDB, which is a more complex Native UltraLite for Java application.

Source code for CustDB is supplied in the *Samples\NativeUltraLiteForJava\CustDB* subdirectory of your SQL Anywhere installation. The source code contains examples of how to implement several tasks using Native UltraLite for Java.

## Timing

The tutorial takes about 30 minutes.

## Competencies and experience

This tutorial assumes that you have completed the tutorial "Tutorial: A Native UltraLite for Java Application" on page 5.

The tutorial uses synchronization, and so requires SQL Anywhere.

For the final lesson in the tutorial you must have access to a Windows CE device with the Jeode VM running on it.

# Lesson 1: Build the CustDB application

The sample comes with a *build.bat* script to build the CustDB sample from the source java files. A *clean.bat* script is provide for removing all results of the build.

❖ **To build the sample:**

1. Locate the sample.

   At a command prompt, change directory to the *Samples\NativeUltraLiteForJava\CustDB* subdirectory of your SQL Anywhere installation. You should leave this command prompt open for the entire tutorial.

2. Set environment variables.

   Ensure the environment variables ASANY8 and JAVA_HOME are properly defined.

   - ASANY8 is set by the Setup program to your SQL Anywhere installation directory.
   - Set the JAVA_HOME environment variable to a JDK on your machine. JDK 1.1.8 is recommended, as it is compatible with the Jeode VM used for deployment.

     For example, if the JDK is version 1.3 in *c:\jdk1.1.8* type the following command:

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

3. Build the application:

   - At the command prompt type `build.bat`.

   The batch file carries out the following operations:

   - Compiles the CustDB sample code.

     The compiled classes are stored under the *builddir* subdirectory.

   - Puts the compiled classes into a JAR file.

     The JAR file is stored in the sample directory, where the *build.bat* file is located.

   - Creats a database schema file.

     The schema file is created from the Adaptive Server Anywhere **UltraLite 8.0 Sample** database, which holds the CustDB database. The batch file runs the ulinit command-line tool to generate a schema from that database.

For more information on the ulinit tool, see "UltraLite initialization utility" on page 17 of the book *UltraLite Foundations*.

## 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 here. |
| *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. |

You can review these *.java* files for specific details and investigate within these files to see how the application works.

# Lesson 2: Run the CustDB application

The following procedure shows you how to run the *CustDB* sample on Windows. Deployment to a Windows CE device is described in a later lesson. This lesson uses MobiLink synchronization, and so requires that you have SQL Anywhere installed.

❖ **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 8.0 Sample** database, which is used in this stage as a consolidated database for the CustDB applcation.

   - 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 ID **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, aprove 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

Deployment of the CustDB sample onto a CE/ARM device 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 carried out 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 described above.

3 Copy the UltraLite runtime files to the Windows CE device:

If you have carried out 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 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\jul8.jar* | \UltraLite\lib |
| *UltraLite\NativeUltraLiteForJava\ce\arm\jul8.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\Cust |

In this tutorial, do not copy the database file to the Windows CE device.

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, open the *SQL Anywhere Studio Online Books* and lookup the index entry **ActiveSync: installing the MobiLink provider**.

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\jul8.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**
The setup steps for ActiveSync synchronization are given in the SQL Anywhere Online books under the index entry **ActiveSync: deploying UltraLite applications**.

❖ **To run the application:**

1. Start the MobiLink synchronization server.

   If you have SQL Anywhere Studio, use the batch file *Samples\NativeUltraLiteForJava\CustDB\ce\startdb.bat*.

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

You should now explore the application.

# Summary

During this tutorial, you:

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

CHAPTER 4

# Understanding UltraLite Development

About this chapter

This chapter describes how to develop applications with the Native UltraLite for Java.

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.

❖ **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();
```

For more details, see the code in *Samples\NativeUltraLiteForJava\Simple\Simple.java* under your SQL Anywhere directory, and **ianywhere.native_ultralite.DatabaseManager** in the API Reference.

2 Declare a Connection object.

Most applications use a single connection to an UltraLite database, and keep the connection open all the time. For this reason, it is often best to declare the Connection object global to the application.

```
static 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.

```
String parms = "file_name=mydata.udb";
try {
  conn = dbMgr.openConnection( parms );
  // more actions here
```

- The **DatabaseManager.openConnection** method establishes a connection to an existing UltraLite database file. It returns an open connection as a Connection object. This method takes a single string as its argument. The string is composed of a set of the following keyword-value pairs.

| Keyword | Alternative | Description |
|---|---|---|
| userid | uid | An authenticated user for the database. Databases are created with a single authenticated user named **DBA**. |
| password | pwd | The password for the user. When a database is created, the password for the DBA user ID is set to **SQL**. |
| con | con | A name for the connection. This is needed only if you create more than one connection to the database. |
| ce_file | ce_file | The path and filename of the UltraLite database on Windows CE. The default extension for UltraLite database files is *.udb*. |
| file_name | dbf | The path and filename of the UltraLite database on Windows. The default extension for UltraLite database files is *.udb*. |
| ce_schema | ce_schema | The path and filename of the UltraLite schema on Windows CE. The default extension for UltraLite schema files is *.usm*. |
| schema_file | schema_file | The path and filename of the UltraLite database on Windows. |

- If no database file exists, a SQLException is thrown. It is common to deploy a schema file rather than a database file, and to let UltraLite create the database file. Opening a connection to an existing database file uses a different method to creating a new database file and connecting to it.

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

```
catch( SQLException en ) {
  if( econn.getErrorCode() ==
      SQLCode.SQLE_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*.

```
String parms = "file_name=mydata.udb;"
             + "schema_file=mydata.usm";
conn = dbMgr.createDatabase( parms );
```

Example

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

```
String parms = "file_name=mydata.udb";
try {
  conn = dbMgr.openConnection( parms );
  System.out.println(
    "Connected to an existing database." );
}
catch( SQLException econn ) {
  if( econn.getErrorCode() ==
      SQLCode.SQLE_DATABASE_NOT_FOUND ){
    conn = dbMgr.createDatabase( parms );
    System.out.println(
    "Connected to a new database." );
  } else {
    throw econn;
  }
}
```

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

For more details, see the code in *Samples\NativeUltraLiteForJava\Simple\Simple.java* under your SQL Anywhere directory, and **ianywhere.native_ultralite.DatabaseManager** in the API Reference.

Using the Connection object

Properties 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 42.

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

  For more information, see "User authentication" on page 46

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

For more information, see **ianywhere.native_ultralite.SyncParms** in the API Reference.

- **Tables** UltraLite tables are accessed using the Connection.getTable method.

For more information, see **ianywhere.native_ultralite.Connection** in the API Reference.

### 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

UltraLite applications 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.

## Scrolling through the rows of a table

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

```
Table t = conn.getTable( "ULCustomer" );
short lname = t.schema.getColumnID( "lname" );
t.open();
t.moveBeforeFirst();
while ( t.moveNext() ){
   System.out.println( t.getString( lname ) );
}
```

The **schema** field is a convenience field to access a TableSchema object.

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 *customer* table as ordered by the *ix_name* index.

```
Table t= conn.getTable("customer");
t.open( "ix_name" );
t.moveFirst();
```

For more information, see **ianywhere.native_ultralite.Table, ianywhere.native_ultralite.TableSchema,** and **ianywhere.native_ultralite.Connection** in the 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" );
String lastname = t.getString( lname );
```

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

```
short cust_id = t.schema.getColumnID( "cust_id" );
int id = t.getInt( cust_id );
```

There are methods for each supported data type.

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 Java data type you wish to assign. UltraLite automatically casts 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.

For more information, see **ianywhere.native_ultralite.Table** in the API Reference.

## Searching for rows with find and lookup

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

- **find methods** These move to the first row that exactly matches a specified search value, under the sort order specified when the Table object was opened.
- **lookup methods** These 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 the findBegin() or lookupBegin() method, respectively. 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:

```
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:

```
t.findFirst();
```

For multi-column indexes, a value for the first column is always used, but you can omit the other columns and use one of the other find or lookup methods to search using only a limited number of columns.

For more information, see the **ianywhere.native_ultralite.Table** class in the API Reference.

## 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 ULTable object was opened, the current row is undefined.

By default, UltraLite 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 42.

> ***Caution***
> *Do 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 rows in the table has no significance.

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

```
t.insertBegin();
t.setInt( id, 3 );
t.setString(name, "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 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.native_ultralite.Connection** class in the API Reference.

## 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 a Table.moveNext method or other method 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.

```
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 Table.insert() method is called.
- **Update mode** The data in the buffer replaces the current row when the Table.update() method is called.
- **Find mode** The data in the buffer is used to locate rows when one of the Table.find*() methods is called.
- **Lookup mode** The data in the buffer is used to locate rows when one of the Table.lookup*() methods is called.

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

1. Enter the mode.

   The Table insertBegin, updateBegin, findBegin, and lookupBegin 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 such as insert, update, find, or lookup to carry out the operation, 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.

# 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 the Connection.schema property. See **ianywhere.native_ultralite.Connection** in the API Reference.

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

  To obtain a TableSchema object, access the Table.schema property. See **ianywhere.native_ultralite.Table** in the API Reference.

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

  To obtain a IndexSchema object, call the TableSchema.getIndex method. See **ianywhere.native_ultralite.IndexSchema** in the API Reference.

- **PublicationSchema** Tables and columns contained in a publication. Publications are also comprised of schema only, and so there is a PublicationSchema object rather than a Publication object.

  To obtain a PublicationSchema object, call the DatabaseSchema.getPublicationSchema method. See **ianywhere.native_ultralite.PublicationSchema** in the API Reference.

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

# 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 particular kind of error.

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

- **ianywhere.native_ultralite.SQLCode**
- **ianywhere.native_ultralite.StreamErrorCode**
- **ianywhere.native_ultralite.StreamErrorID**
- **ianywhere.native_ultralite.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 **ianywhere.native_ultralite.Connection** in the API Reference.

# 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. ActiveSync is available on WIndows CE devices through the Jeode Java VM.

Synchronization requires SQL Anywhere Studio. For general information on setting up ActiveSync synchronization, look up **ActiveSync: deploying UltraLite applictions** in the SQL Anywhere online books index. For general information on adding synchronization to an application, see "Synchronizing UltraLite applications" on page 10 of the book *UltraLite Foundations*.

ActiveSync synchronization can be initiated only by ActiveSync itself. ActiveSync automatically 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:

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

where MyAppClassName is a unique Windows class name for the application. For more information, see **ianywhere.native_ultralite.DatabaseManager.setActiveSyncListener** 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 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 allows synchronization through an application menu using a socket and through ActiveSync.

You can find source code for this sample in *Samples\NativUltraLiteForJava\CustDB\Application.java* 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 determies 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 class Application 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.

```
/** Define my own event class
*/
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 overiding 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.x. 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 Component Suite Setup program enables JBuilder integration. If JBuilder is installed after the UltraLite Component Suite, re-run the UltraLite Component Suite Setup program to enable JBuilder integration.

### Setting the JDK

You should use JDK 1.1.8 or PersonalJava 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 *jul8.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 connecto 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 UltraLite Component Suite online help, choose Help ➤Native UltraLite Reference. The API reference is available as a link on the front page of the documentation. The Native UltraLite for Java documentation is one of the books in the UltraLite Component Suite collection.

# Index

## E



## F



## G



## I



## J



## L



## M



## N

## O



## P



## R



## S



## T



## U

## V



## W