UltraLite®J

June 2008

Version 11.0.0

Copyright and trademarks

Contents

About this book

Subject

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

Audience

This book is intended for application developers who want to use a Java-based database for data storage and synchronization.

About the SQL Anywhere documentation

The complete SQL Anywhere documentation is available in three formats that contain identical information.

- **HTML Help** The online Help contains the complete SQL Anywhere documentation, including the books and the context-sensitive help for SQL Anywhere tools. The online Help is updated with each release of the product.

  If you are using a Microsoft Windows operating system, the online Help is provided in HTML Help (CHM) format. To access the documentation, choose **Start** » **Programs** » **SQL Anywhere 11** » **Documentation** » **Online Books**.

  The administration tools use the same online documentation for their Help features.

- **Eclipse** On Unix platforms, the complete online Help is provided in Eclipse format. To access the documentation, run *sadoc* from the *bin32* or *bin64* directory of your SQL Anywhere 11 installation.

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

  To access the PDF documentation on Microsoft Windows operating systems, choose **Start** » **Programs** » **SQL Anywhere 11** » **Documentation** » **Online Books - PDF Format**.

  To access the PDF documentation on Unix operating systems, use a web browser to open *install-dir/documentation/en/pdf/index.html*.

About the books in the documentation set

SQL Anywhere documentation

The SQL Anywhere documentation consists of the following books:

- **SQL Anywhere 11 - Introduction** This book introduces SQL Anywhere 11—a comprehensive package that provides data management and data exchange, enabling the rapid development of database-powered applications for server, desktop, mobile, and remote office environments.
- **SQL Anywhere 11 - Changes and Upgrading** This book describes new features in SQL Anywhere 11 and in previous versions of the software.
- **SQL Anywhere Server - Database Administration** This book describes how to run, manage, and configure SQL Anywhere databases. It describes database connections, the database server, database files, backup procedures, security, high availability, and replication with Replication Server, as well as administration utilities and options.
- **SQL Anywhere Server - Programming** This book describes how to build and deploy database applications using the C, C++, Java, PHP, Perl, Python, and .NET programming languages such as Visual Basic and Visual C#. A variety of programming interfaces such as ADO.NET and ODBC are described.
- **SQL Anywhere Server - SQL Reference** This book provides reference information for system procedures, and the catalog (system tables and views). It also provides an explanation of the SQL Anywhere implementation of the SQL language (search conditions, syntax, data types, and functions).

- **SQL Anywhere Server - SQL Usage** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- **MobiLink - Getting Started** This book introduces MobiLink, a session-based relational-database synchronization system. MobiLink technology allows two-way replication and is well suited to mobile computing environments.
- **MobiLink - Client Administration** This book describes how to set up, configure, and synchronize MobiLink clients. MobiLink clients can be SQL Anywhere or UltraLite databases. This book also describes the Dbmlsync API, which allows you to integrate synchronization seamlessly into your C++ or .NET client applications.
- **MobiLink - Server Administration** This book describes how to set up and administer MobiLink applications.
- **MobiLink - Server-Initiated Synchronization** This book describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization or other remote actions from the consolidated database.
- **QAnywhere** This book describes QAnywhere, which is a messaging platform for mobile and wireless clients, as well as traditional desktop and laptop clients.
- **SQL Remote** This book describes the SQL Remote data replication system for mobile computing, which enables sharing of data between a SQL Anywhere consolidated database and many SQL Anywhere remote databases using an indirect link such as email or file transfer.
- **UltraLite - Database Management and Reference** This book introduces the UltraLite database system for small devices.
- **UltraLite - C and C++ Programming** This book describes UltraLite C and C++ programming interfaces. With UltraLite, you can develop and deploy database applications to handheld, mobile, or embedded devices.
- **UltraLite - M-Business Anywhere Programming** This book describes UltraLite for M-Business Anywhere. With UltraLite for M-Business Anywhere you can develop and deploy web-based database applications to handheld, mobile, or embedded devices, running Palm OS, Windows Mobile, or Windows.
- **UltraLite - .NET Programming** This book describes UltraLite.NET. With UltraLite.NET you can develop and deploy database applications to computers, or handheld, mobile, or embedded devices.
- **UltraLiteJ** This book describes UltraLiteJ. With UltraLiteJ, you can develop and deploy database applications in environments that support Java. UltraLiteJ supports BlackBerry smartphones and Java SE environments. UltraLiteJ is based on the iAnywhere UltraLite database product.
- **Error Messages** This book provides a complete listing of SQL Anywhere error messages together with diagnostic information.

Documentation conventions

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

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

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

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

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

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

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

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

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

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

  RELEASE SAVEPOINT [*savepoint-name*]

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

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

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

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

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

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

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

Operating system conventions

SQL Anywhere runs on a variety of platforms, including Windows, Windows Mobile, Unix, Linux, and Mac OS X. To simplify references to operating systems, the documentation groups the supported operating systems as follows:

- **Windows** The Microsoft Windows family of operating systems for desktop and laptop computers. The Windows family includes Windows Vista and Windows XP.
- **Windows Mobile** Windows Mobile provides a Windows user interface and additional functionality, such as small versions of applications like Word and Excel. Windows Mobile is most commonly used on mobile devices.

Limitations or variations in SQL Anywhere are commonly based on the underlying operating system, and seldom on the particular variant used (Windows Mobile).

- **Unix** Unless specified, Unix refers to Linux, Mac OS X, and Unix platforms.

File name conventions

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

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

- **install-dir** During the installation process, you choose where to install SQL Anywhere. The documentation refers to this location using the convention *install-dir*.

  After installation is complete, the environment variable SQLANY11 specifies the location of the installation directory containing the SQL Anywhere components (*install-dir*).

  For example, the documentation may refer to a file as *install-dir\readme.txt*. On Windows platforms, this reference is equivalent to *%SQLANY11%\readme.txt*. On Unix platforms, this reference is equivalent to *$SQLANY11/readme.txt*.

  For more information about the default location of *install-dir*, see "SQLANY11 environment variable" [*SQL Anywhere Server - Database Administration*].

- **Uppercase and lowercase directory names** On Windows, directory names often use mixed case. References to directory names can use any case, since the Windows file system is not case sensitive.

  Unix file systems are case sensitive. The use of mixed-case is less common.

  The SQL Anywhere installation program follows operating system conventions for its directory structure. On Windows, the installation contains directories such as *Bin32* and *Documentation*. On Unix, these directories are called *bin32* and *documentation*.

  The documentation often uses the mixed case forms of directory names. Usually, you can convert a mixed case directory name to lowercase for the equivalent directory name on Unix platforms. For example, the directory *MobiLink* is *mobilink* on Unix platforms.

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

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

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

  For example, on Windows, the network database server is *dbsrv11.exe*. On Unix, Linux, and Mac OS X, it is *dbsrv11*.

- **samples-dir** The installation process allows you to choose where to install the samples that are included with SQL Anywhere, and the documentation refers to this location using the convention *samples-dir*.

  After installation is complete, the environment variable SQLANYSAMP11 specifies the location of the directory containing the samples (*samples-dir*). From the Windows **Start** menu, choosing **Programs** » **SQL Anywhere 11** » **Sample Applications And Projects** opens a Windows Explorer window in this directory.

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

- **Environment variables** The documentation refers to setting environment variables. On Windows, environment variables are referred to using the syntax *%ENVVAR%*. On Unix, Linux, and Mac OS X, environment variables are referred to using the syntax *$ENVVAR* or *${ENVVAR}*.

  Unix, Linux, and Mac OS X environment variables are stored in shell and login startup files, such as *.cshrc* or *.tcshrc*.

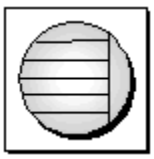
Graphic icons

The following icons are used in this documentation.

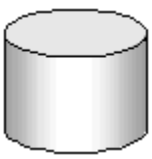
- A client application.

- A database server, such as Sybase SQL Anywhere.

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

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

- A programming interface.

Contacting the documentation team

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

To submit your comments and suggestions, send an email to the SQL Anywhere documentation team at iasdoc@sybase.com. Although we do not reply to emails, your feedback helps us to improve our documentation, so your input is welcome.

Finding out more and requesting technical support

Additional information and resources, including a code exchange, are available at the Sybase iAnywhere Developer Community at http://www.sybase.com/developer/library/sql-anywhere-techcorner.

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

When you write to one of these newsgroups, always provide detailed information about your problem, including the build number of your version of SQL Anywhere. You can find this information by running the following command: **dbeng11 -v**.

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

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

Newsgroup disclaimer

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

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

Part I. Using UltraLiteJ

CHAPTER 1

Introduction to UltraLiteJ

Contents

UltraLiteJ overview

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

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

To redistribute UltraLiteJ, you can add Java Archive (JAR) files to your applications. UltraLiteJ is supported on Java ME, Java SE 1.5 or later, and BlackBerry smartphones running OS 4.1 and later.

UltraLiteJ features

Database store

UltraLiteJ supports databases in memory (non-persistent) and in device storage (persistent). For Java SE, the file system is device storage. For BlackBerry smartphones, it is the BlackBerry object store.

Transactions

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

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

Checkpoint and recovery

UltraLiteJ provides both automatic and manual checkpoint capabilities. When set to automatic, commit statements cause table rows and indexes to be updated; otherwise, checkpoints are invoked using the checkpoint method of the Connection interface.

Concurrency and locking

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

- **Locking** Two different connections cannot modify the same row at the same time. If two connections attempt to operate on the same row, one connection is blocked until the other connection finishes.
- **Visibility** One connection's operation on the database is immediately visible to other connections.

Cache management

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

You can configure the size of the UltraLiteJ cache.

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

Encryption

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

Built-in change tracking

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

HTTP and HTTPS communication

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

Synchronization publications

To make more efficient use of network resources, UltraLiteJ provides a publication model that lets you synchronize selected tables from your database.

Character sets and collations

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

UltraLiteJ limitations

General limitations

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

Feature	Limitation
Database page size	Minimum 256 bytes, maximum 32 KB.
Tables per database	Maximum 32 K.
Row size	Row contents (after possible compression) must not exceed the database page size.
Blob size	Maximum 2^{24} bytes.
Data types	See “Domain interface” on page 153.
BlackBerry SD cards	Not supported.

UltraLiteJ database stores

Supported database stores

Following is a list of database store types supported by UltraLiteJ:

Database store type	Platform support
File system store	Java SE
RIM object store (RIM11)	Java ME
Record store	Java ME
Non-persistent store	All Java platforms

BlackBerry object store limitations

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

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

UltraLiteJ requires an object handle to store a value in the database. For example, a table row with ten columns and two indexes requires a minimum of twelve object handles.

To permit a larger database store, UltraLiteJ requires a persistent object handle for each database page.

Persistent store configurations and recovery

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

- **Non-persistence** Creating a NonPersist object configures a database store that only exists in memory. The database is created at startup, used while the application is running, then discarded when the application closes. When the application closes, all data contained in the non-persistent store is deleted.
- **Write-at-end persistence** When write-at-end persistence is enabled using the setWriteAtEnd method of a persistent configuration object, data is only written to the database when the connection is released. While this form of persistence improves the overall speed of transactions, data could be lost if the application terminates abnormally.
- **Shadow paging persistence** Shadow paging is the strongest form of persistence. It can be enabled using the setShadowPaging method of a persistent configuration object. When enabled on startup, it

provides database recovery to the state at the last checkpoint, even if the application terminates unexpectedly.

- **Simple paging persistence** When shadow paging is not enabled, a simple paging implementation of persistence is used. All data is written to the same page from which data is read. The database is considered corrupt from the time updating starts to the time before updating completes. A corrupt database can be detected during startup, but it is not recoverable. In comparison to shadow paging, simple paging uses significantly less memory and improves performance.

Data synchronization

UltraLiteJ is able to synchronize data with MobiLink 11. When synchronizing with MobiLink, you must use the -x MobiLink server option.

UltraLiteJ supports:

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

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

Concurrent synchronization

Normally only one thread is allowed in the UltraLiteJ runtime at a time. An exception to this rule occurs during synchronization. While one connection is performing a synchronization operation, other connections can access the UltraLiteJ runtime with certain restrictions:

- Schema operations (disableSynchronization, dropDatabase, dropTable, dropPublication, enableSynchronization, renameTable, or schemaCreateBegin) can not be performed on database Connection objects during synchronization.
- While a synchronize operation is taking place, no other thread can call the synchronize method for the database being synchronized.

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

CHAPTER 2

Developing UltraLiteJ applications

Contents

This section provides an introduction to the UltraLiteJ Application Programming Interface (API).

Introduction to UltraLiteJ development

UltraLiteJ adds basic database functionality to your Java applications and is compatible with BlackBerry smartphones. The UltraLiteJ API contains all the methods required to connect to an UltraLiteJ database, perform schema operations, and maintain data using SQL statements. Advanced operations, such as data encryption and synchronization, are also supported.

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

Creating a basic UltraLiteJ application

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

1. Create a new Configuration object.

   A Configuration object defines where the database is located. It also sets the username and password. Variations of the Configuration object are available for different devices and for non-persistent database stores. See "Configuration interface" on page 119.

2. Create a new Connection object.

   A Connection object connects to an UltraLiteJ database using the specifications defined in the Configuration object. If the database does not exist, it is created automatically. See "Connection interface" on page 121.

3. Apply TableSchema, IndexSchema, ForeignKeySchema objects.

   The schema methods provided by the Connection object allow you to create tables, columns, indexes, and foreign keys. See "schemaCreateBegin function" on page 139.

4. Generate PreparedStatement objects.

   A PreparedStatement object queries the database associated with the Connection object. It accepts supported SQL statements, which are passed as Strings. With a PreparedStatement object, you can update the contents of the database. See "PreparedStatement interface" on page 173.

5. Generate ResultSet objects.

   ResultSet objects are created when the Connection object executes a PreparedStatement that contains a SQL SELECT statement. With a ResultSet object, you can view the table contents of the database. See "ResultSet interface" on page 177.

Setting up an UltraLiteJ application

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

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

```
import ianywhere.ultralitej.*;
```


Accessing an UltraLiteJ database store

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

Implementations of the Configuration object

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

- **Object stores** Supported with a ConfigObjectStore.
- **Record stores** Supported with a ConfigRecordStore.
- **Persistent file stores** Supported with a ConfigFile.
- **Non-persistent stores** Supported with a ConfigNonPersistent.

Properties of the Connection object

- **Transactions** In the UltraLiteJ API, there is no AutoCommit mode. Each transaction must be followed by the commit method of the Connection. They can be rolled back with a rollback statement.
- **Prepared SQL statements** Methods are provided to handle the SQL statement, which are executed from the Connection.
- **Synchronization** A set of objects governing synchronization is accessed from the Connection.
- **Table operations** UltraLiteJ database tables are accessed and maintained using methods of the Connection.

Creating a new UltraLiteJ database

An UltraLiteJ database can only be created using the API. You can not create a new database using Sybase Central or the UltraLite command line utilities.

To create a database

1. Create a new Configuration that references the database name.

   The proper syntax depends on the Java platform and the client device. In the following examples, config is the name of the Configuration object and DBname.ulj is the name of the new database.

   For J2ME BlackBerry devices:

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

   For all other J2ME devices:

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

   For J2SE devices:

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

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

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

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

   The following example sets the password to my_password:

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

3. Create a new Connection using the following syntax:

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

   The createDatabase method finalizes the database creation process and connects to the database. After this method is called, you can perform schema and data operations but you can no longer change the name, password, or page size of the database.

Connecting to an existing database

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

To connect to an existing database

1. Create a new Configuration that references the name of the database.

   The proper syntax depends on the Java platform and the client device. In the following examples, config is the name of the Configuration object and DBname.ulj is the name of the database.

   For J2ME BlackBerry devices:

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

   For all other J2ME devices:

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

   For J2SE devices:

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

   Alternatively, you can connect to a non-persistent database supported by all platforms with the following syntax:

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

2. Specify the database password using the setPassword method.

   The following example assumes that the password is my_password:

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

3. Create a new Connection using the following code:

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


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

Disconnecting from a database

Use the release method of the DatabaseManager class to disconnect from the UltraLiteJ database. The release method closes the Connection object and all properties associated with it.

See also

- "Example: Creating a database" on page 31
- "DatabaseManager class" on page 144

Performing schema operations

The UltraLiteJ API contains schema methods that allow you to create tables, columns, indexes, and keys in an UltraLiteJ database. This section explains how to perform schema operations.

Types of schema objects

- **Table schema** Methods in the Connection interface are used to access table properties. The schema is accessed directly using the TableSchema interface. See "TableSchema interface" on page 235.
- **Column schema** Methods in the TableSchema interface are used to access column properties. The schema is accessed directly using the ColumnSchema interface. See "ColumnSchema interface" on page 103.
- **Index schema** Methods in the TableSchema interface are used to access indexes. The schema is accessed directly using the IndexSchema interface. See "IndexSchema interface" on page 170.
- **Foreign key schema** Methods in the Connection interface are used to access foreign keys. The schema is accessed directly using the ForeignKeySchema interface. See "ForeignKeySchema interface" on page 168.

Creating new tables, columns, indexes, and keys

All table, column, index, and key operations must be performed using the API. These operations can only be performed when the connected database is in schema creation mode.

To perform schema operations

1. Connect to an UltraLiteJ database.

   This example assumes that the database is connected to the Connection, conn. For details on how to connect to an UltraLiteJ database, see "Accessing an UltraLiteJ database store" on page 13.

2. Put the database into schema creation mode using the following code:

   ```
   conn.schemaCreateBegin();
   ```

   Schema creation mode prohibits data operations and locks out additional connections to the database.

3. Perform all table, column, index, and key operations.

   The following procedures demonstrate how to create a new Employee table that contains an integer column named emp_number. The emp_number column acts as the primary index for the Employee table and also acts as a foreign key, referencing an integer column named access_number in a Security table.

   - **To create a new table** Use the createTable method on the Connection to define the name of the table and assign the result to a TableSchema:

     ```
     TableSchema table_schema = conn.createTable("Employee");
     ```

     Assigning the result to a TableSchema allows you to perform more detailed schema operations to the table.

   - **To add a column to a table:** Use the createColumn method on the TableSchema to define the column name and type:


```
table_schema.createColumn("emp_number", Domain.INTEGER);
```

- **To assign a primary index to a column:**

  a. Use the createPrimaryIndex method on the TableSchema and assign the result to an IndexSchema:

  ```
  IndexSchema index_schema =
      table_schema.createPrimaryIndex("prime_keys");
  ```

  b. Use the addColumn method on the IndexSchema to specify the primary index column and sort order:

  ```
  index_schema.addColumn("emp_number", IndexSchema.ASCENDING);
  ```

- **To assign a foreign key to a column:** This procedure assumes that a Security table with an access_number integer column exists in the database.

  a. Use the createForeignKey method on the Connection to specify the tables involved:

  ```
  ForeignKeySchema foreign_key_schema = conn.createForeignKey(
      "Employee",
      "Department",
      "fk_emp_to_dept"
  );
  ```

  b. Use the addColumnReference method to specify the two columns involved:

  ```
  foreign_key_schema.addColumnReference("emp_number",
  "access_number");
  ```

4. Put the database out of schema creation mode using the following code:

```
conn.schemaCreateComplete();
```

See also

- “Connection interface” on page 121
- “ColumnSchema interface” on page 103
- “ForeignKeySchema interface” on page 168
- “IndexSchema interface” on page 170
- “TableSchema interface” on page 235

Accessing and manipulating data using SQL

Supported SQL statements

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

SQL Statement	Notes and Restrictions:
COMMIT	See "UltraLite COMMIT statement" [*UltraLite - Database Management and Reference*].
ROLLBACK	See "UltraLite ROLLBACK statement" [*UltraLite - Database Management and Reference*].
INSERT	See "UltraLite INSERT statement" [*UltraLite - Database Management and Reference*].
UPDATE	See "UltraLite UPDATE statement" [*UltraLite - Database Management and Reference*].
DELETE	See "UltraLite DELETE statement" [*UltraLite - Database Management and Reference*].
SELECT	For a general description of the SELECT statement, see "UltraLite SELECT statement" [*UltraLite - Database Management and Reference*]. Following restrictions apply: • The HAVING clause is not supported. • The DISTINCT expression in an aggregate function is not supported. For example, UltraLiteJ supports COUNT(expression) but not COUNT (DISTINCT expression). • The SQLCODE function is not supported. • The ACOS, ASIN, ATAN, ATAN2, and POWER mathematical functions are not supported. • The CURRENT TIMESTAMP, CURRENT TIME, and CURRENT DATE functions are not supported. Use TODAY(*) or NOW(*) instead.

Data manipulation using INSERT, UPDATE, DELETE

You can perform SQL data manipulation 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 SQL statement. For example, the first **?** is referred as parameter 1, and the second as parameter 2.

To INSERT a row in a table

1. Prepare a new SQL statement as a String.

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

2. Assign the String to the PreparedStatement.

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

3. Assign input parameter values for the statement.

   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();
   ```

To UPDATE a row in a table

1. Prepare a new SQL statement as a String.

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

2. Assign the String to the PreparedStatement.

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

3. Assign input parameter values for the statement.

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

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

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

4. Execute the statement.

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

5. Close the PreparedStatement to free resources.

```
inserter.close()
```

6. Commit all changes to the database.

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

To DELETE a row in a table

1. Prepare a new SQL statement as a String.

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

2. Assign the String to the PreparedStatement.

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

3. Assign input parameter values for the statement.

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

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

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

4. Execute the statement.

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

5. Close the PreparedStatement to free resources.

```
inserter.close()
```

6. Commit all changes to the database.

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

Retrieving data using SELECT

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

Navigating the ResultSet object

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

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

Retrieving data using a ResultSet

To SELECT data from a database

1. Prepare a new SQL statement as a String.

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

2. Assign the String to the PreparedStatement.

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

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

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

4. Traverse through the ResultSet and retrieve the data.

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

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

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

5. Close the ResultSet to free resources.

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

6. Close the PreparedStatement to free resources.

```
select_statement.close()
```

Managing transactions using COMMIT, ROLLBACK

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

To commit a transaction, use the commit method.

To roll back a transaction, use the rollback method.

See also

- “commit function” on page 126
- “rollback function” on page 138

Encrypting and obfuscating data

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

To encrypt or obfuscate data in the database

1. Create a class that implements the EncryptionControl interface.

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

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

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

   Your class should now look similar to the following:

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

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

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

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

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


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

4. Connect to the database.

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

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

See also

- “Example: Obfuscating data” on page 44
- “Example: Encrypting data” on page 47
- “EncryptionControl interface” on page 166

Synchronizing with MobiLink

Using UltraLiteJ as a MobiLink client

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

1. Instantiate a syncParms object, which contains information about the consolidated database (name of the server, port number), name of the database to be synchronized, and the definition of the tables to be synchronized.
2. Call the synchronize method from the connection object with the syncParms object to carry out the synchronization.

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

See also

- "SyncParms class" on page 215
- "SyncResult class" on page 229

Example

This example demonstrates how to synchronize data with an UltraLiteJ application.

To start the MobiLink server with SQL Anywhere 11 CustDB as the consolidated database, run *start_ml.bat* from the *samples-dir\UltraLiteJ* directory.

```
package ianywhere.ultralitej.demo;
import ianywhere.ultralitej.*;
/**
 * 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 );
            conn.schemaCreateBegin();

            TableSchema table_schema = conn.createTable( "ULCustomer" );
            table_schema.createColumn( "cust_id", Domain.INTEGER );
```



```
            table_schema.createColumn( "cust_name", Domain.VARCHAR, 30 );
            IndexSchema index_schema =
table_schema.createPrimaryIndex( "prime_keys" );
            index_schema.addColumn( "cust_id", IndexSchema.ASCENDING );

            conn.schemaCreateComplete();

            //
            // Synchronization
            //

            // Version set for MobiLink 11.0.x
           SyncParms syncParms = conn.createSyncParms( SyncParms.HTTP_STREAM,
"50", "custdb 11.0" );
            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 );
        }
    }
}
```

Network protocol options for UltraLiteJ synchronization streams

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

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

Setting up an HTTP network protocol

An HTTP network protocol is set with the StreamHTTPParms interface in the UltraLiteJ API. Use the interface methods to specify the network protocol options defined on the MobiLink server. For a complete list of network options, see “StreamHTTPParms interface” on page 201.

Setting up an HTTPS network protocol

An HTTPS network protocol is set with the StreamHTTPSParms interface in the UltraLiteJ API. Use the interface methods to specify the network protocol options defined on the MobiLink server. For a complete list of network options, see “StreamHTTPSParms interface” on page 205.

Synchronizing the CustDB application

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

Finding and deploying the application

The installation of UltraLiteJ includes a sample BlackBerry application that relates to the CustDB database. The application is named CustDB and the source code and related files are found in the *Samples\CustDB* directory within the UltraLiteJ installation directory. The default installation directory is *SQL Anywhere 11\ultralitej*. Also included in the CustDB directory are the project files that can be opened with the Research In Motion (RIM) JDE.

The CustDB application can be downloaded directly onto your BlackBerry (to see how it works) by directing your BlackBerry browser to this URL:

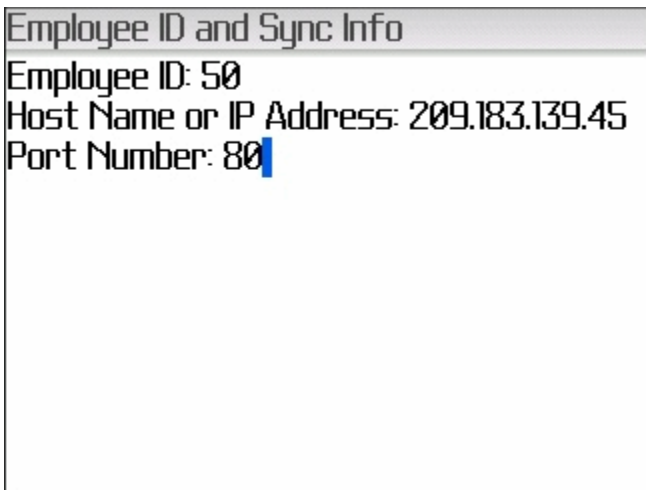
```
http://ultralitej.sybase.com/
```

Files related to CustDB application

- ***CustDB.java*** This file contains all of the basic database access methods. These methods include creating and connecting to databases, inserting, deleting and updating orders. This file contains many of the database calls to communicate to the back-end server.
- ***CustIDJoin.java and ProdIDJoin.java*** These files contains the code to implement a join filter using UltraLiteJ.
- ***SchemaCreator.java*** This file contains the code to create tables on the device using UltraLiteJ.

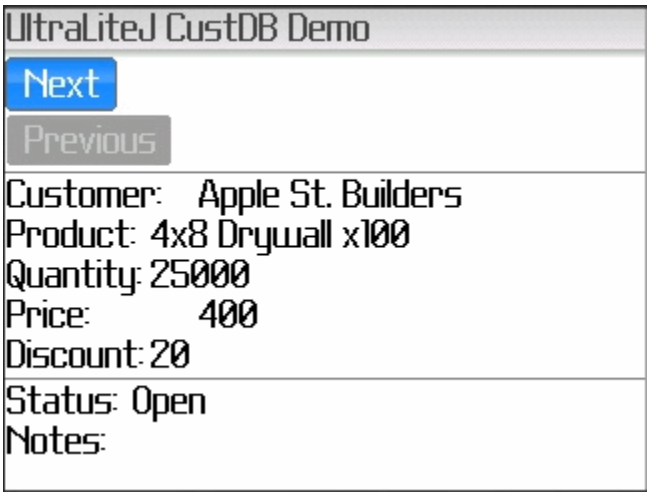
Using the CustDB application

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.



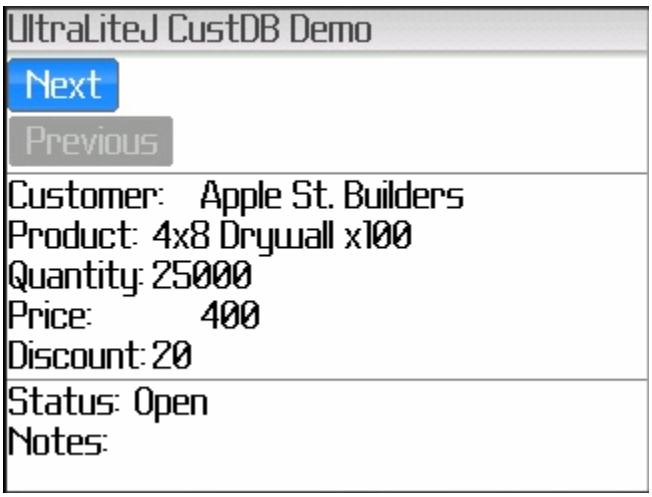
Once these values are specified and the settings are saved (**Menu** » **Save**), the application synchronizes with the specified server. The application only downloads orders from the server that match the Employee ID corresponding to the specified employee number (50). Only orders that are still open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

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



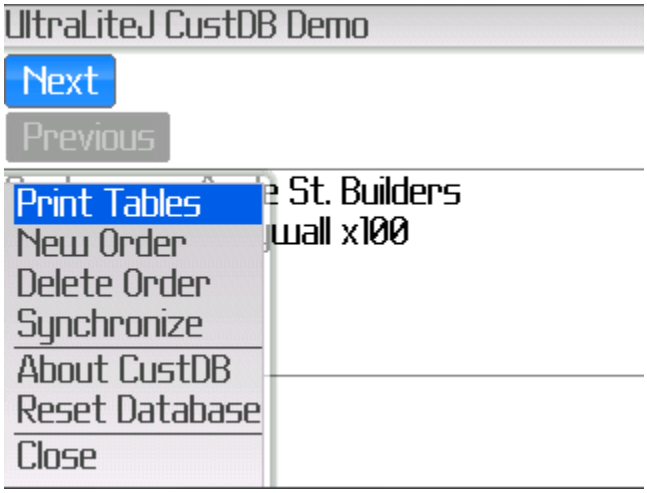
Once on this screen you can add notes to the order, or change the status of the order (to either Approved or Denied). You can navigate through the orders using the **Next** and **Previous** buttons.

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



You may enter the quantity and discount values you require.

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



Deploying UltraLiteJ applications

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

Deployment type	Required files
BlackBerry smartphone	*J2meRim11\UltraLiteJ.cod* *J2meRim11\UltraLiteJ.jad*[1]
J2ME	*J2me11\UltraLiteJ.jar*
J2SE	*J2se\UltraLiteJ.jar*

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

Coding examples

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

All coding examples can be found in the *samples-dir/UltraLiteJ* directory. Create backup copies of the original source code before manipulating the file contents.

Example: Demo class

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

```
// *********************************************************************
// Copyright 2006-2008 iAnywhere Solutions, Inc.  All rights reserved.
// *********************************************************************
package ianywhere.ultralitej.demo;

//import java.io.*;
import ianywhere.ultralitej.*;
//import ianywhere.ultralitej.implementation.*;

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

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

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



```
    }

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

    /** Display message for an exception.
     * @param exc ULjException containing message
     */
    public static void displayException( ULjException exc )
    {
        display( exc.getMessage() );
    }

    /** Display message for an exception.
     * @param exc ULjException containing message
     */
    public static void displayExceptionFull( ULjException exc )
    {
        display( exc.getMessage() );
    }
}
```

Example: Creating a database

This example demonstrates how to create a file system database store in a J2SE 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.

Notes:

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

```
package ianywhere.ultralitej.demo;
import ianywhere.ultralitej.*;
/**
 * CreateDb: sample program to demonstrate Database creation.
 */
public class CreateDb
{
    /**
     * mainline for program.
     *
     * @param args command-line arguments
     *
```

```
     */
    public static void main
        ( String[] args )
    {
        try {
            Configuration config =
DatabaseManager.createConfigurationFile( "Demo1.ulj" );

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

            TableSchema table_schema = conn.createTable( "Employee" );
            table_schema.createColumn( "number", Domain.INTEGER );
            table_schema.createColumn( "last_name", Domain.VARCHAR, 32 );
            table_schema.createColumn( "first_name", Domain.VARCHAR, 32 );
            table_schema.createColumn( "age", Domain.INTEGER );
            table_schema.createColumn( "dept_no", Domain.INTEGER );
            IndexSchema index_schema =
table_schema.createPrimaryIndex( "prime_keys" );
            index_schema.addColumn( "number", IndexSchema.ASCENDING );

            table_schema = conn.createTable( "Department" );
            table_schema.createColumn( "dept_no", Domain.INTEGER );
            table_schema.createColumn( "name", Domain.VARCHAR, 50 );
            index_schema = table_schema.createPrimaryIndex( "prime_keys" );
            index_schema.addColumn( "dept_no", IndexSchema.ASCENDING );

            ForeignKeySchema foreign_key_schema =
conn.createForeignKey( "Employee", "Department", "fk_emp_to_dept" );
            foreign_key_schema.addColumnReference( "dept_no", "dept_no" );

            conn.schemaCreateComplete();

            conn.release();

            Demo.display( "CreateDb completed successfully" );

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

Example: Inserting rows

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

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.

```
package ianywhere.ultralitej.demo;
import ianywhere.ultralitej.*;
/**
 * LoadDb -- sample program to demonstrate loading a Database.
```



```
 */
public class LoadDb
{
    /**
     * Add a Department row.
     * @param conn connection to Database
     * @param dept_no department number
     * @param dept_name department name
     */
    private static void addDepartment( PreparedStatement inserter, int
dept_no, String dept_name )
        throws ULjException
    {
        inserter.set( 1 /* "dept_no" */, dept_no );
        inserter.set( 2 /* "name" */, dept_name );
        inserter.execute();
    }

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

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

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

            inserter = conn.prepareStatement(
```

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

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

Example: Reading a table

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

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").

```
package ianywhere.ultralitej.demo;
import ianywhere.ultralitej.*;
/**
 * ReadSeq -- sample program to demonstrate reading a Database table
 * sequentially.
 */
public class ReadSeq
{
    /**
     * mainline for program.
     *
     * @param args command-line arguments
     *
     */
    public static void main
        ( String[] args )
    {
        try {
            Configuration config =
DatabaseManager.createConfigurationFile( "Demo1.ulj" );
            Connection conn = DatabaseManager.connect( config );
            PreparedStatement stmt = conn.prepareStatement( "SELECT * FROM
Employee ORDER BY number" );
            ResultSet cursor = stmt.executeQuery();
            for( ; cursor.next(); ) {
                /* Can't access columns by name because no meta data */
                int emp_no = cursor.getInt( 1 /* "number" */ );
                String last_name = cursor.getString( 2 /* "last_name" */ );
```



```
                String first_name = cursor.getString( 3 /* "first_name" */ );
                int age = cursor.getInt( 4 /* "age" */ );
                Demo.display( first_name + ' ' + last_name );
                Demo.display( "  empl. no = "
                            , Integer.toString( emp_no )
                            , "  age = "
                            , Integer.toString( age ) );
            }
            cursor.close();
            stmt.close();
            conn.release();
        } catch( ULjException exc ) {
            Demo.displayException( exc );
        }
    }
}
```

Example: Filtering rows to match criteria

This example demonstrates how to read rows where the age field is greater than 50.

Notes:

- To filter the rows in a table according to a specific criteria, the application must contain a class that tests the criteria.
- The filter class returns a boolean value, indicating if the criteria has been met.

```
package ianywhere.ultralitej.demo;
import ianywhere.ultralitej.*;
/**
 * ReadFilter -- sample program to demonstrate reading a Database table
sequentially
 * and filtering rows.
 * <p> Rows are accepted when the age column has a value
 * greater than 50.
 */
public class ReadFilter
{
    /**
     * mainline for program.
     *
     * @param args command-line arguments
     *
     */
    public static void main
        ( String[] args )
    {
        try {
            Configuration config =
DatabaseManager.createConfigurationFile( "Demo1.ulj" );
            Connection conn = DatabaseManager.connect( config );
            PreparedStatement stmt = conn.prepareStatement(
                "SELECT * FROM Employee WHERE age > 50 ORDER BY number"
            );
            ResultSet cursor =  stmt.executeQuery();
            for( ; cursor.next(); ) {
                int emp_no = cursor.getInt( 1 /* "Employee.number" */ );
               String last_name = cursor.getString( 2 /* "Employee.last_name"
```

```
*/ );
                String first_name = cursor.getString( 3 /*
"Employee.first_name" */ );
                int age = cursor.getInt( 4 /* "Employee.age" */ );
                System.out.println( first_name + ' ' + last_name );
                System.out.print( "  empl. no = " );
                System.out.print( emp_no );
                System.out.print( "  age = " );
                System.out.println( age );
            }
            cursor.close();
            stmt.close();
            conn.release();
            Demo.display( "ReadFilter completed successfully" );
        } catch( ULjException exc ) {
            Demo.displayException( exc );
        }
    }

    /**
     * Filter class
     *
     */
    private static class AgeFilter
        implements Filter
    {
        /**
         * Accept child rows when age > 50.
         *
         * @param child The child ResultSet.
         * @return true, if age > 50.
         *
         */
        public boolean evaluate( ResultSet child )
        {
            try {
                ResultSet cursor = (ResultSet) child;
                int age = cursor.getInt( "Employee.age" );
                return age > 50;
            } catch( ULjException exc ) {
                Demo.displayException( exc );
                return false;
            }
        }
    }
}
```

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.

```
package ianywhere.ultralitej.demo;
import ianywhere.ultralitej.*;
/**
 * ReadInnerJoin -- sample program to demonstrate reading the Employee table
```



```
 * and joining to each row the corresponding Department row.
 */
public class ReadInnerJoin
{
    /**
     * mainline for program.
     *
     * @param args command-line arguments
     *
     */
    public static void main
        ( String[] args )
    {
        try {
            Configuration config =
DatabaseManager.createConfigurationFile( "Demo1.ulj" );
            Connection conn = DatabaseManager.connect( config );
            PreparedStatement stmt = conn.prepareStatement(
                "SELECT E.number, E.last_name, E.first_name, E.age,"
                + " E.dept_no, D.name"
                + " FROM Employee E"
                + " JOIN Department D ON E.dept_no = D.dept_no"
                + " ORDER BY E.number"
            );
            ResultSet cursor = stmt.executeQuery();
            for( ; cursor.next(); ) {
                /* Can't access columns by name because no meta data */
                int emp_no = cursor.getInt( 1 /* "E.number" */ );
                String last_name = cursor.getString( 2 /* "E.last_name" */ );
                String first_name = cursor.getString( 3 /* "E.first_name"
*/ );
                int age = cursor.getInt( 4 /* "E.age" */ );
                int dept_no = cursor.getInt( 5 /* "E.dept_no" */ );
                String dept_name = cursor.getString( 6 /* "D.name" */ );
                System.out.println( first_name + ' ' + last_name );
                System.out.print( "  empl. no = " );
                System.out.print( emp_no );
                System.out.print( "  dept = " );
                System.out.println( dept_no );
                System.out.print( "  age = " );
                System.out.print( age );
                System.out.println( ", " + dept_name );
            }
            cursor.close();
            stmt.close();
            conn.release();
            Demo.display( "ReadInnerJoin completed successfully" );
        } catch( ULjException exc ) {
            Demo.displayException( exc );
        }
    }
}
```

Example: Creating a sales database

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

```
package ianywhere.ultralitej.demo;
import ianywhere.ultralitej.*;
```

```
/**
 * CreateDb: sample program to demonstrate creation of simple sales Database
and
 * load it with some data.
 * <p>The program also illustrates the use of ordinals when inserting rows
into tables.
 */
public class CreateSales
{
    static int ORDINAL_INVOICE_INV_NO;
    static int ORDINAL_INVOICE_NAME;
    static int ORDINAL_INVOICE_DATE;

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

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

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

        conn.schemaCreateBegin();

        TableSchema table_schema = conn.createTable( "Product" );
        table_schema.createColumn( "prod_no", Domain.INTEGER );
        table_schema.createColumn( "prod_name", Domain.VARCHAR, 32 );
        table_schema.createColumn( "price", Domain.NUMERIC, 9, (short)2 );
        IndexSchema index_schema =
table_schema.createPrimaryIndex( "prime_keys" );
        index_schema.addColumn( "prod_no", IndexSchema.ASCENDING );

        table_schema = conn.createTable( "Invoice" );
        table_schema.createColumn( "inv_no", Domain.INTEGER );
        table_schema.createColumn( "name", Domain.VARCHAR, 50 );
        table_schema.createColumn( "date", Domain.DATE );
        index_schema = table_schema.createPrimaryIndex( "prime_keys" );
        index_schema.addColumn( "inv_no", IndexSchema.ASCENDING );

        table_schema = conn.createTable( "InvoiceItem" );
        table_schema.createColumn( "inv_no", Domain.INTEGER );
        table_schema.createColumn( "item_no", Domain.INTEGER );
        table_schema.createColumn( "prod_no", Domain.INTEGER );
        table_schema.createColumn( "quantity", Domain.INTEGER );
        table_schema.createColumn( "price", Domain.NUMERIC, 9, (short)2 );
        index_schema = table_schema.createPrimaryIndex( "prime_keys" );
        index_schema.addColumn( "inv_no", IndexSchema.ASCENDING );
        index_schema.addColumn( "item_no", IndexSchema.ASCENDING );

        conn.schemaCreateComplete();

        return conn;
    }
```



```
    /** Populate the Database.
     * @param conn connection to Database
     */
    private static void populateDatabase( Connection conn )
        throws ULjException
    {
        PreparedStatement ri_product = conn.prepareStatement(
            "INSERT INTO Product( prod_no, prod_name, price )
VALUES( ?, ?, ? )"
            );
        ORDINAL_PROD_NO = 1;
        ORDINAL_PROD_NAME = 2;
        ORDINAL_PROD_PRICE = 3;
        addProduct( ri_product, 2001, "blue screw", ".03" );
        addProduct( ri_product, 2002, "red screw", ".09" );
        addProduct( ri_product, 2004, "hammer", "23.99" );
        addProduct( ri_product, 2005, "vice", "39.99" );
        ri_product.close();

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

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

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

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

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

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

        conn.commit();
    }

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

```
     */
    public static void main
        ( String[] args )
    {
        try {
            Connection conn = createDatabase();
            populateDatabase( conn );

            conn.release();

            Demo.display( "CreateSales completed successfully" );

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

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

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

    /** Add a product row.
     * @param conn connection to Database
     * @param prod_no product number
     * @param prod_name product name
     * @param price selling price
     */
    private static void addProduct( PreparedStatement ri, int prod_no, String
prod_name, String price )
        throws ULjException
```



```
        {
            ri.set( ORDINAL_PROD_NO, prod_no );
            ri.set( ORDINAL_PROD_NAME, prod_name );
            ri.set( ORDINAL_PROD_PRICE, price );
            ri.execute();
        }
    }
```

Example: Aggregation and grouping

This example demonstrates UltraLiteJ support for the aggregation of results.

```
package ianywhere.ultralitej.demo;
import ianywhere.ultralitej.*;
/** Create a sales report to illustrate aggregation support.
 */
public class SalesReport
{
    /** Mainline.
     * @param args program arguments (not used)
     */
    public static void main( String[] args )
    {
        try {
            Configuration config =
DatabaseManager.createConfigurationFile( "Sales.ulj" );
            Connection conn = DatabaseManager.connect( config );
            PreparedStatement stmt = conn.prepareStatement(
                "SELECT inv_no, SUM( quantity * price ) AS total"
                + " FROM InvoiceItem"
                + " GROUP BY inv_no ORDER BY inv_no"
            );
            ResultSet agg_cursor = stmt.executeQuery();
            for( ; agg_cursor.next(); ) {
                int inv_no = agg_cursor.getInt( 1 /* "inv_no" */ );
                String total = agg_cursor.getString( 2 /* "total" */ );
                Demo.display( Integer.toString( inv_no ) + ' ' + total );
            }
            Demo.display( "SalesReport completed successfully" );
        } catch( ULjException exc ) {
            Demo.displayException( exc );
        }
    }
}
```

Example: Retrieving rows in an alternative order

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

```
package ianywhere.ultralitej.demo;
import ianywhere.ultralitej.*;

public class SortTransactions
{
    /** Mainline.
```

```
     * @param args program arguments (not used)
     */
    public static void main( String[] args )
    {
        try {
            Configuration config =
DatabaseManager.createConfigurationFile( "Sales.ulj" );
            Connection conn = DatabaseManager.connect( config );
            PreparedStatement stmt = conn.prepareStatement(
                "SELECT inv_no, prod_no, quantity FROM InvoiceItem"
                + " ORDER BY prod_no"
            );
            ResultSet cursor = stmt.executeQuery();
            for( ; cursor.next(); ) {
                /* Can't access columns by name because no meta data */
                int inv_no = cursor.getInt( 1 /* "inv_no" */ );
                int prod_no = cursor.getInt( 2 /* "prod_no" */ );
                int quantity = cursor.getInt( 3 /* "quantity" */ );
                Demo.display( Integer.toString( prod_no ) + ' '
                    + Integer.toString( inv_no ) + ' '
                    + Integer.toString( quantity )
                );
            }
            conn.release();
            Demo.display( "SortTransactions completed successfully" );
        } catch( ULjException exc ) {
            Demo.displayException( exc );
        }
    }
}
```

Example: Modifying table definitions

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

```
package ianywhere.ultralitej.demo;
import ianywhere.ultralitej.*;
/** Reorganize the Invoice table to have a name with an increased size
 *
 * <p>This shows a possible strategy which can be used to reorganize
 * tables, since UltraLiteJ has no table-altering API.
 * <p>The (contrived) example expands the name column to 100 characters.
 *
 */
public class Reorg
{
    /**
     * mainline for program.
     *
     * @param args command-line arguments
     *
     */
    public static void main
        ( String[] args )
    {
        try {
            Configuration config =
DatabaseManager.createConfigurationFile( "Sales.ulj" );
```



```
            Connection conn = DatabaseManager.connect( config );
            createNewInvoiceTable( conn );
            copyInvoicesToNewTable( conn );
            deleteOldInvoicesTable( conn );
            renameNewInvoicesTable( conn );
            enableSynchronizationForNewTable( conn );
            conn.release();
        } catch( ULjException exc ) {
            Demo.displayExceptionFull( exc );
        }
    }

    private static void createNewInvoiceTable( Connection conn )
        throws ULjException
    {
        conn.schemaCreateBegin();
        TableSchema table_schema = conn.createTable( "NewInvoice" );
        table_schema.createColumn( "inv_no", Domain.INTEGER );
       table_schema.createColumn( "name", Domain.VARCHAR, 100 );   // was 50
in old table
        table_schema.createColumn( "date", Domain.DATE );
        IndexSchema index_schema =
table_schema.createPrimaryIndex( "prime_keys" );
        index_schema.addColumn( "inv_no", IndexSchema.ASCENDING );
      table_schema.setNoSync( true );        // we don't want to sync inserts
yet
        conn.schemaCreateComplete();
    }

    private static void copyInvoicesToNewTable( Connection conn )
        throws ULjException
    {
        PreparedStatement inserter = conn.prepareStatement(
            "INSERT INTO NewInvoice( inv_no, name, \"date\" )
VALUES( ?, ?, ? )"
        );
        int ordinal_inv_no = 1;
        int ordinal_inv_name = 2;
        int ordinal_inv_date = 3;

        PreparedStatement stmt = conn.prepareStatement(
            "SELECT inv_no, name, \"date\" FROM Invoice"
        );
        ResultSet cursor = stmt.executeQuery();
        for( ; cursor.next(); ) {
            inserter.set( ordinal_inv_no, cursor.getInt( ordinal_inv_no ) );
            inserter.set( ordinal_inv_name,
cursor.getString( ordinal_inv_name ) );
            inserter.set( ordinal_inv_date,
cursor.getString( ordinal_inv_date ) );
            inserter.execute();
           // in memory-low conditions, we could delete the row from the old
table (specify
            // stopSynchronizationDelete, so these deletes would not
synchronize.
        }
        inserter.close();
        cursor.close();
        stmt.close
        conn.commit();
    }

    private static void deleteOldInvoicesTable( Connection conn )
        throws ULjException
```

```
    {
        conn.dropTable( "Invoice" );
    }

    private static void renameNewInvoicesTable( Connection conn )
        throws ULjException
    {
        conn.renameTable( "NewInvoice", "Invoice" );
    }

    private static void enableSynchronizationForNewTable( Connection conn )
        throws ULjException
    {
        conn.enableSynchronization( "Invoice" );
    }
}
```

Example: Obfuscating data

This example demonstrates a technique for obfuscating data in a database.

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

        conn.schemaCreateBegin();

        TableSchema table_schema = conn.createTable( "Product" );
        table_schema.createColumn( "prod_no", Domain.INTEGER );
        table_schema.createColumn( "prod_name", Domain.VARCHAR, 32 );
        table_schema.createColumn( "price", Domain.NUMERIC, 9, (short)2 );
        IndexSchema index_schema =
table_schema.createPrimaryIndex( "prime_keys" );
        index_schema.addColumn( "prod_no", IndexSchema.ASCENDING );

        conn.schemaCreateComplete();

        return conn;
    }
```



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

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

    /** Display contents of Product table.
     * @param conn connection to database
     */
    private static void displayProducts( Connection conn )
        throws ULjException
    {
        PreparedStatement stmt = conn.prepareStatement(
            "SELECT prod_no, prod_name, price FROM Product"
            + " ORDER BY prod_no"
        );
        ResultSet cursor = stmt.executeQuery();
        for( ; cursor.next(); ) {
            String prod_no = cursor.getString( 1 /* "prod_no" */ );
            String prod_name = cursor.getString( 2 /* "prod_name" */ );
            String price = cursor.getString( 3 /* "price" */ );
            Demo.display( prod_no + " " + prod_name + " " + price );
        }
        cursor.close();
        stmt.close();
    }

    /** mainline for program.
     * @param args command-line arguments (not used)
     */
    public static void main
        ( String[] args )
    {
        try {
            Connection conn = createDatabase();
```

```
            populate( conn );
            displayProducts( conn );
            conn.release();
        } catch( ULjException exc ) {
            Demo.displayException( exc );
        }
    }

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

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

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

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

        /** Initialize the encryption control with a password.
         * @param password the password
         */
        public void initialize( String password )
            throws ULjException
        {
            byte[] bytes = null;
            try {
                bytes = password.getBytes( "UTF8" );
            } catch( Exception e ) {
```



```
                Demo.display( "Encryption initialization failure" );
                throw new ObfuscationError();
            }
            _seed = 0;
            for( int i = bytes.length; i > 0; ) {
                _seed ^= bytes[ --i ];
            }
        }
    }

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

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

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

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

Example: Encrypting data

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

```
package ianywhere.ultralitej.demo;
import ianywhere.ultralitej.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.*;
/**
 * Encrypted -- sample program to demonstrate encryption of database.
 *
 * This sample requires either the Sun JDK 1.4.2 (or later) or a freeware
 * version of the Java encryption classes (JCE).
 */
public class Encrypted
{
    /** Create the database.
     * @return connection for a new database
     */
    private static Connection createDatabase()
        throws ULjException
    {
        ConfigPersistent config =
DatabaseManager.createConfigurationFile( "Encrypt.ulj" );
        config.setEncryption( new Encryptor() );
        Connection conn = DatabaseManager.createDatabase( config );

        conn.schemaCreateBegin();

        TableSchema table_schema = conn.createTable( "Product" );
        table_schema.createColumn( "prod_no", Domain.INTEGER );
        table_schema.createColumn( "prod_name", Domain.VARCHAR, 32 );
        table_schema.createColumn( "price", Domain.NUMERIC, 9, (short)2 );
        IndexSchema index_schema =
table_schema.createPrimaryIndex( "prime_keys" );
        index_schema.addColumn( "prod_no", IndexSchema.ASCENDING );

        conn.schemaCreateComplete();

        return conn;
    }

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

    /** Populate the database.
     * @param conn connection to database
     */
    private static void populate( Connection conn )
        throws ULjException
    {
        PreparedStatement ri = conn.prepareStatement(
            "INSERT INTO Product( prod_no, prod_name, price )"
```



```
                + " VALUES( :prod_no, :prod_name, :price )"
            );
            addProduct( ri, 2001, "blue screw", ".03" );
            addProduct( ri, 2002, "red screw", ".09" );
            addProduct( ri, 2004, "hammer", "23.99" );
            addProduct( ri, 2005, "vise", "39.99" );
            ri.close();
            conn.commit();
        }

        /** Display contents of Product table.
         * @param conn connection to database
         */
        private static void displayProducts( Connection conn )
            throws ULjException
        {
            PreparedStatement stmt = conn.prepareStatement(
                "SELECT prod_no, prod_name, price FROM Product"
                + " ORDER BY prod_no"
            );
            ResultSet cursor = stmt.executeQuery();
            for( ; cursor.next(); ) {
                /* Can't access columns by name because no meta data */
                String prod_no = cursor.getString( 1 /* "prod_no" */ );
                String prod_name = cursor.getString( 2 /* "prod_name" */ );
                String price = cursor.getString( 3 /* "price" */ );
                Demo.display( prod_no + " " + prod_name + " " + price );
            }
            cursor.close();
            stmt.close();
        }

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

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

            /** Encrypt a page stored in the Database.
             * @param page_no the number of the page being encrypted
             * @param src the encrypted source page which was read from the
    Database
             * @param tgt the unencrypted page (filled in by method)
             */
            public void decrypt( int page_no, byte[] src, byte[] tgt )
                throws ULjException
```

```
        {
            byte[] decrypted = null;
            try {
                _cipher.init( Cipher.DECRYPT_MODE, _key );
                decrypted = _cipher.doFinal( src );
            } catch( Exception e ) {
                Demo.display( "Error: decrypting" );
                throw new EncryptionError();
            }
            for( int i = tgt.length; i > 0; ) {
                --i;
                tgt[ i ] = decrypted[ i ];
            }
        }

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

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

    /** Error class for encryption errors.
     */
```



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

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

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

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

Example: Displaying database schema information

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

```
package ianywhere.ultralitej.demo;
import ianywhere.ultralitej.*;

/** Sample program to dump schema of a database.
 * This sample extracts schema information into a set of data structures
 * (TableArray, OptionArray) before dumping out the meta data so as to
 * provide for future schema information lookup.
 */
public class DumpSchema
{
    /** Mainline.
     * @param args program arguments (not used)
     */
```

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

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

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

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

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

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

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



```
private static final int INDEX_COLUMN_ORDER = 5;
private static final int INDEX_COLUMN_COLUMN_ID = 6;
private static final int INDEX_COLUMN_FLAGS = 7;

private static final int INDEX_COLUMN_FLAG_FORWARD = 1;

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

private static String SQL_SELECT_OPTIONS =
        "SELECT name, value FROM " + TableSchema.SYS_INTERNAL
        + " ORDER BY name"
        ;
private static final int OPTION_NAME = 1;
private static final int OPTION_VALUE = 2;

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

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

    // read indexes
    stmt = conn.prepareStatement( SQL_SELECT_INDEX_COLS );
    cursor = stmt.executeQuery();
    int last_index_id = -1;
    Index index = null;
    last_table_id = -1;
    for( ; cursor.next(); ) {
```

```
                int table_id = cursor.getInt( INDEX_TABLE_ID );
                int index_id = cursor.getInt( INDEX_ID );
                if( last_table_id != table_id || last_index_id != index_id ) {
                    String index_name = cursor.getString( INDEX_NAME );
                    int index_flags = cursor.getInt( INDEX_FLAGS );
                    index = new Index( index_id, index_name, index_flags );
                    table = findTable( table_id );
                    table.addIndex( index );
                    last_index_id = index_id;
                    last_table_id = table_id;
                }
                int order = cursor.getInt( INDEX_COLUMN_ORDER );
                int column_id = cursor.getInt( INDEX_COLUMN_COLUMN_ID );
                int index_column_flags = cursor.getInt( INDEX_COLUMN_FLAGS );
                IndexColumn index_column = new IndexColumn( order, column_id,
index_column_flags );
                index.addColumn( index_column );
            }
            cursor.close();
            stmt.close();

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

        /** Dump the schema of a database
         */
        private static void dumpSchema( Connection conn ) throws ULjException
        {
            // Display the metadata options
            Demo.display( "\nMetadata options:\n" );
            for( int opt_no = 0; opt_no < options.count(); ++ opt_no ) {
                Option option = options.elementAt( opt_no );
                option.display();
            }
            // Display the metadata tables
            Demo.display( "\nMetadata tables:" );
            for( int table_no = 0; table_no < tables.count(); ++ table_no ) {
                Table table = tables.elementAt( table_no );
                table.display( table_no );
            }
            // Display the rows for non-system tables.
            for( int table_no = 0; table_no < tables.count(); ++ table_no ) {
                Table table = tables.elementAt( table_no );
                if( 0 == ( table.getFlags() & TABLE_FLAG_SYSTEM ) ) {
                    Demo.display( "\nRows for table: ", table.getName(), "\n" );
                    Index index = table.getIndex( 0 );
                    PreparedStatement stmt = conn.prepareStatement(
                            "SELECT * FROM \"" + table.getName() + "\""
                        );
                    ResultSet cursor = stmt.executeQuery();
                    int column_count = table.getColumnCount();
                    int row_count = 0;
                    for( ; cursor.next(); ) {
                        StringBuffer buf = new StringBuffer();
```



```
                    buf.append( "Row[" );
                    buf.append( Integer.toString( ++row_count ) );
                    buf.append( "]:" );
                    char joiner = ' ';
                    for( int col_no = 1; col_no <= column_count; ++col_no ) {
                        String value = cursor.isNull( col_no )
                                    ? "<NULL>"
                                    : cursor.getString( col_no );
                        buf.append( joiner );
                        buf.append( value );
                        joiner = ',';
                    }
                    Demo.display( buf.toString() );
                }
                cursor.close();
                stmt.close();
            }
        }
    }

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

    /** Representation of a column.
     */
    private static class Column
    {
        private int _column_id;
        private String _column_name;
        private int _column_flags;
        private Domain _domain;
        private int _column_default;

        Column( Connection conn, int column_id, String column_name, int
column_flags, int column_domain, int column_length, int column_default )
            throws ULjException
        {
            _column_id = column_id;
            _column_name = column_name;
            _column_flags = column_flags;
            _column_default = column_default;
            int scale = 0;
            switch( column_domain ) {
              case Domain.NUMERIC :
                scale = column_length >> 8;
                column_length &= 255;
                _domain = conn.createDomain( Domain.NUMERIC, column_length,
scale );
                break;
              case Domain.VARCHAR :
              case Domain.BINARY :
                _domain = conn.createDomain( column_domain, column_length );
```

```
            break;
          default :
            _domain = conn.createDomain( column_domain );
            break;
        }
    }

    String getName()
    {
        return _column_name;
    }

    void display()
    {
        StringBuffer buf = new StringBuffer();
        buf.append( "  \"" );
        buf.append( _column_name );
        buf.append( "\"\t" );
        buf.append( _domain.getName() );
        switch( _domain.getType() ) {
          case Domain.NUMERIC :
            buf.append( '(' );
            buf.append( Integer.toString( _domain.getPrecision() ) );
            buf.append( ',' );
            buf.append( Integer.toString( _domain.getScale() ) );
            buf.append( ')' );
            break;
          case Domain.VARCHAR :
          case Domain.BINARY :
            buf.append( '(' );
            buf.append( Integer.toString( _domain.getSize() ) );
            buf.append( ')' );
            break;
        }
        if( 0 != ( _column_flags & COLUMN_FLAG_IS_NULLABLE ) ) {
            buf.append( " NULL" );
        } else {
            buf.append( " NOT NULL" );
        }
        switch( _column_default ) {
            case ColumnSchema.COLUMN_DEFAULT_NONE:
            default:
                break;
            case ColumnSchema.COLUMN_DEFAULT_AUTOINC:
                buf.append( " DEFAULT AUTOINCREMENT" );
                break;
            case ColumnSchema.COLUMN_DEFAULT_GLOBAL_AUTOINC:
                buf.append( " DEFAULT GLOBAL AUTOINCREMENT" );
                break;
            case ColumnSchema.COLUMN_DEFAULT_CURRENT_DATE:
                buf.append( " DEFAULT CURRENT DATE" );
                break;
            case ColumnSchema.COLUMN_DEFAULT_CURRENT_TIME:
                buf.append( " DEFAULT CURRENT TIME" );
                break;
            case ColumnSchema.COLUMN_DEFAULT_CURRENT_TIMESTAMP:
                buf.append( " DEFAULT CURRENT TIMESTAMP" );
                break;
            case ColumnSchema.COLUMN_DEFAULT_UNIQUE_ID:
                buf.append( " DEFAULT NEWID()" );
                break;
        }
        buf.append( ", /* column_id=" );
        buf.append( Integer.toString( _column_id ) );
```



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

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

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

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

        void display( Table table, boolean constraints )
        {
            StringBuffer buf = new StringBuffer();
            String flags = "";
            String indent = "  ";
            if( 0 != ( _index_flags & INDEX_FLAG_PRIMARY_INDEX ) ) {
                if( !constraints )      return;
                buf.append( "  CONSTRAINT \"" );
                buf.append( _index_name );
                buf.append( "\" PRIMARY KEY (" );
                flags = "PRIMARY_KEY,UNIQUE_KEY";
            } else if( 0 != ( _index_flags & INDEX_FLAG_UNIQUE_KEY ) ) {
                if( !constraints )      return;
                buf.append( "  CONSTRAINT \"" );
                buf.append( _index_name );
                buf.append( "\" UNIQUE (" );
                flags = "UNIQUE_KEY";
            } else {
                if( constraints )       return;
                if( 0 != ( _index_flags & INDEX_FLAG_UNIQUE_INDEX ) ) {
                    buf.append( "UNIQUE " );
                    flags = "UNIQUE_INDEX";
                }
                indent = "";
                buf.append( "INDEX \"" );
```

```
                buf.append( _index_name );
                buf.append( "\" ON \"" );
                buf.append( table.getName() );
                buf.append( "\" (" );
            }
            buf.append( "\n" );
            buf.append( indent );
            buf.append( "  /* index_id=" );
            buf.append( Integer.toString( _index_id ) );
            buf.append( " index_flags=" );
            buf.append( flags );
            if( 0 != ( _index_flags & INDEX_FLAG_PERSISTENT ) ) {
                buf.append( ",PERSISTENT" );
            }
            buf.append( " */" );
            Demo.display( buf.toString() );
            int bounds = _columns.count();
            for( int col_no = 0; col_no < bounds; ++ col_no ) {
                IndexColumn column = _columns.elementAt( col_no );
                column.display( table, indent, col_no + 1 < bounds );
            }
            Demo.display( indent + ")" );
        }

        String getName()
        {
            return _index_name;
        }
    }

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

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

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



```
/** Representation of a database Option.
 */
private static class Option
{
    private String _option_name;
    private String _option_value;

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

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

}

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

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

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

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

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

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

```
    int getFlags()
    {
        return _table_flags;
    }

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

    String getName()
    {
        return _table_name;
    }

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

    int getId()
    {
        return _table_id;
    }
}
```



```
/** Simple adjustable array of objects.
 */
private static class ObjArray
{
    private Object[] _array = new Object[ 10 ];
    private int _used = 0;

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

    int count()
    {
        return _used;
    }

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

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

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

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

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

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

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

}
```

The partial output of the application is shown below.

```
Metadata options:

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

Metadata tables:

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

Table[1] name = "syscolumn"  id = 1 flags = 0xc000,SYSTEM,NO_SYNC
```



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

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

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

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

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

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

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

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


CHAPTER 3

Tutorial: Build a BlackBerry application

Contents

Introduction to UltraLiteJ development

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

The tutorial assumes the following:

- You are familiar with the Java programming language.
- You have Research In Motion BlackBerry JDE 4.0 or later installed on your computer.
- You have UltraLiteJ installed on your computer. The default installation location for UltraLiteJ is *install-dir\UltraLite\UltraLiteJ* .
- You have the Research In Motion MDS Services Simulator installed on your computer. This is required for part 2.

Part 1: Creating an UltraLiteJ application on BlackBerry

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

Lesson 1: Create a BlackBerry JDE project

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

1. From the JDE **File** menu, choose **New Workspace**.
2. Choose a location for your workspace; for example, *c:\tutorials*. Name the workspace **HelloBlackBerry** and click **OK**.
3. In this tutorial, the workspace contains a single project. From the **Project** menu, choose **Create New Project**.
4. Name the project **HelloBlackBerry** and click **OK**.
5. Add the UltraLiteJ JAR file to the project.
   a. In the **Workspace** window, right-click the project and choose **Properties**.
   b. On the **Build** tab, click **Add**, which is located next to the **Imported Jar Files** field.
   c. Browse to the *UltraLiteJ\J2meRim11\UltraLiteJ.jar* file in your UltraLiteJ installation and click **Open**.
   d. Click **OK** to close the **Properties** window.
6. From the **Project** menu, choose **Set Active Projects**. Select the HelloBlackBerry project and click **OK**.
7. Save the project.

Lesson 2: Display a BlackBerry application screen

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

1. In the workspace view, right-click the project and choose **Create New File In Project**.
2. In the **Source File Name** box, type **myapp\Application.java** to create a file named *Application.java* that is part of the myapp package.

   Click **OK** to create the file. The Application class appears in the JDE window.
3. Define the Application class. No imports are needed for this class. Add a constructor and a main method so that your Application class is defined as follows:

```
class Application extends net.rim.device.api.ui.UiApplication {
    public static void main( String[] args )
```

```
        {
            Application instance = new Application();
            instance.enterEventDispatcher();
        }
        Application() {
            pushScreen( new HomeScreen() );
        }
    }
```

4. Add the HomeScreen class to your project.

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

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

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

    The HomeScreen class appears in the JDE window.

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

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

class HomeScreen extends MainScreen {

    HomeScreen() {

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

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

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

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

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

    The BlackBerry simulator launches in a separate window.

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

9. Start the application.

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

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

    The simulator terminates.

Lesson 3: Create an UltraLiteJ database

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

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

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

```
class HomeScreen extends MainScreen {

    HomeScreen() {

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

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

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

2. Create a file named *myapp\DataAccess.java* in the HelloBlackBerry project.
3. Provide a getDataAccess method that ensures a single database connection.

```
package myapp;

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

class DataAccess {
    DataAccess() {    }

    public static synchronized DataAccess getDataAccess(boolean reset)
                  throws Exception
    {
        if( _da == null ){
            _da = new DataAccess();
            ConfigObjectStore _config =
             DatabaseManager.createConfigurationObjectStore("HelloDB");
            if(reset)
```

```
                {
                    _conn = DatabaseManager.createDatabase( _config );
                }
                else
                {
                    try{
                        _conn = DatabaseManager.connect( _config );
                    }
                    catch( ULjException uex1) {
                        if( uex1.getErrorCode() !=
                            ULjException.SQLE_ULTRALITE_DATABASE_NOT_FOUND ) {
                            System.out.println( "Exception: " +
                                                uex1.toString() );
                            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 ianywhere.ultralitej package from the *UltraLiteJ.jar* file. The steps to create or connect to a database are:

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

b. Attempt to connect to the database.

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

4. Press F5 to build and deploy the application to the device simulator.
5. From the **File** menu, choose **Load Java Program**.
6. Browse to the *J2meRim11* folder of your UltraLiteJ installation and open the *UltraLiteJ.cod* file.
7. Run the program from the Simulator.

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

Lesson 4: Create a database table

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

Column name	Data type	Allow null?	Default	Primary key?
ID	UUID	No	None	Yes

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

1. Add a DataAccess method to create the table.

```
private void createDatabaseSchema()
{
    try{
        _conn.schemaCreateBegin();
        ColumnSchema column_schema;
        TableSchema table_schema = _conn.createTable("Names");
        column_schema = table_schema.createColumn( "ID", Domain.UUID );
        column_schema.setDefault( ColumnSchema.COLUMN_DEFAULT_UNIQUE_ID);
        table_schema.createColumn( "Name", Domain.VARCHAR, 254 );
        IndexSchema index_schema =
                table_schema.createPrimaryIndex("prime_keys");
        index_schema.addColumn("ID", IndexSchema.ASCENDING);
        _conn.schemaCreateComplete();
    }
    catch( ULjException uex1){
        System.out.println( "ULjException: " + uex1.toString() );
    }
    catch( Exception ex1){
        System.out.println( "Exception: " + ex1.toString() );
    }
}
```

   If the table already exists, an exception is thrown.

2. Call the DataAccess.getDataAccess method.

   Add the following code before the method returns:

```
_da.createDatabaseSchema();
return _da;
```

3. Run the application again in the simulator.

Altering the table schema

When you alter the table schema, for example, by adding a table definition, you must keep the following information in mind:

- All schema creation must be done inside calls to schemaCreateBegin and schemaCreateEnd calls.
- The Connection.createTable method creates an empty table.
- The TableSchema.createColumn method creates a column.
- The TableSchema.createPrimaryIndex method creates the primary key.

Lesson 5: Add data to the table

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

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

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

1. Add the controls to the screen.

   a. Add the following code before calling the getDataAccess method.

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

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

// Add a menu item
addMenuItem(_addToListMenuItem);

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

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

```
private EditField _nameEditField;
private ObjectListField _nameListField;

private MenuItem _addToListMenuItem = new MenuItem("Add", 10, 1){
    public void run() {
        // TODO
    }
};
```

   c. Recompile the application and confirm that it runs.

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

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

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

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



```
            System.out.println( "Exception: " + ex.toString() );
        }
    }
```

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

```
package myapp;

class NameRow {

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

    public String getNameID(){
        return _nameID;
    }

    public String getName(){
        return _name;
    }

    public String toString(){
        return _name;
    }

    private String _nameID;
    private String _name;

}
```

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

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

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

```
public void refreshNameList(){
    //Clear the list
```

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

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

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

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

```
private void onAddToList(){
    _da.insertName(_nameEditField.getText());
    this.refreshNameList();
    _nameEditField.setText("");
}
```

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

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

3. Compile and run the application.

Resetting the simulator

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

Lesson 6: Deploy application to smartphone

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

Applications running on a BlackBerry must be signed using the BlackBerry Signature Tool. This tool is available from Research in Motion (RIM) as part of the BlackBerry JDE Component Package. The *UltraLiteJ.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, visit the BlackBerry Developer Program web site at http://na.blackberry.com/eng/developers/.

Signing the application

1. Start the BlackBerry Signature Tool. A **File Open** window appears.
2. Browse to and select your compiled application, the *HelloBlackBerry.cod* file.
3. Click **Request To Sign The File**.
4. Click **Close** to close the signature tool.

Deploying the application

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

1. Connect your BlackBerry to your computer using the USB cable, and ensure that the Desktop Manager can see the device.
2. Click **Application Loader** and follow the instructions in the wizard.
3. Browse to the *HelloBlackBerry.alx* file and add it to your device.
4. Browse to the *J2meRim11\UltraLiteJ.alx* file and add it to your device.

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

Part 2: Adding synchronization to the BlackBerry application

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

Lesson 1: Create a SQL Anywhere database

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

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

   ```
   dbinit HelloBlackBerry.db
   ```
3. Create an ODBC data source to connect to the database.
   a. Open the ODBC Administrator.

      From the **Start** menu, choose **Programs** » **SQL Anywhere 11** » **SQL Anywhere** » **ODBC Administrator**.

      The **ODBC Data Source Administrator** window appears.
   b. Select the **User DSN** tab to create an ODBC data source for the current user.
   c. Click **Add**.

      The **Create New Data Source Wizard** appears.
   d. From the list of drivers, choose **SQL Anywhere 11**, and then click **Finish**.

      The **SQL Anywhere 11 ODBC Configuration** window appears.
   e. On the **Login** tab, type **DBA** as the user ID and **SQL** as the password.

      These are the default login name and password for all SQL Anywhere databases, and should not be used in a production environment.
   f. Click the **Database** tab, type **HelloBlackBerry** as the **Server Name** and *c:\tutorial\database\HelloBlackBerry.db* as the **Database File**. Click **OK**.
4. Run the following command to start Interactive SQL and connect to the SQL Anywhere database:

   ```
   dbisql -c dsn=HelloBlackBerry
   ```
5. Execute the following statement to create a table in the database:

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


6. Close Interactive SQL.

Lesson 2: Create MobiLink scripts and start the MobiLink server

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

1. From the **Start** menu, choose **Programs** » **SQL Anywhere 11** » **Sybase Central**.
2. In the Sybase Central **Task** pane, choose **Create A Synchronization Model** from the MobiLink 11 tasks.
   a. Type the Synchronization model name **HelloBlackBerrySyncModel**, and store the model in the *c:\tutorial\database* folder. Click **Next**.
   b. Click **Choose a Consolidated Database**.
   c. In the **Connect** window choose the HelloBlackBerry ODBC Data Source and click **OK**.
   d. Click **Yes** to create the MobiLink system setup.
   e. Choose **No, Create A New Remote Schema**.
   f. For downloads, choose **Timestamp-based Download**.
   g. Click **Finish** to complete creating the synchronization model and save the project.
3. Right-click the synchronization model and choose **Deploy**. Choose to deploy only to the consolidated database.
4. In the **Deploy Synchronization Model Wizard**, choose **Save SQL** and deploy the database. When prompted for the consolidated database, specify the ODBC data source **HelloBlackBerry**.
5. For the MobiLink user and password choose user name **mluser** with password **mlpassword**.
6. Click **Finish** to deploy the synchronization model to the consolidated database.

Lesson 3: Add synchronization to the application

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

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

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

// Add sync menu item
addMenuItem(_syncMenuItem);

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

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

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

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

3. Create the onSync method.

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

4. Define the syncParms and streamParms variables at the class level.

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

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

```
public boolean sync() {
    try {
        if( _syncParms == null ){
            String host = "ultralitej.sybase.com";
            _syncParms = _conn.createSyncParms( "mluser",
"HelloBlackBerrySyncModel" );
            _syncParms.setPassword("mlpassword");
            _streamParms = _syncParms.getStreamParms();
            _streamParms.setPort( 80 ); // use your own
            _streamParms.setHost( host ); // use your own
            if(host.equals("ultralitej.sybase.com"))
            {
                _streamParms.setURLSuffix("scripts/iaredirect.dll/ml/
HelloBlackBerry/");
            }
        }
        System.out.println( "Synchronizing" );
        _conn.synchronize( _syncParms );
        return true;
    }
    catch( ULjException uex){
        System.out.println(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 now refers to the synchronization version, or a set of synchronization logic, that was deployed to your consolidated database.

The stream parameters object, StreamHTTPParms, indicate the host name and port number of the MobiLink server. When you start the MobiLink server in the next lesson, use your own computer name

and select a port that is available. Do not use localhost as your computer name. You can use port 80 if you have no web server running on your computer.

6. Compile your application.

Lesson 4: Run the application on the simulator

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

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

   ```
   mlsrv11 -c " DSN=HelloBlackBerry" -v+ -x http(port=8081) -ot ml.txt
   ```

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

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

3. Enter names at the server.

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

   b. Run the following SQL statements add names:

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

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

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

6. Synchronize the application.

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

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

Code listing for tutorial

This section provides the complete code for the preceding tutorial There are four Java classes that are present in the tutorials.

See also

- "Part 1: Creating an UltraLiteJ application on BlackBerry" on page 67
- "Part 2: Adding synchronization to the BlackBerry application" on page 76

Application.java

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

package myapp;


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

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

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

DataAccess.java

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

package myapp;


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



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

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

    /**
     *
     * Create the table in the database.
     * If the table already exists, a harmless exception is thrown
     */
    private void createDatabaseSchema()
    {
        try{
            _conn.schemaCreateBegin();
            ColumnSchema column_schema;
            TableSchema table_schema = _conn.createTable("Names");
            column_schema = table_schema.createColumn( "ID", Domain.UUID );
            column_schema.setDefault( ColumnSchema.COLUMN_DEFAULT_UNIQUE_ID);
            table_schema.createColumn( "Name", Domain.VARCHAR, 254 );
            IndexSchema index_schema =
               table_schema.createPrimaryIndex("prime_keys");
            index_schema.addColumn("ID", IndexSchema.ASCENDING);
            _conn.schemaCreateComplete();
        }
```

```
            catch( ULjException uex1){
                System.out.println( "ULjException: " + uex1.toString() );
            }
            catch( Exception ex1){
                System.out.println( "Exception: " + ex1.toString() );
            }
        }

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

        public boolean sync() {
            try {
                if( _syncParms == null ){
                    String host = "ultralitej.sybase.com";
                    _syncParms = _conn.createSyncParms( "mluser",
                                 "HelloBlackBerrySyncModel" );
                    _syncParms.setPassword("mlpassword");
                    _streamParms = _syncParms.getStreamParms();
                    _streamParms.setPort( 80 ); // use your own
                    _streamParms.setHost( host ); // use your own
                    if(host.equals("ultralitej.sybase.com"))
                    {
                        _streamParms.setURLSuffix(
```



```
                        "scripts/iaredirect.dll/ml/HelloBlackBerry/");
                }

            }
            System.out.println( "Synchronizing" );
            _conn.synchronize( _syncParms );
            return true;
        }
        catch( ULjException uex){
            System.out.println(uex.toString());
            return false;
        }
    }

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

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

HomeScreen.java

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

package myapp;


import net.rim.device.api.ui.*;
import net.rim.device.api.ui.component.*;
import net.rim.device.api.ui.container.*;
import java.util.*;
/**
 *
 */
class HomeScreen extends MainScreen {


    HomeScreen() {

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

```
            setTitle(applicationTitle);

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

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

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

            // Add a menu item
            addMenuItem(_addToListMenuItem);

            // Add sync menu item
            addMenuItem(_syncMenuItem);

            // Add reset menu item
            addMenuItem(_resetMenuItem);

            // Create database and connect
            try{
                _da = DataAccess.getDataAccess(false);
                _statusLabel.setText("Status: Connected");
            }
            catch( Exception ex)
            {
                System.out.println("Exception: " + ex.toString() );
                _statusLabel.setText("Exception: " + ex.toString() );
            }

            // Fill the ObjectListField
            this.refreshNameList();

        }

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

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



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

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

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

}
```

NameRow.java

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

package myapp;
```

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

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

    public String getNameID(){
        return _nameID;
    }

    public String getName(){
        return _name;
    }


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

    private String _nameID;
    private String _name;

}
```


Part II. UltraLiteJ Reference

CHAPTER 4

UltraLiteJ API reference

Contents

CollectionOfValueReaders interface

Provides methods to return the column value of a given row.

Syntax

public **CollectionOfValueReaders**

Derived classes

- "ResultSet interface" on page 177

Remarks

The returned results are based on a SQL SELECT statement, which is declared by a PreparedStatement, and are stored in a ResultSet.

Values returned by any given method in this interface are accessed based on the integer parameter, ordinal, which specifies the column order as it appears in the SQL SELECT statement. The ordinal has a base index of one.

Values are returned as either java primitive types or read-only Value objects.

The following example demonstrates how to declare a new PreparedStatement with a SQL SELECT statement, execute the statement, store the query results in a new ResultSet, and store a column1 value as a String using a get method.

```
// 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 using getString.
    String row1_col1 = rs.getString(1);
}
```

Members

All members of CollectionOfValueReaders, including all inherited members.

- "getBlobInputStream function" on page 92
- "getBoolean function" on page 92
- "getBytes function" on page 92
- "getClobReader function" on page 93
- "getDate function" on page 93
- "getDecimalNumber function" on page 93
- "getDouble function" on page 94
- "getFloat function" on page 94
- "getInt function" on page 94
- "getLong function" on page 95

getBlobInputStream function

Returns an InputStream.

Syntax

java.io.InputStream **CollectionOfValueReaders.getBlobInputStream(**
  int *ordinal*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The InputStream representation of the named value.

getBoolean function

Returns a boolean value.

Syntax

boolean **CollectionOfValueReaders.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.

getBytes function

Returns a byte array.

Syntax

byte[] **CollectionOfValueReaders.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.

getClobReader function

Returns a Reader.

Syntax

java.io.Reader **CollectionOfValueReaders.getClobReader(**
  int *ordinal*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The Reader representation of the named value.

getDate function

Returns a java.util.Date.

Syntax

java.util.Date **CollectionOfValueReaders.getDate(**
  int *ordinal*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The java.util.Date representation of the named value.

getDecimalNumber function

Returns a DecimalNumber.

Syntax

DecimalNumber **CollectionOfValueReaders.getDecimalNumber(**
  int *ordinal*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The DecimalNumber representation of the named value.

getDouble function

Returns a double value.

Syntax

double **CollectionOfValueReaders.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.

getFloat function

Returns a float value.

Syntax

float **CollectionOfValueReaders.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.

getInt function

Returns an integer value.

Syntax

int **CollectionOfValueReaders.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.

getLong function

Returns a long integer value.

Syntax

long **CollectionOfValueReaders.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.

getOrdinal function

Returns the (base-one) ordinal for the value represented by a String.

Syntax

int **CollectionOfValueReaders.getOrdinal(**
  String *name*
) throws **ULjException**

Parameters

- **name** A String representing the table column name.

Returns

The ordinal value.

getString function

Returns a String value.

Syntax

String **CollectionOfValueReaders.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.

getValue function

Returns a Value object.

Syntax

```
Value CollectionOfValueReaders.getValue(
  int ordinal
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The Value object representation of the named value.

isNull function

Tests if a value is null.

Syntax

```
boolean CollectionOfValueReaders.isNull(
  int ordinal
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

True if the value is null, false otherwise.

CollectionOfValueWriters interface

Provides methods to set the column value of a given row.

Syntax

public **CollectionOfValueWriters**

Derived classes

- "PreparedStatement interface" on page 173

Remarks

All methods are prepared based on a SQL statement, which is defined in the initial declaration of a PreparedStatement, and are applied to the database with the execute method.

When updating a column value, you must reference the column by the order number as it appears in the prepared SQL statement. This number is passed as an integer parameter, ordinal, which has a base index of one.

The following example demonstrates how to declare a new PreparedStatement with a SQL UPDATE statement, prepare your change using the set method, and apply those changes to the UltraLite database.

```
// Define a new prepared SQL statement.
String sql_string = "UPDATE SampleTable SET column1 = ? WHERE pkey = 1";

// Create a new PreparedStatement from an existing connection.
PreparedStatement ps = conn.prepareStatement(sql_string);

// Set a String value to the first column in the SQL statement (column1).
ps.set(1, "New Value");

// Commit the changes to the database.
conn.commit();
```

Members

All members of CollectionOfValueWriters, including all inherited members.

- "getBlobOutputStream function" on page 98
- "getClobWriter function" on page 98
- "getOrdinal function" on page 98
- "set function" on page 99
- "set function" on page 99
- "set function" on page 99
- "set function" on page 100
- "set function" on page 100
- "set function" on page 100
- "set function" on page 101
- "set function" on page 101
- "set function" on page 101
- "set function" on page 102
- "setNull function" on page 102

getBlobOutputStream function

Returns an OutputStream.

Syntax

java.io.OutputStream **CollectionOfValueWriters.getBlobOutputStream(**
  int *ordinal*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The OutputStream for the named value.

getClobWriter function

Returns a Writer.

Syntax

java.io.Writer **CollectionOfValueWriters.getClobWriter(**
  int *ordinal*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

Returns

The Writer for the named value.

getOrdinal function

Returns the (base-one) ordinal for the value represented by the name.

Syntax

int **CollectionOfValueWriters.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.

set function

Sets a boolean value to the column number in the SQL statement that is defined by ordinal.

Syntax

```
void CollectionOfValueWriters.set(
  int ordinal,
  boolean value
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.
- **value** The value to be set.

set function

Sets a DecimalNumber to the column number in the SQL statement that is defined by ordinal.

Syntax

```
void CollectionOfValueWriters.set(
  int ordinal,
  DecimalNumber value
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.
- **value** The value to be set.

set function

Sets an integer value to the column number in the SQL statement that is defined by ordinal.

Syntax

```
void CollectionOfValueWriters.set(
  int ordinal,
  int value
) throws ULjException
```

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.
- **value** The value to be set.

set function

Sets a java.util.Date to the column number in the SQL statement that is defined by ordinal.

Syntax

void **CollectionOfValueWriters.set(**
  int *ordinal*,
  java.util.Date *value*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.
- **value** The value to be set.

set function

Sets a long integer value to the column number in the SQL statement that is defined by ordinal.

Syntax

void **CollectionOfValueWriters.set(**
  int *ordinal*,
  long *value*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.
- **value** The value to be set.

set function

Sets a float value to the column number in the SQL statement that is defined by ordinal.

Syntax

void **CollectionOfValueWriters.set(**
  int *ordinal*,
  float *value*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.
- **value** The value to be set.

set function

Sets a double value to the column number in the SQL statement that is defined by ordinal.

Syntax

void **CollectionOfValueWriters.set(**
  int *ordinal*,
  double *value*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.
- **value** The value to be set.

set function

Sets a byte array value to the column number in the SQL statement that is defined by ordinal.

Syntax

void **CollectionOfValueWriters.set(**
  int *ordinal*,
  byte[] *value*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.
- **value** The value to be set.

set function

Sets a String value to the column number in the SQL statement that is defined by ordinal.

Syntax

void **CollectionOfValueWriters.set(**
  int *ordinal*,
  String *value*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.
- **value** The value to be set.

set function

Sets a Value object to the column number in the SQL statement that is defined by ordinal.

Syntax

void **CollectionOfValueWriters.set(**
  int *ordinal*,
  Value *value*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.
- **value** The value to be set.

setNull function

Sets a null value to the column number in the SQL statement that is defined by ordinal.

Syntax

void **CollectionOfValueWriters.setNull(**
  int *ordinal*
) throws **ULjException**

Parameters

- **ordinal** A base-one integer representing the column number as ordered in the SQL statement.

ColumnSchema interface

Specifies the schema of a column.

Syntax

public **ColumnSchema**

Remarks

An object supporting this interface is returned by the TableSchema.createColumn(String.short), TableSchema.createColumn(String,short,int) and TableSchema.createColumn(String,short,int,int) methods.

The following example demonstrates the creation of the schema for a simple database. The T1 table is created with a primary key integer column which is auto-incrementing.

```
// Assumes a valid Connection object
TableSchema table_schema;
ColumnSchema col_schema;
IndexSchema index_schema;

table_schema = conn.createTable("T1");
col_schema = table_schema.createColumn("num", Domain.INTEGER);
col_schema.setDefault(ColumnSchema.COLUMN_DEFAULT_AUTOINC);

// BIT columns are not nullable by default.
col_schema = table_schema.createColumn("flag", Domain.BIT);
col_schema.setNullable(true);
col_schema = table_schema.createColumn(
    "cost", Domain.NUMERIC, 10, 2
    );
col_schema.setNullable(false);

index_schema = table_schema.createPrimaryIndex("primary");
index_schema.addColumn("num", IndexSchema.ASCENDING);
conn.schemaCreateComplete();
```

Members

All members of ColumnSchema, including all inherited members.

COLUMN_DEFAULT_AUTOINC variable

Specifies the column is auto incrementing.

Syntax

final byte **ColumnSchema.COLUMN_DEFAULT_AUTOINC**

Remarks

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

In UltraLiteJ, the autoincremented value is not set to 0 when the table is created, and AUTOINCREMENT generates negative numbers when a signed data type is used for the column. Therefore, declare AUTOINCREMENT columns as unsigned integers to prevent negative values from being used.

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

See also

- setDefault function

COLUMN_DEFAULT_CURRENT_DATE variable

Specifies the column defaults to the current date (year, month and day).

Syntax

final byte **ColumnSchema.COLUMN_DEFAULT_CURRENT_DATE**

Remarks

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

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

See also

- setDefault function

COLUMN_DEFAULT_CURRENT_TIME variable

Specifies the column defaults to the current time.

Syntax

final byte **ColumnSchema.COLUMN_DEFAULT_CURRENT_TIME**

Remarks

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

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

See also

- setDefault function

COLUMN_DEFAULT_CURRENT_TIMESTAMP variable

Specifies the column defaults to the current timestamp.

Syntax

final byte **ColumnSchema.COLUMN_DEFAULT_CURRENT_TIMESTAMP**

Remarks

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

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

See also

- setDefault function

COLUMN_DEFAULT_GLOBAL_AUTOINC variable

Specifies the column is global auto incrementing.

Syntax

final byte **ColumnSchema.COLUMN_DEFAULT_GLOBAL_AUTOINC**

Remarks

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

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

See also

- setDefault function
- setDatabaseId function

COLUMN_DEFAULT_NONE variable

Specifies the column has no special default value.

Syntax

final byte **ColumnSchema.COLUMN_DEFAULT_NONE**

Remarks

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

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

See also

- setDefault function
- setNullable function

COLUMN_DEFAULT_UNIQUE_ID variable

Specifies the column defaults to a new unique identifier.

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 of the TableSchema.SYS_COLUMNS system table.

See also

- setDefault function

setDefault function

Sets the default value for a column.

Syntax

ColumnSchema **ColumnSchema.setDefault(**
byte *default_code*
)

Parameters

- **default_code** One of the ColumnSchema codes, constants with the COLUMN_DEFAULT suffix, indicating the type of default value that the column should have.

Remarks

The default is COLUMN_DEFAULT_NONE.

See also

- "COLUMN_DEFAULT_AUTOINC variable" on page 103
- "COLUMN_DEFAULT_CURRENT_DATE variable" on page 104
- "COLUMN_DEFAULT_CURRENT_TIME variable" on page 104
- "COLUMN_DEFAULT_CURRENT_TIMESTAMP variable" on page 105
- "COLUMN_DEFAULT_GLOBAL_AUTOINC variable" on page 105
- "COLUMN_DEFAULT_NONE variable" on page 106
- "COLUMN_DEFAULT_UNIQUE_ID variable" on page 106

Returns

This ColumnSchema with the default value defined.

setNullable function

Sets a column as nullable.

Syntax

ColumnSchema **ColumnSchema.setNullable(**
boolean *nullable*
)

Parameters

- **nullable** Set true if this column should accept null values; otherwise, set false.

Remarks

Columns in primary and unique keys are always not nullable. BIT-type columns are not nullable by default.

Returns

This ColumnSchema with nullable defined.

ConfigFile interface

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

Syntax

public **ConfigFile**

Base classes

- "Configuration interface" on page 119
- "ConfigPersistent interface" on page 111

Members

All members of ConfigFile, including all inherited members.

- "getAutoCheckpoint function" on page 111
- "getCacheSize function" on page 112
- "getDatabaseName function" on page 119
- "getLazyLoadIndexes function" on page 112
- "getPageSize function" on page 119
- "hasPersistentIndexes function" on page 112
- "setAutocheckpoint function" on page 112
- "setCacheSize function" on page 113
- "setEncryption function" on page 113
- "setIndexPersistence function" on page 114
- "setLazyLoadIndexes function" on page 114
- "setPageSize function" on page 120
- "setPassword function" on page 120
- "setRowMaximumThreshold function" on page 115
- "setRowMinimumThreshold function" on page 115
- "setShadowPaging function" on page 116
- "setWriteAtEnd function" on page 116
- "writeAtEnd function" on page 117

ConfigNonPersistent interface

Establishes the Configuration for a non-persistent database.

Syntax

public **ConfigNonPersistent**

Base classes

- "Configuration interface" on page 119

Members

All members of ConfigNonPersistent, including all inherited members.

- "getDatabaseName function" on page 119
- "getPageSize function" on page 119
- "setPageSize function" on page 120
- "setPassword function" on page 120

ConfigObjectStore interface (J2ME BlackBerry only)

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

Syntax

public **ConfigObjectStore**

Base classes

- "Configuration interface" on page 119
- "ConfigPersistent interface" on page 111

Members

All members of ConfigObjectStore, including all inherited members.

- "getAutoCheckpoint function" on page 111
- "getCacheSize function" on page 112
- "getDatabaseName function" on page 119
- "getLazyLoadIndexes function" on page 112
- "getPageSize function" on page 119
- "hasPersistentIndexes function" on page 112
- "setAutocheckpoint function" on page 112
- "setCacheSize function" on page 113
- "setEncryption function" on page 113
- "setIndexPersistence function" on page 114
- "setLazyLoadIndexes function" on page 114
- "setPageSize function" on page 120
- "setPassword function" on page 120
- "setRowMaximumThreshold function" on page 115
- "setRowMinimumThreshold function" on page 115
- "setShadowPaging function" on page 116
- "setWriteAtEnd function" on page 116
- "writeAtEnd function" on page 117

ConfigPersistent interface

Establishes the Configuration for a persistent database.

Syntax

public **ConfigPersistent**

Base classes

- "Configuration interface" on page 119

Derived classes

- "ConfigFile interface" on page 108
- "ConfigObjectStore interface (J2ME BlackBerry only)" on page 110
- "ConfigRecordStore interface (J2ME only)" on page 118

Members

All members of ConfigPersistent, including all inherited members.

- "getAutoCheckpoint function" on page 111
- "getCacheSize function" on page 112
- "getDatabaseName function" on page 119
- "getLazyLoadIndexes function" on page 112
- "getPageSize function" on page 119
- "hasPersistentIndexes function" on page 112
- "setAutocheckpoint function" on page 112
- "setCacheSize function" on page 113
- "setEncryption function" on page 113
- "setIndexPersistence function" on page 114
- "setLazyLoadIndexes function" on page 114
- "setPageSize function" on page 120
- "setPassword function" on page 120
- "setRowMaximumThreshold function" on page 115
- "setRowMinimumThreshold function" on page 115
- "setShadowPaging function" on page 116
- "setWriteAtEnd function" on page 116
- "writeAtEnd function" on page 117

getAutoCheckpoint function

Determines if auto checkpoint is turned on.

Syntax

boolean **ConfigPersistent.getAutoCheckpoint()**

Returns

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

getCacheSize function

Returns the cache size of the database, in bytes.

Syntax

int **ConfigPersistent.getCacheSize()**

Returns

The cache size.

getLazyLoadIndexes function

Determines if lazy loading indexes is turned on.

Syntax

boolean **ConfigPersistent.getLazyLoadIndexes()**

Returns

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

hasPersistentIndexes function

Determines if indexes are persistent.

Syntax

boolean **ConfigPersistent.hasPersistentIndexes()**

Returns

true, if indexes are persistent; otherwise, false.

setAutocheckpoint function

Sets auto checkpoint on.

Syntax

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

Parameters

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

Remarks

A database has checkpoint turned on when committed change operations are applied to the persistent rows in the persistent store. When auto checkpoint is active, a checkpoint occurs as part of each commit; otherwise, a transaction record is written to record the change and the persistent row storage is not changed until the application invokes the checkpoint method.

Change and commit operations operate faster when auto checkpoint is not active but database startup can be slower if many transactions without checkpoints exist.

Autocheckpoint is always true if indexes are not persistent or if row limitation is enabled.

Returns

This ConfigPersistent with the auto checkpoint set.

setCacheSize function

Sets the cache size of the database, in bytes.

Syntax

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

Parameters

- **cache_size** The cache size.

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.

Returns

This ConfigPersistent with the cache size set.

setEncryption function

Sets an Encryption.

Syntax

ConfigPersistent **ConfigPersistent.setEncryption(**
  EncryptionControl *control*
)

Parameters

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

Returns

This ConfigPersistent with the encryption set.

setIndexPersistence function

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.

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.

Returns

This ConfigPersistent with the index persistence set.

setLazyLoadIndexes function

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.

Remarks

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

Returns

This ConfigPersistent with the lazy load indexes set.

setRowMaximumThreshold function

Sets the threshold for the maximum number of rows to retain in memory.

Syntax

ConfigPersistent **ConfigPersistent.setRowMaximumThreshold(**
  int *threshold*
)

Parameters

- **threshold** The maximum threshold value.

Remarks

When the maximum number of rows is reached, rows get truncated and retain the minimum number of rows defined by the setRowMinimumThreshold method.

See also

- setRowMinimumThreshold function

Returns

This ConfigPersistent with the maximum threshold set.

setRowMinimumThreshold function

Sets the threshold for the minimum number of rows to retain in memory.

Syntax

ConfigPersistent **ConfigPersistent.setRowMinimumThreshold(**
  int *threshold*
)

Parameters

- **threshold** The minimum threshold value.

Remarks

When the maximum number of rows is reached, rows get truncated and retain the minimum number of rows defined by the setRowMinimumThreshold method.

See also

- setRowMaximumThreshold function

Returns

This ConfigPersistent with the minimum threshold set.

setShadowPaging function

Sets shadow paging on.

Syntax

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

Parameters

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

Remarks

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

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

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

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

Returns

This ConfigPersistent with ShadowPaging set.

setWriteAtEnd function

Sets index persistence during shutdown on.

Syntax

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

Parameters

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

Remarks

Enabling this option speeds up database operations but all changes to the database are lost if the application terminates abnormally.

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

Returns

This ConfigPersistent with WriteAtEnd set.

writeAtEnd function

Determines if index persistence during shutdown is turned on.

Syntax

boolean **ConfigPersistent.writeAtEnd()**

Returns

true, if index persistence during shutdown is on; otherwise, false.

ConfigRecordStore interface (J2ME only)

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

Syntax

public **ConfigRecordStore**

Base classes

- "Configuration interface" on page 119
- "ConfigPersistent interface" on page 111

Members

All members of ConfigRecordStore, including all inherited members.

- "getAutoCheckpoint function" on page 111
- "getCacheSize function" on page 112
- "getDatabaseName function" on page 119
- "getLazyLoadIndexes function" on page 112
- "getPageSize function" on page 119
- "hasPersistentIndexes function" on page 112
- "setAutocheckpoint function" on page 112
- "setCacheSize function" on page 113
- "setEncryption function" on page 113
- "setIndexPersistence function" on page 114
- "setLazyLoadIndexes function" on page 114
- "setPageSize function" on page 120
- "setPassword function" on page 120
- "setRowMaximumThreshold function" on page 115
- "setRowMinimumThreshold function" on page 115
- "setShadowPaging function" on page 116
- "setWriteAtEnd function" on page 116
- "writeAtEnd function" on page 117

Configuration interface

Establishes the Configuration for a database.

Syntax

public **Configuration**

Derived classes

- "ConfigNonPersistent interface" on page 109
- "ConfigPersistent interface" on page 111

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.

Members

All members of Configuration, including all inherited members.

- "getDatabaseName function" on page 119
- "getPageSize function" on page 119
- "setPageSize function" on page 120
- "setPassword function" on page 120

getDatabaseName function

Returns the database name.

Syntax

String **Configuration.getDatabaseName()**

Returns

The name of the database.

getPageSize function

Returns the page size of the database, in bytes.

Syntax

int **Configuration.getPageSize()**

Returns

The page size.

setPageSize function

Sets the page size of the database, in bytes.

Syntax

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

Parameters

- **page_size** The page size.

Remarks

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

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

The page size can range from 256 to 32736 bytes. The default is 2048 bytes.

Returns

This Configuration object with the page size set.

setPassword function

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.

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".

Returns

This Configuration object with the password set.

Connection interface

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

Syntax

public **Connection**

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.

```
index_schema = table_schema.createPrimaryIndex("primary");
index_schema.addColumn("num", IndexSchema.ASCENDING);

table_schema = conn.createTable("T2");
table_schema.createColumn("num", Domain.INTEGER);
table_schema.createColumn("quantity", Domain.INTEGER);

index_schema = table_schema.createPrimaryIndex("primary");
index_schema.addColumn("num", IndexSchema.ASCENDING);
index_schema = table_schema.createIndex("index1");
index_schema.addColumn("quantity", IndexSchema.ASCENDING);

conn.createPublication("PubA", new String[] {"T1"});
conn.schemaCreateComplete();
```

A connection provides the following capabilities:

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

The following example demonstrates how to create a schema for a simple database. 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.

```
TableSchema table_schema;
 IndexSchema index_schema;
 table_schema = conn.createTable("T1");
 table_schema.createColumn("num", Domain.INTEGER);
```

Members

All members of Connection, including all inherited members.

- “checkpoint function” on page 126

- "setDatabaseId function" on page 139
- "setOption function" on page 139
- "startSynchronizationDelete function" on page 140
- "stopSynchronizationDelete function" on page 140
- "SYNC_ALL variable" on page 125
- "SYNC_ALL_DB_PUB_NAME variable" on page 126
- "SYNC_ALL_PUBS variable" on page 126
- "synchronize function" on page 141
- "truncateTable function" on page 141

CONNECTED variable

A connected state.

Syntax

final byte **Connection.CONNECTED**

NOT_CONNECTED variable

A not connected state.

Syntax

final byte **Connection.NOT_CONNECTED**

OPTION_DATABASE_ID variable

Database option: database id.

Syntax

final String **Connection.OPTION_DATABASE_ID**

OPTION_DATE_FORMAT variable

Database option: date format.

Syntax

final String **Connection.OPTION_DATE_FORMAT**

OPTION_DATE_ORDER variable

Database option: date order.

Syntax

final String **Connection.OPTION_DATE_ORDER**

OPTION_ML_REMOTE_ID variable

Database Option: ML remote ID.

Syntax

final String **Connection.OPTION_ML_REMOTE_ID**

Remarks

OPTION_NEAREST_CENTURY variable

Database option: nearest century.

Syntax

final String **Connection.OPTION_NEAREST_CENTURY**

OPTION_PRECISION variable

Database option: precision.

Syntax

final String **Connection.OPTION_PRECISION**

OPTION_SCALE variable

Database option: scale.

Syntax

final String **Connection.OPTION_SCALE**

OPTION_TIMESTAMP_FORMAT variable

Database option: timestamp format.

Syntax

final String **Connection.OPTION_TIMESTAMP_FORMAT**

OPTION_TIMESTAMP_INCREMENT variable

Database option: timestamp increment.

Syntax

final String **Connection.OPTION_TIMESTAMP_INCREMENT**

OPTION_TIME_FORMAT variable

Database option: time format.

Syntax

final String **Connection.OPTION_TIME_FORMAT**

PROPERTY_DATABASE_NAME variable

Database Property: database name.

Syntax

final String **Connection.PROPERTY_DATABASE_NAME**

PROPERTY_PAGE_SIZE variable

Database Property: page size.

Syntax

final String **Connection.PROPERTY_PAGE_SIZE**

SYNC_ALL variable

The publication list used to request synchronization of all tables in the database, including tables not in any publication.

Syntax

final String **Connection.SYNC_ALL**

Remarks

Tables marked as NoSync are never synchronized.

This constant is equivalent to the null reference or an empty string.

SYNC_ALL_DB_PUB_NAME variable

The reserved name of the SYNC_ALL_DB publication.

Syntax

final String **Connection.SYNC_ALL_DB_PUB_NAME**

See also

- getLastDownloadTime function
- resetLastDownloadTime function

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 marked as NoSync are never synchronized.

checkpoint function

Checkpoints the database changes.

Syntax

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

Remarks

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

commit function

Commits the database changes.

Syntax

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

Remarks

Invoking this method makes all database changes since to the last commit or rollback become permanent changes.

createDecimalNumber function

Creates a DecimalNumber.

Syntax

DecimalNumber **Connection.createDecimalNumber(**
  int *precision*,
  int *scale*
) throws **ULjException**

Parameters

- **precision** The number of digits in the number.
- **scale** The number of decimal places in the number.

Returns

The DecimalNumber with the specified type.

createDecimalNumber function

Creates a DecimalNumber.

Syntax

DecimalNumber **Connection.createDecimalNumber(**
  int *precision*,
  int *scale*,
  String *value*
) throws **ULjException**

Parameters

- **precision** The number of digits in the number.
- **scale** The number of decimal places in the number.
- **value** The value to be set.

Returns

The DecimalNumber with the specified type.

createDomain function

Creates a fixed size domain.

Syntax

Domain **Connection.createDomain(**
  int *type*
) throws **ULjException**

Parameters

- **type** The domain type.

Returns

The Domain of the given type.

createDomain function

Creates a variable size Domain.

Syntax

```
Domain Connection.createDomain(
  int type,
  int size
) throws ULjException
```

Parameters

- **type** The domain type.
- **size** The size of domain.

Returns

The Domain of the given type.

createDomain function

Creates a variable size Domain.

Syntax

```
Domain Connection.createDomain(
  int type,
  int size,
  int scale
) throws ULjException
```

Parameters

- **type** The domain type.
- **size** The size of the domain.
- **scale** The scale of the domain.

Returns

The Domain of the given type.

createForeignKey function

Creates a new foreign key.

Syntax

ForeignKeySchema **Connection.createForeignKey(**
  String *table_name*,
  String *primary_table_name*,
  String *name*
) throws **ULjException**

Parameters

- **table_name** The name of table to contain a foreign key. This is the table constrained to have a valid reference to the primary table.
- **primary_table_name** The name of table containing referenced columns.
- **name** The name of the foreign key. The specified name must be a valid SQL identifier.

Remarks

Note
UltraLiteJ does not enforce foreign key constraints on tables. Foreign keys are used to determine the correct order in which tables should be synchronized. Foreign keys on the client database should match the relations on the consolidated database that the client synchronizes to.

Returns

The ForeignKey with the name and table involvement defined.

createPublication function

Creates a new publication in the database.

Syntax

void **Connection.createPublication(**
  String *pub_name*,
  String[] *tables*
) throws **ULjException**

Parameters

- **pub_name** The name of the publication to create.
- **tables** An array of table names.

Remarks

The synchronization of the entire database is done using the special Connection.SYNC_ALL publication list.

See also

- dropPublication function
- setPublications function
- setTableOrder function

createSyncParms function

Creates a synchronization parameters set for HTTP synchronization.

Syntax

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

Parameters

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

See also

- createSyncParms function
- setUserName function

Returns

The SyncParms object.

createSyncParms function

Creates a synchronization parameters set.

Syntax

SyncParms **Connection.createSyncParms(**
  int *streamType*,
  String *userName*,
  String *version*
) throws **ULjException**

Parameters

- **streamType** One of the constants defined in the SyncParms class used to identify the type of synchronization stream.
- **userName** The MobiLink user name.
- **version** The MobiLink script version.

See also

- createSyncParms function
- HTTP_STREAM variable
- HTTPS_STREAM variable

Returns

SyncParms object

createTable function

Creates a new table in the database.

Syntax

```
TableSchema Connection.createTable(
  String table_name
) throws ULjException
```

Parameters

- **table_name** The name of table to create.

Remarks

This method can only be executed when the connected database is in schema creation mode.

Returns

The TableSchema object for the new table.

See also

- schemaCreateBegin function
- schemaCreateComplete function

createUUIDValue function

Creates a UUID value.

Syntax

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

Returns

The Value for the domain.

createValue function

Creates a Value from a Domain.

Syntax

Value **Connection.createValue(**
  Domain *dom*
) throws **ULjException**

Parameters

- **dom** The given domain.

Returns

The Value for the domain.

disableSynchronization function

Disables synchronization for a table.

Syntax

void **Connection.disableSynchronization(**
  String *table_name*
) throws **ULjException**

Parameters

- **table_name** The name of table.

dropDatabase function

Drops a database.

Syntax

void **Connection.dropDatabase()** throws **ULjException**

Remarks

The database referenced by the connection is erased and the connection is released. Only this connection can be active for the database being dropped.

dropForeignKey function

Drops a foreign key.

Syntax

void **Connection.dropForeignKey(**
  String *table_name*,
  String *fkey_name*
) throws **ULjException**

Parameters

- **table_name** The name of table containing the foreign key.
- **fkey_name** The name the foreign key to drop.

dropPublication function

Drops a publication from the database.

Syntax

```
void Connection.dropPublication(
  String pub_name
) throws ULjException
```

Parameters

- **pub_name** The name of the publication to drop.

Remarks

You can not drop the special Connection.SYNC_ALL_DB_PUB_NAME publication.

See also

- createPublication function

dropTable function

Drops a table from the database.

Syntax

```
void Connection.dropTable(
  String table_name
) throws ULjException
```

Parameters

- **table_name** The name of table to drop.

Remarks

A table can only be dropped when there are no outstanding uncommitted transactions for the current connection and when the table does not occur in any publication. Any rows in the dropped table are lost. Any unsynchronized operations are also lost.

emergencyShutdown function

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.

enableSynchronization function

Enables synchronization for a table.

Syntax

void **Connection.enableSynchronization(**
String *table_name*
) throws **ULjException**

Parameters

- **table_name** The table name.

getDatabaseId function

Returns the value of database ID.

Syntax

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

Returns

The database ID.

getDatabaseInfo function

Returns a DataInfo object containing information on database properties.

Syntax

DatabaseInfo **Connection.getDatabaseInfo()** throws **ULjException**

Returns

The DatabaseInfo object.

getDatabasePartitionSize function

The value of database ID.

Syntax

int **Connection.getDatabasePartitionSize()** throws **ULjException**

Remarks

Returns the size of database partition.

See also

- getDatabaseId function

getDatabaseProperty function

Returns a property of the database.

Syntax

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

Parameters

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

Returns

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

getLastDownloadTime function

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.

Remarks

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

This method can only be executed once `schemaCreateComplete()` has been executed.

See also

- createPublication function
- resetLastDownloadTime function
- schemaCreateBegin function
- schemaCreateComplete function

Returns

The timestamp of the last download.

getOption function

Returns a database option.

Syntax

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

Parameters

- **option_name** Any of the Configuration option values, variables with the OPTION suffix, to retrieve a value for.

Remarks

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

When a database is created, a set of required options are created.

Returns

The value of the database option.

See also

- setOption function
- "OPTION_DATABASE_ID variable" on page 123
- "OPTION_DATE_FORMAT variable" on page 123
- "OPTION_DATE_ORDER variable" on page 123
- "OPTION_ML_REMOTE_ID variable" on page 124
- "OPTION_NEAREST_CENTURY variable" on page 124
- "OPTION_PRECISION variable" on page 124
- "OPTION_SCALE variable" on page 124
- "OPTION_TIMESTAMP_FORMAT variable" on page 124
- "OPTION_TIMESTAMP_INCREMENT variable" on page 125
- "OPTION_TIME_FORMAT variable" on page 125

getState function

Returns the state of the connection.

Syntax

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

Remarks

Valid return statements are those supported by the Configuration interface.

See also

- "CONNECTED variable" on page 123
- "NOT_CONNECTED variable" on page 123

prepareStatement function

Prepares a statement for execution.

Syntax

PreparedStatement **Connection.prepareStatement(**
  String *sql*
) throws **ULjException**

Parameters

- **sql** A SQL statement to prepare.

See also

- PreparedStatement interface

Returns

A PreparedStatement object.

release function

Releases the 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.

renameTable function

Renames a table.

Syntax

void **Connection.renameTable(**
  String *old_table_name*,
  String *new_table_name*
) throws **ULjException**

Parameters

- **old_table_name** The name of an existing table.
- **new_table_name** The new name of the table.

resetLastDownloadTime function

Resets the time of the download for the specified publications.

Syntax

void **Connection.resetLastDownloadTime(**
  String *pub_name*
) throws **ULjException**

Parameters

- **pub_name** The name of the publication to check.

Remarks

To reset the download time for when the entire database is synchronized, use the special Connection.SYNC_ALL_DB_PUB_NAME publication.

This method requires that there are no uncommitted transactions on the current connection.

See also

- createPublication function

rollback function

Commits a rollback to undo changes to the database.

Syntax

void **Connection.rollback()** throws **ULjException**

Remarks

Invoking this method undoes all database changes on this connection since the commit or rollback.

schemaCreateBegin function

Puts the connected database into schema creation mode.

Syntax

void **Connection.schemaCreateBegin()** throws **ULjException**

Remarks

When the database is in schema creation mode, data and synchronization operations are prohibited. Incoming connections to the database are also denied.

schemaCreateComplete function

Puts the connected database out of schema creation mode.

Syntax

void **Connection.schemaCreateComplete()** throws **ULjException**

setDatabaseId function

Sets the database id and partition size for global autoincrement.

Syntax

void **Connection.setDatabaseId(**
  int *id*,
  int *size*
) throws **ULjException**

Parameters

- **id** The database id.
- **size** The size of the partition.

setOption function

Sets the database option.

Syntax

void **Connection.setOption(**
  String *option_name*,
  String *option_value*
) throws **ULjException**

Parameters

- **option_name** Any of the Configuration option values, variables with the OPTION suffix, to set.
- **option_value** The new value of the option.

Remarks

If the option is not currently stored on the database, it is created.

There cannot be any uncommitted transactions for this connection when the method is invoked.

See also

- getOption function
- “OPTION_DATABASE_ID variable” on page 123
- “OPTION_DATE_FORMAT variable” on page 123
- “OPTION_DATE_ORDER variable” on page 123
- “OPTION_ML_REMOTE_ID variable” on page 124
- “OPTION_NEAREST_CENTURY variable” on page 124
- “OPTION_PRECISION variable” on page 124
- “OPTION_SCALE variable” on page 124
- “OPTION_TIMESTAMP_FORMAT variable” on page 124
- “OPTION_TIMESTAMP_INCREMENT variable” on page 125
- “OPTION_TIME_FORMAT variable” on page 125

startSynchronizationDelete function

Starts synchronizing deletions.

Syntax

void **Connection.startSynchronizationDelete()** throws **ULjException**

Remarks

This enables any future deletions to be synchronized.

stopSynchronizationDelete function

Stops synchronizing deletions.

Syntax

void **Connection.stopSynchronizationDelete()** throws **ULjException**

Remarks

This disables any future deletions to be synchronized until a subsequent `startSynchronizationDelete` is executed.

synchronize function

Synchronizes the database with a MobiLink server.

Syntax

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

Parameters

- **config** The parameters used for synchronization

Remarks

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

See also

- checkpoint function

truncateTable function

Deletes all rows in a table.

Syntax

void **Connection.truncateTable(**
  String *table_name*
) throws **ULjException**

Parameters

- **table_name** The name of table to be truncated.

Remarks

The rows are not synchronized.

DatabaseInfo interface

Associated with a Connection object and provides methods to reveal database information.

Syntax

public **DatabaseInfo**

Remarks

This interface is invoked with the getDatabaseInfo method of a Connection object.

Members

All members of DatabaseInfo, including all inherited members.

- "getCommitCount function" on page 142
- "getDbFormat function" on page 142
- "getLogSize function" on page 142
- "getNumberRowsToUpload function" on page 143
- "getPageSize function" on page 143
- "getRelease function" on page 143

getCommitCount function

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

Syntax

int **DatabaseInfo.getCommitCount()**

Returns

The total number of commit operations.

getDbFormat function

Returns the database version number.

Syntax

int **DatabaseInfo.getDbFormat()**

Returns

The version number.

getLogSize function

Returns the overall size of the transaction log, in bytes.

Syntax

int **DatabaseInfo.getLogSize()**

Returns

The size of transaction log.

getNumberRowsToUpload function

Returns the number of rows awaiting upload.

Syntax

int **DatabaseInfo.getNumberRowsToUpload()**

Returns

The number of rows.

getPageSize function

Returns the page size of the database, in bytes.

Syntax

int **DatabaseInfo.getPageSize()**

Returns

The page size.

getRelease function

Returns the software release number, expressed as an integer.

Syntax

int **DatabaseInfo.getRelease()**

Remarks

A software release value of 7.1.3 is returned as 713.

Returns

The release number.

DatabaseManager class

Provides static methods to obtain basic configurations, create a new database, and connect to an existing database.

Syntax

public **DatabaseManager**

Remarks

The following examples demonstrate how to open an existing database, and create a new one if it does not exist.

This example works with J2ME devices:

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

This example works with J2ME BlackBerry devices:

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

This example works with J2SE devices:

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

Members

All members of DatabaseManager, including all inherited members.

- "connect function" on page 145
- "createConfigurationFile function" on page 145

connect function

Connects to an existing database based on a Configuration and returns the Connection.

Syntax

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

Parameters

- **config** The Configuration for the existing database.

See also

- Configuration interface

Returns

A Connection to the existing database.

createConfigurationFile function

Creates a Configuration with a file as the physical store and returns a ConfigFile.

Syntax

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

Parameters

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

See also

- "ConfigFile interface" on page 108

Returns

The ConfigFile used to configure a database.

createConfigurationNonPersistent function

Creates a Configuration without a persistent store and returns a ConfigNonPersist.

Syntax

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

Parameters

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

See also

- ConfigNonPersistent interface

Returns

The ConfigNonPersistent used to configure a database.

createConfigurationObjectStore function (J2ME BlackBerry only)

Creates a Configuration with a RIM object store as the physical store and returns a ConfigObjectStore.

Synopsis

ConfigObjectStore **createConfigurationObjectStore(**
String*db_name*
)

Parameters

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

See also

- "ConfigObjectStore interface (J2ME BlackBerry only)" on page 110

Returns

The ConfigObjectStore used to configure a database.

createConfigurationRecordStore function (J2ME only)

Creates a Configuration with a record store as the physical store and returns a ConfigRecordStore.

Syntax

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

Parameters

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

See also

- ConfigRecordStore interface (J2ME only)

Returns

The ConfigRecordStore used to configure a database.

createDatabase function

Creates a new database based on a Configuration and returns the Connection.

Syntax

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

Parameters

- **config** The Configuration for the new database.

Remarks

This method replaces any database with the same name.

See also

- Configuration interface

Returns

A Connection to the new database.

createSISHTTPListener function (J2ME BlackBerry only)

Creates an HTTP SISListener for server-initiated synchronization.

Synopsis

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

Parameters

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

Remarks

4400 is the recommended port setting.

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

See also

- "SISListener interface (J2ME BlackBerry only)" on page 181

Returns

The SISListener for server-initiated synchronization.

release function

Closes the DatabaseManager to release all connections and to shutdown all databases.

Syntax

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

Remarks

Any uncommitted transactions are rolled back.

setErrorLanguage function

Sets the language to use for error messages.

Syntax

void **DatabaseManager.setErrorLanguage(**
  String *lang*
)

Parameters

- **lang** The language code as a double character.

Remarks

Recognized languages are EN, DE, FR, JA, ZH. If an unrecognized language is specified, the system reverts to the default.

In J2SE and BlackBerry J2ME environments, the current Locale is used to determine the default language. In other J2ME environments, this method provides the only way to specify the language. The default language is "EN".

DecimalNumber interface

Describes an exact decimal value and provides decimal arithmetic support for Java platforms where java.math.BigDecimal is not available.

Syntax

public **DecimalNumber**

Members

All members of DecimalNumber, including all inherited members.

add function

Adds two DecimalNumbers and returns the sum.

Syntax

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

Parameters

- **num1** A first number.
- **num2** A second number.

Returns

The sum of num1 and num2.

divide function

Divides the first DecimalNumber by the second DecimalNumber and returns the quotient.

Syntax

```
DecimalNumber DecimalNumber.divide(
  DecimalNumber num1,
```


DecimalNumber *num2*
) throws **ULjException**

Parameters

- **num1** A dividend.
- **num2** A divisor.

Returns

The quotient of num1 divided by num2.

getString function

Returns the String representation of the DecimalNumber.

Syntax

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

Returns

The String value.

isNull function

Determines if the DecimalNumber is null.

Syntax

boolean **DecimalNumber.isNull()**

Returns

true, if the object is null; otherwise, false.

multiply function

Multiplies two DecimalNumbers together and returns the product.

Syntax

DecimalNumber **DecimalNumber.multiply(**
DecimalNumber *num1*,
DecimalNumber *num2*
) throws **ULjException**

Parameters

- **num1** A multiplicand.
- **num2** A multiplier.

Returns

The product of num1 and num2.

set function

Sets the DecimalNumber with a String value.

Syntax

void **DecimalNumber.set(**
  String *value*
) throws **ULjException**

Parameters

- **value** numerical value represented as a String.

setNull function

Sets the DecimalNumber to null.

Syntax

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

subtract function

Subtracts the second DecimalNumber from the first DecimalNumber and returns the difference.

Syntax

DecimalNumber **DecimalNumber.subtract(**
  DecimalNumber *num1*,
  DecimalNumber *num2*
) throws **ULjException**

Parameters

- **num1** A minuend.
- **num2** A subtrahend.

Returns

The difference between num1 and num2.

Domain interface

Describes Domain type information for a column in a table.

Syntax

public **Domain**

Remarks

This interface contains constants to denote the various domains, and methods to extract information from a Domain object.

The following example demonstrates how to create a schema for a simple database. The T2 table is created with an integer column and a variable length character string column that is, at most, 32 bytes long.

```
// Assumes a valid Connection object conn
TableSchema table_schema;
IndexSchema index_schema;

table_schema = conn.createTable("T2");
table_schema.createColumn("num", Domain.INTEGER);
table_schema.createColumn("name", Domain.VARCHAR, 32);

index_schema = table_schema.createPrimaryIndex("primary");
index_schema.addColumn("num", IndexSchema.ASCENDING);
```

Integer Types

Domain Constant	SQL Type	Value Range
BIT	BIT	0 or 1
TINY	TINYINT	0 to 255 (unsigned integer using 1 byte of storage)
SHORT	SMALLINT	-32768 to 32767 (signed integer using 2 bytes of storage)
UNSIGNED_SHORT	UNSIGNED SMALLINT	0 to 65535 (unsigned integer using 2 bytes of storage)
INTEGER	INTEGER	-231 to 231 - 1, or -2147483648 to 2147483647 (signed integer using 4 bytes of storage)
UNSIGNED_INTEGER	UNSIGNED INTEGER	0 to 232 - 1, or 0 to 4294967295 (unsigned integer using 4 bytes of storage)
BIG	BIGINT	-263 to 263 - 1, or -9223372036854775808 to 9223372036854775807 (signed integer using 8 bytes of storage)

Domain Constant	SQL Type	Value Range
UNSIGNED_BIG	UNSIGNED BIGINT	0 to 264 - 1, or 0 to 18446744073709551615 (unsigned integer using 8 bytes of storage)

Non-Integer Numeric Types

Domain Constant	SQL Type	Value Range
REAL	REAL	-3.402823e+38 to 3.402823e+38, with numbers close to zero as small as 1.175495e-38 (single precision floating point number using 4 bytes of storage, rounding errors may occur after the sixth digit)
DOUBLE	DOUBLE	-1.79769313486231e+308 to 1.79769313486231e+308, with numbers close to zero as small as 2.22507385850721e-308 (single precision floating point number using 8 bytes of storage, rounding errors may occur after the fifteenth digit)
NUMERIC	NUMERIC (precision, scale)	any decimal numbers with precision (size) total digits and with scale digits after the decimal point (no rounding within precision)

Character and Binary Types

Domain Constant	SQL Type	Size Range
VARCHAR	VARCHAR(size)	1 to 32767 bytes (characters are stored as 1-3 byte UTF-8 characters). When evaluating expressions, the maximum length for a temporary character value is 2048 bytes.
LONGVARCHAR	LONG VARCHAR	Any length (memory permitting). The only operations allowed on LONG VARCHAR columns are to insert, update, or delete them, or to include them in the select-list of a query.
BINARY	BINARY(size)	1 to 32767 bytes. When evaluating expressions, the maximum length for a temporary character value is 2048 bytes.
LONGBINARY	LONG BINARY	Any length (memory permitting). The only operations allowed on LONG BINARY columns are to insert, update, or delete them, or to include them in the select-list of a query.
UUID	UNIQUEIDENTIFIER	Always 16 bytes binary with special interpretation

Date and Time Types

Domain Constant	SQL Type	Value
DATE	DATE	Year, month, day.
TIME	TIME	Hour, minute, second, and fraction of a second.
TIMESTAMP	TIMESTAMP	DATE and TIME.

BIT columns are not nullable by default. All other types are nullable by default.

Members

All members of Domain, including all inherited members.

BIG variable

Domain ID constant for a 64-bit integer (SQL type BIGINT).

Syntax

final short **Domain.BIG**

See also

- Domain interface

BINARY variable

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

Syntax

final short **Domain.BINARY**

See also

- Domain interface

BINARY_DEFAULT variable

Default size of Binary type.

Syntax

final short **Domain.BINARY_DEFAULT**

BINARY_MAX variable

Maximum size of Binary type.

Syntax

final short **Domain.BINARY_MAX**

BINARY_MIN variable

Minimum size of Binary type.

Syntax

final short **Domain.BINARY_MIN**

BIT variable

Domain ID constant for a bit (SQL type BIT).

Syntax

final short **Domain.BIT**

Remarks

BIT columns are not nullable by default.

See also

- Domain interface

CHARACTER_MAX variable

Maximum size of Character type.

Syntax

final short **Domain.CHARACTER_MAX**

DATE variable

Domain ID constant for a Date (SQL type DATE).

Syntax

final short **Domain.DATE**

See also

- Domain interface

DOMAIN_MAX variable

Maximum kinds of Domain types.

Syntax

final short **Domain.DOMAIN_MAX**

DOUBLE variable

Domain ID constant for a 8-byte floating point (SQL type DOUBLE).

Syntax

final short **Domain.DOUBLE**

See also

- Domain interface

INTEGER variable

Domain ID constant for a 32-bit integer (SQL type INTEGER).

Syntax

final short **Domain.INTEGER**

See also

- Domain interface

LONGBINARY variable

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

Syntax

final short **Domain.LONGBINARY**

See also

- Domain interface

LONGBINARY_DEFAULT variable

Default size of BLOB type.

Syntax

final short **Domain.LONGBINARY_DEFAULT**

LONGBINARY_MIN variable

Minimum size of BLOB type.

Syntax

final short **Domain.LONGBINARY_MIN**

LONGVARCHAR variable

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

Syntax

final short **Domain.LONGVARCHAR**

See also

- Domain interface

LONGVARCHAR_DEFAULT variable

Default size of CLOB type.

Syntax

final short **Domain.LONGVARCHAR_DEFAULT**

LONGVARCHAR_MIN variable

Minimum size of CLOB type.

Syntax

final short **Domain.LONGVARCHAR_MIN**

NUMERIC variable

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

Syntax

final short **Domain.NUMERIC**

See also

- Domain interface

PRECISION_DEFAULT variable

Default size of Precision in Numeric.

Syntax

final short **Domain.PRECISION_DEFAULT**

PRECISION_MAX variable

Maximum size of Precision in Numeric.

Syntax

final short **Domain.PRECISION_MAX**

PRECISION_MIN variable

Minimum size of Precision in Numeric.

Syntax

final short **Domain.PRECISION_MIN**

REAL variable

Domain ID constant for a 4-byte floating point (SQL type REAL).

Syntax

final short **Domain.REAL**

See also

- Domain interface

SCALE_DEFAULT variable

Default size of Scale in Numeric.

Syntax

final short **Domain.SCALE_DEFAULT**

SCALE_MAX variable

Maximum size of Scale in Numeric.

Syntax

final short **Domain.SCALE_MAX**

SCALE_MIN variable

Minimum size of Scale in Numeric.

Syntax

final short **Domain.SCALE_MIN**

SHORT variable

Domain ID constant for a 16-bit integer (SQL type SMALLINT).

Syntax

final short **Domain.SHORT**

See also

- Domain interface

TIME variable

Domain ID constant for a Time (SQL type TIME).

Syntax

final short **Domain.TIME**

See also

- Domain interface

TIMESTAMP variable

Domain ID constant for a Timestamp (SQL type TIMESTAMP).

Syntax

final short **Domain.TIMESTAMP**

See also

- Domain interface

TINY variable

Domain ID constant for a unsigned 8-bit integer (SQL type TINYINT).

Syntax

final short **Domain.TINY**

See also

- Domain interface

UINT16_MAX variable

Maximum size of unsigned 16-bit integer.

Syntax

final int **Domain.UINT16_MAX**

UNSIGNED_BIG variable

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

Syntax

final short **Domain.UNSIGNED_BIG**

See also

- Domain interface

UNSIGNED_INTEGER variable

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

Syntax

final short **Domain.UNSIGNED_INTEGER**

See also

- Domain interface

UNSIGNED_SHORT variable

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

Syntax

final short **Domain.UNSIGNED_SHORT**

See also

- Domain interface

UUID variable

Domain ID constant for a UniqueIdentifier (SQL type UNIQUEIDENTIFIER).

Syntax

final short **Domain.UUID**

See also

- Domain interface

VARCHAR variable

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

Syntax

final short **Domain.VARCHAR**

See also

- Domain interface

VARCHAR_DEFAULT variable

Default size of Character type.

Syntax

final short **Domain.VARCHAR_DEFAULT**

VARCHAR_MIN variable

Minimum size of Character type.

Syntax

final short **Domain.VARCHAR_MIN**

getName function

Returns the name of the domain.

Syntax

String **Domain.getName()**

Returns

The domain name.

getPrecision function

Returns the precision of the domain value.

Syntax

int **Domain.getPrecision()**

Returns

The value precision.

getScale function

Returns the scale of the domain value.

Syntax

int **Domain.getScale()**

Returns

The value scale.

getSize function

Returns the size of the domain value.

Syntax

short **Domain.getSize()**

Returns

The value size.

getType function

Returns the type of the domain.

Syntax

short **Domain.getType()**

Returns

The domain type expressed as an integer.

EncryptionControl interface

Provides encryption control for the database.

Syntax

public **EncryptionControl**

Remarks

This interface is used to implement your own encryption or obfuscation techniques. To encrypt a database, create a new class that implements EncryptionControl, supply the class with your own encryption methods, and use the setEncryption method from the ConfigPersistent interface to initiate a new instance of your EncryptionControl class.

See also

- "setEncryption function" on page 113

Members

All members of EncryptionControl, including all inherited members.

- "decrypt function" on page 166
- "encrypt function" on page 167
- "initialize function" on page 167

decrypt function

Decrypts a byte array in the database.

Syntax

void **EncryptionControl.decrypt(**
  int *page_no*,
  byte[] *src*,
  byte[] *tgt*
) throws **ULjException**

Parameters

- **page_no** The page number of the array data.
- **src** The encrypted source page.
- **tgt** The resulting page that is decrypted by the method.

Remarks

This method is supplied with an encrypted byte array, src, and an associated page number. Your method must decrypt src and store the result into the tgt byte array. tgt is then used for data operations within your application.

encrypt function

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.
- **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.

initialize function

Initializes the encryption control with a password.

Syntax

void **EncryptionControl.initialize(**
  String *password*
) throws **ULjException**

Parameters

- **password** The password used for encryption and decryption.

ForeignKeySchema interface

Specifies the schema of a foreign key.

Syntax

public **ForeignKeySchema**

Remarks

An object supporting this interface is returned by the Connection.createForeignKey(String) method.

All foreign keys must have at least one column reference. The set of referenced columns must be columns in the primary table and the set must be subject to a primary or unique key constraint on the primary table.

The following example demonstrates the creation of the schema for a simple database. The Invoices table has a foreign key into the Products table, which specifies that all invoices should reference valid product IDs.

```
TableSchema table_schema;
IndexSchema index_schema;
ForeignKeySchema fkey_schema;

table_schema = conn.createTable("Invoices");
table_schema.createColumn("inv_id", Domain.INTEGER);
table_schema.createColumn("quantity", Domain.INTEGER);
table_schema.createColumn("sold_prod_id", Domain.INTEGER);

index_schema = table_schema.createPrimaryIndex("primary");
index_schema.addColumn("inv_id", IndexSchema.ASCENDING);

table_schema = conn.createTable("Products");
table_schema.createColumn("prod_id", Domain.INTEGER);
table_schema.createColumn("prod_name", Domain.VARCHAR, 40);

index_schema = table_schema.createPrimaryIndex("primary");
index_schema.addColumn("prod_id", IndexSchema.ASCENDING);

fkey_schema = conn.createForeignKey(
    "Invoices", "Products", "InvoiceToProduct" );
fkey_schema.addColumnReference("sold_prod_is", "prod_id");

conn.schemaCreateComplete();
```

Members

All members of ForeignKeySchema, including all inherited members.

- “addColumnReference function” on page 168

addColumnReference function

Adds a column reference to the foreign key.

Syntax

```
ForeignKeySchema ForeignKeySchema.addColumnReference(
  String foreign_column,
  String primary_column
) throws ULjException
```

Parameters

- **foreign_column** The name of column containing the foreign key. The values in this column are used to reference values in the primary_column_name column of the primary table.
- **primary_column** The name of column in the referenced table. All primary columns, as a set, must be in a primary or unique key constraint of the primary table.

Returns

This ForeignKeySchema with the foreign key assigned to the foreign column.

IndexSchema interface

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

Syntax

public **IndexSchema**

Remarks

An object supporting this interface is returned by the TableSchema.createIndex(String), TableSchema.createPrimaryIndex(String), TableSchema.createUniqueIndex(String) and TableSchema.createUniqueKey(String) methods. For a description of the various index types, see TableSchema interface.

All indexes must have at least one column.

Indexes sort by the first column added to the index, then by the second column (if specified), etc.

Indexes may not contain LONGBINARY or LONGVARCHAR type columns.

The following example demonstrates the creation of a two column index.

```
// Assumes a valid TableSchema object table_schema on
// a table with columns A and B.
IndexSchema index_schema;
index_schema = table_schema.createIndex("AthenBreversed");
index_schema.addColumn("A", IndexSchema.ASCENDING);
index_schema.addColumn("B", IndexSchema.DESCENDING);
```

Members

All members of IndexSchema, including all inherited members.

- “addColumn function” on page 172
- “ASCENDING variable” on page 170
- “DESCENDING variable” on page 171
- “PERSISTENT variable” on page 171
- “PRIMARY_INDEX variable” on page 171
- “UNIQUE_INDEX variable” on page 171
- “UNIQUE_KEY variable” on page 171

ASCENDING variable

The index is sorted in ascending order for a column.

Syntax

final byte **IndexSchema.ASCENDING**

DESCENDING variable

The index is sorted in descending order for a column.

Syntax

final byte **IndexSchema.DESCENDING**

PERSISTENT variable

Bit flag denoting an index is persistent.

Syntax

final byte **IndexSchema.PERSISTENT**

Remarks

This value can be logically combined with other flags in table SYS_INDEXES' index_flags column.

PRIMARY_INDEX variable

Bit flag denoting an index is a primary key.

Syntax

final byte **IndexSchema.PRIMARY_INDEX**

Remarks

This value can be logically combined with other flags in table SYS_INDEXES' index_flags column.

UNIQUE_INDEX variable

Bit flag denoting an index is a unique index.

Syntax

final byte **IndexSchema.UNIQUE_INDEX**

Remarks

This value can be logically combined with other flags in table SYS_INDEXES' index_flags column.

UNIQUE_KEY variable

Bit flag denoting an index is a unique key.

Syntax

final byte **IndexSchema.UNIQUE_KEY**

Remarks

This value can be logically combined with other flags in table SYS_INDEXES' index_flags column.

addColumn function

Adds a column to the index.

Syntax

IndexSchema **IndexSchema.addColumn(**
  String *column_name*,
  byte *sort_order*
) throws **ULjException**

Parameters

- **column_name** The name of the column to add. The specified column must be a column of the table on which this index is being created.
- **sort_order** A constant determining the sort order. Must be IndexSchema.ASCENDING or IndexSchema.DESCENDING.

Remarks

The order that columns are added to the index determines the sorts precedence. The first column has highest precedence.

Returns

This IndexSchema with the added column.

PreparedStatement interface

Provides methods to execute a SQL query to generate a ResultSet, or to execute a prepared SQL statement on an UltraLite database.

Syntax

public **PreparedStatement**

Base classes

- "CollectionOfValueWriters interface" on page 97

Remarks

The following example demonstrates how to execute a PreparedStatement, check if the execution created a ResultSet, save the ResultSet to a local variable, and close the PreparedStatement:

```
// Create a new PreparedStatement object from an existing connection.
String sql_string = "SELECT * FROM SampleTable";
PreparedStatement ps = conn.prepareStatement(sql_string);

// result returns true if the execute statement runs successfully.
boolean result = ps.execute();

// Check if the PreparedStatement contains a ResultSet.
if (ps.hasResultSet()) {
    // Store the ResultSet in the rs variable.
    ResultSet rs = ps.getResultSet;
}

// Close the PreparedStatement to release resources.
ps.close();
```

Members

All members of PreparedStatement, including all inherited members.

- "close function" on page 174
- "execute function" on page 174
- "executeQuery function" on page 174
- "getBlobOutputStream function" on page 98
- "getClobWriter function" on page 98
- "getOrdinal function" on page 98
- "getPlan function" on page 175
- "getResultSet function" on page 175
- "getUpdateCount function" on page 175
- "hasResultSet function" on page 175
- "set function" on page 99
- "set function" on page 99
- "set function" on page 99
- "set function" on page 100
- "set function" on page 100
- "set function" on page 100
- "set function" on page 101

- "set function" on page 101
- "set function" on page 101
- "set function" on page 102
- "setNull function" on page 102

close function

Closes the PreparedStatement to release the memory resources associated with it.

Syntax

void **PreparedStatement.close()** throws **ULjException**

Remarks

No further methods can be used on this object. If the PreparedStatement contains a ResultSet, the ResultSet also gets closed.

execute function

Executes the prepared SQL statement.

Syntax

boolean **PreparedStatement.execute()** throws **ULjException**

See also

- ResultSet interface

Returns

true, if the execute statement runs successfully; otherwise, false.

executeQuery function

Executes the prepared SQL SELECT statement and returns a ResultSet.

Syntax

ResultSet **PreparedStatement.executeQuery()** throws **ULjException**

See also

- ResultSet interface

Returns

The ResultSet containing the query result of the prepared SQL SELECT statement.

getPlan function

Returns a text-based description of the SQL query execution plan.

Syntax

String **PreparedStatement.getPlan()** throws **ULjException**

Remarks

An empty string is returned if there is no plan.

Returns

The String representation of the plan.

getResultSet function

Returns the ResultSet for a prepared SQL statement.

Syntax

ResultSet **PreparedStatement.getResultSet()** throws **ULjException**

See also

- ResultSet interface

Returns

The ResultSet containing the query result of the prepared SQL. statement.

getUpdateCount function

Returns the number of rows inserted, updated or deleted since the last commit statement.

Syntax

int **PreparedStatement.getUpdateCount()** throws **ULjException**

Returns

-1 if no changes were made; otherwise, the number of updated rows.

hasResultSet function

Determines if the PreparedStatement contains a ResultSet.

Syntax

boolean **PreparedStatement.hasResultSet()** throws **ULjException**

See also

- ResultSet interface

Returns

true, if a ResultSet was found; otherwise, false.

ResultSet interface

Provides methods to traverse a table by row, and access the column data.

Syntax

public **ResultSet**

Base classes

- "CollectionOfValueReaders interface" on page 91

Remarks

A ResultSet is generated by executing a PreparedStatement with a SQL SELECT statement using the execute or executeQuery methods.

The following example demonstrates how to execute a new PreparedStatement, fetch a row with the ResultSet, and access data from a specified column.

```
// Define a new SQL SELECT statement.
String sql_string = "SELECT column1, column2 FROM SampleTable";

// Create a new PreparedStatement from an existing connection.
PreparedStatement ps = conn.prepareStatement(sql_string);

// Create a new ResultSet to contain the query results of the SQL statement.
ResultSet rs = ps.executeQuery();

// Check if the PreparedStatement contains a ResultSet.
if (ps.hasResultSet()) {
    // Retrieve the column1 value from the first row using getString.
    String row1_col1 = rs.getString(1);
    // Get the next row in the table.
    if (rs.next) {
        // Retrieve the value of column1 from the second row.
        String row2_col1 = rs.getString(1);
    }
}
```

Members

All members of ResultSet, including all inherited members.

- "close function" on page 178
- "getBlobInputStream function" on page 92
- "getBoolean function" on page 92
- "getBytes function" on page 92
- "getClobReader function" on page 93
- "getDate function" on page 93
- "getDecimalNumber function" on page 93
- "getDouble function" on page 94
- "getFloat function" on page 94
- "getInt function" on page 94
- "getLong function" on page 95
- "getOrdinal function" on page 95

- "getResultSetMetadata function" on page 178
- "getString function" on page 95
- "getValue function" on page 96
- "isNull function" on page 96
- "next function" on page 178
- "previous function" on page 179

close function

Closes the ResultSet to release the memory resources associated with it.

Syntax

void **ResultSet.close()** throws **ULjException**

Remarks

An error is thrown when subsequent attempts are made to fetch rows from this ResultSet.

getResultSetMetadata function

Returns the ResultSetMetadata containing the meta data for the ResultSet.

Syntax

ResultSetMetadata **ResultSet.getResultSetMetadata()** throws **ULjException**

Returns

The ResultSetMetadata object.

next function

Fetches the next row of data in the ResultSet.

Syntax

boolean **ResultSet.next()** throws **ULjException**

See also

- ResultSetMetadata interface

Returns

true when the next row is successfully fetched; otherwise, false.

previous function

Fetches the previous row of data in the ResultSet.

Syntax

boolean **ResultSet.previous()** throws **ULjException**

Returns

true when the previous row is successfully fetched; otherwise, false.

ResultSetMetadata interface

Associated with a ResultSet object and contains a method that provides column information.

Syntax

public **ResultSetMetadata**

Remarks

This interface is invoked using the getResultSetMetadata method of a ResultSet object.

Members

All members of ResultSetMetadata, including all inherited members.

- "getColumnCount function" on page 180

getColumnCount function

Returns the total number of columns in the ResultSet.

Syntax

int **ResultSetMetadata.getColumnCount()** throws **ULjException**

Returns

The number of columns.

SISListener interface (J2ME BlackBerry only)

Listens for server-initiated synchronization messages.

Syntax

public **SISListener**

Remarks

The application creates an instance of SISListener using the appropriate createSISHTTPListener method in the DatabaseManager interface.

Members

All members of SISListener, including all inherited members.

- "startListening function" on page 181
- "stopListening function" on page 181

startListening function

Creates and starts the listening thread.

Syntax

void **SISListener.startListening()**

stopListening function

Stops the listening thread.

Syntax

void **SISListener.stopListening()**

SISRequestHandler interface (J2ME BlackBerry only)

Handles server-initiated synchronization requests.

Syntax

public **SISRequestHandler**

Members

All members of SISRequestHandler, including all inherited members.

- "onError function" on page 182
- "onRequest function" on page 182

onError function

Handles SIS requests on worker threads.

Syntax

void **SISListener.onError(**
  String *text*
)

Parameters

- **text** The string sent by the request.

onRequest function

Handles SIS related errors that occur during SIS listening.

Syntax

void **SISListener.onRequest(**
  String *text*
)

Parameters

- **text** The string representation of the exception.

Remarks

To stop listening, explicitly call the stopListening method in the SISListener interface.

SQLCode interface

Enumerates the SQL codes that may be reported by UltraLiteJ.

Syntax

public **SQLCode**

Derived classes

- "ULjException class" on page 243

Remarks

For a detailed description of each error, please refer to the "SQL Anywhere error messages sorted by SQLCODE" topic in the SQL Anywhere documentation set.

Members

All members of SQLCode, including all inherited members.

- "SQLE_UNCOMMITTED_TRANSACTIONS variable" on page 199
- "SQLE_UNDERFLOW variable" on page 199
- "SQLE_UNKNOWN_FUNC variable" on page 199
- "SQLE_UPLOAD_FAILED_AT_SERVER variable" on page 199
- "SQLE_VALUE_IS_NULL variable" on page 200
- "SQLE_VARIABLE_INVALID variable" on page 200
- "SQLE_WRONG_NUM_OF_INSERT_COLS variable" on page 200
- "SQLE_WRONG_PARAMETER_COUNT variable" on page 200

SQLE_AGGREGATES_NOT_ALLOWED variable

SQLE_AGGREGATES_NOT_ALLOWED(-150).

Syntax

final int **SQLCode.SQLE_AGGREGATES_NOT_ALLOWED**

SQLE_ALIAS_NOT_UNIQUE variable

SQLE_ALIAS_NOT_UNIQUE(-830).

Syntax

final int **SQLCode.SQLE_ALIAS_NOT_UNIQUE**

SQLE_ALIAS_NOT_YET_DEFINED variable

SQLE_ALIAS_NOT_YET_DEFINED(-831).

Syntax

final int **SQLCode.SQLE_ALIAS_NOT_YET_DEFINED**

SQLE_AUTHENTICATION_FAILED variable

SQLE_AUTHENTICATION_FAILED(-218).

Syntax

final int **SQLCode.SQLE_AUTHENTICATION_FAILED**

SQLE_CANNOT_EXECUTE_STMT variable

SQLE_CANNOT_EXECUTE_STMT(111).

Syntax

final int **SQLCode.SQLE_CANNOT_EXECUTE_STMT**

SQLE_CLIENT_OUT_OF_MEMORY variable

SQLE_CLIENT_OUT_OF_MEMORY(-876).

Syntax

final int **SQLCode.SQLE_CLIENT_OUT_OF_MEMORY**

SQLE_COLUMN_AMBIGUOUS variable

SQLE_COLUMN_AMBIGUOUS(-144).

Syntax

final int **SQLCode.SQLE_COLUMN_AMBIGUOUS**

SQLE_COLUMN_CANNOT_BE_NULL variable

SQLE_COLUMN_CANNOT_BE_NULL(-195).

Syntax

final int **SQLCode.SQLE_COLUMN_CANNOT_BE_NULL**

SQLE_COLUMN_NOT_FOUND variable

SQLE_COLUMN_NOT_FOUND(-143).

Syntax

final int **SQLCode.SQLE_COLUMN_NOT_FOUND**

SQLE_COLUMN_NOT_STREAMABLE variable

SQLE_COLUMN_NOT_STREAMABLE(-1100).

Syntax

final int **SQLCode.SQLE_COLUMN_NOT_STREAMABLE**

SQLE_COMMUNICATIONS_ERROR variable

SQLE_COMMUNICATIONS_ERROR(-85).

Syntax

final int **SQLCode.SQLE_COMMUNICATIONS_ERROR**

SQLE_CONFIG_IN_USE variable

SQLE_CONFIG_IN_USE(-1276).

Syntax

final int **SQLCode.SQLE_CONFIG_IN_USE**

SQLE_CONVERSION_ERROR variable

SQLE_CONVERSION_ERROR(-157).

Syntax

final int **SQLCode.SQLE_CONVERSION_ERROR**

SQLE_CURSOR_ALREADY_OPEN variable

SQLE_CURSOR_ALREADY_OPEN(-172).

Syntax

final int **SQLCode.SQLE_CURSOR_ALREADY_OPEN**

SQLE_DATABASE_ACTIVE variable

SQLE_DATABASE_ACTIVE(-664).

Syntax

final int **SQLCode.SQLE_DATABASE_ACTIVE**

SQLE_DEVICE_IO_FAILED variable

SQLE_DEVICE_IO_FAILED(-974).

Syntax

final int **SQLCode.SQLE_DEVICE_IO_FAILED**

SQLE_DIV_ZERO_ERROR variable

SQLE_DIV_ZERO_ERROR(-628).

Syntax

final int **SQLCode.SQLE_DIV_ZERO_ERROR**

SQLE_DOWNLOAD_CONFLICT variable

SQLE_DOWNLOAD_CONFLICT(-839).

Syntax

final int **SQLCode.SQLE_DOWNLOAD_CONFLICT**

SQLE_ERROR variable

SQLE_ERROR(-300).

Syntax

final int **SQLCode.SQLE_ERROR**

SQLE_EXISTING_PRIMARY_KEY variable

SQLE_EXISTING_PRIMARY_KEY(-112).

Syntax

final int **SQLCode.SQLE_EXISTING_PRIMARY_KEY**

SQLE_EXPRESSION_ERROR variable

SQLE_EXPRESSION_ERROR(-156).

Syntax

final int **SQLCode.SQLE_EXPRESSION_ERROR**

SQLE_FILE_BAD_DB variable

SQLE_FILE_BAD_DB(-1006).

Syntax

final int **SQLCode.SQLE_FILE_BAD_DB**

SQLE_FILE_WRONG_VERSION variable

SQLE_FILE_WRONG_VERSION(-1005).

Syntax

final int **SQLCode.SQLE_FILE_WRONG_VERSION**

SQLE_FOREIGN_KEY_NAME_NOT_FOUND variable

SQLE_FOREIGN_KEY_NAME_NOT_FOUND(-145).

Syntax

final int **SQLCode.SQLE_FOREIGN_KEY_NAME_NOT_FOUND**

SQLE_IDENTIFIER_TOO_LONG variable

SQLE_IDENTIFIER_TOO_LONG(-250).

Syntax

final int **SQLCode.SQLE_IDENTIFIER_TOO_LONG**

SQLE_INCOMPLETE_SYNCHRONIZATION variable

SQLE_INCOMPLETE_SYNCHRONIZATION(-1271).

Syntax

final int **SQLCode.SQLE_INCOMPLETE_SYNCHRONIZATION**

SQLE_INDEX_HAS_NO_COLUMNS variable

SQLE_INDEX_HAS_NO_COLUMNS(-1274).

Syntax

final int **SQLCode.SQLE_INDEX_HAS_NO_COLUMNS**

SQLE_INDEX_NOT_FOUND variable

SQLE_INDEX_NOT_FOUND(-183).

Syntax

final int **SQLCode.SQLE_INDEX_NOT_FOUND**

SQLE_INDEX_NOT_UNIQUE variable

SQLE_INDEX_NOT_UNIQUE(-196).

Syntax

final int **SQLCode.SQLE_INDEX_NOT_UNIQUE**

SQLE_INTERRUPTED variable

SQLE_INTERRUPTED(-299).

Syntax

final int **SQLCode.SQLE_INTERRUPTED**

SQLE_INVALID_COMPARISON variable

SQLE_INVALID_COMPARISON(-710).

Syntax

final int **SQLCode.SQLE_INVALID_COMPARISON**

SQLE_INVALID_DISTINCT_AGGREGATE variable

SQLE_INVALID_DISTINCT_AGGREGATE(-863).

Syntax

final int **SQLCode.SQLE_INVALID_DISTINCT_AGGREGATE**

SQLE_INVALID_DOMAIN variable

SQLE_INVALID_DOMAIN(-1275).

Syntax

final int **SQLCode.SQLE_INVALID_DOMAIN**

SQLE_INVALID_FOREIGN_KEY_DEF variable

SQLE_INVALID_FOREIGN_KEY_DEF(-113).

Syntax

final int **SQLCode.SQLE_INVALID_FOREIGN_KEY_DEF**

SQLE_INVALID_GROUP_SELECT variable

SQLE_INVALID_GROUP_SELECT(-149).

Syntax

final int **SQLCode.SQLE_INVALID_GROUP_SELECT**

SQLE_INVALID_INDEX_TYPE variable

SQLE_INVALID_INDEX_TYPE(-650).

Syntax

final int **SQLCode.SQLE_INVALID_INDEX_TYPE**

SQLE_INVALID_LOGON variable

SQLE_INVALID_LOGON(-103).

Syntax

final int **SQLCode.SQLE_INVALID_LOGON**

SQLE_INVALID_OPTION variable

SQLE_INVALID_OPTION(-200).

Syntax

final int **SQLCode.SQLE_INVALID_OPTION**

SQLE_INVALID_OPTION_SETTING variable

SQLE_INVALID_OPTION_SETTING(-201).

Syntax

final int **SQLCode.SQLE_INVALID_OPTION_SETTING**

SQLE_INVALID_ORDER variable

SQLE_INVALID_ORDER(-152).

Syntax

final int **SQLCode.SQLE_INVALID_ORDER**

SQLE_INVALID_PARAMETER variable

SQLE_INVALID_PARAMETER(-735).

Syntax

final int **SQLCode.SQLE_INVALID_PARAMETER**

SQLE_INVALID_UNION variable

SQLE_INVALID_UNION(-153).

Syntax

final int **SQLCode.SQLE_INVALID_UNION**

SQLE_LOCKED variable

SQLE_LOCKED(-210).

Syntax

final int **SQLCode.SQLE_LOCKED**

SQLE_MAX_ROW_SIZE_EXCEEDED variable

SQLE_MAX_ROW_SIZE_EXCEEDED(-1132).

Syntax

final int **SQLCode.SQLE_MAX_ROW_SIZE_EXCEEDED**

SQLE_MUST_BE_ONLY_CONNECTION variable

SQLE_MUST_BE_ONLY_CONNECTION(-211).

Syntax

final int **SQLCode.SQLE_MUST_BE_ONLY_CONNECTION**

SQLE_NAME_NOT_UNIQUE variable

SQLE_NAME_NOT_UNIQUE(-110).

Syntax

final int **SQLCode.SQLE_NAME_NOT_UNIQUE**

SQLE_NOERROR variable

SQLE_NOERROR(0).

Syntax

final int **SQLCode.SQLE_NOERROR**

SQLE_NOT_IMPLEMENTED variable

SQLE_NOT_IMPLEMENTED(-134).

Syntax

final int **SQLCode.SQLE_NOT_IMPLEMENTED**

SQLE_NO_COLUMN_NAME variable

SQLE_NO_COLUMN_NAME(-163).

Syntax

final int **SQLCode.SQLE_NO_COLUMN_NAME**

SQLE_NO_CURRENT_ROW variable

SQLE_NO_CURRENT_ROW(-197).

Syntax

final int **SQLCode.SQLE_NO_CURRENT_ROW**

SQLE_NO_MATCHING_SELECT_ITEM variable

SQLE_NO_MATCHING_SELECT_ITEM(-812).

Syntax

final int **SQLCode.SQLE_NO_MATCHING_SELECT_ITEM**

SQLE_NO_PRIMARY_KEY variable

SQLE_NO_PRIMARY_KEY(-118).

Syntax

final int **SQLCode.SQLE_NO_PRIMARY_KEY**

SQLE_OVERFLOW_ERROR variable

SQLE_OVERFLOW_ERROR(-158).

Syntax

final int **SQLCode.SQLE_OVERFLOW_ERROR**

SQLE_PAGE_SIZE_TOO_BIG variable

SQLE_PAGE_SIZE_TOO_BIG(-97).

Syntax

final int **SQLCode.SQLE_PAGE_SIZE_TOO_BIG**

SQLE_PAGE_SIZE_TOO_SMALL variable

SQLE_PAGE_SIZE_TOO_SMALL(-972).

Syntax

final int **SQLCode.SQLE_PAGE_SIZE_TOO_SMALL**

SQLE_PARAMETER_CANNOT_BE_NULL variable

SQLE_PARAMETER_CANNOT_BE_NULL(-1277).

Syntax

final int **SQLCode.SQLE_PARAMETER_CANNOT_BE_NULL**

SQLE_PERMISSION_DENIED variable

SQLE_PERMISSION_DENIED(-121).

Syntax

final int **SQLCode.SQLE_PERMISSION_DENIED**

SQLE_PRIMARY_KEY_NOT_UNIQUE variable

SQLE_PRIMARY_KEY_NOT_UNIQUE(-193).

Syntax

final int **SQLCode.SQLE_PRIMARY_KEY_NOT_UNIQUE**

SQLE_PUBLICATION_NOT_FOUND variable

SQLE_PUBLICATION_NOT_FOUND(-280).

Syntax

final int **SQLCode.SQLE_PUBLICATION_NOT_FOUND**

SQLE_RESOURCE_GOVERNOR_EXCEEDED variable

SQLE_RESOURCE_GOVERNOR_EXCEEDED(-685).

Syntax

final int **SQLCode.SQLE_RESOURCE_GOVERNOR_EXCEEDED**

SQLE_ROW_LOCKED variable

SQLE_ROW_LOCKED(-1281).

Syntax

final int **SQLCode.SQLE_ROW_LOCKED**

SQLE_ROW_UPDATED_SINCE_READ variable

SQLE_ROW_UPDATED_SINCE_READ(-208).

Syntax

final int **SQLCode.SQLE_ROW_UPDATED_SINCE_READ**

SQLE_SCHEMA_UPGRADE_NOT_ALLOWED variable

SQLE_SCHEMA_UPGRADE_NOT_ALLOWED(-953).

Syntax

final int **SQLCode.SQLE_SCHEMA_UPGRADE_NOT_ALLOWED**

SQLE_SERVER_SYNCHRONIZATION_ERROR variable

SQLE_SERVER_SYNCHRONIZATION_ERROR(-857).

Syntax

final int **SQLCode.SQLE_SERVER_SYNCHRONIZATION_ERROR**

SQLE_SUBQUERY_RESULT_NOT_UNIQUE variable

SQLE_SUBQUERY_RESULT_NOT_UNIQUE(-186).

Syntax

final int **SQLCode.SQLE_SUBQUERY_RESULT_NOT_UNIQUE**

SQLE_SUBQUERY_SELECT_LIST variable

SQLE_SUBQUERY_SELECT_LIST(-151).

Syntax

final int **SQLCode.SQLE_SUBQUERY_SELECT_LIST**

SQLE_SYNCHRONIZATION_IN_PROGRESS variable

SQLE_SYNCHRONIZATION_IN_PROGRESS(-1272).

Syntax

final int **SQLCode.SQLE_SYNCHRONIZATION_IN_PROGRESS**

SQLE_SYNC_INFO_INVALID variable

SQLE_SYNC_INFO_INVALID(-956).

Syntax

final int **SQLCode.SQLE_SYNC_INFO_INVALID**

SQLE_SYNTAX_ERROR variable

SQLE_SYNTAX_ERROR(-131).

Syntax

final int **SQLCode.SQLE_SYNTAX_ERROR**

SQLE_TABLE_HAS_NO_COLUMNS variable

SQLE_TABLE_HAS_NO_COLUMNS(-1273).

Syntax

final int **SQLCode.SQLE_TABLE_HAS_NO_COLUMNS**

SQLE_TABLE_IN_USE variable

SQLE_TABLE_IN_USE(-214).

Syntax

final int **SQLCode.SQLE_TABLE_IN_USE**

SQLE_TABLE_NOT_FOUND variable

SQLE_TABLE_NOT_FOUND(-141).

Syntax

final int **SQLCode.SQLE_TABLE_NOT_FOUND**

SQLE_TOO_MANY_PUBLICATIONS variable

SQLE_TOO_MANY_PUBLICATIONS(-1106).

Syntax

final int **SQLCode.SQLE_TOO_MANY_PUBLICATIONS**

SQLE_ULTRALITEJ_OPERATION_FAILED variable

SQLE_ULTRALITEJ_OPERATION_FAILED(-1279).

Syntax

final int **SQLCode.SQLE_ULTRALITEJ_OPERATION_FAILED**

SQLE_ULTRALITEJ_OPERATION_NOT_ALLOWED variable

SQLE_ULTRALITEJ_OPERATION_NOT_ALLOWED(-1278).

Syntax

final int **SQLCode.SQLE_ULTRALITEJ_OPERATION_NOT_ALLOWED**

SQLE_ULTRALITE_DATABASE_NOT_FOUND variable

SQLE_ULTRALITE_DATABASE_NOT_FOUND(-954).

Syntax

final int **SQLCode.SQLE_ULTRALITE_DATABASE_NOT_FOUND**

SQLE_ULTRALITE_OBJ_CLOSED variable

SQLE_ULTRALITE_OBJ_CLOSED(-908).

Syntax

final int **SQLCode.SQLE_ULTRALITE_OBJ_CLOSED**

SQLE_UNABLE_TO_CONNECT variable

SQLE_UNABLE_TO_CONNECT(-105).

Syntax

final int **SQLCode.SQLE_UNABLE_TO_CONNECT**

SQLE_UNCOMMITTED_TRANSACTIONS variable

SQLE_UNCOMMITTED_TRANSACTIONS(-755).

Syntax

final int **SQLCode.SQLE_UNCOMMITTED_TRANSACTIONS**

SQLE_UNDERFLOW variable

SQLE_UNDERFLOW(-1280).

Syntax

final int **SQLCode.SQLE_UNDERFLOW**

SQLE_UNKNOWN_FUNC variable

SQLE_UNKNOWN_FUNC(-148).

Syntax

final int **SQLCode.SQLE_UNKNOWN_FUNC**

SQLE_UPLOAD_FAILED_AT_SERVER variable

SQLE_UPLOAD_FAILED_AT_SERVER(-794).

Syntax

final int **SQLCode.SQLE_UPLOAD_FAILED_AT_SERVER**

SQLE_VALUE_IS_NULL variable

SQLE_VALUE_IS_NULL(-1050).

Syntax

final int **SQLCode.SQLE_VALUE_IS_NULL**

SQLE_VARIABLE_INVALID variable

SQLE_VARIABLE_INVALID(-155).

Syntax

final int **SQLCode.SQLE_VARIABLE_INVALID**

SQLE_WRONG_NUM_OF_INSERT_COLS variable

SQLE_WRONG_NUM_OF_INSERT_COLS(-207).

Syntax

final int **SQLCode.SQLE_WRONG_NUM_OF_INSERT_COLS**

SQLE_WRONG_PARAMETER_COUNT variable

SQLE_WRONG_PARAMETER_COUNT(-154).

Syntax

final int **SQLCode.SQLE_WRONG_PARAMETER_COUNT**

StreamHTTPParms interface

Represents HTTP stream parameters that define how to communicate with a MobiLink server using HTTP.

Syntax

public **StreamHTTPParms**

Derived classes

- "StreamHTTPSParms interface" on page 205

Remarks

The following example sets the stream parameters to communicate with a MobiLink 11 server on host name "MyMLHost". The server started with the following parameters: "-xo http(port=1234)":

```
SyncParms syncParms = myConnection.createSyncParms(
    SyncParms.HTTP,
    "MyUniqueMLUserID",
    "MyMLScriptVersion"
);
StreamHTTPParms httpParms = syncParms.getStreamParms();
httpParms.setHost("MyMLHost");
httpParms.setPort(1234);
```

Instances implementing this interface are returned by the getStreamParms function method.

Members

All members of StreamHTTPParms, including all inherited members.

- "getHost function" on page 201
- "getOutputBufferSize function" on page 202
- "getPort function" on page 202
- "getURLSuffix function" on page 202
- "setHost function" on page 203
- "setOutputBufferSize function" on page 203
- "setPort function" on page 204
- "setURLSuffix function" on page 204

getHost function

Returns the host name of the MobiLink server.

Syntax

String **StreamHTTPParms.getHost()**

See also

- setHost function
- getPort function
- setPort function

Returns

The name of the host.

getOutputBufferSize function

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()**

Remarks

Increasing this value may reduce the number of network flushes needed to send a large upload at the cost of increased memory use. In HTTP, each flush sends a large (approximately 250 bytes) HTTP header; reducing the number of flushes can reduce the bandwidth use.

See also

- setOutputBufferSize function

Returns

The integer containing the buffer size.

getPort function

Returns the port number used to connect to the MobiLink server.

Syntax

int **StreamHTTPParms.getPort()**

See also

- setPort function

Returns

The port number of MobiLink server.

getURLSuffix function

Returns the String containing the URL suffix..

Syntax

String **StreamHTTPParms.getURLSuffix()**

See also

- setURLSuffix function

Returns

The String containing the URL suffix.

setHost function

Sets the host name of the MobiLink server.

Syntax

void **StreamHTTPParms.setHost(**
String *v*
)

Parameters

- **v** The name of the host.

Remarks

The default is null, which indicates a localhost.

See also

- getHost function
- getPort function
- setPort function

setOutputBufferSize function

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

- getOutputBufferSize function

setPort function

Sets the port number used to connect to the MobiLink server.

Syntax

void **StreamHTTPParms.setPort(**
  int *v*
)

Parameters

- **v** A port number ranging from 1 to 65535. Out of range values revert to default value.

Remarks

The default port is 80 for HTTP synchronizations and 443 for HTTPS synchronizations.

See also

- getPort function

setURLSuffix function

Sets the URL suffix of the MobiLink server.

Syntax

void **StreamHTTPParms.setURLSuffix(**
  String *v*
)

Parameters

- **v** The URL suffix string.

Remarks

The default is null, which implies "Mobilink/".

See also

- getURLSuffix function

StreamHTTPSParms interface

Represents HTTPS stream parameters that define how to communicate with a MobiLink server using secure HTTPS.

Syntax

public **StreamHTTPSParms**

Base classes

- "StreamHTTPParms interface" on page 201

Remarks

The following example sets the stream parameters to communicate with a MobiLink 11 server on host name "MyMLHost". The server started with the following parameters: "-xo https(port=1234;certificate=RSAServer.crt;certificate_password=x)"

```
SyncParms syncParms = myConnection.createSyncParms(
    SyncParms.HTTPS,
    "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 by using one of the following methods:

1. Install the trusted root certificate in the lib/security/cacerts key store of the JRE.

2. Build your own key store using the Java keytool utility and setting the Java system property javax.net.ssl.trustStore to its location (set javax.net.ssl.trustStorePassword to an appropriate value).

3. Using the setTrustedCertificates function parameter to point to the deployed certificate file.

To enhance security, the setCertificateName, setCertificateCompany setCertificateUnit methods should be used to turn on validation of the MobiLink server certificate.

Instances implementing this interface are returned by the getStreamParms function when the SyncParms class object is created for HTTPS synchronization.

Members

All members of StreamHTTPSParms, including all inherited members.

- "getCertificateCompany function" on page 206
- "getCertificateName function" on page 206
- "getCertificateUnit function" on page 206
- "getHost function" on page 201
- "getOutputBufferSize function" on page 202

- "getPort function" on page 202
- "getTrustedCertificates function" on page 207
- "getURLSuffix function" on page 202
- "setCertificateCompany function" on page 207
- "setCertificateName function" on page 207
- "setCertificateUnit function" on page 207
- "setHost function" on page 203
- "setOutputBufferSize function" on page 203
- "setPort function" on page 204
- "setTrustedCertificates function" on page 208
- "setURLSuffix function" on page 204

getCertificateCompany function

Returns the certificate company name for verification of secure connections.

Syntax

String **StreamHTTPSParms.getCertificateCompany()**

Returns

The certificate company name.

getCertificateName function

Returns the certificate common name for verification of secure connections.

Syntax

String **StreamHTTPSParms.getCertificateName()**

Returns

The certificate name.

getCertificateUnit function

Returns the certificate unit name for verification of secure connections.

Syntax

String **StreamHTTPSParms.getCertificateUnit()**

Returns

The organization unit name.

getTrustedCertificates function

Returns the name of the file containing a list of trusted root certificates used for secure synchronization.

Syntax

String **StreamHTTPSParms.getTrustedCertificates()**

Returns

The filename of the trusted root certificates file.

setCertificateCompany function

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 function

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 function

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 function

Sets a file containing a list of trusted root certificates used for secure synchronization.

Syntax

void **StreamHTTPSParms.setTrustedCertificates(**
String *filename*
) throws **ULjException**

Parameters

- **filename** The filename of the trusted root certificate.

Remarks

This parameter is only used on J2SE systems.

The default is null, indicating that the system default certificate store is used to validate certificate chains from the MobiLink server.

SyncObserver interface

Receives synchronization progress information.

Syntax

public **SyncObserver**

Remarks

To receive progress reports during synchronization, you must create a new class that performs the task, and implement it using the setSyncObserver function.

The following example illustrates a simple SyncObserver interface:

```
class MyObserver implements SyncObserver {
    public boolean syncProgress(int state, SyncResult result) {
        System.out.println(
            "sync progress state = " + state
            + " bytes sent = " + result.getSentByteCount()
            + " bytes received = " + result.getReceivedByteCount()
        );
        return false;   // Always continue synchronization.
    }
    public MyObserver() {} // The default constructor.
}
```

The observer class above is enabled as follows:

```
// J2ME Sample
Connection conn;
ConfigRecordStore config = DatabaseManager.createConfigurationRecordStore(
    "test.ulj"
    );
try {
    conn = DatabaseManager.connect(config);
} catch(ULjException ex) {
    conn = DatabaseManager.createDatabase(config);
    // Create the schema here.
}
SyncParms.setSyncObserver(new MyObserver());
```

Members

All members of SyncObserver, including all inherited members.

- "syncProgress function" on page 209

syncProgress function

Informs the user of progress and is invoked during synchronization.

Syntax

boolean **SyncObserver.syncProgress(**
  int *state*,
  SyncResult *data*
)

Parameters

- **state** One of the SyncOberver.States constants, representing the current state of the synchronization.
- **data** A SyncResult containing the latest synchronization results.

Remarks

The various states, which are signaled as packets, are received and sent. Since multiple tables may be uploaded or downloaded in a single packet, syncProgress calls for any given synchronization may skip a number of states.

> **Note**
> With the exception of the SyncResult class methods, no other UltraLiteJ API methods should be invoked during a syncProgress call.

See also

- SyncObserver.States interface
- setSyncObserver function

Returns

Returns true to cancel the synchronization; otherwise, false to continue synchronization.

SyncObserver.States interface

Defines the synchronization states that can be signaled to an observer.

Syntax

public **SyncObserver.States**

See also

- setSyncObserver function
- SyncObserver interface

Members

All members of SyncObserver.States, including all inherited members.

CHECKING_LAST_UPLOAD variable

Checking the status of the previous upload.

Syntax

final int **SyncObserver.States.CHECKING_LAST_UPLOAD**

COMMITTING_DOWNLOAD variable

Syntax

final int **SyncObserver.States.COMMITTING_DOWNLOAD**

Remarks

The downloaded rows are being committed to the database.

DISCONNECTING variable

The synchronization stream is disconnecting.

Syntax

final int **SyncObserver.States.DISCONNECTING**

DONE variable

Synchronization is complete.

Syntax

final int **SyncObserver.States.DONE**

Remarks

No other states are reported.

ERROR variable

Synchronization is complete but an error occurred.

Syntax

final int **SyncObserver.States.ERROR**

FINISHING_UPLOAD variable

The upload is finalizing.

Syntax

final int **SyncObserver.States.FINISHING_UPLOAD**

RECEIVING_TABLE variable

A new table is being downloaded.

Syntax

final int **SyncObserver.States.RECEIVING_TABLE**

RECEIVING_UPLOAD_ACK variable

An upload acknowledged is being downloaded.

Syntax

final int **SyncObserver.States.RECEIVING_UPLOAD_ACK**

ROLLING_BACK_DOWNLOAD variable

Synchronization is rolling back the download because an error was encountered during the download.

Syntax

final int **SyncObserver.States.ROLLING_BACK_DOWNLOAD**

SENDING_DOWNLOAD_ACK variable

An acknowledgement of a complete download is being sent.

Syntax

final int **SyncObserver.States.SENDING_DOWNLOAD_ACK**

SENDING_HEADER variable

The synchronization stream has been opened and the header is about to be sent.

Syntax

final int **SyncObserver.States.SENDING_HEADER**

SENDING_TABLE variable

A new table is being uploaded.

Syntax

final int **SyncObserver.States.SENDING_TABLE**

STARTING variable

A synchronization is starting. No actions have taken place.

Syntax

final int **SyncObserver.States.STARTING**

SyncParms class

Maintains the parameters used during the database synchronization process.

Syntax

public **SyncParms**

Remarks

This interface is invoked using the createSyncParms method of a Connection interface object.

You can only set one synchronization command at a time. These commands are specified using the setDownloadOnly, setPingOnly, and setUploadOnly methods. By setting one of these methods to true, you set the other methods to false.

The UserName and Version parameters must be set. The UserName must be unique for each client database.

The communication stream is configured using the getStreamParms method based on the type of SyncParms class object. For example, the following code prepares and performs an HTTP synchronization:

```
SyncParms syncParms = myConnection.createSyncParms(
    SyncParms.HTTP,
    "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:

```
syncParms.setTableOrder("'Table A',\"Table B,D\",Table C" );
```

specifies "Table A" then "Table B,D" and then "Table C".

Members

All members of SyncParms, including all inherited members.

- "getAcknowledgeDownload function" on page 217
- "getAuthenticationParms function" on page 217
- "getLivenessTimeout function" on page 217
- "getNewPassword function" on page 218
- "getPassword function" on page 218
- "getPublications function" on page 218
- "getSendColumnNames function" on page 219
- "getStreamParms function" on page 219
- "getSyncObserver function" on page 219
- "getSyncResult function" on page 220
- "getTableOrder function" on page 220
- "getUserName function" on page 220

- "getVersion function" on page 221
- "HTTP_STREAM variable" on page 216
- "HTTPS_STREAM variable" on page 216
- "isDownloadOnly function" on page 221
- "isPingOnly function" on page 221
- "isUploadOnly function" on page 221
- "setAcknowledgeDownload function" on page 222
- "setAuthenticationParms function" on page 222
- "setDownloadOnly function" on page 223
- "setLivenessTimeout function" on page 223
- "setNewPassword function" on page 224
- "setPassword function" on page 224
- "setPingOnly function" on page 225
- "setPublications function" on page 225
- "setSendColumnNames function" on page 226
- "setSyncObserver function" on page 226
- "setTableOrder function" on page 226
- "setUploadOnly function" on page 227
- "setUserName function" on page 227
- "setVersion function" on page 228
- "SyncParms function" on page 217

HTTPS_STREAM variable

Creates a SyncParms class object for secure HTTPS synchronizations.

Syntax

final int **SyncParms.HTTPS_STREAM**

See also

- createSyncParms function

HTTP_STREAM variable

Creates a SyncParms class object for HTTP synchronizations.

Syntax

final int **SyncParms.HTTP_STREAM**

See also

- createSyncParms function

SyncParms function

To create a SyncParms class object, use createSyncParms function.

Syntax

SyncParms.SyncParms()

See also

- createSyncParms function

getAcknowledgeDownload function

Determines if the remote sends download acknowledgements.

Syntax

abstract boolean **SyncParms.getAcknowledgeDownload()**

See also

- setAcknowledgeDownload function

Returns

true, if the remote sends download acknowledgements; otherwise, false.

getAuthenticationParms function

Returns parameters provided to a custom user authentication script.

Syntax

abstract String **SyncParms.getAuthenticationParms()**

See also

- setAuthenticationParms function

Returns

The list of authentication parms or null if no parameters are specified.

getLivenessTimeout function

Returns the liveness timeout length, in seconds.

Syntax

abstract int **SyncParms.getLivenessTimeout()**

See also

- setLivenessTimeout function

Returns

The timeout.

getNewPassword function

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

Syntax

abstract String **SyncParms.getNewPassword()**

See also

- setUserName function
- setNewPassword function

Returns

The new password set after the next synchronization.

getPassword function

Returns the MobiLink password for the user specified with setUserName.

Syntax

abstract String **SyncParms.getPassword()**

See also

- setPassword function

Returns

The password for the MobiLink user.

getPublications function

Returns the publications to be synchronized.

Syntax

abstract String **SyncParms.getPublications()**

See also

- setPublications function

Returns

The set of publications to synchronize.

getSendColumnNames function

Returns true if column names are sent to the MobiLink server.

Syntax

abstract boolean **SyncParms.getSendColumnNames()**

See also

- setSendColumnNames function

Returns

true if column names are sent.

getStreamParms function

Returns the parameters used to configure the synchronization stream.

Syntax

abstract StreamHTTPParms **SyncParms.getStreamParms()**

Remarks

The synchronization stream type is specified when the SyncParms class object is created.

See also

- createSyncParms function
- StreamHTTPParms interface
- StreamHTTPSParms interface

Returns

A StreamHTTPParms interface or StreamHTTPSParms interface object specifying the parameters for HTTP or HTTPS synchronization streams. The object is returned by reference.

getSyncObserver function

Returns the current SyncObserver interface.

Syntax

abstract SyncObserver **SyncParms.getSyncObserver()**

Returns

The SyncObserver interface, or null if no observer exists.

getSyncResult function

Returns the SyncResult class object that contains the status of the synchronization.

Syntax

abstract SyncResult **SyncParms.getSyncResult()**

See also

- SyncResult class

Returns

The SyncResult class object.

getTableOrder function

Returns the order in which tables should be uploaded to the consolidated database.

Syntax

abstract String **SyncParms.getTableOrder()**

See also

- setTableOrder function

Returns

A comma separated list of table names; otherwise, null if no table order is specified. See class description for more on comma separated lists.

getUserName function

Returns the MobiLink user name that uniquely identifies the client to the MobiLink server.

Syntax

abstract String **SyncParms.getUserName()**

See also

- setUserName function

Returns

The MobiLink user name.

getVersion function

Returns the synchronization script to use.

Syntax

abstract String **SyncParms.getVersion()**

See also

- setVersion function

Returns

The script version.

isDownloadOnly function

Determines if the synchronization is download-only.

Syntax

abstract boolean **SyncParms.isDownloadOnly()**

See also

- setDownloadOnly function

Returns

true, if uploads are disabled; otherwise, false.

isPingOnly function

Determines if the synchronization pings the MobiLink server instead of performing a synchronization.

Syntax

abstract boolean **SyncParms.isPingOnly()**

See also

- setPingOnly function

Returns

true if the client only pings the server; otherwise, false.

isUploadOnly function

Determines if the synchronization is upload-only.

Syntax

abstract boolean **SyncParms.isUploadOnly()**

See also

- setUploadOnly function

Returns

True if downloads are disabled; otherwise, false.

setAcknowledgeDownload function

Indicates if the remote should send a download acknowledgement.

Syntax

abstract void **SyncParms.setAcknowledgeDownload(**
boolean *ack*
)

Parameters

- **ack** Set true to have the client acknowledge a download; otherwise, set false.

Remarks

The default is false.

See also

- getAcknowledgeDownload function

setAuthenticationParms function

Specifies parameters for a custom user authentication script (MobiLink authenticate_parameters connection event).

Syntax

abstract void **SyncParms.setAuthenticationParms(**
String *v*
) throws **ULjException**

Parameters

- **v** A comma separated list of authentication parameters, or the null reference. See class description for more on comma separated lists.

Remarks

Only the first 255 strings are used and each string should be no longer than 128 characters (longer strings are truncated when sent to MobiLink).

See also

- getAuthenticationParms function

setDownloadOnly function

Sets the synchronization as download-only.

Syntax

```
abstract void SyncParms.setDownloadOnly(
  boolean v
)
```

Parameters

- **v** Set true to disable uploads, or set false to enable uploads.

Remarks

The default is false. Specifying true changes setPingOnly and setUploadOnly to false.

See also

- isDownloadOnly function
- setPingOnly function
- setUploadOnly function

setLivenessTimeout function

Sets the liveness timeout length, in seconds. The default value is 100 seconds.

Syntax

```
abstract void SyncParms.setLivenessTimeout(
  int l
) throws ULjException
```

Parameters

- **l** The new liveness timeout value.

Remarks

The liveness timeout is the length of time the server allows a remote to be idle. If the remote does not communicate with the server for l seconds, the server assumes that the remote has lost the connection, and terminates the sync. The remote automatically sends periodic messages to the server to keep the connection alive.

If a negative value is set, an exception is thrown. The value may be changed by the MobiLink server without notice. This change occurs if the value is set too low or too high.

See also

- getLivenessTimeout function

setNewPassword function

Sets a new MobiLink password for the user specified with setUserName.

Syntax

```
abstract void SyncParms.setNewPassword(
  String v
)
```

Parameters

- **v** A new password for MobiLink user.

Remarks

The new password takes effect after the next synchronization.

The default is null, suggesting that the password does not get replaced.

See also

- getNewPassword function
- setPassword function
- setUserName function

setPassword function

Sets the MobiLink password for the user specified with setUserName.

Syntax

```
abstract void SyncParms.setPassword(
  String v
) throws ULjException
```

Parameters

- **v** A password for the MobiLink user.

Remarks

This user name and password is separate from any database user ID and password. This method is used to authenticate the application against the MobiLink server.

The default is an empty string, suggesting no password.

See also

- getPassword function

- setNewPassword function
- setUserName function

setPingOnly function

Sets the synchronization to ping the MobiLink server instead of performing a synchronization.

Syntax

```
abstract void SyncParms.setPingOnly(
  boolean v
)
```

Parameters

- **v** Set true to only ping the server, or set false to perform a synchronization.

Remarks

The default is false. Specifying true changes setDownloadOnly and setUploadOnly to false.

See also

- isPingOnly function
- setDownloadOnly function
- setUploadOnly function

setPublications function

Sets the publications to be synchronized.

Syntax

```
abstract void SyncParms.setPublications(
  String pubs
) throws ULjException
```

Parameters

- **pubs** A comma separated list of publication names. See class description for more on comma separated lists.

Remarks

The default is Connection.SYNC_ALL, which is used to denote synchronize all tables. To synchronize all publications, use Connection.SYNC_ALL_PUBS.

See also

- getPublications function
- SYNC_ALL variable
- SYNC_ALL_PUBS variable
- createPublication function

setSendColumnNames function

Sets whether or not to send column names to the MobiLink server during a sync. The default value is false.

Syntax

```
abstract void SyncParms.setSendColumnNames(
  boolean c
)
```

Parameters

- **c** true if column names should be sent

Remarks

Column names are used by the server only when using the direct row API.

See also

- getSendColumnNames function

setSyncObserver function

Sets a SyncObserver interface object to monitor the progress of synchronization.

Syntax

```
abstract void SyncParms.setSyncObserver(
  SyncObserver so
)
```

Parameters

- **so** A SyncObserser.

Remarks

The default is null, suggesting no observer.

See also

- SyncObserver interface

setTableOrder function

Sets the order in which tables should be uploaded to the consolidated database.

Syntax

```
abstract void SyncParms.setTableOrder(
  String v
) throws ULjException
```


Parameters

- **v** A comma separated list of table names in the order they should be synchronized, or null, indicating no table order. See class description for more on comma separated lists.

Remarks

The primary table should be listed first, along with all tables containing foreign key relationships in the consolidated database.

All tables selected for synchronization by the Publications get synchronized whether they are specified in the TableOrder parameter or not. Unspecified tables are synchronized by order of the foreign key relations in the client database. They are synchronized after the specified tables.

The default is a null reference, which does not override the default ordering of tables.

See also

- getTableOrder function
- setPublications function

setUploadOnly function

Sets the synchronization as upload-only.

Syntax

```
abstract void SyncParms.setUploadOnly(
  boolean v
)
```

Parameters

- **v** Set true to disable downloads, or set false to enable downloads.

Remarks

The default is false. Specifying true changes setDownloadOnly and setPingOnly to false.

See also

- isUploadOnly function
- setDownloadOnly function
- setPingOnly function

setUserName function

Sets the MobiLink user name that uniquely identifies the client to the MobiLink server.

Syntax

```
abstract void SyncParms.setUserName(
  String v
) throws ULjException
```

Parameters

- **v** The MobiLink user name.

Remarks

This value is used to determine the download content, to record the synchronization state, and to recover from interruptions during synchronization.

This user name and password is separate from any database user ID and password. This method is used to authenticate the application against the MobiLink server.

This parameter is initialized when the SyncParms class object is created.

See also

- getUserName function
- setPassword function
- setNewPassword function
- createSyncParms function

setVersion function

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 class object is created.

See also

- getVersion function
- createSyncParms function

SyncResult class

Reports status-related information on a specified database synchronization.

Syntax

public **SyncResult**

Members

All members of SyncResult, including all inherited members.

getAuthStatus function

Returns the authorization status code of the last synchronization attempt.

Syntax

abstract int **SyncResult.getAuthStatus()**

Returns

An AuthStatusCode value.

getAuthValue function

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 function

Returns the name of the table currently being synced.

Syntax

abstract String **SyncResult.getCurrentTableName()**

Returns

The table name.

getIgnoredRows function

Determines if any uploaded rows were ignored during the last synchronization.

Syntax

abstract boolean **SyncResult.getIgnoredRows()**

Returns

true if any uploaded rows were ignored during the last synchronization; otherwise, false if no rows were ignored.

getReceivedByteCount function

Returns the number of bytes received during data synchronization.

Syntax

abstract long **SyncResult.getReceivedByteCount()**

Returns

The number of bytes.

getReceivedRowCount function

Returns the number of rows received.

Syntax

abstract int **SyncResult.getReceivedRowCount()**

Returns

The number of rows.

getSentByteCount function

Returns the number of bytes sent during data synchronization.

Syntax

abstract long **SyncResult.getSentByteCount()**

Returns

The number of bytes sent.

getSentRowCount function

Returns the number of rows sent.

Syntax

abstract int **SyncResult.getSentRowCount()**

Returns

The number of rows.

getStreamErrorCode function

Returns the error code reported by the stream.

Syntax

abstract int **SyncResult.getStreamErrorCode()**

Remarks

The error code is the HTTP response code.

Returns

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

getStreamErrorMessage function

Returns the error message reported by the stream.

Syntax

abstract String **SyncResult.getStreamErrorMessage()**

Remarks

The error code is the HTTP response message.

Returns

Null, if no message is available; otherwise, the response message.

getSyncedTableCount function

Returns the number of tables synced so far.

Syntax

abstract int **SyncResult.getSyncedTableCount()**

Returns

The number of tables.

getTotalTableCount function

Returns the number of tables to be synced.

Syntax

abstract int **SyncResult.getTotalTableCount()**

Returns

The number of tables.

isUploadOK function

Determines if the last upload synchronization was successful.

Syntax

abstract boolean **SyncResult.isUploadOK()**

Returns

True, if the last upload synchronization was successful; otherwise, false.

SyncResult.AuthStatusCode interface

Enumerates the authorization codes returned by the MobiLink server.

Syntax

public **SyncResult.AuthStatusCode**

See also

- getAuthStatus function

Members

All members of SyncResult.AuthStatusCode, including all inherited members.

EXPIRED variable

User ID or password has expired. Authorization failed.

Syntax

final int **SyncResult.AuthStatusCode.EXPIRED**

INVALID variable

Bad user ID or password. Authorization failed.

Syntax

final int **SyncResult.AuthStatusCode.INVALID**

IN_USE variable

User ID is already in use. Authorization failed.

Syntax

final int **SyncResult.AuthStatusCode.IN_USE**

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 **TableSchema**

Remarks

An object supporting this interface is returned by the createTable function.

All tables must have at least one column and a primary key.

The following example demonstrates the creation of the schema for a simple database. The T2 table is created with two columns, a primary key, and an index.

```
// Assumes a valid Connection object conn
TableSchema table_schema;
IndexSchema index_schema;

table_schema = conn.createTable("T2");
table_schema.createColumn("num", Domain.INTEGER);
table_schema.createColumn("quantity", Domain.INTEGER);

index_schema = table_schema.createPrimaryIndex("primary");
index_schema.addColumn("num", IndexSchema.ASCENDING);
index_schema = table_schema.createIndex("index1");
index_schema.addColumn("quantity", IndexSchema.ASCENDING);

conn.schemaCreateComplete();
```

Primary keys uniquely identify each row in a table. Columns included in primary keys cannot allow nulls. Primary keys are created using createPrimaryIndex function.

A unique key is a constraint to identify one or more columns that uniquely identify each row in the table. No two rows in the table can have the same values in all the named column(s). A table may have more than one unique constraint. Primary keys are unique keys. Unique keys are created using createUniqueKey function.

A unique index ensures that there are not two rows in the table with identical values in all the columns in the index. Each index key must be unique or contain a null in at least one column. Unique indexes are created using createUniqueIndex function.

An unrestricted index allows duplicate index entries and null columns. Plain indexes are created using createIndex function.

Members

All members of TableSchema, including all inherited members.

- “createColumn function” on page 238
- “createColumn function” on page 239
- “createColumn function” on page 239
- “createIndex function” on page 240
- “createPrimaryIndex function” on page 240

SYS_ARTICLES variable

Name of the system table containing information on publication articles.

Syntax

final String **TableSchema.SYS_ARTICLES**

SYS_COLUMNS variable

Name of the system table containing information on the table columns in the database.

Syntax

final String **TableSchema.SYS_COLUMNS**

SYS_FKEY_COLUMNS variable

Name of the system table containing information on foreign key columns.

Syntax

final String **TableSchema.SYS_FKEY_COLUMNS**

SYS_FOREIGN_KEYS variable

Name of the system table containing information on foreign keys in the database.

Syntax

final String **TableSchema.SYS_FOREIGN_KEYS**

SYS_INDEXES variable

Name of the system table containing information on the table indexes in the database.

Syntax

final String **TableSchema.SYS_INDEXES**

SYS_INDEX_COLUMNS variable

Name of the system table containing information on the index columns in the database.

Syntax

final String **TableSchema.SYS_INDEX_COLUMNS**

SYS_INTERNAL variable

Name of the system table containing information on database options and internal database data.

Syntax

final String **TableSchema.SYS_INTERNAL**

SYS_PRIMARY_INDEX variable

Name of the primary key index of system tables.

Syntax

final String **TableSchema.SYS_PRIMARY_INDEX**

SYS_PUBLICATIONS variable

Name of the system table containing information on database publications.

Syntax

final String **TableSchema.SYS_PUBLICATIONS**

SYS_TABLES variable

Name of the system table containing information on the tables in the database.

Syntax

final String **TableSchema.SYS_TABLES**

TABLE_IS_NOSYNC variable

Bit flag denoting a table is a no-sync table (table that is never synchronized).

Syntax

final short **TableSchema.TABLE_IS_NOSYNC**

Remarks

This value can be logically combined with other flags in the table_flags column of the SYS_TABLES table.

TABLE_IS_SYSTEM variable

Bit flag denoting a table is a system table.

Syntax

final short **TableSchema.TABLE_IS_SYSTEM**

Remarks

This value can be logically combined with other flags in the table_flags column of the SYS_TABLES table.

createColumn function

Creates a new column of a fixed size type.

Syntax

ColumnSchema **TableSchema.createColumn(**
  String *column_name*,
  short *column_type*
) throws **ULjException**

Parameters

- **column_name** The name of the new column. The specified name must be a valid SQL identifier.
- **column_type** One of the Domain type constants representing a fixed size column type.

See also

Domain interface

Returns

The ColumnSchema assigned to the created column with the specified name and type.

createColumn function

Creates a new column of varying size type.

Syntax

```
ColumnSchema TableSchema.createColumn(
  String column_name,
  short column_type,
  int column_size
) throws ULjException
```

Parameters

- **column_name** The name of the new column. The specified name must be a valid SQL identifier.
- **column_type** One of the Domain type constants representing a column type with varying size (BINARY, NUMERIC, VARCHAR).
- **column_size** The size of column.

Remarks

If the column type is of fixed size, the size is ignored.

See also

Domain interface

Returns

The ColumnSchema assigned to the created column with the specified name and type.

createColumn function

Creates a new column of varying size and precision type.

Syntax

```
ColumnSchema TableSchema.createColumn(
  String column_name,
  short column_type,
  int column_size,
  int column_scale
) throws ULjException
```

Parameters

- **column_name** The name of the new column. The specified name must be a valid SQL identifier.

- **column_type** One of the Domain type constants representing a column type with varying size and scale (NUMERIC).
- **column_size** The size of column.
- **column_scale** The scale of column.

Remarks

If the column type is fixed, the size or scale is ignored.

See also

Domain interface

Returns

The ColumnSchema assigned to the created column with the specified name and type.

createIndex function

Creates a new index.

Syntax

```
IndexSchema TableSchema.createIndex(
  String index_name
) throws ULjException
```

Parameters

- **index_name** The name of index. The specified name must be a valid SQL identifier.

Returns

The IndexSchema assigned to the created index with the specified name.

createPrimaryIndex function

Creates the primary index for a table.

Syntax

```
IndexSchema TableSchema.createPrimaryIndex(
  String index_name
) throws ULjException
```

Parameters

- **index_name** The name of index. The specified name must be a valid SQL identifier.

Remarks

Each table must have exactly one primary index. The columns in a primary index must be non-nullable.

Returns

The IndexSchema assigned to the created primary index with the specified name.

createUniqueIndex function

Creates a new unique index.

Syntax

```
IndexSchema TableSchema.createUniqueIndex(
  String index_name
) throws ULjException
```

Parameters

- **index_name** The name of index. The specified name must be a valid SQL identifier.

Remarks

Each index key must be unique or contain a null in at least one column.

Unlike the columns in unique key constraints, columns in a unique index are allowed to be null. A foreign key can reference either a primary key or a unique key but a foreign key can not reference a unique index.

Returns

The IndexSchema assigned to the created unique index with the specified name.

createUniqueKey function

Creates a new unique key.

Syntax

```
IndexSchema TableSchema.createUniqueKey(
  String index_name
) throws ULjException
```

Parameters

- **index_name** The name of the key. The specified name must be a valid SQL identifier.

Remarks

A unique key is a constraint to identify one or more columns that uniquely identify each row in the table. A table can have more than one unique constraint.

Returns

The IndexSchema assigned to the created unique key with the specified name.

setNoSync function

Specifies if the table ever synchronizes.

Syntax

TableSchema **TableSchema.setNoSync(**
  boolean *no_sync*
) throws **ULjException**

Parameters

- **no_sync** true, if you plan to synchronize any changes to the table; otherwise, false.

Remarks

When set true, row change information is not maintained. The default value is false.

Returns

This TableSchema with NoSync defined.

ULjException class

Supercedes the exceptions thrown by the UltraLiteJ database.

Syntax

public **ULjException**

Members

All members of ULjException, including all inherited members.

- “SQLE_VALUE_IS_NULL variable” on page 200
- “SQLE_VARIABLE_INVALID variable” on page 200
- “SQLE_WRONG_NUM_OF_INSERT_COLS variable” on page 200
- “SQLE_WRONG_PARAMETER_COUNT variable” on page 200

getCausingException function

Returns the ULjException causing this exception.

Syntax

abstract ULjException **ULjException.getCausingException()** throws **ULjException**

Remarks

Returns

null, if no causing exceptions exist; otherwise, the ULjException.

getErrorCode function

Returns the error code associated with the exception.

Syntax

abstract int **ULjException.getErrorCode()**

Remarks

Returns

The error code.

getSqlOffset function

Returns the error offset within the SQL string.

Syntax

abstract int **ULjException.getSqlOffset()**

Remarks

Returns

-1 when there is no SQL string associated with the error message; otherwise, the zero-base offset within that string where the error occurred.

Value interface

Describes the column value in a fetched row.

Syntax

public **Value**

Base classes

- "ValueReader interface" on page 249
- "ValueWriter interface" on page 253

Members

All members of Value, including all inherited members.

- "compareValue function" on page 247
- "duplicate function" on page 247
- "getBlobInputStream function" on page 249
- "getBlobOutputStream function" on page 253
- "getBoolean function" on page 249
- "getBytes function" on page 250
- "getClobReader function" on page 250
- "getClobWriter function" on page 253
- "getDate function" on page 250
- "getDecimalNumber function" on page 250
- "getDomain function" on page 247
- "getDomainSize function" on page 247
- "getDouble function" on page 251
- "getFloat function" on page 251
- "getInt function" on page 251
- "getLong function" on page 251
- "getSize function" on page 248
- "getString function" on page 252
- "getType function" on page 248
- "getValue function" on page 252
- "isNull function" on page 252
- "release function" on page 248
- "set function" on page 254
- "set function" on page 254
- "set function" on page 254
- "set function" on page 254
- "set function" on page 255
- "set function" on page 255
- "set function" on page 255
- "set function" on page 256
- "set function" on page 256
- "set function" on page 256
- "setNull function" on page 256

compareValue function

Compares two Values.

Syntax

int **Value.compareValue(**
Value *other*
) throws **ULjException**

Parameters

- **other** The Value being compared against.

Returns

0 if the Value interface is equivalent to other, a negative integer if it is less than other, and a positive integer if it is greater than other.

duplicate function

Duplicates the Value and returns it.

Syntax

Value **Value.duplicate()** throws **ULjException**

Returns

The duplicated Value.

getDomain function

Returns the Domain object of the Value.

Syntax

Domain **Value.getDomain()** throws **ULjException**

Returns

The Domain object.

getDomainSize function

Returns the Domain size of the Value.

Syntax

int **Value.getDomainSize()** throws **ULjException**

Returns

The Domain size.

getSize function

Returns the current size of the Value.

Syntax

int **Value.getSize()** throws **ULjException**

Returns

The size.

getType function

Returns the Domain type of the Value.

Syntax

int **Value.getType()** throws **ULjException**

Returns

The Domain type.

release function

Closes the Value to release the memory resources associated with it.

Syntax

void **Value.release()** throws **ULjException**

ValueReader interface

Reads Value objects and interprets them as a java variable types.

Syntax

public **ValueReader**

Derived classes

- "Value interface" on page 246

Members

All members of ValueReader, including all inherited members.

- "getBlobInputStream function" on page 249
- "getBoolean function" on page 249
- "getBytes function" on page 250
- "getClobReader function" on page 250
- "getDate function" on page 250
- "getDecimalNumber function" on page 250
- "getDouble function" on page 251
- "getFloat function" on page 251
- "getInt function" on page 251
- "getLong function" on page 251
- "getString function" on page 252
- "getValue function" on page 252
- "isNull function" on page 252

getBlobInputStream function

The blob InputStream.

Syntax

java.io.InputStream **ValueReader.getBlobInputStream()** throws **ULjException**

Remarks

Returns the blob InputStream for the Value.

getBoolean function

Returns the boolean interpretation of the Value.

Syntax

boolean **ValueReader.getBoolean()** throws **ULjException**

Returns

The boolean value.

getBytes function

Returns the byte array for the Value.

Syntax

byte[] **ValueReader.getBytes()** throws **ULjException**

Returns

The byte array.

getClobReader function

Returns the clob Reader for the Value.

Syntax

java.io.Reader **ValueReader.getClobReader()** throws **ULjException**

Returns

The clob Reader.

getDate function

Returns the date interpretation of the Value.

Syntax

java.util.Date **ValueReader.getDate()** throws **ULjException**

Returns

The date of the Value.

getDecimalNumber function

Returns the DecimalNumber interpretation of the Value.

Syntax

DecimalNumber **ValueReader.getDecimalNumber()** throws **ULjException**

Returns

The DecimalNumber value.

getDouble function

Returns the double interpretation of the Value.

Syntax

double **ValueReader.getDouble()** throws **ULjException**

Returns

The double value.

getFloat function

Returns the float interpretation of the Value.

Syntax

float **ValueReader.getFloat()** throws **ULjException**

Returns

The float value.

getInt function

Returns the integer interpretation of the Value.

Syntax

int **ValueReader.getInt()** throws **ULjException**

Returns

The integer value.

getLong function

Returns the long interpretation of the Value.

Syntax

long **ValueReader.getLong()** throws **ULjException**

Returns

The long value.

getString function

Returns the String interpretation of the Value.

Syntax

String **ValueReader.getString()** throws **ULjException**

Returns

The String value.

getValue function

Returns a Value object.

Syntax

Value **ValueReader.getValue()** throws **ULjException**

isNull function

Tests if the value is null.

Syntax

boolean **ValueReader.isNull()**

Returns

True, if the value contains null; otherwise, false.

ValueWriter interface

Stores the values of java variable types into Value objects.

Syntax

public **ValueWriter**

Derived classes

- "Value interface" on page 246

Members

All members of ValueWriter, including all inherited members.

getBlobOutputStream function

Returns the blob OutputStream for the Value.

Syntax

java.io.OutputStream **ValueWriter.getBlobOutputStream()** throws **ULjException**

Returns

The blob OuputStream.

getClobWriter function

Returns the clob Writer for the Value.

Syntax

java.io.Writer **ValueWriter.getClobWriter()** throws **ULjException**

Returns

The clob Writer.

set function

Sets the boolean value of the Value.

Syntax

void **ValueWriter.set(**
boolean *value*
) throws **ULjException**

Parameters

- **value** The value to set.

set function

Sets a DecimalNumber for the Value.

Syntax

void **ValueWriter.set(**
DecimalNumber *value*
) throws **ULjException**

Parameters

- **value** The value to set.

set function

Sets a Date for the Value.

Syntax

void **ValueWriter.set(**
java.util.Date *value*
) throws **ULjException**

Parameters

- **value** The value to set.

set function

Sets an integer for the Value.

Syntax

void **ValueWriter.set(**
int *value*
) throws **ULjException**

Parameters

- **value** The value to set.

set function

Sets a long integer for the Value.

Syntax

void **ValueWriter.set(**
long *value*
) throws **ULjException**

Parameters

- **value** The value to set.

set function

Sets a float for the Value.

Syntax

void **ValueWriter.set(**
float *value*
) throws **ULjException**

Parameters

- **value** The value to set.

set function

Sets a double for the Value.

Syntax

void **ValueWriter.set(**
double *value*
) throws **ULjException**

Parameters

- **value** The value to set.

set function

Sets a byte array for the Value.

Syntax

void **ValueWriter.set(**
byte[] *value*
) throws **ULjException**

Parameters

- **value** The value to set.

set function

Sets a String for the Value.

Syntax

void **ValueWriter.set(**
String *value*
) throws **ULjException**

Parameters

- **value** The value to set.

set function

Sets a Value object for the Value.

Syntax

void **ValueWriter.set(**
Value *value*
) throws **ULjException**

Parameters

- **value** The value to set.

setNull function

Sets the Value to null.

Syntax

void **ValueWriter.setNull()** throws **ULjException**

CHAPTER 5

UltraLiteJ system tables

Contents

systable system table

Each row in the systable system table describes a table in the database.

Column name	Column type	Description
table_id	INTEGER	A unique identifier for the table.
table_name	VAR-CHAR(128)	The name of the table.
table_flags	UNSIGNED SHORT	A bitwise combination of one of the following flags: • TABLE_IS_SYSTEM • TABLE_IS_NO_SYNC
table_data	INTEGER	Internal use only.
table_autoinc	BIG	Internal use only.

Constraints

PRIMARY INDEX (table_id)

syscolumn system table

Each row in the syscolumn system table describes a column.

Column name	Column type	Description
table_id	INTEGER	The identifier of the table to which the column belongs.
column_id	INTEGER	A unique identifier for the column.
column_name	VARCHAR(128)	The name of the column. See "Domain interface" on page 153.
column_flags	TINY	A bitwise combination of the following flags describing the attributes: • **0x01** Column is in the primary key. • **0x02** Column is nullable.
column_domain	INTEGER	The column domain, which is an enumerated value indicating the domain of the column.
column_length	INTEGER	The column length. For VARCHAR and BINARY type columns, which are defined in the Domain interface, this is the maximum length in bytes. For NUMERIC type columns, the precision is stored in the first byte, and the scale is stored in the second byte.
column_default	TINY	The default value for this column, which is specified by one of the COLUMN_DEFAULT values in the ColumnSchema interface. For example, COLUMN_DEFAULT_AUTOINC denotes an auto-incrementing default value.

Constraints

PRIMARY KEY (table_id, column_id)

sysindex system table

Each row in the sysindex system table describes an index in the database.

Column name	Column type	Description
table_id	INTEGER	A unique identifier for the table to which the index applies.
index_id	INTEGER	A unique identifier for an index.
index_name	VARCHAR(128)	The name of the index.
index_flags	TINY	A bitwise combination of the following flags denoting the type of index and its persistence: • **0x01** Unique key. • **0x02** Unique index. • **0x04** Index is persistent. • **0x08** Primary key.
index_data	INTEGER	Internal use only.

Constraints

PRIMARY KEY (table_id, index_id)

sysindexcolumn system table

Each row in the sysindexcolumn system table describes a column of an index listed in sysindex.

Column name	Column type	Description
table_id	INTEGER	A unique identifier for the table to which the index applies.
index_id	INTEGER	A unique identifier for the index that this index-column belongs to.
order	INTEGER	The order of the column in the index.
column_id	INTEGER	A unique identifier for the column being indexed.
index_column_flag	TINY	An indication of where the column in the index is kept, either in ascending (1) or descending (0) order.

Constraints

PRIMARY KEY (table_id, index_id, order)

sysinternal system table

Each row in the sysinternal system table is used to store system options and other internal data.

Column name	Column type	Description
name	VARCHAR(128)	The name of the option.
value	VARCHAR(128)	The option value.

Constraints

PRIMARY KEY (name)

syspublications system table

Each row in the syspublications system table describes a publication.

Column name	Column type	Description
publication_id	INTEGER	A unique identifier for the publication.
publication_name	VARCHAR(128)	The name of the publication.
download_timestamp	TIMESTAMP	The time of the last download.
last_sync_sent	INTEGER	An integer that tracks an upload sent to MobiLink.
last_sync_confirmed	INTEGER	An integer that tracks an upload confirmed as being received by MobiLink.

Constraints

PRIMARY KEY (publication_id)

sysarticles system table

Each row in the sysarticles system table describes a table that belongs to a publication.

Column name	Column type	Description
publication _id	INTEGER	An identifier for the publication that this article belongs to.
table_id	INTEGER	The identifier of the table that belongs to the publication.

Constraints

PRIMARY KEY (publication_id, table_id)

sysforeignkey system table

Each row in the sysforeignkey system table describes a foreign key that belongs to a table.

Column name	Column type	Description
table_id	INTEGER	The identifier of the table to which the foreign key belongs.
foreign_table_id	INTEGER	The identifier of the table to which this foreign-key-column refers to.
foreign_key_id	INTEGER	The identifier of the foreign key.
name	VARCHAR(128)	The name of the foreign key.
index_name	VARCHAR(128)	The name of the index the foreign key refers to.

Constraints

PRIMARY KEY (table_id, foreign_key_id)

sysfkcol system table

Each row in the sysfkcol system table describes a foreign key column.

Column name	Column type	Description
table_id	INTEGER	A unique identifier for the table to which the foreign key applies.
foreign_key_id	INTEGER	A unique identifier for the foreign key that this column belongs to.
item_no	SHORT	The order of the column in the foreign key.
column_id	INTEGER	A unique identifier of the table column that refers to the foreign column.
foreign_column_id	INTEGER	A unique identifier of the table column that is being referred to.

Constraints

PRIMARY KEY (table_id, foreign_key_id, item_no)

CHAPTER 6

UltraLiteJ utilities

Contents

Utilities are supplied with UltraLiteJ to perform maintenance and administration tasks on UltraLiteJ databases.

Utilities for J2SE

These utilities are for J2SE implementations of UltraLiteJ only—they are not designed for use with the BlackBerry smartphone environment.

UltraLiteJ database information utility (ULjInfo)

The ULjInfo utility displays information about an existing UltraLiteJ database.

Syntax

ULjInfo -c *filename* **-p** *password* [*options*]

Option	Description
-c *filename*	Required. The filename of the UltraLiteJ database to examine.
-p *password*	Required. The password to connect to the UltraLiteJ database.
-q	Run in quiet mode—do not display messages.
-v	Display verbose messages.
-?	Display command line usage information.

Example output (non verbose)

Following is an example of the output from the ULjInfo program (without the -v option):

```
C:\ULj\bin>ULjInfo.cmd -c ..\Samples\Demo1.ulj -p sql
SQL Anywhere UltraLite J Database Information Utility
Database name: ..\Samples\Demo1.ulj
Disk file: '..\Samples\Demo1.ulj'
Database ID: 0
Page size: 1024
0 rows for next upload
Date format: YYYY-MM-DD
Date order: YMD
Nearest century: 50
Numeric precision: 30
Numeric scale: 6
Time format: HH:NN:SS.SSS
Timestamp format: YYYY-MM-DD HH:NN:SS.SSS
Timestamp increment: 1
Number of tables: 1
Number of columns: 2
Number of publications: 0
Number of tables that will always be uploaded: 0
Number of tables that are never synchronized: 0
Number of primary keys: 1
Number of foreign keys: 0
Number of indexes: 0
Last download occurred on Thu Jul 05 11:31:05 EDT 2007
Upload OK: true
```


UltraLiteJ database load (ULjLoad)

The ULjLoad utility provides the capability to load an UltraLiteJ database from an XML source file. The XML file is often produced by the ULjUnload utility and is customizable.

Syntax

ULjLoad -c *filename* **-p** *password* [*options*] *inputfile*

Option	Description
-a	Add information from XML file to an existing database. If this option is not specified, a new database is created.
-c *filename*	Required. Name of database file.
-d	Load data only; ignore schema information.
-f *directory*	Directory to retrieve data for columns larger than max blob size specified with -b option during ULjUnload.
-i	Insert rows for upload synchronization.
-n	Load schema information only; ignore row data.
-p *password*	Required. Password to connect to database.
-q	Run in quiet mode—do not display messages.
-v	Display verbose messages.
-y	Overwrite output file if it exists (and -a option is not specified).
-?	Display command line usage information.
inputfile	Input file containing xml statements.

Example of usage summary

When the ULjLoad utility command line includes the **-?** option, the following usage information appears:

```
SQL Anywhere UltraLite J Database Load Utility
Usage: uljload [options] <XML file>
        Create and load data into a new UltraLite J database from <XML file>.

Options:
        -a      Add to existing database.
        -c      <file> Database file.
        -d      Data only -- ignore schema.
        -f      <directory>
                Directory to store columns larger than <max blob size>.
        -i      Insert rows for upload synchronization.
        -n      Schema only -- ignore data.
        -p      Password to connect to database.
```



```
-q      Quiet: do not print messages.
-v      Verbose messages.
-y      Overwrite file if it already exists.
```

UltraLiteJ database unload (ULjUnload)

The ULjUnload utility provides the capability to unload an UltraLiteJ database — either the data, the schema, or both — to an XML file.

Syntax

ULjUnload -c *filename* **-p** *password* [*options*] *outputfile*

Options	Description
-b *max-blob-size*	Maximum size (in bytes) of blob/char data output to XML.
-c *filename*	Required. Name of database file to unload.
-d	Unload only data; do not output schema information.
-e *table, ...*	Exclude data for tables named in list.
-f *directory*	Directory to store data for columns larger than max blob size specified with -b option.
-n	Unload schema information only; do not output data.
-p *password*	Required. Password to connect to database.
-q	Run in quiet mode—do not display messages.
-t *table, ...*	Only output data for tables named in list.
-v	Display verbose messages.
-y	Overwrite output file if it already exists.
-?	Display option usage/help information.
outputfile	Output file name (this file contains xml statements that describe the database contents).

Example XML file contents

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<ul:ulschema xmlns:ul="urn:ultralite">
 <collation name="1252LATIN1" case_sensitive="no"/>
 <options>
  <option name="dateformat" value="YYYY-MM-DD"/>
  <option name="dateorder" value="YMD"/>
  <option name="nearestcentury" value="50"/>
```



```
  <option name="precision" value="30"/>
  <option name="scale" value="6"/>
  <option name="timeformat" value="HH:NN:SS.SSS"/>
  <option name="timestampformat" value="YYYY-MM-DD HH:NN:SS.SSS"/>
  <option name="timestampincrement" value="1"/>
 </options>
 <tables>
  <table name="ULCustomer" sync="changes">
   <columns>
    <column name="cust_id" type="integer" null="no"/>
    <column name="cust_name" type="char(30)" null="yes"/>
   </columns>
   <primarykey>
    <primarycolumn name="cust_id" direction="asc"/>
   </primarykey>
   <indexes/>
  </table>
 </tables>
 <uldata>
  <table name="ULCustomer">
   <row cust_id="2000" cust_name="Apple St. Builders"/>
   <row cust_id="2001" cust_name="Art's Renovations"/>
   <row cust_id="2002" cust_name="Awnings R Us"/>
   <row cust_id="2003" cust_name="Al's Interior Design"/>
   <row cust_id="2004" cust_name="Alpha Hardware"/>
   <row cust_id="2005" cust_name="Ace Properties"/>
   <row cust_id="2006" cust_name="A1 Contracting"/>
   <row cust_id="2007" cust_name="Archibald Inc."/>
   <row cust_id="2008" cust_name="Acme Construction"/>
   <row cust_id="2009" cust_name="ABCXYZ Inc."/>
   <row cust_id="2010" cust_name="Buy It Co."/>
   <row cust_id="2011" cust_name="Bill's Cages"/>
   <row cust_id="2012" cust_name="Build-It Co."/>
   <row cust_id="2013" cust_name="Bass Interiors"/>
   <row cust_id="2014" cust_name="Burger Franchise"/>
   <row cust_id="2015" cust_name="Big City Builders"/>
   <row cust_id="2016" cust_name="Bob's Renovations"/>
   <row cust_id="2017" cust_name="Basements R Us"/>
   <row cust_id="2018" cust_name="BB Interior Design"/>
   <row cust_id="2019" cust_name="Bond Hardware"/>
   <row cust_id="2020" cust_name="Cat Properties"/>
   <row cust_id="2021" cust_name="C &amp;amp; C Contracting"/>
   <row cust_id="2022" cust_name="Classy Inc."/>
   <row cust_id="2023" cust_name="Cooper Construction"/>
   <row cust_id="2024" cust_name="City Schools"/>
   <row cust_id="2025" cust_name="Can Do It Co."/>
   <row cust_id="2026" cust_name="City Corrections"/>
   <row cust_id="2027" cust_name="City Sports Arenas"/>
   <row cust_id="2028" cust_name="Cantaloupe Interiors"/>
   <row cust_id="2029" cust_name="Chicken Franchise"/>
  </table>
 </uldata>
</ul:ulschema>
```

Utilities for J2ME (for BlackBerry smartphones)

These utilities are designed for use with the BlackBerry smartphone environment only.

UltraLiteJ database transfer (ULjDbT)

The ULjDbT utility provides the capability to transfer an UltraLiteJ database from a BlackBerry smartphone to an external device, such as a desktop, laptop, or server. The utility consists of two applications that must run simultaneously—the Java Swing Server application and the BlackBerry smartphone client application.

The Java Swing Server application

The Java Swing Server application receives UltraLiteJ databases using a USB or HTTP connection method. When you start the server application, it waits for a BlackBerry smartphone to transfer the database through the specified connection with the client application. The connection is closed either manually through the application interface, when the application times out, or when the transfer is complete.

To set up the Java Swing Server application, JDK 1.6.0 or later, and the Java Swing GUI libraries must be installed.

To receive a database using the Java Swing Server application

1. Run *ULjDbTServ.cmd* from the *Bin32* directory of your SQL Anywhere installation.

   The Java Swing Server application loads.
2. On the **Connect** tab, select the desired **Connection Method**.
3. Under **Connection Properties**, specify the following values:
   - **Port** The TCP port number used to establish a connection to the BlackBerry smartphone. This field only applies to HTTP connections.
   - **BlackBerry Password** The password used to access the connected BlackBerry smartphone when it is locked. Leave this field blank if there is no password. This field only applies to USB connections.
   - **Timeout** The number of idle minutes before the server application times out and closes the connection.
   - **Output** The new filename and location of the transferred database.
4. Click **Start** to wait for a BlackBerry smartphone connection.

   The server application waits until it either times out or establishes a connection. Click the **Logs** tab to view detailed information on the server status and the transfer progress.

The BlackBerry Smartphone Client application

The BlackBerry Smartphone Client application sends UltraLiteJ databases through a USB cable or a TCP port specified by the Java Swing Server application. For more information on preparing an external device to receive UltraLiteJ databases, see "The Java Swing Server application" on page 272.

The client application is a signed file located in the *UltraLite\UltraLiteJ\J2meRim11* directory of your SQL Anywhere installation.

To transfer a database using the BlackBerry Smartphone Client application

1. Load *UltraLiteJ.cod* and *ULjDatabaseTransfer.cod* from the *UltraLite\UltraLiteJ\J2meRim11* directory of your SQL Anywhere installation.

   The client application icon appears in your list of applications.

2. Start the application and press the trackwheel.

   The **UltraLiteJ Database Transfer** menu appears.

3. Select the desired connection method from the dropdown menu.

4. Complete the following fields:

   - **Database Name** The name of the database to transfer to the external device.
   - **Database Password** The database password used to allow data transfer.
   - **Server** The IP address or hostname of the server receiving the transfer. All entries must include the **http://** suffix. This field only applies to HTTP connections.
   - **Port** The TCP port number used when connecting to the external device. This number must be identical to the port number specified in the server application. This field only applies to HTTP connections.

5. Activate the **Connect** command.

   Note
   To ensure a successful database transfer, make sure that the connection to the server application is active on the external device.

   The BlackBerry smartphone starts transferring the database to the external device. Progress information can be traced on server application.

Part III. Glossary

CHAPTER 7

Glossary

Adaptive Server Anywhere (ASA)

The relational database server component of SQL Anywhere Studio, intended for use in mobile and embedded environments or as a server for small and medium-sized businesses. In version 10.0.0, Adaptive Server Anywhere was renamed SQL Anywhere Server, and SQL Anywhere Studio was renamed SQL Anywhere.

See also: "SQL Anywhere" on page 300.

agent ID

See also: "client message store ID" on page 279.

article

In MobiLink or SQL Remote, an article is a database object that represents a whole table, or a subset of the columns and rows in a table. Articles are grouped together in a publication.

See also:

- "replication" on page 298
- "publication" on page 295

atomic transaction

A transaction that is guaranteed to complete successfully or not at all. If an error prevents part of an atomic transaction from completing, the transaction is rolled back to prevent the database from being left in an inconsistent state.

base table

Permanent tables for data. Tables are sometimes called **base tables** to distinguish them from temporary tables and views.

See also:

- "temporary table" on page 302
- "view" on page 304

bit array

A bit array is a type of array data structure that is used for efficient storage of a sequence of bits. A bit array is similar to a character string, except that the individual pieces are 0s (zeros) and 1s (ones) instead of characters. Bit arrays are typically used to hold a string of Boolean values.

business rule

A guideline based on real-world requirements. Business rules are typically implemented through check constraints, user-defined data types, and the appropriate use of transactions.

See also:

- "constraint" on page 280
- "user-defined data type" on page 304

carrier

A MobiLink object, stored in MobiLink system tables or a Notifier properties file, that contains information about a public carrier for use by server-initiated synchronization.

See also: "server-initiated synchronization" on page 300.

character set

A character set is a set of symbols, including letters, digits, spaces, and other symbols. An example of a character set is ISO-8859-1, also known as Latin1.

See also:

- "code page" on page 279
- "encoding" on page 284
- "collation" on page 279

check constraint

A restriction that enforces specified conditions on a column or set of columns.

See also:

- "constraint" on page 280
- "foreign key constraint" on page 286
- "primary key constraint" on page 295
- "unique constraint" on page 303

checkpoint

The point at which all changes to the database are saved to the database file. At other times, committed changes are saved only to the transaction log.

checksum

The calculated number of bits of a database page that is recorded with the database page itself. The checksum allows the database management system to validate the integrity of the page by ensuring that the numbers match as the page is being written to disk. If the counts match, it's assumed that page was successfully written.

client message store

In QAnywhere, a SQL Anywhere database on the remote device that stores messages.

client message store ID

In QAnywhere, a MobiLink remote ID that uniquely identifies a client message store.

client/server

A software architecture where one application (the client) obtains information from and sends information to another application (the server). The two applications often reside on different computers connected by a network.

code page

A code page is an encoding that maps characters of a character set to numeric representations, typically an integer between 0 and 255. An example of a code page is Windows code page 1252. For the purposes of this documentation, code page and encoding are interchangeable terms.

See also:

- “character set” on page 278
- “encoding” on page 284
- “collation” on page 279

collation

A combination of a character set and a sort order that defines the properties of text in the database. For SQL Anywhere databases, the default collation is determined by the operating system and language on which the server is running; for example, the default collation on English Windows systems is 1252LATIN1. A collation, also called a collating sequence, is used for comparing and sorting strings.

See also:

- “character set” on page 278
- “code page” on page 279
- “encoding” on page 284

command file

A text file containing SQL statements. Command files can be built manually, or they can be built automatically by database utilities. The dbunload utility, for example, creates a command file consisting of the SQL statements necessary to recreate a given database.

communication stream

In MobiLink, the network protocol used for communication between the MobiLink client and the MobiLink server.

concurrency

The simultaneous execution of two or more independent, and possibly competing, processes. SQL Anywhere automatically uses locking to isolate transactions and ensure that each concurrent application sees a consistent set of data.

See also:

- "transaction" on page 302
- "isolation level" on page 288

conflict resolution

In MobiLink, conflict resolution is logic that specifies what to do when two users modify the same row on different remote databases.

connection ID

A unique number that identifies a given connection between a client application and the database. You can determine the current connection ID using the following SQL statement:

```
SELECT CONNECTION_PROPERTY( 'Number' );
```

connection-initiated synchronization

A form of MobiLink server-initiated synchronization in which synchronization is initiated when there are changes to connectivity.

See also: "server-initiated synchronization" on page 300.

connection profile

A set of parameters that are required to connect to a database, such as user name, password, and server name, that is stored and used as a convenience.

consolidated database

In distributed database environments, a database that stores the master copy of the data. In case of conflict or discrepancy, the consolidated database is considered to have the primary copy of the data.

See also:

- "synchronization" on page 302
- "replication" on page 298

constraint

A restriction on the values contained in a particular database object, such as a table or column. For example, a column may have a uniqueness constraint, which requires that all values in the column be different. A table may have a foreign key constraint, which specifies how the information in the table relates to data in some other table.

See also:

- "check constraint" on page 278
- "foreign key constraint" on page 286
- "primary key constraint" on page 295
- "unique constraint" on page 303

contention

The act of competing for resources. For example, in database terms, two or more users trying to edit the same row of a database contend for the rights to edit that row.

correlation name

The name of a table or view that is used in the FROM clause of a query—either its original name, or an alternate name, that is defined in the FROM clause.

creator ID

In UltraLite Palm OS applications, an ID that is assigned when the application is created.

cursor

A named linkage to a result set, used to access and update rows from a programming interface. In SQL Anywhere, cursors support forward and backward movement through the query results. Cursors consist of two parts: the cursor result set, typically defined by a SELECT statement; and the cursor position.

See also:

- "cursor result set" on page 281
- "cursor position" on page 281

cursor position

A pointer to one row within the cursor result set.

See also:

- "cursor" on page 281
- "cursor result set" on page 281

cursor result set

The set of rows resulting from a query that is associated with a cursor.

See also:

- "cursor" on page 281
- "cursor position" on page 281

data cube

A multi-dimensional result set with each dimension reflecting a different way to group and sort the same results. Data cubes provide complex information about data that would otherwise require self-join queries and correlated subqueries. Data cubes are a part of OLAP functionality.

data definition language (DDL)

The subset of SQL statements for defining the structure of data in the database. DDL statements create, modify, and remove database objects, such as tables and users.

data manipulation language (DML)

The subset of SQL statements for manipulating data in the database. DML statements retrieve, insert, update, and delete data in the database.

data type

The format of data, such as CHAR or NUMERIC. In the ANSI SQL standard, data types can also include a restriction on size, character set, and collation.

See also: "domain" on page 284.

database

A collection of tables that are related by primary and foreign keys. The tables hold the information in the database. The tables and keys together define the structure of the database. A database management system accesses this information.

See also:

- "foreign key" on page 285
- "primary key" on page 295
- "database management system (DBMS)" on page 282
- "relational database management system (RDBMS)" on page 297

database administrator (DBA)

The user with the permissions required to maintain the database. The DBA is generally responsible for all changes to a database schema, and for managing users and groups. The role of database administrator is automatically built into databases as user ID DBA with password sql.

database connection

A communication channel between a client application and the database. A valid user ID and password are required to establish a connection. The privileges granted to the user ID determine the actions that can be carried out during the connection.

database file

A database is held in one or more database files. There is an initial file, and subsequent files are called dbspaces. Each table, including its indexes, must be contained within a single database file.

See also: "dbspace" on page 283.

database management system (DBMS)

A collection of programs that allow you to create and use databases.

See also: "relational database management system (RDBMS)" on page 297.

database name

The name given to a database when it is loaded by a server. The default database name is the root of the initial database file.

See also: "database file" on page 282.

database object

A component of a database that contains or receives information. Tables, indexes, views, procedures, and triggers are database objects.

database owner (dbo)

A special user that owns the system objects not owned by SYS.

See also:

- "database administrator (DBA)" on page 282
- "SYS" on page 302

database server

A computer program that regulates all access to information in a database. SQL Anywhere provides two types of servers: network servers and personal servers.

DBA authority

The level of permission that enables a user to do administrative activity in the database. The DBA user has DBA authority by default.

See also: "database administrator (DBA)" on page 282.

dbspace

An additional database file that creates more space for data. A database can be held in up to 13 separate files (an initial file and 12 dbspaces). Each table, together with its indexes, must be contained in a single database file. The SQL command CREATE DBSPACE adds a new file to the database.

See also: "database file" on page 282.

deadlock

A state where a set of transactions arrives at a place where none can proceed.

device tracking

Functionality in MobiLink server-initiated synchronization that allows you to address messages using the MobiLink user name that identifies a remote device.

See also: "server-initiated synchronization" on page 300.

direct row handling

In MobiLink, a way to synchronize table data to sources other than the MobiLink-supported consolidated databases. You can implement both uploads and downloads with direct row handling.

See also:

- "consolidated database" on page 280
- "SQL-based synchronization" on page 300

domain

Aliases for built-in data types, including precision and scale values where applicable, and optionally including DEFAULT values and CHECK conditions. Some domains, such as the monetary data types, are pre-defined in SQL Anywhere. Also called user-defined data type.

See also: "data type" on page 282.

download

The stage in synchronization where data is transferred from the consolidated database to a remote database.

dynamic SQL

SQL that is generated programmatically by your program before it is executed. UltraLite dynamic SQL is a variant designed for small-footprint devices.

EBF

Express Bug Fix. An express bug fix is a subset of the software with one or more bug fixes. The bug fixes are listed in the release notes for the update. Bug fix updates may only be applied to installed software with the same version number. Some testing has been performed on the software, but the software has not undergone full testing. You should not distribute these files with your application unless you have verified the suitability of the software yourself.

embedded SQL

A programming interface for C programs. SQL Anywhere embedded SQL is an implementation of the ANSI and IBM standard.

encoding

Also known as character encoding, an encoding is a method by which each character in a character set is mapped onto one or more bytes of information, typically represented as a hexadecimal number. An example of an encoding is UTF-8.

See also:

- "character set" on page 278
- "code page" on page 279
- "collation" on page 279

event model

In MobiLink, the sequence of events that make up a synchronization, such as begin_synchronization and download_cursor. Events are invoked if a script is created for them.

external login

An alternate login name and password used when communicating with a remote server. By default, SQL Anywhere uses the names and passwords of its clients whenever it connects to a remote server on behalf of those clients. However, this default can be overridden by creating external logins. External logins are alternate login names and passwords used when communicating with a remote server.

extraction

In SQL Remote replication, the act of unloading the appropriate structure and data from the consolidated database. This information is used to initialize the remote database.

See also: "replication" on page 298.

failover

Switching to a redundant or standby server, system, or network on failure or unplanned termination of the active server, system, or network. Failover happens automatically.

FILE

In SQL Remote replication, a message system that uses shared files for exchanging replication messages. This is useful for testing and for installations without an explicit message-transport system.

See also:"replication" on page 298

file-based download

In MobiLink, a way to synchronize data in which downloads are distributed as files, allowing offline distribution of synchronization changes.

file-definition database

In MobiLink, a SQL Anywhere database that is used for creating download files.

See also: "file-based download" on page 285.

foreign key

One or more columns in a table that duplicate the primary key values in another table. Foreign keys establish relationships between tables.

See also:

- "primary key" on page 295
- "foreign table" on page 286

foreign key constraint

A restriction on a column or set of columns that specifies how the data in the table relates to the data in some other table. Imposing a foreign key constraint on a set of columns makes those columns the foreign key.

See also:

- "constraint" on page 280
- "check constraint" on page 278
- "primary key constraint" on page 295
- "unique constraint" on page 303

foreign table

The table containing the foreign key.

See also: "foreign key" on page 285.

full backup

A backup of the entire database, and optionally, the transaction log. A full backup contains all the information in the database and thus provides protection in the event of a system or media failure.

See also: "incremental backup" on page 287.

gateway

A MobiLink object, stored in MobiLink system tables or a Notifier properties file, that contains information about how to send messages for server-initiated synchronization.

See also: "server-initiated synchronization" on page 300.

generated join condition

A restriction on join results that is automatically generated. There are two types: key and natural. Key joins are generated when you specify KEY JOIN or when you specify the keyword JOIN but do not use the keywords CROSS, NATURAL, or ON. For a key join, the generated join condition is based on foreign key relationships between tables. Natural joins are generated when you specify NATURAL JOIN; the generated join condition is based on common column names in the two tables.

See also:

- "join" on page 289
- "join condition" on page 289

generation number

In MobiLink, a mechanism for forcing remote databases to upload data before applying any more download files.

See also: "file-based download" on page 285.

global temporary table

A type of temporary table for which data definitions are visible to all users until explicitly dropped. Global temporary tables let each user open their own identical instance of a table. By default, rows are deleted on commit, and rows are always deleted when the connection is ended.

See also:

- "temporary table" on page 302
- "local temporary table" on page 290

grant option

The level of permission that allows a user to grant permissions to other users.

hash

A hash is an index optimization that transforms index entries into keys. An index hash aims to avoid the expensive operation of finding, loading, and then unpacking the rows to determine the indexed value, by including enough of the actual row data with its row ID.

histogram

The most important component of column statistics, histograms are a representation of data distribution. SQL Anywhere maintains histograms to provide the optimizer with statistical information about the distribution of values in columns.

iAnywhere JDBC driver

The iAnywhere JDBC driver provides a JDBC driver that has some performance benefits and feature benefits compared to the pure Java jConnect JDBC driver, but which is not a pure-Java solution. The iAnywhere JDBC driver is recommended in most cases.

See also:

- "JDBC" on page 289
- "jConnect" on page 289

identifier

A string of characters used to reference a database object, such as a table or column. An identifier may contain any character from A through Z, a through z, 0 through 9, underscore (_), at sign (@), number sign (#), or dollar sign ($).

incremental backup

A backup of the transaction log only, typically used between full backups.

See also: "transaction log" on page 303.

index

A sorted set of keys and pointers associated with one or more columns in a base table. An index on one or more columns of a table can improve performance.

InfoMaker

A reporting and data maintenance tool that lets you create sophisticated forms, reports, graphs, cross-tabs, and tables, as well as applications that use these reports as building blocks.

inner join

A join in which rows appear in the result set only if both tables satisfy the join condition. Inner joins are the default.

See also:

- "join" on page 289
- "outer join" on page 293

integrated login

A login feature that allows the same single user ID and password to be used for operating system logins, network logins, and database connections.

integrity

Adherence to rules that ensure that data is correct and accurate, and that the relational structure of the database is intact.

See also: "referential integrity" on page 297.

Interactive SQL

A SQL Anywhere application that allows you to query and alter data in your database, and modify the structure of your database. Interactive SQL provides a pane for you to enter SQL statements, as well as panes that return information about how the query was processed and the result set.

isolation level

The degree to which operations in one transaction are visible to operations in other concurrent transactions. There are four isolation levels, numbered 0 through 3. Level 3 provides the highest level of isolation. Level 0 is the default setting. SQL Anywhere also supports three snapshot isolation levels: snapshot, statement-snapshot, and readonly-statement-snapshot.

See also: "snapshot isolation" on page 300.

JAR file

Java archive file. A compressed file format consisting of a collection of one or more packages used for Java applications. It includes all the resources necessary to install and run a Java program in a single compressed file.

Java class

The main structural unit of code in Java. It is a collection of procedures and variables grouped together because they all relate to a specific, identifiable category.

jConnect

A Java implementation of the JavaSoft JDBC standard. It provides Java developers with native database access in multi-tier and heterogeneous environments. However, the iAnywhere JDBC driver is the preferred JDBC driver for most cases.

See also:

- "JDBC" on page 289
- "iAnywhere JDBC driver" on page 287

JDBC

Java Database Connectivity. A SQL-language programming interface that allows Java applications to access relational data. The preferred JDBC driver is the iAnywhere JDBC driver.

See also:

- "jConnect" on page 289
- "iAnywhere JDBC driver" on page 287

join

A basic operation in a relational system that links the rows in two or more tables by comparing the values in specified columns.

join condition

A restriction that affects join results. You specify a join condition by inserting an ON clause or WHERE clause immediately after the join. In the case of natural and key joins, SQL Anywhere generates a join condition.

See also:

- "join" on page 289
- "generated join condition" on page 286

join type

SQL Anywhere provides four types of joins: cross join, key join, natural join, and joins using an ON clause.

See also: "join" on page 289.

Listener

A program, dblsn, that is used by MobiLink server-initiated synchronization. Listeners are installed on remote devices and configured to initiate actions on the device when they receive information from a Notifier.

See also: “server-initiated synchronization” on page 300.

local temporary table

A type of temporary table that exists only for the duration of a compound statement or until the end of the connection. Local temporary tables are useful when you need to load a set of data only once. By default, rows are deleted on commit.

See also:

- “temporary table” on page 302
- “global temporary table” on page 287

lock

A concurrency control mechanism that protects the integrity of data during the simultaneous execution of multiple transactions. SQL Anywhere automatically applies locks to prevent two connections from changing the same data at the same time, and to prevent other connections from reading data that is in the process of being changed.

You control locking by setting the isolation level.

See also:

- “isolation level” on page 288
- “concurrency” on page 279
- “integrity” on page 288

log file

A log of transactions maintained by SQL Anywhere. The log file is used to ensure that the database is recoverable in the event of a system or media failure, to improve database performance, and to allow data replication using SQL Remote.

See also:

- “transaction log” on page 303
- “transaction log mirror” on page 303
- “full backup” on page 286

logical index

A reference (pointer) to a physical index. There is no indexing structure stored on disk for a logical index.

LTM

Log Transfer Manager (LTM) also called Replication Agent. Used with Replication Server, the LTM is the program that reads a database transaction log and sends committed changes to Sybase Replication Server.

See: “Replication Server” on page 298.

maintenance release

A maintenance release is a complete set of software that upgrades installed software from an older version with the same major version number (version number format is *major.minor.patch.build*). Bug fixes and other changes are listed in the release notes for the upgrade.

materialized view

A materialized view is a view that has been computed and stored on disk. Materialized views have characteristics of both views (they are defined using a query specification), and of tables (they allow most table operations to be performed on them).

See also:

- "base table" on page 277
- "view" on page 304

message log

A log where messages from an application such as a database server or MobiLink server can be stored. This information can also appear in a messages window or be logged to a file. The message log includes informational messages, errors, warnings, and messages from the MESSAGE statement.

message store

In QAnywhere, databases on the client and server device that store messages.

See also:

- "client message store" on page 278
- "server message store" on page 300

message system

In SQL Remote replication, a protocol for exchanging messages between the consolidated database and a remote database. SQL Anywhere includes support for the following message systems: FILE, FTP, and SMTP.

See also:

- "replication" on page 298
- "FILE" on page 285

message type

In SQL Remote replication, a database object that specifies how remote users communicate with the publisher of a consolidated database. A consolidated database may have several message types defined for it; this allows different remote users to communicate with it using different message systems.

See also:

- "replication" on page 298
- "consolidated database" on page 280

metadata

Data about data. Metadata describes the nature and content of other data.

See also: "schema" on page 299.

mirror log

See also: "transaction log mirror" on page 303.

MobiLink

A session-based synchronization technology designed to synchronize UltraLite and SQL Anywhere remote databases with a consolidated database.

See also:

- "consolidated database" on page 280
- "synchronization" on page 302
- "UltraLite" on page 303

MobiLink client

There are two kinds of MobiLink clients. For SQL Anywhere remote databases, the MobiLink client is the dbmlsync command line utility. For UltraLite remote databases, the MobiLink client is built in to the UltraLite runtime library.

MobiLink Monitor

A graphical tool for monitoring MobiLink synchronizations.

MobiLink server

The computer program that runs MobiLink synchronization, mlsrv11.

MobiLink system table

System tables that are required by MobiLink synchronization. They are installed by MobiLink setup scripts into the MobiLink consolidated database.

MobiLink user

A MobiLink user is used to connect to the MobiLink server. You create the MobiLink user on the remote database and register it in the consolidated database. MobiLink user names are entirely independent of database user names.

network protocol

The type of communication, such as TCP/IP or HTTP.

network server

A database server that accepts connections from computers sharing a common network.

See also: "personal server" on page 294.

normalization

The refinement of a database structure to eliminate redundancy and improve organization according to rules based on relational database theory.

Notifier

A program that is used by MobiLink server-initiated synchronization. Notifiers run on the same computer as the MobiLink server. They poll the consolidated database for push requests, and then send notifications to Listeners.

See also:

- "server-initiated synchronization" on page 300
- "Listener" on page 289

object tree

In Sybase Central, the hierarchy of database objects. The top level of the object tree shows all products that your version of Sybase Central supports. Each product expands to reveal its own sub-tree of objects.

See also: "Sybase Central" on page 302.

ODBC

Open Database Connectivity. A standard Windows interface to database management systems. ODBC is one of several interfaces supported by SQL Anywhere.

ODBC Administrator

A Microsoft program included with Windows operating systems for setting up ODBC data sources.

ODBC data source

A specification of the data a user wants to access via ODBC, and the information needed to get to that data.

outer join

A join that preserves all the rows in a table. SQL Anywhere supports left, right, and full outer joins. A left outer join preserves the rows in the table to the left of the join operator, and returns a null when a row in the right table does not satisfy the join condition. A full outer join preserves all the rows from both tables.

See also:

- "join" on page 289
- "inner join" on page 288

package

In Java, a collection of related classes.

parse tree

An algebraic representation of a query.

PDB

A Palm database file.

performance statistic

A value reflecting the performance of the database system. The CURRREAD statistic, for example, represents the number of file reads issued by the engine that have not yet completed.

personal server

A database server that runs on the same computer as the client application. A personal database server is typically used by a single user on a single computer, but it can support several concurrent connections from that user.

physical index

The actual indexing structure of an index, as it is stored on disk.

plug-in module

In Sybase Central, a way to access and administer a product. Plug-ins are usually installed and registered automatically with Sybase Central when you install the respective product. Typically, a plug-in appears as a top-level container, in the Sybase Central main window, using the name of the product itself; for example, SQL Anywhere.

See also: "Sybase Central" on page 302.

policy

In QAnywhere, the way you specify when message transmission should occur.

polling

In MobiLink server-initiated synchronization, the way the Notifier detects push requests on the consolidated database.

See also: "server-initiated synchronization" on page 300.

PowerDesigner

A database modeling application. PowerDesigner provides a structured approach to designing a database or data warehouse. SQL Anywhere includes the Physical Data Model component of PowerDesigner.

PowerJ

A Sybase product for developing Java applications.

predicate

A conditional expression that is optionally combined with the logical operators AND and OR to make up the set of conditions in a WHERE or HAVING clause. In SQL, a predicate that evaluates to UNKNOWN is interpreted as FALSE.

primary key

A column or list of columns whose values uniquely identify every row in the table.

See also: “foreign key” on page 285.

primary key constraint

A uniqueness constraint on the primary key columns. A table can have only one primary key constraint.

See also:

- “constraint” on page 280
- “check constraint” on page 278
- “foreign key constraint” on page 286
- “unique constraint” on page 303
- “integrity” on page 288

primary table

The table containing the primary key in a foreign key relationship.

proxy table

A local table containing metadata used to access a table on a remote database server as if it were a local table.

See also: “metadata” on page 292.

publication

In MobiLink or SQL Remote, a database object that identifies data that is to be synchronized. In MobiLink, publications exist only on the clients. A publication consists of articles. SQL Remote users can receive a publication by subscribing to it. MobiLink users can synchronize a publication by creating a synchronization subscription to it.

See also:

- “replication” on page 298
- “article” on page 277
- “publication update” on page 296

publication update

In SQL Remote replication, a list of changes made to one or more publications in one database. A publication update is sent periodically as part of a replication message to the remote database(s).

See also:

- "replication" on page 298
- "publication" on page 295

publisher

In SQL Remote replication, the single user in a database who can exchange replication messages with other replicating databases.

See also: "replication" on page 298.

push notification

In QAnywhere, a special message delivered from the server to a QAnywhere client that prompts the client to initiate a message transmission.

See also: "QAnywhere" on page 296.

push request

In MobiLink server-initiated synchronization, a row in a SQL result set or table on the consolidated database that contains a notification and information about how to send the notification.

See also: "server-initiated synchronization" on page 300.

QAnywhere

Application-to-application messaging, including mobile device to mobile device and mobile device to and from the enterprise, that permits communication between custom programs running on mobile or wireless devices and a centrally located server application.

QAnywhere agent

In QAnywhere, a process running on the client device that monitors the client message store and determines when message transmission should occur.

query

A SQL statement or group of SQL statements that access and/or manipulate data in a database.

See also: "SQL" on page 300.

Redirector

A web server plug-in that routes requests and responses between a client and the MobiLink server. This plug-in also implements load-balancing and failover mechanisms.

reference database

In MobiLink, a SQL Anywhere database used in the development of UltraLite clients. You can use a single SQL Anywhere database as both reference and consolidated database during development. Databases made with other products cannot be used as reference databases.

referencing object

An object, such as a view, whose definition directly references another object in the database, such as a table.

See also: "foreign key" on page 285.

referenced object

An object, such as a table, that is directly referenced in the definition of another object, such as a view.

See also: "primary key" on page 295.

referential integrity

Adherence to rules governing data consistency, specifically the relationships between the primary and foreign key values in different tables. To have referential integrity, the values in each foreign key must correspond to the primary key values of a row in the referenced table.

See also:

- "primary key" on page 295
- "foreign key" on page 285

regular expression

A regular expression is a sequence of characters, wildcards, and operators that defines a pattern to search for within a string.

relational database management system (RDBMS)

A type of database management system that stores data in the form of related tables.

See also: "database management system (DBMS)" on page 282.

remote database

In MobiLink or SQL Remote, a database that exchanges data with a consolidated database. Remote databases may share all or some of the data in the consolidated database.

See also:

- "synchronization" on page 302
- "consolidated database" on page 280

remote DBA authority

In SQL Remote, a level of permission required by the Message Agent. In MobiLink, a level of permission required by the SQL Anywhere synchronization client (dbmlsync). When the Message Agent or synchronization client connects as a user who has this authority, it has full DBA access. The user ID has no additional permissions when not connected through the Message Agent or synchronization client.

See also: “DBA authority” on page 283.

remote ID

A unique identifier in SQL Anywhere and UltraLite databases that is used by MobiLink. The remote ID is initially set to NULL and is set to a GUID during a database's first synchronization.

replication

The sharing of data among physically distinct databases. Sybase has three replication technologies: MobiLink, SQL Remote, and Replication Server.

Replication Agent

See: “LTM” on page 290.

replication frequency

In SQL Remote replication, a setting for each remote user that determines how often the publisher's message agent should send replication messages to that remote user.

See also: “replication” on page 298.

replication message

In SQL Remote or Replication Server, a communication sent between a publishing database and a subscribing database. Messages contain data, passthrough statements, and information required by the replication system.

See also:

- “replication” on page 298
- “publication update” on page 296

Replication Server

A Sybase connection-based replication technology that works with SQL Anywhere and Adaptive Server Enterprise. It is intended for near-real time replication between a small number of databases.

See also: “LTM” on page 290.

role

In conceptual database modeling, a verb or phrase that describes a relationship from one point of view. You can describe each relationship with two roles. Examples of roles are "contains" and "is a member of."

role name

The name of a foreign key. This is called a role name because it names the relationship between the foreign table and primary table. By default, the role name is the table name, unless another foreign key is already using that name, in which case the default role name is the table name followed by a three-digit unique number. You can also create the role name yourself.

See also: "foreign key" on page 285.

rollback log

A record of the changes made during each uncommitted transaction. In the event of a ROLLBACK request or a system failure, uncommitted transactions are reversed out of the database, returning the database to its former state. Each transaction has a separate rollback log, which is deleted when the transaction is complete.

See also: "transaction" on page 302.

row-level trigger

A trigger that executes once for each row that is changed.

See also:

- "trigger" on page 303
- "statement-level trigger" on page 301

schema

The structure of a database, including tables, columns, and indexes, and the relationships between them.

script

In MobiLink, code written to handle MobiLink events. Scripts programmatically control data exchange to meet business needs.

See also: "event model" on page 285.

script-based upload

In MobiLink, a way to customize the upload process as an alternative to using the log file.

script version

In MobiLink, a set of synchronization scripts that are applied together to create a synchronization.

secured feature

A feature specified by the -sf option when a database server is started, so it is not available for any database running on that database server.

server-initiated synchronization

A way to initiate MobiLink synchronization programmatically from the consolidated database.

server management request

A QAnywhere message that is formatted as XML and sent to the QAnywhere system queue as a way to administer the sever message store or monitor QAnywhere applications.

server message store

In QAnywhere, a relational database on the server that temporarily stores messages until they are transmitted to a client message store or JMS system. Messages are exchanged between clients via the server message store.

service

In Windows operating systems, a way of running applications when the user ID running the application is not logged on.

session-based synchronization

A type of synchronization where synchronization results in consistent data representation across both the consolidated and remote databases. MobiLink is session-based.

snapshot isolation

A type of isolation level that returns a committed version of the data for transactions that issue read requests. SQL Anywhere provides three snapshot isolation levels: snapshot, statement-snapshot, and readonly-statement-snapshot. When using snapshot isolation, read operations do not block write operations.

See also: "isolation level" on page 288.

SQL

The language used to communicate with relational databases. ANSI has defined standards for SQL, the latest of which is SQL-2003. SQL stands, unofficially, for Structured Query Language.

SQL Anywhere

The relational database server component of SQL Anywhere that is intended for use in mobile and embedded environments or as a server for small and medium-sized businesses. SQL Anywhere is also the name of the package that contains the SQL Anywhere RDBMS, the UltraLite RDBMS, MobiLink synchronization software, and other components.

SQL-based synchronization

In MobiLink, a way to synchronize table data to MobiLink-supported consolidated databases using MobiLink events. For SQL-based synchronization, you can use SQL directly or you can return SQL using the MobiLink server APIs for Java and .NET.

SQL Remote

A message-based data replication technology for two-way replication between consolidated and remote databases. The consolidated and remote databases must be SQL Anywhere.

SQL statement

A string containing SQL keywords designed for passing instructions to a DBMS.

See also:

- "schema" on page 299
- "SQL" on page 300
- "database management system (DBMS)" on page 282

statement-level trigger

A trigger that executes after the entire triggering statement is completed.

See also:

- "trigger" on page 303
- "row-level trigger" on page 299

stored procedure

A program comprised of a sequence of SQL instructions, stored in the database and used to perform a particular task.

string literal

A string literal is a sequence of characters enclosed in single quotes.

subquery

A SELECT statement that is nested inside another SELECT, INSERT, UPDATE, or DELETE statement, or another subquery.

There are two types of subquery: correlated and nested.

subscription

In MobiLink synchronization, a link in a client database between a publication and a MobiLink user, allowing the data described by the publication to be synchronized.

In SQL Remote replication, a link between a publication and a remote user, allowing the user to exchange updates on that publication with the consolidated database.

See also:

- "publication" on page 295
- "MobiLink user" on page 292

Sybase Central

A database management tool that provides SQL Anywhere database settings, properties, and utilities in a graphical user interface. Sybase Central can also be used for managing other Sybase products, including MobiLink.

synchronization

The process of replicating data between databases using MobiLink technology.

In SQL Remote, synchronization is used exclusively to denote the process of initializing a remote database with an initial set of data.

See also:

- "MobiLink" on page 292
- "SQL Remote" on page 301

SYS

A special user that owns most of the system objects. You cannot log in as SYS.

system object

Database objects owned by SYS or dbo.

system table

A table, owned by SYS or dbo, that holds metadata. System tables, also known as data dictionary tables, are created and maintained by the database server.

system view

A type of view, included in every database, that presents the information held in the system tables in an easily understood format.

temporary table

A table that is created for the temporary storage of data. There are two types: global and local.

See also:

- "local temporary table" on page 290
- "global temporary table" on page 287

transaction

A sequence of SQL statements that comprise a logical unit of work. A transaction is processed in its entirety or not at all. SQL Anywhere supports transaction processing, with locking features built in to allow concurrent transactions to access the database without corrupting the data. Transactions end either with a COMMIT statement, which makes the changes to the data permanent, or a ROLLBACK statement, which undoes all the changes made during the transaction.

transaction log

A file storing all changes made to a database, in the order in which they are made. It improves performance and allows data recovery in the event the database file is damaged.

transaction log mirror

An optional identical copy of the transaction log file, maintained simultaneously. Every time a database change is written to the transaction log file, it is also written to the transaction log mirror file.

A mirror file should be kept on a separate device from the transaction log, so that if either device fails, the other copy of the log keeps the data safe for recovery.

See also: "transaction log" on page 303.

transactional integrity

In MobiLink, the guaranteed maintenance of transactions across the synchronization system. Either a complete transaction is synchronized, or no part of the transaction is synchronized.

transmission rule

In QAnywhere, logic that determines when message transmission is to occur, which messages to transmit, and when messages should be deleted.

trigger

A special form of stored procedure that is executed automatically when a user runs a query that modifies the data.

See also:

- "row-level trigger" on page 299
- "statement-level trigger" on page 301
- "integrity" on page 288

UltraLite

A database optimized for small, mobile, and embedded devices. Intended platforms include cell phones, pagers, and personal organizers.

UltraLite runtime

An in-process relational database management system that includes a built-in MobiLink synchronization client. The UltraLite runtime is included in the libraries used by each of the UltraLite programming interfaces, as well as in the UltraLite engine.

unique constraint

A restriction on a column or set of columns requiring that all non-null values are different. A table can have multiple unique constraints.

See also:

- "foreign key constraint" on page 286
- "primary key constraint" on page 295
- "constraint" on page 280

unload

Unloading a database exports the structure and/or data of the database to text files (SQL command files for the structure, and ASCII comma-separated files for the data). You unload a database with the Unload utility.

In addition, you can unload selected portions of your data using the UNLOAD statement.

upload

The stage in synchronization where data is transferred from a remote database to a consolidated database.

user-defined data type

See "domain" on page 284.

validate

To test for particular types of file corruption of a database, table, or index.

view

A SELECT statement that is stored in the database as an object. It allows users to see a subset of rows or columns from one or more tables. Each time a user uses a view of a particular table, or combination of tables, it is recomputed from the information stored in those tables. Views are useful for security purposes, and to tailor the appearance of database information to make data access straightforward.

window

The group of rows over which an analytic function is performed. A window may contain one, many, or all rows of data that has been partitioned according to the grouping specifications provided in the window definition. The window moves to include the number or range of rows needed to perform the calculations for the current row in the input. The main benefit of the window construct is that it allows additional opportunities for grouping and analysis of results, without having to perform additional queries.

Windows

The Microsoft Windows family of operating systems, such as Windows Vista, Windows XP, and Windows 200x.

Windows CE

See "Windows Mobile" on page 304.

Windows Mobile

A family of operating systems produced by Microsoft for mobile devices.

work table

An internal storage area for interim results during query optimization.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

