UltraLite®
Java Programming

Version 12.0.1

January 2012

Version 12.0.1
January 2012

Contents

About this book

This book describes the UltraLiteJ programming interface. With UltraLiteJ, you can develop and deploy database applications to Android and BlackBerry smartphones, and Java ME and Java SE platforms.

UltraLiteJ

UltraLiteJ is a Java API to the UltraLite database management system that is designed for the following devices and platforms:

- **Android smartphones**
- **BlackBerry smartphones** running OS 4.2 or later
- Java SE 1.6 or later platforms running on a computer or device

Note
When developing for Android smartphones, the UltraLiteJ API shares a common C++ code base with UltraLite for other platforms and its behavior is similar to that of other platforms. There are a few features available for accessing tables and rows without using SQL statements for which no API is provided on Android.

See also

- "UltraLite overview" [*UltraLite - Database Management and Reference*]
- "Choose an UltraLite API for Windows Mobile" [*UltraLite - Database Management and Reference*]

System requirements and supported platforms

Development platforms

To develop UltraLiteJ applications, you require the following:

- A Java IDE, such as Eclipse
- Java SE 1.6 or later

Target platforms

UltraLiteJ supports the following target platforms:

- **Android smartphones**
- **BlackBerry smartphones** running OS 4.2 or later
- Java SE 1.6 or later running on a computer or device

For information about UltraLite supported platforms, see http://www.sybase.com/detail?id=1002288.

UltraLiteJ API architecture

The UltraLiteJ API architecture is defined in the **com.ianywhere.ultralitej12** or **com.ianywhere.ultralitejni12** package. The following list describes some of the commonly used objects:

- **DatabaseManager** Provides methods for managing database connections, such as CreateDatabase and connect.
- **Connection** Represents a connection to an UltraLite database. You can create one or more Connection objects.
- **SyncParms** Synchronizes your UltraLite database with a MobiLink server.
- **PreparedStatement, ResultSet** Create dynamic SQL statements, make queries, and execute INSERT, UPDATE, and DELETE statements, and attain programmatic control over database result sets.

See also

- "UltraLiteJ API reference" on page 67

UltraLiteJ application development

This section provides information for developing applications with the UltraLiteJ API.

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

Quick start guide to UltraLiteJ application development

When creating an UltraLiteJ application, you typically complete the following data management tasks in your application code:

1. Import the UltraLiteJ API package into your Java file(s).

   The UltraLiteJ package name and location depends on the device you are developing applications for.

2. Create a new Configuration object to create or connect to a database.

   Configuration objects define where the client 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.

3. Create a new Connection object.

   Connection objects connect to a client database using the specifications defined in the Configuration object.

4. 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.

   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.

5. 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 also

- "Android and BlackBerry setup considerations" on page 4
- "Configuration interface [UltraLiteJ]" on page 97
- "Connection interface [UltraLiteJ]" on page 99
- "UltraLite SQL statements" [*UltraLite - Database Management and Reference*]
- "PreparedStatement interface [UltraLiteJ]" on page 169
- "ResultSet interface [UltraLiteJ]" on page 185

Android and BlackBerry setup considerations

There are UltraLiteJ API considerations to make before developing applications for Android or BlackBerry smartphones.

JAR resource files

When setting up an application for the UltraLiteJ API, make sure that your project is correctly configured to use the appropriate *UltraLiteJ12.jar* or *UltraLiteJNI12.jar* file.

The UltraLiteJ API for Android is stored in the *UltraLite\UltraLiteJ\Android\UltraLiteJNI12.jar* file of your SQL Anywhere installation. You must configure your Android development project to include the *UltraLiteJNI12.jar* file in the classpath. For more information, see "Tutorial: Building an Android application" on page 35.

For Android development, use the following statement to import the UltraLiteJ package into your Java file:

```
import com.ianywhere.ultralitejni12.*;
```

The UltraLiteJ API for BlackBerry, Java ME, and Java SE can be found in the *UltraLite\UltraLiteJ* directory of your SQL Anywhere installation. There is a subdirectory and *UltraLiteJ12.jar* file each target platform. You must configure your BlackBerry development project to include the *UltraLiteJ12.jar* file in the classpath. For more information, see "Tutorial: Building a BlackBerry application" on page 41.

For BlackBerry development, 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 Eclipse.

Java SE application notes

Java SE applications are not supported by the *UltraLiteJNI.jar* file. You must use the *UltraLite\UltraLiteJ\J2SE\UltraLiteJ12.jar* file.

UltraLite and UltraLite Java edition databases

On Android smartphones, UltraLiteJ provides an interface to the same UltraLite database management system that is provided for Windows Mobile, iPhone, and Windows. On BlackBerry smartphones,

UltraLiteJ provides an interface to the UltraLite Java edition database management system. UltraLite Java edition has similar features to UltraLite, but they are not identical. UltraLite Java edition databases are not interchangeable with UltraLite databases.

Android smartphones only support UltraLite databases. They can be created using Sybase Central or UltraLite command line utilities. For more information about creating an UltraLite database, see "UltraLite database creation" [*UltraLite - Database Management and Reference*].

BlackBerry smartphones only support UltraLite Java edition databases. For more information about creating or using an UltraLite Java edition database, see "Creating or connecting to an UltraLite or UltraLite Java edition database" on page 5.

See also

- "Choose an UltraLite API for Windows Mobile" [*UltraLite - Database Management and Reference*]
- "UltraLite, UltraLite Java edition, and SQL Anywhere feature comparisons" [*UltraLite - Database Management and Reference*]
- "UltraLite and UltraLite Java edition database limitations" [*UltraLite - Database Management and Reference*]

Creating or connecting to an UltraLite or UltraLite Java edition database

Java applications must connect to a database before operations can be performed on the data. This section explains how to create or connect to an UltraLite or UltraLite Java edition database with a specified password using the UltraLiteJ API.

Note

To create an UltraLite database without using the UltraLiteJ API, you can use Sybase Central or UltraLite command line utilities. For more information, see "UltraLite database creation" [*UltraLite - Database Management and Reference*].

To create an UltraLite Java edition database without using the UltraLiteJ API, you can perform one of the following tasks:

- Create the database using the ULjLoad utility. See "UltraLite Java Edition Database Load utility (ULjLoad)" [*UltraLite - Database Management and Reference*].
- Use the ulunload and ULjLoad utilities to convert an UltraLite database. See "UltraLite Database Unload utility (ulunload)" [*UltraLite - Database Management and Reference*].
- Deploy a Java SE application to a BlackBerry smartphone by copying the UltraLite Java database to an SD card or transferring it from MobiLink using the file transfer mechanism. See "MobiLink file transfers" [*UltraLite - Database Management and Reference*].

For more information about the differences between an UltraLite and an UltraLite Java edition database, see "UltraLite and UltraLite Java edition databases" on page 4.

A Configuration object is used to configure a database store. Several implementations of a Configuration object are provided. A unique implementation exists for every type of database store supported by the UltraLiteJ API. Each implementation provides a set of methods used to configure the database store.

The following table lists the available Configuration object implementations for the supported database stores:

Store type	UltraLiteJ API support
Android file system	See "ConfigFileAndroid interface [Android] [UltraLiteJ]".
RIM object (BlackBerry)	See "ConfigObjectStore interface [BlackBerry] [UltraLiteJ]".
Record	See "ConfigRecordStore interface (J2ME only) [UltraLiteJ]".
Java SE file system	See "ConfigFile interface [UltraLiteJ]".
Non-persistent (in memory)	See "ConfigNonPersistent interface [BlackBerry] [UltraLiteJ]".
Internal flash and SD card	See "ConfigFileME interface [BlackBerry] [UltraLiteJ]".

When creating a persistent database store, you can set the form of persistence for your Java application using your Configuration object. By default, shadow paging persistence is enabled. You can use simple paging persistence as an alternative by setting the setShadowPaging method of your persistent Configuration object to false. When simple paging persistence is enabled, you can set write-at-end persistence by setting the setWriteAtEnd method of your persistent Configuration object to true.

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.

For more information about setting persistent and non-persistent stores using a Configuration object, see:

- "ConfigNonPersistent interface [BlackBerry] [UltraLiteJ]" on page 79
- "ConfigPersistent.setShadowPaging method [UltraLiteJ]" on page 93
- "ConfigPersistent.setWriteAtEnd method [UltraLiteJ]" on page 94

After creating and configuring a Configuration object, you use a Connection object to create or connect to the database. Connection objects can also be used to perform the following operations:

- **Transactions** A transaction is 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 UltraLite is atomic. An insert involving multiple columns either inserts data to all the columns or to none of the columns.

  Transactions must be committed to the database using the commit method of the Connection object.

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

Create or connect to a database from your Java application.

Prerequisites

An existing Java application for an Android device or a BlackBerry smartphone that implements the UltraLiteJ API.

Context and remarks

Many.

Create or connect to a database from your Java application

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

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

   For Android smartphones:

   ```
   ConfigFileAndroid config =
       DatabaseManager.createConfigurationFileAndroid("DBname.udb");
   ```

   For BlackBerry smartphones:

   ```
   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 platforms:

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

   For any platform, you can create a non-persistent database Configuration object:

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

2. Set database properties using methods of the Configuration object.

   For example, you can set a new database password using the setPassword method:

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

   For BlackBerry smartphones, you can also set the database persistence using the Configuration object. The following example illustrates how to set write-at-end persistence prior to creating the database:

   ```
   config.setShadowPaging(false);
   config.setWriteAtEnd(true);
   ```

   For Android smartphones, you can use the setCreationString and setConnectionString methods to set additional creation and connection parameters, respectively.

   For more information about creation and connection parameters, see:

   - "UltraLite creation parameters" [*UltraLite - Database Management and Reference*]
   - "ConfigPersistent.setCreationString method [Android] [UltraLiteJ]" on page 89
   - "UltraLite connection parameters" [*UltraLite - Database Management and Reference*]
   - "ConfigPersistent.setConnectionString method [Android] [UltraLiteJ]" on page 89

3. Create a Connection object to create or connect to the database:

   To create a new database, use the following code to create the Connection object:

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

   The DatabaseManager.createDatabase method creates the database and returns a connection to it.

   To connect to an existing database, use the following code to create the Connection object:

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

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

Results

You can execute SQL statements from your Java application to create the tables and indexes in your database but you cannot change certain database creation parameters, such as the database name, password, or page size.

Next

None.

See also

- “Java SE example: Creating a database” on page 27
- “Connection interface [UltraLiteJ]” on page 99
- “DatabaseManager class [UltraLiteJ]” on page 124

Data creation and modification using SQL statements

Some SQL statements that are supported by UltraLite are not supported by UltraLite Java edition.

See also

- “UltraLite SQL statements” [*UltraLite - Database Management and Reference*]

Database objects

SQL statements and queries are used to create and update database objects. You can use these statements to create, update, or retrieve tables, indexes, foreign keys, and publications in your database. Schema information can be created with the Connection.prepareStatement and PreparedStatement.executeQuery methods.

See also

- “Connection.prepareStatement method [UltraLiteJ]” on page 111
- “PreparedStatement.executeQuery method [UltraLiteJ]” on page 172

Example

The following example illustrates how to use SQL statements to create a table:

```
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 **first_name**.

Modifying data using INSERT, UPDATE, and 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.

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, you can prepare, execute, and close an INSERT statement when you only need to insert a single entry, as illustrated in the following example:

```
INSERT INTO Department(dept_no, name) VALUES(2, 'Electronics');
```

The statement can also be created using Java String concatenation.

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 executing 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, the following statement could be prepared, executed and closed when only a single update is to be performed:

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

This statement can also be created using Java String concatenation.

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 executing 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();
   ```

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

   Alternatively, the following statement could be prepared, executed and closed as described in the section above when only a single delete is to be performed:

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

   This statement can also be created using Java String concatenation.

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();
```

Concatenation 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 concatenation 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.
    }
}
```

Retrieving data using SELECT

The SELECT statement allows you to retrieve information from the database. When you execute a SELECT statement, the PreparedStatement.executeQuery method returns a ResultSet object.

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();
   ```

See also

- “Connection.prepareStatement method [UltraLiteJ]” on page 111
- “PreparedStatement.executeQuery method [UltraLiteJ]” on page 172
- “ResultSet interface [UltraLiteJ]” on page 185

SQL result set navigation

Schema information can be obtained with the Connection.prepareStatement and PreparedStatement.executeQuery methods. These methods query the UltraLite database with a user-defined SQL statement, and store the result set in a ResultSet object.

You can traverse ResultSet objects using the navigational methods in the ResultSet interface. This interface contains the following navigational methods:

- **afterLast**[1] Position immediately after the last row.
- **beforeFirst**[1] Position immediately before the first row.
- **first**[1] Move to the first row.
- **last**[1] Move to the last row.
- **next** Move to the next row.
- **previous** Move to the previous row.
- **relative(offset)**[1] Move a specified number of rows relative to the current row, as specified by the signed offset value. Positive offset values move forward in the result set, relative to the current pointer position in the result set. Negative offset values move backward in the result set. An offset value of zero does not move the current location, but allows you to repopulate the row buffer.

[1] This method is not supported for UltraLite Java edition databases.

See also

- “Connection.prepareStatement method [UltraLiteJ]” on page 111
- “PreparedStatement.executeQuery method [UltraLiteJ]” on page 172
- “ResultSet interface [UltraLiteJ]” on page 185
- “ResultSet.afterLast method [Android] [UltraLiteJ]” on page 187
- “ResultSet.beforeFirst method [Android] [UltraLiteJ]” on page 188
- “ResultSet.first method [Android] [UltraLiteJ]” on page 188
- “ResultSet.last method [Android] [UltraLiteJ]” on page 202
- “ResultSet.next method [UltraLiteJ]” on page 203
- “ResultSet.previous method [UltraLiteJ]” on page 203
- “ResultSet.relative method [Android] [UltraLiteJ]” on page 203

Retrieval example

The following example demonstrates how to query a user-defined SQL statement, retrieve a row from the result set using the next method, 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();
```

Navigation example

The following example demonstrates how to build an integer list from a result set using the ResultSet.next method:

```
PreparedStatement ps = conn.prepareStatement("SELECT id FROM t");
ResultSet rs = ps.executeQuery();
List<Integer> l = new ArrayList<Integer> ();

while(rs.next()) {
    l.add(rs.getInt(1));
}
rs.close();
ps.close();
```

Transaction management

The UltraLiteJ API 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

- “Connection.commit method [UltraLiteJ]” on page 103
- “Connection.rollback method [UltraLiteJ]” on page 112

Schema information access

You can programmatically retrieve database schema descriptions. These descriptions are known as **schema information** and are accessible using system tables and the UltraLiteJ API schema interfaces.

Accessing schema information using system tables

Database schema information is stored in UltraLite or UltraLite Java edition system tables. You can access this information by executing a regular SQL query to select the desired information from the appropriate system table, and then accessing the result set.

Accessing schema information using schema interfaces

Some schema information can be accessed using schema interfaces instead of system tables. The UltraLiteJ API contains the following schema interfaces:

- **TableSchema** Returns information about the column and index configurations.
- **IndexSchema** Returns information about the columns in the index. IndexSchema objects can be retrieved from TableSchema objects.
- **ColumnSchema** Returns information about the columns in the table. ColumnSchema objects can be retrieved from TableSchema objects.

See also

- "SQL result set navigation" on page 14
- "UltraLite system tables" [*UltraLite - Database Management and Reference*]
- "UltraLite Java edition system tables" [*UltraLite - Database Management and Reference*]
- "Retrieving data using SELECT" on page 13
- "TableSchema interface [UltraLiteJ]" on page 254
- "IndexSchema interface [UltraLiteJ]" on page 167
- "ColumnSchema interface [UltraLiteJ]" on page 67

Error handling

You can use the ULjException and SQLCode classes to handle errors. Most UltraLite methods throw ULjException errors. You can use the ULjException.getErrorCode method to retrieve the SQLCode value assigned to the error. You can use the ULjException.toString method to obtain a descriptive text of the error. SQLCode errors are negative numbers indicating the error type, and can be referenced using constants such as ULjException.SQLE_INDEX_NOT_FOUND.

Example

The following example illustrates a Java class that uses the ULjException class to handle an error that may occur when connecting to an UltraLite Java edition database:

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

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;
}
```

See also

- “ULjException class [UltraLiteJ]” on page 258
- “ULjException.getErrorCode method [UltraLiteJ]” on page 259
- “SQL Anywhere error messages sorted by SQLCODE” [*Error Messages*]

Encrypting data in an UltraLite Java edition database

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. Data stored in UltraLite Java edition databases is not encrypted by default.

You must supply your own encryption control to encrypt data in an UltraLite Java edition database. The encryption control is set using the ConfigPersistent.setEncryption method, which accepts an EncryptionControl object to encrypt and decrypt pages.

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

Create and customize UltraLite Java edition database encryption or obfuscation API techniques in a BlackBerry application with the EncryptionControl interface.

Prerequisites

An existing Java application for a BlackBerry smartphone that implements the UltraLiteJ API.

Context and remarks

Many.

Encrypt or obfuscate data in an UltraLite Java edition database

1. Create a class that implements the EncryptionControl interface.

   The following example creates a new class named Encryptor that 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.
    }
}
```


3. Update your API code so that the new encryption control class is specified prior to connecting to the database.

   You specify the encryption control with the Configuration.setEncryption method. The following example illustrates how to set the encryption control, assuming that you have a Configuration object named config that references your database:

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

Results

Any data that is added or modified in the database is now encrypted or obfuscated, depending on how you implement your initialize, encrypt, and decrypt methods.

Next

None.

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 built-in security of the device can work for and against applications. For more information, see UltraLiteJ Security on BlackBerry Devices.

See also

- "UltraLite database security" [*UltraLite - Database Management and Reference*]
- "EncryptionControl interface [UltraLiteJ]" on page 145
- "Adjusting the cache size for an UltraLite database" [*UltraLite - Database Management and Reference*]
- "Java SE example: Obfuscating data" on page 31
- "Java SE example: Encrypting data" on page 31
- "EncryptionControl interface [UltraLiteJ]" on page 145

MobiLink data synchronization

The UltraLite Java edition management system contains a built-in change tracking system that allows database changes to be synchronized.

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

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), the 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 perform 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 [UltraLiteJ]" on page 232
- "SyncResult class [UltraLiteJ]" on page 247
- "Tutorial: Building an Android application" on page 35

Example

The following example demonstrates data synchronization with an UltraLiteJ application and can be found in *%SQLANYSAMP12%\UltraLiteJ\J2SE\Sync.java*:

```
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
            //

            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 *%SQLANYSAMP12%\J2SE\UltraLiteJ* directory.

For Android smartphones, a tutorial based on using the CustDB as a consolidated database is available.

Data synchronization on a BlackBerry smartphone

In a BlackBerry environment, data is always encrypted between the device and the BlackBerry Enterprise Server (BES). HTTPS is useful when encryption is required between the BES and the MobiLink server, or when the device is not managed by a BES.

Concurrent synchronization

Normally only one thread is allowed in the UltraLite 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 UltraLite runtime; however, no other thread can call the synchronize method for the database being synchronized while a synchronize operation is taking place.

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

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.

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.

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.

See also

- "MobiLink client network protocol options" [*MobiLink - Client Administration*]
- "StreamHTTPParms interface [UltraLiteJ]" on page 211
- "StreamHTTPSParms interface [UltraLiteJ]" on page 220

Synchronizing the CustDB application on a BlackBerry smartphone

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 *%SQLANYSAMP12%\UltraLiteJ\BlackBerry\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.

Files related to CustDB application

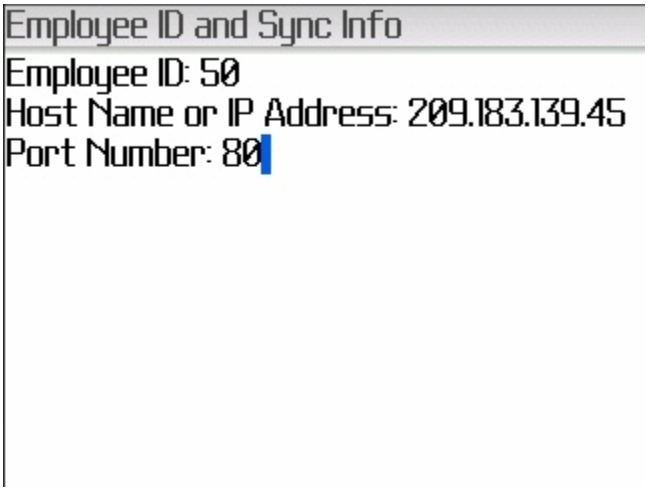
The following files in the CustDB project contain the database access code:

- ***CustDB.java*** This file contains all the basic database access methods. These methods include creating and connecting to databases, and 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

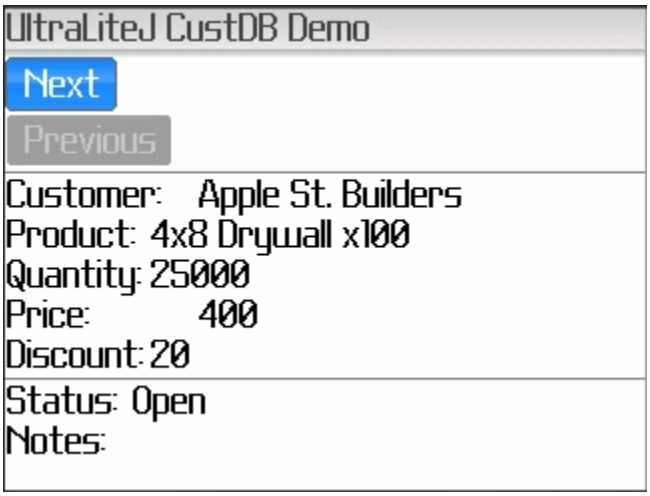
Run *%SQLANYSAMP12%\UltraLiteJ\BlackBerry\CustDB\mobilink.bat* to start a local MobiLink server.

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.



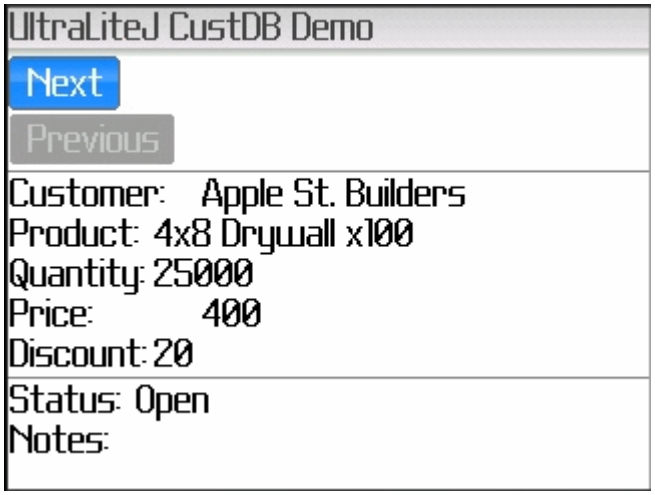
Once these values are specified and the settings are saved (by choosing **Menu** » **Save**), the application synchronizes with the specified server. The application only downloads orders from the server that match the Employee ID corresponding to the specified employee number (50). Only orders that 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 customer name, product ordered, quantity, price, and discount. The screen also shows the current status of the order and any notes pertaining to that order.



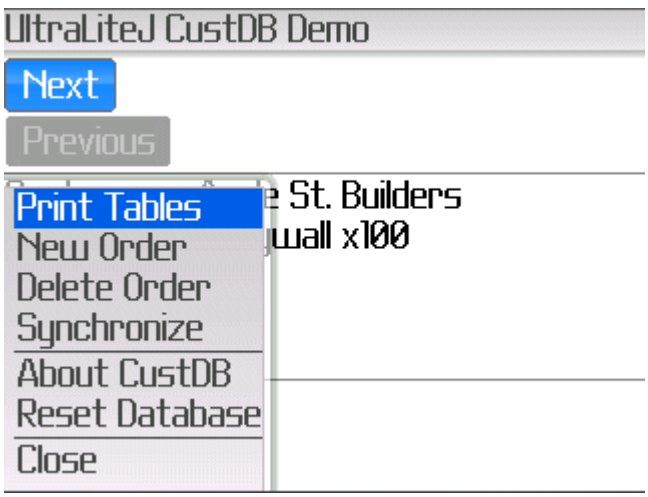
On this screen, you can add notes to the order, or change the status of the order (to either Approved or Denied). You can navigate through the orders using the **Next** and **Previous** buttons.

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



You may enter the quantity and discount values you require.

Before exiting the application, click **Synchronize** from the main menu to synchronize your changes and new orders with the consolidated database.



See also

- "UltraLiteJ application deployment" on page 25

Close an UltraLite or UltraLite Java edition database

An UltraLite or UltraLite Java edition database closes when all the concurrent connections have been released. You can release all the connections using the DatabaseManager.release method.

Note
You can use the Connection.release method to release the current connection.

See also

- "DatabaseManager.release method [UltraLiteJ]" on page 132
- "Connection.release method [UltraLiteJ]" on page 112

UltraLiteJ application deployment

For an UltraLiteJ application to run successfully, the UltraLiteJ API must be deployed with your distribution. The following table lists the 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
Android smartphone	*Android\UltraLiteJNI12.jar* *Android\ARM\libultralitej12.so*
BlackBerry smartphone	*BlackBerry4.2\UltraLiteJ12.cod* *BlackBerry4.2\UltraLiteJ12.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.

See also

- "UltraLite application build and deployment specifications" [*UltraLite - Database Management and Reference*]

Code examples

This section illustrates Java code samples 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 *%SQLANYSAMP12%\UltraLiteJ\J2SE* directory. It is recommended that you create backup copies of the original source code before modifying the file contents.

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.

The demo class found in *%SQLANYSAMP12%\UltraLiteJ\J2SE\Demo.java* is used by all the examples contained in this section of the documentation.

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
```

The output is saved in a file named *demos.out*.

BlackBerry examples

The following demos are available for BlackBerry developers in the *%SQLANYSAMP12%\UltraLiteJ\BlackBerry* directory:

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

A tutorial based on the HelloBlackBerry demo is available.

Android example

A sample Eclipse project that uses the CustDB sample database is available in the *%SQLANYSAMP12%\UltraLiteJ\Android\CustDB* directory. The source code can be found in *%SQLANYSAMP12%\UltraLiteJ\Android\CustDB\src\com\sybase\custdb*.

A tutorial based on the example is available.

See also

- "Tutorial: Building a BlackBerry application" on page 41
- "Tutorial: Building an Android application" on page 35

Java SE 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.

Run the CreateDb.java example

1. Change to the following directory: *%SQLANYSAMP12%\UltraLiteJ\J2SE*.
2. Run the CreateDb example:

   ```
   rundemo CreateDb
   ```

Notes

- Tables in UltraLite Java edition databases 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 a table.
- The primary index (**createPrimaryIndex**) is guaranteed to be unique.

Java SE example: Inserting rows

This example demonstrates how to insert rows in an UltraLite Java edition database.

Run the LoadDb.java example

1. Change to the following directory: *%SQLANYSAMP12%\UltraLiteJ\J2SE*.
2. Run the CreateDb example:

   ```
   rundemo CreateDb
   ```

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

   ```
   rundemo LoadDb
   ```

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.

See also

- "Java SE example: Creating a database" on page 27

Java SE example: Reading a table

In this example, a **PreparedStatement** object is obtained from a connection, and a **ResultSet** object is obtained from the **PreparedStatement** object. 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.

Run the ReadSeq.java example

1. Change to the following directory: *%SQLANYSAMP12%\UltraLiteJ\J2SE*.
2. Run the CreateDb example:

   ```
   rundemo CreateDb
   ```

3. Run the LoadDb example:

   ```
   rundemo LoadDb
   ```

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").

See also

- "Java SE example: Creating a database" on page 27
- "Java SE example: Inserting rows" on page 27

Java SE 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.

Run the ReadInnerJoin.java example

1. Change to the following directory: *%SQLANYSAMP12%\UltraLiteJ\J2SE*.
2. Run the CreateDb example:

   ```
   rundemo CreateDb
   ```

3. Run the LoadDb example:

   ```
   rundemo LoadDb
   ```

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

   ```
   rundemo ReadInnerJoin
   ```

See also

- "Java SE example: Creating a database" on page 27
- "Java SE example: Inserting rows" on page 27

Java SE example: Creating a sales database

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

Run the CreateSales.java example

1. Change to the following directory: *%SQLANYSAMP12%\UltraLiteJ\J2SE*.
2. Run the following command (the command is case sensitive):

   ```
   rundemo CreateSales
   ```

Java SE example: Aggregation and grouping

This example demonstrates UltraLiteJ support for the aggregation of results.

Run the SalesReport.java example

1. Change to the following directory: *%SQLANYSAMP12%\UltraLiteJ\J2SE*.
2. Run the CreateSales example:

   ```
   rundemo CreateSales
   ```

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

```
rundemo SalesReport
```

See also

- "Java SE example: Creating a sales database" on page 29

Java SE example: Retrieving rows in an alternative order

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

Run the SortTransactions.java example

1. Change to the following directory: *%SQLANYSAMP12%\UltraLiteJ\J2SE*.
2. Run the CreateSales example:

   ```
   rundemo CreateSales
   ```

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

   ```
   rundemo SortTransactions
   ```

See also

- "Java SE example: Creating a sales database" on page 29

Java SE 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.

Run the Reorg.java example

1. Change to the following directory: *%SQLANYSAMP12%\UltraLiteJ\J2SE*.
2. Run the CreateSales example:

   ```
   rundemo CreateSales
   ```

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

   ```
   rundemo Reorg
   ```

See also

- "Java SE example: Creating a sales database" on page 29

Java SE 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.

Run the Obfuscate.java example

1. Change to the following directory: *%SQLANYSAMP12%\UltraLiteJ\J2SE*.
2. Run the following command (the command is case sensitive):

```
rundemo Obfuscate
```

Java SE example: Encrypting data

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

Run the Encrypted.java example

1. Change to the following directory: *%SQLANYSAMP12%\UltraLiteJ\J2SE*.
2. Run the following command (the command is case sensitive):

```
rundemo Encrypted
```

This example is for Java SE. For a complete BlackBerry encryption sample, see *%SQLANYSAMP12%\UltraLiteJ\BlackBerryEncryption*.

Java SE 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.

Run the DumpSchema.java example

1. Change to the following directory: *%SQLANYSAMP12%\UltraLiteJ\J2SE*.
2. Run the CreateSales example:

```
rundemo CreateSales
```

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

```
rundemo DumpSchema
```

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
```

See also

- "Java SE example: Creating a sales database" on page 29

Java SE 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.

Run the Sync.java example

1. Change to the following directory: *%SQLANYSAMP12%\UltraLiteJ\J2SE*.
2. Run the CreateSales example:

   ```
   rundemo CreateSales
   ```

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 also

- "Java SE example: Creating a sales database" on page 29
- "Stop the MobiLink server" [*MobiLink - Server Administration*]

Tutorial: Building an Android application

This tutorial guides you through the development of an application for Android smartphones using the UltraLiteJ API and the Eclipse environment. In this tutorial, you run the application on a Windows simulator.

The Android application used in this tutorial is located in the *%SQLANYSAMP12%\UltraLiteJ\Android\CustDB* directory. The application code, located in the *src\com\sybase\custdb* directory, references the UltraLiteJ API to perform the following tasks:

- UltraLite remote database creation.
- SQL operations on the database.
- Data synchronization with the SQL Anywhere CustDB sample database using MobiLink.

The *res\menu* and *res\layout* directories illustrate how to create Android menu items and interfaces. You can view these files through Eclipse when you create the new Android project.

Required software

- Eclipse 3.5.2 or later
- Android SDK Starter Package
- Android Development Tools (ADT) Plug-in for Eclipse 1.1 or later
- SQL Anywhere 12 samples
- UltraLiteJ API

Competencies and experience

- Familiarity with Java
- Familiarity with Eclipse

See also

- http://developer.android.com/sdk/installing.html#Installing
- http://developer.android.com/sdk/eclipse-adt.html#installing
- http://developer.android.com/sdk/adding-components.html
- "UltraLiteJ API reference" on page 67

Lesson 1: Setting up a new Android project

In this lesson, you create a new Android project through the Eclipse Integrated Development Environment.

Set up a new Android project in Eclipse

1. Copy the Android libraries to your Android CustDB sample directory.

   Open a command prompt, change to the *%SQLANYSAMP12%\UltraLiteJ\Android\CustDB* directory, and then run the following command:

```
setup.bat
```

The *UltraLiteJNI12.jar* and *libultralitej12.so* files are copied into the *Android\CustDB\libs* and *Android\CustDB\libs\armeabi* directories.

2. Run Eclipse.

   The default application path is *C:\Eclipse\eclipse.exe*.

3. In the **Workspace** field, specify a working directory that is not your CustDB sample directory, and then click **OK**.

4. Import the CustDB project into Eclipse.
   a. Click **File** » **Import**.
   b. Expand the **General** directory, and then click **Existing Projects into Workspace**. Click **Next**.
   c. In the **Select Root Directory** field, type **%SQLANYSAMP12%\UltraLiteJ\Android\CustDB**. Select **Copy Projects Into Workspace**, and then click **Finish**.

5. Make sure that the appropriate Android SDK path is specified in Eclipse.

   Note
   You must install an Android SDK before specifying the path. For more information about tutorial prerequisites, see "Tutorial: Building an Android application" on page 35.

   a. Click **Window** » **Preferences**.
   b. In the left pane, click **Android**.
   c. In the **SDK Location** field, type the location of the Android SDK and then click **Apply**.
      A list of available build targets appears.
   d. Click **OK**.

6. Make sure that the UltraLiteJ library path is specified in Eclipse.
   a. Click **File** » **Properties**.
   b. In the left pane, click **Java Build Path** » **User libraries**.
   c. Click the **Libraries** tab.
   d. Click **UltraLiteJNI12.jar**, and then click **Edit**.
   e. From your working directory, open *\CustDB\libs\UltraLiteJNI12.jar*.
   f. Click **OK**.

7. Add the path to your UltraLiteJNI Javadoc documentation to the project.
   a. In the left pane, click **Javadoc Location**.
   b. Click **Browse**, and then open *%SQLANY12%\UltraLite\UltraLiteJ\Android\html*.
   c. Click **OK**.
   d. Click **OK** to close the window.

8. Build the project.

   Click **Project** » **Clean**, and then click **OK**.

   The project should build without errors occurring, but you may notice warnings listed under the **Problems** tab.

Lesson 2: Starting the MobiLink server

In this lesson, you start the MobiLink server.

Start the MobiLink server and synchronize the application

- Start MobiLink by running the following command from *%SQLANYSAMP12%\MobiLink\CustDB*:

  ```
  mlsrv12 -v+ -zu+ -c "DSN=SQL Anywhere 12 CustDB;UID=ml_server;PWD=sql" -x
  http(port=80) -ot ml.mls
  ```

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

See also

- "MobiLink server options" [*MobiLink - Server Administration*]

Lesson 3: Running your Android application

In this lesson, you run your application through an Android simulator.

Run your application on an Android simulator

1. Set up your Android virtual device in Eclipse.
   a. Click **Window** » **AVD Manager**.
   b. Click **New**.

      The **Create New Android Virtual Device (AVD)** window appears.
   c. In the **Name** field, type **my_avd**.
   d. In the **Target** field, click **Android 2.2 - API Level 8**.
   e. Click **Create AVD**.
   f. Close the **AVD Manager** window.
2. In the **Package Explorer** window, select **CustDB**.
3. From the **Run** menu, choose **Run As** » **Android Application**.

   The Android simulator loads.

4. Click **Menu**.

   Your Android application loads.

Lesson 4: Testing your Android application and synchronizing

In this lesson, you use your Android application to update the UltraLite remote database and synchronize the CustDB consolidated database.

Test your Android application and synchronize

1. Ensure that the **Employee ID** field is **50**, the **Host** field is **10.0.2.2**, and the **Port** field is **80**, and then click **Save**.

   The application automatically synchronizes and a set of customers, products, and orders is downloaded to the application from the CustDB consolidated database.

2. In the simulator, click **Menu** » **New**.
3. In the **Customer** field, choose **Ace Properties**.
4. In the **Product** field, choose **4x8 Drywall x100**.
5. In the **Quantity** field, type **999**.
6. In the **Discount** field, type **25**.
7. Click **OK** to add the new order.
8. Synchronize the application with the CustDB consolidated database.

   In the simulator, click **Menu** and then click **Sync**.

9. Connect to the CustDB consolidated database with Interactive SQL.
   a. Click **Start** » **Programs** » **SQL Anywhere 12** » **Administration Tools** » **Interactive SQL**, or run the following command:

      ```
      dbisql
      ```

   b. Click **ODBC Data Source Name** and choose **SQL Anywhere 12 CustDB**.
   c. Click **Connect**.
10. Verify that the synchronization was successful.

    Execute the following SQL statement in Interactive SQL:

    ```
    SELECT order_id, disc, quant, notes, status, c.cust_id,
        cust_name, p.prod_id, prod_name, price
      FROM ULOrder o, ULCustomer c, ULProduct p
    ```



```
WHERE o.cust_id = c.cust_id
AND o.prod_id = p.prod_id
AND c.cust_name = 'Ace Properties'
AND p.prod_name = '4x8 Drywall x100';
```

Synchronization was successful when an order entry appears in Interactive SQL.

11. Close the simulator window.

Cleaning up

Remove tutorial materials from your computer.

Remove tutorial materials from your computer

1. Close Eclipse.

   Click **File** » **Exit**.

2. Close MobiLink, Interactive SQL, and synchronization client windows by right-clicking each task bar item and clicking **Exit** or **Shut Down**.

3. Reset the CustDB database.

   Run the following command from the *%SQLANYSAMP12%\UltraLite\CustDB* directory:

   ```
   makedbs
   ```

Tutorial: Building a BlackBerry application

This tutorial guides you through the development of an application for BlackBerry smartphones using the UltraLiteJ API and the Eclipse environment. In this tutorial, you run the application on a Windows simulator and deploy it to a BlackBerry smartphone. Code samples are provided throughout the tutorial, and a complete code listing is available at the end of the tutorial.

Required software

- Eclipse 3.5 or later
- BlackBerry Java Plug-in for Eclipse 1.1 or later
- BlackBerry Email and MDS Services Simulator Package v4.1.4
- BlackBerry JDE 6.0 or later
- UltraLiteJ API

Competencies and experience

- Familiarity with Java
- Familiarity with Eclipse

See also

- http://us.blackberry.com/developers/javaappdev/
- http://us.blackberry.com/developers/javaappdev/javadevenv.jsp

Part 1: Creating a new BlackBerry application

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

Lesson 1: Setting up a new BlackBerry project

In this lesson, you create a new BlackBerry project through the Eclipse Integrated Development Environment.

Set up a new BlackBerry project in Eclipse

1. Run Eclipse.

   The default application path is *C:\Eclipse\eclipse.exe*.

2. In the **Workspace** field, specify a working directory and then click **OK**.

   This tutorial assumes that you are working in the *C:\HelloBlackBerry* directory.

3. Create your new project.

   Click **File** » **New** » **Project**.

4. Expand the **BlackBerry** folder and then select **BlackBerry Project**.
5. Click **Next**.
6. In the **Project Name** field, type **HelloBlackBerry**.
7. Click **Finish**.
8. Add the UltraLiteJ JAR file to the project.
   a. Display the **Package Explorer** window in Eclipse if it is not already shown.

      Click **Window** » **Show View** » **Package Explorer**.
   b. Access the package properties of your project.

      Click **HelloBlackBerry** in the **Package Explorer** window and then click **File** » **Properties**.
   c. In the left pane, click **Java Build Path** and then click the **Libraries** tab.
   d. Click **Add External Jars**, and then open *\UltraLite\UltraLiteJ\BlackBerry4.2\UltraLiteJ12.jar* from your SQL Anywhere installation directory.
9. Add the path to your UltraLiteJ Javadoc documentation to the project.
   a. In the **JARs And Class Folders On The Build Path** list, expand **UltraLiteJ12.jar** and click **JavaDoc Location**.
   b. Click **Edit**.

      The **Javadoc For UltraLiteJ12.Jar** window appears.
   c. Click **Browse** and then open *\UltraLite\UltraLiteJ\BlackBerry4.2\html* from your SQL Anywhere installation directory.
   d. Click **OK** to close the **Javadoc For UltraLiteJ12.Jar** window.
10. Click **OK** to close the window.

Lesson 2: Writing and testing your BlackBerry application

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

Write and test a sample application in Eclipse

1. Add an **Application** class to your project.
   a. In the **Package Explorer** window, expand **HelloBlackBerry** and click **src**.
   b. Click **File** » **New** » **Class**.

      The **New Java Class** window appears.
   c. In the **Name** field, type **Application**.
   d. Under the **Which Method Stubs Would You Like To Create** option, select **Public Static Void Main([String() Args])**.

e. Click **Finish**.

The *Application.java* file appears under your project in the **Package Explorer** window.

2. Modify the **Application** class.

Double-click *Application.java* in the **Package Explorer** window, and then add a constructor and a **main** method.

Your *Application.java* code should look like the following code:

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

3. Add a **HomeScreen** class to your project.

a. In the **Package Explorer** window, expand **HelloBlackBerry** and click **src**.

b. Click **File** » **New** » **Class**.

The **New Java Class** window appears.

c. In the **Name** field, type **HomeScreen**.

d. Click **Finish**.

The *HomeScreen.java* file appears under your project in the **Package Explorer** window.

4. Modify the **HomeScreen** class so that it displays a title and status messages.

Double-click *HomeScreen.java* in the **Package Explorer** window, and then update the code so that it displays a title and a status message.

Your *HomeScreen.java* code should look like the following code:

```
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.

5. Run the simulator.

   In the **Package Explorer** window, click *Application.java*, and then click **Run** » **Run As** » **BlackBerry Simulator**.

   Note
   If multiple projects are open in your workspace, click **Run** » **Run Configurations**, select **HelloBlackBerry**, and then click **Run**.

   The **HelloBlackBerry** project compiles and then the simulator window appears.

   Ensure that the project compiles without errors by selecting the **Problems** tab in Eclipse.

6. From the simulator menu, click **Simulate** » **Set IT Policy**.

   The **Set IT Policy** window appears.

7. In the **Policy** field, click **Allow Third Party Apps To Use Persistent Store** » >>.

8. Click **Set** and then click **Close**.

9. Launch your application.

   In the simulator window, navigate to **Downloads** and then run the **HelloBlackBerry** application.

   A screen appears that displays the **Hello BlackBerry** title bar and the **Status: Started** text.

10. Stop the simulation.

    In the simulator window, click **File** » **Exit**.

Lesson 3: Creating an UltraLite Java edition database

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

Update the sample application to create a database

1. Modify the **HomeScreen** class to instantiate a **DataAccess** object.

   The following is the complete and updated **HomeScreen** class code listing:

```
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);

        // 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;
}
```

Eclipse may report warnings indicating that **DataAccess** cannot be resolved. The **DataAccess** object is held as a class-level variable so that it can be accessed from other parts of the code. You create the **DataAccess** class in the next step.

2. Add a **DataAccess** class to your project.
   a. In the **Package Explorer** window, expand **HelloBlackBerry** and click **src**.
   b. Click **File** » **New** » **Class**.

      The **New Java Class** window appears.
   c. In the **Name** field, type **DataAccess**.
   d. Click **Finish**.

      The *DataAccess.java* file appears under your project in the **Package Explorer** window.

3. Modify the **DataAccess** class so that it contains a **getDataAccess** method that ensures a single database connection.

   Double-click *DataAccess.java* in the **Package Explorer** window, and then replace the code with the following snippet:

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

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 following steps are needed to create or connect to an UltraLite Java edition database:

a. Define a configuration. In this tutorial, the configuration object is defined by the **ConfigObjectStore** interface, which allows you to configure a persistent database that resides in the BlackBerry object store.

b. Attempt to connect to the database. In this tutorial, the database is created using the **createDatabase** method when the connection attempt fails. This method then returns an open connection.

4. Click **File** » **Save**.

5. Run the simulator.

   In the **Package Explorer** window, click *Application.java*, and then click **Run** » **Run As** » **BlackBerry Simulator**.

   Note
   If multiple projects are open in your workspace, click **Run** » **Run Configurations**, select **HelloBlackBerry**, and then click **Run**.

   The **HelloBlackBerry** project compiles and then the simulator window appears.

   Ensure that the project compiles without errors by selecting the **Problems** tab in Eclipse.

6. From the simulator menu, click **File** » **Load Java Program**.

7. Browse to the *\UltraLite\UltraLiteJ\BlackBerry4.2* directory of your SQL Anywhere installation and open the *UltraLiteJ12.cod* file.

   Note
   You may need to copy *UltraLiteJ12.cod* and the DBG files to the working simulator directory (for example, *C:\Eclipse\plugins\net.rim.ejde.componentpack6.0.0_6.0.0.0.26\components\simulator*) to run the application. When copied, you do not need to load the Java program from the simulator menu.

8. From the simulator menu, click **Simulate** » **Set IT Policy**.

   The **Set IT Policy** window appears.

9. In the **Policy** field, click **Allow Third Party Apps to Use Persistent Store** and then click >>.

10. Click **Set** and then click **Close**.

11. Launch your application.

    In the simulator window, navigate to **Downloads** and then run the **HelloBlackBerry** application.

    A screen appears that displays the **Hello BlackBerry** title bar and the **Status: Connected** text, which indicates that the application has successfully connected to the UltraLite Java edition database.

12. Stop the simulation.

    In the simulator window, click **File** » **Exit**.

See also

- "ConfigObjectStore interface [BlackBerry] [UltraLiteJ]" on page 80
- "DatabaseManager.createDatabase method [UltraLiteJ]" on page 129

Lesson 4: Creating a table in the database

In this lesson, you create a table named **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

Update the sample application to create a table in the database

1. Add a new method to the **DataAccess** class that creates the **Names** table.

   Double-click *DataAccess.java* in the **Package Explorer** window, and then insert the following code after the **getDataAccess** method:

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

This method throws an exception if the **Names** table already exists in the database.

2. Call the **createDatabaseSchema** method from the **getDataAccess** method.

   Remove the code comments from the **getDataAccess** method so that the **createDatabaseSchema** calls look like the following code snippet:

   ```
   _da.createDatabaseSchema()
   ```

3. Compare your **DataAccess** code to the complete code listing of the **DataAccess** class to ensure that they are identical.

4. Click **File** » **Save**.

5. Run the simulator to verify that the application compiles and runs.

   In the **Package Explorer** window, click *Application.java*, and then click **Run** » **Run As** » **BlackBerry Simulator**.

   Note
   If multiple projects are open in your workspace, click **Run** » **Run Configurations**, select **HelloBlackBerry**, and then click **Run**.

   The **HelloBlackBerry** project compiles and then the simulator window appears.

   Ensure that the project compiles without errors by selecting the **Problems** tab in Eclipse.

6. From the simulator menu, click **File** » **Load Java Program**.

7. Browse to the *\UltraLite\UltraLiteJ\BlackBerry4.2* directory of your SQL Anywhere installation and open the *UltraLiteJ12.cod* file.

   Note
   You may need to copy *UltraLiteJ12.cod* and the DBG files to the working simulator directory (for example, *C:\Eclipse\plugins\net.rim.ejde.componentpack6.0.0_6.0.0.0.26\components\simulator*) to run the application. When copied, you do not need to load the Java program from the simulator menu.

8. From the simulator menu, click **Simulate** » **Set IT Policy**.

   The **Set IT Policy** window appears.

9. Click **Policy** » **Allow Third Party Apps to Use Persistent Store** and then click >>.

10. Click **Set** and then click **Close**.

11. Launch your application.

    In the simulator window, navigate to **Downloads** and then run the **HelloBlackBerry** application.

    A screen appears that displays the **Hello BlackBerry** title bar and the **Status: Connected** text, which indicates that the application has successfully connected to the UltraLite Java edition database.

12. Stop the simulation.

    In the simulator window, click **File** » **Exit**.

See also

- "DataAccess.java" on page 62

Lesson 5: Adding data to the table

In this lesson, you add the following controls to your application:

- 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.

Update the sample application to request data from users

1. Update the **HomeScreen** class to add the controls.

   Double-click *HomeScreen.java* in the **Package Explorer** window, and then insert the following code above the **try-catch** statement that calls 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);
```

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

   Insert the following code below the **private DataAccess _da;** statement:

   ```
   private EditField _nameEditField;
   private ObjectListField _nameListField;

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

3. Add a new method to the **DataAccess** class that inserts a row into a table.

   Double-click *DataAccess.java* in the **Package Explorer** window, and then insert the following code after the **createDatabaseSchema** method:

   ```
   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();
           ps.close();
       }
       catch(ULjException uex) {
           Dialog.alert("ULjException: " + uex.toString());
       }
       catch( Exception ex ){
           Dialog.alert("Exception: " + ex.toString());
       }
   }
   ```

4. Add a **NameRow** class to your project.

   a. In the **Package Explorer** window, expand **HelloBlackBerry** and click **src**.

   b. Click **File** » **New** » **Class**.

      The **New Java Class** window appears.

   c. In the **Name** field, type **NameRow**.

   d. Click **Finish**.

      The *NameRow.java* file appears under your project in the **Package Explorer** window.

5. Update the **NameRow** class so that it can store a row in the **Names** table as an object.

   Double-click *NameRow.java* in the **Package Explorer** window, and then replace the code with the following snippet:

   ```
   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;

}
```

The **toString** method is used by the **ObjectListField** control.

6. Add a new method to the **DataAccess** class that reads a row into a Vector of objects.

   Double-click *DataAccess.java* in the **Package Explorer** window, and then insert the following code after the **insertName** method:

```
    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());
        }
        return nameVector;
    }
```

7. Add a new method to the **HomeScreen** class that refreshes the contents of the list displayed on the screen.

   Double-click *HomeScreen.java* in the **Package Explorer** window, and then insert the following code after the **_addToListMenuItem** method:

```
    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);
            }
        }
```

8. Update the **HomeScreen** class so that it calls the **refreshNameList** method, ensuring that the list is filled when the application starts.

   Insert the following code before the end of the **HomeScreen** constructor:

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

9. Add a new method to the **HomeScreen** class that adds a row to the list on the screen.

   Insert the following code after the **refreshNameList** method:

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

10. Update the **run** method in the **HomeScreen** class so that it calls the **onAddToList** method.

    Replace the line of code that states \\ **TODO** with the following code snippet:

    ```
                onAddToList();
    ```

11. Click **File** » **Save All**.

12. Run the simulator to verify that the application compiles and runs.

    In the **Package Explorer** window, click *Application.java*, and then click **Run** » **Run As** » **BlackBerry Simulator**.

    Note
    If multiple projects are open in your workspace, click **Run** » **Run Configurations**, select **HelloBlackBerry**, and then click **Run**.

    The **HelloBlackBerry** project compiles and then the simulator window appears.

    Ensure that the project compiles without errors by selecting the **Problems** tab in Eclipse.

13. From the simulator menu, click **File** » **Load Java Program**.

14. Browse to the *\UltraLite\UltraLiteJ\BlackBerry4.2* directory of your SQL Anywhere installation and open the *UltraLiteJ12.cod* file.

Note
You may need to copy *UltraLiteJ12.cod* and the DBG files to the working simulator directory (for example, *C:\Eclipse\plugins\net.rim.ejde.componentpack6.0.0_6.0.0.0.26\components\simulator*) to run the application. When copied, you do not need to load the Java program from the simulator menu.

15. From the simulator menu, click **Simulate** » **Set IT Policy**.

    The **Set IT Policy** window appears.

16. In the **Policy** field, click **Allow Third Party Apps To Use Persistent Store** » >>.

17. Click **Set** and then click **Close**.

18. Launch your application.

    In the simulator window, navigate to **Downloads** and then run the **HelloBlackBerry** application.

    A screen appears that displays the **Hello BlackBerry** title bar, the **Status: Connected** text, and a **Name** field.

19. At the name field, type **John Smith**.

20. Click ***EMPTY*** and then choose **Add**.

    John Smith appears in the list, which indicates that the name entry was added to the **Names** table in the database.

    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.

21. Stop the simulation.

    In the simulator window, click **File** » **Exit**.

Lesson 6: Deploying your application to a BlackBerry 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/.

Sign and deploy your application

1. Start the BlackBerry Signature Tool.
2. Run the following command from the *\bin* directory of your BlackBerry JDE installation (for example, *C:\Program Files\Research in Motion\BlackBerry JDE 6.0.0\bin*).

   ```
   start javaw -jar SignatureTool.jar
   ```

3. Browse to *C:\HelloBlackBerry* and select the *HelloBlackBerry.cod* file, your compiled application.
4. Click **Request To Sign The File**.
5. Click **Close** to close the signature tool.
6. Connect your BlackBerry to your computer using the USB cable, and ensure that the BlackBerry Desktop Manager can see the device.
7. Click **Application Loader** and follow the instructions in the wizard.
8. Browse to the *HelloBlackBerry.alx* file and add it to your device.
9. Browse to the *BlackBerry4.2\UltraLiteJ.alx* file and add it to your device.

   You can now use the application on your BlackBerry smartphone.

Part 2: Using MobiLink to synchronize the BlackBerry application

This part of the tutorial extends the BlackBerry application to support MobiLink synchronization. You complete the following tasks:

- Create a SQL Anywhere database that can synchronize with your UltraLite Java edition database.
- Start a MobiLink server to handle synchronizations.
- Update your BlackBerry application to support MobiLink synchronization.
- Synchronize your BlackBerry application with the consolidated database.

Lesson 1: Setting up your MobiLink consolidated database

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

Set up a consolidated database

1. Create a working directory to store the SQL Anywhere database.

This tutorial assumes that you are working in the *c:\HelloBlackBerry\database* directory.

2. Run the following command to create an empty SQL Anywhere database:

   ```
   dbinit HelloBlackBerry.db
   ```

3. Create an ODBC data source to connect to the database.
   a. Click **Start** » **Programs** » **SQL Anywhere 12** » **Administration Tools** » **ODBC Data Source Administrator**.
   b. Click the **User DSN** tab, and then click **Add**.
   c. In the **Create New Data Source** window, click **SQL Anywhere 12** and click **Finish**.
   d. Click the **ODBC** tab.
   e. In the **Data Source Name** field, type **HelloBlackBerry**.
   f. Click the **Login** tab.
   g. In the **User ID** field, type **DBA**.
   h. In the **Password** field, type **sql**.
   i. From the **Action** list, click **Start And Connect To A Database On This Computer**.
   j. In the **Database File** field, type *c:\tutorial\database\HelloBlackBerry.db*.
   k. In the **Server Name** fields, type **HelloBlackBerry**.
   l. Disable the **Stop Database After Last Disconnect** option.
   m. Click **OK**.
   n. Click **OK** on the **ODBC Data Source Administrator** window.

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

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

5. Execute the following SQL statement in Interactive SQL to create the **Names** table on the consolidated database:

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

6. Close Interactive SQL.

   Click **File** » **Exit**.

Lesson 2: Setting up a MobiLink server and deploying a synchronization model

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

Set up the MobiLink server and deploy a synchronization model

1. Click **Start** » **Programs** » **SQL Anywhere 12** » **Administration Tools** » **Sybase Central**.
2. Click **Tools** » **MobiLink 12** » **New Project**.

   The **Create Project Wizard** appears.
3. In the **Name** field, type **rim_project**.
4. In the **Location** field, type *C:\HelloBlackBerry\database*, and then click **Next**.
5. Select the **Add A Consolidated Database To The Project** option.
6. In the **Database Display Name** field, type **HelloBlackBerry**.
7. Click **Edit**.
8. Perform the following tasks on the **Connect To A Generic ODBC Database** page:
   a. In the **User ID** field, type **DBA**.
   b. In the **Password** field, type **sql**.
   c. In the **ODBC Data Source Name** field, click **Browse** and select **HelloBlackBerry**.
   d. Click **OK**, and then click **Save**.
9. Select the **Remember The Password** option, and then click **Next**.
10. Select **Create A New Model**, and then click **Finish**.
11. Click **OK**.

    The **Create Synchronization Model Wizard** appears.
12. In the **What Do You Want To Name The New Synchronization Model** field, type **HelloBlackBerrySyncModel** and then click **Next**.
13. Select the **HelloBlackBerry** consolidated database from the list, and then click **Next**.
14. Click **No, Create A New Remote Database Schema**, and then click **Next**.
15. On the **New Remote Database Schema** page, ensure that only the **Names** table is selected from the **Which Consolidated Database Tables And Columns Do You Want To Have In Your Remote Database** list, and then click **Next**.
16. Click **Timestamp-based Download**, and then click **Finish**.

    Timestamp-based downloads minimize the amount of data that is transferred because only data that has been updated since the last download is transmitted.
17. In the left pane of Sybase Central under **MobiLink 12**, expand **rim_project**, **Synchronization Models** and then **HelloBlackBerrySyncModel**.

18. Click **File** » **Deploy**.
19. Under the **Specify The Deployment Details For One Or More Of The Following** option, ensure that only the **Consolidated Database** option is selected. Click **Next**.
20. Perform the following tasks on the **Consolidated Database Deployment Destination** page:
    a. Select **Save Changes To The Following SQL File** and accept the default location for the file.

       MobiLink generates a *.sql* file that makes changes to the consolidated database to set up for synchronization. You can examine the *.sql* file later and make your own changes. Then, you must run the *.sql* file yourself.

    b. Immediately apply the changes to the consolidated database.

       Select **Connect To The Consolidated Database To Directly Apply The Changes**.

    c. Select the **HelloBlackBerry** consolidated database from the list.
    d. Click **Next**.

    A prompt appears asking if you want to create the *consolidated* directory. Click **Yes**.
21. On the **MobiLink User and Synchronization Profile** page, type **mluser** for the user name and **mlpassword** for the password, and then click **Finish**.

    The synchronization model to the consolidated database can now be deployed.

Lesson 3: Adding MobiLink support to your BlackBerry application

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

Add MobiLink synchronization capabilities to your BlackBerry application

1. Update the **HomeScreen** class to add a **Sync** menu item.

   Double-click *HomeScreen.java* in the **Package Explorer** window, and then insert the following code above the **try-catch** statement that calls the **getDataAccess** method:

   ```
   // Add sync menu item
   addMenuItem(_syncMenuItem);
   ```

2. Update the **HomeScreen** class to add a new method that defines the menu item in the class variable declarations.

   Insert the following code below the **_addToListMenuItem** method:

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

3. Update the **HomeScreen** class to add the **onSync** method that is called in the previous step.

Insert the following code below the **onAddToList** method:

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

4. Update the **DataAccess** class to define the **_syncParms** variable.

   Double-click *DataAccess.java* in the **Package Explorer** window, and then insert the following code below the **private static DataAccess _da;** call:

   ```
   private static SyncParms _syncParms;
   ```

5. Update the **DataAccess** class to add a **sync** method.

   Insert the following code below the **getNameVector** method:

   Note
   You must replace *your-host-name* with your computer name. You cannot use this term in your application.

   ```
   public boolean sync() {
       try {
           if(_syncParms == null){
               _syncParms = _conn.createSyncParms(SyncParms.HTTP_STREAM,
                                   "mluser",
                                   "HelloBlackBerrySyncModel");
               _syncParms.setPassword("mlpassword");
               _syncParms.getStreamParms().setHost("your-host-name"); //
   USE YOUR OWN
               _syncParms.getStreamParms().setPort(8081); // USE YOUR OWN
           }
           _conn.synchronize(_syncParms);
           return true;
       }
       catch(ULjException uex) {
           Dialog.alert("Exception: " + uex.toString());
           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 can refer to the synchronization version or a set of synchronization logic that was deployed to your consolidated database.

   The stream parameters object, **StreamHTTPParms**, indicates 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.

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

6. Click **File** » **Save All**.

Lesson 4: Starting the MobiLink server and synchronizing the application

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 communication channel between the device simulator and MobiLink.

Start the MobiLink server and synchronize the application

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

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

   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 MobiLink server messages window. The -x option indicates the port number being used for the communications. The -ot option specifies that a log file (*ml.mls*) is to be created in the directory where you started the MobiLink server.

2. Run the MDS simulator so that the BlackBerry simulator can communicate over a network.

   Click **Start** » **Programs** » **Research In Motion** » **BlackBerry Email And MDS Services Simulator 4.1.4** » **MDS**.

3. Add names to the MobiLink consolidated database so that your application updates the UltraLite Java edition database when synchronizing.

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

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

   b. Execute the following SQL statement in Interactive SQL to add names to the **Names** table:

      ```
      INSERT Names (Name) VALUES ('Jane Smith');
      INSERT Names (Name) VALUES ('David Smith');
      COMMIT;
      ```

   c. Close Interactive SQL.

      Click **File** » **Exit**.

4. Run the simulator from Eclipse.

   In the **Package Explorer** window, click *Application.java*, and then click **Run** » **Run As** » **BlackBerry Simulator**.

> **Note**
> If multiple projects are open in your workspace, click **Run** » **Run Configurations**, select **HelloBlackBerry**, and then click **Run**.

The **HelloBlackBerry** project compiles and then the simulator window appears.

Ensure that the project compiles without errors by selecting the **Problems** tab in Eclipse.

5. From the simulator menu, click **File** » **Load Java Program**.
6. Browse to the *\UltraLite\UltraLiteJ\BlackBerry4.2* directory of your SQL Anywhere installation and open the *UltraLiteJ12.cod* file.

> **Note**
> You may need to copy *UltraLiteJ12.cod* and the DBG files to the working simulator directory (for example, *C:\Eclipse\plugins\net.rim.ejde.componentpack6.0.0_6.0.0.0.26\components\simulator*) to run the application. When copied, you do not need to load the Java program from the simulator menu.

7. From the simulator menu, click **Simulate** » **Set IT Policy**.

   The **Set IT Policy** window appears.
8. In the **Policy** field, click **Allow Third Party Apps To Use Persistent Store** » >>.
9. Click **Set** and then click **Close**.
10. Launch your application.

    In the simulator window, navigate to **Downloads** and then run the **HelloBlackBerry** application.

    A screen appears that displays the **Hello BlackBerry** title bar, the **Status: Connected** text, and a **Name** field.
11. Synchronize the application with the MobiLink server.

    Click ***EMPTY*** and then choose **Sync**.

    Jane Smith and **David Smith** appear in the list, which indicates that the application was able to synchronize with the MobiLink consolidated database. 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.
12. Stop the simulation.

    In the simulator window, click **File** » **Exit**.

Cleaning up

Remove tutorial materials from your computer.

Remove tutorial materials from your computer

1. Close Eclipse.

   Click **File** » **Exit**.

2. Close the command window that is running the MDS simulator.

3. Close Sybase Central and Interactive SQL by right-clicking each task bar item and choosing **Exit**.

4. Delete the **HelloBlackBerry** data source:

   a. Start the ODBC Data Source Administrator.

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

   b. Select **HelloBlackBerry** from the list of **User Data Sources**, and click **Remove**.

   c. Close the ODBC Data Source Administrator.

5. Delete the *C:\HelloBlackBerry* directory.

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 *%SQLANYSAMP12%\UltraLiteJ\BlackBerry\HelloBlackBerry\myapp*.

See also

- "Part 1: Creating a new BlackBerry application" on page 41
- "Part 2: Using MobiLink to synchronize the BlackBerry application" on page 54

Application.java

```
// *************************************************
// Copyright (c) 2006-2011 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2011 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 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

```
// ****************************************************
// Copyright (c) 2006-2011 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2011 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 myapp;

import com.ianywhere.ultralitej12.*;

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

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 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();
            ps.close();
        }
        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());
        }
        return nameVector;
    }

    public boolean sync() {
        try {
            if(_syncParms == null){
                _syncParms = _conn.createSyncParms(SyncParms.HTTP_STREAM,
```

```
                                "mluser",
                                "HelloBlackBerrySyncModel");
                _syncParms.setPassword("mlpassword");
                _syncParms.getStreamParms().setHost("your-host-name"); //
USE YOUR OWN
                _syncParms.getStreamParms().setPort(8081); // USE YOUR OWN
            }
            _conn.synchronize(_syncParms);
            return true;
        }
        catch(ULjException uex) {
            Dialog.alert("Exception: " + uex.toString());
            return false;
        }
    }

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

HomeScreen.java

```
// ************************************************
// Copyright (c) 2006-2011 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2011 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 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);

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

    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();
        }
    };

    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);
        }
    }

    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");
```

```
            }
            refreshNameList();
        } catch (Exception ex) {
            Dialog.alert(ex.toString());
        }
    }
}
```

NameRow.java

```
// ****************************************************
// Copyright (c) 2006-2011 iAnywhere Solutions, Inc.
// Portions copyright (c) 2006-2011 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 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;

}
```


UltraLiteJ API reference

Package [Android]

```
com.ianywhere.ultralitejni12
```

Package [BlackBerry]

```
com.ianywhere.ultralitej12
```

See also

- "Android and BlackBerry setup considerations" on page 4

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_AUTOFILE-NAME 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_CUR-RENT_DATE variable	Indicates the CURRENT DATE (year, month, and day) column default attribute.
COLUMN_DEFAULT_CUR-RENT_TIME variable	Indicates the current time column default attribute.
COLUMN_DEFAULT_CUR-RENT_TIMESTAMP variable	Indicates the current timestamp column default attribute.
COLUMN_DEFAULT_CUR-RENT_UTC_TIMESTAMP variable	Indicates the current utc timestamp column default attribute.

Name	Description
COLUMN_DEFAULT_GLOBAL_AUTOINC variable	Indicates the global autoincrement column default attribute.
COLUMN_DEFAULT_NONE variable	Indicates that the column has no default attribute.
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 syscolumn system table.

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 column_default_value column in the TableSchema.SYS_COLUMN system table contains 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 column_default column in the TableSchema.SYS_COLUMNS system table.

See also

- "TableSchema.SYS_COLUMNS variable [UltraLiteJ]" on page 255

COLUMN_DEFAULT_AUTOINC variable

Indicates the autoincrement column default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_AUTOINC
```


Remarks

When using the AUTOINCREMENT attribute, 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 the AUTOINCREMENT attribute 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 column_default column in the TableSchema.SYS_COLUMNS system table.

See also

- "TableSchema.SYS_COLUMNS variable [UltraLiteJ]" on page 255

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 column_default column in the TableSchema.SYS_COLUMNS system table.

See also

- "TableSchema.SYS_COLUMNS variable [UltraLiteJ]" on page 255

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 column_default column in the TableSchema.SYS_COLUMNS system table.

See also

- "TableSchema.SYS_COLUMNS variable [UltraLiteJ]" on page 255

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 column_default column in the TableSchema.SYS_COLUMNS system table.

See also

- "TableSchema.SYS_COLUMNS variable [UltraLiteJ]" on page 255

COLUMN_DEFAULT_CURRENT_TIMESTAMP variable

Indicates the current timestamp column default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_CURRENT_TIMESTAMP
```

Remarks

This constant combines the CURRENT DATE and CURRENT TIME values 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 column_default column in the TableSchema.SYS_COLUMNS system table.

See also

- "TableSchema.SYS_COLUMNS variable [UltraLiteJ]" on page 255

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 the CURRENT DATE and CURRENT TIME values 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 column_default column in the TableSchema.SYS_COLUMNS system table.

See also

- “TableSchema.SYS_COLUMNS variable [UltraLiteJ]” on page 255

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 the AUTOINCREMENT attribute, 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 column_default column in the TableSchema.SYS_COLUMNS system table.

See also

- “TableSchema.SYS_COLUMNS variable [UltraLiteJ]” on page 255
- “Connection.setDatabaseId method [UltraLiteJ]” on page 113

COLUMN_DEFAULT_NONE variable

Indicates that the column has no default attribute.

Syntax

```
final byte ColumnSchema.COLUMN_DEFAULT_NONE
```

Remarks

The following defaults apply when no default attribute is assigned:

- Nullable columns default to null
- Not nullable numeric columns default to zero
- Not nullable varying length columns default to zero length values.

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

See also

- "TableSchema.SYS_COLUMNS variable [UltraLiteJ]" on page 255

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 column_default column in the TableSchema.SYS_COLUMNS system table.

See also

- "TableSchema.SYS_COLUMNS variable [UltraLiteJ]" on page 255

ConfigFile interface

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

Syntax

```
public interface ConfigFile
```

Base classes

- "ConfigPersistent interface [UltraLiteJ]" on page 82

Derived classes

- "ConfigFileAndroid interface [Android] [UltraLiteJ]" on page 75

Members

All members of ConfigFile interface, including all inherited members.

Name	Description
enableAesDBEncryption method [Android]	Enables AES encryption of the database.
enableObfuscation method [Android]	Enables obfuscation of the database.
getAutoCheckpoint method (deprecated)	Determines if auto checkpoint is turned on.
getCacheSize method	Returns the cache size of the database, in bytes.
getConnectionString method [Android]	Gets the connection string registered with the setConnectionString method.
getCreationString method [Android]	Gets the creation string registered with the setCreationString method.
getDatabaseKey method [Android]	Gets the database encryption key registered with the setDatabaseKey method.
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 [BlackBerry]	Returns the current row score flush size.
getRowScoreMaximum method [BlackBerry]	Returns the current row score maximum size.
getUserName method [Android]	Gets the name of user set by the setUserName method.
hasPersistentIndexes method	Determines if indexes are persistent.

Name	Description
hasShadowPaging method	Determines if shadow paging is turned on.
setAutocheckpoint method (deprecated)	Sets auto checkpoint on.
setCacheSize method	Sets the cache size of the database, in bytes.
setConnectionString method [Android]	Sets the connection string to be used to create or connect to a database.
setCreationString method [Android]	Sets the creation string to be used to create a database.
setDatabaseKey method [Android]	Sets the key for encryption.
setDatabaseName method	Sets the database name.
setEncryption method [BlackBerry]	Sets the encryption control.
setIndexPersistence method [BlackBerry]	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.
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 or off.
setUserName method [Android]	Sets the name of the user.
setWriteAtEnd method	Sets database persistence to occur during shutdown only.
writeAtEnd method	Determines if the database uses write-at-end persistence.

Remarks

An object implementing the ConfigFile interface is created by using the DatabaseManager.createConfigurationFile method.

ConfigFileAndroid interface [Android]

Establishes a Configuration object for a persistent database saved in a file on an Android device.

Syntax

```
public interface ConfigFileAndroid
```

Base classes

- "ConfigFile interface [UltraLiteJ]" on page 72

Members

All members of ConfigFileAndroid interface, including all inherited members.

Name	Description
enableAesDBEncryption method [Android]	Enables AES encryption of the database.
enableObfuscation method [Android]	Enables obfuscation of the database.
getAutoCheckpoint method (deprecated)	Determines if auto checkpoint is turned on.
getCacheSize method	Returns the cache size of the database, in bytes.
getConnectionString method [Android]	Gets the connection string registered with the setConnectionString method.
getCreationString method [Android]	Gets the creation string registered with the setCreationString method.
getDatabaseKey method [Android]	Gets the database encryption key registered with the setDatabaseKey method.
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 [BlackBerry]	Returns the current row score flush size.
getRowScoreMaximum method [BlackBerry]	Returns the current row score maximum size.
getUserName method [Android]	Gets the name of user set by the setUserName method.

Name	Description
hasPersistentIndexes method	Determines if indexes are persistent.
hasShadowPaging method	Determines if shadow paging is turned on.
setAutocheckpoint method (deprecated)	Sets auto checkpoint on.
setCacheSize method	Sets the cache size of the database, in bytes.
setConnectionString method [Android]	Sets the connection string to be used to create or connect to a database.
setCreationString method [Android]	Sets the creation string to be used to create a database.
setDatabaseKey method [Android]	Sets the key for encryption.
setDatabaseName method	Sets the database name.
setEncryption method [BlackBerry]	Sets the encryption control.
setIndexPersistence method [BlackBerry]	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.
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 or off.
setUserName method [Android]	Sets the name of the user.
setWriteAtEnd method	Sets database persistence to occur during shutdown only.
writeAtEnd method	Determines if the database uses write-at-end persistence.

Remarks

An object implementing the ConfigFileAndroid interface is created by using the DatabaseManager.createConfigurationFileAndroid method.

See also

- "DatabaseManager class [UltraLiteJ]" on page 124
- "DatabaseManager.createConfigurationFileAndroid method [UltraLiteJ]" on page 127

ConfigFileME interface [BlackBerry]

Establishes a Configuration object for a persistent database saved in a file on a J2ME device file system.

Syntax

```
public interface ConfigFileME
```

Base classes

- "ConfigPersistent interface [UltraLiteJ]" on page 82

Members

All members of ConfigFileME interface, including all inherited members.

Name	Description
enableAesDBEncryption method [Android]	Enables AES encryption of the database.
enableObfuscation method [Android]	Enables obfuscation of the database.
getAutoCheckpoint method (deprecated)	Determines if auto checkpoint is turned on.
getCacheSize method	Returns the cache size of the database, in bytes.
getConnectionString method [Android]	Gets the connection string registered with the setConnectionString method.
getCreationString method [Android]	Gets the creation string registered with the setCreationString method.
getDatabaseKey method [Android]	Gets the database encryption key registered with the setDatabaseKey method.
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 [BlackBerry]	Returns the current row score flush size.

Name	Description
getRowScoreMaximum method [BlackBerry]	Returns the current row score maximum size.
getUserName method [Android]	Gets the name of user set by the setUserName method.
hasPersistentIndexes method	Determines if indexes are persistent.
hasShadowPaging method	Determines if shadow paging is turned on.
setAutocheckpoint method (deprecated)	Sets auto checkpoint on.
setCacheSize method	Sets the cache size of the database, in bytes.
setConnectionString method [Android]	Sets the connection string to be used to create or connect to a database.
setCreationString method [Android]	Sets the creation string to be used to create a database.
setDatabaseKey method [Android]	Sets the key for encryption.
setDatabaseName method	Sets the database name.
setEncryption method [BlackBerry]	Sets the encryption control.
setIndexPersistence method [BlackBerry]	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.
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 or off.
setUserName method [Android]	Sets the name of the user.
setWriteAtEnd method	Sets database persistence to occur during shutdown only.
writeAtEnd method	Determines if the database uses write-at-end persistence.

Remarks

An object implementing the ConfigFileME interface is created by using the DatabaseManager.createConfigurationFileME method.

See also

- "DatabaseManager class [UltraLiteJ]" on page 124
- "DatabaseManager.createConfigurationFileME method [BlackBerry] [UltraLiteJ]" on page 127

ConfigNonPersistent interface [BlackBerry]

Establishes a Configuration object for a non-persistent (in-memory) database.

Syntax

```
public interface ConfigNonPersistent
```

Base classes

- "Configuration interface [UltraLiteJ]" on page 97

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.
setPassword method	Sets the database password.

Remarks

An object implementing the ConfigNonPersistent interface is created by using the DatabaseManager.createConfigurationNonPersistent method.

Creating a NonPersistent 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.

See also

- "DatabaseManager class [UltraLiteJ]" on page 124

ConfigObjectStore interface [BlackBerry]

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

Syntax

```
public interface ConfigObjectStore
```

Base classes

- "ConfigPersistent interface [UltraLiteJ]" on page 82

Members

All members of ConfigObjectStore interface, including all inherited members.

Name	Description
enableAesDBEncryption method [Android]	Enables AES encryption of the database.
enableObfuscation method [Android]	Enables obfuscation of the database.
getAutoCheckpoint method (deprecated)	Determines if auto checkpoint is turned on.
getCacheSize method	Returns the cache size of the database, in bytes.
getConnectionString method [Android]	Gets the connection string registered with the setConnectionString method.
getCreationString method [Android]	Gets the creation string registered with the setCreationString method.
getDatabaseKey method [Android]	Gets the database encryption key registered with the setDatabaseKey method.
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 [BlackBerry]	Returns the current row score flush size.
getRowScoreMaximum method [BlackBerry]	Returns the current row score maximum size.
getUserName method [Android]	Gets the name of user set by the setUserName method.

Name	Description
hasPersistentIndexes method	Determines if indexes are persistent.
hasShadowPaging method	Determines if shadow paging is turned on.
setAutocheckpoint method (deprecated)	Sets auto checkpoint on.
setCacheSize method	Sets the cache size of the database, in bytes.
setConnectionString method [Android]	Sets the connection string to be used to create or connect to a database.
setCreationString method [Android]	Sets the creation string to be used to create a database.
setDatabaseKey method [Android]	Sets the key for encryption.
setDatabaseName method	Sets the database name.
setEncryption method [BlackBerry]	Sets the encryption control.
setIndexPersistence method [BlackBerry]	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.
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 or off.
setUserName method [Android]	Sets the name of the user.
setWriteAtEnd method	Sets database persistence to occur during shutdown only.
writeAtEnd method	Determines if the database uses write-at-end persistence.

Remarks

An object implementing the ConfigObjectStore interface is created by using the DatabaseManager.createConfigurationObjectStore method.

See also

- "DatabaseManager class [UltraLiteJ]" on page 124
- "DatabaseManager.createConfigurationObjectStore method [BlackBerry] [UltraLiteJ]" on page 128

ConfigPersistent interface

Establishes a Configuration object for a persistent database.

Syntax

```
public interface ConfigPersistent
```

Base classes

- "Configuration interface [UltraLiteJ]" on page 97

Derived classes

- "ConfigFile interface [UltraLiteJ]" on page 72
- "ConfigFileME interface [BlackBerry] [UltraLiteJ]" on page 77
- "ConfigObjectStore interface [BlackBerry] [UltraLiteJ]" on page 80
- "ConfigRecordStore interface (J2ME only) [UltraLiteJ]" on page 95

Members

All members of ConfigPersistent interface, including all inherited members.

Name	Description
enableAesDBEncryption method [Android]	Enables AES encryption of the database.
enableObfuscation method [Android]	Enables obfuscation of the database.
getAutoCheckpoint method (deprecated)	Determines if auto checkpoint is turned on.
getCacheSize method	Returns the cache size of the database, in bytes.
getConnectionString method [Android]	Gets the connection string registered with the setConnectionString method.
getCreationString method [Android]	Gets the creation string registered with the setCreationString method.
getDatabaseKey method [Android]	Gets the database encryption key registered with the setDatabaseKey method.
getDatabaseName method	Returns the database name.

Name	Description
getLazyLoadIndexes method	Determines if lazy loading indexes is turned on.
getPageSize method	Returns the page size of the database, in bytes.
getRowScoreFlushSize method [BlackBerry]	Returns the current row score flush size.
getRowScoreMaximum method [BlackBerry]	Returns the current row score maximum size.
getUserName method [Android]	Gets the name of user set by the setUserName method.
hasPersistentIndexes method	Determines if indexes are persistent.
hasShadowPaging method	Determines if shadow paging is turned on.
setAutocheckpoint method (deprecated)	Sets auto checkpoint on.
setCacheSize method	Sets the cache size of the database, in bytes.
setConnectionString method [Android]	Sets the connection string to be used to create or connect to a database.
setCreationString method [Android]	Sets the creation string to be used to create a database.
setDatabaseKey method [Android]	Sets the key for encryption.
setDatabaseName method	Sets the database name.
setEncryption method [BlackBerry]	Sets the encryption control.
setIndexPersistence method [BlackBerry]	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.
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 or off.

Name	Description
setUserName method [Android]	Sets the name of the user.
setWriteAtEnd method	Sets database persistence to occur during shutdown only.
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.

enableAesDBEncryption method [Android]

Enables AES encryption of the database.

Syntax

```
void ConfigPersistent.enableAesDBEncryption()
```

enableObfuscation method [Android]

Enables obfuscation of the database.

Syntax

```
void ConfigPersistent.enableObfuscation()
```

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. Determines if auto checkpoint is turned on.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

- "ConfigPersistent.setCacheSize method [UltraLiteJ]" on page 88

getConnectionString method [Android]

Gets the connection string registered with the setConnectionString method.

Syntax

```
String ConfigPersistent.getConnectionString()
```

Returns

The connection string registered with the setConnectionString method.

See also

- "ConfigPersistent.setConnectionString method [Android] [UltraLiteJ]" on page 89

getCreationString method [Android]

Gets the creation string registered with the setCreationString method.

Syntax

```
String ConfigPersistent.getCreationString()
```

Returns

The creation string registered with the setCreationString method.

See also

- "ConfigPersistent.setCreationString method [Android] [UltraLiteJ]" on page 89

getDatabaseKey method [Android]

Gets the database encryption key registered with the setDatabaseKey method.

Syntax

```
String ConfigPersistent.getDatabaseKey()
```

Returns

The database encryption key registered with the setDatabaseKey method.

See also

- "ConfigPersistent.setDatabaseKey method [Android] [UltraLiteJ]" on page 90

getLazyLoadIndexes method

Determines if lazy loading indexes is turned on.

Syntax

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

Returns

True if lazy loading is turned on; otherwise, returns false.

See also

- "ConfigPersistent.setLazyLoadIndexes method [UltraLiteJ]" on page 91

getRowScoreFlushSize method [BlackBerry]

Returns the current row score flush size.

Syntax

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

Returns

The current row score flush size.

See also

- "ConfigPersistent.setRowScoreFlushSize method [UltraLiteJ]" on page 91

getRowScoreMaximum method [BlackBerry]

Returns the current row score maximum size.

Syntax

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

Returns

The current row score maximum size.

See also

- "ConfigPersistent.setRowScoreMaximum method [UltraLiteJ]" on page 92

getUserName method [Android]

Gets the name of user set by the setUserName method.

Syntax

```
String ConfigPersistent.getUserName()
```

Returns

The name of user set by the setUserName method.

See also

- "ConfigPersistent.setUserName method [Android] [UltraLiteJ]" on page 93

hasPersistentIndexes method

Determines if indexes are persistent.

Syntax

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

Returns

True if indexes are persistent; otherwise, returns false.

See also

- "ConfigPersistent.setIndexPersistence method [BlackBerry] [UltraLiteJ]" on page 90

hasShadowPaging method

Determines if shadow paging is turned on.

Syntax

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

Returns

True if shadow paging is turned on; otherwise, returns false.

See also

- "ConfigPersistent.setShadowPaging method [UltraLiteJ]" on page 93

setAutocheckpoint method (deprecated)

Sets auto checkpoint on.

Syntax

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

Parameters

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

Parameters

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

Returns

This ConfigPersistent, with the auto checkpoint specified, Sets auto checkpoint 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 object 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

- "ConfigPersistent.getCacheSize method [UltraLiteJ]" on page 85

setConnectionString method [Android]

Sets the connection string to be used to create or connect to a database.

Syntax

```
void ConfigPersistent.setConnectionString(String connection_string)
```

Parameters

- **connection_string** The connection string used in database connection or creation.

Remarks

Any other items that have been set in this configuration are also passed to create or connect to a database.

setCreationString method [Android]

Sets the creation string to be used to create a database.

Syntax

```
void ConfigPersistent.setCreationString(String creation_string)
```

Parameters

- **creation_string** The creation string used in database creation.

Remarks

Any other items that have been set in this configuration are also passed to create a database.

setDatabaseKey method [Android]

Sets the key for encryption.

Syntax

```
void ConfigPersistent.setDatabaseKey(String encryption_key)
```

Parameters

- **encryption_key** The string to use for encryption.

setEncryption method [BlackBerry]

Sets the encryption control.

Syntax

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

Parameters

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

Returns

This ConfigPersistent object with the encryption specified.

Remarks

Set this method with an EncryptionControl object to encrypt and decrypt pages.

> **Note**
> You must supply your own encryption method.

setIndexPersistence method [BlackBerry]

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 object with the index persistence specified.

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 object 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

- “ConfigPersistent.getLazyLoadIndexes method [UltraLiteJ]” on page 86
- “ConfigPersistent.setRowScoreFlushSize method [UltraLiteJ]” on page 91

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, which indicates no row limiting.

Returns

This ConfigPersistent object 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 interruptions.

Databases accessed with row limiting enabled always lazy load indexes.

See also

- "ConfigPersistent.setRowScoreMaximum method [UltraLiteJ]" on page 92
- "ConfigPersistent.setLazyLoadIndexes method [UltraLiteJ]" on page 91

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 object with the maximum threshold value specified.

Remarks

Databases accessed with row limiting enabled always lazy load indexes.

See also

- "ConfigPersistent.setRowScoreFlushSize method [UltraLiteJ]" on page 91
- "ConfigPersistent.setLazyLoadIndexes method [UltraLiteJ]" on page 91

setShadowPaging method

Sets shadow paging on or off.

Syntax

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

Parameters

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

Returns

This ConfigPersistent object with ShadowPaging specified.

Remarks

By default, shadow paging is on for persistent database stores.

Shadow paging is the strongest form of persistence and is used for most applications. 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.

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

- "ConfigPersistent.hasShadowPaging method [UltraLiteJ]" on page 88

setUserName method [Android]

Sets the name of the user.

Syntax

```
void ConfigPersistent.setUserName(String user_name)
```

Parameters

- **user_name** The name of the user.

Remarks

This name is used to connect or create to the native database with the UID= phrase in the connection string.

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 to true to retain the database in memory until it is shutdown.

Returns

This ConfigPersistent object 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

- "ConfigPersistent.writeAtEnd method [UltraLiteJ]" on page 94
- "ConfigPersistent.setShadowPaging method [UltraLiteJ]" on page 93

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, returns false.

See also

- "ConfigPersistent.setWriteAtEnd method [UltraLiteJ]" on page 94

ConfigRecordStore interface (J2ME only)

Establishes the Configuration for a persistent database saved in a J2ME record store.

Syntax

```
public interface ConfigRecordStore
```

Base classes

- "ConfigPersistent interface [UltraLiteJ]" on page 82

Members

All members of ConfigRecordStore interface, including all inherited members.

Name	Description
enableAesDBEncryption method [Android]	Enables AES encryption of the database.
enableObfuscation method [Android]	Enables obfuscation of the database.
getAutoCheckpoint method (deprecated)	Determines if auto checkpoint is turned on.
getCacheSize method	Returns the cache size of the database, in bytes.
getConnectionString method [Android]	Gets the connection string registered with the setConnectionString method.
getCreationString method [Android]	Gets the creation string registered with the setCreationString method.
getDatabaseKey method [Android]	Gets the database encryption key registered with the setDatabaseKey method.
getDatabaseName method	Returns the database name.
getLazyLoadIndexes method	Determines if lazy loading indexes is turned on.

Name	Description
getPageSize method	Returns the page size of the database, in bytes.
getRowScoreFlushSize method [BlackBerry]	Returns the current row score flush size.
getRowScoreMaximum method [BlackBerry]	Returns the current row score maximum size.
getUserName method [Android]	Gets the name of user set by the setUserName method.
hasPersistentIndexes method	Determines if indexes are persistent.
hasShadowPaging method	Determines if shadow paging is turned on.
setAutocheckpoint method (deprecated)	Sets auto checkpoint on.
setCacheSize method	Sets the cache size of the database, in bytes.
setConnectionString method [Android]	Sets the connection string to be used to create or connect to a database.
setCreationString method [Android]	Sets the creation string to be used to create a database.
setDatabaseKey method [Android]	Sets the key for encryption.
setDatabaseName method	Sets the database name.
setEncryption method [BlackBerry]	Sets the encryption control.
setIndexPersistence method [BlackBerry]	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.
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 or off.
setUserName method [Android]	Sets the name of the user.

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

Configuration interface

Establishes a Configuration object for a database.

Syntax

```
public interface Configuration
```

Derived classes

- “ConfigNonPersistent interface [BlackBerry] [UltraLiteJ]” on page 79
- “ConfigPersistent interface [UltraLiteJ]” on page 82

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.
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 object with the database name specified.

setPageSize method

Sets the page size of the database.

Syntax

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

Parameters

- **page_size** The page size, in bytes.

Returns

This Configuration object 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.

For Android smartphones, the page size can be 1024, 2048, 4096, 8192, or 16384 bytes. The default is 4096 bytes.

For BlackBerry smartphones, the page size can range from 256 to 16384 bytes. The default is 1024 bytes. The 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 object 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 set of synchronization parameters.
createUUIDValue method	Creates a UUID value.
dropDatabase method	Drops a database.
emergencyShutdown method [BlackBerry]	Performs an emergency shut down of a connected database.
getDatabaseId method [BlackBerry]	Returns the value of database ID.
getDatabaseInfo method	Returns a DataInfo object containing information on database properties.
getDatabaseProperty method	Returns a database property.
getLastDownloadTime method	Returns the time of the most recent download of the specified publication.
getLastIdentity method	Retrieves the most recent value inserted into a DEFAULT AUTOINCREMENT or DEFAULT GLOBAL AUTOINCREMENT column, or zero if the most recent INSERT transaction was made on a table that had no such column.
getOption method [BlackBerry]	Returns a database option.
getState method [BlackBerry]	Returns the state of the connection.
getSyncObserver method	Returns the SyncObserver object that is currently registered for this Connection object.
getSyncResult method	Returns the result of the last SYNCHRONIZE SQL statement.
isSynchronizationDeleteDisabled method [BlackBerry]	Determine if the synchronization of deletes is disabled.
prepareStatement method	Prepares a statement for execution.

Name	Description
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	Denotes a connected state.
NOT_CONNECTED variable	Denotes a not connected state.
OPTION_BLOB_FILE_BASE_DIR variable [BlackBerry]	Database option: blob file base dir.
OPTION_DATABASE_ID variable [BlackBerry]	Database option: database id.
OPTION_DATE_FORMAT variable [BlackBerry]	Database option: date format.
OPTION_DATE_ORDER variable [BlackBerry]	Database option: date order.
OPTION_ML_REMOTE_ID variable [BlackBerry]	Database option: ML remote ID.
OPTION_NEAREST_CENTURY variable [BlackBerry]	Database option: nearest century.
OPTION_PRECISION variable [BlackBerry]	Database option: precision.
OPTION_SCALE variable [BlackBerry]	Database option: scale.
OPTION_TIME_FORMAT variable [BlackBerry]	Database option: time format.
OPTION_TIMESTAMP_FORMAT variable [BlackBerry]	Database option: timestamp format.

Name	Description
OPTION_TIMESTAMP_INCREMENT variable [BlackBerry]	Database option: timestamp increment.
OPTION_TIME-STAMP_WITH_TIME_ZONE_FORMAT variable [BlackBerry]	Database option: timestamp with time zone format.
PROPERTY_DATABASE_NAME variable [Black-Berry]	Database Property: database name.
PROPERTY_PAGE_SIZE variable [BlackBerry]	Database Property: page size.
SYNC_ALL variable	The publication list used to request synchroni-zation 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 synchroni-zation 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 object 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 uncommitted changes to the database

The following example demonstrates how to create a schema for a simple database with a Connection object, 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 [UltraLiteJ]" on page 124
- "DatabaseManager.createDatabase method [UltraLiteJ]" on page 129
- "DatabaseManager.connect method [UltraLiteJ]" on page 126
- "Connection.release method [UltraLiteJ]" on page 112

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. Checkpoints the database changes.

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 object.
createDecimalNumber(int, int, String) method	Creates a DecimalNumber object.

createDecimalNumber(int, int) method

Creates a DecimalNumber object.

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 object with the specified type.

See also

- "DecimalNumber interface [UltraLiteJ]" on page 133

createDecimalNumber(int, int, String) method

Creates a DecimalNumber object.

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 object with the specified type.

See also

- "DecimalNumber interface [UltraLiteJ]" on page 133

createSyncParms method

Creates a set of synchronization parameters.

Overload list

Name	Description
createSyncParms(int, String, String) method	Creates a set of synchronization parameters for HTTP synchronization.
createSyncParms(String, String) method	Creates a set of synchronization parameters for HTTP synchronization.

createSyncParms(int, String, String) method

Creates a set of synchronization parameters for HTTP synchronization.

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

A SyncParms object.

See also

- "Connection.createSyncParms method [UltraLiteJ]" on page 105
- "SyncParms.HTTP_STREAM variable [UltraLiteJ]" on page 247
- "SyncParms.HTTPS_STREAM variable [UltraLiteJ]" on page 247

createSyncParms(String, String) method

Creates a set of synchronization parameters for HTTP synchronization.

Syntax

```
SyncParms Connection.createSyncParms(
    String userName,
    String version
) throws ULjException
```

Parameters

- **userName** The unique MobiLink user name for this client database.
- **version** The MobiLink script version.

Returns

The SyncParms object.

See also

- "Connection.createSyncParms method [UltraLiteJ]" on page 105
- "SyncParms.setUserName method [UltraLiteJ]" on page 246

createUUIDValue method

Creates a UUID value.

Syntax

```
UUIDValue Connection.createUUIDValue() throws ULjException
```

Returns

The UUIDValue instance 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. This connection is the only one that can be active for the database being dropped.

emergencyShutdown method [BlackBerry]

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 [BlackBerry]

Returns the value of database ID.

Syntax

```
int Connection.getDatabaseId() throws ULjException
```

Returns

The database ID.

Exceptions

- **ULjException class** when the database ID is not set.

See also

- "Connection.getLastIdentity method [UltraLiteJ]" on page 109

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 database property.

Syntax

```
String Connection.getDatabaseProperty(String name) throws ULjException
```

Parameters

- **name** The name of the database property. For Android devices, you can set this parameter to any supported UltraLite database property name. For BlackBerry devices, you can set this parameter to any constant in the Connection interface that has a **PROPERTY_** prefix.

Returns

The value of the property that corresponds to the given name.

See also

- "Connection.PROPERTY_DATABASE_NAME variable [BlackBerry] [UltraLiteJ]" on page 119
- "Connection.PROPERTY_PAGE_SIZE variable [BlackBerry] [UltraLiteJ]" on page 119

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. This parameter must reference a single publication or be the special Connection.SYNC_ALL_DB_PUB_NAME publication for the time of the last download of the full database.

Returns

The timestamp of the last download.

See also

- "Connection.SYNC_ALL_DB_PUB_NAME variable [UltraLiteJ]" on page 120
- "Connection.resetLastDownloadTime method [UltraLiteJ]" on page 112

getLastIdentity method

Retrieves the most recent value inserted into a DEFAULT AUTOINCREMENT or DEFAULT GLOBAL AUTOINCREMENT column, or zero if the most recent INSERT transaction was made on a table that had no such column.

Syntax

```
long Connection.getLastIdentity()
```

Returns

The most recent identity value.

Remarks

If a table contains more than one column of the (GLOBAL) AUTOINCREMENT type, then the column to which this value belongs to is undetermined.

getOption method [BlackBerry]

Returns a database option.

Syntax

```
String Connection.getOption(String option_name) throws ULjException
```

Parameters

- **option_name** The name of option to get.
- **option_name** The name of the option to get. You can set this parameter to any constant in the Connection interface that has an **OPTION_** prefix.

Returns

The value of the database option.

Remarks

Database options are stored within the database and may also be obtained when a database is connected at some time after the option has been set.

A set of required options are created when a database is created.

See also

- "Connection.setOption method [UltraLiteJ]" on page 113

getState method [BlackBerry]

Returns the state of the connection.

Syntax

```
byte Connection.getState() throws ULjException
```

Returns

The byte representing the state of the connection.

Remarks

The following example demonstrates how to check the connection state and release the connection.

```
if( _conn.getState() == Connection.CONNECTED ){
    _conn.release();
}
```

See also

- "Connection interface [UltraLiteJ]" on page 99

getSyncObserver method

Returns the SyncObserver object that is currently registered for this Connection object.

Syntax

```
SyncObserver Connection.getSyncObserver()
```

Returns

The SyncObserver, or null if no observer exists.

See also

- "Connection.setSyncObserver method [UltraLiteJ]" on page 114

getSyncResult method

Returns the result of the last SYNCHRONIZE SQL statement.

Syntax

```
SyncResult Connection.getSyncResult()
```

Returns

The SyncResult object representing the result of the last SYNCHRONIZE SQL statement.

Remarks

The following example illustrates how to get the result of the last SYNCHRONIZE SQL statement:

```
PreparedStatement ps = conn.prepareStatement("SYNCHRONIZE PROFILE
myprofile");
ps.execute();
ps.close();
SyncResult result = conn.getSyncResult();
display(
    "*** Synchronized *** sent=" + result.getSentRowCount()
    + ", received=" + result.getReceivedRowCount()
);
```

Note
This method does not return the result of the last call to the Connection.synchronize method. To obtain the SyncResult object for the last Connection.synchronize(SyncParms) method call, use the getSyncResult method on the SyncParms object passed in.

See also

- "SyncResult class [UltraLiteJ]" on page 247
- "SyncParms.getSyncResult method [UltraLiteJ]" on page 237

isSynchronizationDeleteDisabled method [BlackBerry]

Determine if the synchronization of deletes is disabled.

Syntax

```
boolean Connection.isSynchronizationDeleteDisabled()
```

Returns

True if and only if the 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 [UltraLiteJ]" on page 169

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 no uncommitted transactions are 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 object since the last 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 been set explicitly.

See also

- "Connection.getDatabaseId method [BlackBerry] [UltraLiteJ]" on page 107

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. For Android devices, you can set this parameter to any supported UltraLite database option name. For BlackBerry devices, you can set this parameter to any constant in the Connection interface that has an **OPTION_** prefix.
- **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

- "Connection.getOption method [BlackBerry] [UltraLiteJ]" on page 109
- "Connection.OPTION_BLOB_FILE_BASE_DIR variable [BlackBerry] [UltraLiteJ]" on page 115
- "Connection.OPTION_DATABASE_ID variable [BlackBerry] [UltraLiteJ]" on page 115
- "Connection.OPTION_DATE_FORMAT variable [BlackBerry] [UltraLiteJ]" on page 116
- "Connection.OPTION_DATE_ORDER variable [BlackBerry] [UltraLiteJ]" on page 116
- "Connection.OPTION_ML_REMOTE_ID variable [BlackBerry] [UltraLiteJ]" on page 116
- "Connection.OPTION_NEAREST_CENTURY variable [BlackBerry] [UltraLiteJ]" on page 117
- "Connection.OPTION_PRECISION variable [BlackBerry] [UltraLiteJ]" on page 117
- "Connection.OPTION_SCALE variable [BlackBerry] [UltraLiteJ]" on page 117
- "Connection.OPTION_TIME_FORMAT variable [BlackBerry] [UltraLiteJ]" on page 118
- "Connection.OPTION_TIMESTAMP_FORMAT variable [BlackBerry] [UltraLiteJ]" on page 118
- "Connection.OPTION_TIMESTAMP_INCREMENT variable [BlackBerry] [UltraLiteJ]" on page 118
- "Connection.OPTION_TIMESTAMP_WITH_TIME_ZONE_FORMAT variable [BlackBerry] [UltraLiteJ]" on page 119

setSyncObserver method

Sets a SyncObserver object to monitor the progress of synchronizations on this connection.

Syntax

```
void Connection.setSyncObserver(SyncObserver so)
```

Parameters

- **so** A SyncObserver object or null to remove the currently registered SyncObserver object.

Remarks

This SyncObserver object is used by subsequent SYNCHRONIZE SQL statements.

The default is null, suggesting no observer.

See also

- "SyncObserver interface [UltraLiteJ]" on page 225

synchronize method

Synchronizes the database with a MobiLink server.

Syntax

```
void Connection.synchronize(SyncParms config) throws ULjException
```


Parameters

- **config** The SyncParms object containing the parameters used for synchronization.

Remarks

The database is checkpointed when the download is applied to the database.

See also

- "SyncParms class [UltraLiteJ]" on page 232

CONNECTED variable

Denotes a connected state.

Syntax

```
final byte Connection.CONNECTED
```

NOT_CONNECTED variable

Denotes a not connected state.

Syntax

```
final byte Connection.NOT_CONNECTED
```

OPTION_BLOB_FILE_BASE_DIR variable [BlackBerry]

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

- "Connection.setOption method [UltraLiteJ]" on page 113

OPTION_DATABASE_ID variable [BlackBerry]

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

- "Connection.setOption method [UltraLiteJ]" on page 113

OPTION_DATE_FORMAT variable [BlackBerry]

Database option: date format.

Syntax

```
final String Connection.OPTION_DATE_FORMAT
```

Remarks

The default value is "YYYY-MM-DD".

See also

- "Connection.setOption method [UltraLiteJ]" on page 113

OPTION_DATE_ORDER variable [BlackBerry]

Database option: date order.

Syntax

```
final String Connection.OPTION_DATE_ORDER
```

Remarks

The default value is "YMD".

See also

- "Connection.setOption method [UltraLiteJ]" on page 113

OPTION_ML_REMOTE_ID variable [BlackBerry]

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

- "Connection.setOption method [UltraLiteJ]" on page 113

OPTION_NEAREST_CENTURY variable [BlackBerry]

Database option: nearest century.

Syntax

```
final String Connection.OPTION_NEAREST_CENTURY
```

Remarks

The default value is "50".

See also

- "Connection.setOption method [UltraLiteJ]" on page 113

OPTION_PRECISION variable [BlackBerry]

Database option: precision.

Syntax

```
final String Connection.OPTION_PRECISION
```

Remarks

The default value is "30".

See also

- "Connection.setOption method [UltraLiteJ]" on page 113

OPTION_SCALE variable [BlackBerry]

Database option: scale.

Syntax

```
final String Connection.OPTION_SCALE
```

Remarks

The default value is "6".

See also

- "Connection.setOption method [UltraLiteJ]" on page 113

OPTION_TIME_FORMAT variable [BlackBerry]

Database option: time format.

Syntax

```
final String Connection.OPTION_TIME_FORMAT
```

Remarks

The default value is "HH:NN:SS.SSS".

See also

- "Connection.setOption method [UltraLiteJ]" on page 113

OPTION_TIMESTAMP_FORMAT variable [BlackBerry]

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

- "Connection.setOption method [UltraLiteJ]" on page 113

OPTION_TIMESTAMP_INCREMENT variable [BlackBerry]

Database option: timestamp increment.

Syntax

```
final String Connection.OPTION_TIMESTAMP_INCREMENT
```

Remarks

The default value is "1".

See also

- "Connection.setOption method [UltraLiteJ]" on page 113

OPTION_TIMESTAMP_WITH_TIME_ZONE_FORMAT variable [BlackBerry]

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

- "Connection.setOption method [UltraLiteJ]" on page 113

PROPERTY_DATABASE_NAME variable [BlackBerry]

Database Property: database name.

Syntax

```
final String Connection.PROPERTY_DATABASE_NAME
```

See also

- "Connection.getDatabaseProperty method [UltraLiteJ]" on page 108

PROPERTY_PAGE_SIZE variable [BlackBerry]

Database Property: page size.

Syntax

```
final String Connection.PROPERTY_PAGE_SIZE
```

See also

- "Connection.getDatabaseProperty method [UltraLiteJ]" on page 108

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

- "Connection.getLastDownloadTime method [UltraLiteJ]" on page 108
- "Connection.resetLastDownloadTime method [UltraLiteJ]" on page 112

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 [BlackBerry]	Returns the total number of commit operations performed on the database.

Name	Description
getDbFormat method [BlackBerry]	Returns the database version number.
getDbSize method [BlackBerry]	Returns the database size.
getLogSize method [BlackBerry]	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 [UltraLiteJ]" on page 99
- "Connection.getDatabaseInfo method [UltraLiteJ]" on page 107

getCommitCount method [BlackBerry]

Returns the total number of commit operations performed on the database.

Syntax

```
int DatabaseInfo.getCommitCount()
```

Returns

The total number of commit operations.

getDbFormat method [BlackBerry]

Returns the database version number.

Syntax

```
int DatabaseInfo.getDbFormat()
```

Returns

The version number.

getDbSize method [BlackBerry]

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 [BlackBerry]

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 set.
createConfigurationFileME method [BlackBerry]	Creates a Configuration object for a physical database store from a file on a J2ME device file system, and returns a ConfigFileME object.
createConfigurationNonPersistent method [BlackBerry]	Creates a Configuration object for a non-persistent database store, and returns a ConfigNonPersist object.
createConfigurationObjectStore method [BlackBerry]	Creates a Configuration object for a RIM object store, and returns a ConfigObjectStore object.
createConfigurationRecordStore method [J2ME]	Creates a Configuration object for a record store as the physical database store, and returns a ConfigRecordStore object.
createDatabase method	Creates a new database based on a set of configurations, and connects to the database.
createFileTransfer method	Creates a FileTransfer object for transferring files to or from MobiLink.
createObjectStoreTransfer method [BlackBerry]	Creates a FileTransfer object for transferring UltraLite Java edition databases to or from MobiLink, and storing them into a RIM object store.
createSISHTTPListener method [BlackBerry]	Creates an SISListener object for server-initiated synchronizations.
release method	Closes this DatabaseManager object 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 or create a new one if it does not exist on the J2SE platform:

```
Connection conn;
ConfigFile config = DatabaseManager.createConfigurationFile(
        "test.ulj"
    );
try {
    conn = DatabaseManager.connect(config);
} catch(ULjException ex) {
    conn = DatabaseManager.createDatabase(config);
    // Create the schema here.
}
```

The following example demonstrates how to open an existing database or create a new one if it does not exist on a BlackBerry device:

```
Connection conn;
ConfigObjectStore config = DatabaseManager.createConfigurationObjectStore(
        "test.ulj"
    );
try {
    conn = DatabaseManager.connect(config);
} catch(ULjException ex) {
    conn = DatabaseManager.createDatabase(config);
    // Create the schema here.
}
```

The following example demonstrates how to open an existing database or create a new one if it does not exist on a BlackBerry media card:

```
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.
}
```

The following example demonstrates how to open an existing database or create a new one if it does not exist on an Android device:

```
// Android Sample
Connection conn;
ConfigFileAndroid config = DatabaseManager.createConfigurationFileAndroid(
        "test.udb",
        getApplicationContext());
try {
    conn = DatabaseManager.connect(config);
} catch(ULjException ex) {
    conn = DatabaseManager.createDatabase(config);
    // Create the schema here.
}
```

See also

- "Connection interface [UltraLiteJ]" on page 99
- "Configuration interface [UltraLiteJ]" on page 97

connect method

Connects to an existing database based on a configuration set.

Syntax

```
Connection DatabaseManager.connect(
    Configuration config
) throws ULjException
```

Parameters

- **config** The Configuration object containing the specifications for the existing database.

Returns

A Connection object that establishes the connection to the database.

See also

- "Configuration interface [UltraLiteJ]" on page 97
- "Connection interface [UltraLiteJ]" on page 99

createConfigurationFile method

Creates a Configuration object for a physical database store from a file and returns a ConfigFile object.

Syntax

```
ConfigFile DatabaseManager.createConfigurationFile(
    String file_name
) throws ULjException
```

Parameters

- **file_name** The name of file to use or create.

Returns

The ConfigFile object used to configure a database.

See also

- "ConfigFile interface [UltraLiteJ]" on page 72

createConfigurationFileAndroid method

Creates a Configuration object for a physical database store from a file on an Android device, and returns a ConfigFileAndroid object.

Syntax

```
ConfigFileAndroid DatabaseManager.createConfigurationFileAndroid(
    String file_name,
    android.content.Context context
) throws ULjException
```

Parameters

- **file_name** The name of the file to use or create.
- **context** The Context object from an Android application.

Returns

The ConfigFileAndroid object used to configure a database.

Remarks

You must use this method, supply a non-null Context object with this method under either of the following conditions:

- The file_name parameter is not an absolute path to the database file
- The application uses the HTTPS protocol for data synchronization

See also

- "ConfigFileAndroid interface [Android] [UltraLiteJ]" on page 75

createConfigurationFileME method [BlackBerry]

Creates a Configuration object for a physical database store from a file on a J2ME device file system, and returns a ConfigFileME object.

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 object used to configure the database.

Remarks

For the BlackBerry device, the store usually refers to an SD media card on the device 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 have the following restrictions:

- The host portion is case sensitive
- Percent characters have special meaning.
- Relative paths, such as "." and ".." directories, are not allowed.

For more information about file name restrictions, see the BlackBerry JDE API Reference for the optional **javax.microedition.io.file** package.

See also

- "ConfigFileME interface [BlackBerry] [UltraLiteJ]" on page 77

createConfigurationNonPersistent method [BlackBerry]

Creates a Configuration object for a non-persistent database store, and returns a ConfigNonPersist object.

Syntax

```
ConfigNonPersistent DatabaseManager.createConfigurationNonPersistent(
    String db_name
) throws ULjException
```

Parameters

- **db_name** The name of the non-persistent database.

Returns

The ConfigNonPersistent object used to configure the database.

See also

- "ConfigNonPersistent interface [BlackBerry] [UltraLiteJ]" on page 79

createConfigurationObjectStore method [BlackBerry]

Creates a Configuration object for a RIM object store, and returns a ConfigObjectStore object.

Syntax

```
ConfigObjectStore DatabaseManager.createConfigurationObjectStore(
    String db_name
) throws ULjException
```

Parameters

- **db_name** The name of the database.

Returns

The ConfigObjectStore object used to configure the database.

See also

- "ConfigObjectStore interface [BlackBerry] [UltraLiteJ]" on page 80

createConfigurationRecordStore method [J2ME]

Creates a Configuration object for a record store as the physical database store, and returns a ConfigRecordStore object.

Syntax

```
ConfigRecordStore DatabaseManager.createConfigurationRecordStore(
    String db_name
) throws ULjException
```

Parameters

- **db_name** The name of the database.

Returns

The ConfigRecordStore object used to configure the database.

See also

- "ConfigRecordStore interface (J2ME only) [UltraLiteJ]" on page 95

createDatabase method

Creates a new database based on a set of configurations, and connects to the database.

Syntax

```
Connection DatabaseManager.createDatabase(
    Configuration config
) throws ULjException
```

Parameters

- **config** A Configuration object containing the specifications for the new database.

Returns

A Connection object that establishes a connection the new database.

Remarks

This method replaces any existing database on the device that may have the same name.

See also

- "Configuration interface [UltraLiteJ]" on page 97
- "Connection interface [UltraLiteJ]" on page 99

createFileTransfer method

Creates a FileTransfer object for transferring 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. This parameter 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.

See also

- "SyncParms class [UltraLiteJ]" on page 232

createObjectStoreTransfer method [BlackBerry]

Creates a FileTransfer object for transferring UltraLite Java edition databases to or from MobiLink, and storing them into a 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. This parameter 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.

See also

- "SyncParms class [UltraLiteJ]" on page 232

createSISHTTPListener method [BlackBerry]

Creates an SISListener object for server-initiated synchronizations.

Syntax

```
SISListener DatabaseManager.createSISHTTPListener(
    SISRequestHandler handler,
    int port,
    String httpOptions
) throws ULjException
```

Parameters

- **handler** A SISRequestHandler object specified to handle server-initiated synchronization requests.
- **port** An HTTP port to listen for server messages.
- **httpOptions** The HTTP options for connecting to the server.

Returns

The SISListener object to use for server-initiated synchronizations.

Remarks

4400 is the recommended port setting.

"deviceside=false" is the recommended HTTP option for BlackBerry simulators.

The MobiLink side of the BlackBerry HTTP SISListener requires the target device be associated with a BlackBerry Enterprise Server, such as a BES activated device).

release method

Closes this DatabaseManager object to release all connections and to shutdown all databases.

Syntax

```
void DatabaseManager.release() throws ULjException
```

Remarks

On Android, this method releases all resources, and must be called only once by a single thread when the application is finished with the DatabaseManager object, and the objects created by it.

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, represented 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 language, "EN".

In J2SE and BlackBerry environments, the current locale is used to determine the default language. In J2ME environments, this method provides the only way to specify the language.

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 DecimalNumber objects together and returns the sum.
divide method	Divides the first DecimalNumber object by the second DecimalNumber object returns the quotient.
getString method	Returns the String representation of the DecimalNumber object.
isNull method	Determines if the DecimalNumber object is null.
multiply method	Multiplies two DecimalNumber objects together and returns the product.
set method	Sets the DecimalNumber object with a String value.
setNull method	Sets the DecimalNumber object to null.
subtract method	Subtracts the second DecimalNumber object from the first DecimalNumber object and returns the difference.

add method

Adds two DecimalNumber objects together and returns the sum.

Syntax

```
DecimalNumber DecimalNumber.add(
    DecimalNumber num1,
    DecimalNumber num2
) throws ULjException
```

Parameters

- **num1** A number.
- **num2** Another number.

Returns

The sum of num1 and num2.

divide method

Divides the first DecimalNumber object by the second DecimalNumber object 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 object.

Syntax

```
String DecimalNumber.getString() throws ULjException
```

Returns

The String value.

isNull method

Determines if the DecimalNumber object is null.

Syntax

```
boolean DecimalNumber.isNull()
```

Returns

True if the object is null; otherwise, returns false.

multiply method

Multiplies two DecimalNumber objects 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 object 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 object to null.

Syntax

```
void DecimalNumber.setNull() throws ULjException
```

subtract method

Subtracts the second DecimalNumber object from the first DecimalNumber object 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 the Domain object 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	Denotes the domain ID constant for a 64-bit integer (BIGINT SQL type).
BINARY variable	Denotes the domain ID constant for a variable-length binary object of maximum *size* bytes (BINARY(*size*) SQL type).
BIT variable	Denotes the domain ID constant for a bit (BIT SQL type).
DATE variable	Denotes the domain ID constant for a Date (DATE SQL type).
DOMAIN_MAX variable	Denotes the maximum kinds of Domain types.
DOUBLE variable	Denotes the domain ID constant for a 8-byte floating point (DOUBLE SQL type).
INTEGER variable	Denotes the domain ID constant for a 32-bit integer (INTEGER SQL type).
LONGBINARY variable	Denotes the domain ID constant for an arbitrary long block of binary data (BLOB) (LONG BINARY SQL type).

Name	Description
LONGBINARYFILE variable	Denotes the domain ID constant for an arbitrary file of data.
LONGVARCHAR variable	Denotes the domain ID constant for an arbitrary long block of character data (CLOB) (LONG VARCHAR SQL type).
NUMERIC variable	Denotes the domain ID constant for a numeric value of fixed precision (size) total digits and with *scale* digits after the decimal (NUMERIC(*precision*,*scale*) SQL type).
REAL variable	Denotes the domain ID constant for a 4-byte floating point (REAL SQL type).
SHORT variable	Denotes the domain ID constant for a 16-bit integer (SMALLINT SQL type).
ST_GEOMETRY variable	Denotes the denotes the domain ID constant for a geometry (GEOMETRY SQL type)
TIME variable	Denotes the domain ID constant for a Time (TIME SQL type).
TIMESTAMP variable	Denotes the domain ID constant for a Timestamp (TIMESTAMP SQL type).
TIMESTAMP_ZONE variable	Denotes the domain ID constant for a timestamps with time zones (DATETIMEOFFSET SQL type).
TINY variable	Denotes the domain ID constant for a unsigned 8-bit integer (TINYINT SQL type).
UNSIGNED_BIG variable	Denotes the domain ID constant for a unsigned 64-bit integer (UNSIGNED BIGINT SQL type).
UNSIGNED_INTEGER variable	Denotes the domain ID constant for a unsigned 32-bit integer (UNSIGNED INTEGER SQL type).
UNSIGNED_SHORT variable	Denotes the domain ID constant for a unsigned 16-bit integer (UNSIGNED SMALLINT SQL type).
UUID variable	Denotes the domain ID constant for a UniqueIdentifier (UNIQUEIDENTIFIER SQL type).
VARCHAR variable	Denotes the domain ID constant for a variable-length character string of maximum *size* bytes (VARCHAR(*size*) SQL type).

Remarks

This interface contains constants to denote the various domains, and methods that 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)
UN-SIGN-ED_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_IN-TEGER	UNSIGNED IN-TEGER	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 BI-GINT	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)

Domain Constant	SQL Type	Value Range
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.
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.

See also

- "Connection interface [UltraLiteJ]" on page 99

BIG variable

Denotes the domain ID constant for a 64-bit integer (BIGINT SQL type).

Syntax

```
final short Domain.BIG
```

See also

- "Domain interface [UltraLiteJ]" on page 136

BINARY variable

Denotes the domain ID constant for a variable-length binary object of maximum *size* bytes (BINARY(*size*) SQL type).

Syntax

```
final short Domain.BINARY
```

See also

- "Domain interface [UltraLiteJ]" on page 136

BIT variable

Denotes the domain ID constant for a bit (BIT SQL type).

Syntax

```
final short Domain.BIT
```

Remarks

BIT columns are not nullable by default.

See also

- "Domain interface [UltraLiteJ]" on page 136

DATE variable

Denotes the domain ID constant for a Date (DATE SQL type).

Syntax

```
final short Domain.DATE
```

See also

- "Domain interface [UltraLiteJ]" on page 136

DOMAIN_MAX variable

Denotes the maximum kinds of Domain types.

Syntax

```
final short Domain.DOMAIN_MAX
```

DOUBLE variable

Denotes the domain ID constant for a 8-byte floating point (DOUBLE SQL type).

Syntax

```
final short Domain.DOUBLE
```

See also

- "Domain interface [UltraLiteJ]" on page 136

INTEGER variable

Denotes the domain ID constant for a 32-bit integer (INTEGER SQL type).

Syntax

```
final short Domain.INTEGER
```

See also

- "Domain interface [UltraLiteJ]" on page 136

LONGBINARY variable

Denotes the domain ID constant for an arbitrary long block of binary data (BLOB) (LONG BINARY SQL type).

Syntax

```
final short Domain.LONGBINARY
```

See also

- "Domain interface [UltraLiteJ]" on page 136

LONGBINARYFILE variable

Denotes the domain ID constant for an arbitrary file of data.

Syntax

```
final short Domain.LONGBINARYFILE
```

See also

- "Domain interface [UltraLiteJ]" on page 136

LONGVARCHAR variable

Denotes the domain ID constant for an arbitrary long block of character data (CLOB) (LONG VARCHAR SQL type).

Syntax

```
final short Domain.LONGVARCHAR
```

See also

- "Domain interface [UltraLiteJ]" on page 136

NUMERIC variable

Denotes the domain ID constant for a numeric value of fixed precision (size) total digits and with *scale* digits after the decimal (NUMERIC(*precision*,*scale*) SQL type).

Syntax

```
final short Domain.NUMERIC
```

See also

- "Domain interface [UltraLiteJ]" on page 136

REAL variable

Denotes the domain ID constant for a 4-byte floating point (REAL SQL type).

Syntax

```
final short Domain.REAL
```

See also

- "Domain interface [UltraLiteJ]" on page 136

SHORT variable

Denotes the domain ID constant for a 16-bit integer (SMALLINT SQL type).

Syntax

```
final short Domain.SHORT
```

See also

- "Domain interface [UltraLiteJ]" on page 136

ST_GEOMETRY variable

Denotes the denotes the domain ID constant for a geometry (GEOMETRY SQL type)

Syntax

```
final short Domain.ST_GEOMETRY
```

See also

- "Domain interface [UltraLiteJ]" on page 136

TIME variable

Denotes the domain ID constant for a Time (TIME SQL type).

Syntax

```
final short Domain.TIME
```

See also

- "Domain interface [UltraLiteJ]" on page 136

TIMESTAMP variable

Denotes the domain ID constant for a Timestamp (TIMESTAMP SQL type).

Syntax

```
final short Domain.TIMESTAMP
```

See also

- "Domain interface [UltraLiteJ]" on page 136

TIMESTAMP_ZONE variable

Denotes the domain ID constant for a timestamps with time zones (DATETIMEOFFSET SQL type).

Syntax

```
final short Domain.TIMESTAMP_ZONE
```

See also

- "Domain interface [UltraLiteJ]" on page 136

TINY variable

Denotes the domain ID constant for a unsigned 8-bit integer (TINYINT SQL type).

Syntax

```
final short Domain.TINY
```

See also

- "Domain interface [UltraLiteJ]" on page 136

UNSIGNED_BIG variable

Denotes the domain ID constant for a unsigned 64-bit integer (UNSIGNED BIGINT SQL type).

Syntax

```
final short Domain.UNSIGNED_BIG
```

See also

- "Domain interface [UltraLiteJ]" on page 136

UNSIGNED_INTEGER variable

Denotes the domain ID constant for a unsigned 32-bit integer (UNSIGNED INTEGER SQL type).

Syntax

```
final short Domain.UNSIGNED_INTEGER
```

See also

- "Domain interface [UltraLiteJ]" on page 136

UNSIGNED_SHORT variable

Denotes the domain ID constant for a unsigned 16-bit integer (UNSIGNED SMALLINT SQL type).

Syntax

```
final short Domain.UNSIGNED_SHORT
```

See also

- "Domain interface [UltraLiteJ]" on page 136

UUID variable

Denotes the domain ID constant for a UniqueIdentifier (UNIQUEIDENTIFIER SQL type).

Syntax

```
final short Domain.UUID
```

See also

- "Domain interface [UltraLiteJ]" on page 136

VARCHAR variable

Denotes the domain ID constant for a variable-length character string of maximum *size* bytes (VARCHAR(*size*) SQL type).

Syntax

```
final short Domain.VARCHAR
```

See also

- "Domain interface [UltraLiteJ]" on page 136

EncryptionControl interface

Provides EncryptionControl object 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 built-in SDCard encryption if additional security is needed. Note that files encrypted using the built-in 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 object 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 built-in security of the device can work for and against applications. See UltraLiteJ Security on BlackBerry Devices.

For more information about encryption code samples, see Wandering Data blog.

See also

- "ConfigPersistent.setEncryption method [BlackBerry] [UltraLiteJ]" on page 90

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.

Name	Description
getPassword method	Returns the MobiLink password for the user specified with the setUserName method.
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 the setUserName method.
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.

Name	Description
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 DatabaseManager.createFileTransfer or DatabaseManager.createObjectStoreTransfer [BlackBerry] methods.

The instance returned by the createFileTransfer method can be used to transfer any files between MobiLink and the local file system.

For the BlackBerry 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 DatabaseManager.createConfigurationFileME method.

For Android devices and simulators, the local file system is either a media card or the internal file system where the application has appropriate permissions. Eg. /sdcard/Android/data/your.package.name/files/

The instance returned by the createObjectStoreTransfer method can be used to download UltraLite Java edition database files to the local BlackBerry object store, or vice versa.

Only valid, unencrypted UltraLite Java edition database files can be transferred. Attempting to download any files other than UltraLite Java edition databases result in exceptions being thrown.

Note
The application should not simultaneously start two downloads to the same local file.

See also

- “DatabaseManager.createConfigurationFileME method [BlackBerry] [UltraLiteJ]” on page 127
- “DatabaseManager.createFileTransfer method [UltraLiteJ]” on page 130
- “DatabaseManager.createObjectStoreTransfer method [BlackBerry] [UltraLiteJ]” on page 131

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.

Name	Description
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 is 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 the setLocalPath method 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 downloads to a temporary file and only replaces the local file once the download has completed.

For BlackBerry object store downloads, UltraLiteJ starts to write to the local store directly. This is because, in this case, it is not possible to create atomic temporary objects. Any existing local database store with the same name is corrupted upon invoking the transferFile method.

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 the download is successful; otherwise, a ULjException is thrown, and the method does not return normally.

Remarks

Errors may result in no data being sent to the listener.

See also

- "FileTransfer.downloadFile method [UltraLiteJ]" on page 150

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

- "FileTransfer.setAuthenticationParms method [UltraLiteJ]" on page 157

getAuthStatus method

Returns the authorization status code of the last file transfer attempt.

Syntax

```
abstract int FileTransfer.getAuthStatus()
```

Returns

An AuthStatusCode class 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

- "FileTransfer.setLivenessTimeout method [UltraLiteJ]" on page 158

getLocalFileName method

Determines the local file name.

Syntax

```
abstract String FileTransfer.getLocalFileName()
```

Returns

The local file name for the downloaded file.

Remarks

For file downloads, this is the name of the downloaded file. For file uploads, this is the name of the file to upload.

See also

- "FileTransfer.setLocalFileName method [UltraLiteJ]" on page 158

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

- "FileTransfer.setLocalPath method [UltraLiteJ]" on page 159

getPassword method

Returns the MobiLink password for the user specified with the setUserName method.

Syntax

```
abstract String FileTransfer.getPassword()
```

Returns

The password for the MobiLink user.

See also

- "FileTransfer.setPassword method [UltraLiteJ]" on page 160

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

- "FileTransfer.setRemoteKey method [UltraLiteJ]" on page 160

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.

Remarks

For file downloads, this is the name of the file to download. For file uploads, this is the name of the uploaded file.

See also

- "FileTransfer.setServerFileName method [UltraLiteJ]" on page 161

getStreamErrorCode method

Returns the error code reported by the stream.

Syntax

```
abstract int FileTransfer.getStreamErrorCode()
```

Returns

0 if there was no communication stream error; otherwise, returns the response code from the server.

Remarks

The error code is the HTTP response code.

getStreamErrorMessage method

Returns the error message reported by the stream itself.

Syntax

```
abstract String FileTransfer.getStreamErrorMessage()
```

Returns

Null, if no message is available; otherwise, returns 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 or StreamHTTPSParms object specifying the parameters for HTTP or HTTPS streams. The object is returned by reference.

Remarks

The synchronization stream type is specified when the FileTransfer object is created.

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

- "FileTransfer.setUserName method [UltraLiteJ]" on page 162

getVersion method

Returns the synchronization script to use.

Syntax

```
abstract String FileTransfer.getVersion()
```

Returns

The script version.

See also

- "FileTransfer.setVersion method [UltraLiteJ]" on page 162

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, returns false.

See also

- "FileTransfer.setResumePartialTransfer method [UltraLiteJ]" on page 161

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 transferred; otherwise, returns false.

Remarks

If the file is already up-to-date when the transferFile() method is invoked, this method returns true while the isTransferredFile() method returns false.

If an error occurs and the transferFile() method throws an exception, the isTransferredFile() method returns false.

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 the class description of the SyncParms class for more information about 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

- "FileTransfer.getAuthenticationParms method [UltraLiteJ]" on page 152

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 1 second, 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 for BlackBerry/J2SE/J2ME platforms, and 240 seconds for Android platforms.

See also

- "FileTransfer.getLivenessTimeout method [UltraLiteJ]" on page 153

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 is the name of the downloaded file. For file uploads, this is the name of the file to upload. The file name must not include any drive of path information.

See also

- "FileTransfer.getLocalFileName method [UltraLiteJ]" on page 153
- "FileTransfer.setLocalPath method [UltraLiteJ]" on page 159

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 a desktop, the syntax is like "C:\\ulj\\"
- For a BlackBerry file system, the syntax is like "file:///SDCard/ulj/"
- For a BlackBerry object store, this option is ignored
- For an Android file system, the syntax is like "/sdcard/Android/data/your.package.name/files/"

The default local directory also varies depending on the device operating system:

- For a desktop, if the localPath parameter is null, the file is stored in the current directory
- For a BlackBerry file system store,the localPath parameter has no default value, and must be explicitly set.
- For an Android file system store,the localPath parameter has no default value, and must be explicitly set.

See also

- "FileTransfer.getLocalPath method [UltraLiteJ]" on page 154
- "FileTransfer.setLocalFileName method [UltraLiteJ]" on page 158

setPassword method

Sets the MobiLink password for the user specified with the setUserName method.

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

- "FileTransfer.getPassword method [UltraLiteJ]" on page 154
- "FileTransfer.setUserName method [UltraLiteJ]" on page 162

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.

If this value is unspecified, the getFileName() value is used as the remote key.

See also

- "FileTransfer.getRemoteKey method [UltraLiteJ]" on page 154

setResumePartialTransfer method

Specifies whether to resume or discard a previous partial transfer.

Syntax

```
abstract void FileTransfer.setResumePartialTransfer(boolean resume)
```

Parameters

- **resume** Set to true to resume a previous partial download, or 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 object.

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 is discarded and a new download started.

For file uploads, the MobiLink server keeps 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 is discarded and a new upload is started.

See also

- "FileTransfer.isResumePartialTransfer method [UltraLiteJ]" on page 157

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 is the name of the file to download. For file uploads, this is the name of the uploaded file.

This parameter is initialized when the FileTransfer object is created.

MobiLink first searches for the file in the userName subdirectory followed by 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

- "FileTransfer.getServerFileName method [UltraLiteJ]" on page 155

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 on 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 initialized when the FileTransfer object is created.

See also

- "FileTransfer.getUserName method [UltraLiteJ]" on page 156
- "FileTransfer.setPassword method [UltraLiteJ]" on page 160

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 version string. The version string allows an UltraLiteJ application to choose from a set of synchronization scripts.

This parameter is initialized when the FileTransfer object is created.

See also

- "FileTransfer.getVersion method [UltraLiteJ]" on page 156

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 the upload is successful; otherwise, a ULjException is 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 the upload is successful; otherwise, a ULjException is thrown and the method does not return normally.

Remarks

Errors may result in no data being sent to the listener.

See also

- "FileTransfer.uploadFile method [UltraLiteJ]" on page 163

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.

Remarks

This method counts the number of bytes transferred by the current file transfer session, and adds the bytes transferred by any previous interrupted transfers.

Subtract the value returned by the getResumedAtSize method to determine the number of bytes transferred by the current session.

See also

- "FileTransferProgressData.getResumedAtSize method [UltraLiteJ]" on page 165

getFileSize method

Returns the size of the file being transferred.

Syntax

```
abstract long FileTransferProgressData.getFileSize()
```

Returns

The size of the file in bytes.

Remarks

The returned value remains constant for the duration of the file transfer session.

getResumedAtSize method

Returns the point in the file where the transfer was resumed.

Syntax

```
abstract long FileTransferProgressData.getResumedAtSize()
```

Returns

The number of bytes previously transferred.

Remarks

The returned value remains constant for the duration of the file transfer session.

FileTransferProgressListener interface

Receives file transfer progress events.

Syntax

```
public interface FileTransferProgressListener
```

Members

All members of FileTransferProgressListener interface, including all inherited members.

Name	Description
fileTransferProgressed method	Is invoked during a file transfer to inform the user of the transfer progress.

Remarks

Create a new class to receive progress reports during a file transfer.

The following example illustrates a simple SyncObserver interface that implements the FileTransferProgressListener 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

Is invoked during a file transfer to inform the user of the transfer 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; otherwise, return false to continue.

Remarks

The listener is called under the following conditions:

- Before the first disk write
- After every disk write or every 0.5 seconds, whichever is later
- After the file download is complete

Usually, the cancel request is accepted by the UltraLiteJ API. This results in a ULjException object being thrown with the errorCode set to the ULjException.SQLE_INTERRUPTED constant.

However, if the download completed for a BlackBerry object store, UltraLiteJ does not cancel the transfer.

UltraLiteJ API methods should not be invoked during a fileTransferProgressed call.

IndexSchema interface

Specifies the schema of an index and provides constants that are useful for querying system tables.

Syntax

```
public interface IndexSchema
```

Members

All members of IndexSchema interface, including all inherited members.

Name	Description
ASCENDING variable	Denotes that an index is sorted in ascending order for a column.
DESCENDING variable	Denotes that an index is sorted in descending order for a column.
PERSISTENT variable	Denotes that an index is persistent.
PRIMARY_INDEX variable	Denotes that an index is a primary key.
UNIQUE_INDEX variable	Denotes that an index is a unique index.
UNIQUE_KEY variable	Denotes that an index is a unique key.

Remarks

This interface only contains index-related constants, including flags and the sort order of indexes.

See also

- "TableSchema.SYS_INDEXES variable [UltraLiteJ]" on page 256

ASCENDING variable

Denotes that an index is sorted in ascending order for a column.

Syntax

```
final byte IndexSchema.ASCENDING
```

DESCENDING variable

Denotes that an index is sorted in descending order for a column.

Syntax

```
final byte IndexSchema.DESCENDING
```

PERSISTENT variable

Denotes that an index is persistent.

Syntax

```
final byte IndexSchema.PERSISTENT
```

Remarks

This value can be logically combined with other flags in the index_flags column of the SYS_INDEXES system table.

See also

- "TableSchema.SYS_INDEXES variable [UltraLiteJ]" on page 256

PRIMARY_INDEX variable

Denotes that an index is a primary key.

Syntax

```
final byte IndexSchema.PRIMARY_INDEX
```


Remarks

This value can be logically combined with other flags in the index_flags column of the SYS_INDEXES system table.

See also

- "TableSchema.SYS_INDEXES variable [UltraLiteJ]" on page 256

UNIQUE_INDEX variable

Denotes that an index is a unique index.

Syntax

```
final byte IndexSchema.UNIQUE_INDEX
```

Remarks

This value can be logically combined with other flags in the index_flags column of the SYS_INDEXES system table.

See also

- "TableSchema.SYS_INDEXES variable [UltraLiteJ]" on page 256

UNIQUE_KEY variable

Denotes that an index is a unique key.

Syntax

```
final byte IndexSchema.UNIQUE_KEY
```

Remarks

This value can be logically combined with other flags in the index_flags column of the SYS_INDEXES system table.

See also

- "TableSchema.SYS_INDEXES variable [UltraLiteJ]" on page 256

PreparedStatement interface

Provides methods to execute a SQL query to generate a ResultSet object, or to execute a prepared SQL statement on a 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 object.
getBlobOutputStream method	Returns an OutputStream object.
getClobWriter method	Returns a Writer object.
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 object 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 object contains a ResultSet object.
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.

Remarks

The following example demonstrates how to create a PreparedStatement object, check if a SELECT statement execution creates a ResultSet object, save any ResultSet object to a local variable, and then 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 statement runs successfully.
boolean result = ps.execute();
```



```
// Check if the PreparedStatement object contains a ResultSet object.
if (ps.hasResultSet()) {
     // Store the ResultSet in the rs variable.
     ResultSet rs = ps.getResultSet;
}
// Close the PreparedStatement object 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 following example, there are two host variables that may be set using the PreparedStatement object, and were prepared for the SQL statement in question:

```
SELECT * FROM SampleTable WHERE pk > :bound AND pk < ?
```

See also

- "Connection interface [UltraLiteJ]" on page 99
- "Connection.prepareStatement method [UltraLiteJ]" on page 111

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 object contains a ResultSet object, both objects are closed.

execute method

Executes the prepared SQL statement.

Syntax

```
boolean PreparedStatement.execute() throws ULjException
```

Returns

True if the execute statement runs successfully; otherwise, returns false.

See also

- "ResultSet interface [UltraLiteJ]" on page 185

executeQuery method

Executes the prepared SQL SELECT statement and returns a ResultSet object.

Syntax

```
ResultSet PreparedStatement.executeQuery() throws ULjException
```

Returns

The ResultSet object containing the query result of the prepared SQL SELECT statement.

See also

- "ResultSet interface [UltraLiteJ]" on page 185

getBlobOutputStream method

Returns an OutputStream object.

Overload list

Name	Description
getBlobOutputStream(int) method	Returns an OutputStream object.
getBlobOutputStream(String) method	Returns an OutputStream object.

getBlobOutputStream(int) method

Returns an OutputStream object.

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 object for the named value.

getBlobOutputStream(String) method

Returns an OutputStream object.

Syntax

```
java.io.OutputStream PreparedStatement.getBlobOutputStream(
    String name
) throws ULjException
```

Parameters

- **name** A String representing the host variable name.

Returns

The OutputStream object for the named value.

getClobWriter method

Returns a Writer object.

Overload list

Name	Description
getClobWriter(int) method	Returns a Writer object.
getClobWriter(String) method	Returns a Writer object.

getClobWriter(int) method

Returns a Writer object.

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 object for the named value.

getClobWriter(String) method

Returns a Writer object.

Syntax

```
java.io.Writer PreparedStatement.getClobWriter(
    String name
) throws ULjException
```

Parameters

- **name** A String representing the host variable name.

Returns

The Writer object 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(table-scan:5[tab1,prime_key],index-scan:
7[tab2,prime_key]))
```

See also

- "PreparedStatement.getPlanTree method [UltraLiteJ]" on page 175

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

- "PreparedStatement.getPlan method [UltraLiteJ]" on page 174

getResultSet method

Returns the ResultSet object for a prepared SQL statement.

Syntax

```
ResultSet PreparedStatement.getResultSet() throws ULjException
```

Returns

The ResultSet object containing the query result of the prepared SQL statement.

See also

- "ResultSet interface [UltraLiteJ]" on page 185

getUpdateCount method

Returns the number of rows inserted, updated or deleted since the last execute statement.

Syntax

```
int PreparedStatement.getUpdateCount() throws ULjException
```

Returns

-1 when a statement cannot perform changes; otherwise, returns the number of rows that have been changed.

hasResultSet method

Determines if the PreparedStatement object contains a ResultSet object.

Syntax

```
boolean PreparedStatement.hasResultSet() throws ULjException
```

Returns

True if a ResultSet was found; otherwise, returns false.

See also

- "ResultSet interface [UltraLiteJ]" on page 185

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 object 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.

Name	Description
set(String, boolean) method	Sets a boolean value to the host variable in the SQL statement that is defined by name.
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 name.
set(String, DecimalNumber) method	Sets a DecimalNumber object 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 object 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 DecimalNumber 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 name.

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 object 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 DecimalNumber 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.

Overload list

Name	Description
setNull(int) method	Sets a null value to the host variable in the SQL statement.

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.

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
afterLast method [Android]	Moves the cursor after the last row.
beforeFirst method [Android]	Moves the cursor before the first row.

Name	Description
close method	Closes the ResultSet object to release the memory resources associated with it.
first method [Android]	Moves the cursor to the first row.
getBlobInputStream method	Returns an InputStream object for long binary types, including file-based long binaries.
getBoolean method	Returns a boolean value.
getBytes method	Returns a byte array.
getClobReader method	Returns a Reader object.
getDate method	Returns a java.util.Date object.
getDecimalNumber method	Returns a DecimalNumber object.
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 a ResultSetMetadata object that contains the metadata for the ResultSet object.
getRowCount method [Android]	Gets the number of rows in the table.
getSize method	Gets the actual size of a result set column.
getString method	Returns a String value.
getUUIDValue method	Returns a UUIDValue object.
isNull method	Tests if the value at the specified column number is null.
last method [Android]	Moves the cursor to the last row.
next method	Fetches the next row of data in the ResultSet object.
previous method	Fetches the previous row of data in the ResultSet object.

Name	Description
relative method [Android]	Moves the cursor by an offset of rows from the current cursor position.

Remarks

A ResultSet object is generated when the execute or executeQuery method is called on a PreparedStatement object with a SQL SELECT statement.

The following example demonstrates how to fetch a row in the ResultSet object, 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 [UltraLiteJ]” on page 169
- “PreparedStatement.execute method [UltraLiteJ]” on page 171
- “PreparedStatement.executeQuery method [UltraLiteJ]” on page 172
- “Connection interface [UltraLiteJ]” on page 99

afterLast method [Android]

Moves the cursor after the last row.

Syntax

```
boolean ResultSet.afterLast() throws ULjException
```

Returns

True on success; otherwise, returns false.

beforeFirst method [Android]

Moves the cursor before the first row.

Syntax

```
boolean ResultSet.beforeFirst() throws ULjException
```

Returns

True on success; otherwise, returns false.

close method

Closes the ResultSet object to release the memory resources associated with it.

Syntax

```
void ResultSet.close() throws ULjException
```

Remarks

Subsequent attempts to fetch rows from a closed ResultSet object throw an error.

first method [Android]

Moves the cursor to the first row.

Syntax

```
boolean ResultSet.first() throws ULjException
```

Returns

True on success; otherwise, returns false.

getBlobInputStream method

Returns an InputStream object for long binary types, including file-based long binaries.

Overload list

Name	Description
getBlobInputStream(int) method	Returns an InputStream object for long binary types, including file-based long binaries.

Name	Description
getBlobInputStream(String) method	Returns an InputStream object for long binary types, including file-based long binaries.

getBlobInputStream(int) method

Returns an InputStream object for long binary types, including file-based long binaries.

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 object representation of the named value.

getBlobInputStream(String) method

Returns an InputStream object for long binary types, including file-based long binaries.

Syntax

```
java.io.InputStream ResultSet.getBlobInputStream(
    String name
) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The InputStream object 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 in the ResultSet object.

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.

Name	Description
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 object.

Overload list

Name	Description
getClobReader(int) method	Returns a Reader object.
getClobReader(String) method	Returns a Reader object.

getClobReader(int) method

Returns a Reader object.

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 object representation of the named value.

getClobReader(String) method

Returns a Reader object.

Syntax

```
java.io.Reader ResultSet.getClobReader(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The Reader object representation of the named value.

getDate method

Returns a java.util.Date object.

Overload list

Name	Description
getDate(int) method	Returns a java.util.Date object.
getDate(String) method	Returns a java.util.Date object.

getDate(int) method

Returns a java.util.Date object.

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 object representation of the named value.

getDate(String) method

Returns a java.util.Date object.

Syntax

```
java.util.Date ResultSet.getDate(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The java.util.date object representation of the named value.

getDecimalNumber method

Returns a DecimalNumber object.

Overload list

Name	Description
getDecimalNumber(int) method	Returns a DecimalNumber object.
getDecimalNumber(String) method	Returns a DecimalNumber object.

getDecimalNumber(int) method

Returns a DecimalNumber object.

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 object representation of the named value.

See also

- "DecimalNumber interface [UltraLiteJ]" on page 133

getDecimalNumber(String) method

Returns a DecimalNumber object.

Syntax

```
DecimalNumber ResultSet.getDecimalNumber(
    String name
) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The DecimalNumber object representation of the named value.

See also

- "DecimalNumber interface [UltraLiteJ]" on page 133

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 a ResultSetMetadata object that contains the metadata for the ResultSet object.

Syntax

```
ResultSetMetadata ResultSet.getResultSetMetadata() throws ULjException
```

Returns

The ResultSetMetadata object.

getRowCount method [Android]

Gets the number of rows in the table.

Syntax

```
long ResultSet.getRowCount(long threshold) throws ULjException
```

Parameters

- **threshold** The limit on the number of rows to count. 0 indicates no limit.

Returns

The number of rows in the table.

Remarks

This method is equivalent to executing "SELECT COUNT(*) FROM table".

getSize method

Gets the actual size of a result set column.

Overload list

Name	Description
getSize(int) method	Gets the actual size of a result set column.
getSize(String) method	Gets the actual size of a result set column.

getSize(int) method

Gets 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

Gets 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 object.

Overload list

Name	Description
getUUIDValue(int) method	Returns a UUIDValue object.

Name	Description
getUUIDValue(String) method	Returns a UUIDValue object.

getUUIDValue(int) method

Returns a UUIDValue object.

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 object representation of the named value.

See also

- "UUIDValue interface [UltraLiteJ]" on page 264

getUUIDValue(String) method

Returns a UUIDValue object.

Syntax

```
UUIDValue ResultSet.getUUIDValue(String name) throws ULjException
```

Parameters

- **name** A String representing the table column name.

Returns

The UUIDValue object 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.

Name	Description
isNull(String) method	Tests if the value at the specified column name 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; otherwise, returns false.

isNull(String) method

Tests if the value at the specified column name 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; otherwise, returns false.

last method [Android]

Moves the cursor to the last row.

Syntax

```
boolean ResultSet.last() throws ULjException
```

Returns

True on success; otherwise, returns false.

next method

Fetches the next row of data in the ResultSet object.

Syntax

```
boolean ResultSet.next() throws ULjException
```

Returns

True when the next row is successfully fetched; otherwise, returns false.

See also

- "ResultSetMetadata interface [UltraLiteJ]" on page 203

previous method

Fetches the previous row of data in the ResultSet object.

Syntax

```
boolean ResultSet.previous() throws ULjException
```

Returns

True when the previous row is successfully fetched; otherwise, returns false.

relative method [Android]

Moves the cursor by an offset of rows from the current cursor position.

Syntax

```
boolean ResultSet.relative(int offset) throws ULjException
```

Parameters

- **offset** The number of rows to move.

Returns

True on success; otherwise, returns 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 the alias name for a column.
getColumnCount method	Returns the total number of columns in the ResultSet object.
getCorrelationName method	Returns the correlation name for a column.
getDomainName method	Returns the name of the domain.
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 the column name in the table or the derived table.
getTableName method	Returns the table name for a column.
getWrittenName method	Returns the written name for a column.

Remarks

This interface is obtained with the ResultSet.getResultSetMetadata method.

When a column in the select list of the ResultSet object 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 object, the following information about the column domain 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 [UltraLiteJ]" on page 136
- "ResultSet interface [UltraLiteJ]" on page 185
- "ResultSet.getResultSetMetadata method [UltraLiteJ]" on page 198

getAliasName method

Returns the 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, returns the alias name for a column.

Remarks

The alias name ([AS] name) may be specified to reference a column.

getColumnCount method

Returns the total number of columns in the ResultSet object.

Syntax

```
int ResultSetMetadata.getColumnCount() throws ULjException
```

Returns

The number of columns.

getCorrelationName method

Returns the 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 no correlation name for the column; otherwise, returns the correlation name for a column.

Remarks

The correlation name specified ([AS] correlation-name) in the FROM clause to specify a table expression, such as a derived table.

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 no qualified name for the column; otherwise, returns the 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.

When the ResultSet column does not refer to a column in a table and an alias was specified, the alias name is returned.

getTableColumnName method

Returns the column name in the table or the 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 no table name for the column; otherwise, returns the table name for a column.

getTableName method

Returns the 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 no table name for the column; otherwise, returns the table name for a column.

Remarks

The table is the name of the table that ResultSet column references (possibly as through a correlation name).

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 no written name for the column; otherwise, returns the written name for a column.

Remarks

The written name is the simple or compound name specified as the indicated column in the select list.

SISListener interface [BlackBerry]

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 the SISListener interface using the appropriate DatabaseManager.createSISHTTPListener method.

See also

- "DatabaseManager.createSISHTTPListener method [BlackBerry] [UltraLiteJ]" on page 131

startListening method

Creates and starts the listening thread.

Syntax

```
void SISListener.startListening()
```

stopListening method

Stops the listening thread.

Syntax

```
void SISListener.stopListening()
```

SISRequestHandler interface [BlackBerry]

Handles server-initiated synchronization requests.

Syntax

```
public interface SISRequestHandler
```

Members

All members of SISRequestHandler interface, including all inherited members.

Name	Description
onError method	Handles server-initiated synchronization errors that occur while listening.
onRequest method	Handles server-initiated synchronization requests on worker threads.

See also

- "SISListener interface [BlackBerry] [UltraLiteJ]" on page 209

onError method

Handles server-initiated synchronization errors that occur while 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 server-initiated synchronization 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 [UltraLiteJ]" on page 220

Members

All members of StreamHTTPParms interface, including all inherited members.

Name	Description
getE2eePublicKey method [Android]	Get the name of the file containing the end-to-end public key.
getExtraParameters method [Android]	Gets the extra MobiLink client network protocol options.
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.
isRestartable method	Determines whether restartable HTTP is used.
setE2eePublicKey method [Android]	Sets the name of the file containing the end-to-end public key.
setExtraParameters method [Android]	Sets extra MobiLink client network protocol options.
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.
setRestartable method	Enables or disables restartable HTTP.
setURLSuffix method	Specifies the URL suffix to connect to the MobiLink server.
setZlibCompression method	Enables or disables ZLIB compression.
setZlibDownloadWindowSize method	Sets the download window size for ZLIB compression.
setZlibUploadWindowSize method	Sets the upload window size for ZLIB compression.

Name	Description
zlibCompressionEnabled method	Determines if ZLIB compression is enabled.

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: "-x 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 [UltraLiteJ]" on page 232
- "SyncParms.getStreamParms method [UltraLiteJ]" on page 237

getE2eePublicKey method [Android]

Get the name of the file containing the end-to-end public key.

Syntax

```
String StreamHTTPParms.getE2eePublicKey()
```

Returns

the name of the file containing the end-to-end public key.

getExtraParameters method [Android]

Gets the extra MobiLink client network protocol options.

Syntax

```
String StreamHTTPParms.getExtraParameters()
```

Returns

The extra protocol options that have been set.

getHost method

Returns the host name of the MobiLink server.

Syntax

```
String StreamHTTPParms.getHost()
```

Returns

The name of the host.

See also

- "StreamHTTPParms.setHost method [UltraLiteJ]" on page 216
- "StreamHTTPParms.getPort method [UltraLiteJ]" on page 214
- "StreamHTTPParms.setPort method [UltraLiteJ]" on page 217

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

- "StreamHTTPParms.setOutputBufferSize method [UltraLiteJ]" on page 216

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

- "StreamHTTPParms.setPort method [UltraLiteJ]" on page 217

getURLSuffix method

Returns the URL suffix of the MobiLink server.

Syntax

```
String StreamHTTPParms.getURLSuffix()
```

Returns

The String containing the URL suffix.

See also

- "StreamHTTPParms.setURLSuffix method [UltraLiteJ]" on page 218

isRestartable method

Determines whether restartable HTTP is used.

Syntax

```
boolean StreamHTTPParms.isRestartable()
```

Returns

True if restartable HTTP is enabled; otherwise, returns false.

See also

- "StreamHTTPParms.setRestartable method [UltraLiteJ]" on page 217

setE2eePublicKey method [Android]

Sets the name of the file containing the end-to-end public key.

Syntax

```
void StreamHTTPParms.setE2eePublicKey(String public_key)
```

Parameters

- **public_key** The name of RSA public key file used in the encryption.

Remarks

By default, this value is null, indicating that end-to-end encryption is not used.

Corresponds to MobiLink client network protocol option "e2ee_public_key".

setExtraParameters method [Android]

Sets extra MobiLink client network protocol options.

Syntax

```
void StreamHTTPParms.setExtraParameters(String parms)
```

Parameters

- **parms** A semicolon delimited list of protocol options.

Remarks

These options are appended to the list that is built from the settings resulting from the methods of this class.

Options that are set by this method override the same options that are set by other methods. For example, if "host=abc" is contained in the extra parameters, and the setHost("xyz") method is called, then the host option is "abc".

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

- “StreamHTTPParms.getHost method [UltraLiteJ]” on page 214
- “StreamHTTPParms.getPort method [UltraLiteJ]” on page 214
- “StreamHTTPParms.setPort method [UltraLiteJ]” on page 217

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 J2ME; 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

- "StreamHTTPParms.getOutputBufferSize method [UltraLiteJ]" on page 214

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

- "StreamHTTPParms.getPort method [UltraLiteJ]" on page 214

setRestartable method

Enables or disables restartable HTTP.

Syntax

```
void StreamHTTPParms.setRestartable(boolean isRestartable)
```

Parameters

- **isRestartable** Set to true to enable restartable HTTP. The default value is false.

Remarks

When restartable HTTP is enabled, UltraLiteJ can tolerate network interruptions so that synchronizations do not fail as often on unreliable networks.

To use restartable HTTP, both UltraLiteJ and the MobiLink server must have applied CR#690250.

See also

- "StreamHTTPParms.isRestartable method [UltraLiteJ]" on page 215

setURLSuffix method

Specifies the URL suffix to connect to the MobiLink server.

Syntax

```
void StreamHTTPParms.setURLSuffix(String v)
```

Parameters

- **v** The URL suffix string.

Remarks

UltraLiteJ forms URLs in the following format:

```
[http|https]://host-name:port-number/url-suffix
```

By default, *url-suffix* is "Mobilink/". You can set the URL suffix to the default by setting **v** to null.

The following code illustrates how to specify a URL suffix that instructs a BlackBerry smartphone to only connect using a Wi-Fi connection:

```
myHTTPParms.setURLSuffix(";deviceside=true;interface=wifi");
```

The following code illustrates how to specify a URL that instructs a BlackBerry smartphone to synchronize over the HTTPS protocol with a BlackBerry Enterprise Server (BES), which may be required by some BlackBerry smartphones:

```
myHTTPParms.setURLSuffix(";EndToEndRequired");
```

See also

- "StreamHTTPParms.getURLSuffix method [UltraLiteJ]" on page 215

setZlibCompression method

Enables or disables ZLIB compression.

Syntax

```
void StreamHTTPParms.setZlibCompression(boolean enable)
```


Parameters

- **enable** Set to true to enable ZLIB compression, or false to disable ZLIB compression.

Remarks

By default, ZLIB compression is disabled

Corresponds to MobiLink client network protocol option "compression=zlib".

setZlibDownloadWindowSize method

Sets the download window size for ZLIB compression.

Syntax

```
void StreamHTTPParms.setZlibDownloadWindowSize(int size)
```

Parameters

- **size** The compression window size specification, inclusively ranging from 9 to 15. This parameter is the base two logarithm of the window size. (the size of the history buffer)

Remarks

Corresponds to MobiLink client network protocol option "zlib_download_window_size".

setZlibUploadWindowSize method

Sets the upload window size for ZLIB compression.

Syntax

```
void StreamHTTPParms.setZlibUploadWindowSize(int size)
```

Parameters

- **size** The compression window size specification, inclusively ranging from 9 to 15. This parameter is the base two logarithm of the window size. (the size of the history buffer)

Remarks

Corresponds to MobiLink client network protocol option "zlib_upload_window_size".

zlibCompressionEnabled method

Determines if ZLIB compression is enabled.

Syntax

```
boolean StreamHTTPParms.zlibCompressionEnabled()
```

Returns

True if enabled; otherwise, returns false.

StreamHTTPSParms interface

Represents HTTPS stream parameters that define how to communicate with a MobiLink server using secure HTTPS connections.

Syntax

```
public interface StreamHTTPSParms
```

Base classes

- "StreamHTTPParms interface [UltraLiteJ]" on page 211

Members

All members of StreamHTTPSParms interface, including all inherited members.

Name	Description
getCertificateCompany method	Returns the certificate company name for verification of secure connections.
getCertificateName method	Returns the certificate common name for verification of secure connections.
getCertificateUnit method	Returns the certificate unit name for verification of secure connections.
getE2eePublicKey method [Android]	Get the name of the file containing the end-to-end public key.
getExtraParameters method [Android]	Gets the extra MobiLink client network protocol options.
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.

Name	Description
isRestartable method	Determines whether restartable HTTP is used.
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.
setE2eePublicKey method [Android]	Sets the name of the file containing the end-to-end public key.
setExtraParameters method [Android]	Sets extra MobiLink client network protocol options.
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.
setRestartable method	Enables or disables restartable HTTP.
setTrustedCertificates method	Sets a file containing a list of trusted root certificates used for secure synchronization.
setURLSuffix method	Specifies the URL suffix to connect to the MobiLink server.
setZlibCompression method	Enables or disables ZLIB compression.
setZlibDownloadWindowSize method	Sets the download window size for ZLIB compression.
setZlibUploadWindowSize method	Sets the upload window size for ZLIB compression.
zlibCompressionEnabled method	Determines if ZLIB compression is enabled.

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: "-x 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 J2SE, you can deploy the required trusted root certificate 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 javax.net.ssl.trustStore Java system property to its location (set the javax.net.ssl.trustStorePassword method to an appropriate value)
3. Use the setTrustedCertificates(String) parameter to point to the deployed certificate file.

To enhance security, the setCertificateName, setCertificateCompany, and 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 [UltraLiteJ]” on page 232
- “SyncParms.getStreamParms method [UltraLiteJ]” on page 237
- “StreamHTTPSParms.setCertificateCompany method [UltraLiteJ]” on page 223
- “StreamHTTPSParms.setCertificateName method [UltraLiteJ]” on page 224
- “StreamHTTPSParms.setCertificateUnit method [UltraLiteJ]” on page 224
- “StreamHTTPSParms.setTrustedCertificates method [UltraLiteJ]” on page 224

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 file name of the trusted root certificates file.

See also

- "StreamHTTPSParms.setTrustedCertificates method [UltraLiteJ]" on page 224

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 certificate.

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 file name of the trusted root certificate.

Remarks

This method supports any X.509 format that the Java Runtime Environment on the platform allows. The JKS KeyStore type for storing root certificates is used by J2SE, and the BKS KeyStore is used by Android smartphones.

This method can only be used on the J2SE platform, and Android smartphones.

Certificates are used according to the following rules of precedence:

1. If this method is called, then the certificates from the specified file are used.
2. If this method is not called and certificates were set in the database by the ulinit or ulload utilities, then those certificates are used.
3. If certificates are not specified by either this method or by the ulinit or ulload utilities, and you are on Android, then certificates are read from the operating system's trusted certificate store. This certificate store is used by web browsers when they connect to secure web servers via HTTPS.

See also

- "StreamHTTPSParms.getTrustedCertificates method [UltraLiteJ]" on page 223

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.

Remarks

Create a new class that performs synchronization, and implement it using the SyncParms.setSyncObserver method to receive synchronization progress reports.

The following example illustrates a simple SyncObserver object implementation:

```
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 above class can be enabled with the following method call:

```
SyncParms.setSyncObserver(new MyObserver());
```

See also

- "SyncParms class [UltraLiteJ]" on page 232
- "SyncParms.setSyncObserver method [UltraLiteJ]" on page 244

syncProgress method

Informs the user of progress.

Syntax

```
boolean SyncObserver.syncProgress(int state, SyncResult data)
```

Parameters

- **state** One of the SyncObserver.States constants, representing the current state of the synchronization.
- **data** A SyncResult object containing the latest synchronization results.

Returns

return True to cancel the synchronization; otherwise, returns false to continue synchronization.

Remarks

This method is invoked during synchronization.

The various states, which are signaled as packets, are received and sent. Since multiple tables may be uploaded or downloaded in a single packet, calls to this method 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 [UltraLiteJ]" on page 227
- "SyncParms.setSyncObserver method [UltraLiteJ]" on page 244
- "SyncResult class [UltraLiteJ]" 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	Denotes that the downloaded rows are being committed to the database.
CONNECTING variable	Denotes that a synchronization is starting.
DISCONNECTING variable	Denotes that the synchronization stream is disconnecting.
DONE variable	Denotes that synchronization is complete.
ERROR variable	Denotes that synchronization is complete but an error occurred.
FINISHING_UPLOAD variable	Denotes that the upload is finalizing.
RECEIVING_DATA variable	Denotes that schema information or row data is being received.
RECEIVING_TABLE variable	Denotes that a new table is being downloaded.
RECEIVING_UPLOAD_ACK variable	Denotes that an upload acknowledgement is being downloaded.
ROLLING_BACK_DOWNLOAD variable	Denotes that the downloaded rows are being committed to the database.
SENDING_DATA variable	Denotes that schema information or row data is being sent.

Name	Description
SENDING_DOWNLOAD_ACK variable	Denotes that an acknowledgement of a complete download is being sent.
SENDING_HEADER variable	Denotes that 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	Denotes that a new table is being uploaded.
STARTING variable	Denotes that a synchronization is starting.

See also

- "SyncParms.setSyncObserver method [UltraLiteJ]" on page 244
- "SyncObserver interface [UltraLiteJ]" on page 225

CHECKING_LAST_UPLOAD variable

Checking the status of the previous upload.

Syntax

```
final int SyncObserver.States.CHECKING_LAST_UPLOAD
```

Remarks

Denotes that the status of the previous upload is being checked.

COMMITTING_DOWNLOAD variable

Denotes that the downloaded rows are being committed to the database.

Syntax

```
final int SyncObserver.States.COMMITTING_DOWNLOAD
```

CONNECTING variable

Denotes that a synchronization is starting.

Syntax

```
final int SyncObserver.States.CONNECTING
```


Remarks

No actions have taken place yet.

DISCONNECTING variable

Denotes that the synchronization stream is disconnecting.

Syntax

```
final int SyncObserver.States.DISCONNECTING
```

DONE variable

Denotes that synchronization is complete.

Syntax

```
final int SyncObserver.States.DONE
```

Remarks

No other states are reported.

ERROR variable

Denotes that synchronization is complete but an error occurred.

Syntax

```
final int SyncObserver.States.ERROR
```

FINISHING_UPLOAD variable

Denotes that the upload is finalizing.

Syntax

```
final int SyncObserver.States.FINISHING_UPLOAD
```

RECEIVING_DATA variable

Denotes that schema information or row data is being received.

Syntax

```
final int SyncObserver.States.RECEIVING_DATA
```

RECEIVING_TABLE variable

Denotes that a new table is being downloaded.

Syntax

```
final int SyncObserver.States.RECEIVING_TABLE
```

RECEIVING_UPLOAD_ACK variable

Denotes that an upload acknowledgement is being downloaded.

Syntax

```
final int SyncObserver.States.RECEIVING_UPLOAD_ACK
```

ROLLING_BACK_DOWNLOAD variable

Denotes that the downloaded rows are being committed to the database.

Syntax

```
final int SyncObserver.States.ROLLING_BACK_DOWNLOAD
```

Remarks

Denotes that synchronization is rolling back the download because an error was encountered during the download.

SENDING_DATA variable

Denotes that schema information or row data is being sent.

Syntax

```
final int SyncObserver.States.SENDING_DATA
```

SENDING_DOWNLOAD_ACK variable

Denotes that an acknowledgement of a complete download is being sent.

Syntax

```
final int SyncObserver.States.SENDING_DOWNLOAD_ACK
```

SENDING_HEADER variable

Denotes that the synchronization stream has been opened and the header is about to be sent.

Syntax

```
final int SyncObserver.States.SENDING_HEADER
```

Remarks

Denotes that the synchronization stream has opened, and that the header is about to be sent.

SENDING_SCHEMA variable

The schema is being sent.

Syntax

```
final int SyncObserver.States.SENDING_SCHEMA
```

Remarks

Denotes that the schema is being sent.

SENDING_TABLE variable

Denotes that a new table is being uploaded.

Syntax

```
final int SyncObserver.States.SENDING_TABLE
```

STARTING variable

Denotes that 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
getAcknowledgeDownload method	Determines if the client sends download acknowledgments.
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 the setUserName method.
getPassword method	Returns the MobiLink password for the user specified with the setUserName method.
getPublications method	Returns the publications to be synchronized.
getSendColumnNames method	Returns true if column names are sent to the MobiLink server.
getStreamParms method	Returns the parameters used to configure the synchronization stream.
getSyncObserver method	Returns the currently specified SyncObserver object.
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 script version to use.
isDownloadOnly method	Determines if the synchronization is download-only.

Name	Description
isPingOnly method	Determines whether the client pings the MobiLink server or performs a synchronization.
isUploadOnly method	Determines if the synchronization is upload-only.
setAcknowledgeDownload method	Specifies whether the client should send download acknowledgements.
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 method.
setPassword method	Sets the MobiLink password for the user specified with the setUserName method.
setPingOnly method	Sets the client to ping the MobiLink server rather than perform a synchronization.
setPublications method	Sets the publications to be synchronized.
setSendColumnNames method	Specifies whether column names are sent to the MobiLink server during a synchronization.
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 with the Connection.createSyncParms method.

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 trailing spaces in values are ignored unless quoted. For example, the following code specifies **Table A**, then **Table B,D**, then **Table C**":

```
syncParms.setTableOrder("'Table A',\"Table B,D",Table C" );
```

specifies

See also

- “SyncParms.getStreamParms method [UltraLiteJ]” on page 237
- “SyncParms.setUserName method [UltraLiteJ]” on page 246
- “Connection.createSyncParms method [UltraLiteJ]” on page 105
- “StreamHTTPParms interface [UltraLiteJ]” on page 211
- “StreamHTTPSParms interface [UltraLiteJ]” on page 220

getAcknowledgeDownload method

Determines if the client sends download acknowledgments.

Syntax

```
abstract boolean SyncParms.getAcknowledgeDownload()
```

Returns

True if the client sends download acknowledgments; otherwise, returns false.

See also

- "SyncParms.setAcknowledgeDownload method [UltraLiteJ]" on page 240

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

- "SyncParms.setAuthenticationParms method [UltraLiteJ]" on page 240

getLivenessTimeout method

Returns the liveness timeout length, in seconds.

Syntax

```
abstract int SyncParms.getLivenessTimeout()
```

Returns

The timeout.

See also

- "SyncParms.setLivenessTimeout method [UltraLiteJ]" on page 241

getNewPassword method

Returns the new MobiLink password for the user specified with the setUserName method.

Syntax

```
abstract String SyncParms.getNewPassword()
```

Returns

The new password set after the next synchronization.

See also

- "SyncParms.setUserName method [UltraLiteJ]" on page 246
- "SyncParms.setNewPassword method [UltraLiteJ]" on page 242

getPassword method

Returns the MobiLink password for the user specified with the setUserName method.

Syntax

```
abstract String SyncParms.getPassword()
```

Returns

The password for the MobiLink user.

See also

- "SyncParms.setPassword method [UltraLiteJ]" on page 242

getPublications method

Returns the publications to be synchronized.

Syntax

```
abstract String SyncParms.getPublications()
```

Returns

The set of publications to synchronize.

See also

- "SyncParms.setPublications method [UltraLiteJ]" on page 243

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

- "SyncParms.setSendColumnNames method [UltraLiteJ]" on page 244

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

- "Connection.createSyncParms method [UltraLiteJ]" on page 105
- "StreamHTTPParms interface [UltraLiteJ]" on page 211
- "StreamHTTPSParms interface [UltraLiteJ]" on page 220

getSyncObserver method

Returns the currently specified SyncObserver object.

Syntax

```
abstract SyncObserver SyncParms.getSyncObserver()
```

Returns

The SyncObserver object, or null if an observer was not specified.

See also

- "SyncParms.setSyncObserver method [UltraLiteJ]" on page 244

getSyncResult method

Returns the SyncResult object that contains the status of the synchronization.

Syntax

```
abstract SyncResult SyncParms.getSyncResult()
```

Returns

The SyncResult object representing the result of the last call to the Connection.synchronize method.

Remarks

The following example illustrates how to get the result set of the last call to the Connection.synchronize method:

```
conn.synchronize( mySyncParms );
SyncResult result = mySyncParms.getSyncResult();
display(
    "*** Synchronized *** sent=" + result.getSentRowCount()
    + ", received=" + result.getReceivedRowCount()
);
```

Note
This method does not return the result of the last SYNCHRONIZE SQL statement. To obtain the SyncResult object for the last SYNCHRONIZE SQL statement, use the getSyncResult method on the Connection object passed in.

See also

- "SyncResult class [UltraLiteJ]" on page 247
- "Connection.getSyncResult method [UltraLiteJ]" on page 110

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, returns null if a table order was not specified. See the class description for more information about comma separated lists.

See also

- "SyncParms.setTableOrder method [UltraLiteJ]" on page 245

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

- "SyncParms.setUserName method [UltraLiteJ]" on page 246

getVersion method

Returns the script version to use.

Syntax

```
abstract String SyncParms.getVersion()
```

Returns

The script version.

See also

- "SyncParms.setVersion method [UltraLiteJ]" on page 246

isDownloadOnly method

Determines if the synchronization is download-only.

Syntax

```
abstract boolean SyncParms.isDownloadOnly()
```

Returns

True if uploads are disabled; otherwise, returns false.

See also

- "SyncParms.setDownloadOnly method [UltraLiteJ]" on page 241

isPingOnly method

Determines whether the client pings the MobiLink server or performs a synchronization.

Syntax

```
abstract boolean SyncParms.isPingOnly()
```

Returns

True if the client only pings the server; otherwise, returns false.

See also

- "SyncParms.setPingOnly method [UltraLiteJ]" on page 243

isUploadOnly method

Determines if the synchronization is upload-only.

Syntax

```
abstract boolean SyncParms.isUploadOnly()
```

Returns

True if downloads are disabled; otherwise, returns false.

See also

- "SyncParms.setUploadOnly method [UltraLiteJ]" on page 245

setAcknowledgeDownload method

Specifies whether the client should send download acknowledgements.

Syntax

```
abstract void SyncParms.setAcknowledgeDownload(boolean ack)
```

Parameters

- **ack** Set to true to have the client acknowledge a download; otherwise, set to false.

Remarks

The default is false.

See also

- "SyncParms.getAcknowledgeDownload method [UltraLiteJ]" on page 234

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 the class description for more information about comma separated lists.

Remarks

Only the first 255 strings are used and each string should be no longer than the MobiLink server's limit for authentication parameters. (currently 4000 UTF8 bytes)

Strings longer than 21K characters are truncated when sent to MobiLink, and strings that exceed the server's limit for authentication parameters cause a server-side synchronization error.

See also

- "SyncParms.getAuthenticationParms method [UltraLiteJ]" on page 235

setDownloadOnly method

Sets the synchronization as download-only.

Syntax

```
abstract void SyncParms.setDownloadOnly(boolean v)
```

Parameters

- **v** Set to true to disable uploads, or set false to enable uploads.

Remarks

The default is false. Specifying true automatically calls the setPingOnly and setUploadOnly methods, and sets them to false.

See also

- "SyncParms.isDownloadOnly method [UltraLiteJ]" on page 239
- "SyncParms.setPingOnly method [UltraLiteJ]" on page 243
- "SyncParms.setUploadOnly method [UltraLiteJ]" on page 245

setLivenessTimeout method

Sets the liveness timeout length, in seconds.

Syntax

```
abstract void SyncParms.setLivenessTimeout(
    int seconds
) throws ULjException
```

Parameters

- **seconds** 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 for BlackBerry/J2SE/J2ME platforms, and 240 seconds for Android platforms.

See also

- "SyncParms.getLivenessTimeout method [UltraLiteJ]" on page 235

setNewPassword method

Sets a new MobiLink password for the user specified with setUserName method.

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

- "SyncParms.getNewPassword method [UltraLiteJ]" on page 235
- "SyncParms.setPassword method [UltraLiteJ]" on page 242
- "SyncParms.setUserName method [UltraLiteJ]" on page 246

setPassword method

Sets the MobiLink password for the user specified with the setUserName method.

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

- "SyncParms.getPassword method [UltraLiteJ]" on page 236
- "SyncParms.setNewPassword method [UltraLiteJ]" on page 242
- "SyncParms.setUserName method [UltraLiteJ]" on page 246

setPingOnly method

Sets the client to ping the MobiLink server rather than perform a synchronization.

Syntax

```
abstract void SyncParms.setPingOnly(boolean v)
```

Parameters

- **v** Set to true to only ping the server, or set false to perform a synchronization.

Remarks

The default is false. Specifying true automatically calls the setDownloadOnly and setUploadOnly methods, and sets them to false.

See also

- "SyncParms.isPingOnly method [UltraLiteJ]" on page 239
- "SyncParms.setDownloadOnly method [UltraLiteJ]" on page 241
- "SyncParms.setUploadOnly method [UltraLiteJ]" on page 245

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 the class description for more information about comma separated lists.

Remarks

The default is set to the Connection.SYNC_ALL constant, which is used to denote the synchronization of all tables in the database. To synchronize all publications, set this method to the Connection.SYNC_ALL_PUBS constant.

See also

- "SyncParms.getPublications method [UltraLiteJ]" on page 236
- "Connection.SYNC_ALL variable [UltraLiteJ]" on page 119
- "Connection.SYNC_ALL_PUBS variable [UltraLiteJ]" on page 120

setSendColumnNames method

Specifies whether column names are sent to the MobiLink server during a synchronization.

Syntax

```
abstract void SyncParms.setSendColumnNames(boolean c)
```

Parameters

- **c** Set to true if column names should be sent.

Remarks

Column names are used by the server only when using a MobiLink server API.

The default value is false.

See also

- "SyncParms.getSendColumnNames method [UltraLiteJ]" on page 236

setSyncObserver method

Sets a SyncObserver object to monitor the progress of the synchronization.

Syntax

```
abstract void SyncParms.setSyncObserver(SyncObserver so)
```

Parameters

- **so** A SyncObserver object.

Remarks

The default is null, suggesting no observer.

See also

- "SyncObserver interface [UltraLiteJ]" on page 225

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 the class description for more information about 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 parameter are 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 synchronized after the specified tables.

The default is a null reference, which does not override the default ordering of tables.

See also

- "SyncParms.getTableOrder method [UltraLiteJ]" on page 238
- "SyncParms.setPublications method [UltraLiteJ]" on page 243

setUploadOnly method

Sets the synchronization as upload-only.

Syntax

```
abstract void SyncParms.setUploadOnly(boolean v)
```

Parameters

- **v** Set to true to disable downloads, or set or false to enable downloads.

Remarks

The default is false. Specifying true automatically calls the setDownloadOnly and setPingOnly methods, and sets them to false.

See also

- "SyncParms.isUploadOnly method [UltraLiteJ]" on page 240
- "SyncParms.setDownloadOnly method [UltraLiteJ]" on page 241
- "SyncParms.setPingOnly method [UltraLiteJ]" on page 243

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 following:

- Download content
- Whether to record the synchronization state
- Whether 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 initialized when the SyncParms object is created.

See also

- "SyncParms.getUserName method [UltraLiteJ]" on page 238
- "SyncParms.setPassword method [UltraLiteJ]" on page 242
- "SyncParms.setNewPassword method [UltraLiteJ]" on page 242
- "Connection.createSyncParms method [UltraLiteJ]" on page 105

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 initialized when the SyncParms object is created.

See also

- "SyncParms.getVersion method [UltraLiteJ]" on page 239
- "Connection.createSyncParms method [UltraLiteJ]" on page 105

HTTP_STREAM variable

Creates a SyncParms object for HTTP synchronizations.

Syntax

```
final int SyncParms.HTTP_STREAM
```

See also

- "Connection.createSyncParms method [UltraLiteJ]" on page 105

HTTPS_STREAM variable

Creates a SyncParms object for secure HTTPS synchronizations.

Syntax

```
final int SyncParms.HTTPS_STREAM
```

See also

- "Connection.createSyncParms method [UltraLiteJ]" on page 105

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.

Name	Description
getCurrentTableName method	Returns the name of the table currently being synchronized.
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 synchronized tables so far.
getTotalTableCount method	Returns the number of tables to be synchronized.
isUploadOK method	Determines if the last upload synchronization was successful.

See also

- "SyncParms.getSyncResult method [UltraLiteJ]" on page 237

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 synchronized.

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, returns 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 received.

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 sent.

getStreamErrorCode method

Returns the error code reported by the stream itself.

Syntax

```
abstract int SyncResult.getStreamErrorCode()
```

Returns

0 if there was no communication stream error; otherwise, returns the response code from the server.

Remarks

This method returns 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, returns the response message.

Remarks

This method returns the HTTP response message.

getSyncedTableCount method

Returns the number of synchronized tables so far.

Syntax

```
abstract int SyncResult.getSyncedTableCount()
```

Returns

The number of tables synchronized.

getTotalTableCount method

Returns the number of tables to be synchronized.

Syntax

```
abstract int SyncResult.getTotalTableCount()
```

Returns

The number of tables to synchronize.

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, returns 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

- "SyncResult.getAuthStatus method [UltraLiteJ]" on page 248

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	Contains the name of the system table containing information on publication articles.
SYS_COLUMNS variable	Contains the name of the system table containing information on the table columns in the database.
SYS_FKEY_COLUMNS variable	Contains the name of the system table containing information on foreign key columns.
SYS_FOREIGN_KEYS variable	Contains the name of the system table containing information on foreign keys in the database.
SYS_INDEX_COLUMNS variable	Contains the name of the system table containing information on the index columns in the database.
SYS_INDEXES variable	Contains the name of the system table containing information on the table indexes in the database.
SYS_INTERNAL variable	Contains the name of the system table containing internal information.
SYS_PRIMARY_INDEX variable	Contains the name of the primary key index of system tables.
SYS_PUBLICATIONS variable	Contains the name of the system table containing information on database publications.
SYS_TABLES variable	Contains the name of the system table containing information on the tables in the database.
SYS_ULDATA variable	Contains the name of the system table containing information on system values.
SYS_ULDATA_INTERNAL variable	Contains the type for internal system data.

Name	Description
SYS_ULDATA_OPTION variable	Contains the type for option system data.
SYS_ULDATA_PROPERTY variable	Contains the type for property system data.
TABLE_IS_DOWNLOAD_ONLY variable	Denotes that a table is a download-only table (a table that is uploaded when synchronized).
TABLE_IS_NOSYNC variable	Denotes that a table is a non-synchronizing table.
TABLE_IS_SYSTEM variable	Denotes that 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 the **sysuldata** system table.

SYS_ARTICLES variable

Contains the name of the system table containing information on publication articles.

Syntax

```
final String TableSchema.SYS_ARTICLES
```

SYS_COLUMNS variable

Contains the name of the system table containing information on the table columns in the database.

Syntax

```
final String TableSchema.SYS_COLUMNS
```

SYS_FKEY_COLUMNS variable

Contains the name of the system table containing information on foreign key columns.

Syntax

```
final String TableSchema.SYS_FKEY_COLUMNS
```

SYS_FOREIGN_KEYS variable

Contains the name of the system table containing information on foreign keys in the database.

Syntax

```
final String TableSchema.SYS_FOREIGN_KEYS
```

SYS_INDEX_COLUMNS variable

Contains the name of the system table containing information on the index columns in the database.

Syntax

```
final String TableSchema.SYS_INDEX_COLUMNS
```

SYS_INDEXES variable

Contains the name of the system table containing information on the table indexes in the database.

Syntax

```
final String TableSchema.SYS_INDEXES
```

SYS_INTERNAL variable

Contains the name of the system table containing internal information.

Syntax

```
final String TableSchema.SYS_INTERNAL
```

SYS_PRIMARY_INDEX variable

Contains the name of the primary key index of system tables.

Syntax

```
final String TableSchema.SYS_PRIMARY_INDEX
```

SYS_PUBLICATIONS variable

Contains the name of the system table containing information on database publications.

Syntax

```
final String TableSchema.SYS_PUBLICATIONS
```

SYS_TABLES variable

Contains the name of the system table containing information on the tables in the database.

Syntax

```
final String TableSchema.SYS_TABLES
```

SYS_ULDATA variable

Contains the name of the system table containing information on system values.

Syntax

```
final String TableSchema.SYS_ULDATA
```

SYS_ULDATA_INTERNAL variable

Contains the type for internal system data.

Syntax

```
final String TableSchema.SYS_ULDATA_INTERNAL
```

SYS_ULDATA_OPTION variable

Contains the type for option system data.

Syntax

```
final String TableSchema.SYS_ULDATA_OPTION
```

SYS_ULDATA_PROPERTY variable

Contains the type for property system data.

Syntax

```
final String TableSchema.SYS_ULDATA_PROPERTY
```

TABLE_IS_DOWNLOAD_ONLY variable

Denotes that a table is a download-only table (a table that is uploaded when synchronized).

Syntax

```
final short TableSchema.TABLE_IS_DOWNLOAD_ONLY
```

Remarks

This value can be logically combined with other flags in the table_flags column of the SYS_TABLES table except the TABLE_IS_NOSYNC flag.

TABLE_IS_NOSYNC variable

Denotes that a table is a non-synchronizing table.

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 except the TABLE_IS_DOWNLOAD_ONLY flag.

TABLE_IS_SYSTEM variable

Denotes that 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

Supersedes the exceptions thrown by the database.

Syntax

```
public class ULjException
```


Members

All members of ULjException class, including all inherited members.

Name	Description
getCausingException method	Returns the ULjException object when an exception is caused.
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 object when an exception is caused.

Syntax

```
abstract ULjException ULjException.getCausingException()
```

Returns

Null if no causing exceptions exist; otherwise, returns the ULjException object.

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, returns the zero-base offset within that string where the error occurred.

Unsigned64 class

Implements unsigned 64-bit binary values.

Syntax

```
public class Unsigned64
```

Members

All members of Unsigned64 class, including all inherited members.

Name	Description
add method	Adds two values together and places the result in self.
compare method	Compares two long values.
divide method	Divides two values and places the result in self.
multiply method	Multiplies two values together and places the result in self.
remainder method	Returns the remainder when one value is divided by another.
subtract method	Subtracts two values and places the result in self.

Remarks

The intent of this class is to keep values as long integers and interpret them using the static methods in this class.

This class cannot be instantiated.

add method

Adds two values together and places the result in self.

Syntax

```
final long Unsigned64.add(long v1, long v2)
```

Parameters

- **v1** The first operand
- **v2** The second operand

Returns

The sum of the operands.

compare method

Compares two long values.

Overload list

Name	Description
compare(int, int) method	Compares two integer values.
compare(long, long) method	Compares two long values.

compare(int, int) method

Compares two integer 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, or 1 when v2 is less than v1.

compare(long, long) method

Compares 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, or 1 when v2 is less than v1.

divide method

Divides two values and places the result in self.

Syntax

```
final long Unsigned64.divide(long v1, long v2)
```

Parameters

- **v1** The first operand
- **v2** The second operand

Returns

The first operand divided by the second operand.

multiply method

Multiplies two values together and places the result in self.

Syntax

```
final long Unsigned64.multiply(long v1, long v2)
```

Parameters

- **v1** The first operand.
- **v2** The second operand.

Returns

The product of v1 and 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 quotient 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 remainder, represented as a long integer.

remainder(long, long, long) method

Returns the remainder when one value multiplied by a given quotient 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 quotient.

Returns

The remainder, represented as a long integer.

subtract method

Subtracts two values and places the result in self.

Syntax

```
final long Unsigned64.subtract(long v1, long v2)
```

Parameters

- **v1** The first operand.
- **v2** The second operand.

Returns

The result of v2 being subtracted from v1.

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 object.
isNull method	Determines if the UUIDValue object is null.
set method	Sets the UUIDValue object with a String value.
setNull method	Sets the UUIDValue object to null.

Remarks

Such entities 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.

A Connection object can be used to create a UUIDValue using the createUUIDValue method.

See also

- "Connection.createUUIDValue method [UltraLiteJ]" on page 106

getString method

Returns the String representation of the UUIDValue object.

Syntax

```
String UUIDValue.getString() throws ULjException
```

Returns

The String value.

isNull method

Determines if the UUIDValue object is null.

Syntax

```
boolean UUIDValue.isNull()
```

Returns

True if the object is null; otherwise, returns false.

set method

Sets the UUIDValue object 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 object to null.

Syntax

```
void UUIDValue.setNull() throws ULjException
```

Index

A

B

C

D

E

F

G

H

P

R

S

T

U

V

W

Z

