



UltraLite™ Static Java User's Guide

Part number: DC50033-01-0900-01

Last modified: June 2003

Copyright © 1989–2003 Sybase, Inc. Portions copyright © 2001–2003 iAnywhere Solutions, Inc. All rights reserved.

No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of iAnywhere Solutions, Inc. iAnywhere Solutions, Inc. is a subsidiary of Sybase, Inc.

Sybase, SYBASE (logo), AccelaTrade, ADA Workbench, Adaptable Windowing Environment, Adaptive Component Architecture, Adaptive Server, Adaptive Server Anywhere, Adaptive Server Enterprise, Adaptive Server Enterprise Monitor, Adaptive Server Enterprise Replication, Adaptive Server Everywhere, Adaptive Server IQ, Adaptive Warehouse, AnswerBase, Anywhere Studio, Application Manager, AppModeler, APT Workbench, APT-Build, APT-Edit, APT-Execute, APT-Library, APT-Translator, ASEP, AvantGo, AvantGo Application Alerts, AvantGo Mobile Delivery, AvantGo Mobile Document Viewer, AvantGo Mobile Inspection, AvantGo Mobile Marketing Channel, AvantGo Mobile Pharma, AvantGo Mobile Sales, AvantGo Pylon, AvantGo Pylon Application Server, AvantGo Pylon Conduit, AvantGo Pylon PIM Server, AvantGo Pylon Pro, Backup Server, BayCam, Bit-Wise, BizTracker, Certified PowerBuilder Developer, Certified SYBASE Professional, Certified SYBASE Professional (logo), ClearConnect, Client Services, Client-Library, CodeBank, Column Design, ComponentPack, Connection Manager, Convoy/DM, Copernicus, CSP, Data Pipeline, Data Workbench, DataArchitect, Database Analyzer, DataExpress, DataServer, DataWindow, DB-Library, dbQueue, Developers Workbench, Direct Connect Anywhere, DirectConnect, Distribution Director, Dynamic Mobility Model, Dynamo, e-ADK, E-Anywhere, e-Biz Integrator, E-Whatever, EC Gateway, ECMAP, ECRT, eFulfillment Accelerator, Electronic Case Management, Embedded SQL, EMS, Enterprise Application Studio, Enterprise Client/Server, Enterprise Connect, Enterprise Data Studio, Enterprise Manager, Enterprise Portal (logo), Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, eProcurement Accelerator, eremote, Everything Works Better When Everything Works Together, EWA, Financial Fusion, Financial Fusion (and design), Financial Fusion Server, Formula One, Fusion Powered e-Finance, Fusion Powered Financial Destinations, Fusion Powered STP, Gateway Manager, GeoPoint, GlobalFIX, iAnywhere, iAnywhere Solutions, ImpactNow, Industry Warehouse Studio, InfoMaker, Information Anywhere, Information Everywhere, InformationConnect, InstaHelp, InternetBuilder, iremote, iScript, Jaguar CTS, jConnect for JDBC, KnowledgeBase, Logical Memory Manager, M-Business Channel, M-Business Network, M-Business Server, Mail Anywhere Studio, MainframeConnect, Maintenance Express, Manage Anywhere Studio, MAP, MDI Access Server, MDI Database Gateway, media.splash, Message Anywhere Server, MetaWorks, MethodSet, ML Query, MobicATS, My AvantGo, My AvantGo Media Channel, My AvantGo Mobile Marketing, MySupport, Net-Gateway, Net-Library, New Era of Networks, Next Generation Learning, Next Generation Learning Studio, O DEVICE, OASIS, OASIS (logo), ObjectConnect, ObjectCycle, OmniConnect, OmniSQL Access Module, OmniSQL Toolkit, Open Biz, Open Business Interchange, Open Client, Open Client/Server, Open Client/Server Interfaces, Open ClientConnect, Open Gateway, Open Server, Open ServerConnect, Open Solutions, Optima++, Partnerships that Work, PB-Gen, PC APT Execute, PC DB-Net, PC Net Library, PhysicalArchitect, Pocket PowerBuilder, PocketBuilder, Power Through Knowledge, Power++, power.stop, PowerAMC, PowerBuilder, PowerBuilder Foundation Class Library, PowerDesigner, PowerDimensions, PowerDynamo, Powering the New Economy, PowerJ, PowerScript, PowerSite, PowerSocket, Powersoft, Powersoft Portfolio, Powersoft Professional, PowerStage, PowerStudio, PowerTips, PowerWare Desktop, PowerWare Enterprise, ProcessAnalyst, QAnywhere, Rapport, Relational Beans, RepConnector, Replication Agent, Replication Driver, Replication Server, Replication Server Manager, Replication Toolkit, Report Workbench, Report-Execute, Resource Manager, RW-DisplayLib, RW-Library, S.W.I.F.T. Message Format Libraries, SAFE, SAFE/PRO, SDF, Secure SQL Server, Secure SQL Toolset, Security Guardian, SKILS, smart.partners, smart.parts, smart.script, SQL Advantage, SQL Anywhere, SQL Anywhere Studio, SQL Code Checker, SQL Debug, SQL Edit, SQL Edit/TPU, SQL Everywhere, SQL Modeler, SQL Remote, SQL Server, SQL Server Manager, SQL Server SNMP SubAgent, SQL Server/CFT, SQL Server/DBM, SQL SMART, SQL Station, SQL Toolset, SQLJ, Stage III Engineering, Startup.Com, STEP, SupportNow, Sybase Central, Sybase Client/Server Interfaces, Sybase Development Framework, Sybase Financial Server, Sybase Gateways, Sybase Learning Connection, Sybase MPP, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase Synergy Program, Sybase User Workbench, Sybase Virtual Server Architecture, SybaseWare, Syber Financial, SyberAssist, SybMD, SyBooks, System 10, System 11, System XI (logo), SystemTools, Tabular Data Stream, The Enterprise Client/Server Company, The Extensible Software Platform, The Future Is Wide Open, The Learning Connection, The Model For Client/Server Solutions, The Online Information Center, The Power of One, TradeForce, Transact-SQL, Translation Toolkit, Turning Imagination Into Reality, UltraLite, UltraLite.NET, UNIBOM, Unilib, Uninull, Unisep, Unistring, URK Runtime Kit for UniCode, Versacore, Viewer, VisualWriter, VQL, Warehouse Control Center, Warehouse Studio, Warehouse WORKS, WarehouseArchitect, Watcom, Watcom SQL, Watcom SQL Server, Web Deployment Kit, Web.PB, Web.SQL, WebSights, WebViewer, WorkGroup SQL Server, XA-Library, XA-Server, and XP Server are trademarks of Sybase, Inc. or its subsidiaries.

Certicom and SSL Plus are trademarks and Security Builder is a registered trademark of Certicom Corp. Copyright © 1997–2001 Certicom Corp. Portions are Copyright © 1997–1998, Consensus Development Corporation, a wholly owned subsidiary of Certicom Corp. All rights reserved. Contains an implementation of NR signatures, licensed under U.S. patent 5,600,725. Protected by U.S. patents 5,787,028; 4,745,568; 5,761,305. Patents pending.

All other trademarks are property of their respective owners.

Contents

About This Manual	v
SQL Anywhere Studio documentation	vi
Documentation conventions	ix
The CustDB sample database	xi
Finding out more and providing feedback	xii
1 Introduction to the Static Java API	1
System requirements and supported platforms	2
Developing static Java applications	3
Benefits and limitations of the static Java API	4
2 Tutorial: Build an Application Using Java	5
Introduction	6
Lesson 1: Add SQL statements to your reference database	8
Lesson 2: Run the UltraLite generator	10
Lesson 3: Write the application code	11
Lesson 4: Build and run the application	15
Lesson 5: Add synchronization to your application	16
Lesson 6: Undo the changes you have made	18
3 Data Access Using Pure Java	19
Introduction	20
The UltraLite Java sample application	21
Connecting to and configuring your UltraLite database	26
Including SQL statements in UltraLite Java applications	32
UltraLite Java development notes	33
Building UltraLite Java applications	34
4 Adding Non Data Access Features to UltraLite Applications	37
Adding user authentication to your application	38
Configuring and managing database storage	41
Adding synchronization to your application	45
Developing multi-threaded applications	56
5 UltraLite Static Java API Reference	57
UltraLite API reference	58
6 Synchronization Parameters Reference	69
Synchronization parameters	70

About This Manual

Subject	This manual describes the UltraLite static Java API. It is a complement for the <i>UltraLite Database User's Guide</i> .
Audience	This manual is intended for application developers writing Java programs that use the UltraLite database. Familiarity with relational databases and Adaptive Server Anywhere is assumed.

SQL Anywhere Studio documentation

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

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

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

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

Documentation formats SQL Anywhere Studio provides documentation in the following formats:

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

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

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

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

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

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

Documentation conventions

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

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

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

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

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

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

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

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

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

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

RELEASE SAVEPOINT [*savepoint-name*]

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

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

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

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

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

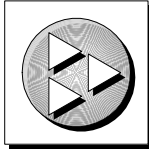
[**QUOTES** { **ON** | **OFF** }]

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

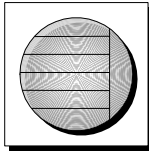
Graphic icons

The following icons are used in this documentation.

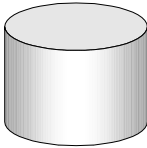
- ◆ A client application.



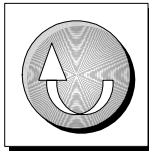
- ◆ A database server, such as Sybase Adaptive Server Anywhere.



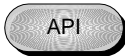
- ◆ A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



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



- ◆ A programming interface.



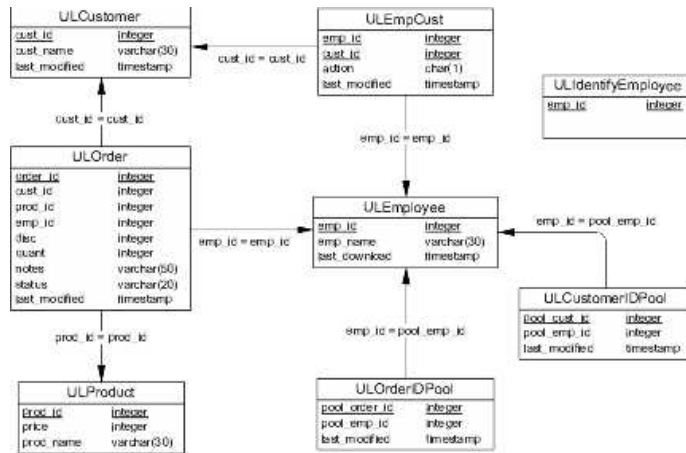
The CustDB sample database

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

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

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

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



Finding out more and providing feedback

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

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

The newsgroups include the following:

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

Newsgroup disclaimer

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

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

CHAPTER 1

Introduction to the Static Java API

About this chapter

This chapter introduces the static Java interface to UltraLite databases. It assumes that you are familiar with the UltraLite database system and the development models it offers.

☞ For more information, see “Welcome to UltraLite” [*UltraLite Database User’s Guide*, page 3].

Contents

Topic:	page
System requirements and supported platforms	2
Developing static Java applications	3
Benefits and limitations of the static Java API	4

System requirements and supported platforms

The supported target platform is a Sun JRE version 1.1.8 or later.

Application development requires a supported JDK. You must also have an Adaptive Server Anywhere reference database.

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

Developing static Java applications

When developing static Java UltraLite applications, you use a JDBC-like programming interface. In order to develop these applications you should be familiar with the Java programming language.

The development process for static Java UltraLite applications is as follows:

1. Design your database.

Prepare an Adaptive Server Anywhere reference database that contains the tables and indexes you wish to include in your UltraLite database.

2. Add SQL statements to the database.

The SQL Statements you wish to use in your application must be added to the reference database.

3. Generate the classes for your application.

The UltraLite generator provides the classes your application needs.

4. Write your application.

Data access features in your application code use JDBC and other function calls.

☞ For a guide to the interface, see [“UltraLite Static Java API Reference” on page 57](#).

5. Compile your .java files.

You can compile the generated .java files just as you compile other .java files.

☞ For a full description of the development process, see [“Building UltraLite Java applications” on page 34](#).

Benefits and limitations of the static Java API

UltraLite provides several programming interfaces, including both static development models (of which the static Java interface is one) and UltraLite components. A Java-based component (Native UltraLite for Java) is among those available.

The static Java API has the following advantages:

- ◆ **Pure Java solution** The UltraLite runtime library for the static Java API is a pure Java application. This is different from the Native UltraLite for Java component, which shares the same C++-based UltraLite runtime library as other UltraLite interfaces. In the Native UltraLite for Java component, access to the UltraLite runtime is provided by native methods.
- ◆ **Extensive SQL support** With the static Java API you can use a wider range of SQL in your applications than using the component-based interface.

The static Java API has the following disadvantages:

- ◆ **Complex development model** The use of a reference database to hold the UltraLite database schema, together with the need to generate classes for your specific application, makes the static Java API development process complex. The UltraLite components, including Native UltraLite for Java, provide a much simpler development process.
- ◆ **SQL must be specified at design time** Only SQL statements defined at compile time can be included in your application. The UltraLite components allow dynamic use of SQL statements.

The choice of development model is guided by the needs of your particular project, and by the programming skills and experience available.

CHAPTER 2

Tutorial: Build an Application Using Java

About this chapter

This chapter provides a tutorial that guides you through the process of developing a Java UltraLite application. The first section describes how to build a very simple Java UltraLite application. The second section describes how to add synchronization to your application.

☞ For an overview of the development process and background information on the UltraLite database, see [“Adding Non Data Access Features to UltraLite Applications” on page 37](#).

☞ For information on developing Java UltraLite Applications, see [“Data Access Using Pure Java” on page 19](#).

Contents

Topic:	page
Introduction	6
Lesson 1: Add SQL statements to your reference database	8
Lesson 2: Run the UltraLite generator	10
Lesson 3: Write the application code	11
Lesson 4: Build and run the application	15
Lesson 5: Add synchronization to your application	16
Lesson 6: Undo the changes you have made	18

Introduction

This tutorial describes how to construct a very simple application using UltraLite Java. The application is a command-line application, developed using the Sun JDK, which queries data in the `ULProduct` table of the *UltraLite 9.0 Sample* database.

In this tutorial, you create a Java source file, create a project in a reference database, and use these sources to build and run your application. The early lessons describe a version of the application without synchronization. Synchronization is added in a later lesson.

To follow the tutorial, you should have a Java Development Kit installed.

Overview

In the first lesson, you write and build an application that carries out the following tasks.

1. Connects to an UltraLite database, consisting of a single table. The table is a subset of the `ULProduct` table of the UltraLite Sample database.
2. Inserts rows into the table. Initial data is usually added to an UltraLite application by synchronizing with a consolidated database. Synchronization is added later in the chapter.
3. Writes the rows of the table to standard output.

In order to build the application, you must carry out the following steps:

1. Create an Adaptive Server Anywhere reference database.
Here we use the UltraLite sample database (CustDB).
2. Add the SQL statements to be used in your application to the reference database.
3. Run the UltraLite generator to generate the Java code and also an additional source file for this UltraLite database.

The generator writes out a `.java` file holding the SQL statements, in a form you can use in your application, and a `.java` file holding the code that executes the queries.

4. Write source code that implements the logic of the application.
Here, the source code is a single file, named *Sample.java*.
5. Compile and run the application.

In the second lesson you add synchronization to your application.

Create a directory to hold your files

In this tutorial, you will be creating a set of files, including source files and executable files. You should make a directory to hold these files. In addition, you should make a copy of the UltraLite sample database so that you can work on it, and be sure you still have the original sample database for other projects.

Copies of the files used in this tutorial can be found in the *Samples\UltraLite\JavaTutorial* subdirectory of your SQL Anywhere directory.

❖ To prepare a tutorial directory

1. Create a directory to hold the files you will create. In the remainder of the tutorial, we assume that this directory is *c:\JavaTutorial*.
2. Make a backup copy of the UltraLite 9.0 Sample database into the tutorial directory. The UltraLite 9.0 Sample database is the file *custdb.db*, in the *UltraLite\Samples\CustDB* subdirectory of your SQL Anywhere installation directory. In this tutorial, we use the original UltraLite 9.0 Sample database, and at the end of the tutorial you can copy the untouched version from the *APITutorial* directory back into place.

Lesson 1: Add SQL statements to your reference database

The reference database for this tutorial is the UltraLite 9.0 Sample database. In a later step, you use this same directory as a consolidated database for synchronization. These two uses are separate, and in your work you may use different databases for the two roles.

Add the SQL statements to the reference database using the `ul_add_statement` stored procedure. In this simple application, use the following statements:

- ◆ **Insert** An INSERT statement adds an initial copy of the data into the ULProduct table. This statement is not needed when synchronization is added to the application.
- ◆ **Select** A SELECT statement queries the ULProduct table.

When you add a SQL statement, you must associate it with an UltraLite project. Here, we use a project name of **Product**. You must also add a name for the statement, which by convention is in upper case.

❖ To add the SQL statements to the reference database

1. Start Sybase Central, and connect to the UltraLite 9.0 Sample data source using the Adaptive Server Anywhere plug-in.
2. Add a project to the database:
 - ◆ In Sybase Central, open the `custdb` database.
 - ◆ Open the UltraLite projects folder.
The folder contains one project already: the `custapi` project used for the sample application. You must create a new project.
 - ◆ Double-click **Add UltraLite Project**.
 - ◆ Enter **Product** as the project name, and click **Finish**.
3. Add the INSERT statement to the Product project.
 - ◆ Double-click **Product** to open the project.
 - ◆ Double-click **Add UltraLite Statement**.
 - ◆ Enter **InsertProduct** as the statement name. Click **Next**.
 - ◆ Enter the statement text:

```
INSERT INTO ULProduct ( prod_id, price, prod_name)
VALUES (?, ?, ?)
```

The first argument is the project name, the second is the statement name, and the third is the SQL statement itself. The question marks in the SQL statement are placeholders, and you can supply values at runtime.

- ◆ Click Finish to complete the operation.

This operation in Sybase Central is equivalent to executing the following stored procedure call:

```
call ul_add_statement( 'Product', 'InsertProduct',  
    'INSERT INTO ULProduct( prod_id, price, prod_name)  
    VALUES (?, ?, ?) ' )
```

4. Add the SELECT statement to the Product project.

- ◆ From the Product project, double-click Add UltraLite Statement.
- ◆ Enter **SelectProduct** as the statement name. Click Next.
- ◆ Enter the statement text:

```
SELECT prod_id, prod_name, price FROM ULProduct
```

- ◆ Click Finish to complete the operation.

This operation in Sybase Central is equivalent to executing the following stored procedure call:

```
call ul_add_statement( 'Product', 'SelectProduct',  
    'SELECT prod_id, prod_name, price FROM ULProduct')
```

5. Close Sybase Central.

You have now added the SQL statements to the database, and you are ready to generate the UltraLite database.

☞ For more information, see “ul_add_project system procedure” [*UltraLite Database User's Guide*, page 212], and “ul_add_statement system procedure” [*UltraLite Database User's Guide*, page 212].

Lesson 2: Run the UltraLite generator

The UltraLite generator writes out two Java files. One contains the SQL statements, as an interface definition, which is here named *ISampleSQL.java*. You can use this interface definition in your main application code. The second file holds the code that implements the queries and the database, and is here named *SampleDB.java*.

❖ To generate the UltraLite database code

1. Open a command prompt, and go to your *JavaTutorial* directory.
2. Run the UltraLite generator with the following arguments (all on one line):

```
ulgen -a -t java -c "dsn=UltraLite 9.0 Sample"  
-j Product -s ISampleSQL -f SampleDB
```

The arguments have the following meanings:

- ◆ **-a** Generate SQL string names in upper case. The **InsertProduct** and **SelectProduct** statements come to **INSERT_PRODUCT** and **SELECT_PRODUCT**.
- ◆ **-t** The language of the generated code. Generate Java code instead of C code.
- ◆ **-c** The connection string to connect to the database.
- ◆ **-j** The UltraLite project name. This name corresponds to the project name you provided when you added the SQL statement to the database. The generator produces code only for those statements associated with this project.
- ◆ **-s** The name of the interface that contains the SQL statements as strings.
- ◆ **-f** The name of the file that holds the generated database code and query execution code.

Lesson 3: Write the application code

The following code listing holds a very simple UltraLite application.

You can copy the code into a new file and save it as *Sample.java* in your *c:\JavaTutorial* directory, or open a new file and type the content. You can find this source code in *Samples\UltraLite\JavaTutorial\Sample.java*.

```

// (1) Import required packages
import java.sql.*;
import ISampleSQL.*;
import anywhere.ultralite.jdbc.*;
import anywhere.ultralite.support.*;
// (2) Class implements the interface containing SQL statements
public class Sample implements ISampleSQL
{
    public static void main( String[] args )
    {
        try{
            // (3) Connect to the database
            java.util.Properties p = new
                java.util.Properties();
            p.put( "persist", "file" );
            SampleDB db = new SampleDB( p );
            Connection conn = db.connect();
            // (4) Initialize the database with data
            PreparedStatement pstmt1 =
                conn.prepareStatement( INSERT_PRODUCT );
            pstmt1.setInt(1, 1);
            pstmt1.setInt(2, 400);
            pstmt1.setString(3, "4x8 Drywall x100");
            int rows1=pstmt1.executeUpdate();
            pstmt1.setInt(1, 2);
            pstmt1.setInt(2, 3000);
            pstmt1.setString(3, "8' 2x4 Studs x1000");
            int rows2=pstmt1.executeUpdate();
            // (5) Query the data and write out the results
            Statement stmt = conn.createStatement();
            ResultSet result = stmt.executeQuery(
                SELECT_PRODUCT );
            while( result.next() ) {
                int id = result.getInt( 1 );
                String name = result.getString( 2 );
                int price = result.getInt( 3 );
                System.out.println( name +
                    "\tId=" + id +
                    "\tPrice=" + price );
            }
            // (6) Close the connection to end
            conn.close();
        } catch (SQLException e) {
            Support.printStackTrace( e );
        }
    }
}

```

Explanation of the
sample program

Although too simple to be useful, this example contains elements that must be present in all Java programs used for database access. The following describes the key elements in the sample program. Use these steps as a guide when creating your own Java UltraLite application.

The numbered steps correspond to the numbered comments in the source code.

1. Import required packages.

The sample program utilizes JDBC interfaces and classes and therefore must import this package. It also requires the UltraLite runtime classes, and the generated interface that contains the SQL statement strings.

2. Define the class.

The SQL statements used in the application are stored in a separate file, as an interface. The class must declare that it implements the interface to be able to use the SQL statements for the project. The class names are based on the statement names you provided when adding the statements to the database.

3. Connect to the database.

The connection is established using an instance of the database class. The database name must match the name of the generated Java class (in this case **SampleDB**). The `file` value of the `persist` Properties object states that the database should be persistent.

4. Insert sample data.

In a production application, you would generally not insert sample data. Instead, you would obtain an initial copy of data by synchronization. In the early stages of development, it can simplify your work to directly insert data.

- ◆ Create a **PreparedStatement** object using the **prepareStatement()** method.
- ◆ To execute SQL commands, you must create a **Statement** or **PreparedStatement** object. Use a **Statement** object to execute simple SQL commands without any parameters and a **PreparedStatement** object to execute SQL commands with parameters. The sample program first creates a **PreparedStatement** object to execute an insert command:

```
PreparedStatement pstmt1 =  
conn.prepareStatement( INSERT_PRODUCT );
```

The **prepareStatement** method takes a SQL string as an argument; this SQL string is included from the generated interface.

5. Execute a select SQL command using a Statement object

- ◆ Create a **Statement** object using the **createStatement()** method. Unlike the **PreparedStatement** object, you do not need to supply a SQL statement when you create a **Statement** object. Therefore, a

single **Statement** object can be used to execute more than one SQL statement.

```
Statement stmt = conn.createStatement();
```

- ◆ Execute your SQL query.

Use the **executeQuery()** method to execute a select query. A select statement returns a **ResultSet** object.

- ◆ Implement a loop to sequentially obtain query results.

The **ResultSet** object maintains a cursor that initially points just before the first row. The cursor is incremented by one row each time the **next()** method is called. The **next()** method returns a true value when the cursor moves to a row with data and returns a false value when it has moved beyond the last row.

```
while(result.next()) {  
    ...  
}
```

- ◆ Retrieve query results using the **getxxx()** methods.

Supply the column number as an argument to these methods. The sample program uses the **getInt()** method to retrieve the product ID and price from the first and second columns respectively, and the **getString()** method to retrieve the product name from the third.

```
int id = result.getInt( 1 );  
int price = result.getInt( 2 );  
String name = result.getString( 3 );
```

6. End your Java UltraLite program

- ◆ Close the connection to the database, using the **Connection.close()** method:

```
conn.close();
```

Lesson 4: Build and run the application

After you have created a source file *Sample.java* using the sample code in the previous section, you are ready to build your UltraLite application.

❖ To build your application

1. Start the Adaptive Server Anywhere personal database server.

By starting the database server, the UltraLite generator has access to your reference database. Start the database server from the Start menu:

Start ► Programs ► Sybase SQL Anywhere 9 ► UltraLite ► Personal Server Sample for UltraLite.

2. Compile your Java source files.

Include the following locations in your classpath:

- ◆ The current directory (use a dot in your classpath).
- ◆ The Java runtime classes. For JDK 1.2, include the *jre\lib\rt.jar* file in your classpath. For JDK 1.1, include the *classes.zip* file from your Java installation.
- ◆ The UltraLite runtime classes. These classes are in the following location

```
%ASANY8%\UltraLite\java\lib\ulrt.jar
```

where %ASANY9% represents your SQL Anywhere directory.

Use the *javac* function of the Java development kit as follows:

```
javac *.java
```

You are now ready to run your application.

❖ To run your application

1. Go to a command prompt in the *Javatutorial* directory.
2. Include the same classes in the classpath as in the earlier step.
3. Enter the following command to run the application

```
java Sample
```

The list of two items is written out to the screen, and the application terminates.

You have now built and run your first UltraLite Java application. The next step is to add synchronization to the application.

Lesson 5: Add synchronization to your application

Once you have tested that your program is functioning properly, you can remove the lines of code that manually insert data into the `ULProduct` table. Replace these statements with a call to the `JdbcConnection.synchronize()` function to synchronize the remote database with the consolidated database. This process will fill the tables with data and you can subsequently execute a select query.

Adding synchronization actually simplifies the code. Your initial version of *Sample.java* uses the following lines to insert data into your UltraLite database.

```
PreparedStatement pstmt1 = conn.prepareStatement( ADD_PRODUCT_1
    );
    pstmt1.setInt(1, 1);
    pstmt1.setInt(2, 400);
    pstmt1.setString(3, "4x8 Drywall x100");
    int rows1=pstmt1.executeUpdate();
    pstmt1.setInt(1, 2);
    pstmt1.setInt(2, 3000);
    pstmt1.setString(3, "8' 2x4 Studs x1000");
    int rows2=pstmt1.executeUpdate();
```

This code is included to provide an initial set of data for your application. In a production application, you would not insert an initial copy of your data from source code, but would carry out a synchronization.

❖ To add synchronization to your application

1. Replace the hard-coded inserts with a synchronization call.
 - ◆ Delete the instructions listed above, which insert code.
 - ◆ Add the following line in their place:

```
ULSyncOptions synch_opts = new ULSynchOptions();
synch_opts.setUserName( "50" );
synch_opts.setPassword( "pwd50" );
synch_opts.setScriptVersion( "custdb" );
synch_opts.setStream( new ULSocketStream() );
synch_opts.setStreamParms( "host=localhost" );
( (JdbcConnection)conn ).synchronize( synch_opts );
```

The `ULSocketStream` argument instructs the application to synchronize over TCP/IP, to a MobiLink synchronization server on the current machine (**localhost**), using a MobiLink user name of 50.

2. Compile and link your application.

Enter the following command, with a `CLASSPATH` that includes the current directory, the UltraLite runtime classes, and the Java runtime

classes:

```
javac *.java
```

3. Start the MobiLink synchronization server running against the sample database.

From a command prompt in your *JavaTutorial* directory, enter the following command:

```
start dbmlsrv9 -c "dsn=UltraLite 9.0 Sample"
```

4. Run your application.

From a command prompt in your *JavaTutorial* directory, enter the following command:

```
java Sample
```

The application connects, synchronizes to receive data, and writes out information to the command line. The output is as follows:

```
Connecting to server:port = localhost(a.b.c.d):2439
4x8 Drywall x100      Id=1      Price=400
8' 2x4 Studs x1000    Id=2      Price=3000
Drywall Screws 101b   Id=3      Price=40
Joint Compound 1001b  Id=4      Price=75
Joint Tape x25x500    Id=5      Price=100
Putty Knife x25       Id=6      Price=400
8' 2x10 Supports x 200 Id=7      Price=3000
400 Grit Sandpaper    Id=8      Price=75
Screwmaster Drill     Id=9      Price=40
200 Grit Sandpaper    Id=10     Price=100
```

In this lesson, you have added synchronization to a simple UltraLite application.

☞ For more information on the **JdbcConnection.synchronize()** function, see [“synchronize method” on page 63](#).

Lesson 6: Undo the changes you have made

To complete the tutorial, you should shut down the MobiLink synchronization server and restore the UltraLite 9.0 Sample database.

❖ To finish the tutorial

1. Close down the MobiLink synchronization server.
2. Restore the UltraLite 9.0 Sample database.
 - ◆ Delete the *custdb.db* and *custdb.log* files in the *Samples\UltraLite\custdb* subdirectory of your SQL Anywhere directory.
 - ◆ Copy the *custdb.db* file from your *Javatutorial* directory to the *Samples\UltraLite\custdb* directory.
3. Delete the UltraLite database.
 - ◆ The UltraLite database is in the same directory as the jar file, and has a *.udb* extension. The application will initialize a new database next time the application is run.

CHAPTER 3

Data Access Using Pure Java

About this chapter

This chapter provides details of the UltraLite development process that are specific to Java. It explains how to write UltraLite applications using Java and provides instructions on building and deploying a Java UltraLite application.

Contents

Topic:	page
Introduction	20
The UltraLite Java sample application	21
Connecting to and configuring your UltraLite database	26
Including SQL statements in UltraLite Java applications	32
UltraLite Java development notes	33
Building UltraLite Java applications	34

Introduction

UltraLite applications can be written in the Java language using JDBC for database access.

The UltraLite development process for Java is similar to that for other development models. For a description, see “Using UltraLite Static Interfaces” [*UltraLite Database User’s Guide*, page 195].

This chapter describes only those aspects of application development that are specific to UltraLite Java applications. It assumes an elementary familiarity with Java and JDBC.

The UltraLite Java sample application

This section describes how to compile and run the UltraLite Java version of the CustDB sample application.

The sample application is provided in the *Samples\UltraLite\CustDB\java* subdirectory of your SQL Anywhere directory.

The applet version of the sample uses the Sun appletviewer to view the file *custdb.html*, which contains a simple `<APPLET>` tag.

The appletviewer security restrictions require the applet to be downloaded from a Web server, rather than to be run from the file system, for socket connections to be permitted and synchronization to succeed.

The application version of CustDB persists its data to a file, while the applet version does not use persistence.

☞ For a walkthrough of the C/C++ version of the application, which has very similar features, see [“Tutorial: A Sample UltraLite Application” on page ??](#).

The UltraLite Java sample files

The code for the UltraLite Java sample application is held in the *Samples\UltraLite\CustDB\java* subdirectory of your SQL Anywhere directory.

The directory holds the following files:

- ◆ **Data access code** The file *CustDB.java* holds the UltraLite-specific data access logic. The SQL statements are stored in *SQL.sql*.
- ◆ **User interface code** The files *DialogDelOrder.java*, *Dialogs.java*, *DialogNewOrder.java*, and *DialogUserID.java* all hold user interface features.
- ◆ **readme.txt** A text file containing detailed, release-dependent information about the sample.
- ◆ **Subdirectories** There are two subdirectories in which you can run the sample. These are *java11* (for Java 1) and *java13* (for Java 2). You should make the former your current directory if you are using a 1.1.x version of the JDK, and the latter if you are using 1.2.x or later. These subdirectories contain batch files to run the samples. In each directory, the batch files depend on the `JAVA_HOME` environment variable, which should be set to the directory containing the JDK. For example:

```
SET JAVA_HOME=c:\jdk1.3.1
```

-
- ◆ **Batch files to build the application** The files *build.bat* and *clean.bat* compile the application and delete all files except the source files, respectively.
 - ◆ **Files to run the sample as an application** The *Application.java* file contains instructions necessary for running the example as a Java application, and *run.bat* runs the sample application.
 - ◆ **Files to run the sample as an applet** The *Applet.java* file contains instructions necessary for running the example as a Java applet, and *avapplet.bat* runs the sample applet using the appletviewer, with *custdb.html* as the Web page.

You must install and start a Web server to run the sample as an applet. The applet can be run using the appletviewer utility or by using a Web browser. For more information, see the *Samples\UltraLite\CustDB\Java\readme.txt* file.

Building the UltraLite Java sample

This section describes how to build the UltraLite Java sample application for the Sun Java 1 or 2 environment.

❖ To build the UltraLite Java sample

1. Ensure you have the right JDK.

You must have JDK 1.1 or JDK 1.3 to build the sample application, and the JDK tools must be in your path.

2. Open a command prompt.
3. Change to the *Samples\UltraLite\CustDB\java\java13* subdirectory of your SQL Anywhere directory, or the *java11* directory if you are using Java 1.
4. Build the sample:

- ◆ Set the JAVA_HOME environment variable. For example:

```
SET JAVA_HOME=c:\jdk1.3.1
```

- ◆ From the command prompt, enter the following command:

```
build
```

The build procedure carries out the following operations:

- ◆ Loads the SQL statements into the UltraLite sample database.
This step uses Interactive SQL, the *SQL.sql* file, and relies on the UltraLite 9.0 Sample data source.

- ◆ Generates the Java database class **custdb.Database**.
This step uses the UltraLite generator and the UltraLite 9.0 Sample data source.
- ◆ Compiles the Java files.
This step uses the JDK compiler (*javac*) and *jar* utility.

Running the UltraLite Java sample

You can run the sample application as a Java application or as an applet. In either case, you need to prepare to run the sample by starting the MobiLink synchronization server running on the same machine that the application is running on.

❖ To prepare to run the sample

1. Start the MobiLink synchronization server running on the UltraLite sample database:

From the Start menu, choose Programs ► SQL Anywhere 9 ► MobiLink ► Synchronization Server Sample.

❖ To run the sample as an application

1. Open a command prompt in the *Samples\UltraLite\CustDB\java\java13* directory (or the *java11* directory if you are using Java 1).
2. Run the sample:
 - ◆ Set the JAVA_HOME environment variable. For example:

```
SET JAVA_HOME=c:\jdk1.3.1
```

- ◆ Enter the following command:

```
run
```

The application starts and the Enter ID dialog is displayed.

3. Enter the employee ID.

Enter an employee ID of 50, and click OK.

The UltraLite Customer Demonstration window is displayed. If you have run the sample as either an application or applet before, there is data in the database.

4. If there is no data in the database, synchronize.

From the Actions menu, choose Synchronize. The application synchronizes, and the window displays an order.

You can now carry out operations on the data in the database.

☞ For more information on the sample database and the UltraLite features it demonstrates, see [“Tutorial: A Sample UltraLite Application” on page ??](#).

❖ To run the sample as an applet using appletviewer

1. Start a Web server and ensure that the appropriate subdirectory is configured as the default directory for the server, or as one of the virtual directories.
2. Open a command prompt in the *UltraLite\samples\CustDB\java\java13* directory, or *java11* if you are using Java 1.
3. Enter the following command:

```
avapplet
```

- ◆ The applet starts and a field to enter an employee ID is displayed.

4. Enter the employee ID.

Enter an employee ID of 50, and click OK.

The UltraLite Customer Demonstration window is displayed. The first time you run the sample, there is no data in the database. If you have run the sample as either an application or applet before, there is data in the database.

5. Synchronize the application:

From the Actions menu, choose Synchronize. The application synchronizes, and the window displays an order.

You can now carry out operations on the data in the database.

❖ To run the sample as an applet using A Web browser

1. Start a Web server and ensure that the appropriate subdirectory is configured as the default directory for the server, or as one of the virtual directories.

2. Start a Web browser and enter the URL for the *Samples\UltraLite\CustDB\java\custdb.htm* file into the browser.

- ◆ The applet starts and a field to enter an employee ID is displayed.


3. Enter the employee ID.

Enter an employee ID of 50, and click OK.

The UltraLite Customer Demonstration window is displayed. The first time you run the sample, there is no data in the database. If you have run the sample as either an application or applet before, there is data in the database.

4. Synchronize the application:

From the Actions menu, choose Synchronize. The application synchronizes, and the window displays an order.

 For more information on the sample database and the UltraLite features it demonstrates, see [“Tutorial: A Sample UltraLite Application”](#) on page ??.

Resetting the sample

You can delete all compiled files, the sample database, and the generated code by running the *clean.bat* file.

Connecting to and configuring your UltraLite database

This section describes how to connect to an UltraLite database. It describes the recommended UltraLite method for connecting to your database, and also how you can use the standard JDBC connection model to connect.

Connections to UltraLite databases have no user IDs or passwords. For more information, see “User authentication” [*UltraLite Database User’s Guide*, page 38].

UltraLite Java databases can be **persistent** (stored in a file when the application closes) or **transient** (the database vanishes when the application is closed). By default, they are transient.

You configure the persistence of your UltraLite database when connecting to it. This section describes how to configure your UltraLite database.

Using the UltraLite JdbcDatabase.connect method

The generated UltraLite database code is in the form of a class that extends **JdbcDatabase**, which has a **connect** method that establishes a connection.

The following example illustrates typical code, for a generated database class called **SampleDB**:

```
try {
    SampleDB db = new SampleDB();
    java.sql.Connection conn = db.connect();
} catch( SQLException e ){
    // error processing here
}
```

The generated database class is supplied on the UltraLite generator command line, using the `-f` option.

If you wish to use a persistent database, the characteristics are specified on the connection as a **Properties** object. The following example illustrates typical code:

```
java.util.Properties p = new java.util.Properties();
p.put( "persist", "file" );
p.put( "persistfile", "c:\\dbdir\\database.udb" );
SampleDB db = new SampleDB( p );
java.sql.Connection conn = db.connect( );
```

The **Properties** are used on the database constructor. You cannot change the persistence model of the database between connections.

The two properties specify that the database is persistent, and is stored in the file `c:\dbdir\database.udb`.

☞ For more information on the properties you can specify in the URL, see [“UltraLite JDBC URLs” on page 28](#).

☞ For more information see [“Configuring the UltraLite Java database” on page 30](#), and [“The generated database class” on page 66](#).

Loading and registering the JDBC driver

The UltraLite **JdbcDatabase.connect()** method discussed in the previous section provides the simplest method of connecting to an UltraLite database. However, you can also establish a connection in the standard JDBC manner, and this section describes how to do so.

UltraLite applications connect to their database using a JDBC driver, which is included in the UltraLite runtime classes (*ulrt.jar*). You must load and register the JDBC driver in your application before connecting to the database. Use the **Class.forName()** method to load the driver. This method takes the driver package name as its argument:

```
Class.forName( "ianywhere.ultralite.jdbc.JdbcDriver" );
```

The JDBC driver automatically registers itself when it is loaded.

Loading multiple drivers	Although there is typically only one driver registered in each application, you can load multiple drivers in one application. Load each driver using the same methods as above. The DriverManager decides which driver to use when connecting to the database.
getDriver method	The DriverManager.getDriver(url) method returns the Driver for the specified URL.
Error handling	To handle the case where the driver cannot be found, catch ClassNotFoundException as follows:

```
try{
    Class.forName(
        "ianywhere.ultralite.jdbc.JdbcDriver");
} catch(ClassNotFoundException e){
    System.out.println( "Exception: " + e.getMessage() );
    e.printStackTrace();
}
```

Connecting to the database using JDBC

Once the driver is declared, you can connect to the database using the standard JDBC **DriverManager.getConnection** method.

getConnection
prototypes

The JDBC **DriverManager.getConnection** method has several prototypes. These take the following arguments:

```
DriverManager.getConnection( String url, Properties info )
DriverManager.getConnection( String url )
```

The UltraLite driver supports each of these prototypes. The arguments are discussed in the following sections.

Driver Manager

The **DriverManager** class maintains a list of the **Driver** classes that are currently loaded. It asks each driver in the list if it is capable of connecting to the URL. Once such a driver is found, the **DriverManager** attempts to use it to connect to the database.

Error handling

To handle the case where a connection cannot be made, catch the **SQLException** as follows:

```
try{
    Class.forName(
        "ianywhere.ultralite.jdbc.JdbcDriver");
    Connection conn = DriverManager.getConnection(
        "jdbc:ultralite:asademo" );
} catch(SQLException e){
    System.out.println( "Exception: " + e.getMessage() );
    e.printStackTrace();
}
```

UltraLite JDBC URLs

The URL is a required argument to the **DriverManager.getConnection** method used to connect to UltraLite databases.

☞ For an overview of connection methods, see [“Connecting to the database using JDBC” on page 27](#).

The syntax for UltraLite JDBC URLs is as follows:

```
jdbc:ultralite:[database:persist:persistfile][:option=value...]
```

The components are all case sensitive, and have the following meanings:

- ◆ **jdbc** Identifies the driver as a JDBC driver. This is mandatory.
- ◆ **ultralite** Identifies the driver as the UltraLite driver. This is mandatory.
- ◆ **database** The class name for the database. It is required and must be a fully-qualified name: if the database class is in a package, you must include the package name.

For example, the URL `jdbc:ultralite:MyProject` causes a class named `MyProject` to be loaded.

As Java classes share their name with the `.java` file in which they are defined, this component is the same as the output file parameter from the UltraLite generator.

☞ For more information, see “The UltraLite generator” [*UltraLite Database User’s Guide*, page 96].

- ◆ **persist** Specifies whether or not the database should be persistent. By default, it is transient.

☞ For more information, see “Configuring the UltraLite Java database” on page 30.

- ◆ **persistfile** For persistent databases, specifies the filename.

☞ For more information, see “Configuring the UltraLite Java database” on page 30. The UltraLite Java properties are very similar to those for C/C++ applications. Their names differ only in the absence of underscore characters., except that **persistfile** is analogous to **file_name**. See “UL_STORE_PARMS macro” [*UltraLite Database User’s Guide*, page 216].

- ◆ **options** The following options are provided:

- **uid** A user ID.
- **pwd** A password for the user ID.

Alternatively, you can connect using a Properties object. The following properties may be specified. Each have the same meaning as in the explicit URL syntax above:

- ◆ database
- ◆ persist
- ◆ persistfile
- ◆ user
- ◆ password

Using a Properties object to store connection information

You can use a **Properties** object to store connection information, and supply this object as an argument to **getConnection** along with the URL.

☞ For an overview of connection methods, see “Connecting to the database using JDBC” on page 27.

The following components of the URL, described in “UltraLite JDBC URLs” on page 28, can be supplied either as part of the URL or as a member of a **Properties** object.

- ◆ **persist**

- ◆ **persistfile**

- ◆ The `jdbc:ultralite` components must be supplied in the URL.

If you wish to encrypt your database, you can do so by supplying a **key** property. For more information, see [“Encrypting UltraLite databases” on page 41](#).

Connecting to multiple databases

UltraLite Java applications can connect to more than one database, unlike UltraLite C/C++ applications.

To connect to more than one database, simply create more than one connection object.

☞ For more information see [“Connecting to the database using JDBC” on page 27](#).

Configuring the UltraLite Java database

You can configure the following aspects of the UltraLite Java database:

- ◆ Whether the database is transient or persistent.
- ◆ If the database is persistent, you can supply a filename.
- ◆ If the database is transient, you can supply a URL for an initializing database.
- ◆ You can set an encryption key.

These aspects can be configured by supplying special values in the database URL, or by supplying a Properties object when creating the database. The encryption key cannot be set on the URL, but must be set in a Properties object.

☞ For more information, see [“Using the UltraLite JdbcDatabase.connect method” on page 26](#), and [“Using a Properties object to store connection information” on page 29](#).

Transient and persistent databases

By default, UltraLite Java databases are transient: they are initialized when the database object is instantiated, and vanish when the application closes down. The next time the application starts, it must reinitialize a database.

You can make UltraLite Java databases persistent by storing them in a file. You do this by specifying the **persist** and **persistfile** elements of the JDBC URL, or by supplying **persist** and **persistfile** Properties to the database **connect** method.

Initializing transient databases

The database for C/C++ UltraLite applications is initialized on the first synchronization call. For UltraLite Java applications that use a transient database, there is an alternative method of initializing the database. The URL for an UltraLite database that is used as the initial database is supplied in the **persistfile** component to the URL.

Configuring the database The database configuration components of the URL are as follows:

- ◆ **persist** This can take one of the following values:
 - **none** In this case, the database is transient. It is stored in memory while the application runs, but vanishes once the application terminates.
 - **file** In this case, the database is stored as a file. The default filename is *database.udb*, where *database* is the database class name.

The default setting is **none**.

- ◆ **persistfile** The meaning of this component depends on the setting for **persist**.

- If the **persist** component has a value of **none**, the **persistfile** component is a URL for an UltraLite database file that is used to initialize the database.

Both the schema and the data from the URL are used to initialize the application database, but there is no further connection between the two. Any changes made by your application apply only to the transient database, not to the initializing database.

The following JDBC URL is an example:

```
jdbc:ultralite:transient:none:http://www.address.com/transient.u  
db
```

You can prepare the initializing database with an application that uses the persistent form of the URL to create the database, synchronize, and exit.

- ◆ If the **persist** component has a value of **file**, the **persistfile** component is a filename for the persistent UltraLite database. The filename should include any extension (such as *.udb*) that you wish to use.

Including SQL statements in UltraLite Java applications

This section describes how to add SQL statements to your UltraLite application.

☞ For information on SQL features that can and cannot be used in UltraLite application, see “Overview of SQL support in UltraLite” [*UltraLite Database User’s Guide*, page 108].

☞ The SQL statements to be used in your application must be added to the reference database. The UltraLite generator writes out an interface that defines these SQL statements as **public static final strings**. You invoke the statements in your application by implementing the interface and referencing the SQL statement by its identifier, or by referencing it directly from the interface.

Defining the SQL statements for your application

The SQL statements to be included in the UltraLite application, and the structure of the UltraLite database itself, are defined by adding the SQL statements to the reference database for your application.

☞ For information on reference databases, see “Preparing a reference database” [*UltraLite Database User’s Guide*, page 200].

Defining projects

Each SQL statement stored in the reference database is associated with a **project**. A project is a name, defined in the reference database, which groups the SQL statements for one application. You can store SQL statements for multiple applications in a single reference database by defining multiple projects.

☞ For information on creating projects, see “Creating an UltraLite project” [*UltraLite Database User’s Guide*, page 204].

Adding statements to your project

The data access statements you are going to use in your UltraLite application must be added to your project.

☞ For information on adding SQL statements to your database, see “Adding SQL statements to an UltraLite project” [*UltraLite Database User’s Guide*, page 205].

UltraLite Java development notes

This section provides notes for development of UltraLite Java applications.

Creating UltraLite Java applets

If you create your JDBC program as an applet, your application can only synchronize with the machine from which the applet is loaded, which is usually the same as the HTML.

Including an applet in an HTML page

The following is a sample HTML page used to create an UltraLite applet:

```
<html>
  <head>
  </head>
  <body bgcolor="FFFF00">
    <applet code="CustDbApplet.class" width=440
      height=188 archive="custdb.zip,ulrt.jar" >
    </applet>
  </body>
</html>
```

The applet tag specifies the following:

- ◆ The class that the applet starts:

```
code="CustDbApplet.class"
```

- ◆ The size of the window in the web browser to display the applet to.

```
width=440 height=188
```

- ◆ The zip files that are necessary in order to run the applet.

```
archive="custdb.zip,ulrt.jar"
```

In this case, the *custdb.zip* file and the UltraLite runtime zip file are necessary in order to run the UltraLite CustDB sample application.

Building UltraLite Java applications

This section covers the following subjects:

- ◆ “Generating UltraLite Java classes” on page 34
- ◆ “Compiling UltraLite Java applications” on page 35

Generating UltraLite Java classes

When you have prepared a reference database, defined an UltraLite project for your application, and added SQL statements to define your data access features, all the information the generator needs is inside your reference database.

☞ For general information on the UltraLite generator, see “Generating the UltraLite data access code” [*UltraLite Database User’s Guide*, page 209]. For command-line options, see “The UltraLite generator” [*UltraLite Database User’s Guide*, page 96].

The generator output is a Java source file with a filename of your choice. Depending on the design of your database and the sophistication of the database functionality your application requires, this file can vary greatly in both size and content.

There are several ways to customize the UltraLite generator output, depending on the nature of your application.

Overview

You generate the classes by running the UltraLite generator against the reference database.

❖ To run the UltraLite generator

1. Enter the following command at a command-prompt:

```
ulgen -c "connection-string" options
```

where *options* depend on the specifics of your project.

Common command-line combinations

When you are generating Java code, there are several options you may want to specify:

- ◆ **-t java** Generate Java code. The generator is the same tool used for C/C++ development, so this option is required for all Java use.
- ◆ **-i** Some Java compilers do not support inner classes correctly, and so the default behavior of the generator is not to generate Java code that includes inner classes. If you wish to take advantage of a compiler that does support inner classes, use this option.

- ◆ **-p** It is common to include your generated classes in a package, which may include other classes from your application. You can use this switch to instruct the generator to include a package name for the classes in the generated files
- ◆ **-s** In addition to the code for executing the SQL statements, generate the SQL statements themselves as an interface. Without this option, the strings are written out as members of the database class itself.
- ◆ **-a** Make SQL string names upper case. If you choose the **-a** option, the identifier used in the generated file to represent each SQL statement is derived from the name you gave the statement when you added it to the database. It is a common convention in Java to use upper case letters to represent constants. As the SQL string names are constants in your Java code, you should use this option to generate string identifiers that conform to the common convention.
- ◆ The following command (which should be all on one line) generates code that represents the SQL statements in the **CustDemo** project, and the required database schema, with output in the file *uldemo.java*.

Example

```
ulgen -c "dsn=Ultralite 9.0 Sample;uid=DBA;pwd=SQL"
-a -t java -s IStatements CustDemo uldemo.java
```

Compiling UltraLite Java applications

❖ To compile the generated file

1. Set your classpath

When you compile your UltraLite Java application, the Java compiler must have access to the following classes:

- ◆ The Java runtime classes.
- ◆ The UltraLite runtime classes
- ◆ The target classes (usually in the current directory).

The following classpath gives access to these classes.

```
%JAVA_HOME%\jre\lib\rt.jar;%ASANY8%\ultralite\java\lib\
ulrt.jar;.
```

where **JAVA_HOME** represents your Java installation directory, and **ASANY8** represents your SQL Anywhere installation directory.

For JDK 1.1 development, *ulrt.jar* is in a *jdk11\lib* subdirectory of the *UltraLite\java* directory.

2. Compile the classes.

With the classpath set as in step one, use *javac* and enter the following command (on a single line):

```
javac file.java
```

The compiler creates the class files for *file.java*.

The compilation step produces a number of class files. You must include *all* the generated *.class* files in your deployment.

Deploying Java applications

Your UltraLite application consists of the following:

- ◆ Class files you created to implement your application.
- ◆ Generated class files.
- ◆ The Java core classes (*rt.jar*).
- ◆ UltraLite runtime JAR file (*ulrt.jar*).

Your UltraLite application can be deployed in whatever manner is appropriate. You may wish to package together these class files in a JAR file, for ease of deployment.

Your UltraLite application automatically initializes its own database the first time it is invoked. At first, your database will contain no data. You can add data explicitly using INSERT statements in your application, or you can import data from a consolidated database through synchronization. Explicit INSERT statements are especially useful when developing prototypes.

CHAPTER 4

Adding Non Data Access Features to UltraLite Applications

About this chapter

This chapter describes features in addition to data access you can add to UltraLite applications.

☞ For information about data access features, see [“Data Access Using Pure Java” on page 19](#).

Contents

Topic:	page
Adding user authentication to your application	38
Configuring and managing database storage	41
Adding synchronization to your application	45
Developing multi-threaded applications	56

Adding user authentication to your application

UltraLite provides an optional built-in user authentication scheme. You can take advantage of this scheme to authenticate users before allowing them to connect to the UltraLite database. By default, UltraLite databases have no user authentication mechanism.

The UltraLite user authentication scheme does not provide the permissions features implemented in multi-user database systems and in MobiLink.

☞ For a general description of UltraLite user authentication, see “User authentication” [*UltraLite Database User's Guide*, page 38].

☞ When you create an UltraLite database with user authentication enabled, one authenticated user is created, with user ID **DBA** and password **SQL**. UltraLite permits up to four different users to be defined at a time, with both user ID and password being less than 16 characters long. Each user has full access to the database once successfully authenticated.

☞ The case sensitivity of the UltraLite user ID and password is determined by the reference database. If the reference database is case insensitive (the default) then the UltraLite database is also case insensitive, including user authentication.

Enabling user authentication

Enabling user authentication requires the application to supply a valid UltraLite user ID and password when connecting to the UltraLite database. If you do not explicitly enable user authentication, UltraLite does not authenticate users.

❖ To enable user authentication (Java)

1. Call the **JdbcSupport.enableUserAuthentication** method before creating a new database object. For example:

```
JdbcSupport.enableUserAuthentication();
java.util.Properties p = new java.util.Properties();
p.put( "persist", "file" );
SampleDB db = new SampleDB( p );
```

☞ Once you have enabled user authentication, you must add user management code to your application. For more information, see [“Managing user IDs and passwords” on page 38](#).

Managing user IDs and passwords

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 attempt to 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, call the same function as adding a user ID. This function is **JdbcDatabase.grant**.

User authentication example

The following code fragment performs user management and authentication for an UltraLite Java application.

A complete sample can be found in the *Samples\UltraLite\javaauth* subdirectory of your SQL Anywhere directory. The code below is based on that in *Samples\UltraLite\javaauth\Sample.java*.

```
JdbcSupport.enableUserAuthentication();
// Create database environment
java.util.Properties p = new java.util.Properties();
p.put( "persist", "file" );
SampleDB db = new SampleDB( p );

// Get new user ID and password
try{
    conn = db.connect( "dba", "sql" );
    // Set user ID and password
    // a real application would prompt the user.
    uid = "50";
    pwd = "pwd50";

    db.grant( uid, pwd );
    db.revoke( "dba" );
    conn.close();
}
catch( SQLException e ){
    // dba connection failed - prompt for user ID and password
    uid = "50";
    pwd = "pwd50";
}

// Connect
conn = db.connect( uid, pwd );
```

The code carries out the following tasks:

1. Opening the database object.
2. Attempt to connect using the default user ID and password.

-
3. If the connection attempt is successful, add a new user.
 4. Delete the default user from the UltraLite database.
 5. Disconnect. An updated user ID and password is now added to the database.
 6. Connect using the updated user ID and password.

☞ For more information, see “GrantConnectTo method” [*UltraLite Static C++ User’s Guide*, page 81], and “RevokeConnectFrom method” on page ??.

Sharing MobiLink and UltraLite user IDs

Although UltraLite and MobiLink user authentication mechanisms are separate, you may wish to provide your end users with a single user ID and password that provides both MobiLink and UltraLite user authentication. To share user IDs and passwords, store them in variables and use the same variable in the UltraLite user authentication calls and the synchronization call.

You can design your application so that, if passwords are reset at a MobiLink consolidated site, your application prompts for the new password.

❖ To prompt for a new MobiLink or UltraLite password

1. Save the user ID and password in variables.
2. Synchronize.
3. If synchronization fails because the user was not authenticated, prompt the user for a new password.
4. Update the UltraLite user’s password using the appropriate function or method:
 - ◆ **JdbcDatabase.grant** method
5. Update the UISynchOptions object and synchronize again.

☞ For information on MobiLink user authentication, see “Authenticating MobiLink Users” [*MobiLink Synchronization User’s Guide*, page 103].

Configuring and managing database storage

You can configure the following aspects of UltraLite persistent storage:

- ◆ The amount of memory used as a cache by the UltraLite database engine.
- ◆ Database encryption.
- ◆ Preallocation of file-system space.
- ◆ The file name for the database.
- ◆ The database page size.

This configuration is controlled by the `UL_STORE_PARMS` macro, which is placed in the header of your application source code so that it is visible to all `db_init()` or `ULPalmLaunch` calls. The encryption key and page size can be used on any supported C/C++ platform, while the other keys cannot be used on the Palm Computing Platform.

☞ For more information, see “`UL_STORE_PARMS` macro” [*UltraLite Database User’s Guide*, page 216].

Encrypting UltraLite databases

By default, UltraLite databases are unencrypted on disk and in permanent memory. Text and binary columns are plainly readable within the database store when using a viewing tool such as a hex editor. Two options are provided for greater security:

- ◆ **Obfuscation** Obfuscating databases provides security against straightforward attempts to view data in the database directly using a viewing tool. It is not proof against skilled and determined attempts to gain access to the data. Obfuscation has little or no performance impact.

☞ For more information, see “[Obfuscating an UltraLite database](#)” on [page 42](#).

- ◆ **Strong encryption** UltraLite database files can be strongly encrypted using the AES 128-bit algorithm, which is the same algorithm used to encrypt Adaptive Server Anywhere databases. Use of strong encryption does provide security against skilled and determined attempts to gain access to the data, but has a significant performance impact.

Caution

If the encryption key for a strongly encrypted database is lost or forgotten, there is no way to access the database. Under these circumstances, technical support cannot gain access to the database for you. It must be discarded and you must create a new database.

☞ For more information, see “[Encrypting an UltraLite database](#)” on page 42, and “[Changing the encryption key for a database](#)” on page 43.

Obfuscating an UltraLite database

❖ To obfuscate an UltraLite database

1. Add the following line to your code before creating the database (that is, before connecting to the database for the first time):

```
UltraDatabase.setDefaultObfuscation( true );
```

Encrypting an UltraLite database

UltraLite databases are created on the first connection attempt. To encrypt an UltraLite database, you supply an encryption key before that connection attempt. On the first attempt, the supplied key is used to encrypt the database. On subsequent attempts, the supplied key is checked against the encryption key, and connection fails unless the key matches.

❖ To strongly encrypt an UltraLite database (Java)

1. Set a property named key before creating a database object for the first time.

Here is a code fragment that reads the encryption key from the command line.

```
InputStreamReader isr = new InputStreamReader( System.in );
BufferedReader br = new BufferedReader( isr );
String key = null ;
System.out.print( "Enter encryption key:" );
key = br.readLine() ;
System.out.println( "The key is: " + key );

// (3) Connect to the database
java.util.Properties p = new java.util.Properties();
p.setProperty( "persist", "file" );
p.setProperty( "key", key );
SampleDB db = new SampleDB( p );
```

Here, SampleDB is the database filename as supplied in the UltraLite generator -f command-line option.

☞ For more information, see “[The UltraLite generator](#)” [*UltraLite Database User’s Guide*, page 96], and “[Using a Properties object to store connection information](#)” on page 29.

2. Create the database object using the properties.

For example:

```
Connection conn = db.connect();
```

After the first connection attempt, subsequent attempts to access the database produce an `Incorrect or missing encryption key SQLException` if the wrong key is supplied.

You can find a sample Java application demonstrating encryption in the directory `\Samples\UltraLite\JavaSecurity`. The encryption code is held in `\Samples\UltraLite\JavaSecurity\Sample.java`.

Here is a code snippet from the sample:

```
// Obtain the encryption key
InputStreamReader isr = new InputStreamReader( System.in );
BufferedReader br = new BufferedReader( isr );
String key = null ;
System.out.print( "Enter encryption key:" );
key = br.readLine() ;
System.out.println( "The key is: " + key );

java.util.Properties p = new java.util.Properties();
p.setProperty( "persist", "file" );
p.setProperty( "key", key );
SampleDB db = new SampleDB( p );
Connection conn = db.connect();
```

Changing the encryption key for a database

You can change the encryption key for a database. The application must already be connected to the database using the existing key before the change can be made.

Caution

When the key is changed, every row in the database is decrypted using the old key and re-encrypted using the new key. This operation is unrecoverable. If the application is interrupted part-way through, the database is invalid and cannot be accessed. A new one must be created.

❖ To change the encryption key on an UltraLite database

1. Call **changeEncryptionKey** on the database object, supplying the new key as an argument.

```
db.changeEncryptionKey( "new key" );
```

☞ For more information, see [“changeEncryptionKey method” on page 64](#).

Defragmenting UltraLite databases

The UltraLite store is designed to efficiently reuse free space, so explicit defragmentation is not required under normal circumstances. This section describes a technique to explicitly defragment UltraLite databases, for use by applications with extremely strict space requirements.

UltraLite provides a defragmentation step function, which defragments a small part of the database. To defragment the entire database at once, call the defragmentation step function in a loop until it returns **ul_true**. This can be an expensive operation, and **SQLCODE** must also be checked to detect errors (an error here usually indicates a file I/O error).

Explicit defragmentation occurs incrementally under application control during idle time. Each step is a small operation.

☞ For more information, see [“Class JdbcDefragIterator” on page 66](#).

❖ To defragment an UltraLite database

1. Cast a **Connection** to a **JdbcConnection** object. For example,

```
...  
Connection conn = db.connect();  
JdbcConnection jconn = (JdbcConnection)conn ;
```

UltraLite provides a defragmentation step function, which defragments a small part of the database. To defragment the entire database at once, call the defragmentation step function in a loop until it returns **ul_true**. This can be an expensive operation, and **SQLCODE** must also be checked to detect errors (an error here usually indicates a file I/O error).

2. Call **getDefragIterator()** to obtain a **JdbcDefragIterator** object. For example:

```
JdbcDefragIterator defrag = jconn.getDefragIterator();
```

3. During idle time, call **ulStoreDefragStep()** to defragment a piece of the database.

```
defrag.ulStoreDefragStep();
```


Adding synchronization to your application

Synchronization is a key feature of many UltraLite applications. This section describes how to add synchronization to your application.

The synchronization logic that keeps UltraLite applications up to date with the consolidated database is not held in the application itself.

Synchronization scripts stored in the consolidated database, together with the MobiLink synchronization server and the UltraLite runtime library, control how changes are processed when they are uploaded and determines which changes are to be downloaded.

Overview

The specifics of each synchronization is controlled by a set of synchronization parameters. These parameters are gathered into a structure (C/C++) or object (Java), which is then supplied as an argument in a function call to synchronize. The outline of the method is the same in each development model.

❖ To add synchronization to your application

1. Initialize the structure (C/C++) or object (Java) that holds the synchronization parameters.

☞ For information, see [“Initializing the synchronization parameters” on page 45](#).

2. Assign the parameter values for your application.

☞ For information, see [“Synchronization stream parameters” on page ??](#).

3. Call the synchronization function, supplying the structure or object as argument.

☞ For information, see [“Invoking synchronization” on page 47](#).

You must ensure that there are no uncommitted changes when you synchronize. For more information, see [“Commit all changes before synchronizing” on page 49](#).

Synchronization parameters

Synchronization specifics are controlled through a set of synchronization parameters. For information on these parameters, see [“Synchronization stream parameters” on page ??](#).

Initializing the synchronization parameters

The synchronization parameters are stored in a C/C++ structure or Java object.

In C/C++ the members of the structure may not be well-defined on initialization. You must set your parameters to their initial values with a call to a special function. The synchronization parameters are defined in a structure declared in the UltraLite header file *ulglobal.h*.

In Java, the details of any synchronization, including the URL of the MobiLink synchronization server, the script version to use, the MobiLink user ID, and so on, are all held in a **UISynchOptions** object.

☞ For a complete list of synchronization parameters, see “Synchronization parameters” [*UltraLite Database User’s Guide*, page 162].

❖ To initialize the synchronization parameters (Java)

1. Create a **UISynchOptions** object. For example:

```
UISynchOptions synch_options = new UISynchOptions();
```

2. Set the required parameters.

The **UISynchOptions()** object has a set of methods to set and get its fields. For a list, see “Synchronization parameters” [*UltraLite Database User’s Guide*, page 162]. Use these methods to set the required synchronization parameters before synchronizing. For example:

```
opts.setUserName( "50" );  
opts.setScriptVersion( "default" );  
opts.setStream( new UISocketStream() );
```

Setting synchronization parameters

The synchronization streams for UltraLite Java applications are objects, and are set by their constructors. The available streams are as follows:

- ◆ **UISocketStream** TCP/IP synchronization
- ◆ **UISecureSocketStream** TCP/IP synchronization with Certicom elliptic-curve transport-layer security.
- ◆ **UISecureRSASocketStream** TCP/IP synchronization with Certicom RSA transport-layer security.
- ◆ **UIHTTPStream** HTTP synchronization.
- ◆ **UIHTTPSSStream** HTTPS synchronization.

The following line sets the stream to TCP/IP:

```
synch_opts.setStream( new UISocketStream() );
```

☞ For more information, see “Synchronization parameters” [*UltraLite Database User’s Guide*, page 162].

Separately-licensable option required

Use of **UIHTTPStream**, **UISecureSocketStream** and **UISecureRSASocketStream** require Certicom technology, which in turn requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations. For more information on this option, see “Welcome to SQL Anywhere Studio” [*Introducing SQL Anywhere Studio*, page 4].

☞ For information on the individual parameters, see [“Synchronization stream parameters” on page ??](#).

Once you have initialized the synchronization parameters, and set them to the values needed for your application, you can initiate synchronization using the **JdbcConnection.synchronize()** method.

The method takes a **UISynchOptions** object as argument. The set of calls needed to synchronize is as follows:

```
UISynchOptions opts = new UISynchOptions();
opts.setUserName( "50" );
opts.setScriptVersion( "default" );
opts.setStream( new UISocketStream() );
opts.setStreamParms( "host=123.45.678.90" );
conn.synchronize( opts );
```

Invoking synchronization

The details of how to invoke synchronization depends on your target platform and on the synchronization stream.

The synchronization process can only work if the device running the UltraLite application is able to communicate with the synchronization server. For some platforms, this means that the device needs to be physically connected by placing it in its cradle or by attaching it to a server computer using a cable. You need to add error handling code to your application in case the synchronization cannot be carried out.

❖ To invoke synchronization (TCP/IP, HTTP, or HTTPS streams)

1. Construct a new **UISynchInfo** object to initialize the synchronization parameters, and **JdbcConnection.synchronize()** to synchronize. See [“Adding synchronization to your application” on page ??](#).

The synchronization call requires a structure that holds a set of parameters describing the specifics of the synchronization. The particular parameters

used depend on the stream.

Using transport-layer security

For additional security during synchronization, you can use transport-layer security encrypt messages as they pass between UltraLite application and the consolidated database.

☞ For information about encryption technology, see “Transport-Layer Security” [*MobiLink Synchronization User’s Guide*, page 337].

☞ Transport-layer security from UltraLite Java client applications uses a separate synchronization stream. You must set up your MobiLink synchronization server as well as your UltraLite client to use this synchronization stream.

Client changes

☞ At the client, you need to choose the **UISecureSocketStream** or **UISecureRSASocketStream** synchronization stream, and supply a set of stream parameters. The stream parameters include parameters that control security.

Set the parameter as follows:

```
UISynchOptions opts = new UISynchOptions();
opts.setStream(new UISecureSocketStream() );
opts.setStreamParms( "host=myserver;"
    + "port=2439;"
    + "certificate_company=Sybase Inc.;"
    + "certificate_unit="MEC";"
    + "certificate_name=Mobilink");
// set other options here
conn.synchronize( opts );
```

☞ For details on the stream parameters, see “[UISecureSocketStream synchronization parameters](#)” on page ??.

Setting up the MobiLink server

As the secure synchronization streams for Java applications are separate streams, you must ensure that the MobiLink synchronization server is listening for it. To do this, you must supply the **java_certicom_tls** or **java_rsa_tls** synchronization streams, to match your choice on the client.

The following command line is an example:

```
dbmlsrv9 -x java_certicom_
        tls(certificate=mycertificate.crt;port=1234)
```

The security parameters for the **java_certicom_tls** and **java_rsa_tls** streams are as follows:

◆ **certificate** The name of the certificate file that contains the server’s

identity. This file needs to include the server's certificate, the certificates of all the certificate authorities in the certificate signing chain, and the server's private key.

The certificate parameter defaults to *sample.crt* for **java_certicom_tls** and *rsaserver.crt* for **java_rsa_tls**, which is the default identity for MobiLink. These files are distributed with SQL Anywhere Studio, in the same directory as the MobiLink server.

- ◆ **certificate_password** The password used to encrypt the private key in the certificate file.

The default is the password for the private key in *sample.crt* and *rsaserver.crt*, which is **test**.

Commit all changes before synchronizing

☞ An UltraLite database cannot have uncommitted changes when it is synchronized. If you attempt to synchronize an UltraLite database when any connection has an uncommitted transaction, the synchronization fails, an exception is thrown and the `SQLE_UNCOMMITTED_TRANSACTIONS` error is set. This error code also appears in the MobiLink synchronization server log.

☞ For more information on download-only synchronizations, see “download_only synchronization parameter” [*UltraLite Database User's Guide*, page 165].

Adding initial data to your application

Many UltraLite application need data in order to start working. You can download data into your application by synchronizing. You may want to add logic to your application to ensure that, the first time it is run, it downloads all necessary data before any other actions are carried out.

Development tip

It is easier to locate errors if you develop an application in stages. When developing a prototype, temporarily code `INSERT` statements in your application to provide data for testing and demonstration purposes. Once your prototype is working correctly, enable synchronization and discard the temporary `INSERT` statements.

For more synchronization development tips, see “Development tips” [*MobiLink Synchronization User's Guide*, page 71].

Monitoring and canceling synchronization

This section describes how to monitor and cancel synchronization from UltraLite applications.

- ◆ An API for monitoring synchronization progress and for canceling synchronization.
- ◆ A progress indicator component that implements the interface, which you can add to your application.

Monitoring synchronization

To monitor synchronization, write a class that implements the **UISynchObserver** interface. This interface contains a single method:

```
void updateSynchronizationStatus( UISynchStatus status )
```

- ◆ Register your **UISynchObserver** object using the **UISynchOptions** class.
- ◆ Call the **synchronize()** method to synchronize.
- ◆ UltraLite calls the **updateSynchronizationStatus** method of your observer class whenever the synchronization state changes. The following section describes the synchronization state.

Here is a typical sequence of instructions for synchronization. In this example, the class **MyObserver** implements the **UISynchObserver** interface:

```
UISynchObserver observer = new MyObserver ( );
UISynchOptions opts = new UISynchOptions();
// set options
opts.setUserName( "mluser" );
opts.setPassword( "mlpwd" );
opts.setStream( new ULSocketStream() );
opts.setStreamParms( "localhost" );
opts.setObserver( observer );
opts.setUserData( myDataObject );
// synchronize
conn.synchronize( opts );
```

Handling synchronization status information

The class that implements **UISynchObserver**, the **UISynchStatus** object holds synchronization status information. This object is filled by UltraLite with synchronization status information each time your **updateSynchronizationStatus** method is called.

The **UISynchStatus** object has the following methods:

```
int getState()
int getTableCount()
int getTableIndex()
Object getUserData()
UlSynchOptions getSynchOptions()
UlSqlStmt getStatement()
int getErrorCode()
boolean isOKToContinue()
void cancelSynchronization()
```

- ◆ **getState** One of the following states:
 - **STARTING** No synchronization actions have yet been taken.
 - **CONNECTING** The synchronization stream has been built, but not yet opened.
 - **SENDING_HEADER** The synchronization stream has been opened, and the header is about to be sent.
 - **SENDING_TABLE** A table is being sent.
 - **UL_SYNCH_STATE_SENDING_DATA** Schema information or data is being sent.
 - **UL_SYNCH_STATE_FINISHING_UPLOAD** The upload stage is completed and a commit is being carried out.
 - **RECEIVING_UPLOAD_ACK** An acknowledgement that the upload is complete is being received.
 - **RECEIVING_TABLE** A table is being received.
 - **UL_SYNCH_STATE_SENDING_DATA** Schema information or data is being received.
 - **UL_SYNCH_STATE_COMMITTING_DOWNLOAD** The download stage is completed and a commit is being carried out.
 - **SENDING_DOWNLOAD_ACK** An acknowledgement that download is complete is being sent.
 - **DISCONNECTING** The synchronization stream is about to be closed.
 - **DONE** Synchronization has completed successfully.
 - **ERROR** Synchronization has completed, but with an error.
 - ☞ For a description of the synchronization process, see “The synchronization process” [*MobiLink Synchronization User’s Guide*, page 21].
- ◆ **getTableCount** Returns the number of tables being synchronized. For each table there is a sending and receiving phase, so this number may be more than the number of tables being synchronized.

-
- ◆ **getTableIndex** The current table which is being uploaded or downloaded, starting at 0. This number may skip values when not all tables are being synchronized.
 - ◆ **getSyncOptions** Returns the **UISyncOptions** object.
 - ◆ **sent.inserts** The number of inserted rows that have been uploaded so far.
 - ◆ **sent.updates** The number of updated rows that have been uploaded so far.
 - ◆ **sent.deletes** The number of deleted rows that have been uploaded so far.
 - ◆ **sent.bytes** The number of bytes that have been uploaded so far.
 - ◆ **received.inserts** The number of inserted rows that have been downloaded so far.
 - ◆ **received.updates** The number of updated rows that have been downloaded so far.
 - ◆ **received.deletes** The number of deleted rows that have been downloaded so far.
 - ◆ **received.bytes** The number of bytes that have been downloaded so far.
 - ◆ **cancelSynchronization** Set this member to true to interrupt the synchronization. The SQL exception `SQLE_INTERRUPTED` is set, and the synchronization stops as if a communications error had occurred. The observer is *always* called with either the `DONE` or `ERROR` state so that it can do proper cleanup.
 - ◆ **getUserData** Returns the user data object.
 - ◆ **getStatement** Returns the statement that called the synchronization. The statement is an internal UltraLite statement, and this method is unlikely to be of practical use, but is included for completion.
 - ◆ **getErrorCode** When the synchronization state is set to `ERROR`, this method returns a diagnostic error code.
 - ◆ **isOKToContinue** This is set to **false** when **cancelSynchronization** is called. Otherwise, it is **true**.

Example

The following code illustrates a very simple observer function:

```
void updateSynchronizationStatus( ULSynchStatus status )
{
    int state = status.getState();
    System.out.println( "Sync status: " + state );
    if( state == ULSynchStatus.SENDING_TABLE ||
        state == ULSynchStatus.RECEIVING_TABLE ){
        System.out.println( "send/receive table " +
            ( status.getTableIndex() + 1 ) +
            " of " + status.getTableCount() );
    }
}
```

CustDB example

An example of an observer function is included in the CustDB sample application. The implementation in CustDB provides a dialog that displays synchronization progress and allows the user to cancel synchronization. The user-interface component makes the observer function platform specific.

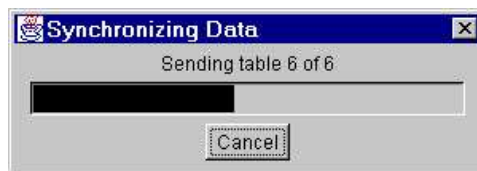
The CustDB sample code is in the *Samples\UltraLite\CustDB* subdirectory of your SQL Anywhere directory. The observer function is contained in the platform-specific subdirectories of the *CustDB* directory.

Using the progress viewer

The UltraLite runtime library includes two progress viewer classes, which provide an implementation of synchronization monitoring, together with the ability for end users to cancel synchronization. The progress viewer classes are as follows:

- ◆ **ianywhere.ultralite.ui.SynchProgressViewer** A heavyweight AWT version.
- ◆ **ianywhere.ultralite.ui.SynchProgressViewer** A Swing version of the viewer that respects the Swing threading model.

The two classes are used identically. The viewer displays a modal or modeless dialog, which shows a series of messages and a progress bar. Both the messages and the bar are updated during synchronization. The viewer also provides a Cancel button. If the user clicks the Cancel button, synchronization stops and the SQL exception `SQL_INTERRUPTED` is thrown.



Threading issues

In a Java application, all events occur on a single thread called the **event**

thread. Also, all user interface objects are created on the event thread, even if the application is on a different thread at the time. There is only one event thread in an application.

The event thread must never block. Consequently, you should not perform long operations on the event thread, as this leads to painting aberrations. Even calling the **show()** method on a modal dialog suspends execution of the event thread. You must therefore avoid calling the **synchronize()** method on the event thread.

Displaying a modal viewer

The following code snippet illustrates how to invoke a modal instance of the viewer. The **import** statement uses the AWT version:

```
import ianywhere.ultralite.ui.SynchProgressViewer;
// create a frame to display a dialog
java.awt.Frame frame = ...;
// get UltraLite connection
Connection conn = ...;
// set synchronization options
UlSynchOptions options = new UlSynchOptions();
options.setUserName( "my_user" );
...
// create the viewer
SynchProgressViewer viewer = new SynchProgressViewer( frame );
viewer.synchronize( frame, options );
// execution stops here until synchronization is complete
```

When invoked in this manner, the viewer carries out the following operations:

1. registers itself as a synchronization observer,
2. spawns a thread to do the synchronization,
3. displays itself, blocking the current thread.
4. When synchronization finishes, the observer callback disposes of the dialog, which lets the thread continue.

Displaying a modeless viewer

The following code snippet illustrates how to invoke a modeless instance of the viewer:

```
SynchProgressViewer viewer = new SynchProgressViewer( frame,
    false );
options.setObserver( viewer );
conn.synchronize( options );
```

In this case, you must ensure that the synchronization occurs on a thread other than the event thread, so that the viewer is not blocked.

Notes

- ◆ All messages come from the **SynchProgressViewerResources** resource bundle.

- ◆ The viewer implements the **UISynchObserver** interface so it can hook into the synchronization process.
- ◆ The CustDB sample application includes a progress viewer. The CustDB sample code is in the *UltraLite\samples\CustDB\java* subdirectory of your SQL Anywhere directory.

Developing multi-threaded applications

The UltraLite Java runtime library is thread-safe. Users of the Sun Java VM must use version 1.2 or later to run multi-threaded UltraLite applications. Users of the Jeode VM on Pocket PC and the IBM Java VM can run multi-threaded UltraLite applications even though these VMs are based on JDK 1.1.8.

The entire runtime is treated as a single critical section, only allowing one thread to enter it at a time.

☞ For more information, see [“Using the UltraLite JdbcDatabase.connect method” on page 26](#).

CHAPTER 5

UltraLite Static Java API Reference

About this chapter

This chapter describes the API for developing pure Java UltraLite applications using the static Java API.

Only those parts that differ from JDBC are explicitly documented.

Contents

Topic:	page
UltraLite API reference	58

UltraLite API reference

This section describes extensions to the JDBC interface provided by UltraLite, and also describes JDBC features unsupported in UltraLite.

JDBC features in UltraLite

The following are features and limitations specific to the development of JDBC UltraLite applications.

The UltraLite static Java API is modeled on JDBC 1.2, with the addition of the following **ResultSet** methods from JDBC 2.0:

- ◆ `absolute()`,
- ◆ `afterLast()`,
- ◆ `beforeFirst()`,
- ◆ `first()`,
- ◆ `isAfterLast()`,
- ◆ `isBeforeFirst()`,
- ◆ `isFirst()`,
- ◆ `isLast()`,
- ◆ `last()`,
- ◆ `previous()`,
- ◆ `relative()`

The following features are incompatible with the UltraLite development model and are not supported by UltraLite.

- ◆ There is only limited support for metadata access (system table access). Therefore, you cannot use the **DatabaseMetaData** Interface. Metadata access is limited to the number and type of columns.
- ◆ Java objects cannot be stored in the database
- ◆ There is no support for stored procedures or stored functions.
- ◆ Only static SQL statements are supported and they must be added to the database so that the UltraLite generator can generate them.

Unsupported JDBC methods

UltraLite does not support the following JDBC 1.2 methods. An attempt to use any of the following methods results in a **SQLException** with a vendor code indicating that the feature is not supported in UltraLite.

Connection interface	<ul style="list-style-type: none"> ◆ <code>getCatalog</code> ◆ <code>getMetaData</code> ◆ <code>getTransactionIsolation</code> ◆ <code>setCatalog</code> ◆ <code>setTransactionIsolation</code>
ResultSet interface	<ul style="list-style-type: none"> ◆ <code>getMetaData</code>
Statement interface	<ul style="list-style-type: none"> ◆ <code>cancel</code> ◆ <code>getMaxFieldSize</code> ◆ <code>getMaxRows</code> ◆ <code>setMaxFieldSize</code> ◆ <code>setMaxRows</code>

Class JdbcConnection

Package	<code>ianywhere.ultralite.jdbc</code>
Description	Represents an UltraLite database connection. Most methods are inherited from the JDBC Connection class. Unsupported methods throw an <code>unsupported feature exception</code> .

countUploadRows method

Prototype	<code>long countUploadRows(int <i>mask</i>, long <i>threshold</i>)</code>
Description	<p>Returns the number of rows that need to be uploaded when the next synchronization takes place.</p> <p>You can use this function to determine if a synchronization is needed.</p>
Parameters	<p>mask A set of publications to check. A value of 0 corresponds to the entire database. The set is supplied as a mask. For example, the following mask corresponds to publications PUB1 and PUB2.:</p>

`UL_PUB_PUB1 | UL_PUB_PUB2`

☞ For more information on publication masks, see [“publication synchronization parameter” on page 77](#).

threshold A value that determines the maximum number of rows to count, and so limits the amount of time taken by the call. A value of 0 corresponds to no limit. A value of 1 determines if any rows need to be synchronized.

Returns The number of rows to be uploaded.

Throws `java.sql.SQLException`

getDefragIterator method

Prototype `JdbcDefragIterator getDefragIterator()`

Description Initializes and returns a defragmentation iterator.

Parameters **user_name** The MobiLink user name. See [“user_name synchronization parameter” on page 85](#).

password The password associated with `user_name`. See [“password synchronization parameter” on page 75](#).

script_version The script version. See [“version synchronization parameter” on page 85](#).

stream_defn The stream to use for synchronization. See [“stream synchronization parameter” on page 80](#).

parms Any user-supplied parameters used for the synchronization.

☞ See [“stream_parms synchronization parameter” on page 83](#).

Returns The defragmentation iterator.

Throws `java.sql.SQLException`

See also [“Defragmenting UltraLite databases” on page 44](#)

getLastDownloadTimeDate method

Prototype `java.util.Date getLastDownloadTimeDate(int mask)`

Description Returns the last time changes to the result set of a given statement were downloaded.

Parameters **mask** A set of publications for which the last download time is retrieved. A value of 0 corresponds to the entire database. The set is supplied as a mask. For example, the following mask corresponds to publications PUB1 and PUB2.:

`UL_PUB_PUB1 | UL_PUB_PUB2`

☞ For more information on publication masks, see “[publication synchronization parameter](#)” on page 77.

Returns ♦ The last time the statement was downloaded.

getLastDownloadTimeLong method

Prototype long **getLastDownloadTimeLong**(int *mask*)

Description Returns the last time changes to the result set of a given statement were downloaded.

Parameters **mask** A set of publications for which the last download time is retrieved. A value of 0 corresponds to the entire database. The set is supplied as a mask. For example, the following mask corresponds to publications PUB1 and PUB2.:

```
UL_PUB_PUB1 | UL_PUB_PUB2
```

☞ For more information on publication masks, see “[publication synchronization parameter](#)” on page 77.

Returns ♦ The last time the statement was downloaded.

getLastIdentity method

Prototype long **getLastIdentity**()

Description Returns the most recent identity value used. This function is equivalent to the following SQL statement:

```
SELECT @@identity
```

The function is particularly useful in the context of global autoincrement columns.

Returns The last identity value.

See also “Determining the most recently assigned value” [*UltraLite Database User’s Guide*, page 154]

“Setting the global database identifier” [*UltraLite Database User’s Guide*, page 152]

globalAutoincUsage method

Prototype short **globalAutoincUsage**()

Description Returns the maximum global autoincrement counter percentage of all tables in the database. The value is useful when deciding whether to set a database ID.

Returns	The percentage of global autoincrement values that have been used.
Throws	java.sql.SQLException
See also	“Declaring default global autoincrement columns” [<i>UltraLite Database User’s Guide</i> , page 151] “setDatabaseID method” on page 62

grant method

Prototype	void grant (String <i>user</i> , String <i>password</i>)
Description	Grants a user name and password permission to connect to an UltraLite database. To take effect, this method requires that user authentication has been enabled with JdbcSupport.enableUserAuthentication . The grant method is supplied on JdbcConnection for applications that do not have an explicit JdbcDatabase object.
Parameters	user A string that must be entered as a user name when connecting. password A string that must be entered as a password when connecting.
Returns	void.
Throws	java.sql.SQLException

revoke method

Prototype	void revoke (String <i>user</i>)
Description	Revokes permission to connect to an UltraLite database from a user name. To take effect, this method requires that user authentication has been enabled with JdbcSupport.enableUserAuthentication . The grant method is supplied on JdbcConnection for applications that do not have an explicit JdbcDatabase object.
Parameters	user The user name that can no longer connect to the database.
Returns	void.
Throws	java.sql.SQLException

setDatabaseID method

Prototype	void setDatabaseID (int <i>value</i>)
Description	Sets the
Parameters	value The integer value to use as the global database identifier.

Throws `java.sql.SQLException`

See also [“globalAutoincUsage method” on page 61](#)

synchronize method

Prototype `void synchronize(
java.lang.String user_name,
java.lang.String password,
java.lang.String script_version,
UIStream stream_defn,
java.lang.String parms)`


Description Synchronizes data with a MobiLink synchronization server.

Parameters **user_name** The MobiLink user name. See [“user_name synchronization parameter” on page 85](#).

password The password associated with user_name. See [“password synchronization parameter” on page 75](#).

script_version The script version. See [“version synchronization parameter” on page 85](#).

stream_defn The stream to use for synchronization. See [“stream synchronization parameter” on page 80](#).

parms Any user-supplied parameters used for the synchronization.
 See [“stream_parms synchronization parameter” on page 83](#).

Throws `java.sql.SQLException`

startSynchronizationDelete method

Prototype `void startSynchronizationDelete()`

Description Restart logging of deletes for MobiLink synchronization

Throws `java.sql.SQLException`

See also [“START SYNCHRONIZATION DELETE statement \[MobiLink\]” \[ASA SQL Reference, page 573\]](#)

stopSynchronizationDelete method

Prototype `void stopSynchronizationDelete()`

Description Prevent logging of deletes for MobiLink synchronization.

Throws `java.sql.SQLException`

See also “STOP SYNCHRONIZATION DELETE statement [MobiLink]” [ASA *SQL Reference*, page 580]

Class JdbcDatabase

Package ianywhere.ultralite.jdbc

Description The **JdbcDatabase** is used directly only for obfuscating databases. The generated database class extends **JdbcDatabase** and provides an object that represents the UltraLite database. Most JdbcDatabase methods are used from the generated database class.

☞ For more information, see “[The generated database class](#)” on page 66.

changeEncryptionKey method

Prototype Connection **changeEncryptionKey**()

Description Changes the encryption key for an UltraLite database.

Returns A JDBC Connection object.

Throws java.sql.SQLException

See also “[Encrypting UltraLite databases](#)” on page 41

close method

Prototype void **close**()

Description Closes all connections to an UltraLite database. This method must be executed before an UltraLite database can be deleted.

Returns void

Throws java.sql.SQLException

connect method

Prototype Connection **connect**()

Connection **connect**(String *user*, String *password*)

Connection **connect**(String *user*, String *password*, Properties *info*)

Description Connects to an UltraLite database. The user name and password are checked only when user authentication has been enabled with **JdbcSupport.enableUserAuthentication**.

Parameters **user** A user name that can connect to the database.

password A string that must be entered as a password when connecting.

info A **Properties** object holding the user name and password.

Returns A JDBC Connection object.

Throws java.sql.SQLException

drop method

Prototype void **drop**()

Description Deletes an UltraLite database file. This method should be used with care, and can be executed only after the `JdbcDatabase.close()` method has been called.

Returns void

Throws java.sql.SQLException

See also [“close method” on page 64](#)

grant method

Prototype void **grant**(String *user*, String *password*)

Description Grants a user name and password permission to connect to an UltraLite database. To take effect, this method requires that user authentication has been enabled with **JdbcSupport.enableUserAuthentication**.

Parameters **user** A string that must be entered as a user name when connecting.

password A string that must be entered as a password when connecting.

Returns void.

Throws java.sql.SQLException

revoke method

Prototype void **revoke**(String *user*)

Description Revokes permission to connect to an UltraLite database from a user name. To take effect, this method requires that user authentication has been enabled with **JdbcSupport.enableUserAuthentication**.

Parameters **user** The user name that can no longer connect to the database.

Returns void.

Throws java.sql.SQLException

setDefaultObfuscation method

Prototype	setDefaultObfuscation (true false)
Description	Obfuscates the database
See also	“Obfuscating an UltraLite database” on page 42

The generated database class

Description	The generated database class extends JdbcDatabase . It provides an object that represents the UltraLite database. JdbcDatabase methods are typically used on the generated database class.
Constructor	<p><code>new database-name(Properties props)</code></p> <p>where <i>database-name</i> is the name of the generated database class. You can specify the class name using the UltraLite generator <code>-f</code> command-line option.</p> <p>☞ For more information, see “The UltraLite generator” [<i>UltraLite Database User’s Guide</i>, page 96].</p>
Parameters	<p>props A Properties object containing some or all of the following items:</p> <ul style="list-style-type: none">◆ persist◆ persistfile◆ key <p>☞ For more information, see “Using a Properties object to store connection information” on page 29.</p>

Class JdbcDefragIterator

Package	ianywhere.ultralite.jdbc
Description	Provides an object used for explicit defragmentation of the database store.

ulStoreDefragStep method

Prototype	boolean ulStoreDefragStep(UIConnection conn)
Description	Defragments a portion of an UltraLite database.
Parameters	conn The current connection, as a JdbcConnection object.
Returns	true if successful.

	false in unsuccessful.
Throws	java.sql.SQLException
See also	“STOP SYNCHRONIZATION DELETE statement [MobiLink]” [ASA <i>SQL Reference</i> , page 580]

Class JdbcSupport

Package	ianywhere.ultralite.jdbc
Description	A static class that provides methods to enable UltraLite features.

enableUserAuthentication method

Prototype	void enableUserAuthentication ()
Description	Sets the UltraLite database so that user authentication is required to connect to it. Must be called before the database object is created.
Parameters	None.
Returns	Void.
Throws	java.sql.SQLException
See also	“User authentication example” on page 39

disableUserAuthentication method

Prototype	void disableUserAuthentication ()
Description	Sets the UltraLite database so that user authentication is not required to connect to it. Must be called before the database object is created.
Parameters	None.
Returns	Void.
Throws	java.sql.SQLException
See also	“enableUserAuthentication method” on page 67

CHAPTER 6

Synchronization Parameters Reference

About this chapter

This chapter provides reference information about synchronization parameters.

Contents

Topic:	page
Synchronization parameters	70

Synchronization parameters

The synchronization parameters are fields of the `ianywhere.ultralite.runtime.UISynchOptions` object. The fields are private, and methods are provided to get and set their values. The `UISynchOptions` object is provided as an argument in the call to `ianywhere.ultralite.jdbc.JdbcConnection.synchronize`.

Field	Access methods	See
auth_status	<code>getAuthStatus</code> <code>setAuthStatus</code>	“auth_status member” on page 71
auth_value	<code>getAuthValue</code> <code>setAuthValue</code>	“auth_value synchronization parameter” on page 72
download_only	<code>getDownloadOnly</code> <code>setDownloadOnly</code>	“download_only synchronization parameter” on page 73
ignored_rows	<code>getIgnoredRows</code> <code>setIgnoredRows</code>	“ignored_rows synchronization parameter” on page 74
new_password	<code>getNewPassword</code> <code>setNewPassword</code>	“new_password synchronization parameter” on page 74
observer	<code>getObserver</code> <code>setObserver</code>	“observer synchronization parameter” on page 75
password	<code>getPassword</code> <code>setPassword</code>	“password synchronization parameter” on page 75
ping	<code>getPing</code> <code>setPing</code>	“ping synchronization parameter” on page 76
publications	<code>getSynchPublication</code> <code>setSynchPublication</code>	“publication synchronization parameter” on page 77
stream	<code>getStream</code> <code>setStream</code>	“stream synchronization parameter” on page 80
stream_parms	<code>getStreamParms</code> <code>setStreamParms</code>	“stream_parms synchronization parameter” on page 83
upload_ok	<code>getUploadOK</code> <code>setUploadOK</code>	“upload_ok synchronization parameter” on page 83

Field	Access methods	See
upload_only	getUploadOnly setUploadOnly	“upload_only synchronization parameter” on page 84
user_data	getUserData setUserData	“user_data synchronization parameter” on page 84
user_name	getUserName setUserName	“user_name synchronization parameter” on page 85
version	getScriptVersion setScriptVersion	“version synchronization parameter” on page 85

☞ For a description of the role of each synchronization parameter, see “Synchronization parameters” [*UltraLite Database User’s Guide*, page 162].

auth_parms parameter

Function Provides parameters to a custom user authentication script.

Access method `String[] getAuthParms()`

`void setAuthParms(String[] auth_parms)`

Usage Set the parameters as follows:

```
params = new String[ num_params ];
// set params values
UlSynchOptions opts = new UlSynchOptions();
opts.setAuthParms( params );
opts.setAuthParmsNumber( num_params );
```

See also [“num_auth_parms parameter” on page 74](#)

“authenticate_parameters connection event” [*MobiLink Synchronization Reference*, page 98]

“authenticate_user connection event” [*MobiLink Synchronization Reference*, page 100]

auth_status parameter

Function Reports the status of MobiLink user authentication.

Access method `short getAuthStatus()`

`void setAuthStatus(short auth_status)`

Usage Access the parameter as follows:

```
UlSynchOptions opts = new UlSynchOptions;  
// set options here  
conn.synchronize( opts );  
returncode = opts.getAuthStatus();
```

Allowed values After synchronization, the parameter must hold one of the following values. If a custom **authenticate_user** synchronization script at the consolidated database returns a different value, the value is interpreted according to the rules given in “authenticate_user connection event” [*MobiLink Synchronization Reference*, page 100].

Constant	Value	Description
UIDefnUL_AUTH_STATUS_-UNKNOWN	0	Authorization status is unknown, possibly because the connection has not yet synchronized.
UIDefnUL_AUTH_STATUS_-VALID	1000	User ID and password were valid at the time of synchronization.
UIDefnUL_AUTH_STATUS_-VALID_BUT_EXPIRES_SOON	2000	User ID and password were valid at the time of synchronization but will expire soon.
UIDefnUL_AUTH_STATUS_-EXPIRED	3000	Authorization failed: user ID or password have expired.
UIDefnUL_AUTH_STATUS_-INVALID	4000	Authorization failed: bad user ID or password.
UIDefnUL_AUTH_STATUS_-IN_USE	5000	Authorization failed: user ID is already in use.

See also “Authenticating MobiLink Users” [*MobiLink Synchronization User’s Guide*, page 103].

auth_value synchronization parameter

Function Reports return values from custom user authentication synchronization scripts.

Access methods long **getAuthValue()**

void **setAuthValue(long auth_value)**

Default The values set by the default MobiLink user authentication mechanism are

described in [“auth_status synchronization parameter”](#) on page 71.

Usage

The parameter is read-only.

Access the parameter as follows:

```
UlsynchOptions opts = new UlsynchOptions();
// set other options here
conn.synchronize( opts );
returncode = opts.getAuthValue();
```

See also

“authenticate_user connection event” [*MobiLink Synchronization Reference*, page 100]

“authenticate_user_hashed connection event” [*MobiLink Synchronization Reference*, page 104]

[“auth_status synchronization parameter”](#) on page 71

checkpoint_store synchronization parameter

Function

Adds additional checkpoints of the database during synchronization to limit database growth during the synchronization process.

Access methods

This parameter is not available in Java

Default

By default, limited checkpointing is done.

Usage

Set the parameter as follows:

disable_concurrency synchronization parameter

Function

Disallow database access from other threads during synchronization.

Access methods

This parameter is not available in Java

Default

By default, data access is available. Data access is read-write during the download phase, and read-only otherwise.

Usage

Set the parameter as follows:

See also

“Threading in UltraLite applications” [*UltraLite Database User’s Guide*, page 47]

download_only synchronization parameter

Function

Do not upload any changes from the UltraLite database during this synchronization.

Access methods

boolean **getDownloadOnly()**

	void setDownloadOnly (boolean <i>download_only</i>)
Default	The parameter is an optional Boolean value, and by default is false.
Usage	Set the parameter as follows:

```
UlsynchOptions opts = new UlsynchOptions;  
opts.setDownloadOnly( true );  
// set other options here  
conn.synchronize( opts );
```

See also	“Including read-only tables in an UltraLite database” on page ?? . “upload_only synchronization parameter” on page 84
----------	--

ignored_rows synchronization parameter

Function	Reports if any rows were ignored by the MobiLink synchronization server during synchronization because of absent scripts. The parameter is read-only.
Access methods	boolean getIgnoredRows () void setIgnoredRows (boolean <i>ignored_rows</i>)

new_password synchronization parameter


Function	Sets a new MobiLink password associated with the user name.
Access methods	java.lang.String getNewPassword () void setNewPassword (java.lang.String <i>new_password</i>)
Default	There is no default.
Usage	Set the parameter as follows:

```
UlsynchOptions opts = new UlsynchOptions;  
opts.setUserName( "50" );  
opts.setPassword( "mypassword" );  
opts.setNewPassword( "mynewpassword" );  
// set other options here  
conn.synchronize( opts );
```

See also	“Authenticating MobiLink Users” [MobiLink Synchronization User’s Guide, page 103] .
----------	---

num_auth_parms parameter

Function	The number of authentication parameter strings passed to a custom
----------	---

	authentication script.
Access methods	byte getAuthParmsNumber() void setAuthParmsNumber(byte value)
Default	No parameters passed to a custom authentication script.
Usage	The parameter is used together with <code>auth_parms</code> to supply information to custom authentication scripts.  For more information, see “auth_parms parameter” on page 71 .
See also	“auth_parms parameter” on page 71 “authenticate_parameters connection event” [<i>MobiLink Synchronization Reference</i> , page 98] “authenticate_user connection event” [<i>MobiLink Synchronization Reference</i> , page 100]

observer synchronization parameter

Function	A pointer to a callback function that monitors synchronization.
Access methods	ianywhere.ultralite.runtime.UISynchObserver getObserver() void setObserver(ianywhere.ultralite.runtime.UISynchObserver observer)
See also	“Monitoring and canceling synchronization” on page 50 “user_data synchronization parameter” on page 84

password synchronization parameter

Function	A string specifying the MobiLink password associated with the user_name . This user name and password are separate from any database user ID and password, and serves to identify and authenticate the application to the MobiLink synchronization server.
Access methods	java.lang.String getPassword() void setPassword(java.lang.String password)
Default	There is no default.
Usage	Set the parameter as follows:

```

UlSynchOptions opts = new UlSynchOptions;
opts.setUserName( "50" );
opts.setPassword( "mypassword" );
// set other options here
conn.synchronize( opts );

```

See also “Authenticating MobiLink Users” [*MobiLink Synchronization User’s Guide*, page 103].

ping synchronization parameter

Function Confirm communications between the UltraLite client and the MobiLink synchronization server. When this parameter is set to true, no synchronization takes place.

When the MobiLink synchronization server receives a ping request, it connects to the consolidated database, authenticates the user, and then sends the authenticating user status and value back to the client.

If the ping succeeds, the MobiLink server issues an information message. If the ping does not succeed, it issues an error message.

If the MobiLink user name cannot be found in the ml_user system table and the MobiLink server is running with the command line option -zu+, the MobiLink server adds the user to ml_user.

The MobiLink synchronization server may execute the following scripts, if they exist, for a ping request:

- ◆ begin_connection
- ◆ authenticate_user
- ◆ authenticate_user_hashed
- ◆ end_connection

Access methods boolean **getPing()**

void **setPing(boolean ping)**

Default The parameter is optional, and is a boolean.

Usage Set the parameter as follows:

```

UlSynchOptions opts = new UlSynchOptions;
opts.setUserName( "50" );
opts.setPing( true );
// set other options here
conn.synchronize( opts );

```


See also [“-pi option” \[MobiLink Synchronization Reference, page 76\]](#)

publication synchronization parameter

Function	Specifies the publications to be synchronized.
Access methods	int getSynchPublication() void setSynchPublication (int <i>publication</i>)
Default	If you do not specify a publication, all data is synchronized.
Usage	The UltraLite generator identifies the publications specified on the <i>ulgen -v</i> command line option as upper case constants with the name <code>UL_PUB_pubname</code> , where <i>pubname</i> is the name given to the -v option. For example, the following command line generates a publication identified by the constant <code>salesproject.UL_PUB_SALES</code> :

```
ulgen -v sales ...
```

When synchronizing, set the publication parameter to a **publication mask**: an OR'd list of publication constants. For example:

```
UlSynchOptions opts = new UlSynchOptions;
opts.setSynchPublication(
    projectname.UL_PUB_MYPUB1 |
    projectname.UL_PUB_MYPUB2 );
// set other options here
conn.synchronize( opts );
```

where *projectname* is the name of the main project class generated by the UltraLite generator.

The special publication mask **UL_SYNC_ALL** describes all the tables in the database, whether in a publication or not. The mask **UL_SYNC_ALL_PUBS** describes all tables in publications in the database.

See also [“The UltraLite generator” on page ??](#)
[“Designing sets of data to synchronize separately” on page ??](#)

security synchronization parameter

Function	Set the UltraLite client to use Certicom encryption technology when exchanging messages with the MobiLink synchronization server.
----------	---

Separately-licensable option required

Use of Certicom technology requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations. For more information on this option, see “Welcome to SQL Anywhere Studio” [*Introducing SQL Anywhere Studio*, page 4].

Access methods	<p>This parameter is not used in Java.</p> <p>To use secure synchronization from UltraLite Java applications, choose a separate stream. For more information, see “Initializing the synchronization options” on page ??.</p>
Default	<p>The Security parameter is null by default, corresponding to no transport-layer security.</p>
Usage	<p>The security stream is specified in addition to the synchronization stream. Allowed values are as follows:</p> <ul style="list-style-type: none">◆ ULSecureCerticomTLSStream() Elliptic-curve transport-layer security provided by Certicom.◆ ULSecureRSATLSStream() RSA transport-layer security provided by Certicom.
See also	<p>“Transport-Layer Security” [<i>MobiLink Synchronization User’s Guide</i>, page 337].</p>

security_parms synchronization parameter

Function	<p>Sets the parameters required when using transport-layer security. This parameter must be used together with the security parameter.</p> <p>☞ For more information, see “security synchronization parameter” on page 77.</p>
Access methods	<p>This parameter is not used in Java.</p> <p>To use secure synchronization from UltraLite Java applications, choose a separate stream. For more information, see “Initializing the synchronization options” on page ??.</p>
Usage	<p>The ULSecureCerticomTLSStream() and ULSecureRSATLSStream() security parameters take a string composed of the following optional parameters, supplied in an semicolon-separated string.</p> <ul style="list-style-type: none">◆ certificate_company The UltraLite application only accepts server certificates when the organization field on the certificate matches this value. By default, this field is not checked.◆ certificate_unit The UltraLite application only accepts server certificates when the organization unit field on the certificate matches this

value. By default, this field is not checked.

- ◆ **certificate_name** The UltraLite application only accepts server certificates when the common name field on the certificate matches this value. By default, this field is not checked.

For example:

```
ul_synch_info info;
...
info.stream = ULSocketStream();
info.security = ULSecureCerticomTLSStream();
info.security_parms =
    UL_TEXT( "certificate_company=Sybase" )
    UL_TEXT( ";" )
    UL_TEXT( "certificate_unit=Sales" );
```

The **security_parms** parameter is a string, and by default is null.

If you use secure synchronization, you must also use the `-r` command-line option on the UltraLite generator. For more information, see [“The UltraLite generator” on page ??](#).


send_column_names synchronization parameter

Function	When send_column_names is set to ul_true UltraLite sends each column name to the MobiLink synchronization server. By default UltraLite does not send column names. This parameter is typically used together with the <code>-za</code> or <code>-ze</code> switch on the MobiLink synchronization server for automatically generating synchronization scripts.
Access methods	This parameter is not available for UltraLite Java applications.
See also	“ <code>-za</code> option” [<i>MobiLink Synchronization Reference</i> , page 28]

send_download_ack synchronization parameter



Function	Set this boolean parameter to false to instruct the MobiLink synchronization server that the client will not provide a download acknowledgement. If the client does send download acknowledgement, the MobiLink synchronization server worker thread must wait for the client to apply the download. If the client does not sent a download acknowledgement, the MobiLink synchronization server is freed up sooner for its next synchronization.
Access methods	This parameter is not available for UltraLite Java applications.

stream synchronization parameter

Function	Set the MobiLink synchronization stream to use for synchronization.  For more information, see “stream_parms synchronization parameter” on page 83.
Access methods	ianywhere.ultralite.runtime.UIStream getStream() void setStream(ianywhere.ultralite.runtime.UIStream <i>stream</i>)
Default	The parameter has no default value, and must be explicitly set.
Usage	<pre>UISynchOptions opts = new UISynchOptions; opts.setStream(new UISocketStream()); opts.setStreamParms("host=myserver;port=2439"); // set other options here conn.synchronize(opts);</pre>

When the type of stream requires a parameter, pass that parameter using the **stream_parms** parameter; otherwise, set the **stream_parms** parameter to null.

The following stream functions are available, but not all are available on all target platforms:

Stream	Description
ActiveSync (not available for static Java)	ActiveSync synchronization (Windows CE only).  For a list of stream parameters, see “ActiveSync parameters” [<i>UltraLite Database User’s Guide</i> , page 179].
UIHTTPStream()	Synchronize via HTTP. The HTTP stream uses TCP/IP as its underlying transport. UltraLite applications act as Web browsers and the MobiLink synchronization server acts as a Web server. UltraLite applications send POST requests to send data to the server and GET requests to read data from the server.  For a list of stream parameters, see “HTTP stream parameters” [<i>UltraLite Database User’s Guide</i> , page 184].

Stream	Description
UIHTTPSSStream()	<p>Synchronize via the HTTPS synchronization stream.</p> <p>The HTTPS stream uses SSL or TLS as its underlying protocol. It operates over Internet protocols (HTTP and TCP/IP).</p> <p>The HTTPS stream requires the use of technology supplied by Certicom. Use of Certicom technology requires that you obtain the separately-licensable SQL Anywhere Studio security option and is subject to export regulations. For more information on this option, see “Welcome to SQL Anywhere Studio” [<i>Introducing SQL Anywhere Studio</i>, page 4].</p> <p>☞ For a list of stream parameters, see “HTTPS stream parameters” [<i>UltraLite Database User’s Guide</i>, page 186].</p>
UISocketStream()	<p>Synchronize via TCP/IP.</p> <p>☞ For a list of stream parameters, see “TCP/IP stream parameters” [<i>UltraLite Database User’s Guide</i>, page 182].</p>
UISecureSocketStream()	<p>TCP/IP or HTTP synchronization with transport-layer security using elliptic curve encryption.</p> <p>☞ For a list of stream parameters, see “UISecureSocketStream synchronization parameters” [<i>UltraLite Database User’s Guide</i>, page 189].</p>
UISecureRSASocketStream()	<p>TCP/IP or HTTP synchronization with transport-layer security using RSA encryption.</p> <p>☞ For a list of stream parameters, see “UISecureRSASocketStream synchronization parameters” [<i>UltraLite Database User’s Guide</i>, page 188].</p>

☞ For information on Java synchronization streams, see [“Initializing the synchronization options” on page ??](#).

stream_error synchronization parameter

Function	Sets a structure to hold communications error reporting information.
Access methods	This feature is not available to Java applications.

Default

The parameter has no default value, and must be explicitly set.

Description

The **stream_error** field is a structure of type **ul_stream_error**.

```
typedef struct ss_error {
    ss_stream_id      stream_id;
    ss_stream_context stream_context;
    ss_error_code     stream_error_code;
    asa_uint32        system_error_code;
    rp_char           *error_string;
    asa_uint32        error_string_length;
} ss_error, *p_ss_error;
```

The structure is defined in *sserror.h*, in the *h* subdirectory of your SQL Anywhere directory.

The **ul_stream_error** fields are as follows:

- ◆ **stream_id** The network layer reporting the error. This enumeration has the following constants:

```
STREAM_ID_TCPIP
STREAM_ID_HTTP
STREAM_ID_CERTICOM_TLS
STREAM_ID_PALM_CONDUIT
STREAM_ID_ACTIVESYNC
```

- ◆ **stream_context** The basic network operation being performed, such as open, read, or write. For details, see *sserror.h*.

- ◆ **stream_error_code** The error reported by the stream itself. The **stream_error_code** is of type **ss_error_code**. The stream error codes are all prefixed with **STREAM_ERROR_**. A write error, for example, is **STREAM_ERROR_WRITE**.

☞ For a listing of error numbers, see “MobiLink Communication Error Messages” [*MobiLink Synchronization Reference*, page 347]. For the error code suffixes, see *sserror.h*.

In this version, to find the constant associated with each number you must count down the number of lines prefixed by **DO_STREAM_Error** in *sserror.h*. For example, to find the constant for error number 10, you use the tenth **DO_STREAM_ERROR** entry in *sserror.h*, which is as follows:

```
DO_STREAM_ERROR( WRITE )
```

The constant associated with this error is therefore **STREAM_ERROR_WRITE**.

- ◆ **stream_error** The network operation being performed (the context) and the error itself as an enumeration constant.
- ◆ **system_error_code** A system-specific error code.

- ◆ **error_string** An application-provided error message

Usage

Check for SQLE_COMMUNICATIONS_ERROR as follows:

This feature is not available for Java applications.

stream_parms synchronization parameter

Function

Sets parameters to configure the synchronization stream.

A semi-colon separated list of parameter assignments. Each assignment is of the form *keyword=value*, where the allowed sets of keywords depends on the synchronization stream.

For a list of available parameters for each stream, see the following sections:

- ◆ “HTTP stream parameters” [*UltraLite Database User’s Guide*, page 184]
- ◆ “HTTPS stream parameters” [*UltraLite Database User’s Guide*, page 186]
- ◆ “TCP/IP stream parameters” [*UltraLite Database User’s Guide*, page 182]
- ◆ “UISecureRSASocketStream synchronization parameters” [*UltraLite Database User’s Guide*, page 188]

Access methods

java.lang.String **getStreamParms()**

void **setStreamParms(** java.lang.String *stream_parms* **)**

Default

The parameter is optional, is a string, and by default is null.

Usage

Set the parameter as follows:

```
UISynchOptions synch_options = new UISynchOptions();
synch_opts.setStream( new UIStream() );
synch_opts.setStreamParms( "host=myserver;port=2439" );
```

See also

[“Synchronization stream parameters” on page ??](#).

upload_ok synchronization parameter

Function

Reports the status of MobiLink uploads. The MobiLink synchronization server provides this information to the client.

The parameter is read-only.

Access methods

boolean **getUploadOK()**

void **setUploadOK(** boolean *upload_ok* **)**

Usage

After synchronization, the **upload_ok** parameter holds **true** if the upload was successful, and **false** otherwise.

Access the parameter as follows:

```
UlsynchOptions opts = new UlsynchOptions;  
// set options here  
conn.synchronize( opts );  
returncode = opts.getUploadOK();
```

upload_only synchronization parameter

Function	Indicates that there should be no downloads in the current synchronization, which can save communication time, especially over slow communication links. When set to true, the client waits for the upload acknowledgement from the MobiLink synchronization server, after which it terminates the synchronization session successfully.
Access methods	boolean getUploadOnly() void setUploadOnly(boolean <i>upload_only</i>)
Default	The parameter is an optional Boolean value, and by default is false.
Usage	Set the parameter to true as follows: <pre>UlsynchOptions opts = new UlsynchOptions; opts.setUploadOnly(true);</pre>
See also	“Synchronizing high-priority changes” on page ?? “download_only synchronization parameter” on page 73

user_data synchronization parameter

Function	Make application-specific information available to the synchronization observer.
Access methods	java.lang.Object getUserData() void setUserData(java.lang.Object <i>user_data</i>)
Usage	When implementing the synchronization observer interface UlsynchObserver , you may can make application-specific information to the synchronization observer class by providing an object in the setUserData method.
See also	“observer synchronization parameter” on page 75 “Monitoring and canceling synchronization” on page ??

user_name synchronization parameter

Function	A string specifying the user name that uniquely identifies the MobiLink client to the MobiLink synchronization server. MobiLink uses this value to determine the download content, to record the synchronization state, and to recover from interruptions during synchronization.
Access methods	<code>java.lang.String getUserName()</code> <code>void setUserName(java.lang.String <i>user_name</i>)</code>
Default	The parameter is required, and is a string.
Usage	Set the parameter as follows: <pre>UlsynchOptions synch_options = new UlsynchOptions(); synch_opts.setUserName("mluser");</pre>
See also	“Authenticating MobiLink Users” [<i>MobiLink Synchronization User’s Guide</i> , page 103]. “The MobiLink user” [<i>MobiLink Synchronization User’s Guide</i> , page 20].

version synchronization parameter

Function	Each synchronization script in the consolidated database is marked with a version string. For example, there may be two different download_cursor scripts, identified by different version strings. The version string allows an UltraLite application to choose from a set of synchronization scripts.
Access methods	<code>java.lang.String getScriptVersion()</code> <code>void setScriptVersion(java.lang.String <i>version</i>)</code>
Default	The parameter is a string, and by default is the MobiLink default version string.
Usage	Set the parameter as follows: <pre>UlsynchOptions synch_options = new UlsynchOptions(); synch_opts.setVersion("default");</pre>
See also	“Script versions” [<i>MobiLink Synchronization User’s Guide</i> , page 49].

Index

A

absolute method	
UltraLite Java JDBC support	58
AES encryption algorithm	
UltraLite databases	41
afterLast method	
UltraLite Java JDBC support	58
applets	
running the UltraLite Java sample	21
UltraLite	33
applications	
building	15
deploying	36
writing in Java	6, 20

B

beforeFirst method	
UltraLite Java JDBC support	58
benefits	
UltraLite static Java API	4
building	
Java applications	15
sample application	22

C

cache_size persistent storage parameter	41
case sensitivity	
UltraLite user authentication	38
certificate option	
MobiLink synchronization server -x	48
certificate_password option	
MobiLink synchronization server -x	48
changeEncryptionKey method	43
JdbcDatabase class	43, 64
checkpoint_store synchronization	
parameter	
MobiLink synchronization	73
Class.forName method	27
ClassNotFoundException	27
close method	

JdbcDatabase class	64
compiling	
UltraLite Java	35
connect method	
JdbcDatabase class	59, 64
connecting	
multiple UltraLite Java databases	30
Properties object and UltraLite Java	29
UltraLite databases	38
UltraLite Java databases	26, 27
Connection object	27
conventions	
documentation	viii
creating	
UltraLite Java databases	66

D

database files	
changing the encryption key	43
defragmenting UltraLite databases	44
encrypting	42
obfuscating	41
setting the file name	41
databases	
connections from UltraLite Java	26
generating UltraLite Java	34
multiple UltraLite Java	30
UltraLite Java	30
definitions	
persistent storage parameters	41
defragmenting	
UltraLite databases	44
deploying	
UltraLite Java applications	36
disable_concurrency synchronization	
parameter	
MobiLink synchronization	73
disableUserAuthentication method	
JdbcSupport class	67
documentation	
conventions	viii
SQL Anywhere Studio	vi

download-only synchronization		about (static Java API)	74
getDownloadOnly method (static Java API)	73	getAuthStatus method	
getNewPassword method (static Java API)	74	about (static Java API)	71
setDownloadOnly method (static Java API)	73	getAuthValue method	
Driver class	27	about (static Java API)	72
DriverManager class	27	getDefragIterator method	
DriverManager.getConnection() method	27	JdbcConnection class	60
drop method		getDownloadOnly method	
JdbcDatabase class	65	about (static Java API)	73
E		getDriver method	27
enableUserAuthentication method		getLastDownloadTimeDate method	
JdbcSupport class	67	JdbcConnection class	60
encryption		getLastDownloadTimeLong method	
changing UltraLite encryption keys	43	JdbcConnection class	61
UltraLite databases	41, 42	getLastIdentity method	
encryption keys		JdbcConnection class	61
guidelines	42	getNewPassword method	
error handling		about (static Java API)	74
UltraLite applications	26	getPassword method	
UltraLite JDBC	27	about (static Java API)	75
F		getScriptVersion method	
feedback		about (static Java API)	85
documentation	xii	getStream method	
providing	xii	about (static Java API)	80
file_name persistent storage parameter	41	getUploadOK method	
first method		about (static Java API)	83
UltraLite Java JDBC support	58	getUploadOnly method	
first time		about (static Java API)	84
synchronization	49	getUserName method	
G		about (static Java API)	85
generated database class		global autoincrement	
UltraLite Java databases	66	UltraLite Java getLastIdentity method	61
generating		UltraLite Java globalAutoincUsage method	61
database	34	UltraLite Java setDatabaseID method	62
generator		global database identifier	
about	34	UltraLite Java	62
getAuthParms method		globalAutoincUsage method	
about (static Java API)	71	JdbcConnection class	61
getAuthParmsNumber method		grant method	
		JdbcDatabase class	62, 65
		I	
		icons	
		used in manuals	x

- ignored rows
 - synchronization 74
- ignored_rows synchronization parameter
 - MobiLink synchronization 74
- isAfterLast method
 - UltraLite Java JDBC support 58
- isBeforeFirst method
 - UltraLite Java JDBC support 58
- isFirst method
 - UltraLite Java JDBC support 58
- isLast method
 - UltraLite Java JDBC support 58
- J**
- Java
 - sample program 6
 - UltraLite limitations 58
 - UltraLite tutorial 6
- Java API
 - UltraLite 4
- Java applets
 - UltraLite 33
- java_certicom_tls stream
 - MobiLink synchronization server 48
- java_rsa_tls stream
 - MobiLink synchronization server 48
- JDBC
 - about 6
 - database parameter in UltraLite URL 28
 - loading drivers 27
 - registering drivers 27
 - UltraLite Java SQL statements 32
 - UltraLite limitations 58
 - URLs 28
- JDBC drivers
 - loading multiple drivers 27
 - loading UltraLite 27
 - registering UltraLite 27
 - UltraLite 27
- JdbcConnection class
 - about 59
 - getDefragIterator method 60
 - getLastIdentity method 61
 - globalAutoincUsage method 61
 - setDatabaseID method 62
 - startSynchronizationDelete method 63
 - stopSynchronizationDelete method 63
 - synchronize method 63
- JdbcConnection.synchronize method
 - about 16, 47
- JdbcDatabase class
 - about 26, 64, 66
 - close method 64
 - connect method 26, 59, 64
 - drop method 65
 - grant method 62, 65
 - revoke method 62, 65
- JdbcDefragIterator class
 - about 66
 - ulStoreDefragStep method 66
- JdbcSupport class
 - about 67
 - disableUserAuthentication method 67
 - enableUserAuthentication method 67
- JSynchProgressViewer class
 - about 53
- K**
- key property
 - UltraLite Java databases 29
- L**
- last method
 - UltraLite Java JDBC support 58
- limitations
 - JDBC UltraLite 59
- loading
 - JDBC driver 27
- M**
- monitoring synchronization
 - setObserver method (static Java API) 75
- multi-threaded applications
 - UltraLite applications 56
- N**
- new_password synchronization
 - parameter
 - about 74
- newsgroups
 - technical support xii

O

obfuscating	
UltraLite databases	41
UltraLite Java databases	66
obfuscation	
UltraLite databases	41
observer	
synchronization example	53

P

Palm Computing Platform	
user authentication	39
passwords	
MobiLink synchronization	74
Palm Computing Platform	39
UltraLite case sensitivity	38
UltraLite databases	38
UltraLite Java	29
persist property	
UltraLite Java databases	29
persistent storage	
parameters	41
UltraLite databases	26, 30
persistfile property	
UltraLite Java databases	29
previous method	
UltraLite Java JDBC support	58
progress viewer	
synchronization	53
projects	
Java	34
Properties object	
UltraLite Java connections	29, 30
publications	
setSynchPublication method (static Java API)	77

R

registering	
JDBC driver	27
relative method	
UltraLite Java JDBC support	58
revokemethod	
JdbcDatabase class	62, 65
running	
sample application	23

S

sample application	
building UltraLite Java	22
running UltraLite Java	23
UltraLite Java	21–23
script versions	
getScriptVersion method (static Java API)	85
setScriptVersion method (static Java API)	85
security	
changing the encryption key	43
database encryption	42
database obfuscation	41
UltraLite Java transport-layer security	48
setAuthParms method	
about (static Java API)	71
setAuthParmsNumber method	
about (static Java API)	74
setDatabaseID method	
JdbcConnection class	62
setDefaultObfuscation method	
JdbcDatabase class	66
UIDatabase class	42
setDownloadOnly method	
about (static Java API)	73
setNewPassword method	
about (static Java API)	74
setObserver method	
about (static Java API)	75
setPassword method	
about (static Java API)	75
setPing method	
about (static Java API)	76
setScriptVersion method	
about (static Java API)	85
setStream method	
about (static Java API)	80
setStreamParms method	
about (static Java API)	83
setSynchPublication method	
about (static Java API)	77
setting	
persistent storage parameters	41
setUploadOnly method	
about (static Java API)	84

setUserData method		getAuthParmsNumber method (static	
about (static Java API)	84	Java API)	74
setUserName method		getAuthStatus method (static Java	
about (static Java API)	85	API)	71
SQL Anywhere Studio		getAuthValue method (static Java API)	
documentation	vi	72	
SQL statements		getDownloadOnly method (static Java	
UltraLite Java	32	API)	73
SQLException		getNewPassword method (static Java	
UltraLite applications	26	API)	74
startSynchronizationDelete method		getPassword method (static Java API)	
JdbcConnection class	63	75	
static Java API		getScriptVersion method (static Java	
UltraLite benefits	4	API)	85
stopSynchronizationDelete method		getStream method (static Java API)	80
JdbcConnection class	63	getUploadOK method (static Java	
storage parameters	41	API)	83
strong encryption		getUploadOnly method (static Java	
UltraLite databases	41	API)	84
support		getUserName method (static Java API)	
newsgroups	xii	85	
Sybase Central		new_password	74
adding SQL statements to an UltraLite		setAuthParms method (static Java	
project	8	API)	71
SynchProgressViewer class		setAuthParmsNumber method (static	
about	53	Java API)	74
synchronization		setDownloadOnly method (static Java	
about	45	API)	73
adding to UltraLite applications	45	setNewPassword method (static Java	
applets	33	API)	74
canceling	50	setObserver method (static Java API)	
checkpoint_store	73	75	
commit before	49	setPassword method (static Java API)	
disable_concurrency	73	75	
ignored rows	74	setPing method (static Java API)	76
initial copy	49	setScriptVersion method (static Java	
invoking	47	API)	85
Java application	16	setStream method (static Java API)	80
Java applications	47	setStreamParms method (static Java	
JdbcConnection.synchronize method		API)	83
16, 47		setSynchPublication method (static	
monitoring	50	Java API)	77
progress viewer	53	setUploadOnly method (static Java	
UltraLite Java	47	API)	84
synchronization parameters		setUserData method (static Java API)	
getAuthParms method (static Java		84	
API)	71	setUserName method (static Java API)	

85
 synchronization streams
 getStream method (static Java API) 80
 setStream method (static Java API) 80
 setStreamParms method (static Java API) 83
 UIHTTPSSStream 46
 UIHTTPSSStream (static Java API) 80
 UIHTTPStream 46
 UIHTTPStream (static Java API) 80
 UISecureRSASocketStream 46
 UISecureSocketStream 46, 48
 UISecureSocketStream (static Java API) 80
 UISocketStream 46
 UISocketStream (static Java API) 80
 synchronize method
 JdbcConnection class 63
 JdbcConnection object 47

T

technical support
 newsgroups xii
 threads
 Java synchronization 53
 synchronization monitoring 53
 UltraLite applications 56
 UltraLite Java 56
 tips
 UltraLite development 49
 transient databases
 UltraLite 26, 30
 transport-layer security
 java_certicom_tls stream 48
 java_rsa_tls stream 48
 UltraLite Java applications 48
 UltraLite Java clients 46
 troubleshooting
 commit all changes before synchronizing 49
 getUploadOK method (static Java API) 83
 setPing method (static Java API) 76
 UltraLite development 49
 tutorials
 UltraLite Java 6

U

UL_STORE_PARMS macro
 using 41
 UL_SYNC_ALL macro
 publication mask 77
 UL_SYNC_ALL_PUBS macro
 publication mask 77
 ULChangeEncryptionKey function
 using 43
 UIDatabase class
 obfuscating databases 42
 UIDefnUL_AUTH_STATUS_EXPIRED
 auth_status value
 about 71
 UIDefnUL_AUTH_STATUS_IN_USE
 auth_status value
 about 71
 UIDefnUL_AUTH_STATUS_INVALID
 auth_status value
 about 71
 UIDefnUL_AUTH_STATUS_-
 UNKNOWN auth_status value
 about 71
 UIDefnUL_AUTH_STATUS_VALID
 auth_status value
 about 71
 UIDefnUL_AUTH_STATUS_VALID_-
 BUT_EXPIRES_SOON
 auth_status value
 about 71
 ULEnableUserAuthentication function
 about 39
 using 38
 ulgen utility
 about 34
 UIHTTPSSStream object
 Java synchronization stream 46
 Java synchronization stream (static Java API) 80
 UIHTTPStream object
 Java synchronization stream 46
 Java synchronization stream (static Java API) 80
 ULSecureCerticomTLSStream function
 security (static Java API) 78
 ULSecureRSASocketStream object

about	48	URL	
Java synchronization stream	46	UltraLite Java database	27, 28
ULSecureRSATLSStream function		user authentication	
security (static Java API)	78	embedded SQL UltraLite applications	
ULSecureSocketStream object		39	
about	48	getAuthStatus method (static Java	
Java synchronization stream	46	API)	71
Java synchronization stream (static		getAuthValue method (static Java API)	
Java API)	80	72	
UICollection object		getPassword method (static Java API)	
Java synchronization stream	46	75	
Java synchronization stream (static		getUserName method (static Java API)	
Java API)	80	85	
ulStoreDefragStep method		MobiLink and UltraLite	40
JdbcDefragIterator class	66	setNewPassword method (static Java	
UISynchObserver interface		API)	74
implementing	50	setPassword method (static Java API)	
UISynchOptions object		75	
members (static Java API)	70	setUserName method (static Java API)	
UltraLite		85	
JDBC driver	27	UltraLite case sensitivity	38
UltraLite databases		UltraLite databases	38
encrypting	41	user IDs	
multiple Java	30	Palm Computing Platform	39
user IDs	38	UltraLite case sensitivity	38
UltraLite Java		UltraLite databases	38
threads	56	UltraLite Java	29
UltraLite passwords			
about	38	W	
maximum length	38	wizards	
UltraLite project creation wizard		UltraLite project creation	8
using	8	UltraLite statement creation	8
UltraLite statement creation wizard		writing applications in Java	6, 20
using	8		
UltraLite user IDs			
about	38		
limit	38		
maximum length	38		
unsupported features			
UltraLite limitations	58		
unsupported JDBC methods			
UltraLite limitations	59		
upload only synchronization			
getUploadOnly method (static Java			
API)	84		
setUploadOnly method (static Java			
API)	84		