UltraLite®J

Contents

About this book

This book describes UltraLiteJ. With UltraLiteJ, you can develop and deploy database applications in environments that support Java. UltraLiteJ supports BlackBerry smartphones, Java ME, and Java SE environments. UltraLiteJ is based on the iAnywhere UltraLite database product.

About the SQL Anywhere documentation

The complete SQL Anywhere documentation is available in four formats:

- **DocCommentXchange** DocCommentXchange is a community for accessing and discussing SQL Anywhere documentation on the web.

  To access the documentation, go to http://dcx.sybase.com.

- **HTML Help** On Windows platforms, the HTML Help contains the complete SQL Anywhere documentation, including the books and the context-sensitive help for SQL Anywhere tools.

  To access the documentation, choose **Start** » **Programs** » **SQL Anywhere 12** » **Documentation** » **HTML Help (English)**.

- **Eclipse** On Unix platforms, the complete Help is provided in Eclipse format. To access the documentation, run *sadoc* from the *bin32* or *bin64* directory of your SQL Anywhere installation.

- **PDF** The complete set of SQL Anywhere books is provided as a set of Portable Document Format (PDF) files. You must have a PDF reader to view information.

  To access the PDF documentation on Windows operating systems, choose **Start** » **Programs** » **SQL Anywhere 12** » **Documentation** » **PDF (English)**.

  To access the PDF documentation on Unix operating systems, use a web browser to open */documentation/en/pdf/index.html* under the SQL Anywhere installation directory.

Documentation conventions

This section lists the conventions used in this documentation.

Operating systems

SQL Anywhere runs on a variety of platforms. Typically, the behavior of the software is the same on all platforms, but there are variations or limitations. These are commonly based on the underlying operating system (Windows, Unix), and seldom on the particular variant (IBM AIX, Windows Mobile) or version.

To simplify references to operating systems, the documentation groups the supported operating systems as follows:

- **Windows** The Microsoft Windows family includes platforms that are used primarily on server, desktop, and laptop computers, as well as platforms used on mobile devices. Unless otherwise specified, when the documentation refers to Windows, it refers to all supported Windows-based platforms, including Windows Mobile.

  Windows Mobile is based on the Windows CE operating system, which is also used to build a variety of platforms other than Windows Mobile. Unless otherwise specified, when the documentation refers to Windows Mobile, it refers to all supported platforms built using Windows CE.

- **Unix** Unless otherwise specified, when the documentation refers to Unix, it refers to all supported Unix-based platforms, including Linux and Mac OS X.

For the complete list of platforms supported by SQL Anywhere, see “Supported platforms” [*SQL Anywhere 12 - Introduction*].

Directory and file names

Usually references to directory and file names are similar on all supported platforms, with simple transformations between the various forms. In these cases, Windows conventions are used. Where the details are more complex, the documentation shows all relevant forms.

These are the conventions used to simplify the documentation of directory and file names:

- **Uppercase and lowercase directory names** On Windows and Unix, directory and file names may contain uppercase and lowercase letters. When directories and files are created, the file system preserves letter case.

  On Windows, references to directories and files are *not* case sensitive. Mixed case directory and file names are common, but it is common to refer to them using all lowercase letters. The SQL Anywhere installation contains directories such as *Bin32* and *Documentation*.

  On Unix, references to directories and files *are* case sensitive. Mixed case directory and file names are not common. Most use all lowercase letters. The SQL Anywhere installation contains directories such as *bin32* and *documentation*.

  The documentation uses the Windows forms of directory names. You can usually convert a mixed case directory name to lowercase for the equivalent directory name on Unix.

- **Slashes separating directory and file names** The documentation uses backslashes as the directory separator. For example, the PDF form of the documentation is found in *install-dir \Documentation\en\PDF* (Windows form).

  On Unix, replace the backslash with the forward slash. The PDF documentation is found in *install-dir/ documentation/en/pdf*.

- **Executable files** The documentation shows executable file names using Windows conventions, with a suffix such as *.exe* or *.bat*. On Unix, executable file names have no suffix.

  For example, on Windows, the network database server is *dbsrv12.exe*. On Unix, it is *dbsrv12*.

- ***install-dir*** During the installation process, you choose where to install SQL Anywhere. The environment variable SQLANY12 is created and refers to this location. The documentation refers to this location as *install-dir*.

  For example, the documentation may refer to the file *install-dir/readme.txt*. On Windows, this is equivalent to *%SQLANY12%\readme.txt*. On Unix, this is equivalent to *$SQLANY12/readme.txt* or *${SQLANY12}/readme.txt*.

  For more information about the default location of *install-dir*, see "SQLANY12 environment variable" [*SQL Anywhere Server - Database Administration*].

- ***samples-dir*** During the installation process, you choose where to install the samples included with SQL Anywhere. The environment variable SQLANYSAMP12 is created and refers to this location. The documentation refers to this location as *samples-dir*.

  To open a Windows Explorer window in *samples-dir*, choose **Start** » **Programs** » **SQL Anywhere 12** » **Sample Applications And Projects**.

  For more information about the default location of *samples-dir*, see "SQLANYSAMP12 environment variable" [*SQL Anywhere Server - Database Administration*].

Command prompts and command shell syntax

Most operating systems provide one or more methods of entering commands and parameters using a command shell or command prompt. Windows command prompts include Command Prompt (DOS prompt) and 4NT. Unix command shells include Korn shell and bash. Each shell has features that extend its capabilities beyond simple commands. These features are driven by special characters. The special characters and features vary from one shell to another. Incorrect use of these special characters often results in syntax errors or unexpected behavior.

The documentation provides command line examples in a generic form. If these examples contain characters that the shell considers special, the command may require modification for the specific shell. The modifications are beyond the scope of this documentation, but generally, use quotes around the parameters containing those characters or use an escape character before the special characters.

These are some examples of command line syntax that may vary between platforms:

- **Parentheses and curly braces** Some command line options require a parameter that accepts detailed value specifications in a list. The list is usually enclosed with parentheses or curly braces. The documentation uses parentheses. For example:

  ```
  -x tcpip(host=127.0.0.1)
  ```

  Where parentheses cause syntax problems, substitute curly braces:

  ```
  -x tcpip{host=127.0.0.1}
  ```

  If both forms result in syntax problems, the entire parameter should be enclosed in quotes as required by the shell:

  ```
  -x "tcpip(host=127.0.0.1)"
  ```

- **Semicolons** On Unix, semicolons should be enclosed in quotes.
- **Quotes** If you must specify quotes in a parameter value, the quotes may conflict with the traditional use of quotes to enclose the parameter. For example, to specify an encryption key whose value contains double-quotes, you might have to enclose the key in quotes and then escape the embedded quote:

  ```
  -ek "my \"secret\" key"
  ```

  In many shells, the value of the key would be `my "secret" key`.
- **Environment variables** The documentation refers to setting environment variables. In Windows shells, environment variables are specified using the syntax *%ENVVAR%*. In Unix shells, environment variables are specified using the syntax *$ENVVAR* or *${ENVVAR}*.

Contacting the documentation team

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

You can leave comments directly on help topics using DocCommentXchange. DocCommentXchange (DCX) is a community for accessing and discussing SQL Anywhere documentation. Use DocCommentXchange to:

- View documentation
- Check for clarifications users have made to sections of documentation
- Provide suggestions and corrections to improve documentation for all users in future releases

Go to http://dcx.sybase.com.

Finding out more and requesting technical support

Newsgroups

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

When you write to one of these newsgroups, always provide details about your problem, including the build number of your version of SQL Anywhere. You can find this information by running the following command: **dbeng12 -v**.

The newsgroups are located on the *forums.sybase.com* news server.

The newsgroups include the following:

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

For web development issues, see http://groups.google.com/group/sql-anywhere-web-development.

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

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

Developer Centers

The **SQL Anywhere Tech Corner** gives developers easy access to product technical documentation. You can browse technical white papers, FAQs, tech notes, downloads, techcasts and more to find answers to your questions as well as solutions to many common issues. See http://www.sybase.com/developer/library/sql-anywhere-techcorner.

The following table contains a list of the developer centers available for use on the SQL Anywhere Tech Corner:

Name	URL	Description
SQL Anywhere .NET Developer Center	www.sybase.com/developer/library/sql-anywhere-techcorner/microsoft-net	Get started and get answers to specific questions regarding SQL Anywhere and .NET development.
PHP Developer Center	www.sybase.com/developer/library/sql-anywhere-techcorner/php	An introduction to using the PHP (PHP Hypertext Preprocessor) scripting language to query your SQL Anywhere database.

Name	URL	Description
SQL Anywhere Windows Mobile Developer Center	www.sybase.com/developer/library/sql-anywhere-techcorner/windows-mobile	Get started and get answers to specific questions regarding SQL Anywhere and Windows Mobile development.

Using UltraLiteJ

Introduction to UltraLiteJ

UltraLiteJ is a Java-based subset of UltraLite that is designed for Java SE platforms and BlackBerry smartphones. With UltraLiteJ, you can bring transactions, primary and foreign keys, indexes, and other features of relational databases to BlackBerry smartphones. UltraLiteJ provides built-in change tracking and synchronization functions that allow you to build data synchronization into your BlackBerry applications. When used with a MobiLink server, you can use UltraLiteJ to extend Oracle, SQL Server, DB2, Sybase Adaptive Server Enterprise, and SQL Anywhere databases to mobile devices.

UltraLiteJ includes many characteristics typical of relational databases, including storing data in tables, using primary keys, and a transactional database store.

To redistribute UltraLiteJ, you can add Java Archive (JAR) files to your applications. UltraLiteJ is supported on Java SE 1.6 or later and BlackBerry smartphones running OS 4.2 and later.

Note
UltraLiteJ databases are not interchangeable with UltraLite databases. For more information on UltraLite databases, see "Introducing UltraLite" [*UltraLite - Database Management and Reference*].

UltraLiteJ features

Database store

UltraLiteJ supports databases in memory (non-persistent) and in device storage (persistent), including the BlackBerry store (internal flash) and SD cards.

Transactions

Transactions are the set of operations between commits or rollbacks. For persistent database stores, a commit makes permanent any changes since the last commit or rollback. A rollback returns the database to the state it was in when the last commit was invoked.

Each transaction and row-level operation in UltraLiteJ is atomic. An insert involving multiple columns either inserts data to all the columns or to none of the columns.

Concurrency and locking

UltraLiteJ uses isolation level zero (read uncommitted) to provide the maximum level of concurrency.

- **Locking** Two different connections cannot modify the same row at the same time. If two connections attempt to operate on the same row, the second connection receives an error and becomes prohibited from modifying the row until the first connection commits or rolls back its current transaction.

- **Visibility** One connection's operation on the database is immediately visible to other connections.

Cache management

Persistent stores are page based; UltraLiteJ operates on pages in cache. A working set of pages is maintained in cache and is managed using a first-in, first-out (FIFO) scheme. Pages that are currently in use are locked in cache to avoid being swapped out.

You can configure the size of the UltraLiteJ cache.

UltraLiteJ can use the lazy loading of indexes and row pages to improve the startup of a persistent database. Indexes and row pages are only loaded when first accessed by an application.

For small databases, UltraLiteJ keeps copies of all row and index pages in memory.

For larger databases, the database can be configured each time it is opened to limit the number of rows and index pages that resident in memory at one time.

Encryption

Encryption is set using the setEncryption method of a Configuration object, which takes an EncryptionControl to encrypt and decrypt pages. You must supply your own encryption control.

Built-in change tracking

As a MobiLink synchronization client, UltraLiteJ has a built-in transaction log-based change tracking system so that database changes can be synchronized.

HTTP and HTTPS communication

Data synchronization can be performed using HTTP or HTTPS network protocols. HTTPS synchronization provides secure encryption to the MobiLink server.

Synchronization publications

UltraLiteJ provides a publication model that lets you synchronize selected tables and selected rows from those tables from your database to make efficient use of network resources.

Character sets and collations

UltraLiteJ uses Unicode (encoded as UTF-8 in the database). The collation is the default sort order for Java and is equivalent to the UTF8BIN collation supported by SQL Anywhere. During synchronization with a MobiLink server, UltraLiteJ notifies MobiLink that it uses the UTF8 character set and collation.

Blackberry support

Blackberry internal flash and SD cards are supported by UltraLiteJ.

UltraLiteJ limitations

General limitations

The following is a list of general restrictions that apply to UltraLiteJ databases:

Feature	Limitation
Blob size	Up to 2^{24} bytes.
Data types	See "Domain interface" on page 151.
Database access	Only one application can access a database at a time. Simultaneous access is not supported.
Database page size	Minimum 256 bytes, up to 16 KB.
Database pages	Up to 2^{16}.
Row size	Row contents (after possible compression) must not exceed the database page size.
Tables per database	Up to 32 K.
Number of publications	Up to 63.
UltraLite compatibility	UltraLiteJ databases are not interchangeable with UltraLite databases. An UltraLiteJ database can be converted into an UltraLite database and vice versa using the respective load and unload utilities.

UltraLiteJ database stores

Supported database stores

The following is a list of database configurations supported by UltraLiteJ:

Database store type	Configuration	Platform
File system	ConfigFile	Java SE
RIM object store	ConfigObjectStore	BlackBerry
Record store	ConfigRecordStore	Java ME
BlackBerry file system	ConfigFileME	BlackBerry

BlackBerry object store limitations

On a BlackBerry smartphone, the size of the database store is limited by the number of object handles available. The number of available object handles is determined by the size of the flash memory:

Flash memory	Persistent handles	Handles
8 MB	12000	24000
16 MB	27000	56000
32 MB	65000	132000

UltraLiteJ requires one or more object handles, depending on the handle type, to store a database value in memory. For example, a table row with ten columns and two indexes requires a minimum of twelve object handles.

To permit larger database stores, UltraLiteJ allows users to limit the number of rows kept in memory. Stored database rows are combined on a database page. UltraLiteJ requires only one persistent object handle for each database page.

Persistent store configurations and recovery

When creating your database, use the Configuration object to choose one of the following forms of persistence for your application.

- **Non-persistence** Creating a NonPersist object configures a database store that only exists in memory. The database is created at startup, used while the application is running, then discarded when the application closes. When the application closes, all data contained in the non-persistent store is deleted.
- **Shadow paging persistence** Shadow paging is the strongest form of persistence. It can be enabled using the setShadowPaging method of a persistent configuration object. If it is enabled when the database is created, it provides database recovery to the state at the last commit or rollback, even if the application terminates unexpectedly.
- **Write-at-end persistence** When write-at-end persistence is enabled using the setWriteAtEnd method of a persistent configuration object, data is only written to the database when the connection is released. While this form of persistence improves the overall speed of transactions, data could be lost if the application terminates abnormally.
- **Simple paging persistence** When shadow paging is not enabled, a simple paging implementation of persistence is used. All data is written to the same page from which data is read. The database is considered corrupt from the time updating starts to the time before updating completes. A corrupt database can be detected during startup, but it is not recoverable. In comparison to shadow paging, simple paging uses significantly less memory and improves performance.

Note
Write-at-end and simple paging persistence should only be used in applications when the loss of data can be tolerated, the database can be easily recreated, or the database is not updated.

Data synchronization

UltraLiteJ is able to synchronize data with MobiLink.

UltraLiteJ supports:

- MobiLink user authentication
- MobiLink user authentication scripts
- Publication-based synchronization
- HTTP and HTTPS network protocols
- Upload-only, download-only, ping-only, and full upload/download modes
- Synchronization observer API

In a BlackBerry environment, data is always encrypted between the device and the BlackBerry Enterprise Server (BES). HTTPS is used when encryption is required between the BES and the MobiLink server.

Concurrent synchronization

Normally only one thread is allowed in the UltraLiteJ runtime at a time. An exception to this rule occurs during synchronization. While one connection is performing a synchronization operation, other connections can access the UltraLiteJ runtime, however no other thread can call the synchronize method for the database being synchronized while a synchronize operation is taking place.

A connection can access downloaded rows during the sync (that is, before they are committed) that may later vanish if the sync fails. If, during a synchronization, a connection modifies a row that the synchronization then attempts to change, the sync fails. During a synchronization, if a connection attempts to modify a row that the synchronization has changed, the attempt to modify fails.

Developing UltraLiteJ applications

This section introduces the UltraLiteJ Application Programming Interface (API).

Introduction to UltraLiteJ development

UltraLiteJ provides basic database functionality to your Java applications. It is designed to work specifically with BlackBerry smartphones but is fully compatible with Java ME and Java SE environments. The UltraLiteJ API contains all the methods required to connect to an UltraLiteJ database, perform schema operations, and maintain data using SQL statements. Advanced operations, such as data encryption and synchronization, are also supported.

The API for each supported platform is stored in the *UltraLiteJ12.jar* file in your UltraLiteJ directory, which is typically located in the *UltraLite\UltraLiteJ* folder of your SQL Anywhere installation.

Basic steps for creating an UltraLiteJ application

When creating an UltraLiteJ application, you typically complete the following tasks:

1. Create a new Configuration object.

   Configuration objects define where an UltraLiteJ database is located or where it should be created. They also specify the username and password required to connect to the database. Variations of a Configuration object are available for different devices and for non-persistent database stores. See "Configuration interface" on page 115.

2. Create a new Connection object.

   Connection objects connect to an UltraLiteJ database using the specifications defined in the Configuration object. See "Connection interface" on page 117.

3. Create or modify the database schema using SQL statements, and use the PreparedStatement interface to query the database.

   You can use SQL statements to create or update tables, indexes, foreign keys, and publications for your database. See "Supported SQL statements" on page 14.

   PreparedStatement objects query the database associated with the Connection object. They accept supported SQL statements, which are passed as strings. You can use PreparedStatement objects to update the contents of the database. See "PreparedStatement interface" on page 182.

4. Generate ResultSet objects.

   ResultSet objects are created when the Connection object executes a PreparedStatement containing a SQL SELECT statement. You can use ResultSet objects to obtain rows of query results to view the table contents of the database. See "ResultSet interface" on page 198.

Setting up an UltraLiteJ application

When setting up an UltraLiteJ application in your preferred Java IDE, make sure that your project is correctly configured to use the *UltraLiteJ.jar12* resource file, which is located in your UltraLiteJ directory.

Use the following statement to import the UltraLiteJ package into your Java file:

```
import com.ianywhere.ultralitej12.*;
```

All coding samples and tutorials contained in this document assume that the above statement is specified and that you are familiar with developing Java applications in your preferred IDE.

Accessing an UltraLiteJ database store

Applications must connect to an UltraLiteJ database before operations can be performed on the data. This section explains how to create or connect to a database with a specified password.

Implementations of the Configuration object

A Configuration is used to create and connect to a database. There are several different implementations of a Configuration provided in the API. A unique implementation exists for every type of database store supported by UltraLiteJ. Each implementation provides a set of methods used to configure the database store.

- **RIM object stores** Supported with a ConfigObjectStore. See "ConfigObjectStore interface (Java ME BlackBerry only)" on page 103.
- **Record stores** Supported with a ConfigRecordStore. See "ConfigRecordStore interface (Java ME only)" on page 113.
- **File system stores** Supported with a ConfigFile. See "ConfigFile interface" on page 99.
- **Non-persistent stores** Supported with a ConfigNonPersistent. See "ConfigNonPersistent interface" on page 102.
- **Internal flash and SD card stores** Supported with a ConfigFileME. See "ConfigFileME interface (BlackBerry only)" on page 101.

Properties of the Connection object

- **Transactions** Transactions must be committed to the database using the commit method of the Connection. They can be rolled back using the rollback method.
- **Prepared SQL statements** Methods are provided by the PreparedStatement interface to handle SQL statements. A PreparedStatement can be created using the prepareStatement method of the Connection.
- **Synchronization** A set of objects governing MobiLink synchronization is accessed from the Connection.

Creating a new UltraLiteJ database

You can not create a new UltraLiteJ database directly using Sybase Central or UltraLite command line utilities. An UltraLite database can be converted into an UltraLiteJ database using the ulunload and uljload utilities. An UltraLiteJ database created using uljload or a Java SE application can be deployed to a BlackBerry device by copying the database to an SD card or transferring it from MobiLink using the file transfer mechanism.

To create a database

1. Create a new Configuration that references the database name and is appropriate for your platform.

   In the following examples, **config** is the name of the Configuration object and *DBname.ulj* is the name of the new database.

   For Java ME BlackBerry devices:

   ```
   ConfigObjectStore config =
       DatabaseManager.createConfigurationObjectStore("DBname.ulj");

   ConfigFileME config =
   DatabaseManager.createConfigFileME( "file:///store/home/user/
   DBname.ulj" );

   ConfigFileME config =
   DatabaseManager.createConfigFileME( "file:///SDCard/DBname.ulj" );
   ```

   For all other Java ME devices:

   ```
   ConfigRecordStore config =
       DatabaseManager.createConfigurationRecordStore("DBname.ulj");
   ```

   For Java SE devices:

   ```
   ConfigFile config =
       DatabaseManager.createConfigurationFile("DBname.ulj");
   ```

   Alternatively, you can create a non-persistent database Configuration, which is supported by all platforms:

   ```
   ConfigNonPersistent config =
       DatabaseManager.createConfigurationNonPersistent("DBname.ulj");
   ```

2. Set a new database password using the setPassword method:

   ```
   config.setPassword("my_password");
   ```

3. Create a new Connection:

   ```
   Connection conn = DatabaseManager.createDatabase(config);
   ```

   The createDatabase method creates the database and returns a Connection to it. After this method is called, you can execute SQL statements to create the tables and indexes for your application but you can not change the name, password, or page size of the database.

Connecting to an existing database

An UltraLiteJ database must already exist on the client device before you can connect to it.

To connect to an existing database

1. Create a new Configuration that references the name of the database and is appropriate for your platform.

   In the following examples, **config** is the name of the Configuration object and *DBname.ulj* is the name of the database.

   For Java ME BlackBerry devices:

   ```
   ConfigObjectStore config =
       DatabaseManager.createConfigurationObjectStore("DBname.ulj");
   ```

   ```
   ConfigFileME config =
   DatabaseManager.createConfigFileME( "file:///store/home/user/
   DBname.ulj" );
   ```

   ```
   ConfigFileME config =
   DatabaseManager.createConfigFileME( "file:///SDCard/DBname.ulj" );
   ```

   For all other Java ME devices:

   ```
   ConfigRecordStore config =
       DatabaseManager.createConfigurationRecordStore("DBname.ulj");
   ```

   For Java SE devices:

   ```
   ConfigFile config =
       DatabaseManager.createConfigurationFile("DBname.ulj");
   ```

   Alternatively, you can connect to a non-persistent database Configuration, which is supported by all platforms:

   ```
   ConfigNonPersistent config =
       DatabaseManager.createConfigurationNonPersistent("DBname.ulj");
   ```

2. Specify the database password using the setPassword method:

   ```
   config.setPassword("my_password");
   ```

3. Create a new Connection:

   ```
   Connection conn = DatabaseManager.connect(config);
   ```

   The connect method finalizes the database connection process. If the database does not exist, an error is thrown.

Disconnecting from a database

A Connection object is disconnected from the database when the release method is called. The database is closed when all connections for a database have been released.

See "release method" on page 147.

See also

- “Example: Creating a database” on page 29
- “Connection interface” on page 117
- “DatabaseManager class” on page 141

Creating and updating a database schema

SQL statements are used to create and update a database schema. You can use these statements to create or update tables, indexes, foreign keys, and publications for your database. Use the prepareStatement and execute methods to apply the schema updates to your database.

The following example demonstrates the use of SQL statements for table creation:

```
static String stmt_1 = "CREATE TABLE Department("
    + "id int PRIMARY KEY, "
    + "name char(50) NOT NULL)";

static String stmt_2 = "CREATE TABLE Employee("
    + "id int PRIMARY KEY, "
    + "last_name char(50) NOT NULL, "
    + "first_name char(50) NOT NULL, "
    + "dept_id int NOT NULL, "
    + "NOT NULL FOREIGN KEY(dept_id) "
    + "REFERENCES Department(id))";

static String stmt_3 = "CREATE INDEX ON Employee(last_name, first_name)";

void createDb(Connection connection) throws ULjException {
    PreparedStatement ps;
    ps = connection.prepareStatement(stmt_1);
    ps.execute();
    ps.close();
    ps = connection.prepareStatement(stmt_2);
    ps.execute();
    ps.close();
    ps = connection.prepareStatement(stmt_3);
    ps.execute();
    ps.close();
}
```

In this example, the createDB method uses SQL statement strings to prepare and execute statements, which create two tables and an additional index on last_name and last_name.

See also

- “Supported SQL statements” on page 14
- “prepareStatement method” on page 129

Accessing and modifying data using SQL

Supported SQL statements

Some SQL statements that are supported by UltraLite are not supported by UltraLiteJ. The following is a complete list of SQL statements supported by UltraLiteJ:

SQL Statement	**Notes and Restrictions:**
ALTER PUBLICATION	See "ALTER PUBLICATION statement [UltraLite][UltraLiteJ]" [*UltraLite - Database Management and Reference*].
ALTER SYNCHRONIZATION PROFILE	See "ALTER SYNCHRONIZATION PROFILE statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
ALTER TABLE	See "ALTER TABLE statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*]. MAX HASH SIZE is not supported.
COMMIT	See "COMMIT statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
CREATE INDEX	See "CREATE INDEX statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*]. MAX HASH SIZE is not supported.
CREATE PUBLICATION	See "CREATE PUBLICATION statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
CREATE SYNCHRONIZATION PROFILE	See "CREATE SYNCHRONIZATION PROFILE statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
CREATE TABLE	See "CREATE TABLE statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*]. MAX HASH SIZE is not supported.
DELETE	See "DELETE statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
DROP INDEX	See "DROP INDEX statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
DROP PUBLICATION	See "DROP PUBLICATION statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].

SQL Statement	**Notes and Restrictions:**
DROP SYNCHRONIZATION PROFILE	See "DROP SYNCHRONIZATION PROFILE statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
DROP TABLE	See "DROP TABLE statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
INSERT	See "INSERT statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
ROLLBACK	See "ROLLBACK statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
SELECT	For a general description of the SELECT statement, see "SELECT statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*]. The following restrictions apply: ● The SQLCODE function is not supported. ● The ACOS, ASIN, ATAN, ATAN2, and POWER mathematical functions are not supported.
SET OPTION	See "SET OPTION statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
START SYNCHRONIZATION DELETE	See "START SYNCHRONIZATION DELETE statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
STOP SYNCHRONIZATION DELETE	See "STOP SYNCHRONIZATION DELETE statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
SYNCHRONIZE	See "SYNCHRONIZE statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
TRUNCATE TABLE	See "TRUNCATE TABLE statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*].
UPDATE	For a general description of the SELECT statement, see "UPDATE statement [UltraLite] [UltraLiteJ]" [*UltraLite - Database Management and Reference*]. The JOIN clause is not supported.

Data modification using INSERT, UPDATE, DELETE

You can perform SQL data modification using the execute method of a PreparedStatement. A PreparedStatement queries the database with a user-defined SQL statement.

When applying a SQL statement to a PreparedStatement, query parameters are indicated by the **?** character. For any INSERT, UPDATE or DELETE statement, each **?** parameter is referenced according to its ordinal position in the statement. For example, the first **?** is referenced as parameter one, and the second as parameter two.

To INSERT a row in a table

1. Prepare a new SQL statement as a String.

   ```
   String sql_string =
       "INSERT INTO Department(dept_no, name) VALUES( ?, ? )";
   ```

2. Pass the String to the PreparedStatement.

   ```
   PreparedStatement inserter =
       conn.prepareStatement(sql_string);
   ```

3. Pass input values to the PreparedStatement using the set method.

   This example sets 101 for the dept_no, referenced as parameter 1, and "Electronics" for the name, referenced as parameter 2.

   ```
   inserter.set(1, 101);
   inserter.set(2, "Electronics");
   ```

4. Execute the statement.

   ```
   inserter.execute();
   ```

5. Close the PreparedStatement to free resources.

   ```
   inserter.close()
   ```

6. Commit all changes to the database.

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

Steps(3) and (4) can be repeated as many times as needed.

Alternatively, when only a single INSERT is to be performed, the statement INSERT INTO Department(dept_no, name) VALUES(2, 'Electronics') could be PREPAREd, EXECUTEd and CLOSEd.

The statement could be created using Java String concatenation.

To UPDATE a row in a table

1. Prepare a new SQL statement as a String.

```
String sql_string =
    "UPDATE Department SET dept_no = ? WHERE dept_no = ?";
```

2. Pass the String to the PreparedStatement.

   ```
   PreparedStatement updater =
       conn.prepareStatement(sql_string);
   ```

3. Pass input values to the PreparedStatement using the set method.

   ```
   updater.set(1, 102);
   updater.set(2, 101);
   ```

   The example above is the equivalent of declaring the following SQL statement:

   ```
   UPDATE Department SET dept_no = 102 WHERE dept_no = 101
   ```

4. Execute the statement.

   ```
   updater.execute();
   ```

5. Close the PreparedStatement to free resources.

   ```
   updater.close()
   ```

6. Commit all changes to the database.

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

Steps(3) and (4) can be repeated as many times as needed.

Alternatively, when only a single INSERT is to be performed, the statement INSERT INTO Department(dept_no, name) VALUES(2, 'Electronics') could be PREPAREd, EXECUTEd and CLOSEd.

The statement could be created using Java String concatenation.

To DELETE a row in a table

1. Prepare a new SQL statement as a String.

   ```
   String sql_string =
       "DELETE FROM Department WHERE dept_no = ?";
   ```

2. Pass the String to the PreparedStatement.

   ```
   PreparedStatement deleter =
       conn.prepareStatement(sql_string);
   ```

3. Pass input values to the PreparedStatement using the set method.

   ```
   deleter.set(1, 102);
   ```

   The example above is the equivalent of declaring the following SQL statement:

   ```
   DELETE FROM Department WHERE dept_no = 102
   ```

4. Execute the statement.

   ```
   deleter.execute();
   ```

5. Close the PreparedStatement to free resources.

   ```
   deleter.close()
   ```

6. Commit all changes to the database.

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

Example

Prepared statements can be executed several times. The following example demonstrates multiple row insertion using host variables:

```
String stmt = "INSERT INTO Department(dept_no, name) VALUES (?,?)";
PreparedStatement inserter = conn.prepareStatement(stmt);

inserter.set(1, 101);
inserter.set(2, "Electronics");
inserter.execute();

inserter.set(1, 105);
inserter.set(2, "Sales");
inserter.execute();

inserter.set(1, 109);
inserter.set(2, "Accounting");
inserter.execute();

inserter.close();
```

Concatination is recommended over host variables when writing SQL statements that only need to be executed once. The following example demonstrates a row insertion method that uses concatination to execute SQL statements:

```
void addRow(int id, String name, Connection conn) throws ULjException {
    PreparedStatement ps = conn.prepareStatement(
        "INSERT INTO Department(id, name) VALUES("
        + id
        + ", '"
        + name
        + "')";
    try {
        ps.execute();
    }
    finally {
        ps.close(); \\ close the PreparedStatement even if an error occurs
after execution.
    }
}
```

Steps(3) and (4) can be repeated as many times as needed.

Alternatively, when only a single INSERT is to be performed, the statement INSERT INTO Department(dept_no, name) VALUES(2, 'Electronics') could be PREPAREd, EXECUTEd and CLOSEd as described in the section above.

The statement could be created using Java String concatenation.

Retrieving data using SELECT

You can retrieve data using the executeQuery method of a PreparedStatement, which queries the database with a user-defined SQL statement. This method returns the query result as a ResultSet. The ResultSet can then be traversed to fetch the queried data.

Navigating the ResultSet object

A ResultSet contains the following methods that allow you to navigate through the query results of a SQL SELECT statement:

- **next** Move to the next row.
- **previous** Move to the previous row.

Retrieving data using a ResultSet

To SELECT data from a database

1. Prepare a new SQL statement as a String.

```
String sql_string =
    "SELECT * FROM Department ORDER BY dept_no";
```

2. Pass the String to the PreparedStatement.

```
PreparedStatement select_statement =
    conn.prepareStatement(sql_string);
```

3. Execute the statement and assign the query results to a ResultSet.

```
ResultSet cursor =
    select_statement.executeQuery();
```

4. Traverse through the ResultSet and retrieve the data.

```
// Get the next row stored in the ResultSet.
cursor.next();

// Store the data from the first column in the table.
int dept_no = cursor.getInt(1);

// Store the data from the second column in the table.
String dept_name = cursor.getString(2);
```

5. Close the ResultSet to free resources.

```
cursor.close();
```

6. Close the PreparedStatement to free resources.

```
select_statement.close()
```

Managing transactions using COMMIT, ROLLBACK

UltraLiteJ does not support AutoCommit mode. Transactions must be explicitly committed or rolled back using the methods supported by the Connection interface.

Use the commit method to commit transactions. Use the rollback method to roll back transactions.

See also

- "commit method" on page 122
- "rollback method" on page 130

Encrypting and obfuscating data

By default, data stored in UltraLiteJ databases are not encrypted. You can encrypt or obfuscate data using the API. Encryption provides secure representation of the data whereas obfuscation provides a simplistic level of security that is intended to prevent casual observation of the database contents.

To encrypt or obfuscate data in the database

1. Create a class that implements the EncryptionControl interface.

   The following example creates a new class, Encryptor, which implements the encryption interface.

   ```
   static class Encryptor
       implements EncryptionControl
   {
   ```

2. Implement the initialize, encrypt, and decrypt methods in the new class.

   Your class should now look similar to the following:

   ```
   static class Encryptor
       implements EncryptionControl
   {
       /** Decrypt a page stored in the database.
         * @param page_no the number of the page being decrypted
         * @param src the encrypted source page which was read from the
   database
         * @param tgt the decrypted page (filled in by method)
         */
       public void decrypt( int page_no, byte[] src, byte[] tgt )
           throws ULjException
       {
           // Your decryption method goes here.
       }

       /** Encrypt a page stored in the database.
         * @param page_no the number of the page being encrypted
         * @param src the unencrypted source
         * @param tgt the encrypted target page which will be written to the
   database (filled in by method)
         */
       public void encrypt( int page_no, byte[] src, byte[] tgt )
   ```

```
            throws ULjException
        {
            // Your encryption method goes here.
        }

        /** Initialize the encryption control with a password.
         * @param password the password
         */
        public void initialize(String password)
            throws ULjException
        {
            // Your initialization method goes here.
        }
    }
```

For more information on EncryptionControl methods, see "EncryptionControl interface" on page 160.

3. Configure the database to use your new class for encryption control.

   You can specify the encryption control with the setEncryption method. The following example assumes that you have created a new Configuration, config, that references the name of the database:

   ```
   config.setEncryption(new Encryptor());
   ```

4. Connect to the database.

   Any data that is added or modified in the database is now encrypted.

Note
Encryption and obfuscation is not available for non-persistent database stores.

See also

- "Example: Obfuscating data" on page 43
- "Example: Encrypting data" on page 48
- "EncryptionControl interface" on page 160

Synchronizing with MobiLink

Using UltraLiteJ as a MobiLink client

To synchronize data, your application must perform the following steps:

1. Instantiate a syncParms object, which contains information about the consolidated database (name of the server, port number), name of the database to be synchronized, and the definition of the tables to be synchronized.

2. Call the synchronize method from the connection object with the syncParms object to carry out the synchronization.

The data to be synchronized can be defined at the table level. You cannot configure synchronization for portions of a table.

See also

- "SyncParms class" on page 235
- "SyncResult class" on page 250

Example

The following example demonstrates data synchronization with an UltraLiteJ application and can be found in *samples-dir\UltraLiteJ\Sync.java*:

```
package ianywhere.ultralitej.demo;
import com.ianywhere.ultralitej12.*;
/**
 * Sync: sample program to demonstrate Database synchronization.
 *
 * Requires starting the MobiLink Server Sample using start_ml.bat
 */
public class Sync
{
    /**
     * mainline for program.
     *
     * @param args command-line arguments
     *
     */
    public static void main
        ( String[] args )
    {
        try {
            Configuration config =
DatabaseManager.createConfigurationFile( "Demo1.ulj" );
            Connection conn = DatabaseManager.createDatabase( config );

            PreparedStatement ps = conn.prepareStatement(
                "CREATE TABLE ULCustomer" +
                "( cust_id int NOT NULL PRIMARY KEY" +
                ", cust_name VARCHAR(30) NOT NULL" +
                ")"
            );
            ps.execute();
            ps.close();

            //
            // Synchronization
            //

           SyncParms syncParms = conn.createSyncParms( SyncParms.HTTP_STREAM,
"50", "custdb 12.0" );
            syncParms.getStreamParms().setPort( 9393 );
            conn.synchronize( syncParms );
            SyncResult result = syncParms.getSyncResult();
            Demo.display(
                    "*** Synchronized *** bytes sent=" +
result.getSentByteCount()
                    + ", bytes received=" + result.getReceivedByteCount()
                    + ", rows received=" + result.getReceivedRowCount()
                );

            conn.release();

        } catch( ULjException exc ) {
            Demo.displayException( exc );
            }
        }
    }
```

To start the MobiLink server with CustDB as the consolidated database, run *start_ml.bat* from the *samples-dir\UltraLiteJ* directory.

Network protocol options for UltraLiteJ synchronization streams

When synchronizing with a MobiLink server, you must set the network protocol in your application. Each database synchronizes over a network protocol. Two network protocols are available for UltraLiteJ—HTTP and HTTPS.

For the network protocol you set, you can choose from a set of corresponding protocol options to ensure that the UltraLiteJ application can locate and communicate with the MobiLink server. Network protocol options provide information such as addressing information (host and port) and protocol-specific information. To determine which options you can use for the stream type you are using, see "MobiLink client network protocol option summary" [*MobiLink - Client Administration*].

Setting up an HTTP network protocol

An HTTP network protocol is set with the StreamHTTPParms interface in the UltraLiteJ API. Use the interface methods to specify the network protocol options defined on the MobiLink server. For a complete list of network options, see "StreamHTTPParms interface" on page 221.

Setting up an HTTPS network protocol

An HTTPS network protocol is set with the StreamHTTPSParms interface in the UltraLiteJ API. Use the interface methods to specify the network protocol options defined on the MobiLink server. For a complete list of network options, see "StreamHTTPSParms interface" on page 225.

Synchronizing the CustDB application

CustDB (customer database) is a sample database and UltraLite application installed with SQL Anywhere. The CustDB database is a simple sales order database.

Finding and deploying the application

UltraLiteJ includes a sample BlackBerry application that is based on the CustDB database. The application is named CustDB; the source code and related files are found in the *sample-dir\ultralitej\CustDB\* directory. The CustDB directory includes the project files that can be opened with the BlackBerry JDE. Additional information about the CustDB application can be found in *readme.txt*.

After the CustDB application has been build, deploy *CustDB12.cod* and the required files to a simulator or device. See "Deploying UltraLiteJ applications" on page 26.

Files related to CustDB application

The following files in the CustDB project contain the database access code:

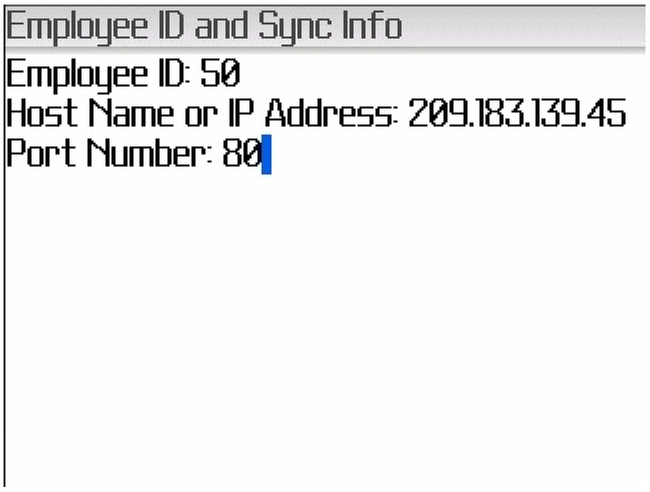
- ***CustDB.java*** This file contains all the basic database access methods. These methods include creating and connecting to databases, inserting, deleting and updating orders. This file contains many of the database calls to communicate to the back-end server.

- ***SchemaCreator.java*** This file contains the code to create tables on the device using UltraLiteJ.

Using the CustDB application

Run *sample-dir\ultralitej\CustDB\mobilink.bat* to start a local MobiLink server.

When initially started, the CustDB program collects information to use to interact with the server where the CustDB database is hosted. You specify the Employee ID to use for queries ("50" is recommended), the host name or IP address of the server where the data is hosted, and a port number for the connection to the server.

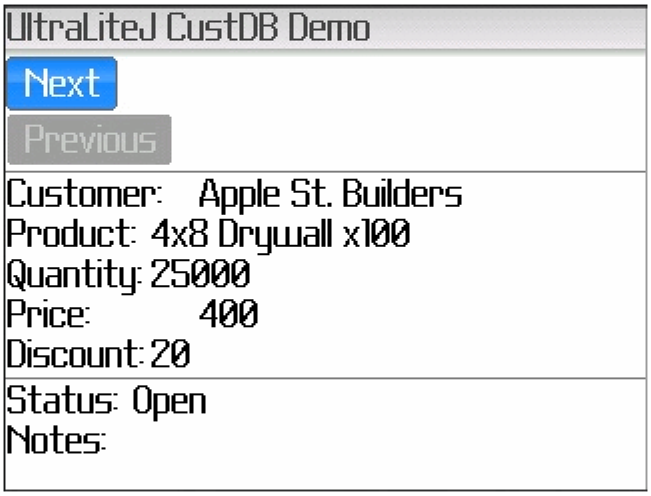

Employee ID and Sync Info

Employee ID: 50
Host Name or IP Address: 209.183.139.45
Port Number: 80

Note
It is recommended that you specify the IP address rather than the host name when using a simulator.

Once these values are specified and the settings are saved (**Menu** » **Save**), the application synchronizes with the specified server. The application only downloads orders from the server that match the Employee ID corresponding to the specified employee number (50). Only orders that are still open are selected (orders can be in one of three states: Open, Approved, or Denied).

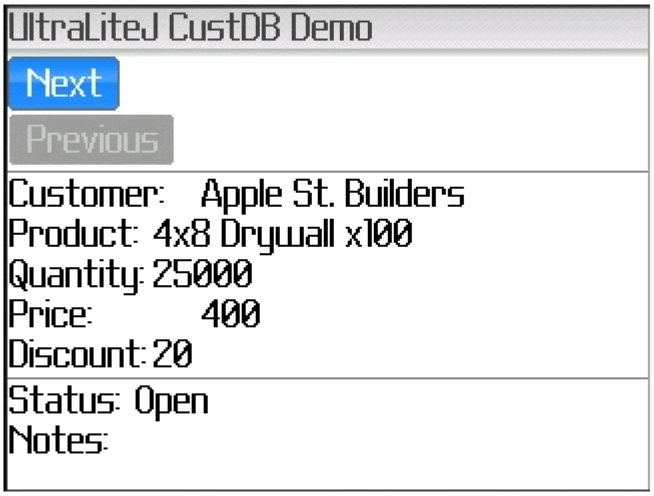
Each order is displayed on the screen with the following information: Customer Name, Ordered Product, Ordered Quantity, Price, and Discount. The screen also shows the current status of the order and any notes pertaining to that order.



Customer: Apple St. Builders
Product: 4x8 Drywall x100
Quantity: 25000
Price: 400
Discount: 20
Status: Open
Notes:

Once on this screen you can add notes to the order, or change the status of the order (to either Approved or Denied). You can navigate through the orders using the **Next** and **Previous** buttons.

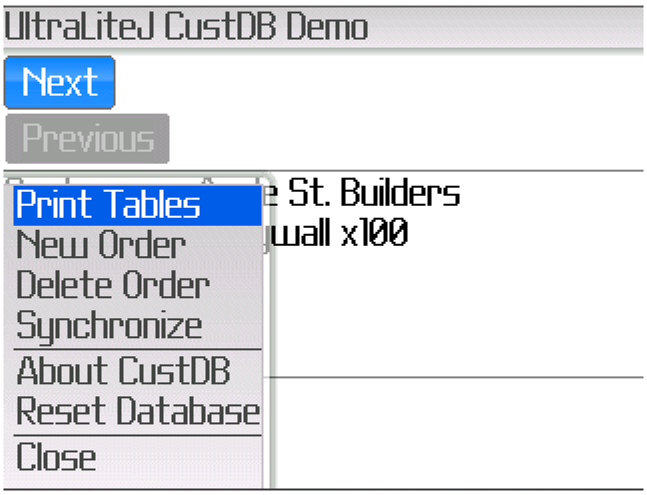
The CustDB program also allows you to add new orders into the database. To add a new order, click **Menu** » **New Order**.



Customer: Apple St. Builders
Product: 4x8 Drywall x100
Quantity: 25000
Price: 400
Discount: 20
Status: Open
Notes:

You may enter the quantity and discount values you require.

Before exiting the application, select **Synchronize** from the main menu to synchronize your changes and new orders with the server.



Deploying UltraLiteJ applications

For an UltraLiteJ application to run successfully, the UltraLiteJ API must be deployed with your distribution. The following table displays the list of files required for various deployments of UltraLiteJ. All file paths are relative to the *UltraLite\UltraLiteJ* directory of your SQL Anywhere installation.

Deployment type	Required files
BlackBerry smartphone	*BlackBerry4.2\UltraLiteJ.cod* *BlackBerry4.2\UltraLiteJ.jad* [1]
Java ME	*Java ME11\UltraLiteJ12.jar*
Java SE	*Java SE\UltraLiteJ12.jar*

[1] Required for over-the-air (OTA) deployment only. Alternatively, you can create your own jad file that deploys UltraLiteJ with your application.

Coding examples

This section illustrates examples of Java code that utilize the UltraLiteJ API. The examples use a demo class that displays messages and handles ULjException objects for debugging purposes.

All coding examples can be found in the *samples-dir\UltraLiteJ* directory. It is recommended that you create backup copies of the original source code before modifying the file contents. For information about the default location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

Although these code examples are written for the Windows desktop environment, the concepts presented apply to all UltraLiteJ platforms (unless otherwise noted).

To run these examples, set your **JAVA_HOME** environment variable to an installed version of the JDK 1.6. You can then run *rundemo.cmd* with the name of the example.

For example, the following command runs the CreateDb sample:

```
rundemo CreateDb
```

The command produces the following output:

```
Executing:
CREATE TABLE department
( dept_no INT NOT NULL PRIMARY KEY
, name VARCHAR(50)
)

Executing:
CREATE TABLE Employee
( number INT NOT NULL PRIMARY KEY
, last_name VARCHAR(32)
, first_name VARCHAR(32)
, age INT
, dept_no INT
, FOREIGN KEY fk_emp_to_dept( dept_no ) REFERENCES department( dept_no ))

CreateDb completed successfully
```

All the output into a file called *demos.out*.

BlackBerry examples

In addition to the Windows samples, the following demos are available for BlackBerry developers in the *samples-dir\UltraLiteJ* directory:

- The BinaryStoreAsFile demo illustrates the use of the ability to associate external files as part of a database.
- The CustDB demo shows a mobile customer order application.
- The BlackBerryEncryption demo demonstrates advance concepts for ultra-secure UltraLiteJ database.

Example: Demo class

This class is used by all the examples contained in this section of the documentation.

```
// *****************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// *****************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
```

```
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *********************************************************************
package com.ianywhere.ultralitej.demo;

//import java.io.*;
import com.ianywhere.ultralitej12.*;

/**
 * Demonstration class.
 *
 * <p>This class is not part of the Database library.  It is used
 * only by the demonstration programs.
 *
 * @author ianywhere
 * @version 1.0
 */
public class Demo
{
    /** Display a message.
     * @param msg message to be displayed
     */
    public static void display( String msg )
    {
    System.out.println( msg );
    }

    /** Display a message.
     * @param msg1 message(1) to be displayed
     * @param msg2 message(2) to be displayed
     */
    public static void display( String msg1, String msg2 )
    {
    display( msg1 + msg2 );
    }

    /** Display a message.
     * @param msg1 message(1) to be displayed
     * @param msg2 message(2) to be displayed
     * @param msg3 message(3) to be displayed
     */
    public static void display( String msg1, String msg2, String msg3 )
    {
    display( msg1 + msg2 + msg3 );
    }

    /** Display a message.
     * @param msg1 message(1) to be displayed
     * @param msg2 message(2) to be displayed
     * @param msg3 message(3) to be displayed
     * @param msg4 message(4) to be displayed
     */
    public static void display( String msg1, String msg2, String msg3, String
msg4 )
    {
    display( msg1 + msg2 + msg3 + msg4 );
    }

    /** Display message for an exception.
     * @param exc ULjException containing message
     */
    public static void displayException( ULjException exc )
```

```
            {
            display( exc.getMessage() );
            }
        }
```

Example: Creating a database

This example demonstrates how to create a file system database store in a Java SE Java environment. The Configuration object is used to create the database. Once created, a Connection object is returned. To create tables, the schemaUpdateBegin method is invoked to start changes to the underlying schema and the schemaUpdateComplete method completes changing the schema.

To run the CreateDb.java example

1. Change to the following directory: *samples-dir\UltraLiteJ*.

   For information about the default location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

2. Run the CreateDb example:

   ```
   rundemo CreateDb
   ```

Use the following command to run this example (the command is case sensitive):

```
rundemo CreateDb
```

Notes

- Tables in UltraLiteJ do not have owners and are identified only by name.
- The Domain interface defines constants to denote the various data types supported in a column in an UltraLiteJ table.
- The primary index (createPrimaryIndex) is guaranteed to be unique.

```
// *****************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// *****************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *********************************************************************
package com.ianywhere.ultralitej.demo;

import com.ianywhere.ultralitej12.*;

/**
 * CreateDb: sample program to demonstrate Database creation.
 */
public class CreateDb
```

```
{
    /**
     * mainline for program.
     *
     * @param args command-line arguments
     *
     */

    public static void main
    ( String[] args )
    {
    try {
        Configuration config =
DatabaseManager.createConfigurationFile( "Demo1.ulj" );
        Connection conn = DatabaseManager.createDatabase( config );

        executeSql( conn
              , "CREATE TABLE department\n"
               +"( dept_no INT NOT NULL PRIMARY KEY\n"
               +", name VARCHAR(50)\n"
               +")"
              );

        executeSql( conn
              , "CREATE TABLE Employee\n"
               +"( number INT NOT NULL PRIMARY KEY\n"
               +", last_name VARCHAR(32)\n"
               +", first_name VARCHAR(32)\n"
               +", age INT\n"
               +", dept_no INT\n"
               +", FOREIGN KEY fk_emp_to_dept( dept_no ) REFERENCES
department( dept_no )"
               +")"
              );

        Demo.display( "\nCreateDb completed successfully" );

    } catch( ULjException exc ) {
        Demo.displayException( exc );
    }
    }

    /** Execute a SQL statement.
     * @param conn connection to current database
     * @param stmt SQL statement to be executed
     */
    private static void executeSql( Connection conn, String stmt )
        throws ULjException
    {
    Demo.display( "\nExecuting:\n", stmt );
    PreparedStatement ps = conn.prepareStatement( stmt );
    ps.execute();
    ps.close();
    }
}
```

Example: Inserting rows

This example demonstrates how to insert rows in an UltraLiteJ database.

To run the LoadDb.java example

1. Change to the following directory: *samples-dir\UltraLiteJ*.

   For information about the default location of *samples-dir*, see “Samples directory” [*SQL Anywhere Server - Database Administration*].

2. Run the CreateDb example:

   ```
   rundemo CreateDb
   ```

   See “Example: Creating a database” on page 29.

3. Run the following command (the command is case sensitive):

   ```
   rundemo LoadDb
   ```

See .

Notes

- Inserted data is persisted in the database only when the commit method is called from the Connection object.
- When a row is inserted, but not yet committed, it is visible to other connections. This introduces the potential for a connection to retrieve row data that has not actually been committed.

```
// *****************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// *****************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *********************************************************************
package com.ianywhere.ultralitej.demo;

import com.ianywhere.ultralitej12.*;

/**
 * LoadDb -- sample program to demonstrate loading a Database.
 */
public class LoadDb
{
    /**
     * Add a Department row.
     * @param conn connection to Database
     * @param dept_no department number
     * @param dept_name department name
     */
    private static void addDepartment( PreparedStatement inserter, int 
dept_no, String dept_name )
        throws ULjException
    {
    inserter.set( 1 /* "dept_no" */, dept_no );
```

```
    inserter.set( 2 /* "name" */, dept_name );
    inserter.execute();
    }

    /**
     * Add an Employee row.
     * @param conn connection to Database
     * @param emp_no employee number
     * @param last_name employee last name
     * @param first_name employee first name
     * @param age employee age
     * @param dept_no department number where employee works
     */
    private static void addEmployee( PreparedStatement inserter, int emp_no,
String last_name
                   , String first_name, int age, int dept_no )
        throws ULjException
    {
    inserter.set( 1 /* "number" */, emp_no );
    inserter.set( 2 /* "last_name" */, last_name );
    inserter.set( 3 /* "first_name" */, first_name );
    inserter.set( 4 /* "age" */, age );
    inserter.set( 5 /* "dept_no" */, dept_no );
    inserter.execute();
    }

    /**
     * mainline for program.
     *
     * @param args command-line arguments
     *
     */
    public static void main
    ( String[] args )
    {
    try {
        Configuration config =
DatabaseManager.createConfigurationFile( "Demo1.ulj" );
        Connection conn = DatabaseManager.connect( config );
        PreparedStatement inserter;

        inserter = conn.prepareStatement( "INSERT INTO Department( dept_no,
name ) VALUES( ?, ? )" );
        addDepartment( inserter, 100, "Engineering" );
        addDepartment( inserter, 110, "Sales" );
        addDepartment( inserter, 103, "Marketing" );
        inserter.close();

        inserter = conn.prepareStatement( "INSERT INTO employee( \"number\",
last_name, first_name, age, dept_no ) VALUES( ?, ?, ?, ?, ? )" );
        addEmployee( inserter, 1000, "Welch", "James", 58, 100 );
        addEmployee( inserter, 1010, "Iverson", "Victoria", 23, 103 );
        inserter.close();

        conn.commit();
        conn.release();
        Demo.display( "LoadDb completed successfully" );
    } catch( ULjException exc ) {
        Demo.displayException( exc );
    }
    }
}
```

Example: Reading a table

In this example, a PreparedStatement object is obtained from a connection, and a ResultSet object is obtained from the PreparedStatement. The next method on the ResultSet returns true each time a subsequent row can be obtained. Values for the columns in the current row can then be obtained from the ResultSet object.

To run the ReadSeq.java example

1. Change to the following directory: *samples-dir\UltraLiteJ*.

   For information about the default location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

2. Run the CreateDb example:

   ```
   rundemo CreateDb
   ```

   See "Example: Creating a database" on page 29.

3. Run the LoadDb example:

   ```
   rundemo LoadDb
   ```

   See "Example: Inserting rows" on page 30.

4. Run the following command (the command is case sensitive):

   ```
   rundemo ReadSeq
   ```

Notes

- When a ResultSet is created, it is positioned before the first row of the result set. The code must invoke the next method to move to the first row in the table.
- Table and column names in UltraLiteJ are case insensitive. Columns can be referenced either by the column name alone ("age") or by qualifying the column name with the table name ("Employee.age").

```
// *****************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// *****************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *********************************************************************
package com.ianywhere.ultralitej.demo;

import com.ianywhere.ultralitej12.*;

/**
 * ReadSeq -- sample program to demonstrate reading a Database table
```

```
 * sequentially.
 */
public class ReadSeq
{
    /**
     * mainline for program.
     *
     * @param args command-line arguments
     *
     */
    public static void main
    ( String[] args )
    {
    try {
        Configuration config =
DatabaseManager.createConfigurationFile( "Demo1.ulj" );
        Connection conn = DatabaseManager.connect( config );
        PreparedStatement stmt = conn.prepareStatement( "SELECT * FROM
Employee ORDER BY number" );
        ResultSet cursor = stmt.executeQuery();
        for( ; cursor.next(); ) {
        int emp_no = cursor.getInt( 1 /* "number" */ );
        String last_name = cursor.getString( "last_name" );
        String first_name = cursor.getString( 3 /* "first_name" */ );
        int age = cursor.getInt( 4 /* "age" */ );
        Demo.display( first_name + ' ' + last_name );
        Demo.display( "  empl. no = "
                , Integer.toString( emp_no )
                , "  age = "
                , Integer.toString( age ) );
        }
        cursor.close();
        stmt.close();
        conn.release();
    } catch( ULjException exc ) {
        Demo.displayException( exc );
    }
    }
}
```

Example: Inner join operations

This example demonstrates how to perform an inner join operation. In this scenario, every employee has corresponding department information. The join operation associates data from the employee table with corresponding data from the department table. The association is made with the department number in the employee table to locate the related information in the department table.

To run the ReadInnerJoin.java example

1. Change to the following directory: *samples-dir\UltraLiteJ*.

   For information about the default location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

2. Run the CreateDb example:

   ```
   rundemo CreateDb
   ```

See “Example: Creating a database” on page 29.

3. Run the LoadDb example:

```
rundemo LoadDb
```

See “Example: Inserting rows” on page 30.

4. Run the following command (the command is case sensitive):

```
rundemo ReadInnerJoin
```

```
// *****************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// *****************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *********************************************************************
package com.ianywhere.ultralitej.demo;

import com.ianywhere.ultralitej12.*;

/**
 * ReadInnerJoin -- sample program to demonstrate reading the Employee table
 * and joining to each row the corresponding Department row.
 */
public class ReadInnerJoin
{
    /**
     * mainline for program.
     *
     * @param args command-line arguments
     *
     */
    public static void main
    ( String[] args )
    {
    try {
        Configuration config =
DatabaseManager.createConfigurationFile( "Demo1.ulj" );
        Connection conn = DatabaseManager.connect( config );
        PreparedStatement stmt = conn.prepareStatement(
            "SELECT E.number, E.last_name, E.first_name, E.age,"
            + " E.dept_no, D.name"
            + " FROM Employee E"
            + " JOIN Department D ON E.dept_no = D.dept_no"
            + " ORDER BY E.number"
        );
        ResultSet cursor = stmt.executeQuery();
        for( ; cursor.next(); ) {
        int emp_no = cursor.getInt( 1 /* "E.number" */ );
        String last_name = cursor.getString( "E.last_name" );
        String first_name = cursor.getString( 3 /* "E.first_name" */ );
        int age = cursor.getInt( 4 /* "E.age" */ );
        int dept_no = cursor.getInt( 5 /* "E.dept_no" */ );
```

```
                String dept_name = cursor.getString( 6 /* "D.name" */ );
                System.out.println( first_name + ' ' + last_name );
                System.out.print( "  empl. no = " );
                System.out.print( emp_no );
                System.out.print( "  dept = " );
                System.out.println( dept_no );
                System.out.print( "  age = " );
                System.out.print( age );
                System.out.println( ", " + dept_name );
                }
                cursor.close();
                stmt.close();
                conn.release();
                Demo.display( "ReadInnerJoin completed successfully" );
            } catch( ULjException exc ) {
                Demo.displayException( exc );
            }
            }
        }
```

Example: Creating a sales database

In this example, a sales-oriented database is created.

To run the CreateSales.java example

1. Change to the following directory: *samples-dir\UltraLiteJ*.

   For information about the default location of *samples-dir*, see “Samples directory” [*SQL Anywhere Server - Database Administration*].

2. Run the following command (the command is case sensitive):

   ```
   rundemo CreateSales
   ```

```
// *****************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// *****************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *********************************************************************
package com.ianywhere.ultralitej.demo;
import com.ianywhere.ultralitej12.*;

/**
 * CreateDb: sample program to demonstrate creation of simple sales Database
and
 * load it with some data.
 * <p>The program also illustrates the use of ordinals when inserting rows
into tables.
 */
```

```
public class CreateSales
{

    static int ORDINAL_INVOICE_INV_NO;
    static int ORDINAL_INVOICE_NAME;
    static int ORDINAL_INVOICE_DATE;

    static int ORDINAL_INV_ITEM_INV_NO;
    static int ORDINAL_INV_ITEM_ITEM_NO;
    static int ORDINAL_INV_ITEM_PROD_NO;
    static int ORDINAL_INV_ITEM_QUANTITY;
    static int ORDINAL_INV_ITEM_PRICE;

    static int ORDINAL_PROD_NO;
    static int ORDINAL_PROD_NAME;
    static int ORDINAL_PROD_PRICE;

    /** Create the Database.
     * @return connection for a new Database
     */
    private static Connection createDatabase()
        throws ULjException
    {
    Configuration config =
DatabaseManager.createConfigurationFile( "Sales.ulj" );
    Connection conn = DatabaseManager.createDatabase( config );

    executeSql( conn
          , "CREATE TABLE Product\n"
           +"( prod_no INT NOT NULL PRIMARY KEY\n"
           +", prod_name VARCHAR(32)\n"
           +", price NUMERIC(9,2)"
           +", SYNCHRONIZE OFF"
           +")"
          );

    executeSql( conn
          , "CREATE TABLE Invoice\n"
           +"( inv_no INT NOT NULL PRIMARY KEY\n"
           +", name VARCHAR(50)\n"
           +", \"date\" DATE\n"
           +", SYNCHRONIZE OFF"
           +")"
          );

    executeSql( conn
          , "CREATE TABLE InvoiceItem\n"
           +"( inv_no INT NOT NULL\n"
           +", item_no INT NOT NULL\n"
           +", prod_no INT NOT NULL\n"
           +", quantity INT NOT NULL\n"
           +", price NUMERIC(9,2)"
           +", PRIMARY KEY( inv_no, item_no )\n"
           +", SYNCHRONIZE OFF"
           +")"
          );

    return conn;
    }

    /** Populate the Database.
     * @param conn connection to Database
     */
    private static void populateDatabase( Connection conn )
```

```
            throws ULjException
        {
        PreparedStatement ri_product = conn.prepareStatement(
            "INSERT INTO Product( prod_no, prod_name, price ) VALUES( ?, ?, ? )"
            );
        ORDINAL_PROD_NO = 1;
        ORDINAL_PROD_NAME = 2;
        ORDINAL_PROD_PRICE = 3;
        addProduct( ri_product, 2001, "blue screw", ".03" );
        addProduct( ri_product, 2002, "red screw", ".09" );
        addProduct( ri_product, 2004, "hammer", "23.99" );
        addProduct( ri_product, 2005, "vice", "39.99" );
        ri_product.close();

        PreparedStatement ri_invoice = conn.prepareStatement(
            "INSERT INTO Invoice( inv_no, name, \"date\" )"
            + " VALUES( :inv_no, :name, :inv_date )"
            );
        ORDINAL_INVOICE_INV_NO = ri_invoice.getOrdinal( "inv_no" );
        ORDINAL_INVOICE_NAME = ri_invoice.getOrdinal( "name" );
        ORDINAL_INVOICE_DATE = ri_invoice.getOrdinal( "inv_date" );

        PreparedStatement ri_item = conn.prepareStatement(
            "INSERT INTO InvoiceItem( inv_no, item_no, prod_no, quantity,
    price )"
            + " VALUES( ?, ?, ?, ?, ? )"
            );
        ORDINAL_INV_ITEM_INV_NO = 1;
        ORDINAL_INV_ITEM_ITEM_NO = 2;
        ORDINAL_INV_ITEM_PROD_NO = 3;
        ORDINAL_INV_ITEM_QUANTITY = 4;
        ORDINAL_INV_ITEM_PRICE = 5;

        addInvoice( ri_invoice, 2006001, "Jones Mfg.", "2006/12/23" );
        addInvoiceItem( ri_item, 2006001, 1, 2001, 3000, ".02" );
        addInvoiceItem( ri_item, 2006001, 2, 2002, 5000, ".08" );

        addInvoice( ri_invoice, 2006002, "Smith Inc.", "2006/12/24" );
        addInvoiceItem( ri_item, 2006002, 1, 2004, 2, "23.99" );
        addInvoiceItem( ri_item, 2006002, 2, 2005, 3, "39.99" );

        addInvoice( ri_invoice, 2006003, "Lee Ltd.", "2006/12/24" );
        addInvoiceItem( ri_item, 2006003, 1, 2004, 5, "23.99" );
        addInvoiceItem( ri_item, 2006003, 2, 2005, 4, "39.99" );
        addInvoiceItem( ri_item, 2006003, 3, 2001, 800, ".03" );
        addInvoiceItem( ri_item, 2006003, 4, 2002, 700, ".09" );

        ri_item.close();
        ri_invoice.close();

        conn.commit();
        }

        /**
         * mainline for program.
         *
         * @param args command-line arguments
         *
         */
        public static void main
        ( String[] args )
        {
        try {
            Connection conn = createDatabase();
            populateDatabase( conn );

            conn.release();

            Demo.display( "CreateSales completed successfully" );

        } catch( ULjException exc ) {
            Demo.displayException( exc );
        }
        }

        /** Add an invoice row.
         * @param conn connection to Database
         * @param inv_no invoice number
         * @param name name to whome invoice was sent
         */
        private static void addInvoice( PreparedStatement ri, int inv_no, String
    name
                              , String date )
            throws ULjException
        {
        ri.set( ORDINAL_INVOICE_INV_NO, inv_no );
        ri.set( ORDINAL_INVOICE_NAME, name );
        ri.set( ORDINAL_INVOICE_DATE, date );
        ri.execute();
        }

        /** Add an invoice-item row.
         * @param conn connection to Database
         * @param inv_no invoice number
         * @param item_no line number for item
         * @param prod_no product number sold
         * @param quantity quantity sold
         * @param price price of one item
         */
        private static void addInvoiceItem( PreparedStatement ri, int inv_no, int
    item_no
                              , int prod_no, int quantity, String price )
            throws ULjException
        {
        ri.set( ORDINAL_INV_ITEM_INV_NO, inv_no );
        ri.set( ORDINAL_INV_ITEM_ITEM_NO, item_no );
        ri.set( ORDINAL_INV_ITEM_PROD_NO, prod_no );
        ri.set( ORDINAL_INV_ITEM_QUANTITY, quantity );
        ri.set( ORDINAL_INV_ITEM_PRICE, price );
        ri.execute();
        }

        /** Add a product row.
         * @param conn connection to Database
         * @param prod_no product number
         * @param prod_name product name
         * @param price selling price
         */
        private static void addProduct( PreparedStatement ri, int prod_no, String
    prod_name, String price )
            throws ULjException
        {
        ri.set( ORDINAL_PROD_NO, prod_no );
        ri.set( ORDINAL_PROD_NAME, prod_name );
        ri.set( ORDINAL_PROD_PRICE, price );
        ri.execute();
        }
        /** Execute a SQL statement.
         * @param conn connection to current database
         * @param stmt SQL statement to be executed
         */
        private static void executeSql( Connection conn, String stmt )
        {
        try {
            PreparedStatement ps = conn.prepareStatement( stmt );
            ps.execute();
            ps.close();
        } catch( ULjException exc ) {
            Demo.displayException( exc );
        }
        }
    }
```

Example: Aggregation and grouping

This example demonstrates UltraLiteJ support for the aggregation of results.

To run the SalesReport.java example

1. Change to the following directory: *samples-dir\UltraLiteJ*.

   For information about the default location of *samples-dir*, see “Samples directory” [*SQL Anywhere Server - Database Administration*].

2. Run the CreateSales example:

   ```
   rundemo CreateSales
   ```

   See “Example: Creating a sales database” on page 36.

3. Run the following command (the command is case sensitive):

   ```
   rundemo SalesReport
   ```

```
// *****************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// *****************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *********************************************************************
package com.ianywhere.ultralitej.demo;
import com.ianywhere.ultralitej12.*;

/** Create a sales report to illustrate aggregation support.
 */
public class SalesReport
{
```

```
    /** Mainline.
     * @param args program arguments (not used)
     */
    public static void main( String[] args )
    {
    try {
        Configuration config = 
DatabaseManager.createConfigurationFile( "Sales.ulj" );
        Connection conn = DatabaseManager.connect( config );
        PreparedStatement stmt = conn.prepareStatement(
            "SELECT inv_no, SUM( quantity * price ) AS total"
            + " FROM InvoiceItem"
            + " GROUP BY inv_no ORDER BY inv_no"
        );
        ResultSet agg_cursor = stmt.executeQuery();
        for( ; agg_cursor.next(); ) {
        int inv_no = agg_cursor.getInt( 1 /* "inv_no" */ );
        String total = agg_cursor.getString( "total" ); // access by name
        Demo.display( Integer.toString( inv_no ) + ' ' + total );
        }
        Demo.display( "SalesReport completed successfully" );
    } catch( ULjException exc ) {
        Demo.displayException( exc );
    }
    }
}
```

Example: Retrieving rows in an alternative order

This example demonstrates UltraLiteJ support for processing rows in an alternate order.

To run the SortTransactions.java example

1. Change to the following directory: *samples-dir\UltraLiteJ*.

   For information about the default location of *samples-dir*, see “Samples directory” [*SQL Anywhere Server - Database Administration*].

2. Run the CreateSales example:

   ```
   rundemo CreateSales
   ```

   See “Example: Creating a sales database” on page 36.

3. Run the following command (the command is case sensitive):

   ```
   rundemo SortTransactions
   ```

```
// *****************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// *****************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
```

```
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *******************************************************************
package com.ianywhere.ultralitej.demo;
import com.ianywhere.ultralitej12.*;

/** Illustrate use of SortCursor to produce a
 *  stream of rows in a different order.
 */
public class SortTransactions
{
    /** Mainline.
     * @param args program arguments (not used)
     */
    public static void main( String[] args )
    {
    try {
        Configuration config =
DatabaseManager.createConfigurationFile( "Sales.ulj" );
        Connection conn = DatabaseManager.connect( config );
        PreparedStatement stmt = conn.prepareStatement(
            "SELECT inv_no, prod_no, quantity FROM InvoiceItem"
            + " ORDER BY prod_no"
        );
        ResultSet cursor = stmt.executeQuery();
        for( ; cursor.next(); ) {
        int inv_no = cursor.getInt( 1 /* "inv_no" */ );
        int prod_no = cursor.getInt( "prod_no" ); //access by name
        int quantity = cursor.getInt( 3 /* "quantity" */ );
        Demo.display( Integer.toString( prod_no ) + ' '
                + Integer.toString( inv_no ) + ' '
                + Integer.toString( quantity ) );
        }
        conn.release();
        Demo.display( "SortTransactions completed successfully" );
    } catch( ULjException exc ) {
        Demo.displayException( exc );
    }
    }
}
```

Example: Modifying table definitions

This example demonstrates how to change table definitions. In this scenario, an Invoice table is modified to expand a column length from 50 characters to 100 characters.

To run the Reorg.java example

1. Change to the following directory: *samples-dir\UltraLiteJ*.

   For information about the default location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

2. Run the CreateSales example:

   ```
   rundemo CreateSales
   ```

See “Example: Creating a sales database” on page 36.

3. Run the following command (the command is case sensitive):

```
rundemo Reorg
```

```
// ***************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// ***************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *******************************************************************
package com.ianywhere.ultralitej.demo;
import com.ianywhere.ultralitej12.*;

/** Reorganize the Invoice table to have a name with an increased size
 *
 * <p>This shows a possible strategy which can be used to reorganize
 * tables, since UltraLiteJ has no table-altering API.
 * <p>The (contrived) example expands the name column to 100 characters.
 *
 */
public class Reorg
{
    /**
     * mainline for program.
     *
     * @param args command-line arguments
     *
     */
    public static void main
    ( String[] args )
    {
    try {
        Configuration config =
DatabaseManager.createConfigurationFile( "Sales.ulj" );
        Connection conn = DatabaseManager.connect( config );
        PreparedStatement ps = conn.prepareStatement(
            "ALTER TABLE Invoice ALTER name VARCHAR(100)" // was VARCHAR(50)
        );
        ps.execute();
        ps.close();
        conn.release();
    } catch( ULjException exc ) {
        Demo.displayException( exc );
    }
    }
}
```

Example: Obfuscating data

This example demonstrates a technique for obfuscating data in a database. The purpose of obfuscation is to make the data unrecognizable if it was simply displayed; it is not the same as encryption, which also has the property that there is a fair chance that the data cannot be decrypted, even when sophisticated tools are employed.

To run the Obfuscate.java example

1. Change to the following directory: *samples-dir\UltraLiteJ*.

   For information about the default location of *samples-dir*, see “Samples directory” [*SQL Anywhere Server - Database Administration*].

2. Run the following command (the command is case sensitive):

   ```
   rundemo Obfuscate
   ```

```
// *****************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// *****************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *********************************************************************
package com.ianywhere.ultralitej.demo;

import com.ianywhere.ultralitej12.*;

/**
 * Obfuscate -- sample program to a possible obfuscation of the database.
 *
 * Obfuscation is not very good encryption.  It merely makes the data
unreadable
 * with a file dumping program.  The original data can be probably recovered
by
 * someone with knowledge of the algorithms used.
 */
public class Obfuscate
{
    /** Create the database.
     * @return connection for a new database
     */
    private static Connection createDatabase()
        throws ULjException
    {
    ConfigPersistent config =
DatabaseManager.createConfigurationFile( "Obfuscate.ulj" );
    config.setEncryption( new Obfuscator() );
    Connection conn = DatabaseManager.createDatabase( config );

    PreparedStatement ps = conn.prepareStatement(
        "CREATE TABLE Product" +
        "( prod_no INT NOT NULL PRIMARY KEY" +
        ", prod_name VARCHAR(32)" +
        ", price NUMERIC(9,2)" +
        ")"
        );
    ps.execute();
```

```
    ps.close();

    return conn;
    }

    /** Add a product row.
     * @param ri PreparedStatement for the Product table
     * @param prod_no product number
     * @param prod_name product name
     * @param price selling price
     */
    private static void addProduct( PreparedStatement ri, int prod_no, String
prod_name, String price )
        throws ULjException
    {
    ri.set( "prod_no", prod_no );
    ri.set( "prod_name", prod_name );
    ri.set( "price", price );
    ri.execute();
    }

    /** Populate the database.
     * @param conn connection to database
     */
    private static void populate( Connection conn )
        throws ULjException
    {
    PreparedStatement ri = conn.prepareStatement(
        "INSERT INTO Product( prod_no, prod_name, price )"
        + " VALUES( :prod_no, :prod_name, :price )"
        );
    addProduct( ri, 2001, "blue screw", ".03" );
    addProduct( ri, 2002, "red screw", ".09" );
    addProduct( ri, 2004, "hammer", "23.99" );
    addProduct( ri, 2005, "vise", "39.99" );
    ri.close();
    conn.commit();
    }

    /** Display contents of Product table.
     * @param conn connection to database
     */
    private static void displayProducts( Connection conn )
        throws ULjException
    {
    PreparedStatement stmt = conn.prepareStatement(
        "SELECT prod_no, prod_name, price FROM Product"
        + " ORDER BY prod_no"
        );
    ResultSet cursor = stmt.executeQuery();
    for( ; cursor.next(); ) {
        String prod_no = cursor.getString( 1 /* "prod_no" */ );
        String prod_name = cursor.getString( "prod_name" );
        String price = cursor.getString( 3 /* "price" */ );
        Demo.display( prod_no + " " + prod_name + " " + price );
    }
    cursor.close();
    stmt.close();
    }

    /** mainline for program.
     * @param args command-line arguments (not used)
     */
    public static void main
```

```
        ( String[] args )
        {
        try {
            Connection conn = createDatabase();
            populate( conn );
            displayProducts( conn );
            conn.release();
        } catch( ULjException exc ) {
            Demo.displayException( exc );
        }
        }

        /** Class to implement encryption/decryption of the database.
         */
        static class Obfuscator
            implements EncryptionControl
        {
        /** seed for obfuscator        */    private int _seed;

        /** (un)Obfuscate a page.
         * @param page_no the number of the page being encrypted
         * @param src the encrypted source page which was read from the database
         * @param tgt the unencrypted page (filled in by method)
         * @param num_bytes The number of bytes to decrypt.  This is always
         *          either the page size or 128.
         */
        private void transform( int page_no, byte[] src, byte[] tgt, int
    num_bytes )
        {
            int seed = ( _seed + page_no ) % 256;
            for( int i = 0; i < num_bytes; ++i ) {
            tgt[ i ] = (byte)( seed ^ src[ i ] );
            seed = ( seed + 93 ) % 256;
            }
        }

        /** Encrypt a page stored in the database.
         * @param page_no the number of the page being encrypted
         * @param src the encrypted source page which was read from the database
         * @param tgt the unencrypted page (filled in by method)
         * @param num_bytes The number of bytes to decrypt.  This is always
         *          either the page size or 128.
         */
        public void decrypt( int page_no, byte[] src, byte[] tgt, int num_bytes )
            throws ULjException
        {
            transform( page_no, src, tgt, num_bytes );
        }

        /** Encrypt a page stored in the database.
         * @param page_no the number of the page being encrypted
         * @param src the unencrypted source
         * @param tgt the encrypted target page which will be written to the
    database (filled in by method)
         */
        public void encrypt( int page_no, byte[] src, byte[] tgt )
            throws ULjException
        {
            transform( page_no, src, tgt, tgt.length );
        }

        /** Initialize the encryption control with a password.
         * @param password the password
         */
        public void initialize( String password )
            throws ULjException
        {
            byte[] bytes = null;
            try {
            bytes = password.getBytes( "UTF8" );
            } catch( Exception e ) {
            Demo.display( "Encryption initialization failure" );
            throw new ObfuscationError();
            }
            _seed = 0;
            for( int i = bytes.length; i > 0; ) {
            _seed ^= bytes[ --i ];
            }
        }
        }

        /** Error class for encryption errors.
         */
        static class ObfuscationError
            extends ULjException
        {
        /** Constructor.
         */
        ObfuscationError()
        {
            super( "Obfuscation Error" );
        }

        /**
         * Get the error code, associated with this exception.
         * @return the error code (from the list at the top of this class)
    associated
         *  with this exception
         */
        public int getErrorCode()
        {
            return ULjException.SQLE_ERROR;
        }

        /** Get exception causing this exception, if it exists.
         * @return null, if there exists no causing exception; otherwise, the
    exception causing this exception
         */
        public ULjException getCausingException()
        {
            return null;
        }

        /** Get offset of error within a SQL string.
         * @return (-1) when there is no SQL string associated with the error
    message; otherwise,
         * the (base 0) offset within that string where the error occurred.
         */
        public int getSqlOffset()
        {
            return -1;
        }
        }
    }
```

Example: Encrypting data

This example demonstrates a technique for encrypting data in a database. In this scenario, decrypting the data incurs a performance penalty.

To run the Encrypted.java example

1. Change to the following directory: *samples-dir\UltraLiteJ*.

   For information about the default location of *samples-dir*, see “Samples directory” [*SQL Anywhere Server - Database Administration*].

2. Run the following command (the command is case sensitive):

   ```
   rundemo Encrypted
   ```

Note
This example is for Java SE. For a complete BlackBerry encryption sample, see *samples-dir\UltraLiteJ\BlackBerryEncryption*.

```
// *****************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// *****************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *********************************************************************
package com.ianywhere.ultralitej.demo;

import com.ianywhere.ultralitej12.*;

import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;

/**
 * Encrypted -- sample program to demonstrate encryption of database.
 *
 * This sample requires either the Sun JDK 1.5 (or later) or a freeware
 * version of the Java encryption classes (JCE).
 */
public class Encrypted
{
    /** Create the database.
     * @return connection for a new database
     */
    private static Connection createDatabase()
        throws ULjException
    {
    ConfigPersistent config =
DatabaseManager.createConfigurationFile( "Encrypt.ulj" );
```

```
        config.setEncryption( new Encryptor() );
        Connection conn = DatabaseManager.createDatabase( config );

        PreparedStatement ps = conn.prepareStatement(
            "CREATE TABLE Product" +
            "( prod_no INT NOT NULL PRIMARY KEY" +
            ", prod_name VARCHAR(32)" +
            ", price NUMERIC(9,2)" +
            ")"
            );
        ps.execute();
        ps.close();

        return conn;
        }

        /** Add a product row.
         * @param ri PreparedStatement for the Product table
         * @param prod_no product number
         * @param prod_name product name
         * @param price selling price
         */
        private static void addProduct( PreparedStatement ri, int prod_no, String
    prod_name, String price )
            throws ULjException
        {
        ri.set( "prod_no", prod_no );
        ri.set( "prod_name", prod_name );
        ri.set( "price", price );
        ri.execute();
        }

        /** Populate the database.
         * @param conn connection to database
         */
        private static void populate( Connection conn )
            throws ULjException
        {
        PreparedStatement ri = conn.prepareStatement(
            "INSERT INTO Product( prod_no, prod_name, price )"
            + " VALUES( :prod_no, :prod_name, :price )"
            );
        addProduct( ri, 2001, "blue screw", ".03" );
        addProduct( ri, 2002, "red screw", ".09" );
        addProduct( ri, 2004, "hammer", "23.99" );
        addProduct( ri, 2005, "vise", "39.99" );
        ri.close();
        conn.commit();
        }

        /** Display contents of Product table.
         * @param conn connection to database
         */
        private static void displayProducts( Connection conn )
            throws ULjException
        {
        PreparedStatement stmt = conn.prepareStatement(
            "SELECT prod_no, prod_name, price FROM Product"
            + " ORDER BY prod_no"
            );
        ResultSet cursor = stmt.executeQuery();
        for( ; cursor.next(); ) {
            String prod_no = cursor.getString( 1 /* "prod_no" */ );
            String prod_name = cursor.getString( "prod_name" ); //access by name
            String price = cursor.getString( 3 /* "price" */ );
            Demo.display( prod_no + " " + prod_name + " " + price );
        }
        cursor.close();
        stmt.close();
        }

        /** mainline for program.
         * @param args command-line arguments (not used)
         */
        public static void main
        ( String[] args )
        {
        try {
            Connection conn = createDatabase();
            populate( conn );
            displayProducts( conn );
            conn.release();
        } catch( ULjException exc ) {
            Demo.displayException( exc );
        }
        }

        /** Class to implement encryption/decryption of the database.
         */
        static class Encryptor
            implements EncryptionControl
        {
        private SecretKeySpec _key = null;
        private Cipher _cipher = null;

        /** Encrypt a page stored in the Database.
         * @param page_no the number of the page being encrypted
         * @param src the encrypted source page which was read from the Database
         * @param tgt the unencrypted page (filled in by method)
         * @param num_bytes the number of bytes to be decrypted
         */
        public void decrypt( int page_no, byte[] src, byte[] tgt, int num_bytes )
            throws ULjException
        {
            byte[] decrypted = null;
            try {
            _cipher.init( Cipher.DECRYPT_MODE, _key );
            decrypted = _cipher.doFinal( src );
            } catch( Exception e ) {
            Demo.display( "Error: decrypting" );
            throw new EncryptionError();
            }
            System.arraycopy( decrypted, 0, tgt, 0, num_bytes );
        }

        /** Encrypt a page stored in the Database.
         * @param page_no the number of the page being encrypted
         * @param src the unencrypted source
         * @param tgt the encrypted target page which will be written to the
    Database (filled in by method)
         */
        public void encrypt( int page_no, byte[] src, byte[] tgt )
            throws ULjException
        {
            byte[] encrypted = null;
            try {
            _cipher.init( Cipher.ENCRYPT_MODE, _key );
            encrypted = _cipher.doFinal( src );
        } catch( Exception e ) {
        Demo.display( "Error: encrypting" );
        throw new EncryptionError();
        }
        System.arraycopy( encrypted, 0, tgt, 0, tgt.length );
    }

    /** Initialize the encryption control with a password.
     * @param password the password
     */
    public void initialize( String password )
        throws ULjException
    {
        try {
        byte[] bytes = password.getBytes( "UTF8" );
        MessageDigest md = MessageDigest.getInstance( "SHA" );
        bytes = md.digest( bytes );
        byte[] key_bytes = new byte[16];
        for( int i = key_bytes.length; i > 0; ) {
            --i;
            key_bytes[ i ] = bytes[ i ];
        }
        _key = new SecretKeySpec( key_bytes, "AES" );
        _cipher = Cipher.getInstance( "AES/ECB/NoPadding" );
        } catch( Exception e ) {
        Demo.display( "Error: initializing encryption" );
        throw new EncryptionError();
        }
    }
    }

    /** Error class for encryption errors.
     */
    static class EncryptionError
        extends ULjException
    {
    /** Constructor.
     */
    EncryptionError()
    {
        super( "Encryption Error" );
    }

    /**
     * Get the error code, associated with this exception.
     * @return the error code (from the list at the top of this class)
associated
     *  with this exception
     */
    public int getErrorCode()
    {
        return ULjException.SQLE_ERROR;
    }

    /** Get exception causing this exception, if it exists.
     * @return null, if there exists no causing exception; otherwise, the
exception causing this exception
     */
    public ULjException getCausingException()
    {
        return null;
    }

    /** Get offset of error within a SQL string.
```

```
     * @return (-1) when there is no SQL string associated with the error
message; otherwise,
     * the (base 0) offset within that string where the error occurred.
     */
    public int getSqlOffset()
    {
        return -1;
    }
    }
}
```

Example: Displaying database schema information

This example demonstrates how to navigate the system tables of an UltraLiteJ database to examine the schema information. The data for each row of the tables also appears.

To run the DumpSchema.java example

1. Change to the following directory: *samples-dir\UltraLiteJ*.

   For information about the default location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

2. Run the CreateSales example:

   ```
   rundemo CreateSales
   ```

   See "Example: Creating a sales database" on page 36.

3. Run the following command (the command is case sensitive):

   ```
   rundemo DumpSchema
   ```

```
// *****************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// *****************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *******************************************************************
package com.ianywhere.ultralitej.demo;
import com.ianywhere.ultralitej12.*;

/** Sample program to dump schema of a database.
 * This sample extracts schema information into a set of data structures
 * (TableArray, OptionArray) before dumping out the meta data so as to
 * provide for future schema information lookup.
 */
public class DumpSchema
{
    /** Mainline.
```

```
     * @param args program arguments (not used)
     */
    public static void main( String[] args )
    {
    try {
        Configuration config =
DatabaseManager.createConfigurationFile( "Sales.ulj" );
        Connection conn = DatabaseManager.connect( config );

        Demo.display(
            TableSchema.SYS_TABLES
            + " table_flags are:\nTableSchema.TABLE_IS_SYSTEM(0x"
            + Integer.toHexString( ((int)TABLE_FLAG_SYSTEM) & 0xffff )
            + "),\nTableSchema.TABLE_IS_NOSYNC(0x"
            + Integer.toHexString( ((int)TABLE_FLAG_NO_SYNC) & 0xffff )
            + ")"
        );
        getSchema( conn );
        dumpSchema( conn );

    } catch( ULjException exc ) {
        Demo.displayException( exc );
    }
    }

    // Some constants for metadata
    private static String SQL_SELECT_TABLE_COLS =
        "SELECT T.table_id, T.table_name, T.table_flags,"
        + " C.column_id, C.column_name, C.column_flags,"
        + " C.column_domain, C.column_length, C.column_default,"
        + " C.column_default_value, C.filename_colid"
        + " FROM " + TableSchema.SYS_TABLES + " T"
        + " JOIN " + TableSchema.SYS_COLUMNS + " C"
        + " ON T.table_id = C.table_id"
        + " ORDER BY T.table_id"
        ;
    private static final int TABLE_ID = 1;
    private static final int TABLE_NAME = 2;
    private static final int TABLE_FLAGS = 3;
    private static final int COLUMN_ID = 4;
    private static final int COLUMN_NAME = 5;
    private static final int COLUMN_FLAGS = 6;
    private static final int COLUMN_DOMAIN_TYPE = 7;
    private static final int COLUMN_DOMAIN_LENGTH = 8;
    private static final int COLUMN_DEFAULT = 9;
    private static final int COLUMN_DEFAULT_VALUE = 10;
    private static final int COLUMN_FILENAME_COLID = 11;

    private static final int TABLE_FLAG_SYSTEM = TableSchema.TABLE_IS_SYSTEM;
    private static final int TABLE_FLAG_NO_SYNC =
TableSchema.TABLE_IS_NOSYNC;

    private static final int COLUMN_FLAG_IN_PRIMARY_INDEX = 0x01;
    private static final int COLUMN_FLAG_IS_NULLABLE = 0x02;
    private static final int COLUMN_FLAG_IS_CASCADE_DELETE = 0x04;

    private static String SQL_SELECT_INDEX_COLS =
        "SELECT I.table_id, I.index_id, I.index_name, I.index_flags,"
        + " X.\"order\", X.column_id, X.index_column_flags"
        + " FROM " + TableSchema.SYS_INDEX_COLUMNS + " X"
        + " JOIN " + TableSchema.SYS_INDEXES + " I"
        + " ON I.table_id = X.table_id AND I.index_id = X.index_id"
        + " ORDER BY X.table_id, X.index_id, X.\"order\""
        ;
    private static final int INDEX_TABLE_ID = 1;
    private static final int INDEX_ID = 2;
    private static final int INDEX_NAME = 3;
    private static final int INDEX_FLAGS = 4;
    private static final int INDEX_COLUMN_ORDER = 5;
    private static final int INDEX_COLUMN_COLUMN_ID = 6;
    private static final int INDEX_COLUMN_FLAGS = 7;

    private static final int INDEX_COLUMN_FLAG_FORWARD = 1;

    private static final int INDEX_FLAG_UNIQUE_KEY = 0x01;
    private static final int INDEX_FLAG_UNIQUE_INDEX = 0x02;
    private static final int INDEX_FLAG_PERSISTENT = 0x04;
    private static final int INDEX_FLAG_PRIMARY_INDEX = 0x08;

    private static String SQL_SELECT_OPTIONS =
        "SELECT name, setting FROM " + TableSchema.SYS_ULDATA
        + " WHERE type = '" + TableSchema.SYS_ULDATA_OPTION
        + "' ORDER BY name"
        ;
    private static final int OPTION_NAME = 1;
    private static final int OPTION_VALUE = 2;

    // Metadata:
    private static TableArray tables = new TableArray();
    private static OptionArray options = new OptionArray();

    /**
     * Extracts the schema of a database
     */
    private static void getSchema( Connection conn ) throws ULjException
    {
    PreparedStatement stmt = conn.prepareStatement(
        SQL_SELECT_TABLE_COLS
        );
    ResultSet cursor = stmt.executeQuery();
    Table table = null;
    int last_table_id = -1;
    for( ; cursor.next(); ) {
        int table_id = cursor.getInt( TABLE_ID );
        if( table_id != last_table_id ) {
        String table_name = cursor.getString( TABLE_NAME );
        int table_flags = cursor.getInt( TABLE_FLAGS );
        table = new Table( table_id, table_name, table_flags );
        tables.append( table );
        last_table_id = table_id;
        }
        int column_id = cursor.getInt( COLUMN_ID );
        String column_name = cursor.getString( COLUMN_NAME );
        int column_flags = cursor.getInt( COLUMN_FLAGS );
        int column_domain = cursor.getInt( COLUMN_DOMAIN_TYPE );
        int column_length = cursor.getInt( COLUMN_DOMAIN_LENGTH );
        short column_default = (short)cursor.getInt( COLUMN_DEFAULT );
        String column_default_value =
cursor.getString( COLUMN_DEFAULT_VALUE );
        int filename_colid;
        if ( cursor.isNull( COLUMN_FILENAME_COLID ) ) {
        filename_colid = -1;
        } else {
        filename_colid = cursor.getInt( COLUMN_FILENAME_COLID );
        }
        Column column = new Column(
            conn, table_id, column_id, column_name, column_flags,
            column_domain, column_length, column_default,
                column_default_value, filename_colid
            );
            table.addColumn( column );
        }
        cursor.close();
        stmt.close();

        // read indexes
        stmt = conn.prepareStatement( SQL_SELECT_INDEX_COLS );
        cursor = stmt.executeQuery();
        int last_index_id = -1;
        Index index = null;
        last_table_id = -1;
        for( ; cursor.next(); ) {
            int table_id = cursor.getInt( INDEX_TABLE_ID );
            int index_id = cursor.getInt( INDEX_ID );
            if( last_table_id != table_id || last_index_id != index_id ) {
            String index_name = cursor.getString( INDEX_NAME );
            int index_flags = cursor.getInt( INDEX_FLAGS );
            index = new Index( index_id, index_name, index_flags );
            table = findTable( table_id );
            table.addIndex( index );
            last_index_id = index_id;
            last_table_id = table_id;
            }
            int order = cursor.getInt( INDEX_COLUMN_ORDER );
            int column_id = cursor.getInt( INDEX_COLUMN_COLUMN_ID );
            int index_column_flags = cursor.getInt( INDEX_COLUMN_FLAGS );
            IndexColumn index_column = new IndexColumn( order, column_id,
    index_column_flags );
            index.addColumn( index_column );
        }
        cursor.close();
        stmt.close();

        // read database options
        stmt = conn.prepareStatement( SQL_SELECT_OPTIONS );
        cursor = stmt.executeQuery();
        for( ; cursor.next(); ) {
            String option_name = cursor.getString( OPTION_NAME );
            String option_value = cursor.getString( OPTION_VALUE );
            Option option = new Option( option_name, option_value );
            options.append( option );
        }
        cursor.close();
        stmt.close();
        }

        /** Dump the schema of a database
         */
        private static void dumpSchema( Connection conn ) throws ULjException
        {
        // Display the metadata options
        Demo.display( "\nMetadata options:\n" );
        for( int opt_no = 0; opt_no < options.count(); ++ opt_no ) {
            Option option = options.elementAt( opt_no );
            option.display();
        }
        // Display the metadata tables
        Demo.display( "\nMetadata tables:" );
        for( int table_no = 0; table_no < tables.count(); ++ table_no ) {
            Table table = tables.elementAt( table_no );
            table.display( table_no );
        }
```

```
// Display the rows for non-system tables.
for( int table_no = 0; table_no < tables.count(); ++ table_no ) {
    Table table = tables.elementAt( table_no );
    if( 0 == ( table.getFlags() & TABLE_FLAG_SYSTEM ) ) {
    Demo.display( "\nRows for table: ", table.getName(), "\n" );
    Index index = table.getIndex( 0 );
    PreparedStatement stmt = conn.prepareStatement(
        "SELECT * FROM \"" + table.getName() + "\""
        );
    ResultSet cursor = stmt.executeQuery();
    int column_count = table.getColumnCount();
    int row_count = 0;
    for( ; cursor.next(); ) {
        StringBuffer buf = new StringBuffer();
        buf.append( "Row[" );
        buf.append( Integer.toString( ++row_count ) );
        buf.append( "]:" );
        char joiner = ' ';
        for( int col_no = 1; col_no <= column_count; ++col_no ) {
        String value = cursor.isNull( col_no )
                 ? "<NULL>"
                 : cursor.getString( col_no );
        buf.append( joiner );
        buf.append( value );
        joiner = ',';
        }
        Demo.display( buf.toString() );
    }
    cursor.close();
    stmt.close();
    }
}
}

/** Find a table.
 */
private static Table findTable( int table_id )
{
Table retn = null;
for( int i = tables.count(); i > 0; ) {
    Table table = tables.elementAt( --i );
    if( table_id == table.getId() ) {
    retn = table;
    break;
    }
}
return retn;
}

/** Representation of a column.
 */
private static class Column
{
private int _table_id;
private int _column_id;
private String _column_name;
private int _column_flags;
private int _column_domain;
private short _column_scale;
private short _column_precision;
private int _column_length;
private short _column_default;
private String _column_default_value;
private int _filename_colid;
```

```
    Column( Connection conn, int table_id, int column_id, String column_name,
        int column_flags, int column_domain, int column_length, short
column_default,
        String column_default_value, int filename_colid )
        throws ULjException
    {
        _column_id = column_id;
        _column_name = column_name;
        _column_flags = column_flags;
        _column_default = column_default;
        _column_domain = column_domain;
        switch( column_domain ) {
          case Domain.NUMERIC :
        _column_scale = (short)(column_length & 255);
        _column_precision = (short)(column_length >> 8);
        break;
          case Domain.VARCHAR :
          case Domain.BINARY :
        _column_length = column_length;
        break;
          default :
        break;
        }
        _column_default_value = column_default_value;
        _filename_colid = filename_colid;
    }

    /** Corrects inconsistencies in type names between
     *  Ultralite and UltraliteJ. Returns names used in
     *  Ultralite.
     */
    static String getDomainName( int dom_type )
    {
        String name;
        switch( dom_type ) {
        default :
            name = null;
            break;
        case Domain.BIG:
            name = "bigint";
            break;
        case Domain.BINARY:
            name = "binary";
            break;
        case Domain.BIT:
            name = "bit";
            break;
        case Domain.DATE:
            name = "date";
            break;
        case Domain.DOUBLE:
            name = "double";
            break;
        case Domain.INTEGER:
            name = "integer";
            break;
        case Domain.LONGBINARY:
            name = "long binary";
            break;
        case Domain.LONGVARCHAR:
            name = "long varchar";
            break;
        case Domain.NUMERIC:
                name = "numeric";
                break;
            case Domain.REAL:
                name = "real";
                break;
            case Domain.SHORT:
                name = "smallint";
                break;
            case Domain.ST_GEOMETRY:
                name = "ST_GEOMETRY";
                break;
            case Domain.TIME:
                name = "time";
                break;
            case Domain.TIMESTAMP:
                name = "timestamp";
                break;
            case Domain.TIMESTAMP_ZONE:
                name = "timestamp with time zone";
                break;
            case Domain.TINY:
                name = "tiny";
                break;
            case Domain.UNSIGNED_BIG:
                name = "unsigned bigint";
                break;
            case Domain.UNSIGNED_INTEGER:
                name = "unsigned integer";
                break;
            case Domain.UNSIGNED_SHORT:
                name = "unsigned smallint";
                break;
            case Domain.UUID:
                name = "uniqueidentifier";
                break;
            case Domain.VARCHAR:
                name = "varchar";
                break;
            case Domain.LONGBINARYFILE:
                name = "long binary store as file";
                break;
            }
            return name;
        }

        String getName()
        {
            return _column_name;
        }

        void display()
        {
            StringBuffer buf = new StringBuffer();
            buf.append( "  \"" );
            buf.append( _column_name );
            buf.append( "\"\t" );
            buf.append( getDomainName( _column_domain ) );
            switch( _column_domain ) {
              case Domain.NUMERIC :
                buf.append( '(' );
            buf.append( Integer.toString( _column_precision ) );
            buf.append( ',' );
            buf.append( Integer.toString( _column_scale ) );
            buf.append( ')' );
        break;
          case Domain.VARCHAR :
          case Domain.BINARY :
          case Domain.ST_GEOMETRY:
            buf.append( '(' );
        buf.append( Integer.toString( _column_length ) );
        buf.append( ')' );
        break;
          case Domain.LONGBINARYFILE:
        String referenced_column =
findTable( _table_id ).getColumn( _filename_colid ).getName();
        buf.append( " STORE AS FILE (" );
        buf.append( referenced_column );
        buf.append( ')' );
        if ( ( _column_flags & COLUMN_FLAG_IS_CASCADE_DELETE ) != 0 ) {
            buf.append( " CASCADE DELETE" );
        }
        break;
        }
        if( 0 != ( _column_flags & COLUMN_FLAG_IS_NULLABLE ) ) {
        buf.append( " NULL" );
        } else {
        buf.append( " NOT NULL" );
        }
        switch( _column_default ) {
        case ColumnSchema.COLUMN_DEFAULT_NONE:
        default:
            break;
        case ColumnSchema.COLUMN_DEFAULT_AUTOINC:
            buf.append( " DEFAULT AUTOINCREMENT" );
            break;
        case ColumnSchema.COLUMN_DEFAULT_GLOBAL_AUTOINC:
            buf.append( " DEFAULT GLOBAL AUTOINCREMENT" );
            break;
        case ColumnSchema.COLUMN_DEFAULT_CURRENT_DATE:
            buf.append( " DEFAULT CURRENT DATE" );
            break;
        case ColumnSchema.COLUMN_DEFAULT_CURRENT_TIME:
            buf.append( " DEFAULT CURRENT TIME" );
            break;
        case ColumnSchema.COLUMN_DEFAULT_CURRENT_TIMESTAMP:
            buf.append( " DEFAULT CURRENT TIMESTAMP" );
            break;
        case ColumnSchema.COLUMN_DEFAULT_UNIQUE_ID:
            buf.append( " DEFAULT NEWID()" );
            break;
        case ColumnSchema.COLUMN_DEFAULT_CONSTANT:
            buf.append( " DEFAULT " );
            buf.append( _column_default_value );
        case ColumnSchema.COLUMN_DEFAULT_AUTOFILENAME:
            buf.append( " DEFAULT AUTOFILENAME ( '" );
            int index = _column_default_value.indexOf( '|' );
            String prefix = _column_default_value.substring( 0, index );
            String extension = _column_default_value.substring( index + 1 );

buf.append( '\'' ).append( prefix ).append( "','" ).append( extension ).appen
d( '\'' );
            buf.append( ')' );
            break;
        }
        buf.append( ", /* column_id=" );
        buf.append( Integer.toString( _column_id ) );
        buf.append( " column_flags=" );
        int c = 0;
```

```
            if( 0 != ( _column_flags & COLUMN_FLAG_IN_PRIMARY_INDEX ) ) {
            buf.append( "IN_PRIMARY_INDEX" );
            c++;
            }
            if( 0 != ( _column_flags & COLUMN_FLAG_IS_NULLABLE ) ) {
            if( c > 0 ) {
                buf.append( "," );
            }
            buf.append( "NULLABLE" );
            }
            buf.append( " */" );
            Demo.display( buf.toString() );
        }
        }

        /** Representation of Index schema.
         */
        private static class Index
        {
        private String _index_name;
        private int _index_id;
        private int _index_flags;
        private IndexColumnArray _columns;

        Index( int index_id, String index_name, int index_flags )
        {
            _index_id = index_id;
            _index_name = index_name;
            _index_flags = index_flags;
            _columns = new IndexColumnArray();
        }

        void addColumn( IndexColumn column )
        {
            _columns.append( column );
        }

        void display( Table table, boolean constraints )
        {
            StringBuffer buf = new StringBuffer();
            String flags = "";
            String indent = "  ";
            if( 0 != ( _index_flags & INDEX_FLAG_PRIMARY_INDEX ) ) {
            if( !constraints )    return;
            buf.append( "  CONSTRAINT \"" );
            buf.append( _index_name );
            buf.append( "\" PRIMARY KEY (" );
            flags = "PRIMARY_KEY,UNIQUE_KEY";
            } else if( 0 != ( _index_flags & INDEX_FLAG_UNIQUE_KEY ) ) {
            if( !constraints )    return;
            buf.append( "  CONSTRAINT \"" );
            buf.append( _index_name );
            buf.append( "\" UNIQUE (" );
            flags = "UNIQUE_KEY";
            } else {
            if( constraints )    return;
            if( 0 != ( _index_flags & INDEX_FLAG_UNIQUE_INDEX ) ) {
                buf.append( "UNIQUE " );
                flags = "UNIQUE_INDEX";
            }
            indent = "";
            buf.append( "INDEX \"" );
            buf.append( _index_name );
            buf.append( "\" ON \"" );
            buf.append( table.getName() );
            buf.append( "\" (" );
            }
            buf.append( "\n" );
            buf.append( indent );
            buf.append( "  /* index_id=" );
            buf.append( Integer.toString( _index_id ) );
            buf.append( " index_flags=" );
            buf.append( flags );
            if( 0 != ( _index_flags & INDEX_FLAG_PERSISTENT ) ) {
            buf.append( ",PERSISTENT" );
            }
            buf.append( " */" );
            Demo.display( buf.toString() );
            int bounds = _columns.count();
            for( int col_no = 0; col_no < bounds; ++ col_no ) {
            IndexColumn column = _columns.elementAt( col_no );
            column.display( table, indent, col_no + 1 < bounds );
            }
            Demo.display( indent + ")" );
        }

        String getName()
        {
            return _index_name;
        }
        }

        /** Representation of IndexColumn schema.
         */
        private static class IndexColumn
        {
        private int _index_column_id;
        private int _index_column_column_id;
        private int _index_column_flags;

        IndexColumn( int index_column_id, int index_column_column_id, int
    index_column_flags )
        {
            _index_column_id = index_column_id;
            _index_column_column_id = index_column_column_id;
            _index_column_flags = index_column_flags;
        }

        void display( Table table, String indent, boolean notlast )
        {
            StringBuffer buf = new StringBuffer( indent );
            buf.append( "  \"" );
            Column column = table.getColumn( _index_column_column_id );
            buf.append( column.getName() );
            if( 0 != ( _index_column_flags & INDEX_COLUMN_FLAG_FORWARD ) ) {
            buf.append( "\" ASC" );
            } else {
            buf.append( "\" DESC" );
            }
            if( notlast ) {
            buf.append( "," );
            }
            Demo.display( buf.toString() );
        }
        }

        /** Representation of a database Option.
         */
```

```
private static class Option
{
private String _option_name;
private String _option_value;

Option( String name, String value )
{
    _option_name = name;
    _option_value = value;
}

void display()
{
    StringBuffer buf = new StringBuffer();
    buf.append( "Option[ " );
    buf.append( _option_name );
    buf.append( " ] = '" );
    buf.append( _option_value );
    buf.append( "'" );
    Demo.display( buf.toString() );
}

}

/** Representation of Table schema.
 */
private static class Table
{
private String _table_name;
private int _table_id;
private int _table_flags;
private ColumnArray _columns;
private IndexArray _indexes;

Table( int table_id, String table_name, int table_flags )
{
    _table_name = table_name;
    _table_id = table_id;
    _table_flags = table_flags;
    _columns = new ColumnArray();
    _indexes = new IndexArray();
}

void addColumn( Column column )
{
    _columns.append( column );
}

void addIndex( Index index )
{
    _indexes.append( index );
}

Column getColumn( int id )
{
    return _columns.elementAt( id );
}

int getColumnCount()
{
    return _columns.count();
}

int getFlags()
{
    return _table_flags;
}

Index getIndex( int id )
{
    return _indexes.elementAt( id );
}

String getName()
{
    return _table_name;
}

void display( int logical_number )
{
    StringBuffer str = new StringBuffer();
    str.append( "\nTABLE \"" );
    str.append( _table_name );
    str.append( "\" /* table_id=" );
    str.append( Integer.toString( _table_id ) );
    str.append( " table_flags=" );
    if( 0 == _table_flags ) {
    str.append( "0" );
    } else {
    int c = 0;
    if( 0 != ( _table_flags & TABLE_FLAG_SYSTEM ) ) {
        str.append( "SYSTEM" );
        c++;
    }
    if( 0 != ( _table_flags & TABLE_FLAG_NO_SYNC ) ) {
        if( c > 0 ) {
        str.append( "," );
        }
        str.append( "NO_SYNC" );
        c++;
    }
    }
    str.append( " */ (" );
    Demo.display( str.toString() );
    int bound = _columns.count();
    for( int col_no = 0; col_no < bound; ++col_no ) {
    Column column = _columns.elementAt( col_no );
    column.display();
    }
    bound = _indexes.count();
    for( int idx_no = 0; idx_no < bound; ++idx_no ) {
    Index index = _indexes.elementAt( idx_no );
    index.display( this, true );
    }
    Demo.display( ")" );
    for( int idx_no = 0; idx_no < bound; ++idx_no ) {
    Index index = _indexes.elementAt( idx_no );
    index.display( this, false );
    }
}

int getId()
{
    return _table_id;
}
}

/** Simple adjustable array of objects.
```

```
 */
private static class ObjArray
{
private Object[] _array = new Object[ 10 ];
private int _used = 0;

void append( Object str )
{
    if( _used >= _array.length ) {
    Object[] new_array = new Object[ _used * 2 ];
    for( int i = _used; i > 0; ) {
        --i;
        new_array[ i ] = _array[ i ];
    }
    _array = new_array;
    }
    _array[ _used++ ] = str;
}

int count()
{
    return _used;
}

Object getElementAt( int position )
{
    return _array[ position ];
}
}

/** Simple adjustable array of Strings.
 */
private static class StrArray
    extends ObjArray
{
String elementAt( int position )
{
    return (String)getElementAt( position );
}
}

/** Simple adjustable array of Table objects.
 */
private static class TableArray
    extends ObjArray
{
Table elementAt( int position )
{
    return (Table)getElementAt( position );
}
}

/** Simple adjustable array of Column objects.
 */
private static class ColumnArray
    extends ObjArray
{
Column elementAt( int position )
{
    return (Column)getElementAt( position );
}
}

/** Simple adjustable array of Index objects.
```

```
         */
        private static class IndexArray
            extends ObjArray
        {
        Index elementAt( int position )
        {
            return (Index)getElementAt( position );
        }
        }

        /** Simple adjustable array of IndexColumn objects.
         */
        private static class IndexColumnArray
            extends ObjArray
        {
        IndexColumn elementAt( int position )
        {
            return (IndexColumn)getElementAt( position );
        }
        }

        /** Simple adjustable array of Option objects.
         */
        private static class OptionArray
            extends ObjArray
        {
        Option elementAt( int position )
        {
            return (Option)getElementAt( position );
        }
        }

    }
```

The partial output of the application is shown below.

```
Metadata options:

Option[ date_format ] = 'YYYY-MM-DD'
Option[ date_order ] = 'YMD'
Option[ global_database_id ] = '0'
Option[ nearest_century ] = '50'
Option[ precision ] = '30'
Option[ scale ] = '6'
Option[ time_format ] = 'HH:NN:SS.SSS'
Option[ timestamp_format ] = 'YYYY-MM-DD HH:NN:SS.SSS'
Option[ timestamp_increment ] = '1'

Metadata tables:

Table[0] name = "systable"  id = 0 flags = 0xc000,SYSTEM,NO_SYNC
  column[0 ]: name = "table_id" flags = 0x1,IN-PRIMARY-INDEX domain = INTEGER
  column[1 ]: name = "table_name" flags = 0x0 domain = VARCHAR(128)
  column[2 ]: name = "table_flags" flags = 0x0 domain = UNSIGNED-SHORT
  column[3 ]: name = "table_data" flags = 0x0 domain = INTEGER
  column[4 ]: name = "table_autoinc" flags = 0x0 domain = BIG
  index[0 ]: name = "primary" flags = 0xf,UNIQUE-KEY,UNIQUE-
INDEX,PERSISTENT,PRIMARY-INDEX
    key[0 ]: name = "table_id" flags = 0x1,FORWARD

Table[1] name = "syscolumn"  id = 1 flags = 0xc000,SYSTEM,NO_SYNC
  column[0 ]: name = "table_id" flags = 0x1,IN-PRIMARY-INDEX domain = INTEGER
  column[1 ]: name = "column_id" flags = 0x1,IN-PRIMARY-INDEX domain =
INTEGER
  column[2 ]: name = "column_name" flags = 0x0 domain = VARCHAR(128)
  column[3 ]: name = "column_flags" flags = 0x0 domain = TINY
  column[4 ]: name = "column_domain" flags = 0x0 domain = TINY
  column[5 ]: name = "column_length" flags = 0x0 domain = INTEGER
  column[6 ]: name = "column_default" flags = 0x0 domain = TINY
  index[0 ]: name = "primary" flags = 0xf,UNIQUE-KEY,UNIQUE-
INDEX,PERSISTENT,PRIMARY-INDEX
    key[0 ]: name = "table_id" flags = 0x1,FORWARD
    key[1 ]: name = "column_id" flags = 0x1,FORWARD

Table[2] name = "sysindex"  id = 2 flags = 0xc000,SYSTEM,NO_SYNC
  column[0 ]: name = "table_id" flags = 0x1,IN-PRIMARY-INDEX domain = INTEGER
  column[1 ]: name = "index_id" flags = 0x1,IN-PRIMARY-INDEX domain = INTEGER
  column[2 ]: name = "index_name" flags = 0x0 domain = VARCHAR(128)
  column[3 ]: name = "index_flags" flags = 0x0 domain = TINY
  column[4 ]: name = "index_data" flags = 0x0 domain = INTEGER
  index[0 ]: name = "primary" flags = 0xf,UNIQUE-KEY,UNIQUE-
INDEX,PERSISTENT,PRIMARY-INDEX
    key[0 ]: name = "table_id" flags = 0x1,FORWARD
    key[1 ]: name = "index_id" flags = 0x1,FORWARD

Table[3] name = "sysindexcolumn"  id = 3 flags = 0xc000,SYSTEM,NO_SYNC
  column[0 ]: name = "table_id" flags = 0x1,IN-PRIMARY-INDEX domain = INTEGER
  column[1 ]: name = "index_id" flags = 0x1,IN-PRIMARY-INDEX domain = INTEGER
  column[2 ]: name = "order" flags = 0x1,IN-PRIMARY-INDEX domain = INTEGER
  column[3 ]: name = "column_id" flags = 0x0 domain = INTEGER
  column[4 ]: name = "index_column_flags" flags = 0x0 domain = TINY
  index[0 ]: name = "primary" flags = 0xf,UNIQUE-KEY,UNIQUE-
INDEX,PERSISTENT,PRIMARY-INDEX
    key[0 ]: name = "table_id" flags = 0x1,FORWARD
    key[1 ]: name = "index_id" flags = 0x1,FORWARD
    key[2 ]: name = "order" flags = 0x1,FORWARD

Table[4] name = "sysinternal"  id = 4 flags = 0xc000,SYSTEM,NO_SYNC
  column[0 ]: name = "name" flags = 0x1,IN-PRIMARY-INDEX domain =
VARCHAR(128)
  column[1 ]: name = "value" flags = 0x0 domain = VARCHAR(128)
  index[0 ]: name = "primary" flags = 0xf,UNIQUE-KEY,UNIQUE-
INDEX,PERSISTENT,PRIMARY-INDEX
    key[0 ]: name = "name" flags = 0x1,FORWARD

Table[5] name = "syspublications"  id = 5 flags = 0xc000,SYSTEM,NO_SYNC
  column[0 ]: name = "publication_id" flags = 0x1,IN-PRIMARY-INDEX domain =
INTEGER
  column[1 ]: name = "publication_name" flags = 0x0 domain = VARCHAR(128)
  column[2 ]: name = "download_timestamp" flags = 0x0 domain = TIMESTAMP
  column[3 ]: name = "last_sync_sent" flags = 0x0 domain = INTEGER
  column[4 ]: name = "last_sync_confirmed" flags = 0x0 domain = INTEGER
  index[0 ]: name = "primary" flags = 0xf,UNIQUE-KEY,UNIQUE-
INDEX,PERSISTENT,PRIMARY-INDEX
    key[0 ]: name = "publication_id" flags = 0x1,FORWARD

Table[6] name = "sysarticles"  id = 6 flags = 0xc000,SYSTEM,NO_SYNC
  column[0 ]: name = "publication_id" flags = 0x1,IN-PRIMARY-INDEX domain =
INTEGER
  column[1 ]: name = "table_id" flags = 0x1,IN-PRIMARY-INDEX domain = INTEGER
  index[0 ]: name = "primary" flags = 0xf,UNIQUE-KEY,UNIQUE-
INDEX,PERSISTENT,PRIMARY-INDEX
    key[0 ]: name = "publication_id" flags = 0x1,FORWARD
    key[1 ]: name = "table_id" flags = 0x1,FORWARD

Table[7] name = "sysforeignkey"  id = 7 flags = 0xc000,SYSTEM,NO_SYNC
  column[0 ]: name = "table_id" flags = 0x1,IN-PRIMARY-INDEX domain = INTEGER
  column[1 ]: name = "foreign_table_id" flags = 0x0 domain = INTEGER
  column[2 ]: name = "foreign_key_id" flags = 0x1,IN-PRIMARY-INDEX domain =
INTEGER
  column[3 ]: name = "name" flags = 0x0 domain = VARCHAR(128)
  column[4 ]: name = "index_name" flags = 0x0 domain = VARCHAR(128)
  index[0 ]: name = "primary" flags = 0xf,UNIQUE-KEY,UNIQUE-
INDEX,PERSISTENT,PRIMARY-INDEX
    key[0 ]: name = "table_id" flags = 0x1,FORWARD
    key[1 ]: name = "foreign_key_id" flags = 0x1,FORWARD

Table[8] name = "sysfkcol"  id = 8 flags = 0xc000,SYSTEM,NO_SYNC
  column[0 ]: name = "table_id" flags = 0x1,IN-PRIMARY-INDEX domain = INTEGER
  column[1 ]: name = "foreign_key_id" flags = 0x1,IN-PRIMARY-INDEX domain =
INTEGER
  column[2 ]: name = "item_no" flags = 0x1,IN-PRIMARY-INDEX domain = SHORT
  column[3 ]: name = "column_id" flags = 0x0 domain = INTEGER
  column[4 ]: name = "foreign_column_id" flags = 0x0 domain = INTEGER
  index[0 ]: name = "primary" flags = 0xf,UNIQUE-KEY,UNIQUE-
INDEX,PERSISTENT,PRIMARY-INDEX
    key[0 ]: name = "table_id" flags = 0x1,FORWARD
    key[1 ]: name = "foreign_key_id" flags = 0x1,FORWARD
    key[2 ]: name = "item_no" flags = 0x1,FORWARD
```

Example: Synchronizing a database

This example demonstrates connecting to a MobiLink server to synchronize the client database with a consolidated (in this case, SQL Anywhere) database.

To run the Sync.java example

1. Change to the following directory: *samples-dir\UltraLiteJ*.

   For information about the default location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

2. Run the CreateSales example:

   ```
   rundemo CreateSales
   ```

   See "Example: Creating a sales database" on page 36.

3. Run the following command to start the MobiLink server:

   ```
   start_ml
   ```

4. Run the following command (the command is case sensitive):

   ```
   rundemo Sync
   ```

5. Once the example completes, shut down the MobiLink server. See "Stopping the MobiLink server" [*MobiLink - Server Administration*].

```
// *******************************************************
// Copyright (c) 2006-2010 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2010 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
```

```
// *****************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original iAnywhere code.
//
// *********************************************************************
package com.ianywhere.ultralitej.demo;

import com.ianywhere.ultralitej12.*;

/**
 * Sync: sample program to demonstrate Database synchronization.
 *
 * Requires starting the MobiLink Server Sample using start_ml.bat
 */
public class Sync
{
    /**
     * mainline for program.
     *
     * @param args command-line arguments
     *
     */
    public static void main
    ( String[] args )
    {
    try {
        Configuration config =
DatabaseManager.createConfigurationFile( "Demo1.ulj" );
        Connection conn = DatabaseManager.createDatabase( config );

        PreparedStatement ps = conn.prepareStatement(
            "CREATE TABLE ULCustomer" +
        "( cust_id int NOT NULL PRIMARY KEY" +
        ", cust_name VARCHAR(30) NOT NULL" +
        ")"
        );
        ps.execute();
        ps.close();

        //
        // Synchronization
        //

        // Version set for MobiLink 12.0.x
        SyncParms syncParms = conn.createSyncParms( SyncParms.HTTP_STREAM,
"50", "custdb 12.0" );
        syncParms.getStreamParms().setPort( 9393 );
        conn.synchronize( syncParms );
        SyncResult result = syncParms.getSyncResult();
        Demo.display(
            "*** Synchronized *** bytes sent=" + result.getSentByteCount()
            + ", bytes received=" + result.getReceivedByteCount()
            + ", rows received=" + result.getReceivedRowCount()
        );

        conn.release();

    } catch( ULjException exc ) {
        Demo.displayException( exc );
    }
```

```
        }
    }
```

Tutorial: Building an UltraLiteJ application for BlackBerry

Introduction to BlackBerry development

This tutorial guides you through the development of an UltraLiteJ application for BlackBerry smartphones using the Research In Motion BlackBerry Java Development Environment. In this tutorial, you run the application on a BlackBerry simulator and deploy the application to a physical device. Code samples are provided throughout the tutorial, and a complete listing of the code is available at the end of the tutorial.

The tutorial assumes the following:

- You are familiar with the Java programming language.
- You have Research In Motion BlackBerry JDE 4.2 or later installed on your computer.
- You have UltraLiteJ installed on your computer. The default installation location for UltraLiteJ is *install-dir\UltraLite\UltraLiteJ* .
- You have the Research In Motion MDS Services Simulator installed on your computer. This is required for part 2.

Part 1: Create a new BlackBerry UltraLiteJ application for BlackBerry

This part explains how to create a BlackBerry application that maintains a list of names in a simple UltraLiteJ database. The second part explains how to synchronize the application with a MobiLink server.

Lesson 1: Create a BlackBerry JDE project

In this lesson, you create a new BlackBerry Java Development Environment (JDE) project.

1. Launch the IDE the **Start** menu.

2. From the JDE **File** menu, choose **New Workspace**.

3. Choose a location for your workspace; for example, *c:\tutorials*. Name the workspace **HelloBlackBerry** and click **OK**.

4. In this tutorial, the workspace contains a single project. From the **Project** menu, choose **Create New Project**.

5. Name the project **HelloBlackBerry** and click **OK**.

6. Add the UltraLiteJ JAR file to the project.

    a. In the **Files** window, right-click the project and choose **Properties**.

    b. On the **Build** tab, click **Add**, which is located next to the **Imported Jar Files** field.

    c. Browse to the *UltraLiteJ\BlackBerry4.2\UltraLiteJ12.jar* file in your UltraLiteJ installation and click **Open**.

    d. Click **OK** to close the **Properties** window.

7. From the **Project** menu, choose **Set Active Projects**. Select the HelloBlackBerry project and click **OK**.

8. Save the project.

Lesson 2: Display a BlackBerry application screen

In this lesson, you create a class with a main method that opens a HomeScreen, which contains a title and a status message.

1. In the workspace view, right-click the project and choose **Create New File In Project**.

2. In the **Source File Name** box, type **myapp\Application.java** to create a file named *Application.java* that is part of the myapp package.

    Click **OK** to create the file. The Application class appears in the JDE window.

3. Define the Application class. No imports are needed for this class. Add a constructor and a main method so that your Application class is defined as follows:

```
class Application extends net.rim.device.api.ui.UiApplication {
    public static void main( String[] args )
    {
        Application instance = new Application();
        instance.enterEventDispatcher();
    }
    Application() {
        pushScreen( new HomeScreen() );
    }
}
```

4. Add the HomeScreen class to your project.

    a. In the workspace view, right-click the project and choose **Create New File In Project**.

    b. In the **Source File Name** box, type **myapp\HomeScreen.java**.

    c. Click **OK** to create the file.

        The HomeScreen class appears in the JDE window.

5. Define the HomeScreen class so that it displays a title and a Status message.

   ```
   package myapp;
   import net.rim.device.api.ui.*;
   import net.rim.device.api.ui.component.*;
   import net.rim.device.api.ui.container.*;
   import java.util.*;

   class HomeScreen extends MainScreen {

       HomeScreen() {

           // Set the window title
           LabelField applicationTitle = new LabelField("Hello BlackBerry");
           setTitle(applicationTitle);

           // Add a label to show application status
           _statusLabel = new LabelField( "Status: Started" );
           add( _statusLabel );
       }
       private LabelField _statusLabel;
   }
   ```

   The _statusLabel is defined as a class variable so that it can be accessed from other parts of the application later.

6. Right-click the project and choose **Build**. Ensure that it compiles without errors.

7. Press F5 to run the application in the device simulator.

   The BlackBerry simulator launches in a separate window.

8. On the simulator, navigate to the **Applications** window and select the HelloBlackBerry application.

9. Start the application.

   A window appears showing the title bar **Hello BlackBerry** and the status line **Status: started**.

10. From the JDE **Debug** menu choose **Stop Debugging**.

    The simulator terminates.

Lesson 3: Create an UltraLiteJ database

In this lesson, you write code to create and connect to an UltraLiteJ database. The code to create a new database is defined in a singleton class called DataAccess, and is invoked from the HomeScreen constructor. Using a singleton class ensures that only one database connection is open at a time. While UltraLiteJ supports multiple connections, it is a common design pattern to use a single connection.

1. Modify the HomeScreen constructor to instantiate a DataAccess object.

   Here is a complete, updated HomeScreen class. The DataAccess object is held as a class-level variable, so that it can be accessed from other parts of the code.

```
class HomeScreen extends MainScreen {

    HomeScreen() {

        // Set the window title
        LabelField applicationTitle = new LabelField("Hello BlackBerry");
        setTitle(applicationTitle);

        // Add a label to show application status
        _statusLabel = new LabelField( "Status: Started");
        add( _statusLabel );

        // Create database and connect
        try{
            _da = DataAccess.getDataAccess(true);
            _statusLabel.setText("Status: Connected");
        }
        catch( Exception ex)
        {
            _statusLabel.setText("Exception: " + ex.toString() );
        }
    }
    private LabelField _statusLabel;
    private DataAccess _da;
}
```

2. Create a file named *myapp\DataAccess.java* in the HelloBlackBerry project.

3. Provide a getDataAccess method that ensures a single database connection.

```
package myapp;

import com.ianywhere.ultralitej12.*;
import java.util.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

class DataAccess {
    DataAccess() {    }

    public static synchronized DataAccess getDataAccess(boolean reset)
                  throws Exception
    {
        if( _da == null ){
            _da = new DataAccess();
            ConfigObjectStore config =
             DatabaseManager.createConfigurationObjectStore("HelloDB");
            if(reset)
            {
                _conn = DatabaseManager.createDatabase( config );
                // _da.createDatabaseSchema();
            }
            else
            {
                try{
                    _conn = DatabaseManager.connect( _config );

                }
                catch( ULjException uex1) {
                    if( uex1.getErrorCode() !=
                        ULjException.SQLE_ULTRALITE_DATABASE_NOT_FOUND ) {
                        Dialog.alert( "Exception: " + uex1.toString() +
                                  ". Recreating database..." );
                        }
                        _conn = DatabaseManager.createDatabase( _config );
                        _da.createDatabaseSchema();
                    }
                }
            }
            return _da;
        }
        private static Connection _conn;
        private static DataAccess _da;
    }
```

This class imports the com.ianywhere.ultralitej12 package from the *UltraLiteJ12.jar* file. The steps to create or connect to a database are:

a. Define a configuration. In this example, it is a ConfigObjectStore configuration object, meaning that the UltraLiteJ database is persisted in the BlackBerry object store.

b. Attempt to connect to the database.

   If the connection attempt fails, create the database. The createDatabase method then returns an open connection.

4. Press F5 to build and deploy the application to the device simulator.

5. From the **File** menu, choose **Load Java Program**.

6. Browse to the *BlackBerry4.2* folder of your UltraLiteJ installation and open the *UltraLiteJ12.cod* file.

7. Run the program from the Simulator.

   You should see a status message indicating that the application successfully connected to the database.

Lesson 4: Create a database table

In this lesson, you create a simple table called Names, which contains two columns that have the following properties:

Column name	Data type	Allow null?	Default	Primary key?
ID	UUID	No	None	Yes
Name	Varchar(254)	No	None	No

1. Add a DataAccess method to create the table.

```
private void createDatabaseSchema()
{
    try{
        String sql = "CREATE TABLE Names ( ID UNIQUEIDENTIFIER DEFAULT
NEWID(), Name VARCHAR(254), " +
            "PRIMARY KEY ( ID ) )";
            PreparedStatement ps = _conn.prepareStatement(sql);
            ps.execute();
            ps.close();
        }
        catch( ULjException uex1){
            Dialog.alert( "ULjException: " + uex1.toString() );
        }
        catch( Exception ex1){
            Dialog.alert( "Exception: " + ex1.toString() );
        }
    }
```

If the table already exists, an exception is thrown.

2. Call the DataAccess.getDataAccess method.

   Uncomment the call to createDatabaseSchema in the sample code from Part 1, Lesson 3, Step 3. The call to createDatabaseSchema should look like the following:

   ```
   _da.createDatabaseSchema()
   ```

3. Run the application again in the simulator.

Lesson 5: Add data to the table

In this lesson, you add the following controls to the screen:

- a text field in which you can enter a name.
- a menu item to add the name in the text field to the database.
- a list field that displays the names in the table.

You then add code to insert the name in the text field and refresh the list.

1. Add the controls to the screen.

   a. Change the call to getDataAccess to false as follows:

      ```
      _da = DataAccess.getDataAccess(false);
      ```

   b. In the HomeScreen class, add the following code before calling the getDataAccess method.

      ```
      // Add an edit field for entering new names
      _nameEditField = new EditField( "Name: ", "", 50,
      EditField.USE_ALL_WIDTH );
      add (_nameEditField );

      // Add an ObjectListField for displaying a list of names
      _nameListField = new ObjectListField();
      add( _nameListField );

      // Add a menu item
      addMenuItem(_addToListMenuItem);

      // Create database and connect
      try{
          _da = DataAccess.getDataAccess(false);
      ```

c. Add class-level declarations for _nameEditField and _nameListField. Also, define a _addToListMenuItem MenuItem with a run method (empty for now). These declarations belong next to the declarations of _statusLabel and _da.

```
private EditField _nameEditField;
private ObjectListField _nameListField;

private MenuItem _addToListMenuItem = new MenuItem("Add", 1, 1){
    public void run() {
        // TODO
    }
};
```

d. Recompile the application and confirm that it runs.

2. Add the following methods and objects to your application:

- A DataAccess method to insert a row into a table
- An object to hold a row of the Names table as an object
- A DataAccess method to read the rows from a table into a Vector of objects
- A method to refresh the contents of the list displayed on the HomeScreen
- A method to add an item to the list on the HomeScreen.

a. Add the DataAccess method to insert a row into a table:

```
public void insertName( String name ){
    try{
        UUID Value nameID = _conn.createUUIDValue();
        String sql = "INSERT INTO Names( ID, Name ) VALUES
                     ( ?, ? )";
        PreparedStatement ps = _conn.prepareStatement(sql);
        ps.set(1, nameID );
        ps.set(2, name );
        ps.execute();
        _conn.commit();
    }
    catch( ULjException uex ){
        Dialog.alert("ULjException: " + uex.toString());
    }
    catch( Exception ex ){
        Dialog.alert("Exception: " + ex.toString());
    }
}
```

b. Add the class that holds a row of the Names table. The toString method is used by the ObjectListField control.

```
package myapp;

class NameRow {

    public NameRow( String nameID, String name ) {
        _nameID = nameID;
        _name = name;
    }

    public String getNameID(){
        return _nameID;
    }
```

```
        public String getName(){
            return _name;
        }

        public String toString(){
            return _name;
        }

        private String _nameID;
        private String _name;

    }
```

c. Add the DataAccess method to read the rows from the table into a Vector of objects:

```
public Vector getNameVector(){
    Vector nameVector = new Vector();
    try{
        String sql = "SELECT ID, Name FROM Names";
        PreparedStatement ps = _conn.prepareStatement(sql);
        ResultSet rs = ps.executeQuery();
        while ( rs.next() ){
            String nameID = rs.getString(1);
            String name = rs.getString(2);
            NameRow nr = new NameRow( nameID, name);
            nameVector.addElement(nr);
        }
    }
    catch( ULjException uex ){
        Dialog.alert("ULjException: " + uex.toString());
    }
    catch( Exception ex ){
        Dialog.alert("Exception: " + ex.toString());
    }
    finally{
        return nameVector;
    }
}
```

d. Add the user interface methods to the HomeScreen class. Following is the method to refresh the list of names:

```
public void refreshNameList(){
    //Clear the list
    _nameListField.setSize(0);
    //Refill from the list of names
    Vector nameVector = _da.getNameVector();
    for( Enumeration e = nameVector.elements(); e.hasMoreElements(); ){
        NameRow nr = ( NameRow )e.nextElement();
        _nameListField.insert(0, nr);
    }
}
```

e. Call the refreshNameList method before the end of the HomeScreen constructor so that the list is filled when the application starts.

```
// Fill the ObjectListField
this.refreshNameList();
```

f. Add a HomeScreen method that is used to add a row to the list:

```
private void onAddToList(){
    String name = _nameEditField.getText();
    _da.insertName(name);
    this.refreshNameList();
    _nameEditField.setText("");
    _statusLabel.setText(name + " added to list");
}
```

g. Call this method from within the run method of the _addToListMenuItem MenuItem (which currently says //TODO):

```
public void run() {
    onAddToList();
}
```

3. Compile and run the application.

Resetting the simulator
If you need to reset the simulator to a clean state, choose **Erase Simulator File** from the BlackBerry JDE **File** menu (not the **Simulator** menu), and erase the items in the submenu. If you reset the simulator this way, you must re-import the *UltraLiteJ12.cod* file before running your application again.

Names are stored in the database as you add them. They are retrieved from the database and added to the list when you close and re-open the application.

Lesson 6: Deploy application to smartphone

There are several ways to deploy applications to BlackBerry smartphones. This lesson explains how to deploy the application using the BlackBerry Desktop Manager software.

Applications running on a BlackBerry must be signed using the BlackBerry Signature Tool. This tool is available from Research in Motion (RIM) as part of the BlackBerry JDE Component Package. The *UltraLiteJ12.cod* file is already signed, but you must sign the *HelloBlackBerry.cod* file.

Note
You must obtain a key from RIM so that you can use the BlackBerry Signature Tool to sign your application. For more information about obtaining keys, see the BlackBerry Developer Program web site at http://na.blackberry.com/eng/developers/.

Signing the application

1. Start the BlackBerry Signature Tool.

2. Browse to and select your compiled application, the *HelloBlackBerry.cod* file.

3. Click **Request To Sign The File**.

4. Click **Close** to close the signature tool.

Deploying the application

These steps describe how to deploy your file to the device using the BlackBerry Desktop Manager.

1. Connect your BlackBerry to your computer using the USB cable, and ensure that the Desktop Manager can see the device.
2. Click **Application Loader** and follow the instructions in the wizard.
3. Browse to the *HelloBlackBerry.alx* file and add it to your device.
4. Browse to the *BlackBerry4.2\UltraLiteJ.alx* file and add it to your device.

You should now be able to use the application on your BlackBerry smartphone.

Part 2: Synchronize the UltraLiteJ application using MobiLink

This part extends the application to handle synchronization. It involves creating a SQL Anywhere database, running a MobiLink server, and adding a synchronization function from your BlackBerry application.

Lesson 1: Create a SQL Anywhere database

Data synchronization requires a consolidated database for UltraLiteJ to synchronize with. In this lesson, you create a SQL Anywhere database.

1. In your application directory, create a subdirectory to hold the SQL Anywhere database called *c:\tutorial \database*.
2. Run the following command from *c:\tutorial\database* to create an empty SQL Anywhere database:

   ```
   dbinit HelloBlackBerry.db
   ```

3. Create an ODBC data source to connect to the database.

   a. Open the ODBC Data Source Administrator.

      Choose **Start** » **Programs** » **SQL Anywhere 12** » **Administration Tools** » **ODBC Data Source Administrator**.

   b. Click the **User DSN** tab to create an ODBC data source for the current user.

   c. Click **Add**.

   d. From the list of drivers, choose **SQL Anywhere 12**, and then click **Finish**.

   e. Click the **ODBC** tab.

   f. In the **Data Source Name** field, type **HelloBlackBerry**.

   g. Click the **Login** tab.

   h. In the **User ID** field, type **DBA** and in the **Password** field type **sql**.

      These are the default login name and password for all SQL Anywhere databases, and should not be used in a production environment.

   i. Click the **Database** tab, type **HelloBlackBerry** as the **Server Name** and *c:\tutorial\database \HelloBlackBerry.db* as the **Database File**. Click **OK**.

4. Run the following command to start Interactive SQL and connect to the SQL Anywhere database:

```
dbisql -c dsn=HelloBlackBerry
```

5. Execute the following statement to create a table in the database:

```
CREATE TABLE Names (
 ID UNIQUEIDENTIFIER NOT NULL DEFAULT newID(),
 Name varchar(254),
 PRIMARY KEY (ID)
);
```

6. Close Interactive SQL.

Lesson 2: Create MobiLink scripts

In this lesson, you prepare your consolidated database for synchronization using Sybase Central.

1. From the **Start** menu, choose **Programs** » **SQL Anywhere 12** » **Administration Tools** » **Sybase Central**.

2. In the Sybase Central **Task** pane, choose **Create A Synchronization Model** from the MobiLink 12 tasks.

   a. Type the Synchronization model name **HelloBlackBerrySyncModel**, and store the model in the *c:\tutorial\database* folder. Click **Next**.

   b. Click **Choose a Consolidated Database**.

   c. In the **Connect to Consolidated Database** window choose the HelloBlackBerry ODBC Data Source and click **OK**.

   d. Click **Yes** to create the MobiLink system setup.

   e. Choose **No, Create A New Remote Schema**.

   f. For downloads, choose **Timestamp-based Download**.

   g. Click **Finish** to complete creating the synchronization model and save the project.

3. Right-click the synchronization model and choose **Deploy**. Choose to deploy only to the consolidated database.
4. In the **Deploy Synchronization Model Wizard**, choose **Save SQL** and deploy the database. When prompted for the consolidated database, specify the ODBC data source **HelloBlackBerry**.
5. For the MobiLink user and password choose user name **mluser** with password **mlpassword**.
6. Click **Finish** to deploy the synchronization model to the consolidated database.

Lesson 3: Add synchronization to the application

In this lesson, you add synchronization capabilities to your application.

1. In the HomeScreen constructor, add a Sync menu item.

```
// Add a menu item
addMenuItem(_addToListMenuItem);

// Add sync menu item
addMenuItem(_syncMenuItem);

// Create database and connect
try{ ...
```

2. Define the menu item in the class variables declarations.

```
private MenuItem _addToListMenuItem = new MenuItem("Add", 1, 1){
    public void run() {
        onAddToList();
    }
};
private MenuItem _syncMenuItem = new MenuItem("Sync", 2, 1){
    public void run() {
        onSync();
    }
};
```

3. Create the onSync method.

```
private void onSync(){
    try{
        if( _da.sync() ){
            _statusLabel.setText("Synchronization succeeded");
        } else {
            _statusLabel.setText("Synchronization failed");
          }
        this.refreshNameList();
    } catch ( Exception ex){
        Dialog.alert( ex.toString() );
    }
}
```

4. Define the syncParms and streamParms variables within the DataAccess class.

```
private static SyncParms _syncParms;
private static StreamHTTPParms _streamParms;
```

5. In the DataAccess class, add a sync method.

```
public boolean sync() {
    try {
        if(_syncParms == null){
            String host = "relayserver.sybase.com/ias_relay_server/client/
rs_client.dll/tomslee.HelloBlackBerry";
            _syncParms = _conn.createSyncParms( "mluser",
"HelloBlackBerrySyncModel" );
            _syncParms.setPassword("mlpassword");
            _streamParms = _syncParms.getStreamParms();
            _streamParms.setPort( 8081 ); // use your own
            _streamParms.setHost( host ); // use your own
            if(host != null && _host.equals("relayserver.sybase.com/
ias_relay_server/client/rs_client.dll/tomslee.HelloBlackBerry")) {
                _streamParms.setURLSuffix("/ias_relay_server/client/
rs_client.dll/tomslee.HelloBlackBerry");
            }
        }

        public boolean complete() {
            try{
                _conn.checkpoint();
                _conn.release();
                _conn = null;
                _da = null;
                return true;
            }
            catch(Exception e){
                return false;
    }
}
```

The synchronization parameters object, SyncParms, includes the user name and password that you specified when deploying the synchronization model. It also includes the name of the synchronization model you created. In MobiLink, this name now refers to the synchronization version, or a set of synchronization logic, that was deployed to your consolidated database.

The stream parameters object, StreamHTTPParms, indicate the host name and port number of the MobiLink server. When you start the MobiLink server in the next lesson, use your own computer name for simulator testing and select a port that is available. Do not use localhost as your computer name. You can use port 80 if you have no web server running on your computer.

Note
When using a real device, use an externally visible machine or a machine that is accessible from the BlackBerry Enterprise Server your device is paired to, such as the Sybase hosted relay server. For more information on the relay server, see "Introduction to the Relay Server" [*Relay Server*].

6. Compile your application.

Lesson 4: Start the MobiLink server and run the application on the simulator

Before you can run the BlackBerry application and synchronize, the MobiLink server must be running. The MDS Simulator must also be running to provide a communications channel between the device simulator and MobiLink.

1. Start MobiLink by running the following command from *c:\tutorial\database\*:

   ```
   mlsrv12 -c " DSN=HelloBlackBerry" -v+ -x http(port=8081) -ot ml.txt
   ```

   The -c option connects MobiLink to the SQL Anywhere database. The -v+ option sets a high level of verbosity so that you can follow what is happening in the server messages window. The -x option indicates the port number being used for the communications. The -ot option specifies that a log file (*ml.txt*) is to be created in the directory where you started the MobiLink server.

2. Choose **Start** » **Programs** » **Research In Motion** » **BlackBerry Email And MDS Services Simulator 4.1.2** » **MDS**.

3. Enter names at the server.

   a. Start Interactive SQL and connect to the HelloBlackBerry data source.

   b. Run the following SQL statements add names:

      ```
      INSERT Names ( Name ) VALUES ( 'ServerName1' );
      INSERT Names ( Name ) VALUES ( 'ServerName2' );
      COMMIT;
      ```

4. From the JDE, press F5 to compile your application and run it in the device simulator.

5. Navigate to the main screen and add names to the list.

6. Synchronize the application.

7. From the main screen, display the menu items and choose **Sync**.

   The names entered at the server appear in the screen. If you query the names in the Names table from Interactive SQL, you should see that any names you have entered in the simulator have reached the server.

Code listing for tutorial

This section provides the complete code for the preceding tutorial There are four Java classes that are present in the tutorials. They are available in *samples-dir\UltraLiteJ\HelloBlackBerry*.

See also

- "Part 1: Create a new BlackBerry UltraLiteJ application for BlackBerry" on page 71
- "Part 2: Synchronize the UltraLiteJ application using MobiLink" on page 80

Application.java

```
/*
 * Application.java
 *
 * © <your company here>, 2003-2005
 * Confidential and proprietary.
 */

package myapp;


/**
 *
 */
class Application extends net.rim.device.api.ui.UiApplication {

    public static void main( String[] args )
    {
        Application instance = new Application();
        instance.enterEventDispatcher();
    }

    Application() {
        pushScreen( new HomeScreen() );
    }
}
```

DataAccess.java

```
/*
 * DataAccess.java
 *
 * © <your company here>, 2003-2005
 * Confidential and proprietary.
 */

package myapp;

import com.ianywhere.ultralitej12.*;
import java.util.*;
import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;

/**
 *
 */
class DataAccess {
    DataAccess() {    }

   public static synchronized DataAccess getDataAccess(boolean reset) throws
Exception
    {
        if( _da == null ){
            _da = new DataAccess();
            ConfigObjectStore config =
DatabaseManager.createConfigurationObjectStore("HelloDB");
            if(reset)
            {
                _conn = DatabaseManager.createDatabase( config );
                _da.createDatabaseSchema();
            }
            else
            {
                try{
                    _conn = DatabaseManager.connect( config );
                }
                catch( ULjException uex1) {
                    if( uex1.getErrorCode() !=
ULjException.SQLE_ULTRALITE_DATABASE_NOT_FOUND ) {
                        Dialog.alert( "Exception: " + uex1.toString() + ".
Recreating database..." );
                    }
                    _conn = DatabaseManager.createDatabase( config );
                    _da.createDatabaseSchema();
                }
            }
        }
        return _da;
    }

    /**
     *
     * Create the table in the database.
     * If the table already exists, a harmless exception is thrown
     */
    private void createDatabaseSchema()
    {
        try{
            String sql = "CREATE TABLE Names ( ID UNIQUEIDENTIFIER DEFAULT
NEWID(), Name VARCHAR(254), " +
                "PRIMARY KEY ( ID ) )";
            PreparedStatement ps = _conn.prepareStatement(sql);
            ps.execute();
            ps.close();
        }
        catch( ULjException uex1){
            Dialog.alert( "ULjException: " + uex1.toString() );
        }
        catch( Exception ex1){
            Dialog.alert( "Exception: " + ex1.toString() );
        }
    }

    public void insertName( String name ){
        try{
            UUIDValue nameID = _conn.createUUIDValue();
            String sql = "INSERT INTO Names( ID, Name ) VALUES
( ?, ? )";
            PreparedStatement ps = _conn.prepareStatement(sql);
            ps.set(1, nameID );
            ps.set(2, name );
            ps.execute();
            _conn.commit();
        }
        catch( ULjException uex ){
            Dialog.alert( "ULjException: " + uex.toString() );
        }
        catch( Exception ex ){
            Dialog.alert( "Exception: " + ex.toString() );
        }
    }

    public Vector getNameVector(){
        Vector nameVector = new Vector();
        try{
            String sql = "SELECT ID, Name FROM Names";
            PreparedStatement ps = _conn.prepareStatement(sql);
            ResultSet rs = ps.executeQuery();
            while ( rs.next() ){
                String nameID = rs.getString(1);
                String name = rs.getString(2);
                NameRow nr = new NameRow( nameID, name);
                nameVector.addElement(nr);
            }
        }
        catch( ULjException uex ){
            Dialog.alert( "ULjException: " + uex.toString() );
        }
        catch( Exception ex ){
            Dialog.alert( "Exception: " + ex.toString() );
        }
        finally{
            return nameVector;
        }
    }

    public boolean sync() {
        try {
            if( _syncParms == null ){
                String host = "relayserver.sybase.com/ias_relay_server/client/
rs_client.dll/tomslee.HelloBlackBerry";
                _syncParms = _conn.createSyncParms( "mluser",
"HelloBlackBerrySyncModel" );
                _syncParms.setPassword("mlpassword");
                _streamParms = _syncParms.getStreamParms();
                _streamParms.setPort( 8081 ); // use your own
                _streamParms.setHost( host ); // use your own
               if(host.equals("relayserver.sybase.com/ias_relay_server/client/
rs_client.dll/tomslee.HelloBlackBerry"))
                {
                    _streamParms.setURLSuffix("/ias_relay_server/client/
rs_client.dll/tomslee.HelloBlackBerry");
                }

            }
            _conn.synchronize( _syncParms );
            return true;
        }
        catch( ULjException uex){
            Dialog.alert(uex.toString());
            return false;
        }
    }

    public boolean complete() {
        try{
            _conn.checkpoint();
            _conn.release();
            _conn = null;
            _da = null;
```

```
                return true;
            }
            catch(Exception e){
                return false;
            }
        }

        private static Connection _conn;
        private static DataAccess _da;
        private static SyncParms _syncParms;
        private static StreamHTTPParms _streamParms;
    }
```

HomeScreen.java

```
/*
 * HomeScreen.java
 *
 * © <your company here>, 2003-2005
 * Confidential and proprietary.
 */

package myapp;

import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import java.util.*;

/**
 *
 */
class HomeScreen extends MainScreen {
    HomeScreen() {
        // Set the window title
        LabelField applicationTitle = new LabelField("Hello BlackBerry");
        setTitle(applicationTitle);

        // Add a label to show application status
        _statusLabel = new LabelField( "Status: Started");
        add( _statusLabel );

        // Add an edit field for entering new names
        _nameEditField = new EditField( "Name: ", "", 50,
EditField.USE_ALL_WIDTH );
        add (_nameEditField );

        // Add an ObjectListField for displaying a list of names
        _nameListField = new ObjectListField();
        add( _nameListField );

        // Add a menu item
        addMenuItem(_addToListMenuItem);

        // Add sync menu item
        addMenuItem(_syncMenuItem);

        // Add reset menu item
        addMenuItem(_resetMenuItem);

        // Create database and connect
```

```
            try{
                _da = DataAccess.getDataAccess(false);
                _statusLabel.setText("Status: Connected");
            }
            catch( Exception ex)
            {
                _statusLabel.setText("Exception: " + ex.toString() );
            }

            // Fill the ObjectListField
            this.refreshNameList();
        }

        public void refreshNameList(){
            try{
                //Clear the list
                _nameListField.setSize(0);
                //Refill from the list of names
                Vector nameVector = _da.getNameVector();
               for( Enumeration e = nameVector.elements(); e.hasMoreElements(); )
    {
                    NameRow nr = ( NameRow )e.nextElement();
                    _nameListField.insert(0, nr);
                }
            } catch ( Exception ex){
                Dialog.alert(ex.toString());
            }
        }

        private void onAddToList(){
            String name = _nameEditField.getText();
            _da.insertName(name);
            this.refreshNameList();
            _nameEditField.setText("");
            _statusLabel.setText(name + " added to list");
        }

        private void onSync(){
            try{
                if( _da.sync() ){
                    _statusLabel.setText("Synchronization succeeded");
                } else {
                    _statusLabel.setText("Synchronization failed");
                }
                this.refreshNameList();
            } catch ( Exception ex){
                Dialog.alert( ex.toString() );
            }
        }

        private void onReset(){
            _da.complete();
            try{
                _da = DataAccess.getDataAccess(true);
                _statusLabel.setText("Status: Connected");
                this.refreshNameList();
            }
            catch( Exception ex)
            {
                _statusLabel.setText("Exception: " + ex.toString() );
            }
        }

        private LabelField _statusLabel;
    private DataAccess _da;
    private EditField _nameEditField;
    private ObjectListField _nameListField;
    private MenuItem _addToListMenuItem = new MenuItem("Add", 1, 1){
        public void run() {
            onAddToList();
        }
    };
    private MenuItem _syncMenuItem = new MenuItem("Sync", 2, 1){
        public void run() {
            onSync();
        }
    };
    private MenuItem _resetMenuItem = new MenuItem("Reset", 3, 1){
        public void run() {
            onReset();
        }
    };
}
```

NameRow.java

```
/*
 * NameRow.java
 *
 * © <your company here>, 2003-2005
 * Confidential and proprietary.
 */

package myapp;



/**
 * Hold a row of the Name table as an object
 */
class NameRow {

    public NameRow( String nameID, String name ) {
        _nameID = nameID;
        _name = name;
    }

    public String getNameID(){
        return _nameID;
    }

    public String getName(){
        return _name;
    }


    /**
     * Required for use by the ObjectListField in HomeScreen
     *
     * @return The Name as a string
     */
    public String toString(){
        return _name;
    }
```

```
        private String _nameID;
        private String _name;

    }
```

UltraLiteJ reference

UltraLiteJ API reference

Package

```
com.ianywhere.ultralitej12
```

ColumnSchema interface

Specifies the schema of a column.

Syntax

```
public interface ColumnSchema
```

Members

All members of ColumnSchema interface, including all inherited members.

Name	Description
"COLUMN_DEFAULT_AUTOFILENAME variable"	Indicates the autofilename column default attribute.
"COLUMN_DEFAULT_AUTOINC variable"	Indicates the autoincrement column default attribute.
"COLUMN_DEFAULT_CONSTANT variable"	Indicates a constant column default attribute.
"COLUMN_DEFAULT_CURRENT_DATE variable"	Indicates the current date (year, month, and day) column default attribute.
"COLUMN_DEFAULT_CURRENT_TIME variable"	Indicates the current time column default attribute.
"COLUMN_DEFAULT_CURRENT_TIMESTAMP variable"	Indicates the current timestamp column default attribute.
"COLUMN_DEFAULT_CURRENT_UTC_TIMESTAMP variable"	Indicates the current utc timestamp column default attribute.
"COLUMN_DEFAULT_GLOBAL_AUTOINC variable"	Indicates the global autoincrement column default attribute.
"COLUMN_DEFAULT_NONE variable"	Indicates no column default attribute.

Name	Description
"COLUMN_DEFAULT_UNIQUE_ID variable"	Indicates a new unique identifier column default attribute.

Remarks

As of version 12, this interface only contains constants for different column default values stored in *column_default* column of the system table *syscolumn*

COLUMN_DEFAULT_AUTOFILENAME variable

Indicates the autofilename column default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_AUTOFILENAME
```

Remarks

When a VARCHAR column has this default value, the column is the filename column in an external blob definition.

When a column has this type of default, the system table TableSchema.SYS_COLUMN's column_default_value column will contain the prefix and extension strings found in the external blob definition, in the form 'prefix|extension'.

The default value of existing tables can be determined by querying the system table TableSchema.SYS_COLUMNS's column_default column.

COLUMN_DEFAULT_AUTOINC variable

Indicates the autoincrement column default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_AUTOINC
```

Remarks

When using AUTOINCREMENT, the column must be one of the integer data types, or an exact numeric type. On an INSERT, if a value is not specified for the AUTOINCREMENT column, a unique value larger than any other value in the column is generated. If an INSERT specifies a value for the column that is larger than the current maximum value for the column, the value is used as a starting point for subsequent inserts.

In UltraLiteJ, the autoincremented value is not set to 0 when the table is created, and AUTOINCREMENT generates negative numbers when a signed data type is used for the column.

Therefore, declare AUTOINCREMENT columns as unsigned integers to prevent negative values from being used.

The default value of existing tables can be determined by querying the system table TableSchema.SYS_COLUMNS's column_default column.

COLUMN_DEFAULT_CONSTANT variable

Indicates a constant column default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_CONSTANT
```

Remarks

The default value of existing tables can be determined by querying the system table TableSchema.SYS_COLUMNS's column_default column.

COLUMN_DEFAULT_CURRENT_DATE variable

Indicates the current date (year, month, and day) column default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_CURRENT_DATE
```

Remarks

See "CURRENT DATE special value" under "Special values in UltraLite" in the SQL Anywhere documentation set.

The default value of existing tables can be determined by querying the system table TableSchema.SYS_COLUMNS's column_default column.

COLUMN_DEFAULT_CURRENT_TIME variable

Indicates the current time column default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_CURRENT_TIME
```

Remarks

See "CURRENT TIME special value" under "Special values in UltraLite" in the SQL Anywhere documentation set.

The default value of existing tables can be determined by querying the system table TableSchema.SYS_COLUMNS's column_default column.

COLUMN_DEFAULT_CURRENT_TIMESTAMP variable

Indicates the current timestamp column default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_CURRENT_TIMESTAMP
```

Remarks

This constant combines CURRENT DATE and CURRENT TIME to form a TIMESTAMP value, which contains the year, month, day, hour, minute, second, and fraction of a second. The precision of the fraction is set to 3 decimal places. The accuracy of this constant is limited by the accuracy of the system clock. See "CURRENT TIMESTAMP special value" under "Special values in UltraLite" in the SQL Anywhere documentation set.

The default value of existing tables can be determined by querying the system table TableSchema.SYS_COLUMNS's column_default column.

COLUMN_DEFAULT_CURRENT_UTC_TIMESTAMP variable

Indicates the current utc timestamp column default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_CURRENT_UTC_TIMESTAMP
```

Remarks

This constant combines CURRENT DATE and CURRENT TIME to form a UTC TIMESTAMP value, which contains the year, month, day, hour, minute, second, and fraction of a second as observed in GMT. The precision of the fraction is set to 3 decimal places. The accuracy of this constant is limited by the accuracy of the system clock. See "CURRENT UTC TIMESTAMP special value" under "Special values in UltraLite" in the SQL Anywhere documentation set.

The default value of existing tables can be determined by querying the system table TableSchema.SYS_COLUMNS's column_default column.

COLUMN_DEFAULT_GLOBAL_AUTOINC variable

Indicates the global autoincrement column default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_GLOBAL_AUTOINC
```

Remarks

This constant is similar to AUTOINCREMENT, but the domain is partitioned. Each partition contains the same number of values. You must assign each copy of the database a unique global database identification number. UltraLiteJ supplies default values in a database from the partition uniquely identified by that database's number.

The default value of existing tables can be determined by querying the system table TableSchema.SYS_COLUMNS's column_default column.

See also

- “setDatabaseId method” on page 130

COLUMN_DEFAULT_NONE variable

Indicates no column default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_NONE
```

Remarks

Nullable columns default to null, not nullable numeric columns default to zero, and not nullable varying length columns default to zero length values.

The default value of existing tables can be determined by querying the system table TableSchema.SYS_COLUMNS's column_default column.

COLUMN_DEFAULT_UNIQUE_ID variable

Indicates a new unique identifier column default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_UNIQUE_ID
```

Remarks

UUIDs can be used to uniquely identify rows in a table. The generated values are unique on every computer or device, meaning they can be used as keys in synchronization and replication environments.

The default value of existing tables can be determined by querying the system table TableSchema.SYS_COLUMNS's column_default column.

ConfigFile interface

Establishes the Configuration for a persistent database saved in a file.

Syntax

```
public interface ConfigFile
```

Base classes

- “ConfigPersistent interface” on page 104

Members

All members of ConfigFile interface, including all inherited members.

Name	Description
“getAutoCheckpoint method (deprecated)”	Determines if auto checkpoint is turned on.
“getCacheSize method”	Returns the cache size of the database, in bytes.
“getDatabaseName method”	Returns the database name.
“getLazyLoadIndexes method”	Determines if lazy loading indexes is turned on.
“getPageSize method”	Returns the page size of the database, in bytes.
“getRowScoreFlushSize method”	Returns the current row score flush size.
“getRowScoreMaximum method”	Returns the current row score maximum size.
“hasPersistentIndexes method”	Determines if indexes are persistent.
“hasShadowPaging method”	Determines if shadow paging is on.
“setAutocheckpoint method (deprecated)”	Sets auto checkpoint on.
“setCacheSize method”	Sets the cache size of the database, in bytes.
“setDatabaseName method”	Sets the database name.
“setEncryption method”	Sets an Encryption.
“setIndexPersistence method”	Sets persistent indexes on.
“setLazyLoadIndexes method”	Sets indexes to load as they are required, or to load all indexes at once on startup.
“setPageSize method”	Sets the page size of the database, in bytes.
“setPassword method”	Sets the database password.
“setRowScoreFlushSize method”	Enables row limiting by specifying the score used to flush out old rows.

Name	Description
"setRowScoreMaximum method"	Sets the threshold for the maximum row score to retain in memory.
"setShadowPaging method"	Sets shadow paging on.
"setWriteAtEnd method"	Sets database persistence to occur during shutdown only.
"writeAtEnd method"	Determines if the database uses write-at-end persistence.

ConfigFileME interface (BlackBerry only)

Establishes the Configuration for a persistent database saved in a file on a Java ME device file system.

Syntax

```
public interface ConfigFileME
```

Base classes

- "ConfigPersistent interface" on page 104

Members

All members of ConfigFileME interface, including all inherited members.

Name	Description
"getAutoCheckpoint method (deprecated)"	Determines if auto checkpoint is turned on.
"getCacheSize method"	Returns the cache size of the database, in bytes.
"getDatabaseName method"	Returns the database name.
"getLazyLoadIndexes method"	Determines if lazy loading indexes is turned on.
"getPageSize method"	Returns the page size of the database, in bytes.
"getRowScoreFlushSize method"	Returns the current row score flush size.
"getRowScoreMaximum method"	Returns the current row score maximum size.
"hasPersistentIndexes method"	Determines if indexes are persistent.
"hasShadowPaging method"	Determines if shadow paging is on.
"setAutocheckpoint method (deprecated)"	Sets auto checkpoint on.

Name	Description
“setCacheSize method”	Sets the cache size of the database, in bytes.
“setDatabaseName method”	Sets the database name.
“setEncryption method”	Sets an Encryption.
“setIndexPersistence method”	Sets persistent indexes on.
“setLazyLoadIndexes method”	Sets indexes to load as they are required, or to load all indexes at once on startup.
“setPageSize method”	Sets the page size of the database, in bytes.
“setPassword method”	Sets the database password.
“setRowScoreFlushSize method”	Enables row limiting by specifying the score used to flush out old rows.
“setRowScoreMaximum method”	Sets the threshold for the maximum row score to retain in memory.
“setShadowPaging method”	Sets shadow paging on.
“setWriteAtEnd method”	Sets database persistence to occur during shutdown only.
“writeAtEnd method”	Determines if the database uses write-at-end persistence.

See also

- “createConfigurationFileME method (BlackBerry only)” on page 143

ConfigNonPersistent interface

Establishes the Configuration for a non-persistent database.

Syntax

```
public interface ConfigNonPersistent
```

Base classes

- “Configuration interface” on page 115

Members

All members of ConfigNonPersistent interface, including all inherited members.

Name	Description
“getDatabaseName method”	Returns the database name.
“getPageSize method”	Returns the page size of the database, in bytes.
“setDatabaseName method”	Sets the database name.
“setPageSize method”	Sets the page size of the database, in bytes.
“setPassword method”	Sets the database password.

ConfigObjectStore interface (Java ME BlackBerry only)

Establishes the Configuration for a persistent database saved in an object store.

Syntax

```
public interface ConfigObjectStore
```

Base classes

- “ConfigPersistent interface” on page 104

Members

All members of ConfigObjectStore interface, including all inherited members.

Name	Description
“getAutoCheckpoint method (deprecated)”	Determines if auto checkpoint is turned on.
“getCacheSize method”	Returns the cache size of the database, in bytes.
“getDatabaseName method”	Returns the database name.
“getLazyLoadIndexes method”	Determines if lazy loading indexes is turned on.
“getPageSize method”	Returns the page size of the database, in bytes.
“getRowScoreFlushSize method”	Returns the current row score flush size.
“getRowScoreMaximum method”	Returns the current row score maximum size.
“hasPersistentIndexes method”	Determines if indexes are persistent.
“hasShadowPaging method”	Determines if shadow paging is on.

Name	Description
“setAutocheckpoint method (deprecated)”	Sets auto checkpoint on.
“setCacheSize method”	Sets the cache size of the database, in bytes.
“setDatabaseName method”	Sets the database name.
“setEncryption method”	Sets an Encryption.
“setIndexPersistence method”	Sets persistent indexes on.
“setLazyLoadIndexes method”	Sets indexes to load as they are required, or to load all indexes at once on startup.
“setPageSize method”	Sets the page size of the database, in bytes.
“setPassword method”	Sets the database password.
“setRowScoreFlushSize method”	Enables row limiting by specifying the score used to flush out old rows.
“setRowScoreMaximum method”	Sets the threshold for the maximum row score to retain in memory.
“setShadowPaging method”	Sets shadow paging on.
“setWriteAtEnd method”	Sets database persistence to occur during shutdown only.
“writeAtEnd method”	Determines if the database uses write-at-end persistence.

ConfigPersistent interface

Establishes the Configuration for a persistent database.

Syntax

```
public interface ConfigPersistent
```

Base classes

- “Configuration interface” on page 115

Derived classes

- “ConfigFile interface” on page 99
- “ConfigFileME interface (BlackBerry only)” on page 101
- “ConfigObjectStore interface (Java ME BlackBerry only)” on page 103
- “ConfigRecordStore interface (Java ME only)” on page 113

Members

All members of ConfigPersistent interface, including all inherited members.

Name	Description
“getAutoCheckpoint method (deprecated)”	Determines if auto checkpoint is turned on.
“getCacheSize method”	Returns the cache size of the database, in bytes.
“getDatabaseName method”	Returns the database name.
“getLazyLoadIndexes method”	Determines if lazy loading indexes is turned on.
“getPageSize method”	Returns the page size of the database, in bytes.
“getRowScoreFlushSize method”	Returns the current row score flush size.
“getRowScoreMaximum method”	Returns the current row score maximum size.
“hasPersistentIndexes method”	Determines if indexes are persistent.
“hasShadowPaging method”	Determines if shadow paging is on.
“setAutocheckpoint method (deprecated)”	Sets auto checkpoint on.
“setCacheSize method”	Sets the cache size of the database, in bytes.
“setDatabaseName method”	Sets the database name.
“setEncryption method”	Sets an Encryption.
“setIndexPersistence method”	Sets persistent indexes on.
“setLazyLoadIndexes method”	Sets indexes to load as they are required, or to load all indexes at once on startup.
“setPageSize method”	Sets the page size of the database, in bytes.
“setPassword method”	Sets the database password.
“setRowScoreFlushSize method”	Enables row limiting by specifying the score used to flush out old rows.
“setRowScoreMaximum method”	Sets the threshold for the maximum row score to retain in memory.
“setShadowPaging method”	Sets shadow paging on.
“setWriteAtEnd method”	Sets database persistence to occur during shutdown only.

Name	Description
"writeAtEnd method"	Determines if the database uses write-at-end persistence.

Remarks

The database type is determined as follows: The database is a shadow paging persistent database if shadow paging is on; otherwise, the database is an in-memory database with persistence during shutdown only (and when re-opened) if write-at-end is on. The database is a non-shadow paging persistent database if both shadow paging and write-at-end is off.

Options such as lazy loading, row score flush size and row score maximum only apply to shadow and non-shadow persistent databases.

getAutoCheckpoint method (deprecated)

Determines if auto checkpoint is turned on.

Syntax

```
boolean ConfigPersistent.getAutoCheckpoint()
```

Returns

true, if the database has auto checkpoint turned on; otherwise, false.

getCacheSize method

Returns the cache size of the database, in bytes.

Syntax

```
int ConfigPersistent.getCacheSize()
```

Returns

The cache size.

See also

- "setCacheSize method" on page 108

getLazyLoadIndexes method

Determines if lazy loading indexes is turned on.

Syntax

`boolean **ConfigPersistent.getLazyLoadIndexes**()`

Returns

true, if lazy loading is on; otherwise, false.

See also

- "setLazyLoadIndexes method" on page 110

getRowScoreFlushSize method

Returns the current row score flush size.

Syntax

```
int ConfigPersistent.getRowScoreFlushSize()
```

Returns

the current row score flush size.

See also

- "setRowScoreFlushSize method" on page 110

getRowScoreMaximum method

Returns the current row score maximum size.

Syntax

```
int ConfigPersistent.getRowScoreMaximum()
```

Returns

the current row score maximum size.

See also

- "setRowScoreMaximum method" on page 111

hasPersistentIndexes method

Determines if indexes are persistent.

Syntax

```
boolean ConfigPersistent.hasPersistentIndexes()
```

Returns

true, if indexes are persistent; otherwise, false.

See also

- "setIndexPersistence method" on page 109

hasShadowPaging method

Determines if shadow paging is on.

Syntax

```
boolean ConfigPersistent.hasShadowPaging( )
```

Returns

true, if shadow paging is on; otherwise, false.

See also

- "setShadowPaging method" on page 112

setAutocheckpoint method (deprecated)

Sets auto checkpoint on.

Syntax

```
ConfigPersistent ConfigPersistent.setAutocheckpoint(
    boolean auto_checkpoint
) throws ULjException
```

Parameters

- **auto_checkpoint** true, to set autocheckpoint on.

Returns

This ConfigPersistent, with the auto checkpoint specified,

setCacheSize method

Sets the cache size of the database, in bytes.

Syntax

```
ConfigPersistent ConfigPersistent.setCacheSize(
    int cache_size
) throws ULjException
```

Parameters

- **cache_size** The cache size. The default cache size is 20480 (20KB) on all platforms.

Returns

This ConfigPersistent, with the cache size specified.

Remarks

The cache size determines the number of database pages resident in the page cache. Increasing the size means less reading and writing of database pages, at the expense of increased time to locate pages in the cache.

See also

- "getCacheSize method" on page 106

setEncryption method

Sets an Encryption.

Syntax

```
ConfigPersistent ConfigPersistent.setEncryption(
    EncryptionControl control
)
```

Parameters

- **control** An EncryptionControl object used to encrypt the database.

Returns

This ConfigPersistent, with the encryption specified.

setIndexPersistence method

Sets persistent indexes on.

Syntax

```
ConfigPersistent ConfigPersistent.setIndexPersistence(
    boolean store
) throws ULjException
```

Parameters

- **store** Set true to store indexes; otherwise, set false so that indexes are built prior to the first time they are used.

Returns

This ConfigPersistent, with the index persistence set.

Remarks

This setting is used only when the database is created. It determines which strategy of index persistence is to be used for the database.

When an existing database is opened, the creation-time setting is used and the configuration is updated to reflect that value.

setLazyLoadIndexes method

Sets indexes to load as they are required, or to load all indexes at once on startup.

Syntax

```
ConfigPersistent ConfigPersistent.setLazyLoadIndexes(
    boolean lazy_load
) throws ULjException
```

Parameters

- **lazy_load** Set true so that indexes load as required; otherwise, set false to load all indexes at once on startup.

Returns

This ConfigPersistent, with the index loading schema specified.

Remarks

Enabling this option reduces the startup time of the database but future operations may perform slower.

See also

- "getLazyLoadIndexes method" on page 106
- "setRowScoreFlushSize method" on page 110

setRowScoreFlushSize method

Enables row limiting by specifying the score used to flush out old rows.

Syntax

```
ConfigPersistent ConfigPersistent.setRowScoreFlushSize(
    int flushSize
) throws ULjException
```

Parameters

- **flushSize** The row score value used to determine how many rows to flush. The default is 0 (indicating no row limiting).

Returns

This ConfigPersistent, with the flush size value specified.

Remarks

Row score is a measure of the references used to maintain recently used rows in memory. Each row in memory is assigned a score based on the number and types of columns they have, which approximates the maximum number of references they could use. Most columns score as 1; varchar binary, long binary and UUID score as 2; long varchar score as 4.

When the maximum score threshold is reached, the flush size is used to determine how many old rows to remove.

It is recommended that the flush size (measured as a row score) be kept reasonable (less than 1000) to prevent large interuptions.

Row limiting is only enabled when index lazy loading and persistent indexes is turned on.

See also

- "setRowScoreMaximum method" on page 111
- "setLazyLoadIndexes method" on page 110

setRowScoreMaximum method

Sets the threshold for the maximum row score to retain in memory.

Syntax

```
ConfigPersistent ConfigPersistent.setRowScoreMaximum(
    int threshold
) throws ULjException
```

Parameters

- **threshold** The maximum threshold value. The maximum value is 200,000. The default value is 50,000.

Returns

This ConfigPersistent, with the maximum threshold value specified.

Remarks

See setRowScoreFlushSize(int) for more details.

See also

- “setRowScoreFlushSize method” on page 110
- “setLazyLoadIndexes method” on page 110

setShadowPaging method

Sets shadow paging on.

Syntax

```
ConfigPersistent ConfigPersistent.setShadowPaging(
    boolean shadow
) throws ULjException
```

Parameters

- **shadow** Set true to set shadow paging on; otherwise, false.

Returns

This ConfigPersistent with ShadowPaging specified.

Remarks

Shadow paging means that all writing to the persistent store occurs to unused database pages, which do not become permanently stored until a commit operation completes. All committed changes are guaranteed to be permanently saved, even if the application terminates abnormally.

If shadow paging is set to false, the database may be corrupt when change operations have occurred but are not yet committed.

Persisting without shadow paging means that database operations can proceed more quickly and return smaller database results.

A database should only be processed without shadow paging if the data is non-critical or can be recovered by synchronization.

See also

- “hasShadowPaging method” on page 108

setWriteAtEnd method

Sets database persistence to occur during shutdown only.

Syntax

```
ConfigPersistent ConfigPersistent.setWriteAtEnd(
    boolean write_at_end
) throws ULjException
```

Parameters

- **write_at_end** Set true to retain the database in memory until it is shutdown.

Returns

This ConfigPersistent with write-at-end persistence set as specified.

Remarks

Enabling this option speeds up database operations but all changes to the database are lost if the application terminates abnormally. The database must be sufficiently small to fit in memory because the database is essentially an in-memory database with one large write during shutdown and one large read on subsequent database opens.

A database should only be processed with write-at-end persistence if the data is non-critical or can be recovered by synchronization.

This option is ignored if shadow paging is turned on.

See also

- "writeAtEnd method" on page 113
- "setShadowPaging method" on page 112

writeAtEnd method

Determines if the database uses write-at-end persistence.

Syntax

```
boolean ConfigPersistent.writeAtEnd()
```

Returns

true, if the database is retained in memory until shutdown; otherwise, false.

See also

- "setWriteAtEnd method" on page 112

ConfigRecordStore interface (Java ME only)

Establishes the Configuration for a persistent database saved in a Java ME record store.

Syntax

```
public interface ConfigRecordStore
```

Base classes

- “ConfigPersistent interface” on page 104

Members

All members of ConfigRecordStore interface, including all inherited members.

Name	Description
“getAutoCheckpoint method (deprecated)”	Determines if auto checkpoint is turned on.
“getCacheSize method”	Returns the cache size of the database, in bytes.
“getDatabaseName method”	Returns the database name.
“getLazyLoadIndexes method”	Determines if lazy loading indexes is turned on.
“getPageSize method”	Returns the page size of the database, in bytes.
“getRowScoreFlushSize method”	Returns the current row score flush size.
“getRowScoreMaximum method”	Returns the current row score maximum size.
“hasPersistentIndexes method”	Determines if indexes are persistent.
“hasShadowPaging method”	Determines if shadow paging is on.
“setAutocheckpoint method (deprecated)”	Sets auto checkpoint on.
“setCacheSize method”	Sets the cache size of the database, in bytes.
“setDatabaseName method”	Sets the database name.
“setEncryption method”	Sets an Encryption.
“setIndexPersistence method”	Sets persistent indexes on.
“setLazyLoadIndexes method”	Sets indexes to load as they are required, or to load all indexes at once on startup.
“setPageSize method”	Sets the page size of the database, in bytes.
“setPassword method”	Sets the database password.
“setRowScoreFlushSize method”	Enables row limiting by specifying the score used to flush out old rows.
“setRowScoreMaximum method”	Sets the threshold for the maximum row score to retain in memory.

Name	Description
"setShadowPaging method"	Sets shadow paging on.
"setWriteAtEnd method"	Sets database persistence to occur during shutdown only.
"writeAtEnd method"	Determines if the database uses write-at-end persistence.

Configuration interface

Establishes the Configuration for a database.

Syntax

```
public interface **Configuration**
```

Derived classes

- "ConfigNonPersistent interface" on page 102
- "ConfigPersistent interface" on page 104

Members

All members of Configuration interface, including all inherited members.

Name	Description
"getDatabaseName method"	Returns the database name.
"getPageSize method"	Returns the page size of the database, in bytes.
"setDatabaseName method"	Sets the database name.
"setPageSize method"	Sets the page size of the database, in bytes.
"setPassword method"	Sets the database password.

Remarks

Some attributes are used only during database creation while others apply to the initial connection to a database. Attributes are ignored if they are set after creating a database, or connecting to a database.

getDatabaseName method

Returns the database name.

Syntax

```
String Configuration.getDatabaseName()
```

Returns

The name of the database.

getPageSize method

Returns the page size of the database, in bytes.

Syntax

```
int Configuration.getPageSize()
```

Returns

The page size.

setDatabaseName method

Sets the database name.

Syntax

```
Configuration Configuration.setDatabaseName(
    String db_name
) throws ULjException
```

Parameters

- **db_name**   The name of database.

Returns

This Configuration, with the database name specified.

setPageSize method

Sets the page size of the database, in bytes.

Syntax

```
Configuration Configuration.setPageSize(
    int page_size
) throws ULjException
```

Parameters

- **page_size**   The page size.

Returns

This Configuration, with the page size specified.

Remarks

The page size setting is used to determine the maximum size of a row stored in a persistent database. It establishes the size of an index page, and determines the number of children that each page can have.

When using an existing database, the size is already set to the page size of the database when it was created. You can not reset the page size of an existing database using this method.

The page size can range from 256 to 32736 bytes. The default is 1024 bytes. Page size is always adjusted to be a multiple of 32.

setPassword method

Sets the database password.

Syntax

```
Configuration Configuration.setPassword(
    String password
) throws ULjException
```

Parameters

- **password**   A password for a new database, or the password to gain access to an existing database.

Returns

This Configuration, with the database password set.

Remarks

The password is used to gain access to the database, and must match the password specified when the database was created. The default is "dba".

Connection interface

Describes a database connection, which is required to initiate database operations.

Syntax

```
public interface Connection
```

Members

All members of Connection interface, including all inherited members.

Name	Description
“checkpoint method”	Checkpoints the database changes.
“commit method”	Commits the database changes.
“createDecimalNumber method”	Creates a new DecimalNumber object.
“createSyncParms method”	Creates a new synchronization parameter set.
“createUUIDValue method”	Creates a UUID value.
“dropDatabase method”	Drops a database.
“emergencyShutdown method”	Performs an emergency shut down of a connected database.
“getDatabaseId method”	Returns the value of database ID.
“getDatabaseInfo method”	Returns a DataInfo object containing information on database properties.
“getDatabaseProperty method”	Returns a property of the database.
“getLastDownloadTime method”	Returns the time of the most recent download of the specified publication.
“getLastIdentity method”	Retrieves the most recent value inserted into an DEFAULT AUTOINCREMENT or DEFAULT GLOBAL AUTOINCREMENT column, or zero if the most recent insert was into a table that had no such column.
“getOption method”	Returns a database option.
“getState method”	Returns the state of the connection.

Name	Description
“getSyncObserver method”	Returns the currently registered SyncObserver object for this Connection.
“getSyncResult method”	Returns the result of the last SYNCHRONIZE SQL statement, but not the last Connection.synchronize API call.
“isSynchronizationDeleteDisabled method”	Determine if synchronization of deletes is disabled.
“prepareStatement method”	Prepares a statement for execution.
“release method”	Releases this connection.
“resetLastDownloadTime method”	Resets the time of the download for the specified publications.
“rollback method”	Commits a rollback to undo changes to the database.
“setDatabaseId method”	Sets the database id for global autoincrement.
“setOption method”	Sets the database option.
“setSyncObserver method”	Sets a SyncObserver object to monitor the progress of synchronizations on this connection.
“synchronize method”	Synchronizes the database with a MobiLink server.
“CONNECTED variable”	A connected state.
“NOT_CONNECTED variable”	A not connected state.
“OPTION_BLOB_FILE_BASE_DIR variable”	Database option: blob file base dir.
“OPTION_DATABASE_ID variable”	Database option: database id.
“OPTION_DATE_FORMAT variable”	Database option: date format.

Name	Description
"OPTION_DATE_ORDER variable"	Database option: date order.
"OPTION_ML_REMOTE_ID variable"	Database option: ML remote ID.
"OPTION_NEAREST_CENTURY variable"	Database option: nearest century.
"OPTION_PRECISION variable"	Database option: precision.
"OPTION_SCALE variable"	Database option: scale.
"OPTION_TIME_FORMAT variable"	Database option: time format.
"OPTION_TIMESTAMP_FORMAT variable"	Database option: timestamp format.
"OPTION_TIMESTAMP_INCREMENT variable"	Database option: timestamp increment.
"OPTION_TIMESTAMP_WITH_TIME_ZONE_FORMAT variable"	Database option: timestamp with time zone format.
"PROPERTY_DATABASE_NAME variable"	Database Property: database name.
"PROPERTY_PAGE_SIZE variable"	Database Property: page size.
"SYNC_ALL variable"	The publication list used to request synchronization of all tables in the database, including tables not in any publication.
"SYNC_ALL_DB_PUB_NAME variable"	The reserved name of the SYNC_ALL_DB publication.
"SYNC_ALL_PUBS variable"	The publication list used to request synchronization of all publications in the database.

Remarks

A connection is obtained using the connect or createDatabase methods of the DatabaseManager class. Use the release method when the connection is no longer needed. When all connections for a database are released, the database is closed.

A connection provides the following capabilities:

- create new schema (tables, indexes and publications)
- create new value and domain objects
- permanently commit changes to the database
- prepare SQL statements for execution
- roll back uncommited changes to the database
- checkpoint (update the underlying persistent store with committed changes, rather than just storing the change transactions) the database.

The following example demonstrates how to create a schema for a simple database with a connection, conn, created for it. The database contains a table named T1, which has a single integer primary key column named num, and a table named T2, which has an integer primary key column named num and an integer column named quantity. T2 has an addition index on quantity. A publication named PubA contains T1.

```
// Assumes a valid connection object, conn, for the current database.

PreparedStatement ps;

ps = conn.prepareStatement( "CREATE TABLE T1 ( num INT NOT NULL PRIMARY
KEY )" );
ps.execute();
ps.close();

ps = conn.prepareStatement( "CREATE TABLE T2 ( num INT NOT NULL PRIMARY KEY,
quantity INT)" );
ps.execute();
ps.close();

ps = conn.prepareStatement( "CREATE INDEX index1 ON T2( quantity )" );
ps.execute();
ps.close();

ps = conn.prepareStatement( "CREATE Publication PubA ( Table T1 )" );
ps.execute();
ps.close();
```

See also

- “DatabaseManager class” on page 141
- “createDatabase method” on page 145
- “connect method” on page 143
- “release method” on page 129

checkpoint method

Checkpoints the database changes.

Syntax

```
void Connection.checkpoint() throws ULjException
```

Remarks

Invoking this applies all committed transactions to the persistent version of the database.

commit method

Commits the database changes.

Syntax

```
void Connection.commit() throws ULjException
```

Remarks

Invoking this method causes all table data changes since the last commit or rollback to become permanent.

createDecimalNumber method

Creates a new DecimalNumber object.

Overload list

Name	Description
"createDecimalNumber(int, int) method"	Creates a DecimalNumber.
"createDecimalNumber(int, int, String) method"	Creates a DecimalNumber.

createDecimalNumber(int, int) method

Creates a DecimalNumber.

Syntax

```
DecimalNumber Connection.createDecimalNumber(
    int precision,
    int scale
) throws ULjException
```

Parameters

- **precision** The number of digits in the number.
- **scale** The number of decimal places in the number.

Returns

The DecimalNumber with the specified type.

createDecimalNumber(int, int, String) method

Creates a DecimalNumber.

Syntax

```
DecimalNumber Connection.createDecimalNumber(
    int precision,
    int scale,
    String value
) throws ULjException
```

Parameters

- **precision** The number of digits in the number.
- **scale** The number of decimal places in the number.
- **value** The value to be set.

Returns

The DecimalNumber with the specified type.

createSyncParms method

Creates a new synchronization parameter set.

Overload list

Name	Description
"createSyncParms(int, String, String) method"	Creates a synchronization parameters set.
"createSyncParms(String, String) method"	Creates a synchronization parameters set for HTTP synchronization.

createSyncParms(int, String, String) method

Creates a synchronization parameters set.

Syntax

```
SyncParms Connection.createSyncParms(
    int streamType,
    String userName,
    String version
) throws ULjException
```

Parameters

- **streamType** One of the constants defined in the SyncParms class used to identify the type of synchronization stream.
- **userName** The MobiLink user name.
- **version** The MobiLink script version.

Returns

SyncParms object

See also

- "createSyncParms method" on page 123
- "HTTP_STREAM variable" on page 250
- "HTTPS_STREAM variable" on page 250

createSyncParms(String, String) method

Creates a synchronization parameters set for HTTP synchronization.

Syntax

```
SyncParms Connection.createSyncParms(
    String userName,
    String version
) throws ULjException
```

Parameters

- **userName** The unique MobiLink user name for this client database. For more information, see the SyncParms#setUserName method.
- **version** The MobiLink script version.

Returns

The SyncParms object.

See also

- "createSyncParms method" on page 123
- "setUserName method" on page 249

createUUIDValue method

Creates a UUID value.

Syntax

```
UUIDValue Connection.createUUIDValue() throws ULjException
```

Returns

The Value for the domain.

dropDatabase method

Drops a database.

Syntax

```
void Connection.dropDatabase() throws ULjException
```

Remarks

The database referenced by the connection is erased and the connection is released. Only this connection can be active for the database being dropped.

emergencyShutdown method

Performs an emergency shut down of a connected database.

Syntax

```
void Connection.emergencyShutdown() throws ULjException
```

Remarks

This method should only be invoked in severe error situations. It should only be used if physical hardware or data is destroyed.

The method closes all open connections and shuts down the connected database.

getDatabaseId method

Returns the value of database ID.

Syntax

```
int Connection.getDatabaseId() throws ULjException
```

Returns

The database ID.

Exceptions

- **"ULjException class"** If the database id has not been set.

See also

- "getLastIdentity method" on page 127
- "setDatabaseId method" on page 130

getDatabaseInfo method

Returns a DataInfo object containing information on database properties.

Syntax

```
DatabaseInfo Connection.getDatabaseInfo() throws ULjException
```

Returns

The DatabaseInfo object.

getDatabaseProperty method

Returns a property of the database.

Syntax

```
String Connection.getDatabaseProperty(String name) throws ULjException
```

Parameters

- **name** The name of the database property.

Returns

The value of the property that corresponds to the given name.

getLastDownloadTime method

Returns the time of the most recent download of the specified publication.

Syntax

```
Date Connection.getLastDownloadTime(String pub_name) throws ULjException
```

Parameters

- **pub_name** The name of the publication to check.

Returns

The timestamp of the last download.

Remarks

The parameter pub_name must reference a single publication or be the special publication Connection.SYNC_ALL_DB_PUB_NAME for the time of the last download of the full database.

See also

- "SYNC_ALL_DB_PUB_NAME variable" on page 137
- "resetLastDownloadTime method" on page 130

getLastIdentity method

Retrieves the most recent value inserted into an DEFAULT AUTOINCREMENT or DEFAULT GLOBAL AUTOINCREMENT column, or zero if the most recent insert was into a table that had no such column.

Syntax

```
long Connection.getLastIdentity()
```

Returns

The most recent identity value.

getOption method

Returns a database option.

Syntax

```
String Connection.getOption(String option_name) throws ULjException
```

Parameters

- **option_name** The name of option to get.

Returns

The value of the database option.

Remarks

Database options are stored within the database (and may be obtained when a database is connected at some later time after the option has been set).

A set of required options are created when a database is created.

See also

- "setOption method" on page 131

getState method

Returns the state of the connection.

Syntax

```
byte Connection.getState() throws ULjException
```

Remarks

The following example demonstrates how to check the connection state and release the connection.

```
if( _conn.getState() == Connection.CONNECTED ){
    _conn.release();
}
```

getSyncObserver method

Returns the currently registered SyncObserver object for this Connection.

Syntax

```
SyncObserver Connection.getSyncObserver()
```

Returns

The SyncObserver, or null if no observer exists.

See also

- "setSyncObserver method" on page 131

getSyncResult method

Returns the result of the last SYNCHRONIZE SQL statement, but not the last Connection.synchronize API call.

Syntax

```
SyncResult Connection.getSyncResult()
```

Returns

The SyncResult object.

Remarks

To obtain the SyncResult for synchronize(SyncParms), call getSyncResult() on the SyncParms object passed in.

See also

- “SyncResult class” on page 250

isSynchronizationDeleteDisabled method

Determine if synchronization of deletes is disabled.

Syntax

```
boolean Connection.isSynchronizationDeleteDisabled()
```

Returns

true, iff synchronization of deletes is disabled.

prepareStatement method

Prepares a statement for execution.

Syntax

```
PreparedStatement Connection.prepareStatement(
    String sql
) throws ULjException
```

Parameters

- **sql** A SQL statement to prepare.

Returns

A PreparedStatement object.

See also

- “PreparedStatement interface” on page 182

release method

Releases this connection.

Syntax

```
void Connection.release() throws ULjException
```

Remarks

Once a connection has been released, it can no longer be used to access the database.

It is an error to attempt to release a connection for which there exist uncommitted transactions.

resetLastDownloadTime method

Resets the time of the download for the specified publications.

Syntax

```
void Connection.resetLastDownloadTime(
    String pub_name
) throws ULjException
```

Parameters

- **pub_name** The name of the publication to check.

Remarks

To reset the download time for when the entire database is synchronized, use the special Connection.SYNC_ALL_DB_PUB_NAME publication.

This method requires that there are no uncommitted transactions on the current connection.

rollback method

Commits a rollback to undo changes to the database.

Syntax

```
void Connection.rollback() throws ULjException
```

Remarks

Invoking this method undoes all table data changes on this connection since the commit or rollback.

setDatabaseId method

Sets the database id for global autoincrement.

Syntax

```
void Connection.setDatabaseId(int id) throws ULjException
```

Parameters

- **id** The database id.

Remarks

The database id does not have a default value. During an insert, GLOBAL AUTOINCREMENT columns have NULL values inserted unless a database id has explicitly been set.

See also

- "The GLOBAL AUTOINCREMENT default" [*SQL Anywhere Server - SQL Usage*]
- "getDatabaseId method" on page 125

setOption method

Sets the database option.

Syntax

```
void Connection.setOption(
    String option_name,
    String option_value
) throws ULjException
```

Parameters

- **option_name** The name of the option to set.
- **option_value** The new value of the option.

Remarks

If the option is not currently stored on the database, it is created.

There cannot be any uncommitted transactions for this connection when the method is invoked.

See also

- "getOption method" on page 127

setSyncObserver method

Sets a SyncObserver object to monitor the progress of synchronizations on this connection.

Syntax

```
void Connection.setSyncObserver(SyncObserver so)
```

Parameters

- **so** An object implementing the SyncObserser interface or null to remove the current SyncObserver.

Remarks

This SyncObserver is used by subsequent SYNCHRONIZE SQL statements.

The default is null, suggesting no observer.

See also

- “SyncObserver interface” on page 230

synchronize method

Synchronizes the database with a MobiLink server.

Syntax

```
void Connection.synchronize(SyncParms config) throws ULjException
```

Parameters

- **config** The parameters used for synchronization

Remarks

The database is checkpointed when the download is applied to the database.

See also

- “checkpoint method” on page 121

CONNECTED variable

A connected state.

Syntax

```
final byte Connection.CONNECTED
```

NOT_CONNECTED variable

A not connected state.

Syntax

```
final byte Connection.NOT_CONNECTED
```

OPTION_BLOB_FILE_BASE_DIR variable

Database option: blob file base dir.

Syntax

```
final String Connection.OPTION_BLOB_FILE_BASE_DIR
```

Remarks

For BlackBerry devices, the default value is "file:///SDCard/"; otherwise, it is "".

See also

- "setOption method" on page 131

OPTION_DATABASE_ID variable

Database option: database id.

Syntax

```
final String Connection.OPTION_DATABASE_ID
```

Remarks

A default value is not specified. It must be explicitly assigned.

See also

- "setOption method" on page 131

OPTION_DATE_FORMAT variable

Database option: date format.

Syntax

```
final String Connection.OPTION_DATE_FORMAT
```

Remarks

The default value is "YYY-MM-DD".

See also

- "setOption method" on page 131

OPTION_DATE_ORDER variable

Database option: date order.

Syntax

```
final String Connection.OPTION_DATE_ORDER
```

Remarks

The default value is "YMD".

See also

- “setOption method” on page 131

OPTION_ML_REMOTE_ID variable

Database option: ML remote ID.

Syntax

```
final String Connection.OPTION_ML_REMOTE_ID
```

Remarks

A default value is not specified. A value is set after the first MobiLink synchronization.

See also

- “setOption method” on page 131

OPTION_NEAREST_CENTURY variable

Database option: nearest century.

Syntax

```
final String Connection.OPTION_NEAREST_CENTURY
```

Remarks

The default value is "50".

See also

- “setOption method” on page 131

OPTION_PRECISION variable

Database option: precision.

Syntax

```
final String Connection.OPTION_PRECISION
```

Remarks

The default value is "30".

See also

- “setOption method” on page 131

OPTION_SCALE variable

Database option: scale.

Syntax

final String **Connection.OPTION_SCALE**

Remarks

The default value is "6".

See also

- “setOption method” on page 131

OPTION_TIME_FORMAT variable

Database option: time format.

Syntax

final String **Connection.OPTION_TIME_FORMAT**

Remarks

The default value is "HH:NN:SS.SSS".

See also

- “setOption method” on page 131

OPTION_TIMESTAMP_FORMAT variable

Database option: timestamp format.

Syntax

final String **Connection.OPTION_TIMESTAMP_FORMAT**

Remarks

The default value is "YYYY-MM-DD HH:NN:SS.SSS".

See also

- “setOption method” on page 131

OPTION_TIMESTAMP_INCREMENT variable

Database option: timestamp increment.

Syntax

```
final String Connection.OPTION_TIMESTAMP_INCREMENT
```

Remarks

The default value is "1".

See also

- “setOption method” on page 131

OPTION_TIMESTAMP_WITH_TIME_ZONE_FORMAT variable

Database option: timestamp with time zone format.

Syntax

```
final String Connection.OPTION_TIMESTAMP_WITH_TIME_ZONE_FORMAT
```

Remarks

The default value is "YYYY-MM-DD HH:NN:SS.SSS+HH:NN".

See also

- “setOption method” on page 131

PROPERTY_DATABASE_NAME variable

Database Property: database name.

Syntax

```
final String Connection.PROPERTY_DATABASE_NAME
```

PROPERTY_PAGE_SIZE variable

Database Property: page size.

Syntax

```
final String Connection.PROPERTY_PAGE_SIZE
```

SYNC_ALL variable

The publication list used to request synchronization of all tables in the database, including tables not in any publication.

Syntax

```
final String Connection.SYNC_ALL
```

Remarks

Tables marked as NoSync are never synchronized.

This constant is equivalent to the null reference or an empty string.

SYNC_ALL_DB_PUB_NAME variable

The reserved name of the SYNC_ALL_DB publication.

Syntax

```
final String Connection.SYNC_ALL_DB_PUB_NAME
```

See also

- "getLastDownloadTime method" on page 126
- "resetLastDownloadTime method" on page 130

SYNC_ALL_PUBS variable

The publication list used to request synchronization of all publications in the database.

Syntax

```
final String Connection.SYNC_ALL_PUBS
```

Remarks

Tables that are marked as NoSync are never synchronized.

DatabaseInfo interface

Associated with a Connection object and provides methods to reveal database information.

Syntax

```
public interface DatabaseInfo
```

Members

All members of DatabaseInfo interface, including all inherited members.

Name	Description
"getCommitCount method"	Returns the total number of commit operations performed on the database.
"getDbFormat method"	Returns the database version number.
"getDbSize method"	Returns the database size.
"getLogSize method"	Returns the overall size of the transaction log, in bytes.
"getNumberRowsToUpload method"	Returns the number of rows awaiting upload.
"getPageReads method"	Returns the number of page reads.
"getPageSize method"	Returns the page size of the database, in bytes.
"getPageWrites method"	Returns the number of page writes.
"getRelease method"	Returns the software release number.

Remarks

This interface is invoked with the getDatabaseInfo method of a Connection object.

See also

- "Connection interface" on page 117
- "getDatabaseInfo method" on page 126

getCommitCount method

Returns the total number of commit operations performed on the database.

Syntax

```
int DatabaseInfo.getCommitCount()
```

Returns

The total number of commit operations.

getDbFormat method

Returns the database version number.

Syntax

```
int DatabaseInfo.getDbFormat()
```

Returns

The version number.

getDbSize method

Returns the database size.

Syntax

```
int DatabaseInfo.getDbSize()
```

Returns

(-1) when the database is not persistent and when write-at-end is configured; otherwise, the current size of the persistent store is returned.

getLogSize method

Returns the overall size of the transaction log, in bytes.

Syntax

```
int DatabaseInfo.getLogSize()
```

Returns

The size of transaction log.

getNumberRowsToUpload method

Returns the number of rows awaiting upload.

Syntax

```
int DatabaseInfo.getNumberRowsToUpload()
```

Returns

The number of rows.

getPageReads method

Returns the number of page reads.

Syntax

```
int DatabaseInfo.getPageReads( )
```

Returns

The number of page reads.

getPageSize method

Returns the page size of the database, in bytes.

Syntax

```
int DatabaseInfo.getPageSize( )
```

Returns

The page size.

getPageWrites method

Returns the number of page writes.

Syntax

```
int DatabaseInfo.getPageWrites( )
```

Returns

The number of page writes.

getRelease method

Returns the software release number.

Syntax

```
String DatabaseInfo.getRelease( )
```

Returns

The release number.

Remarks

For example, a software release value of "12.0.0.1234" represents the 12.0.0 release and the 1234 build number.

DatabaseManager class

Provides static methods to obtain basic configurations, create a new database, and connect to an existing database.

Syntax

```
public class DatabaseManager
```

Members

All members of DatabaseManager class, including all inherited members.

Name	Description
“connect method”	Connects to an existing database based on a Configuration and returns the Connection.
“createConfigurationFileME method (BlackBerry only)”	Creates a Configuration with a file on a Java ME device file system as the physical store and returns a ConfigFileME.
“createConfigurationNonPersistent method”	Creates a Configuration without a persistent store and returns a ConfigNonPersist object.
“createConfigurationObjectStore method (Java ME BlackBerry only)”	Creates a Configuration with a RIM object store as the physical store and returns a ConfigObjectStore.
“createDatabase method”	Creates a new database based on a Configuration and returns the Connection.
“createFileTransfer method”	Creates a FileTransfer object for transfering files to or from MobiLink.

Name	Description
"createObjectStoreTransfer method (Java ME BlackBerry only)"	Creates a FileTransfer object for transfering UltraLiteJ database files to or from MobiLink, and storing them into the RIM object store.
"createSISHTTPListener method (Java ME BlackBerry only)"	Creates an HTTP SISListener for server-initiated synchronizations.
"release method"	Closes the DatabaseManager to release all connections and to shutdown all databases.
"setErrorLanguage method"	Sets the language to use for error messages.

Remarks

The following example demonstrates how to open an existing database, and create a new one if it does not exist.

```
// Java SE Sample
Connection conn;
ConfigFile config = DatabaseManager.createConfigurationFile(
        "test.ulj"
    );
try {
    conn = DatabaseManager.connect(config);
} catch(ULjException ex) {
    conn = DatabaseManager.createDatabase(config);
    // Create the schema here.
}


// BlackBerry Sample
Connection conn;
ConfigObjectStore config = DatabaseManager.createConfigurationObjectStore(
        "test.ulj"
    );
try {
    conn = DatabaseManager.connect(config);
} catch(ULjException ex) {
    conn = DatabaseManager.createDatabase(config);
    // Create the schema here.
}


// BlackBerry media card Sample
Connection conn;
ConfigFileME config = DatabaseManager.createConfigurationFileME(
        "file:///SDCard/ulj/test.ulj"
    );
```

```
try {
    conn = DatabaseManager.connect(config);
} catch(ULjException ex) {
    conn = DatabaseManager.createDatabase(config);
    // Create the schema here.
}


// Java ME Sample
Connection conn;
ConfigRecordStore config = DatabaseManager.createConfigurationRecordStore(
        "test.ulj"
    );
try {
    conn = DatabaseManager.connect(config);
} catch(ULjException ex) {
    conn = DatabaseManager.createDatabase(config);
    // Create the schema here.
}
```

See also

- "Connection interface" on page 117

connect method

Connects to an existing database based on a Configuration and returns the Connection.

Syntax

```
Connection DatabaseManager.connect(
    Configuration config
) throws ULjException
```

Parameters

- **config** The Configuration for the existing database.

Returns

A Connection to the existing database.

See also

- "Configuration interface" on page 115

createConfigurationFileME method (BlackBerry only)

Creates a Configuration with a file on a Java ME device file system as the physical store and returns a ConfigFileME.

Syntax

```
ConfigFileME DatabaseManager.createConfigurationFileME(
    String db_name
) throws ULjException
```

Parameters

- **db_name** The full URI to the database file on the device. For example, *file:///SDCard/ulj/uljtest.ulj*

Returns

The ConfigFileME

Remarks

For the BlackBerry device, this usually means the device's media card, but it can be other file systems.

For BlackBerry devices, the internal flash is accessed using URIs starting with file:///store/ (case sensitive) while the SD media card is accessed using URIs starting with file:///SDCard/ (case sensitive). File paths are case insensitive **except for the host portion which is case sensitive**. Percent (%) characters have special meaning and relative paths (directories "." and "..") are not allowed. For more information on file name restrictions, please consult the BlackBerry JDE API Reference for the *javax.microedition.io.file* optional package.

See also

- "ConfigFileME interface (BlackBerry only)" on page 101

createConfigurationNonPersistent method

Creates a Configuration without a persistent store and returns a ConfigNonPersist object.

Syntax

```
ConfigNonPersistent DatabaseManager.createConfigurationNonPersistent(
    String db_name
) throws ULjException
```

Parameters

- **db_name** The name of the database.

Returns

The ConfigNonPersistent used to configure a database.

See also

- "ConfigNonPersistent interface" on page 102

createConfigurationObjectStore method (Java ME BlackBerry only)

Creates a Configuration with a RIM object store as the physical store and returns a ConfigObjectStore.

Syntax

```
ConfigObjectStore DatabaseManager.createConfigurationObjectStore(
    String db_name
) throws ULjException
```

Parameters

- **db_name** The name of the database.

Returns

The ConfigObjectStore used to configure a database.

See also

- "ConfigObjectStore interface (Java ME BlackBerry only)" on page 103

createDatabase method

Creates a new database based on a Configuration and returns the Connection.

Syntax

```
Connection DatabaseManager.createDatabase(
    Configuration config
) throws ULjException
```

Parameters

- **config** The Configuration for the new database.

Returns

A Connection to the new database.

Remarks

This method replaces any database with same name.

See also

- "Configuration interface" on page 115

createFileTransfer method

Creates a FileTransfer object for transfering files to or from MobiLink.

Syntax

```
FileTransfer DatabaseManager.createFileTransfer(
    String fileName,
    int streamType,
    String userName,
    String version
) throws ULjException
```

Parameters

- **fileName** The name of the server file to transfer. Must not contain any path information.
- **streamType** One of the constants defined in the SyncParms class used to identify the type of communication stream
- **userName** The MobiLink user name
- **version** The MobiLink script version

Returns

The FileTransfer object

createObjectStoreTransfer method (Java ME BlackBerry only)

Creates a FileTransfer object for transfering UltraLiteJ database files to or from MobiLink, and storing them into the RIM object store.

Syntax

```
FileTransfer DatabaseManager.createObjectStoreTransfer(
    String fileName,
    int streamType,
    String userName,
    String version
) throws ULjException
```

Parameters

- **fileName** The name of the server file to transfer. Must not contain any path information.
- **streamType** One of the constants defined in the SyncParms class used to identify the type of communication stream
- **userName** The MobiLink user name
- **version** The MobiLink script version

Returns

The FileTransfer object

createSISHTTPListener method (Java ME BlackBerry only)

Creates an HTTP SISListener for server-initiated synchronizations.

Syntax

```
SISListener DatabaseManager.createSISHTTPListener(
    SISRequestHandler handler,
    int port,
    String httpOptions
) throws ULjException
```

Parameters

- **handler** A SISRequestHandler specified to handle SIS requests.
- **port** An HTTP port to listen for server messages.
- **httpOptions** The HTTP options for connecting to the server.

Returns

The HTTPListener for server-initiated synchronizations.

Remarks

4400 is the recommended port setting.

"deviceside=false" is the recommended HTTP option for BlackBerry simulators.

release method

Closes the DatabaseManager to release all connections and to shutdown all databases.

Syntax

```
void DatabaseManager.release() throws ULjException
```

Remarks

Any uncommitted transactions are rolled back.

setErrorLanguage method

Sets the language to use for error messages.

Syntax

```
void DatabaseManager.setErrorLanguage(String lang)
```

Parameters

- **lang** The language code as a double character.

Remarks

Recognized languages are EN, DE, FR, JA, ZH. If an unrecognized language is specified, the system reverts to the default.

In Java SE and BlackBerry Java ME environments, the current Locale is used to determine the default language. In other Java ME environments, this method provides the only way to specify the language. The default language is "EN".

DecimalNumber interface

Describes an exact decimal value and provides decimal arithmetic support for Java platforms where java.math.BigDecimal is not available.

Syntax

```
public interface DecimalNumber
```

Members

All members of DecimalNumber interface, including all inherited members.

Name	Description
"add method"	Adds two DecimalNumbers together and returns the sum.
"divide method"	Divides the first DecimalNumber by the second DecimalNumber and returns the quotient.
"getString method"	Returns the String representation of the DecimalNumber.
"isNull method"	Determines if the DecimalNumber is null.
"multiply method"	Multiplies two DecimalNumbers together and returns the product.
"set method"	Sets the DecimalNumber with a String value.
"setNull method"	Sets the DecimalNumber to null.
"subtract method"	Subtracts the second DecimalNumber from the first DecimalNumber and returns the difference.

add method

Adds two DecimalNumbers together and returns the sum.

Syntax

```
DecimalNumber DecimalNumber.add(
    DecimalNumber num1,
    DecimalNumber num2
) throws ULjException
```

Parameters

- **num1** A first number.
- **num2** A second number.

Returns

The sum of num1 and num2.

divide method

Divides the first DecimalNumber by the second DecimalNumber and returns the quotient.

Syntax

```
DecimalNumber DecimalNumber.divide(
    DecimalNumber num1,
    DecimalNumber num2
) throws ULjException
```

Parameters

- **num1** A dividend.
- **num2** A divisor.

Returns

The quotient of num1 divided by num2.

getString method

Returns the String representation of the DecimalNumber.

Syntax

```
String DecimalNumber.getString() throws ULjException
```

Returns

The String value.

isNull method

Determines if the DecimalNumber is null.

Syntax

```
boolean DecimalNumber.isNull()
```

Returns

true, if the object is null; otherwise, false.

multiply method

Multiplies two DecimalNumbers together and returns the product.

Syntax

```
DecimalNumber DecimalNumber.multiply(
    DecimalNumber num1,
    DecimalNumber num2
) throws ULjException
```

Parameters

- **num1** A multiplicand.
- **num2** A multiplier.

Returns

The product of num1 and num2.

set method

Sets the DecimalNumber with a String value.

Syntax

```
void DecimalNumber.set(String value) throws ULjException
```

Parameters

- **value** A numerical value represented as a String.

setNull method

Sets the DecimalNumber to null.

Syntax

```
void DecimalNumber.setNull() throws ULjException
```

subtract method

Subtracts the second DecimalNumber from the first DecimalNumber and returns the difference.

Syntax

```
DecimalNumber DecimalNumber.subtract(
    DecimalNumber num1,
    DecimalNumber num2
) throws ULjException
```

Parameters

- **num1** A minuend.
- **num2** A subtrahend.

Returns

The difference between num1 and num2.

Domain interface

Describes Domain type information for a column in a table.

Syntax

```
public interface Domain
```

Members

All members of Domain interface, including all inherited members.

Name	Description
"BIG variable"	Domain ID constant for a 64-bit integer (SQL type BIGINT).
"BINARY variable"	Domain ID constant for a variable-length binary object of maximum *size* bytes (SQL type BINARY(*size*)).
"BIT variable"	Domain ID constant for a bit (SQL type BIT).

Name	Description
“DATE variable”	Domain ID constant for a Date (SQL type DATE).
“DOMAIN_MAX variable”	Maximum kinds of Domain types.
“DOUBLE variable”	Domain ID constant for a 8-byte floating point (SQL type DOUBLE).
“INTEGER variable”	Domain ID constant for a 32-bit integer (SQL type INTEGER).
“LONGBINARY variable”	Domain ID constant for an arbitrary long block of binary data (BLOB) (SQL type LONG BINARY).
“LONGBINARYFILE variable”	Domain ID constant for an arbitrary file of data.
“LONGVARCHAR variable”	Domain ID constant for an arbitrary long block of character data (CLOB) (SQL type LONG VARCHAR).
“NUMERIC variable”	Domain ID constant for a numeric value of fixed precision (size) total digits and with *scale* digits after the decimal (SQL type NUMERIC(*precision*,*scale*)).
“REAL variable”	Domain ID constant for a 4-byte floating point (SQL type REAL).
“SHORT variable”	Domain ID constant for a 16-bit integer (SQL type SMALLINT).
“ST_GEOMETRY variable”	Domain ID constant for a geometry (SQL type GEOMETRY).
“TIME variable”	Domain ID constant for a Time (SQL type TIME).
“TIMESTAMP variable”	Domain ID constant for a Timestamp (SQL type TIMESTAMP).
“TIMESTAMP_ZONE variable”	Domain ID constant for a timestamps with time zones (SQL type DATETIMEOFFSET).
“TINY variable”	Domain ID constant for a unsigned 8-bit integer (SQL type TINYINT).
“UNSIGNED_BIG variable”	Domain ID constant for a unsigned 64-bit integer (SQL type UNSIGNED BIGINT).
“UNSIGNED_INTEGER variable”	Domain ID constant for a unsigned 32-bit integer (SQL type UNSIGNED INTEGER).
“UNSIGNED_SHORT variable”	Domain ID constant for a unsigned 16-bit integer (SQL type UNSIGNED SMALLINT).

Name	Description
“UUID variable”	Domain ID constant for a UniqueIdentifier (SQL type UNIQUE-IDENTIFIER).
“VARCHAR variable”	Domain ID constant for a variable-length character string of maximum *size* bytes (SQL type VARCHAR(*size*)).

Remarks

This interface contains constants to denote the various domains, and methods to extract information from a Domain object.

See the *Connection* interface for an example of creating a schema for a simple database.

Types can be classified as follows:

Integer Types:

Domain Constant	SQL Type	Value Range
BIT	BIT	0 or 1
TINY	TINYINT	0 to 255 (unsigned integer using 1 byte of storage)
SHORT	SMALLINT	-32768 to 32767 (signed integer using 2 bytes of storage)
UNSIGNED_SHORT	UNSIGNED SMALLINT	0 to 65535 (unsigned integer using 2 bytes of storage)
INTEGER	INTEGER	-2^{31} to 2^{31} - 1, or -2147483648 to 2147483647 (signed integer using 4 bytes of storage)
UNSIGNED_INTEGER	UNSIGNED INTEGER	0 to 2^{32} - 1, or 0 to 4294967295 (unsigned integer using 4 bytes of storage)
BIG	BIGINT	-2^{63} to 2^{63} - 1, or -9223372036854775808 to 9223372036854775807 (signed integer using 8 bytes of storage)
UNSIGNED_BIG	UNSIGNED BIGINT	0 to 2^{64} - 1, or 0 to 18446744073709551615 (unsigned integer using 8 bytes of storage)

Non-Integer Numeric Types:

Domain Constant	SQL Type	Value Range
REAL	REAL	-3.402823e+38 to 3.402823e+38, with numbers close to zero as small as 1.175495e-38 (single precision floating point number using 4 bytes of storage, rounding errors may occur after the sixth digit)
DOUBLE	DOUBLE	-1.79769313486231e+308 to 1.79769313486231e+308, with numbers close to zero as small as 2.22507385850721e-308 (single precision floating point number using 8 bytes of storage, rounding errors may occur after the fifteenth digit)
NUMERIC	NUMERIC(precision,scale)	any decimal numbers with *precision* (size) total digits and with *scale* digits after the decimal point (no rounding within precision)

Character and Binary Types:

Domain Constant	SQL Type	Size Range
VARCHAR	VARCHAR(size)	1 to 32767 bytes (characters are stored as 1-3 byte UTF-8 characters).When evaluating expressions, the maximum length for a temporary character value is 2048 bytes.
LONGVARCHAR	LONG VARCHAR	Any length (memory permitting).The only operations allowed on LONG VARCHAR columns are to insert, update, or delete them, or to include them in the select-list of a query.
BINARY	BINARY(size)	1 to 32767 bytes.When evaluating expressions, the maximum length for a temporary character value is 2048 bytes.
LONGBINARY	LONG BINARY	Any length (memory permitting).The only operations allowed on LONG BINARY columns are to insert, update, or delete them, or to include them in the select-list of a query.
UUID	UNIQUEIDENTIFIER	always 16 bytes binary with special interpretation.

Date and Time Types:

Domain Constant	SQL Type	Value
DATE	DATE	Year, month, day.

Domain Constant	SQL Type	Value
TIME	TIME	Hour, minute, second, and fraction of a second.
TIMESTAMP	TIMESTAMP	DATE and TIME.
TIMESTAMP_ZONE	TIMESTAMP_ZONE	DATE and TIME with time zone.

BIT columns are not nullable by default. All other types are nullable by default.

BIG variable

Domain ID constant for a 64-bit integer (SQL type BIGINT).

Syntax

```
final short Domain.BIG
```

See also

- "Domain interface" on page 151

BINARY variable

Domain ID constant for a variable-length binary object of maximum *size* bytes (SQL type BINARY(*size*)).

Syntax

```
final short Domain.BINARY
```

See also

- "Domain interface" on page 151

BIT variable

Domain ID constant for a bit (SQL type BIT).

Syntax

```
final short Domain.BIT
```

See also

- "Domain interface" on page 151

DATE variable

Domain ID constant for a Date (SQL type DATE).

Syntax

```
final short Domain.DATE
```

See also

- "Domain interface" on page 151

DOMAIN_MAX variable

Maximum kinds of Domain types.

Syntax

```
final short Domain.DOMAIN_MAX
```

DOUBLE variable

Domain ID constant for a 8-byte floating point (SQL type DOUBLE).

Syntax

```
final short Domain.DOUBLE
```

See also

- "Domain interface" on page 151

INTEGER variable

Domain ID constant for a 32-bit integer (SQL type INTEGER).

Syntax

```
final short Domain.INTEGER
```

See also

- "Domain interface" on page 151

LONGBINARY variable

Domain ID constant for an arbitrary long block of binary data (BLOB) (SQL type LONG BINARY).

Syntax

```
final short Domain.LONGBINARY
```

See also

- “Domain interface” on page 151

LONGBINARYFILE variable

Domain ID constant for an arbitrary file of data.

Syntax

```
final short Domain.LONGBINARYFILE
```

See also

- “Domain interface” on page 151

LONGVARCHAR variable

Domain ID constant for an arbitrary long block of character data (CLOB) (SQL type LONG VARCHAR).

Syntax

```
final short Domain.LONGVARCHAR
```

See also

- “Domain interface” on page 151

NUMERIC variable

Domain ID constant for a numeric value of fixed precision (size) total digits and with *scale* digits after the decimal (SQL type NUMERIC(*precision*,*scale*)).

Syntax

```
final short Domain.NUMERIC
```

See also

- “Domain interface” on page 151

REAL variable

Domain ID constant for a 4-byte floating point (SQL type REAL).

Syntax

```
final short Domain.REAL
```

See also

- “Domain interface” on page 151

SHORT variable

Domain ID constant for a 16-bit integer (SQL type SMALLINT).

Syntax

```
final short Domain.SHORT
```

See also

- “Domain interface” on page 151

ST_GEOMETRY variable

Domain ID constant for a geometry (SQL type GEOMETRY).

Syntax

```
final short Domain.ST_GEOMETRY
```

See also

- “Domain interface” on page 151

TIME variable

Domain ID constant for a Time (SQL type TIME).

Syntax

```
final short Domain.TIME
```

See also

- “Domain interface” on page 151

TIMESTAMP variable

Domain ID constant for a Timestamp (SQL type TIMESTAMP).

Syntax

```
final short Domain.TIMESTAMP
```

See also

- “Domain interface” on page 151

TIMESTAMP_ZONE variable

Domain ID constant for a timestamps with time zones (SQL type DATETIMEOFFSET).

Syntax

```
final short Domain.TIMESTAMP_ZONE
```

See also

- “Domain interface” on page 151

TINY variable

Domain ID constant for a unsigned 8-bit integer (SQL type TINYINT).

Syntax

```
final short Domain.TINY
```

See also

- “Domain interface” on page 151

UNSIGNED_BIG variable

Domain ID constant for a unsigned 64-bit integer (SQL type UNSIGNED BIGINT).

Syntax

```
final short Domain.UNSIGNED_BIG
```

See also

- “Domain interface” on page 151

UNSIGNED_INTEGER variable

Domain ID constant for a unsigned 32-bit integer (SQL type UNSIGNED INTEGER).

Syntax

```
final short Domain.UNSIGNED_INTEGER
```

See also

- "Domain interface" on page 151

UNSIGNED_SHORT variable

Domain ID constant for a unsigned 16-bit integer (SQL type UNSIGNED SMALLINT).

Syntax

```
final short Domain.UNSIGNED_SHORT
```

See also

- "Domain interface" on page 151

UUID variable

Domain ID constant for a UniqueIdentifier (SQL type UNIQUEIDENTIFIER).

Syntax

```
final short Domain.UUID
```

See also

- "Domain interface" on page 151

VARCHAR variable

Domain ID constant for a variable-length character string of maximum *size* bytes (SQL type VARCHAR(*size*)).

Syntax

```
final short Domain.VARCHAR
```

See also

- "Domain interface" on page 151

EncryptionControl interface

Provides EncryptionControl functionality for the database.

Syntax

```
public interface EncryptionControl
```

Members

All members of EncryptionControl interface, including all inherited members.

Name	Description
"decrypt method"	Decrypts a byte array in the database.
"encrypt method"	Encrypts a byte array in the database.
"initialize method"	Initializes the encryption control with a password.

Remarks

This interface is used to implement your own encryption or obfuscation techniques. To encrypt a database, create a new class that implements the EncryptionControl interface, supplying the class with your own encryption methods, and then use the setEncryption method from the ConfigPersistent interface to create a configuration object that creates and accesses your encrypted database.

Note
Columns stored as files (defined as LONG BINARY STORE AS FILE) are not encrypted or decrypted. Use the BlackBerry builtin SDCard encryption if additional security is needed. Note that files encrypted using the builtin security can not be accessed while the device is locked.

Normal LONG BINARY columns are stored in the database. Therefore, encrypted and decrypted with the rest of the database.

Databases encrypted using an EncryptionControl on BlackBerry devices can be accessed while the device is locked. The **UltraLiteJ Security on BlackBerry Devices** white paper contains information about the security options and concerns for UltraLiteJ applications on BlackBerry devices, including how the builtin security of the device can work for and against applications. See SQL Anywhere White Paper Technical Documents.

See also

- "setEncryption method" on page 109

decrypt method

Decrypts a byte array in the database.

Syntax

```
void EncryptionControl.decrypt(
    int page_no,
    byte[] src,
    byte[] tgt,
    int num_bytes
) throws ULjException
```

Parameters

- **page_no** The page number of the array data.
- **src** The encrypted source page. This array must not be modified.
- **tgt** The resulting page that is decrypted by the method.
- **num_bytes** The number of bytes to decrypt. This is always either the page size or 128.

Remarks

This method is supplied with an encrypted byte array, src, and an associated page number. Your method must decrypt first num_bytes bytes from the src byte array and store the result into the tgt byte array. tgt is then used for data operations within your application.

Any algorithm used must preserve the size of the data (encrypted data is the same length as the original data) and must be able to decrypt partial pages (page 0 is first read and decrypted as a 128 bytes long page and then read and decrypted as a full size page). The src array must not be modified.

encrypt method

Encrypts a byte array in the database.

Syntax

```
void EncryptionControl.encrypt(
    int page_no,
    byte[] src,
    byte[] tgt
) throws ULjException
```

Parameters

- **page_no** The page number of the array data.
- **src** The decrypted source page. This array must not be modified.
- **tgt** The resulting page that is encrypted by the method.

Remarks

This method is supplied with an unencrypted byte array, src, and an associated page number. Your method must encrypt or obfuscate src and store the result into the tgt byte array. tgt is then stored into the database.

Any algorithm used must preserve the size of the data (encrypted data is the same length as the original data) and must be able to decrypt partial pages (page 0 is first read and decrypted as a 128 bytes long page and then read and decrypted as a full size page). The src array must not be modified.

initialize method

Initializes the encryption control with a password.

Syntax

```
void EncryptionControl.initialize(String password) throws ULjException
```

Parameters

- **password** The password used for encryption and decryption.

FileTransfer interface

Provides a mechanism to transfer files between the client and a MobiLink server.

Syntax

```
public interface FileTransfer
```

Members

All members of FileTransfer interface, including all inherited members.

Name	Description
"downloadFile method"	Downloads the file with the specified properties of this object.
"getAuthenticationParms method"	Returns parameters provided to a custom user authentication script.
"getAuthStatus method"	Returns the authorization status code of the last file transfer attempt.
"getAuthValue method"	Returns the value specified in custom user authentication synchronization scripts.
"getFileAuthCode method"	Returns the return value from the authenticate_file_transfer script for the last file transfer attempt.
"getLivenessTimeout method"	Returns the liveness timeout length, in seconds.
"getLocalFileName method"	Determines the local file name.
"getLocalPath method"	Specifies where to find or store the file in the local file system.
"getPassword method"	Returns the MobiLink password for the user specified with setUserName.

Name	Description
“getRemoteKey method”	Determines the current remote key value.
“getServerFileName method”	Returns the name of the file on the server.
“getStreamErrorCode method”	Returns the error code reported by the stream.
“getStreamErrorMessage method”	Returns the error message reported by the stream itself.
“getStreamParms method”	Returns the parameters used to configure the synchronization stream.
“getUserName method”	Returns the MobiLink user name that uniquely identifies the client to the MobiLink server.
“getVersion method”	Returns the synchronization script to use.
“isResumePartialTransfer method”	Determines whether to resume or discard a previous partial transfer.
“isTransferredFile method”	Checks whether the file was actually downloaded during the last file transfer attempt.
“setAuthenticationParms method”	Specifies parameters for a custom user authentication script (MobiLink authenticate_parameters connection event).
“setLivenessTimeout method”	Sets the liveness timeout length, in seconds.
“setLocalFileName method”	Specifies the local file name.
“setLocalPath method”	Specifies where to find or store the file in the local file system.
“setPassword method”	Sets the MobiLink password for the user specified with setUserName.
“setRemoteKey method”	Specifies the remote key.
“setResumePartialTransfer method”	Specifies whether to resume or discard a previous partial transfer.
“setServerFileName method”	Specifies the name of the file on the server.
“setUserName method”	Sets the MobiLink user name that uniquely identifies the client to the MobiLink server.
“setVersion method”	Sets the synchronization script to use.
“uploadFile method”	Uploads the file with the specified properties of this object.

Remarks

A FileTransfer object is obtained by calling the createFileTransfer() or createObjectStoreTransfer() (BlackBerry only) methods of the DatabaseManager class.

The instance returned by *createFileTransfer()* can be used to transfer any files between MobiLink and the local file system. For the BlackBerry 4.2 and later devices and simulators, the local file system is either a media card or internal flash file system (if available on the device). For file name restrictions, see the description of the createConfigurationFileME method on the DatabaseManager class.

The instance returned by *createObjectStoreTransfer()* can be used to download UltraLiteJ database files to the local BlackBerry object store, or vice versa. Only valid, unencrypted UltraLiteJ database files could be transferred. Attempting to download any files other than UltraLiteJ databases will result in exceptions being thrown.

Note that the application should not simutaneously start two downloads to the same local file.

See also

- "createConfigurationFileME method (BlackBerry only)" on page 143
- "createFileTransfer method" on page 145
- "createObjectStoreTransfer method (Java ME BlackBerry only)" on page 146

downloadFile method

Downloads the file with the specified properties of this object.

Overload list

Name	Description
"downloadFile() method"	Downloads the file with the specified properties of this object.
"downloadFile(FileTransfer-ProgressListener) method"	Download the file specified by the properties of this object with progress events posted to the specified listener.

downloadFile() method

Downloads the file with the specified properties of this object.

Syntax

```
abstract boolean FileTransfer.downloadFile() throws ULjException
```

Returns

True if download is successful; otherwise a ULjException will be thrown and the method does not return normally.

Remarks

The file specified by the *setServerFileName()* method is downloaded from the MobiLink server to the path specified by *setLocalPath()* using the specified stream, userName, password, and script version.

Further options can be specified using the *setLocalFileName()*, *setAuthenticationParms()* and *setResumePartialTransfer()* methods.

To avoid file corruption, for desktop and BlackBerry file system downloads, UltraLiteJ will download to a temporary file and only replace the local file once the download has completed.

For BlackBerry object store downloads, UltraLiteJ will start to write to the local store directly. This is because, in this case, it is not possible to create atomic temporary objects. If there is an existing local database store with the same name, it will be corrupted upon invoking *transferFile()*.

A detailed result status can be fetched using the *getAuthStatus()*, *getAuthValue()*, *getFileAuthCode()*, *isTransferredFile()*, *getStreamErrorCode()*, and *getStreamErrorMessage()* methods.

downloadFile(FileTransferProgressListener) method

Download the file specified by the properties of this object with progress events posted to the specified listener.

Syntax

```
abstract boolean FileTransfer.downloadFile(
    FileTransferProgressListener listener
) throws ULjException
```

Parameters

- **listener** The object that receives file transfer progress events.

Returns

True if download is successful; otherwise a ULjException will be thrown and the method does not return normally.

Remarks

Errors may result in no data being sent to the listener.

See also

- "downloadFile method" on page 165

getAuthenticationParms method

Returns parameters provided to a custom user authentication script.

Syntax

```
abstract String FileTransfer.getAuthenticationParms()
```

Returns

The list of authentication parms or null if no parameters are specified.

See also

- "setAuthenticationParms method" on page 171

getAuthStatus method

Returns the authorization status code of the last file transfer attempt.

Syntax

```
abstract int FileTransfer.getAuthStatus()
```

Returns

An AuthStatusCode value.

getAuthValue method

Returns the value specified in custom user authentication synchronization scripts.

Syntax

```
abstract long FileTransfer.getAuthValue()
```

Returns

An integer returned from custom user authentication synchronization scripts.

getFileAuthCode method

Returns the return value from the authenticate_file_transfer script for the last file transfer attempt.

Syntax

```
abstract int FileTransfer.getFileAuthCode()
```

Returns

An integer returned from the authenticate_file_transfer script for the last file transfer attempt.

getLivenessTimeout method

Returns the liveness timeout length, in seconds.

Syntax

```
abstract int FileTransfer.getLivenessTimeout()
```

Returns

The timeout.

See also

- “setLivenessTimeout method” on page 172

getLocalFileName method

Determines the local file name.

Syntax

```
abstract String FileTransfer.getLocalFileName()
```

Returns

the local file name for the downloaded file.

See also

- “setLocalFileName method” on page 172

getLocalPath method

Specifies where to find or store the file in the local file system.

Syntax

```
abstract String FileTransfer.getLocalPath()
```

Returns

the local directory

See also

- “setLocalPath method” on page 173

getPassword method

Returns the MobiLink password for the user specified with setUserName.

Syntax

```
abstract String FileTransfer.getPassword()
```

Returns

The password for the MobiLink user.

See also

- “setPassword method” on page 173

getRemoteKey method

Determines the current remote key value.

Syntax

```
abstract String FileTransfer.getRemoteKey()
```

Returns

The remote key value or null if the remote key is unspecified.

See also

- “setRemoteKey method” on page 174

getServerFileName method

Returns the name of the file on the server.

Syntax

```
abstract String FileTransfer.getServerFileName()
```

Returns

The name of the file in the server side.

See also

- “setServerFileName method” on page 175

getStreamErrorCode method

Returns the error code reported by the stream.

Syntax

```
abstract int FileTransfer.getStreamErrorCode()
```

Returns

Zero, if there was no communication stream error; otherwise, the response code from the server.

getStreamErrorMessage method

Returns the error message reported by the stream itself.

Syntax

```
abstract String FileTransfer.getStreamErrorMessage()
```

Returns

Null, if no message is available; otherwise, the response message.

Remarks

This is the HTTP response message.

getStreamParms method

Returns the parameters used to configure the synchronization stream.

Syntax

```
abstract StreamHTTPParms FileTransfer.getStreamParms()
```

Returns

A StreamHTTPParms interface or StreamHTTPSParms object specifying the parameters for HTTP or HTTPS streams. The object is returned by reference.

getUserName method

Returns the MobiLink user name that uniquely identifies the client to the MobiLink server.

Syntax

```
abstract String FileTransfer.getUserName()
```

Returns

The MobiLink user name.

See also

- "setUserName method" on page 176

getVersion method

Returns the synchronization script to use.

Syntax

```
abstract String FileTransfer.getVersion()
```

Returns

The script version.

See also

- “setVersion method” on page 176

isResumePartialTransfer method

Determines whether to resume or discard a previous partial transfer.

Syntax

```
abstract boolean FileTransfer.isResumePartialTransfer()
```

Returns

true if to resume the download; otherwise, false.

See also

- “setResumePartialTransfer method” on page 174

isTransferredFile method

Checks whether the file was actually downloaded during the last file transfer attempt.

Syntax

```
abstract boolean FileTransfer.isTransferredFile()
```

Returns

True if the file was transferred, false otherwise.

setAuthenticationParms method

Specifies parameters for a custom user authentication script (MobiLink authenticate_parameters connection event).

Syntax

```
abstract void FileTransfer.setAuthenticationParms(
    String authParms
) throws ULjException
```

Parameters

- **authParms** A comma separated list of authentication parameters, or the null reference. See class description of the *SyncParms* class for more on comma separated lists.

Remarks

Only the first 255 strings are used and each string should be no longer than 128 characters (longer strings will be truncated when sent to MobiLink).

See also

- "getAuthenticationParms method" on page 166

setLivenessTimeout method

Sets the liveness timeout length, in seconds.

Syntax

```
abstract void FileTransfer.setLivenessTimeout(
    int timeout
) throws ULjException
```

Parameters

- **timeout** The new liveness timeout value.

Remarks

The liveness timeout is the length of time the server allows a remote to be idle. If the remote does not communicate with the server for l seconds, the server assumes that the remote has lost the connection, and terminates the file transfer. The remote automatically sends periodic messages to the server to keep the connection alive.

If a negative value is set, an exception is thrown. The value may be changed by the MobiLink server without notice. This change occurs if the value is set too low or too high.

The default value is 100 seconds.

See also

- "getLivenessTimeout method" on page 167

setLocalFileName method

Specifies the local file name.

Syntax

```
abstract void FileTransfer.setLocalFileName(String localFileName)
```

Parameters

- **localFileName** A string specifying the local file name for the downloaded file. If the value is a null reference, fileName is used. The default is a null reference.

Remarks

For file downloads, this will be the name of the downloaded file. For file uploads, this will be the name of the file to upload. The file name must not include any drive of path information.

See also

- "getLocalFileName method" on page 168
- "setLocalPath method" on page 173

setLocalPath method

Specifies where to find or store the file in the local file system.

Syntax

```
abstract void FileTransfer.setLocalPath(String localPath)
```

Parameters

- **localPath** A string specifying the local directory of the file. The default is a null reference.

Remarks

The syntax of the local directory varies among platforms:

- For the desktop, the syntax would be like "C:\\ulj\\"
- For the BlackBerry file system, the syntax would be like "file:///SDCard/ulj/"
- For the BlackBerry object store, this option is ignored.

The default local directory also varies depending on the device's operating system:

See also

- "getLocalPath method" on page 168
- "setLocalFileName method" on page 172

setPassword method

Sets the MobiLink password for the user specified with setUserName.

Syntax

```
abstract void FileTransfer.setPassword(
    String password
) throws ULjException
```

Parameters

- **password** A password for the MobiLink user.

Remarks

This user name and password is separate from any database user ID and password. This method is used to authenticate the application against the MobiLink server.

The default is an empty string, suggesting no password.

See also

- “getPassword method” on page 168
- “setUserName method” on page 176

setRemoteKey method

Specifies the remote key.

Syntax

```
abstract void FileTransfer.setRemoteKey(String remoteKey)
```

Parameters

- **remoteKey** The remote key value or null to leave it unspecified.

Remarks

The remote key is a parameter passed to the server authenticate_file_upload script. The script can use this parameter to determine the name and location of the file to be stored on the server.

See also

- “getRemoteKey method” on page 169

setResumePartialTransfer method

Specifies whether to resume or discard a previous partial transfer.

Syntax

```
abstract void FileTransfer.setResumePartialTransfer(boolean resume)
```

Parameters

- **resume** Sets true to resume a previous partial download, false to discard a previous partial download.

Remarks

The default is true.

UltraLiteJ has the ability to restart file transfers that fail because of communication errors or user aborts through the FileTransferProgressListener.

For file downloads, UltraLiteJ processes the download as it is received. If a download is interrupted, then the partially download file is retained and can be resumed during the next file transfer. If the file has been updated on the server, the partial download will be discarded and a new download started.

For file uploads, the MobiLink server will keep partially uploaded files so that a subsequent file upload can resume a previous one. However, if the file has been updated locally, the partial upload will be discarded and a new upload started.

See also

- "isResumePartialTransfer method" on page 171

setServerFileName method

Specifies the name of the file on the server.

Syntax

```
abstract void FileTransfer.setServerFileName(
    String fileName
) throws ULjException
```

Parameters

- **fileName** A string specifying the name of the file as recognized by the MobiLink server.

Remarks

For file downloads, this will be the name of the file to download. For file uploads, this will be the name of the uploaded file.

This parameter is initalized when the FileTransfer object is created.

FileName is the name of the file on the server running MobiLink. MobiLink will first search for this file in the userName subdirectory and then in the root directory (the root download directory is specified via the MobiLink server's -ftr option, and the root upload directory is specified via the -ftru option). FileName must not include any drive or path information, or the MobiLink server will be unable to find it. For example, "myfile.txt" is valid, but "somedir\myfile.txt", "..\myfile.txt", and "c:\myfile.txt" are all invalid.

See also

- “getServerFileName method” on page 169

setUserName method

Sets the MobiLink user name that uniquely identifies the client to the MobiLink server.

Syntax

```
abstract void FileTransfer.setUserName(
    String userName
) throws ULjException
```

Parameters

- **userName** The MobiLink user name.

Remarks

The MobiLink server uses this value to locate the file in the server side. The MobiLink user name and password are separate from any database user ID and password, and serve to identify and authenticate the application to the MobiLink server.

This parameter is initalized when the FileTransfer object is created.

See also

- “getUserName method” on page 170
- “setPassword method” on page 173

setVersion method

Sets the synchronization script to use.

Syntax

```
abstract void FileTransfer.setVersion(
    String version
) throws ULjException
```

Parameters

- **version** The script version.

Remarks

Each synchronization script in the consolidated database is marked with a version string. The version string allows an UltraLiteJ application to choose from a set of synchronization scripts.

See also

- "getVersion method" on page 170

uploadFile method

Uploads the file with the specified properties of this object.

Overload list

Name	Description
"uploadFile() method"	Uploads the file with the specified properties of this object.
"uploadFile(FileTransferProgressListener) method"	Upload the file specified by the properties of this object with progress events posted to the specified listener.

uploadFile() method

Uploads the file with the specified properties of this object.

Syntax

```
abstract boolean FileTransfer.uploadFile() throws ULjException
```

Returns

True if upload is successful; otherwise a ULjException will be thrown and the method does not return normally.

Remarks

The file specified by the *setLocalFileName* and *setLocalPath* methods is uploaded to the MobiLink server to the file specified by the *setServerFileName* method using the specified stream, userName, password, and script version.

Further options can be specified using the *setAuthenticationParms()* and *setResumePartialTransfer()* methods.

A detailed result status can be fetched using the *getAuthStatus()*, *getAuthValue()*, *getFileAuthCode()*, *isTransferredFile()*, *getStreamErrorCode()*, and *getStreamErrorMessage()* methods.

uploadFile(FileTransferProgressListener) method

Upload the file specified by the properties of this object with progress events posted to the specified listener.

Syntax

```
abstract boolean FileTransfer.uploadFile(
    FileTransferProgressListener listener
) throws ULjException
```

Parameters

- **listener** The object that receives file transfer progress events.

Returns

True if upload is successful; otherwise a ULjException will be thrown and the method does not return normally.

Remarks

Errors may result in no data being sent to the listener.

See also

- "uploadFile method" on page 177

FileTransferProgressData interface

Reports file transfer progress monitoring data.

Syntax

```
public interface FileTransferProgressData
```

Members

All members of FileTransferProgressData interface, including all inherited members.

Name	Description
"getBytesTransferred method"	Returns the number of bytes transferred so far.
"getFileSize method"	Returns the size of the file being transferred.
"getResumedAtSize method"	Returns the point in the file where the transfer was resumed.

getBytesTransferred method

Returns the number of bytes transferred so far.

Syntax

```
abstract long FileTransferProgressData.getBytesTransferred()
```

Returns

The number of bytes transferred so far.

getFileSize method

Returns the size of the file being transferred.

Syntax

```
abstract long FileTransferProgressData.getFileSize()
```

Returns

The size of the file in bytes.

getResumedAtSize method

Returns the point in the file where the transfer was resumed.

Syntax

```
abstract long FileTransferProgressData.getResumedAtSize()
```

Returns

The number of bytes transferred previously.

FileTransferProgressListener interface

The listener interface for receiving file transfer progress events.

Syntax

```
public interface FileTransferProgressListener
```

Members

All members of FileTransferProgressListener interface, including all inherited members.

Name	Description
"fileTransferProgressed method"	Invoked during a file transfer to inform the user of progress.

Remarks

To receive progress reports during file transfer, you must create a new class that performs the task.

The following example illustrates a simple SyncObserver interface:

```
class MyObserver implements FileTransferProgressListener {
   public boolean fileTransferProgressed( FileTransferProgressData data ) {
       System.out.println(
           "file transfer progress "
           + " bytes received = " + data.getBytesTransferred()
       );
       return false;   // Always continue file transfer.
   }
   public MyObserver() {} // The default constructor.
 }
```

fileTransferProgressed method

Invoked during a file transfer to inform the user of progress.

Syntax

```
boolean FileTransferProgressListener.fileTransferProgressed(
    FileTransferProgressData data
)
```

Parameters

- **data** A FileTransferProgressData object containing the latest file transfer progress data.

Returns

This method should return true to cancel the transfer or return false to continue.

Remarks

The listener will be called:

- Before the first disk write
- After every disk write or every 0.5 seconds, whichever is later
- After the file download is complete

This method should return true to cancel the transfer or return false to continue.

Usually, the cancel request will be accepted by UltraLiteJ. This will result in a ULjException being thrown with the errorCode set to ULjException.SQLE_INTERRUPTED. However, if the download has been completed for BlackBerry object store, UltraLiteJ will not cancel the transfer.

IndexSchema interface

Specifies the schema of an index and provides constants useful for querying system tables.

Syntax

```
public interface IndexSchema
```

Members

All members of IndexSchema interface, including all inherited members.

Name	Description
"ASCENDING variable"	The index is sorted in ascending order for a column.
"DESCENDING variable"	The index is sorted in descending order for a column.
"PERSISTENT variable"	Bit flag denoting an index is persistent.
"PRIMARY_INDEX variable"	Bit flag denoting an index is a primary key.
"UNIQUE_INDEX variable"	Bit flag denoting an index is a unique index.
"UNIQUE_KEY variable"	Bit flag denoting an index is a unique key.

Remarks

This interface only contains index-related constants, including flags and the sort order of indices.

ASCENDING variable

The index is sorted in ascending order for a column.

Syntax

```
final byte IndexSchema.ASCENDING
```

DESCENDING variable

The index is sorted in descending order for a column.

Syntax

```
final byte IndexSchema.DESCENDING
```

PERSISTENT variable

Bit flag denoting an index is persistent.

Syntax

```
final byte IndexSchema.PERSISTENT
```

Remarks

This value can be logically combined with other flags in table SYS_INDEXES' index_flags column.

PRIMARY_INDEX variable

Bit flag denoting an index is a primary key.

Syntax

```
final byte IndexSchema.PRIMARY_INDEX
```

Remarks

This value can be logically combined with other flags in table SYS_INDEXES' index_flags column.

UNIQUE_INDEX variable

Bit flag denoting an index is a unique index.

Syntax

```
final byte IndexSchema.UNIQUE_INDEX
```

Remarks

This value can be logically combined with other flags in table SYS_INDEXES' index_flags column.

UNIQUE_KEY variable

Bit flag denoting an index is a unique key.

Syntax

```
final byte IndexSchema.UNIQUE_KEY
```

Remarks

This value can be logically combined with other flags in table SYS_INDEXES' index_flags column.

PreparedStatement interface

Provides methods to execute a SQL query to generate a ResultSet, or to execute a prepared SQL statement on an UltraLite database.

Syntax

```
public interface PreparedStatement
```

Members

All members of PreparedStatement interface, including all inherited members.

Name	Description
"close method"	Closes the PreparedStatement to release the memory resources associated with it.
"execute method"	Executes the prepared SQL statement.
"executeQuery method"	Executes the prepared SQL SELECT statement and returns a ResultSet.
"getBlobOutputStream method"	Returns an OutputStream.
"getClobWriter method"	Returns a Writer.
"getOrdinal method"	Returns the (base-one) ordinal for the value represented by the name.
"getPlan method"	Returns a text-based description of the SQL query execution plan.
"getPlanTree method"	Returns a text-based description of the SQL query execution plan, represented as a tree.
"getResultSet method"	Returns the ResultSet for a prepared SQL statement.
"getUpdateCount method"	Returns the number of rows inserted, updated or deleted since the last execute statement.
"hasResultSet method"	Determines if the PreparedStatement contains a ResultSet.
"set method"	Sets a value to the host variable in the SQL statement.
"setNull method"	Sets a null value to the host variable in the SQL statement that is defined by ordinal.

Remarks

The following example demonstrates how to execute a PreparedStatement, check if the execution created a ResultSet, save the ResultSet to a local variable, and close the PreparedStatement:

```
// Create a new PreparedStatement object from an existing connection.
String sql_string = "SELECT * FROM SampleTable";
PreparedStatement ps = conn.prepareStatement(sql_string);

// result returns true if the execute statement runs successfully.
boolean result = ps.execute();

// Check if the PreparedStatement contains a ResultSet.
if (ps.hasResultSet()) {
     // Store the ResultSet in the rs variable.
     ResultSet rs = ps.getResultSet;
}
```

```
// Close the PreparedStatement to release resources.
ps.close();


When a statement contains expressions, it may contain a host variable wherever
a column name could appear. Host variables are entered as either a '?'
characters (unnamed host variables) or as ":name" (named host variable). In
the example,
     SELECT * FROM SampleTable WHERE pk > :bound AND pk < ?
 there are two host variables which may be set using the PreparedStatement
that was prepared for the SQL statement in question.
```

See also

- "Connection interface" on page 117
- "prepareStatement method" on page 129

close method

Closes the PreparedStatement to release the memory resources associated with it.

Syntax

void **PreparedStatement.close**() throws **ULjException**

Remarks

No further methods can be used on this object. If the PreparedStatement contains a ResultSet, the ResultSet also gets closed.

execute method

Executes the prepared SQL statement.

Syntax

boolean **PreparedStatement.execute**() throws **ULjException**

Returns

true, if the execute statement runs successfully; otherwise, false.

See also

- "ResultSet interface" on page 198

executeQuery method

Executes the prepared SQL SELECT statement and returns a ResultSet.

Syntax

```
ResultSet PreparedStatement.executeQuery() throws ULjException
```

Returns

The ResultSet containing the query result of the prepared SQL SELECT statement.

See also

- "ResultSet interface" on page 198

getBlobOutputStream method

Returns an OutputStream.

Overload list

Name	Description
"getBlobOutputStream(int) method"	Returns an OutputStream.
"getBlobOutputStream(String) method"	Returns an OutputStream.

getBlobOutputStream(int) method

Returns an OutputStream.

Syntax

```
java.io.OutputStream PreparedStatement.getBlobOutputStream(
    int ordinal
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.

Returns

The OutputStream for the named value

getBlobOutputStream(String) method

Returns an OutputStream.

Syntax

```
java.io.OutputStream PreparedStatement.getBlobOutputStream(
    String name
) throws ULjException
```

Parameters

- **name** A String representing the host variable name.

Returns

The OutputStream for the named value

getClobWriter method

Returns a Writer.

Overload list

Name	Description
"getClobWriter(int) method"	Returns a Writer.
"getClobWriter(String) method"	Returns a Writer.

getClobWriter(int) method

Returns a Writer.

Syntax

```
java.io.Writer PreparedStatement.getClobWriter(
    int ordinal
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.

Returns

The Writer for the named value

getClobWriter(String) method

Returns a Writer.

Syntax

```
java.io.Writer PreparedStatement.getClobWriter(
    String name
) throws ULjException
```

Parameters

- **name** A String representing the host variable name.

Returns

The Writer for the named value

getOrdinal method

Returns the (base-one) ordinal for the value represented by the name.

Syntax

```
int PreparedStatement.getOrdinal(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The (base-one) ordinal for the value represented by the name.

getPlan method

Returns a text-based description of the SQL query execution plan.

Syntax

```
String PreparedStatement.getPlan() throws ULjException
```

Returns

The String representation of the plan.

Remarks

This method is intended for use during development.

This plan contains the same information as is presented by the getPlanTree method. The difference is in the presentation.

An empty string is returned if there is no plan. Plans exist when the prepared statement is a SQL query.

The plan shows the operations used to execute the query when the plan is obtained before the associated query has been executed. Additionally, the plan shows the number of rows that each operation produced when the plan is obtained after the query has been executed. This plan can be used to gain insight about the execution of the query.

The following is an example of a plan tree, expressed as a String. It is displayed on multiple lines with '|' characters to represent the structure.

```
SELECT * FROM tab1, tab2 WHERE col1 > pk2
row: 2 20 10 banana
row: 3 30 10 banana
```

```
row: 4 40 10 banana
row: 4 40 30 peach
row: 5 50 10 banana
row: 5 50 30 peach
row: 5 50 40 apple
plan:
 root:7
 |
 inner-join:7
 | |
 | index-scan:7[tab2,prime_key]
 |
 table-scan:5[tab1,prime_key]
```

See also

- "getPlanTree method" on page 188

getPlanTree method

Returns a text-based description of the SQL query execution plan, represented as a tree.

Syntax

String **PreparedStatement.getPlanTree**( ) throws **ULjException**

Returns

The String representation of the plan, represented as a tree.

Remarks

This method is intended for use during development.

This plan contains the same information as is presented by the getPlan method. The difference is in the presentation.

An empty string is returned if there is no plan. Plans exist when the prepared statement is a SQL query.

The plan shows the operations used to execute the query when the plan is obtained before the associated query has been executed. Additionally, the plan shows the number of rows that each operation produced when the plan is obtained after the query has been executed. This plan can be used to gain insight about the execution of the query.

The following is an example of a plan tree, expressed as a String. It is displayed on multiple lines with '|' characters to represent the structure.

```
SELECT * FROM tab1, tab2 WHERE col1 > pk2
row: 2 20 10 banana
row: 3 30 10 banana
row: 4 40 10 banana
row: 4 40 30 peach
row: 5 50 10 banana
row: 5 50 30 peach
row: 5 50 40 apple
plan:
```

```
root:7
|
inner-join:7
| |
| index-scan:7[tab2,prime_key]
|
table-scan:5[tab1,prime_key]
```

See also

- “getPlan method” on page 187

getResultSet method

Returns the ResultSet for a prepared SQL statement.

Syntax

`ResultSet **PreparedStatement.getResultSet**() throws **ULjException**`

Returns

The ResultSet containing the query result of the prepared SQL. statement.

See also

- “ResultSet interface” on page 198

getUpdateCount method

Returns the number of rows inserted, updated or deleted since the last execute statement.

Syntax

`int **PreparedStatement.getUpdateCount**() throws **ULjException**`

Returns

Returns -1 when a statement cannot perform changes; otherwise, returns the number of rows that have been changed.

hasResultSet method

Determines if the PreparedStatement contains a ResultSet.

Syntax

`boolean **PreparedStatement.hasResultSet**() throws **ULjException**`

Returns

true, if a ResultSet was found; otherwise, false.

See also

- "ResultSet interface" on page 198

set method

Sets a value to the host variable in the SQL statement.

Overload list

Name	Description
"set(int, boolean) method"	Sets a boolean value to the host variable in the SQL statement that is defined by ordinal.
"set(int, byte[]) method"	Sets a byte array value to the host variable in the SQL statement that is defined by ordinal.
"set(int, Date) method"	Sets a java.util.Date to the host variable in the SQL statement that is defined by ordinal.
"set(int, DecimalNumber) method"	Sets a DecimalNumber to the host variable in the SQL statement that is defined by ordinal.
"set(int, double) method"	Sets a double value to the host variable in the SQL statement that is defined by ordinal.
"set(int, float) method"	Sets a float value to the host variable in the SQL statement that is defined by ordinal.
"set(int, int) method"	Sets an integer value to the host variable in the SQL statement that is defined by ordinal.
"set(int, long) method"	Sets a long integer value to the host variable in the SQL statement that is defined by ordinal.
"set(int, String) method"	Sets a String value to the host variable in the SQL statement that is defined by ordinal.
"set(int, UUIDValue) method"	Sets a UUIDValue value to the host variable in the SQL statement that is defined by ordinal.
"set(String, boolean) method"	Sets a boolean value to the host variable in the SQL statement that is defined by name.

Name	Description
"set(String, byte[]) method"	Sets a byte array value to the host variable in the SQL statement that is defined by name.
"set(String, Date) method"	Sets a java.util.Date to the host variable in the SQL statement that is defined by ordinal.
"set(String, DecimalNumber) method"	Sets a DecimalNumber to the host variable in the SQL statement that is defined by name.
"set(String, double) method"	Sets a double value to the host variable in the SQL statement that is defined by name.
"set(String, float) method"	Sets a float value to the host variable in the SQL statement that is defined by name.
"set(String, int) method"	Sets an integer value to the host variable in the SQL statement that is defined by name.
"set(String, long) method"	Sets a long integer value to the host variable in the SQL statement that is defined by name.
"set(String, String) method"	Sets a String value to the host variable in the SQL statement that is defined by name.
"set(String, UUIDValue) method"	Sets a UUIDValue value to the host variable in the SQL statement that is defined by name.

set(int, boolean) method

Sets a boolean value to the host variable in the SQL statement that is defined by ordinal.

Syntax

```
void PreparedStatement.set(
    int ordinal,
    boolean value
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.
- **value** The value to be set.

set(int, byte[]) method

Sets a byte array value to the host variable in the SQL statement that is defined by ordinal.

Syntax

```
void PreparedStatement.set(
    int ordinal,
    byte[] value
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.
- **value** The value to be set.

set(int, Date) method

Sets a java.util.Date to the host variable in the SQL statement that is defined by ordinal.

Syntax

```
void PreparedStatement.set(
    int ordinal,
    java.util.Date value
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.
- **value** The value to be set.

set(int, DecimalNumber) method

Sets a DecimalNumber to the host variable in the SQL statement that is defined by ordinal.

Syntax

```
void PreparedStatement.set(
    int ordinal,
    DecimalNumber value
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.
- **value** The value to be set.

set(int, double) method

Sets a double value to the host variable in the SQL statement that is defined by ordinal.

Syntax

```
void PreparedStatement.set(
    int ordinal,
    double value
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.
- **value** The value to be set.

set(int, float) method

Sets a float value to the host variable in the SQL statement that is defined by ordinal.

Syntax

```
void PreparedStatement.set(int ordinal, float value) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.
- **value** The value to be set.

set(int, int) method

Sets an integer value to the host variable in the SQL statement that is defined by ordinal.

Syntax

```
void PreparedStatement.set(int ordinal, int value) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.
- **value** The value to be set.

set(int, long) method

Sets a long integer value to the host variable in the SQL statement that is defined by ordinal.

Syntax

```
void PreparedStatement.set(int ordinal, long value) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.

- **value** The value to be set.

set(int, String) method

Sets a String value to the host variable in the SQL statement that is defined by ordinal.

Syntax

```
void PreparedStatement.set(
    int ordinal,
    String value
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.

- **value** The value to be set.

set(int, UUIDValue) method

Sets a UUIDValue value to the host variable in the SQL statement that is defined by ordinal.

Syntax

```
void PreparedStatement.set(
    int ordinal,
    UUIDValue value
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.

- **value** The value to be set.

set(String, boolean) method

Sets a boolean value to the host variable in the SQL statement that is defined by name.

Syntax

```
void PreparedStatement.set(
    String name,
    boolean value
) throws ULjException
```

Parameters

- **name** A String representing the host variable name.

- **value** The value to be set.

set(String, byte[]) method

Sets a byte array value to the host variable in the SQL statement that is defined by name.

Syntax

```
void PreparedStatement.set(
    String name,
    byte[] value
) throws ULjException
```

Parameters

- **name** A String representing the host variable name.
- **value** The value to be set.

set(String, Date) method

Sets a java.util.Date to the host variable in the SQL statement that is defined by ordinal.

Syntax

```
void PreparedStatement.set(
    String name,
    java.util.Date value
) throws ULjException
```

Parameters

- **name** A String representing the host variable name.
- **value** The value to be set.

set(String, DecimalNumber) method

Sets a DecimalNumber to the host variable in the SQL statement that is defined by name.

Syntax

```
void PreparedStatement.set(
    String name,
    DecimalNumber value
) throws ULjException
```

Parameters

- **name** A String representing the host variable name.
- **value** The value to be set.

set(String, double) method

Sets a double value to the host variable in the SQL statement that is defined by name.

Syntax

```
void PreparedStatement.set(
    String name,
    double value
) throws ULjException
```

Parameters

- **name** A String representing the host variable name.
- **value** The value to be set.

set(String, float) method

Sets a float value to the host variable in the SQL statement that is defined by name.

Syntax

```
void PreparedStatement.set(String name, float value) throws ULjException
```

Parameters

- **name** A String representing the host variable name.
- **value** The value to be set.

set(String, int) method

Sets an integer value to the host variable in the SQL statement that is defined by name.

Syntax

```
void PreparedStatement.set(String name, int value) throws ULjException
```

Parameters

- **name** A String representing the host variable name.
- **value** The value to be set.

set(String, long) method

Sets a long integer value to the host variable in the SQL statement that is defined by name.

Syntax

```
void PreparedStatement.set(String name, long value) throws ULjException
```

Parameters

- **name** A String representing the host variable name.
- **value** The value to be set.

set(String, String) method

Sets a String value to the host variable in the SQL statement that is defined by name.

Syntax

```
void PreparedStatement.set(
    String name,
    String value
) throws ULjException
```

Parameters

- **name** A String representing the host variable name.
- **value** The value to be set.

set(String, UUIDValue) method

Sets a UUIDValue value to the host variable in the SQL statement that is defined by name.

Syntax

```
void PreparedStatement.set(
    String name,
    UUIDValue value
) throws ULjException
```

Parameters

- **name** A String representing the host variable name.
- **value** The value to be set.

setNull method

Sets a null value to the host variable in the SQL statement that is defined by ordinal.

Overload list

Name	Description
"setNull(int) method"	Sets a null value to the host variable in the SQL statement that is defined by ordinal.

Name	Description
"setNull(String) method"	Sets a null value to the host variable in the SQL statement that is defined by name.

setNull(int) method

Sets a null value to the host variable in the SQL statement that is defined by ordinal.

Syntax

```
void PreparedStatement.setNull(int ordinal) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the host variable as ordered in the SQL statement.

setNull(String) method

Sets a null value to the host variable in the SQL statement that is defined by name.

Syntax

```
void PreparedStatement.setNull(String name) throws ULjException
```

Parameters

- **name** A String representing the host variable name.

ResultSet interface

Provides methods to traverse a table by row, and access the column data.

Syntax

```
public interface ResultSet
```

Members

All members of ResultSet interface, including all inherited members.

Name	Description
"close method"	Closes the ResultSet to release the memory resources associated with it.
"getBlobInputStream method"	Returns an InputStream for long binary types, including file-based long binarys.

Name	Description
"getBoolean method"	Returns a boolean value.
"getBytes method"	Returns a byte array.
"getClobReader method"	Returns a Reader.
"getDate method"	Returns a java.util.Date.
"getDecimalNumber method"	Returns a DecimalNumber.
"getDouble method"	Returns a double value based on the column number.
"getFloat method"	Returns a float value.
"getInt method"	Returns an integer value.
"getLong method"	Returns a long integer value.
"getOrdinal method"	Returns the (base-one) ordinal for the value represented by a String.
"getResultSetMetadata method"	Returns the ResultSetMetadata containing the meta data for the ResultSet.
"getSize method"	Get the actual size of a result set column.
"getString method"	Returns a String value.
"getUUIDValue method"	Returns a UUIDValue value.
"isNull method"	Tests if the value at the specified column number is null.
"next method"	Fetches the next row of data in the ResultSet.
"previous method"	Fetches the previous row of data in the ResultSet.

Remarks

A ResultSet is generated by executing a PreparedStatement with a SQL SELECT statement using the execute or executeQuery methods.

The following example demonstrates how to execute a new PreparedStatement, fetch a row with the ResultSet, and access data from a specified column.

```
// Define a new SQL SELECT statement.
String sql_string = "SELECT column1, column2 FROM SampleTable";

// Create a new PreparedStatement from an existing connection.
PreparedStatement ps = conn.prepareStatement(sql_string);

// Create a new ResultSet to contain the query results of the SQL statement.
```

```
ResultSet rs = ps.executeQuery();

// Check if the PreparedStatement contains a ResultSet.
if (ps.hasResultSet()) {
     // Retrieve the column1 value from the first row using getString.
     String row1_col1 = rs.getString(1);
     // Get the next row in the table.
     if (rs.next) {
          // Retrieve the value of column1 from the second row.
          String row2_col1 = rs.getString(1);
     }
}

rs.close();
ps.close();
```

See also

- "PreparedStatement interface" on page 182
- "execute method" on page 184
- "executeQuery method" on page 184
- "Connection interface" on page 117

close method

Closes the ResultSet to release the memory resources associated with it.

Syntax

```
void ResultSet.close() throws ULjException
```

Remarks

Subsequent attempts to fetch rows from this ResultSet will throw an error.

getBlobInputStream method

Returns an InputStream for long binary types, including file-based long binarys.

Syntax

```
java.io.InputStream ResultSet.getBlobInputStream(
    int ordinal
) throws ULjException
```

Parameters

- **ordinal**   A base-one integer representing the column number as ordered in the SQL statement.

Returns

The InputStream representation of the named value.

getBoolean method

Returns a boolean value.

Overload list

Name	Description
"getBoolean(int) method"	Returns a boolean value.
"getBoolean(String) method"	Returns a boolean value.

getBoolean(int) method

Returns a boolean value.

Syntax

```
boolean ResultSet.getBoolean(int ordinal) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The boolean representation of the named value.

getBoolean(String) method

Returns a boolean value.

Syntax

```
boolean ResultSet.getBoolean(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The boolean representation of the named value.

getBytes method

Returns a byte array.

Overload list

Name	Description
"getBytes(int) method"	Returns a byte array.
"getBytes(String) method"	Returns a byte array.

getBytes(int) method

Returns a byte array.

Syntax

```
byte[] ResultSet.getBytes(int ordinal) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The byte array representation of the named value.

getBytes(String) method

Returns a byte array.

Syntax

```
byte[] ResultSet.getBytes(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The byte array representation of the named value.

getClobReader method

Returns a Reader.

Overload list

Name	Description
"getClobReader(int) method"	Returns a Reader.

Name	Description
"getClobReader(String) method"	Returns a Reader.

getClobReader(int) method

Returns a Reader.

Syntax

```
java.io.Reader ResultSet.getClobReader(int ordinal) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The Reader representation of the named value.

getClobReader(String) method

Returns a Reader.

Syntax

```
java.io.Reader ResultSet.getClobReader(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The Reader representation of the named value.

getDate method

Returns a java.util.Date.

Overload list

Name	Description
"getDate(int) method"	Returns a java.util.Date.
"getDate(String) method"	Returns a java.util.Date.

getDate(int) method

Returns a java.util.Date.

Syntax

```
java.util.Date ResultSet.getDate(int ordinal) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The java.util.Date representation of the named value.

getDate(String) method

Returns a java.util.Date.

Syntax

```
java.util.Date ResultSet.getDate(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The java.util.date representation of the named value.

getDecimalNumber method

Returns a DecimalNumber.

Overload list

Name	Description
"getDecimalNumber(int) method"	Returns a DecimalNumber.
"getDecimalNumber(String) method"	Returns a DecimalNumber.

getDecimalNumber(int) method

Returns a DecimalNumber.

Syntax

```
DecimalNumber ResultSet.getDecimalNumber(
    int ordinal
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The DecimalNumber representation of the named value.

getDecimalNumber(String) method

Returns a DecimalNumber.

Syntax

```
DecimalNumber ResultSet.getDecimalNumber(
    String name
) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The DecimalNumber representation of the named value.

getDouble method

Returns a double value based on the column number.

Overload list

Name	Description
"getDouble(int) method"	Returns a double value based on the column number.
"getDouble(String) method"	Returns a DecimalNumber.

getDouble(int) method

Returns a double value based on the column number.

Syntax

```
double ResultSet.getDouble(int ordinal) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The double representation of the named value.

getDouble(String) method

Returns a DecimalNumber.

Syntax

```
double ResultSet.getDouble(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The double representation of the named value.

getFloat method

Returns a float value.

Overload list

Name	Description
"getFloat(int) method"	Returns a float value.
"getFloat(String) method"	Returns a float value.

getFloat(int) method

Returns a float value.

Syntax

```
float ResultSet.getFloat(int ordinal) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The float representation of the named value.

getFloat(String) method

Returns a float value.

Syntax

```
float ResultSet.getFloat(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The float representation of the named value.

getInt method

Returns an integer value.

Overload list

Name	Description
"getInt(int) method"	Returns an integer value.
"getInt(String) method"	Returns an integer value.

getInt(int) method

Returns an integer value.

Syntax

```
int ResultSet.getInt(int ordinal) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The integer representation of the named value.

getInt(String) method

Returns an integer value.

Syntax

```
int ResultSet.getInt(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The integer representation of the named value.

getLong method

Returns a long integer value.

Overload list

Name	Description
"getLong(int) method"	Returns a long integer value.
"getLong(String) method"	Returns a long integer value.

getLong(int) method

Returns a long integer value.

Syntax

```
long ResultSet.getLong(int ordinal) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The long integer representation of the named value.

getLong(String) method

Returns a long integer value.

Syntax

```
long ResultSet.getLong(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The long integer representation of the named value.

getOrdinal method

Returns the (base-one) ordinal for the value represented by a String.

Syntax

```
int ResultSet.getOrdinal(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The ordinal value.

getResultSetMetadata method

Returns the ResultSetMetadata containing the meta data for the ResultSet.

Syntax

```
ResultSetMetadata ResultSet.getResultSetMetadata() throws ULjException
```

Returns

The ResultSetMetadata object.

getSize method

Get the actual size of a result set column.

Overload list

Name	Description
"getSize(int) method"	Get the actual size of a result set column.
"getSize(String) method"	Get the actual size of a result set column.

getSize(int) method

Get the actual size of a result set column.

Syntax

```
int ResultSet.getSize(int ordinal) throws ULjException
```

Parameters

- **ordinal**   A base-one integer representing the column number as ordered in the SQL statement.

Returns

the actual size of a result set column

getSize(String) method

Get the actual size of a result set column.

Syntax

```
int ResultSet.getSize(String name) throws ULjException
```

Parameters

- **name**   A String representing the table column name.

Returns

the actual size of a result set column

getString method

Returns a String value.

Overload list

Name	Description
"getString(int) method"	Returns a String value.
"getString(String) method"	Returns a String value.

getString(int) method

Returns a String value.

Syntax

```
String ResultSet.getString(int ordinal) throws ULjException
```

Parameters

- **ordinal**   A base-one integer representing the column number as ordered in the SQL statement.

Returns

The String representation of the named value.

getString(String) method

Returns a String value.

Syntax

```
String ResultSet.getString(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The String representation of the named value.

getUUIDValue method

Returns a UUIDValue value.

Overload list

Name	Description
"getUUIDValue(int) method"	Returns a UUIDValue value.
"getUUIDValue(String) method"	Returns a UUIDValue value.

getUUIDValue(int) method

Returns a UUIDValue value.

Syntax

```
UUIDValue ResultSet.getUUIDValue(int ordinal) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The UUIDValue representation of the named value.

getUUIDValue(String) method

Returns a UUIDValue value.

Syntax

```
UUIDValue ResultSet.getUUIDValue(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The UUIDValue representation of the named value.

isNull method

Tests if the value at the specified column number is null.

Overload list

Name	Description
"isNull(int) method"	Tests if the value at the specified column number is null.
"isNull(String) method"	Tests if the value at the specified column number is null.

isNull(int) method

Tests if the value at the specified column number is null.

Syntax

```
boolean ResultSet.isNull(int ordinal) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

True if the value is null, false otherwise.

isNull(String) method

Tests if the value at the specified column number is null.

Syntax

```
boolean ResultSet.isNull(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

True if the value is null, false otherwise.

next method

Fetches the next row of data in the ResultSet.

Syntax

```
boolean ResultSet.next() throws ULjException
```

Returns

true when the next row is successfully fetched; otherwise, false.

See also

- "ResultSetMetadata interface" on page 213

previous method

Fetches the previous row of data in the ResultSet.

Syntax

```
boolean ResultSet.previous() throws ULjException
```

Returns

true when the previous row is successfully fetched; otherwise, false.

ResultSetMetadata interface

Associated with a ResultSet object and contains a method that provides column information.

Syntax

```
public interface ResultSetMetadata
```

Members

All members of ResultSetMetadata interface, including all inherited members.

Name	Description
"getAliasName method"	Returns alias name for a column.
"getColumnCount method"	Returns the total number of columns in the ResultSet.
"getCorrelationName method"	Returns correlation name for a column.
"getDomainName method"	Returns the name of the domain.

Name	Description
"getDomainPrecision method"	Returns the precision of the domain value.
"getDomainScale method"	Returns the scale of the domain value.
"getDomainSize method"	Returns the size of the domain value.
"getDomainType method"	Returns the type of the domain.
"getQualifiedName method"	Returns the qualified name for a column.
"getTableColumnName method"	Returns column name in the table (or derived table).
"getTableName method"	Returns table name for a column.
"getWrittenName method"	Returns the written name for a column.

Remarks

This interface is obtained using the getResultSetMetadata method of a ResultSet object.

When a column in the select list of the ResultSet is a simple name or a compound name (table-name.column-name, correlation-name.column-name), then, the following information about that name can be extracted if it exists:

- Alias name
- Correlation name
- Qualified version of the name
- Table name
- The name as written

For each column in the select list of the ResultSet, information about the domain of that column can be obtained:

- Column Type: an integer from the Domain interface
- Name of the domain
- Size of the domain, for VARCHAR and BINARY domains
- Scale and precision, for NUMERIC domains

See also

- “Domain interface” on page 151
- “ResultSet interface” on page 198
- “getResultSetMetadata method” on page 209

getAliasName method

Returns alias name for a column.

Syntax

```
String ResultSetMetadata.getAliasName(int column_no) throws ULjException
```

Parameters

- **column_no** the (base 1) number of the column in the select list.

Returns

null, if there is not an alias name for the column; otherwise, alias name for a column.

getColumnCount method

Returns the total number of columns in the ResultSet.

Syntax

```
int ResultSetMetadata.getColumnCount() throws ULjException
```

Returns

The number of columns.

getCorrelationName method

Returns correlation name for a column.

Syntax

```
String ResultSetMetadata.getCorrelationName(
     int column_no
) throws ULjException
```

Parameters

- **column_no** the (base 1) number of the column in the select list.

Returns

null, if there is not an correlation name for the column; otherwise, correlation name for a column.

getDomainName method

Returns the name of the domain.

Syntax

```
String ResultSetMetadata.getDomainName(
    int column_no
) throws ULjException
```

Parameters

- **column_no** the (base 1) number of the column in the select list.

Returns

The domain name.

getDomainPrecision method

Returns the precision of the domain value.

Syntax

```
int ResultSetMetadata.getDomainPrecision(
    int column_no
) throws ULjException
```

Parameters

- **column_no** the (base 1) number of the column in the select list.

Returns

The precision.

getDomainScale method

Returns the scale of the domain value.

Syntax

```
int ResultSetMetadata.getDomainScale(int column_no) throws ULjException
```

Parameters

- **column_no** the (base 1) number of the column in the select list.

Returns

The scale.

getDomainSize method

Returns the size of the domain value.

Syntax

```
int ResultSetMetadata.getDomainSize(int column_no) throws ULjException
```

Parameters

- **column_no** the (base 1) number of the column in the select list.

Returns

The size.

getDomainType method

Returns the type of the domain.

Syntax

```
short ResultSetMetadata.getDomainType(int column_no) throws ULjException
```

Parameters

- **column_no** the (base 1) number of the column in the select list.

Returns

The domain type expressed as an integer.

getQualifiedName method

Returns the qualified name for a column.

Syntax

```
String ResultSetMetadata.getQualifiedName(
    int column_no
) throws ULjException
```

Parameters

- **column_no** the (base 1) number of the column in the select list.

Returns

null, if there is not a qualified name for the column; otherwise, qualified name for a column.

Remarks

When the ResultSet column refers to a column in a table, the name returned is a compound name consisting of a correlation name (or table name if the correlation name was not given) followed by the name of the column in the table.

getTableColumnName method

Returns column name in the table (or derived table).

Syntax

```
String ResultSetMetadata.getTableColumnName(
    int column_no
) throws ULjException
```

Parameters

- **column_no** the (base 1) number of the column in the select list.

Returns

null, if there is not a table name for the column; otherwise, table name for a column.

getTableName method

Returns table name for a column.

Syntax

```
String ResultSetMetadata.getTableName(int column_no) throws ULjException
```

Parameters

- **column_no** the (base 1) number of the column in the select list.

Returns

null, if there is not a table name for the column; otherwise, table name for a column.

getWrittenName method

Returns the written name for a column.

Syntax

```
String ResultSetMetadata.getWrittenName(
    int column_no
) throws ULjException
```

Parameters

- **column_no** the (base 1) number of the column in the select list.

Returns

null, if there is not a written name for the column; otherwise, written name for a column.

SISListener interface (Java ME BlackBerry only)

Listens for server-initiated synchronization messages.

Syntax

```
public interface SISListener
```

Members

All members of SISListener interface, including all inherited members.

Name	Description
"startListening method"	Creates and starts the listening thread.
"stopListening method"	Stops the listening thread.

Remarks

The application creates an instance of SISListener using the appropriate createSISHTTPListener method in the DatabaseManager interface.

See also

- "createSISHTTPListener method (Java ME BlackBerry only)" on page 147

startListening method

Creates and starts the listening thread.

Syntax

```
void SISListener.startListening()
```

stopListening method

Stops the listening thread.

Syntax

```
void SISListener.stopListening()
```

SISRequestHandler interface (Java ME BlackBerry only)

Handles server-initiated synchronization requests.

Syntax

```
public interface SISRequestHandler
```

Members

All members of SISRequestHandler interface, including all inherited members.

Name	Description
"onError method"	Handles SIS related errors that occur during SIS listening.
"onRequest method"	Handles SIS requests on worker threads.

See also

- "SISListener interface (Java ME BlackBerry only)" on page 219

onError method

Handles SIS related errors that occur during SIS listening.

Syntax

```
void SISRequestHandler.onError(String text)
```

Parameters

- **text**   The string representation of the exception.

Remarks

To stop listening, explicitly call the stopListening method in the SISListener interface.

onRequest method

Handles SIS requests on worker threads.

Syntax

```
void SISRequestHandler.onRequest(String text)
```

Parameters

- **text** The string sent by the request.

StreamHTTPParms interface

Represents HTTP stream parameters that define how to communicate with a MobiLink server using HTTP.

Syntax

```
public interface StreamHTTPParms
```

Derived classes

- "StreamHTTPSParms interface" on page 225

Members

All members of StreamHTTPParms interface, including all inherited members.

Name	Description
"getHost method"	Returns the host name of the MobiLink server.
"getOutputBufferSize method"	Returns the size, in bytes, of the output buffer used to store data before it is sent to the MobiLink server.
"getPort method"	Returns the port number used to connect to the MobiLink server.
"getURLSuffix method"	Returns the URL suffix of the MobiLink server.
"setHost method"	Sets the host name of the MobiLink server.
"setOutputBufferSize method"	Sets the size, in bytes, of the output buffer used to store data before it is sent to the MobiLink server.
"setPort method"	Sets the port number used to connect to the MobiLink server.
"setURLSuffix method"	Sets the URL suffix of the MobiLink server.

Remarks

The following example sets the stream parameters to communicate with a MobiLink server on host name "MyMLHost". The server started with the following parameters: "-xo http(port=1234)":

```
SyncParms syncParms = myConnection.createSyncParms(
        SyncParms.HTTP_STREAM,
        "MyUniqueMLUserID",
            "MyMLScriptVersion"
        );
    StreamHTTPParms httpParms = syncParms.getStreamParms();
    httpParms.setHost("MyMLHost");
    httpParms.setPort(1234);
```

Instances implementing this interface are returned by the SyncParms.getStreamParms method.

See also

- "SyncParms class" on page 235
- "getStreamParms method" on page 240

getHost method

Returns the host name of the MobiLink server.

Syntax

String **StreamHTTPParms.getHost**()

Returns

The name of the host.

See also

- "setHost method" on page 223
- "getPort method" on page 223
- "setPort method" on page 224

getOutputBufferSize method

Returns the size, in bytes, of the output buffer used to store data before it is sent to the MobiLink server.

Syntax

int **StreamHTTPParms.getOutputBufferSize**()

Returns

The integer containing the buffer size.

Remarks

Increasing this value may reduce the number of network flushes needed to send a large upload at the cost of increased memory use. In HTTP, each flush sends a large (approximately 250 bytes) HTTP header; reducing the number of flushes can reduce the bandwidth use.

See also

- "setOutputBufferSize method" on page 224

getPort method

Returns the port number used to connect to the MobiLink server.

Syntax

```
int StreamHTTPParms.getPort()
```

Returns

The port number of MobiLink server.

See also

- “setPort method” on page 224

getURLSuffix method

Returns the URL suffix of the MobiLink server.

Syntax

```
String StreamHTTPParms.getURLSuffix()
```

Returns

The String containing the URL suffix.

See also

- “setURLSuffix method” on page 224

setHost method

Sets the host name of the MobiLink server.

Syntax

```
void StreamHTTPParms.setHost(String v)
```

Parameters

- **v** The name of the host.

Remarks

The default is null, which indicates a localhost.

See also

- “getHost method” on page 222
- “getPort method” on page 223
- “setPort method” on page 224

setOutputBufferSize method

Sets the size, in bytes, of the output buffer used to store data before it is sent to the MobiLink server.

Syntax

```
void StreamHTTPParms.setOutputBufferSize(int size)
```

Parameters

- **size** The new buffer size.

Remarks

The default is 512 on non-Blackberry Java ME; otherwise, 4096. Valid values range between 512 and 32768. Increasing this value may cause the Java runtime to send chunked HTTP, which the MobiLink server cannot process. If the MobiLink server outputs an "unknown transfer encoding" error, try decreasing this value.

See also

- “getOutputBufferSize method” on page 222

setPort method

Sets the port number used to connect to the MobiLink server.

Syntax

```
void StreamHTTPParms.setPort(int v)
```

Parameters

- **v** A port number ranging from 1 to 65535. Out of range values revert to default value.

Remarks

The default port is 80 for HTTP synchronizations and 443 for HTTPS synchronizations.

See also

- “getPort method” on page 223

setURLSuffix method

Sets the URL suffix of the MobiLink server.

Syntax

```
void StreamHTTPParms.setURLSuffix(String v)
```

Parameters

- **v**   The URL suffix string.

Remarks

The default setting is null, which implies a URL suffix of "Mobilink/".

You can include information in the URL suffix. The following example demonstrates how a client synchronizing through a hosted relay server can use this method:

```
setURLSuffix("/ias_relay_server/client/rs_client.dll/username.FarmName");
```

The MobiLink client network protocol options for HTTP authentication may be added as a separated list, as demonstrated in the following example:

```
setURLSuffix("http_userid=userid;http_password=pwd");
```

See also

- "getURLSuffix method" on page 223

StreamHTTPSParms interface

Represents HTTPS stream parameters that define how to communicate with a MobiLink server using secure HTTPS.

Syntax

```
public interface StreamHTTPSParms
```

Base classes

- "StreamHTTPParms interface" on page 221

Members

All members of StreamHTTPSParms interface, including all inherited members.

Name	Description
"getCertificateCompany method"	Returns the certificate company name for verification of secure connections.

Name	Description
"getCertificateName method"	Returns the certificate common name for verification of secure connections.
"getCertificateUnit method"	Returns the certificate unit name for verification of secure connections.
"getHost method"	Returns the host name of the MobiLink server.
"getOutputBufferSize method"	Returns the size, in bytes, of the output buffer used to store data before it is sent to the MobiLink server.
"getPort method"	Returns the port number used to connect to the MobiLink server.
"getTrustedCertificates method"	Returns the name of the file containing a list of trusted root certificates used for secure synchronization.
"getURLSuffix method"	Returns the URL suffix of the MobiLink server.
"setCertificateCompany method"	Sets the certificate company name for verification of secure connections.
"setCertificateName method"	Sets the certificate common name for verification of secure connections.
"setCertificateUnit method"	Sets the certificate unit name for verification of secure connections.
"setHost method"	Sets the host name of the MobiLink server.
"setOutputBufferSize method"	Sets the size, in bytes, of the output buffer used to store data before it is sent to the MobiLink server.
"setPort method"	Sets the port number used to connect to the MobiLink server.
"setTrustedCertificates method"	Sets a file containing a list of trusted root certificates used for secure synchronization.
"setURLSuffix method"	Sets the URL suffix of the MobiLink server.

Remarks

The following example sets the stream parameters to communicate with a MobiLink server on host name "MyMLHost". The server started with the following parameters: "-xo https(port=1234;certificate=RSAServer.crt;certificate_password=x)"

```
SyncParms syncParms = myConnection.createSyncParms(
        SyncParms.HTTPS_STREAM,
        "MyUniqueMLUserID",
        "MyMLScriptVersion"
    );
```

```
StreamHTTPSParms httpsParms =
    (StreamHTTPSParms) syncParms.getStreamParms();
httpsParms.setHost("MyMLHost");
httpsParms.setPort(1234);
```

The above example assumes that the certificate in RSAServer.crt is chained to a trusted root certificate already installed on the client host or device.

For Java SE, you can deploy the required trusted root certificate by using one of the following methods:

1. Install the trusted root certificate in the lib/security/cacerts key store of the JRE.

2. Build your own key store using the Java keytool utility and setting the Java system property javax.net.ssl.trustStore to its location (set javax.net.ssl.trustStorePassword to an appropriate value).

3. Using the setTrustedCertificates(String) parameter to point to the deployed certificate file.

To enhance security, the setCertificateName, setCertificateCompany setCertificateUnit methods should be used to turn on validation of the MobiLink server certificate.

Instances implementing this interface are returned by the SyncParms.getStreamParms method when the SyncParms object is created for HTTPS synchronization.

See also

- "SyncParms class" on page 235
- "getStreamParms method" on page 240
- "setCertificateCompany method" on page 228
- "setCertificateName method" on page 229
- "setCertificateUnit method" on page 229
- "setTrustedCertificates method" on page 229

getCertificateCompany method

Returns the certificate company name for verification of secure connections.

Syntax

```
String StreamHTTPSParms.getCertificateCompany()
```

Returns

The certificate company name.

getCertificateName method

Returns the certificate common name for verification of secure connections.

Syntax

```
String StreamHTTPSParms.getCertificateName()
```

Returns

The certificate name.

getCertificateUnit method

Returns the certificate unit name for verification of secure connections.

Syntax

```
String StreamHTTPSParms.getCertificateUnit()
```

Returns

The organization unit name.

getTrustedCertificates method

Returns the name of the file containing a list of trusted root certificates used for secure synchronization.

Syntax

```
String StreamHTTPSParms.getTrustedCertificates()
```

Returns

The filename of the trusted root certificates file.

setCertificateCompany method

Sets the certificate company name for verification of secure connections.

Syntax

```
void StreamHTTPSParms.setCertificateCompany(String val)
```

Parameters

- **val** The company name.

Remarks

The default is null, indicating that the company name does not get verified in the certificate.

setCertificateName method

Sets the certificate common name for verification of secure connections.

Syntax

```
void StreamHTTPSParms.setCertificateName(String val)
```

Parameters

- **val** The certificate common name.

Remarks

The default is null, indicating that the common name does not get verified in the certificate.

setCertificateUnit method

Sets the certificate unit name for verification of secure connections.

Syntax

```
void StreamHTTPSParms.setCertificateUnit(String val)
```

Parameters

- **val** The company unit name.

Remarks

The default is null, indicating that the organization unit name does not get verified in the certiciate.

setTrustedCertificates method

Sets a file containing a list of trusted root certificates used for secure synchronization.

Syntax

```
void StreamHTTPSParms.setTrustedCertificates(
    String filename
) throws ULjException
```

Parameters

- **filename** The filename of the trusted root certificate.

Remarks

This parameter is only used on Java SE systems.

The default is null, indicating that the system default certificate store is used to validate certificate chains from the MobiLink server.

SyncObserver interface

Receives synchronization progress information.

Syntax

```
public interface SyncObserver
```

Members

All members of SyncObserver interface, including all inherited members.

Name	Description
"syncProgress method"	Informs the user of progress and is invoked during synchronization.

Remarks

To receive progress reports during synchronization, you must create a new class that performs the task, and implement it using the SyncParms.setSyncObserver method.

The following example illustrates a simple SyncObserver:

```
class MyObserver implements SyncObserver {
  public boolean syncProgress(int state, SyncResult result) {
    System.out.println(
        "sync progress state = " + state
        + " bytes sent = " + result.getSentByteCount()
        + " bytes received = " + result.getReceivedByteCount()
      );
    return false;   // Always continue synchronization.
  }
  public MyObserver() {} // The default constructor.
}
```

The observer class above is enabled as follows:

```
SyncParms.setSyncObserver(new MyObserver());
```

See also

- "SyncParms class" on page 235
- "setSyncObserver method" on page 247

syncProgress method

Informs the user of progress and is invoked during synchronization.

Syntax

```
boolean SyncObserver.syncProgress(int state, SyncResult data)
```

Parameters

- **state** One of the SyncOberver.States constants, representing the current state of the synchronization.
- **data** A SyncResult containing the latest synchronization results.

Returns

return true to cancel the synchronization; otherwise, false to continue csynchronization.

Remarks

The various states, which are signaled as packets, are received and sent. Since multiple tables may be uploaded or downloaded in a single packet, syncProgress calls for any given synchronization may skip a number of states.

Note
With the exception of the SyncResult methods, no other UltraLiteJ API methods should be invoked during a syncProgress call.

See also

- "SyncObserver.States interface" on page 231
- "setSyncObserver method" on page 247

SyncObserver.States interface

Defines the synchronization states that can be signaled to an observer.

Syntax

```
public interface SyncObserver.States
```

Members

All members of SyncObserver.States interface, including all inherited members.

Name	Description
"CHECKING_LAST_UPLOAD variable"	Checking the status of the previous upload.
"COMMITTING_DOWNLOAD variable"	The downloaded rows are being committed to the data-base.
"DISCONNECTING variable"	The synchronization stream is disconnecting.
"DONE variable"	Synchronization is complete.

Name	Description
"ERROR variable"	Synchronization is complete but an error occurred.
"FINISHING_UPLOAD variable"	The upload is finalizing.
"RECEIVING_TABLE variable"	A new table is being downloaded.
"RECEIVING_UPLOAD_ACK variable"	An upload acknowledged is being downloaded.
"ROLLING_BACK_DOWNLOAD variable"	Synchronization is rolling back the download because an error was encountered during the download.
"SENDING_DOWNLOAD_ACK variable"	An acknowledgement of a complete download is being sent.
"SENDING_HEADER variable"	The synchronization stream has been opened and the header is about to be sent.
"SENDING_SCHEMA variable"	The schema is being sent.
"SENDING_TABLE variable"	A new table is being uploaded.
"STARTING variable"	A synchronization is starting.

See also

- "setSyncObserver method" on page 247
- "SyncObserver interface" on page 230

CHECKING_LAST_UPLOAD variable

Checking the status of the previous upload.

Syntax

```
final int SyncObserver.States.CHECKING_LAST_UPLOAD
```

COMMITTING_DOWNLOAD variable

The downloaded rows are being committed to the database.

Syntax

```
final int SyncObserver.States.COMMITTING_DOWNLOAD
```

DISCONNECTING variable

The synchronization stream is disconnecting.

Syntax

```
final int SyncObserver.States.DISCONNECTING
```

DONE variable

Synchronization is complete.

Syntax

```
final int SyncObserver.States.DONE
```

Remarks

No other states are reported.

ERROR variable

Synchronization is complete but an error occurred.

Syntax

```
final int SyncObserver.States.ERROR
```

FINISHING_UPLOAD variable

The upload is finalizing.

Syntax

```
final int SyncObserver.States.FINISHING_UPLOAD
```

RECEIVING_TABLE variable

A new table is being downloaded.

Syntax

```
final int SyncObserver.States.RECEIVING_TABLE
```

RECEIVING_UPLOAD_ACK variable

An upload acknowledged is being downloaded.

Syntax

```
final int SyncObserver.States.RECEIVING_UPLOAD_ACK
```

ROLLING_BACK_DOWNLOAD variable

Synchronization is rolling back the download because an error was encountered during the download.

Syntax

```
final int SyncObserver.States.ROLLING_BACK_DOWNLOAD
```

SENDING_DOWNLOAD_ACK variable

An acknowledgement of a complete download is being sent.

Syntax

```
final int SyncObserver.States.SENDING_DOWNLOAD_ACK
```

SENDING_HEADER variable

The synchronization stream has been opened and the header is about to be sent.

Syntax

```
final int SyncObserver.States.SENDING_HEADER
```

SENDING_SCHEMA variable

The schema is being sent.

Syntax

```
final int SyncObserver.States.SENDING_SCHEMA
```

SENDING_TABLE variable

A new table is being uploaded.

Syntax

```
final int SyncObserver.States.SENDING_TABLE
```

STARTING variable

A synchronization is starting.

Syntax

```
final int SyncObserver.States.STARTING
```

Remarks

No actions have taken place yet.

SyncParms class

Maintains the parameters used during the database synchronization process.

Syntax

```
public class SyncParms
```

Members

All members of SyncParms class, including all inherited members.

Name	Description
“SyncParms constructor”	To create a SyncParms object, use Connection.createSyncParms.
“getAcknowledgeDownload method”	Determines if the remote sends download acknowledgements.
“getAuthenticationParms method”	Returns parameters provided to a custom user authentication script.
“getLivenessTimeout method”	Returns the liveness timeout length, in seconds.
“getNewPassword method”	Returns the new MobiLink password for the user specified with setUserName.
“getPassword method”	Returns the MobiLink password for the user specified with se-tUserName.
“getPublications method”	Returns the publications to be synchronized.
“getSendColumnNames method”	Returns true if column names are sent to the MobiLink server.

Name	Description
“getStreamParms method”	Returns the parameters used to configure the synchronization stream.
“getSyncObserver method”	Returns the current SyncObserver.
“getSyncResult method”	Returns the SyncResult object that contains the status of the synchronization.
“getTableOrder method”	Returns the order in which tables should be uploaded to the consolidated database.
“getUserName method”	Returns the MobiLink user name that uniquely identifies the client to the MobiLink server.
“getVersion method”	Returns the synchronization script to use.
“isDownloadOnly method”	Determines if the synchronization is download-only.
“isPingOnly method”	Determines if the synchronization pings the MobiLink server instead of performing a synchronization.
“isUploadOnly method”	Determines if the synchronization is upload-only.
“setAcknowledgeDownload method”	Indicates if the remote should send a download acknowledgement.
“setAuthenticationParms method”	Specifies parameters for a custom user authentication script (MobiLink authenticate_parameters connection event).
“setDownloadOnly method”	Sets the synchronization as download-only.
“setLivenessTimeout method”	Sets the liveness timeout length, in seconds.
“setNewPassword method”	Sets a new MobiLink password for the user specified with setUserName.
“setPassword method”	Sets the MobiLink password for the user specified with setUserName.
“setPingOnly method”	Sets the synchronization to ping the MobiLink server instead of performing a synchronization.
“setPublications method”	Sets the publications to be synchronized.
“setSendColumnNames method”	Sets whether column names are sent to the MobiLink server during a sync.

Name	Description
“setSyncObserver method”	Sets a SyncObserver object to monitor the progress of the synchronization.
“setTableOrder method”	Sets the order in which tables should be uploaded to the consolidated database.
“setUploadOnly method”	Sets the synchronization as upload-only.
“setUserName method”	Sets the MobiLink user name that uniquely identifies the client to the MobiLink server.
“setVersion method”	Sets the synchronization script to use.
“HTTP_STREAM variable”	Creates a SyncParms object for HTTP synchronizations.
“HTTPS_STREAM variable”	Creates a SyncParms object for secure HTTPS synchronizations.

Remarks

This interface is invoked using the createSyncParms method of a Connection object.

You can only set one synchronization command at a time. These commands are specified using the setDownloadOnly, setPingOnly, and setUploadOnly methods. By setting one of these methods to true, you set the other methods to false.

The UserName and Version parameters must be set. The UserName must be unique for each client database.

The communication stream is configured using the getStreamParms method based on the type of SyncParms object. For example, the following code prepares and performs an HTTP synchronization:

```
SyncParms syncParms = myConnection.createSyncParms(
        SyncParms.HTTP_STREAM,
        "MyUniqueMLUserID",
        "MyMLScriptVersion"
    );
syncParms.setPassword("ThePWDforMyUniqueMLUserID");
syncParms.getStreamParms().setHost("MyMLHost");
myConnection.synchronize(syncParms);
```

Comma Separated Lists AuthenticationParms, Publications, and TableOrder parameters are all specified using a string value that contains a comma separated list of values. Values within the list may be quoted using either single quotes or double quotes, but there are no escape characters. Leading and trailling spaces in values are ignored unless quoted. For example:

```
syncParms.setTableOrder("'Table A',\"Table B,D\",Table C" );
 specifies "Table A" then "Table B,D" and then "Table C".
```

See also

- "getStreamParms method" on page 240
- "setUserName method" on page 249
- "createSyncParms method" on page 123
- "StreamHTTPParms interface" on page 221
- "StreamHTTPSParms interface" on page 225

SyncParms constructor

To create a SyncParms object, use Connection.createSyncParms.

Syntax

```
SyncParms.SyncParms( )
```

See also

- "createSyncParms method" on page 123

getAcknowledgeDownload method

Determines if the remote sends download acknowledgements.

Syntax

```
abstract boolean SyncParms.getAcknowledgeDownload( )
```

Returns

true, if the remote sends download acknowledgements; otherwise, false.

See also

- "setAcknowledgeDownload method" on page 243

getAuthenticationParms method

Returns parameters provided to a custom user authentication script.

Syntax

```
abstract String SyncParms.getAuthenticationParms( )
```

Returns

The list of authentication parms or null if no parameters are specified.

See also

- “setAuthenticationParms method” on page 244

getLivenessTimeout method

Returns the liveness timeout length, in seconds.

Syntax

```
abstract int SyncParms.getLivenessTimeout()
```

Returns

The timeout.

See also

- “setLivenessTimeout method” on page 245

getNewPassword method

Returns the new MobiLink password for the user specified with setUserName.

Syntax

```
abstract String SyncParms.getNewPassword()
```

Returns

The new password set after the next synchronization.

See also

- “setUserName method” on page 249
- “setNewPassword method” on page 245

getPassword method

Returns the MobiLink password for the user specified with setUserName.

Syntax

```
abstract String SyncParms.getPassword()
```

Returns

The password for the MobiLink user.

See also

- “setPassword method” on page 246

getPublications method

Returns the publications to be synchronized.

Syntax

```
abstract String SyncParms.getPublications()
```

Returns

The set of publications to synchronize.

See also

- “setPublications method” on page 246

getSendColumnNames method

Returns true if column names are sent to the MobiLink server.

Syntax

```
abstract boolean SyncParms.getSendColumnNames()
```

Returns

true if column names are sent.

See also

- “setSendColumnNames method” on page 247

getStreamParms method

Returns the parameters used to configure the synchronization stream.

Syntax

```
abstract StreamHTTPParms SyncParms.getStreamParms()
```

Returns

A StreamHTTPParms or StreamHTTPSParms object specifying the parameters for HTTP or HTTPS synchronization streams. The object is returned by reference.

Remarks

The synchronization stream type is specified when the SyncParms object is created.

See also

- "createSyncParms method" on page 123
- "StreamHTTPParms interface" on page 221
- "StreamHTTPSParms interface" on page 225

getSyncObserver method

Returns the current SyncObserver.

Syntax

```
abstract SyncObserver SyncParms.getSyncObserver()
```

Returns

The SyncObserver, or null if no observer exists.

getSyncResult method

Returns the SyncResult object that contains the status of the synchronization.

Syntax

```
abstract SyncResult SyncParms.getSyncResult()
```

Returns

The SyncResult object.

See also

- "SyncResult class" on page 250

getTableOrder method

Returns the order in which tables should be uploaded to the consolidated database.

Syntax

```
abstract String SyncParms.getTableOrder()
```

Returns

A comma separated list of table names; otherwise, null if no table order is specified. See class description for more on comma separated lists.

See also

- “setTableOrder method” on page 248

getUserName method

Returns the MobiLink user name that uniquely identifies the client to the MobiLink server.

Syntax

```
abstract String SyncParms.getUserName()
```

Returns

The MobiLink user name.

See also

- “setUserName method” on page 249

getVersion method

Returns the synchronization script to use.

Syntax

```
abstract String SyncParms.getVersion()
```

Returns

The script version.

See also

- “setVersion method” on page 249

isDownloadOnly method

Determines if the synchronization is download-only.

Syntax

```
abstract boolean SyncParms.isDownloadOnly()
```

Returns

true, if uploads are disabled; otherwise, false.

See also

- “setDownloadOnly method” on page 244

isPingOnly method

Determines if the synchronization pings the MobiLink server instead of performing a synchronization.

Syntax

```
abstract boolean SyncParms.isPingOnly()
```

Returns

true if the client only pings the server; otherwise, false.

See also

- “setPingOnly method” on page 246

isUploadOnly method

Determines if the synchronization is upload-only.

Syntax

```
abstract boolean SyncParms.isUploadOnly()
```

Returns

true if downloads are disabled; otherwise, false.

See also

- “setUploadOnly method” on page 248

setAcknowledgeDownload method

Indicates if the remote should send a download acknowledgement.

Syntax

```
abstract void SyncParms.setAcknowledgeDownload(boolean ack)
```

Parameters

- **ack** Set true to have the client acknowledge a download; otherwise, set false.

Remarks

The default is false.

See also

- “getAcknowledgeDownload method” on page 238

setAuthenticationParms method

Specifies parameters for a custom user authentication script (MobiLink authenticate_parameters connection event).

Syntax

```
abstract void SyncParms.setAuthenticationParms(
    String v
) throws ULjException
```

Parameters

- **v** A comma separated list of authentication parameters, or the null reference. See class description for more on comma separated lists.

Remarks

Only the first 255 strings are used and each string should be no longer than 128 characters (longer strings are truncated when sent to MobiLink).

See also

- “getAuthenticationParms method” on page 238

setDownloadOnly method

Sets the synchronization as download-only.

Syntax

```
abstract void SyncParms.setDownloadOnly(boolean v)
```

Parameters

- **v** Set true to disable uploads, or set false to enable uploads.

Remarks

The default is false. Specifying true changes setPingOnly and setUploadOnly to false.

See also

- “isDownloadOnly method” on page 242
- “setPingOnly method” on page 246
- “setUploadOnly method” on page 248

setLivenessTimeout method

Sets the liveness timeout length, in seconds.

Syntax

```
abstract void SyncParms.setLivenessTimeout(int l) throws ULjException
```

Parameters

- **l** The new liveness timeout value.

Remarks

The liveness timeout is the length of time the server allows a remote to be idle. If the remote does not communicate with the server for l seconds, the server assumes that the remote has lost the connection, and terminates the sync. The remote automatically sends periodic messages to the server to keep the connection alive.

If a negative value is set, an exception is thrown. The value may be changed by the MobiLink server without notice. This change occurs if the value is set too low or too high.

The default value is 100 seconds.

See also

- "getLivenessTimeout method" on page 239

setNewPassword method

Sets a new MobiLink password for the user specified with setUserName.

Syntax

```
abstract void SyncParms.setNewPassword(String v)
```

Parameters

- **v** A new password for MobiLink user.

Remarks

The new password takes effect after the next synchronization.

The default is null, suggesting that the password does not get replaced.

See also

- "getNewPassword method" on page 239
- "setPassword method" on page 246
- "setUserName method" on page 249

setPassword method

Sets the MobiLink password for the user specified with setUserName.

Syntax

```
abstract void SyncParms.setPassword(String v) throws ULjException
```

Parameters

- **v** A password for the MobiLink user.

Remarks

This user name and password is separate from any database user ID and password. This method is used to authenticate the application against the MobiLink server.

The default is an empty string, suggesting no password.

See also

- "getPassword method" on page 239
- "setNewPassword method" on page 245
- "setUserName method" on page 249

setPingOnly method

Sets the synchronization to ping the MobiLink server instead of performing a synchronization.

Syntax

```
abstract void SyncParms.setPingOnly(boolean v)
```

Parameters

- **v** Set true to only ping the server, or set false to perform a synchronization.

Remarks

The default is false. Specifying true changes setDownloadOnly and setUploadOnly to false.

See also

- "isPingOnly method" on page 243
- "setDownloadOnly method" on page 244
- "setUploadOnly method" on page 248

setPublications method

Sets the publications to be synchronized.

Syntax

```
abstract void SyncParms.setPublications(String pubs) throws ULjException
```

Parameters

- **pubs** A comma separated list of publication names. See class description for more on comma separated lists.

Remarks

The default is Connection.SYNC_ALL, which is used to denote synchronize all tables. To synchronize all publications, use Connection.SYNC_ALL_PUBS.

See also

- "getPublications method" on page 240
- "SYNC_ALL variable" on page 137
- "SYNC_ALL_PUBS variable" on page 137

setSendColumnNames method

Sets whether column names are sent to the MobiLink server during a sync.

Syntax

```
abstract void SyncParms.setSendColumnNames(boolean c)
```

Parameters

- **c** true if column names should be sent

Remarks

Column names are used by the server only when using the direct row API.

The default value is false.

See also

- "getSendColumnNames method" on page 240

setSyncObserver method

Sets a SyncObserver object to monitor the progress of the synchronization.

Syntax

```
abstract void SyncParms.setSyncObserver(SyncObserver so)
```

Parameters

- **so** A SyncObserser.

Remarks

The default is null, suggesting no observer.

See also

- "SyncObserver interface" on page 230

setTableOrder method

Sets the order in which tables should be uploaded to the consolidated database.

Syntax

```
abstract void SyncParms.setTableOrder(String v) throws ULjException
```

Parameters

- **v** A comma separated list of table names in the order they should be synchronized, or null, indicating no table order. See class description for more on comma separated lists.

Remarks

The primary table should be listed first, along with all tables containing foreign key relationships in the consolidated database.

All tables selected for synchronization by the Publications get synchronized whether they are specified in the TableOrder parameter or not. Unspecified tables are synchronized by order of the foreign key relations in the client database. They are sychronized after the specified tables.

The default is a null reference, which does not override the default ordering of tables.

See also

- "getTableOrder method" on page 241
- "setPublications method" on page 246

setUploadOnly method

Sets the synchronization as upload-only.

Syntax

```
abstract void SyncParms.setUploadOnly(boolean v)
```

Parameters

- **v** Set true to disable downloads, or set false to enable downloads.

Remarks

The default is false. Specifying true changes setDownloadOnly and setPingOnly to false.

See also

- "isUploadOnly method" on page 243
- "setDownloadOnly method" on page 244
- "setPingOnly method" on page 246

setUserName method

Sets the MobiLink user name that uniquely identifies the client to the MobiLink server.

Syntax

```
abstract void SyncParms.setUserName(String v) throws ULjException
```

Parameters

- **v** The MobiLink user name.

Remarks

This value is used to determine the download content, to record the synchronization state, and to recover from interruptions during synchronization.

This user name and password is separate from any database user ID and password. This method is used to authenticate the application against the MobiLink server.

This parameter is initalized when the SyncParms object is created.

See also

- "getUserName method" on page 242
- "setPassword method" on page 246
- "setNewPassword method" on page 245
- "createSyncParms method" on page 123

setVersion method

Sets the synchronization script to use.

Syntax

```
abstract void SyncParms.setVersion(String v) throws ULjException
```

Parameters

- **v** The script version.

Remarks

Each synchronization script in the consolidated database is marked with a version string. For example, there can be two different download_cursor scripts, and each one is identified by different version strings. The version string allows an application to choose from a set of synchronization scripts.

This parameter is initalized when the SyncParms object is created.

See also

- "getVersion method" on page 242
- "createSyncParms method" on page 123

HTTP_STREAM variable

Creates a SyncParms object for HTTP synchronizations.

Syntax

```
final int SyncParms.HTTP_STREAM
```

See also

- "createSyncParms method" on page 123

HTTPS_STREAM variable

Creates a SyncParms object for secure HTTPS synchronizations.

Syntax

```
final int SyncParms.HTTPS_STREAM
```

See also

- "createSyncParms method" on page 123

SyncResult class

Reports status-related information on a specified database synchronization.

Syntax

```
public class SyncResult
```

Members

All members of SyncResult class, including all inherited members.

Name	Description
"getAuthStatus method"	Returns the authorization status code of the last synchronization attempt.
"getAuthValue method"	Returns the value specified in custom user authentication synchronization scripts.
"getCurrentTableName method"	Returns the name of the table currently being synced.
"getIgnoredRows method"	Determines if any uploaded rows were ignored during the last synchronization.
"getReceivedByteCount method"	Returns the number of bytes received during data synchronization.
"getReceivedRowCount method"	Returns the number of rows received.
"getSentByteCount method"	Returns the number of bytes sent during data synchronization.
"getSentRowCount method"	Returns the number of rows sent.
"getStreamErrorCode method"	Returns the error code reported by the stream itself.
"getStreamErrorMessage method"	Returns the error message reported by the stream itself.
"getSyncedTableCount method"	Returns the number of tables synced so far.
"getTotalTableCount method"	Returns the number of tables to be synced.
"isUploadOK method"	Determines if the last upload synchronization was successful.

See also

- "getSyncResult method" on page 241

getAuthStatus method

Returns the authorization status code of the last synchronization attempt.

Syntax

```
abstract int SyncResult.getAuthStatus()
```

Returns

An AuthStatusCode value.

getAuthValue method

Returns the value specified in custom user authentication synchronization scripts.

Syntax

```
abstract int SyncResult.getAuthValue()
```

Returns

An integer returned from custom user authentication synchronization scripts.

getCurrentTableName method

Returns the name of the table currently being synced.

Syntax

```
abstract String SyncResult.getCurrentTableName()
```

Returns

The table name.

getIgnoredRows method

Determines if any uploaded rows were ignored during the last synchronization.

Syntax

```
abstract boolean SyncResult.getIgnoredRows()
```

Returns

true if any uploaded rows were ignored during the last synchronization; otherwise, false if no rows were ignored.

getReceivedByteCount method

Returns the number of bytes received during data synchronization.

Syntax

```
abstract long SyncResult.getReceivedByteCount()
```

Returns

The number of bytes.

getReceivedRowCount method

Returns the number of rows received.

Syntax

```
abstract int SyncResult.getReceivedRowCount()
```

Returns

The number of rows.

getSentByteCount method

Returns the number of bytes sent during data synchronization.

Syntax

```
abstract long SyncResult.getSentByteCount()
```

Returns

The number of bytes sent.

getSentRowCount method

Returns the number of rows sent.

Syntax

```
abstract int SyncResult.getSentRowCount()
```

Returns

The number of rows.

getStreamErrorCode method

Returns the error code reported by the stream itself.

Syntax

```
abstract int SyncResult.getStreamErrorCode()
```

Returns

Zero, if there was no communication stream error; otherwise, the response code from the server.

Remarks

This is the HTTP response code.

getStreamErrorMessage method

Returns the error message reported by the stream itself.

Syntax

```
abstract String SyncResult.getStreamErrorMessage()
```

Returns

Null, if no message is available; otherwise, the response message.

Remarks

This is the HTTP response message.

getSyncedTableCount method

Returns the number of tables synced so far.

Syntax

```
abstract int SyncResult.getSyncedTableCount()
```

Returns

The number of tables.

getTotalTableCount method

Returns the number of tables to be synced.

Syntax

```
abstract int SyncResult.getTotalTableCount()
```

Returns

The number of tables.

isUploadOK method

Determines if the last upload synchronization was successful.

Syntax

```
abstract boolean SyncResult.isUploadOK()
```

Returns

true, if the last upload synchronization was successful; otherwise, false.

SyncResult.AuthStatusCode interface

Enumerates the authorization codes returned by the MobiLink server.

Syntax

```
public interface SyncResult.AuthStatusCode
```

Members

All members of SyncResult.AuthStatusCode interface, including all inherited members.

Name	Description
"EXPIRED variable"	User ID or password has expired.
"IN_USE variable"	User ID is already in use.
"INVALID variable"	Bad user ID or password.
"UNKNOWN variable"	Authorization status is unknown.
"VALID variable"	User ID and password were valid at time of synchronization.
"VALID_BUT_EXPIRES_SOON variable"	User ID and password were valid at time of synchronization but expire soon.

See also

- "getAuthStatus method" on page 251

EXPIRED variable

User ID or password has expired.

Syntax

```
final int SyncResult.AuthStatusCode.EXPIRED
```

Remarks

Authorization fails.

IN_USE variable

User ID is already in use.

Syntax

```
final int SyncResult.AuthStatusCode.IN_USE
```

Remarks

Authorization fails.

INVALID variable

Bad user ID or password.

Syntax

```
final int SyncResult.AuthStatusCode.INVALID
```

Remarks

Authorization fails.

UNKNOWN variable

Authorization status is unknown.

Syntax

```
final int SyncResult.AuthStatusCode.UNKNOWN
```

Remarks

This code suggests that a synchronization has not been performed.

VALID variable

User ID and password were valid at time of synchronization.

Syntax

```
final int SyncResult.AuthStatusCode.VALID
```

VALID_BUT_EXPIRES_SOON variable

User ID and password were valid at time of synchronization but expire soon.

Syntax

```
final int SyncResult.AuthStatusCode.VALID_BUT_EXPIRES_SOON
```

TableSchema interface

Specifies the schema of a table and provides constants defining the names of system tables.

Syntax

```
public interface TableSchema
```

Members

All members of TableSchema interface, including all inherited members.

Name	Description
“SYS_ARTICLES variable”	Name of the system table containing information on publication articles.
“SYS_COLUMNS variable”	Name of the system table containing information on the table columns in the database.
“SYS_FKEY_COLUMNS variable”	Name of the system table containing information on foreign key columns.
“SYS_FOREIGN_KEYS variable”	Name of the system table containing information on foreign keys in the database.
“SYS_INDEX_COLUMNS variable”	Name of the system table containing information on the index columns in the database.
“SYS_INDEXES variable”	Name of the system table containing information on the table indexes in the database.
“SYS_PRIMARY_INDEX variable”	Name of the primary key index of system tables.
“SYS_PUBLICATIONS variable”	Name of the system table containing information on database publications.
“SYS_TABLES variable”	Name of the system table containing information on the tables in the database.

Name	Description
“SYS_ULDATA variable”	Name of the system table containing information on system values.
“SYS_ULDATA_INTERNAL variable”	Type for internal system data.
“SYS_ULDATA_OPTION variable”	Type for option system data.
“SYS_ULDATA_PROPERTY variable”	Type for property system data.
“TABLE_IS_NOSYNC variable”	Bit flag denoting a table is a no-sync table (table that is never synchronized).
“TABLE_IS_SYSTEM variable”	Bit flag denoting a table is a system table.

Remarks

As of version 12, this interface only contains table-related constants. They include system table names, table flags, and types of data in *sysuldata*

SYS_ARTICLES variable

Name of the system table containing information on publication articles.

Syntax

```
final String TableSchema.SYS_ARTICLES
```

SYS_COLUMNS variable

Name of the system table containing information on the table columns in the database.

Syntax

```
final String TableSchema.SYS_COLUMNS
```

SYS_FKEY_COLUMNS variable

Name of the system table containing information on foreign key columns.

Syntax

```
final String TableSchema.SYS_FKEY_COLUMNS
```

SYS_FOREIGN_KEYS variable

Name of the system table containing information on foreign keys in the database.

Syntax

```
final String TableSchema.SYS_FOREIGN_KEYS
```

SYS_INDEX_COLUMNS variable

Name of the system table containing information on the index columns in the database.

Syntax

```
final String TableSchema.SYS_INDEX_COLUMNS
```

SYS_INDEXES variable

Name of the system table containing information on the table indexes in the database.

Syntax

```
final String TableSchema.SYS_INDEXES
```

SYS_PRIMARY_INDEX variable

Name of the primary key index of system tables.

Syntax

```
final String TableSchema.SYS_PRIMARY_INDEX
```

SYS_PUBLICATIONS variable

Name of the system table containing information on database publications.

Syntax

```
final String TableSchema.SYS_PUBLICATIONS
```

SYS_TABLES variable

Name of the system table containing information on the tables in the database.

Syntax

```
final String TableSchema.SYS_TABLES
```

SYS_ULDATA variable

Name of the system table containing information on system values.

Syntax

```
final String TableSchema.SYS_ULDATA
```

SYS_ULDATA_INTERNAL variable

Type for internal system data.

Syntax

```
final String TableSchema.SYS_ULDATA_INTERNAL
```

SYS_ULDATA_OPTION variable

Type for option system data.

Syntax

```
final String TableSchema.SYS_ULDATA_OPTION
```

SYS_ULDATA_PROPERTY variable

Type for property system data.

Syntax

```
final String TableSchema.SYS_ULDATA_PROPERTY
```

TABLE_IS_NOSYNC variable

Bit flag denoting a table is a no-sync table (table that is never synchronized).

Syntax

```
final short TableSchema.TABLE_IS_NOSYNC
```

Remarks

This value can be logically combined with other flags in the table_flags column of the SYS_TABLES table.

TABLE_IS_SYSTEM variable

Bit flag denoting a table is a system table.

Syntax

```
final short TableSchema.TABLE_IS_SYSTEM
```

Remarks

This value can be logically combined with other flags in the table_flags column of the SYS_TABLES table.

ULjException class

Supercedes the exceptions thrown by the UltraLiteJ database.

Syntax

```
public class ULjException
```

Members

All members of ULjException class, including all inherited members.

Name	Description
“getCausingException method”	Returns the ULjException causing this exception.
“getErrorCode method”	Returns the error code associated with the exception.
“getSqlOffset method”	Returns the error offset within the SQL string.

getCausingException method

Returns the ULjException causing this exception.

Syntax

```
abstract ULjException ULjException.getCausingException()
```

Returns

null, if no causing exceptions exist; otherwise, the ULjException.

getErrorCode method

Returns the error code associated with the exception.

Syntax

```
abstract int ULjException.getErrorCode( )
```

Returns

The error code.

getSqlOffset method

Returns the error offset within the SQL string.

Syntax

```
abstract int ULjException.getSqlOffset( )
```

Returns

-1 when there is no SQL string associated with the error message; otherwise, the zero-base offset within that string where the error occurred.

Unsigned64 class

Class to implement unsigned 64-bit binary value.

Syntax

```
public class Unsigned64
```

Members

All members of Unsigned64 class, including all inherited members.

Name	Description
"add method"	Add two values and place result in self.
"compare method"	Compare two long values.
"divide method"	Divide two values and place result in self.
"multiply method"	Multiply two values and place result in self.
"remainder method"	Returns the remainder when one value is divided by another.

Name	Description
“subtract method”	Subtract two values and place result in self.

Remarks

The intent is to keep values as longs, but to interpret them using the static methods in this class. Because only the static methods are of interest, the class cannot be instantiated.

Neither the C nor Java languages care about overflows; so this class doesn't either.

add method

Add two values and place result in self.

Syntax

```
final long Unsigned64.add(long v1, long v2)
```

Parameters

- **v1** the first operand
- **v2** the second operand

Returns

v1+v2

compare method

Compare two long values.

Overload list

Name	Description
“compare(int, int) method”	Compare two int values.
“compare(long, long) method”	Compare two long values.

compare(int, int) method

Compare two int values.

Syntax

```
final byte Unsigned64.compare(int v1, int v2)
```

Parameters

- **v1** the first value to be compared.
- **v2** the second value to be compared.

Returns

-1, when v2 is greater than v1; 0, when v1 equals v2; 1, when v2 is less than v1.

compare(long, long) method

Compare two long values.

Syntax

```
final byte Unsigned64.compare(long v1, long v2)
```

Parameters

- **v1** the first value to be compared.
- **v2** the second value to be compared.

Returns

-1, when v2 is greater than v1; 0, when v1 equals v2; 1, when v2 is less than v1.

divide method

Divide two values and place result in self.

Syntax

```
final long Unsigned64.divide(long v1, long v2)
```

Parameters

- **v1** the first operand
- **v2** the second operand

Returns

v1 \ v2

multiply method

Multiply two values and place result in self.

Syntax

```
final long Unsigned64.multiply(long v1, long v2)
```

Parameters

- **v1** the first operand
- **v2** the second operand

Returns

v1 * v2

remainder method

Returns the remainder when one value is divided by another.

Overload list

Name	Description
“remainder(long, long) method”	Returns the remainder when one value is divided by another.
“remainder(long, long, long) method”	Returns the remainder when one value multiplied by a given quotien is subtracted from another value (v1 - quot * v2).

remainder(long, long) method

Returns the remainder when one value is divided by another.

Syntax

```
final long Unsigned64.remainder(long v1, long v2)
```

Parameters

- **v1** the value to be divided.
- **v2** the value to divide by.

Returns

the Unsigned64 remainder as a long.

remainder(long, long, long) method

Returns the remainder when one value multiplied by a given quotien is subtracted from another value (v1 - quot * v2).

Syntax

```
final long Unsigned64.remainder(long v1, long v2, long quot)
```

Parameters

- **v1** the value to be divided.
- **v2** the value to divide by.
- **quot** the value of the quotien.

Returns

the Unsigned64 remainder as a long.

subtract method

Subtract two values and place result in self.

Syntax

```
final long Unsigned64.subtract(long v1, long v2)
```

Parameters

- **v1** the first operand
- **v2** the second operand

Returns

v1 - v2

UUIDValue interface

Describes a unique identifier (UUID or Universally Unique IDentifier) object.

Syntax

```
public interface UUIDValue
```

Members

All members of UUIDValue interface, including all inherited members.

Name	Description
“getString method”	Returns the String representation of the UUIDValue.
“isNull method”	Determines if the UUIDValue is null.

Name	Description
"set method"	Sets the UUIDValue with a String value.
"setNull method"	Sets the UUIDValue to null.

Remarks

Such enities are useful when a unique identifier is required and where the value can be arbitrary.

A UUIDValue can also be created by the SQL INSERT statement when no value is supplied for a column in a table where the column was created with the DEFAULT NEWID() clause.

See also

- "createUUIDValue method" on page 124

getString method

Returns the String representation of the UUIDValue.

Syntax

```
String UUIDValue.getString() throws ULjException
```

Returns

The String value.

isNull method

Determines if the UUIDValue is null.

Syntax

```
boolean UUIDValue.isNull()
```

Returns

true, if the object is null; otherwise, false.

set method

Sets the UUIDValue with a String value.

Syntax

```
void UUIDValue.set(String value) throws ULjException
```

Parameters

- **value** A numerical value represented as a String.

setNull method

Sets the UUIDValue to null.

Syntax

```
void UUIDValue.setNull() throws ULjException
```

UltraLiteJ system tables

systable system table

Each row in the systable system table describes a table in the database.

Column name	Column type	Description
table_id	INTEGER	A unique identifier for the table.
table_name	VAR-CHAR(128)	The name of the table.
table_flags	UNSIGNED SHORT	A bitwise combination of one of the following flags: ● TABLE_IS_SYSTEM ● TABLE_IS_NO_SYNC
table_data	UNSIGNED IN-TEGER	Internal use only.
table_partition_size	UNSIGNED IN-TEGER	The partition size of the table, if specified or if the table contains a default value; otherwise, null. This value is 0 if the partition size is out of range.

Constraints

PRIMARY INDEX (table_id)

syscolumn system table

Each row in the syscolumn system table describes a column.

Column name	Column type	Description
table_id	UNSIGNED INTE-GER	The identifier of the table to which the column belongs.
column_id	UNSIGNED INTE-GER	A unique identifier for the column.
column_name	VARCHAR(128)	The name of the column. See "Domain inter-face" on page 151.

Column name	Column type	Description
column_flags	TINY	A bitwise combination of the following flags describing the attributes: ● **0x01** Column is in the primary key. ● **0x02** Column is nullable.
column_domain	TINY	The column domain, which is an enumerated value indicating the domain of the column in the low-order 6 bits. The remaining bits are used internally.
column_length	UNSIGNED SHORT	The column length. For VARCHAR and BINARY type columns, which are defined in the Domain interface, this is the maximum length in bytes. For NUMERIC type columns, the precision is stored in the first byte, and the scale is stored in the second byte.
column_default_value	VARCHAR(128)	The default value for this column, which is specified by one of the COLUMN_DEFAULT values in the ColumnSchema interface. For example, COLUMN_DEFAULT_AUTOINC denotes an auto-incrementing default value. If a varchar column has DEFAULT AUTOFILENAME specified, then it stores the parameters prefix and extension using the encoding prefix|extension.
filename_colid	UNSIGNED INTEGER	Stores the column id of the referenced file_name column in the schema definition; otherwise, this column is null.

Constraints

PRIMARY KEY (table_id, column_id)

sysindex system table

Each row in the sysindex system table describes an index in the database.

Column name	**Column type**	**Description**
table_id	UNSIGNED INTEGER	A unique identifier for the table to which the index applies.
index_id	UNSIGNED INTEGER	A unique identifier for an index.
index_name	VARCHAR(128)	The name of the index.
index_flags	TINY	A bitwise combination of the following flags denoting the type of index and its persistence: ● **0x01** Unique key. ● **0x02** Unique index. ● **0x04** Index is persistent. ● **0x08** Primary key.
index_data	UNSIGNED INTEGER	Internal use only.

Constraints

PRIMARY KEY (table_id, index_id)

sysindexcolumn system table

Each row in the sysindexcolumn system table describes a column of an index listed in sysindex.

Column name	**Column type**	**Description**
table_id	UNSIGNED INTEGER	A unique identifier for the table to which the index applies.
index_id	UNSIGNED INTEGER	A unique identifier for the index that this index-column belongs to.
order	UNSIGNED INTEGER	The order of the column in the index.
column_id	UNSIGNED SHORT	A unique identifier for the column being indexed.

Column name	Column type	Description
index_column_flag	TINY	An indication of where the column in the index is kept, either in ascending (1) or descending (0) order.

Constraints

PRIMARY KEY (table_id, index_id, order)

syspublications system table

Each row in the syspublications system table describes a publication.

Column name	Column type	Description
publication_id	UNSIGNED INTEGER	A unique identifier for the publication.
publication_name	VARCHAR(128)	The name of the publication.
download_timestamp	TIMESTAMP	The time of the last download.
last_sync_sent	UNSIGNED INTEGER	An integer that tracks an upload sent to MobiLink.
last_sync_confirmed	UNSIGNED INTEGER	An integer that tracks an upload confirmed as being received by MobiLink.

Constraints

PRIMARY KEY (publication_id)

sysarticles system table

Each row in the sysarticles system table describes a table that belongs to a publication.

Column name	Column type	Description
publication _id	UNSIGNED INTEGER	An identifier for the publication that this article belongs to.
table_id	UNSIGNED INTEGER	The identifier of the table that belongs to the publication.

Column name	Column type	Description
predicate	VARCHAR(256)	The predicate expression specified by the associated WHERE clause of a CREATE PUBLICATION or ALTER PUBLICATION statement.

Constraints

PRIMARY KEY (publication_id, table_id)

sysforeignkey system table

Each row in the sysforeignkey system table describes a foreign key that belongs to a table.

Column name	Column type	Description
table_id	UNSIGNED INTEGER	The identifier of the table to which the foreign key belongs.
foreign_table_id	UNSIGNED SHORT	The identifier of the table to which this foreign-key-column refers to.
foreign_key_id	UNSIGNED INTEGER	The identifier of the foreign key.
name	VARCHAR(128)	The name of the foreign key.
index_name	VARCHAR(128)	The name of the index the foreign key refers to.

Constraints

PRIMARY KEY (table_id, foreign_key_id)

sysfkcol system table

Each row in the sysfkcol system table describes a foreign key column.

Column name	Column type	Description
table_id	UNSIGNED INTEGER	A unique identifier for the table to which the foreign key applies.

Column name	Column type	Description
foreign_key_id	UNSIGNED INTEGER	A unique identifier for the foreign key that this column belongs to.
item_no	UNSIGNED SHORT	The order of the column in the foreign key.
column_id	UNSIGNED SHORT	A unique identifier of the table column that refers to the foreign column.
foreign_column_id	UNSIGNED SHORT	A unique identifier of the table column that is being referred to.

Constraints

PRIMARY KEY (table_id, foreign_key_id, item_no)

sysuldata system table

Each row in the sysuldata system table names value pairs of options and properties.

Synchronization profiles are stored in the UltraLiteJ sysuldata system table with **name** set to the profile's name, **type** set to **ulsync**, and **long_setting** set to the UTF-8 encoding of the profile string. Exactly one of the column settings and **long_setting** has an assigned value depending on usage.

Column name	Column type	Description
long_setting	LONGBINARY	A BLOB for long values.
name	VARCHAR(128)	The name of the property.
setting	VARCHAR(128)	The value of the property.
type	VARCHAR(128)	One of either **sys** for internals, **opt** for options, or **prop** for properties

Constraints

PRIMARY KEY (name, type)

See also

- “UltraLite database properties” [*UltraLite - Database Management and Reference*]

UltraLiteJ utilities

Utilities are supplied with UltraLiteJ to perform maintenance and administration tasks on UltraLiteJ databases.

Utilities for Java SE

These utilities are for Java SE implementations of UltraLiteJ only—they are not designed for use with the BlackBerry smartphone environment.

UltraLiteJ Database Information utility (ULjInfo)

Displays information about an existing UltraLiteJ database.

Syntax

ULjInfo -c *filename* **-p** *password* [*options*]

Option	Description
-c *filename*	Required. The filename of the UltraLiteJ database to examine.
-p *password*	The password to connect to the UltraLiteJ database. The default is dba.
-q	Run in quiet mode—do not display messages.
-v	Display verbose messages.
-?	Display command line usage information.

Example

The following is an example of the output from the ULjInfo program (without the -v option):

```
C:\ULj\bin>ULjInfo.cmd -c ..\Samples\Demo1.ulj -p sql
SQL Anywhere UltraLite J Database Information Utility
Database name: ..\Samples\Demo1.ulj
Disk file: '..\Samples\Demo1.ulj'
Database ID: 0
Page size: 1024
0 rows for next upload
Date format: YYYY-MM-DD
Date order: YMD
Nearest century: 50
Numeric precision: 30
Numeric scale: 6
Time format: HH:NN:SS.SSS
Timestamp format: YYYY-MM-DD HH:NN:SS.SSS
Timestamp increment: 1
Number of tables: 1
Number of columns: 2
Number of publications: 0
Number of tables that will always be uploaded: 0
Number of tables that are never synchronized: 0
Number of primary keys: 1
Number of foreign keys: 0
Number of indexes: 0
Last download occurred on Thu Jul 05 11:31:05 EDT 2007
Upload OK: true
```

UltraLiteJ Database Load utility (ULjLoad)

Provides the capability to load an UltraLiteJ database from an XML source file. The XML file is often produced by the ULjUnload utility and is customizable.

Database configuration parameters
Database configuration parameters cannot be set in the command line. ULjLoad uses the default configuration when a new database is created. If you need to create a database with different configuration parameters, such as a different page size, you must create the database in Java first and then use **ULjLoad -a** to load the XML content into the database you created.

Syntax

ULjLoad -c *filename* **-p** *password* [*options*] *inputfile*

Option	Description
-a	Add information from XML file to an existing database. If this option is not specified, a new database is created.
-c *filename*	Required. Name of database file.
-d	Load data only; ignore schema information.
-f *directory*	Directory to retrieve data for columns larger than max blob size specified with -b option during ULjUnload.
-i	Insert rows for upload synchronization.
-n	Load schema information only; ignore row data.
-p *password*	Optional. The default password is DBA.
-q	Run in quiet mode—do not display messages.
-v	Display verbose messages.
-y	Overwrite output file if it exists (and -a option is not specified).

Option	Description
-?	Display command line usage information.
inputfile	Input file containing xml statements.

Example for loading blobfile types

This example assumes that you have applied the following SQL statement to your UltraLiteJ database and unloaded the file using the UltraLiteJ database unload utility. For more information about this example and unloading blobfile types, see "UltraLiteJ Database Unload utility (ULjUnload)" on page 277.

```
CREATE TABLE blobfile_example
    file_name CHAR(size) DEFAULT AUTOFILENAME( prefix, extension ),
    file_contents LONG BINARY STORE AS FILE( file_name ) [ CASCADE DELETE ]
```

ULLoad, while encountering a file exported by ULjUnload, treats the file_name column as a CHAR column and the file_contents column as a LONG BINARY column.

ULjLoad, while encountering the sam XML file, reconstructs the blobfile type. The file_name column stores the string <base_dir>/<reference_to_external_file>, where <base_dir> is the exact string in the -f option, and <reference_to_external_file> is the exact string found in the file_contents column in the XML file. The newly constructed database then contains valid references to the external files.

The load fails with an error message prompting the user to increase the size of the file_name column if the generated file name is too long.

Example

The -? option displays command line information.

```
uljload -?
```

When the ULjLoad utility command line includes the **-?** option, the following usage information appears:

```
SQL Anywhere UltraLiteJ Database Load Utility
Usage: uljload [options] <XML file>
        Create and load data into a new UltraLite J database from <XML file>.

Options:
        -a      Add to existing database.
        -c      <file> Database file.
        -d      Data only -- ignore schema.
        -f      <directory>
                Directory for loading .File paths.
        -i      Insert rows for upload synchronization.
        -n      Schema only -- ignore data.
        -p      Password to connect to database.
        -q      Quiet: do not print messages.
        -v      Verbose messages.
        -y      Overwrite file if it already exists.
```

UltraLiteJ Database Unload utility (ULjUnload)

Provides the capability to unload an UltraLiteJ database—either the data, the schema, or both—to an XML file.

Syntax

ULjUnload -c *filename* **-p** *password* [*options*] *outputfile*

Options	Description
-b *max-blob-size*	Maximum size (in bytes) of blob/char data output to XML.
-c *filename*	Required. Name of database file to unload.
-d	Unload only data; do not output schema information.
-e *table, ...*	Exclude data for tables named in list.
-f *directory*	Directory to store data for columns larger than max blob size specified with -b option.
-n	Unload schema information only; do not output data.
-p *password*	Password to connect to database. The default is dba.
-q	Run in quiet mode—do not display messages.
-t *table, ...*	Only output data for tables named in list.
-v	Display verbose messages.
-y	Overwrite output file if it already exists.
-?	Display option usage/help information.
outputfile	Output file name (this file contains xml statements that describe the data-base contents).

Example for unloading blobfile types

Apply the following sample SQL statement to your database to unload a blobfile type:

```
CREATE TABLE blobfile_example
    file_name CHAR(size) DEFAULT AUTOFILENAME( prefix, extension ),
    file_contents LONG BINARY STORE AS FILE( file_name ) [ CASCADE DELETE ]
```

When applying this example and using this utility, ULjUnload exports the file_name column as a regular CHAR but with an additional attribute, default_autofilename, which stores the prefix and extension strings in the form 'prefix', 'extension'. The file_contents column is exported as a LONG BINARY but with an additional attribute, filename_col, which stores the name of the referenced file_name column. In the row containing the blobfile columns, the content of the file_name column is unchanged. The file_contents column behaves like an externally-stored blob column and has the following form:

```
file_contents.File="tablename-columname-rownumber.bin"
```

The contents of the file_contents column are saved as bin files to the location specified by the -f option.

For more information about loading a blobfile type from the created XML file, see "UltraLiteJ Database Load utility (ULjLoad)" on page 276.

Example XML file contents

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<ul:ulschema xmlns:ul="urn:ultralite">
 <collation name="1252LATIN1" case_sensitive="no"/>
 <options>
  <option name="dateformat" value="YYYY-MM-DD"/>
  <option name="dateorder" value="YMD"/>
  <option name="nearestcentury" value="50"/>
  <option name="precision" value="30"/>
  <option name="scale" value="6"/>
  <option name="timeformat" value="HH:NN:SS.SSS"/>
  <option name="timestampformat" value="YYYY-MM-DD HH:NN:SS.SSS"/>
  <option name="timestampincrement" value="1"/>
 </options>
 <tables>
  <table name="ULCustomer" sync="changes">
   <columns>
    <column name="cust_id" type="integer" null="no"/>
    <column name="cust_name" type="char(30)" null="yes"/>
   </columns>
   <primarykey>
    <primarycolumn name="cust_id" direction="asc"/>
   </primarykey>
   <indexes/>
  </table>
 </tables>
 <uldata>
  <table name="ULCustomer">
   <row cust_id="2000" cust_name="Apple St. Builders"/>
   <row cust_id="2001" cust_name="Art's Renovations"/>
   <row cust_id="2002" cust_name="Awnings R Us"/>
   <row cust_id="2003" cust_name="Al's Interior Design"/>
   <row cust_id="2004" cust_name="Alpha Hardware"/>
   <row cust_id="2005" cust_name="Ace Properties"/>
   <row cust_id="2006" cust_name="A1 Contracting"/>
   <row cust_id="2007" cust_name="Archibald Inc."/>
   <row cust_id="2008" cust_name="Acme Construction"/>
   <row cust_id="2009" cust_name="ABCXYZ Inc."/>
   <row cust_id="2010" cust_name="Buy It Co."/>
   <row cust_id="2011" cust_name="Bill's Cages"/>
   <row cust_id="2012" cust_name="Build-It Co."/>
   <row cust_id="2013" cust_name="Bass Interiors"/>
   <row cust_id="2014" cust_name="Burger Franchise"/>
   <row cust_id="2015" cust_name="Big City Builders"/>
   <row cust_id="2016" cust_name="Bob's Renovations"/>
   <row cust_id="2017" cust_name="Basements R Us"/>
   <row cust_id="2018" cust_name="BB Interior Design"/>
   <row cust_id="2019" cust_name="Bond Hardware"/>
   <row cust_id="2020" cust_name="Cat Properties"/>
   <row cust_id="2021" cust_name="C &amp; C Contracting"/>
   <row cust_id="2022" cust_name="Classy Inc."/>
   <row cust_id="2023" cust_name="Cooper Construction"/>
   <row cust_id="2024" cust_name="City Schools"/>
   <row cust_id="2025" cust_name="Can Do It Co."/>
   <row cust_id="2026" cust_name="City Corrections"/>
```

```
      <row cust_id="2027" cust_name="City Sports Arenas"/>
      <row cust_id="2028" cust_name="Cantaloupe Interiors"/>
      <row cust_id="2029" cust_name="Chicken Franchise"/>
    </table>
  </uldata>
</ul:ulschema>
```

Utilities for Java ME (for BlackBerry smartphones)

These utilities are designed for use with the BlackBerry smartphone environment only.

UltraLiteJ Database Transfer utility (ULjDbT)

The ULjDbT utility provides the capability to transfer an UltraLiteJ database from a BlackBerry smartphone to an external device, such as a desktop, laptop, or server. In addition, you can delete a database, display database information, or view or email the database transfer log. The utility consists of two applications that must run simultaneously—the UltraLiteJ Database Transfer desktop application (ULjDbt) and the BlackBerry smartphone client application (*ULjDatabaseTransfer.cod*).

The UltraLiteJ Database Transfer desktop application

The desktop application receives UltraLiteJ databases using a USB or HTTP connection method. When you start the server application, it waits for a BlackBerry smartphone to transfer the database through the specified connection with the client application. The connection is closed either manually through the application interface, when the application times out, or when the transfer is complete.

The BlackBerry Smartphone client application

The BlackBerry Smartphone client application sends UltraLiteJ databases through a USB cable or a TCP port specified to the desktop application.

In addition, you can delete a database, display database information, or view or email the database transfer log.

The client application is a signed file located in the *UltraLite\UltraLiteJ\BlackBerry4.2* directory of your SQL Anywhere installation.

To start the client application

1. Load *ULjDatabaseTransfer.cod* from the *UltraLite\UltraLiteJ\BlackBerry4.2* directory of your SQL Anywhere installation.

   The client application icon appears in your list of applications.

2. Start the application and press the trackwheel.

3. On the **Database Connection** screen, complete the following fields:

- **Database Name** The name of the database to transfer to the external device. If the name starts with *file://* (this is case sensitive), then the client application will try to find the database in the file system; otherwise it will find the database in the object store.
- **Database Password** The database password used to allow data transfer. If you leave **Database Password** blank, the default password is used.

4. Click **Next**. The **Action** screen appears. This is the screen from which you can access all of the client application functionality.

To transfer a database using the BlackBerry Smartphone client application

1. On the **Action** screen, select the desired connection method (USB or HTTP).

2. For a USB transfer, select **USB database transfer**. For an HTTP transfer, proceed to step 4.

3. Follow the directions to start the database transfer desktop application (see **To receive a database using the UltraLiteJ Database Transfer application**).

   Note
   To ensure a successful database transfer, make sure that the device or simulator is connected to the BlackBerry Device Manager. For a simulator, make sure that a USB connection is simulated using **USB Cable Connected**.

   a. Click **Next** on the client application.
   b. On the desktop application, make sure **USB** is selected and click **Start**.
   c. The BlackBerry smartphone starts transferring the database to the external device. Progress information is displayed on the desktop application.
   d. Click **OK** on both the client and desktop applications to close them.

4. For an HTTP transfer, select **HTTP database transfer**.
   a. On the **HTTP Transfer** screen, click **Next**.
   b. Specify the following values:
      - **Host** The IP address of your desktop.
      - **Port** The port specified in **Connection Properties** on the desktop application.
      - **URL Suffix** The hostname of the server receiving the transfer, including the http:// suffix (this is required).

      and click **Next**.

Note
To transfer through the BES on a non-ident device, leave **Suffix** empty. On ident devices, use the suffix **;deviceside=false**.

To transfer through Direct TCP use the suffix **;deviceside=true**. Note that not all carriers support this.

The carrier's WAP gateway can be used if you know the APN information for it. You have to append that information to the suffix as well. Note that even if you are going through a BES, there may be a firewall between the BES and the machine you are running the UltraLiteJ Database Transfer utility on. In this case you need to use an SSL tunneler. On the **HTTP Transfer Parms** screen, specify the port and name or IP address of the SSL server running on the BES-side of the firewall. You also need to specify the port the SSL client is mapped to the transfer application.

If you are transferring a database from a BlackBerry simulator, you need to have a BlackBerry MDS simulator running or specify a URL suffix of **;deviceside=true** to the UltraLiteJ Database Transfer utility running in the BlackBerry simulator.

c. On the desktop application, make sure **HTTP** is selected and click **Start**.

d. Click **Next** on the client application.

   The BlackBerry smartphone starts transferring the database to the external device. Progress information is displayed on the desktop application.

e. Click **OK** on both the client and desktop applications to close them.

To receive a database using the UltraLiteJ Database Transfer application

1. Run *ULjDbTServ.cmd* from the *Bin32* directory of your SQL Anywhere installation.

   The UltraLiteJ Database Transfer application loads.

2. On the **Connect** tab, select the desired **Connection Method**.

3. Under **Connection Properties**, specify the following values:

   - **Port** This field only applies to HTTP connections. Type the TCP port number that you want the BlackBerry smartphone to connect to. Usually, this port number matches the port number specified to the UltraLiteJ Database Transfer utility running on the BlackBerry device; however, if you are using SSL, then this number can be different.
   - **BlackBerry Password** This field only applies to USB connections. Type the password used to access the connected BlackBerry smartphone when it is locked. Leave this field blank if there is no password.
   - **Timeout** The number of idle minutes before the server application times out and closes the connection.
   - **Output** Specify a file name and location to which to save the transferred database.

4. Click **Start** to open a connection to the BlackBerry smartphone. The server application waits until it either times out or establishes a connection. If you have specified an existing file, you are asked whether you want to overwrite it.

The **Logs** tab provides details on the server status and transfer progress, including error messages.

To delete a database

1. On the **Action** screen, select **Delete The Database**.
2. In the confirmation window, click **Delete** to delete the database.
3. In the **Database Deleted** window, click **OK** to close the client.

To view database information

1. On the **Action** screen, select **View Database Info**. Scroll down to view all the database information.
2. Click **Back** to return to the **Action** screen.

To view the log file

1. On the **Database Connection** screen, display the menu.
2. Click **Log**. The log screen is displayed.
3. To email the log file, enter the email address to send the log to and click **Send Email**. To return to the previous screen, press the return key.

Index