AVEVA™
SQLData Script Library User's Guide

Publication date: Tuesday, December 1, 2020

Contact Information

AVEVA Group plc
High Cross
Madingley Road
Cambridge
CB3 0HB. UK

https://sw.aveva.com/

For information on how to contact sales and customer training, see https://sw.aveva.com/contact.

For information on how to contact technical support, see https://sw.aveva.com/support.

Contents

CHAPTER 1

Using the SQLData Script Library

About Using the SQLData Script Library

The SQLData Script Library provides database integration using ArchestrA scripting. The SQLData Script Library provides the following benefits:

- Your resources are managed more efficiently because the connection manager reduces the number of open connections to the database provider. This activity is known as *connection pooling*.
- You can process scripts asynchronously, which reduces impact on the hosting engine during the following activities:
    - Opening a connection to a data source
    - Running SQL queries
    - Running SQL transactions

Note: It is recommended, to avoid returning excessive amount of data using script library commands, for example executing 15,000 SQL commands which each select 200 records from the Person.Contact table in the AdventureWorks sample database. This can lead to a crash of the host engine. To avoid this problem, consider using the TOP command to limit the number of records returned.

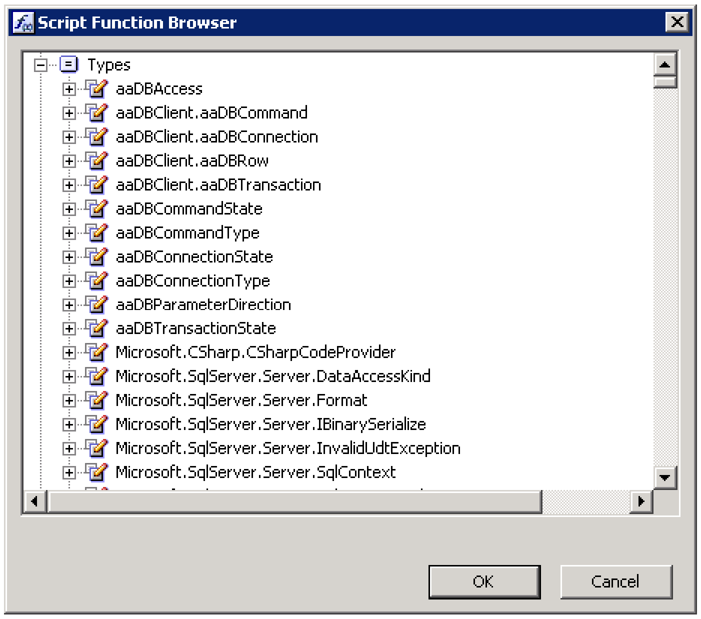
Importing and Accessing the SQLData Script Library

The SQLData Script Library is contained in the file named aaDBIntegration.aaSLIB. Copy the file to your ArchestrA development computer.

To access the SQLData Script Library

1. Open the ArchestrA IDE.
2. On the main menu, click **Galaxy/Import/Script Library**.

After you import the SQLData Script Library, you can access the library functions from the **Script Function Browser** in the **Script** tab of any object. The functions are visible in the **Types** section.



Types
- aaDBAccess
- aaDBClient.aaDBCommand
- aaDBClient.aaDBConnection
- aaDBClient.aaDBRow
- aaDBClient.aaDBTransaction
- aaDBCommandState
- aaDBCommandType
- aaDBConnectionState
- aaDBConnectionType
- aaDBParameterDirection
- aaDBTransactionState
- Microsoft.CSharp.CSharpCodeProvider
- Microsoft.SqlServer.Server.DataAccessKind
- Microsoft.SqlServer.Server.Format
- Microsoft.SqlServer.Server.IBinarySerialize
- Microsoft.SqlServer.Server.InvalidUdtException
- Microsoft.SqlServer.Server.SqlContext

For more information about working with function libraries, see the *AVEVA Application Server User's Guide*.

SQLData Script Library Interface

This section provides an overview of the SQLData Script Library interface.

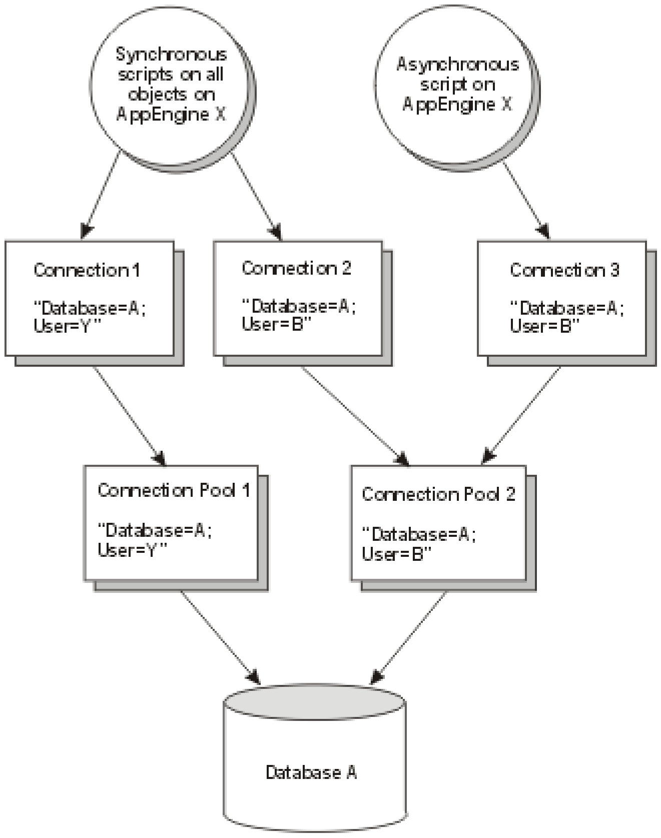
SQLData Script Library Architecture

You can integrate the SQLData Script Library into ArchestrA by using synchronous or asynchronous scripting. The SQLData Script Library contains the following public objects:

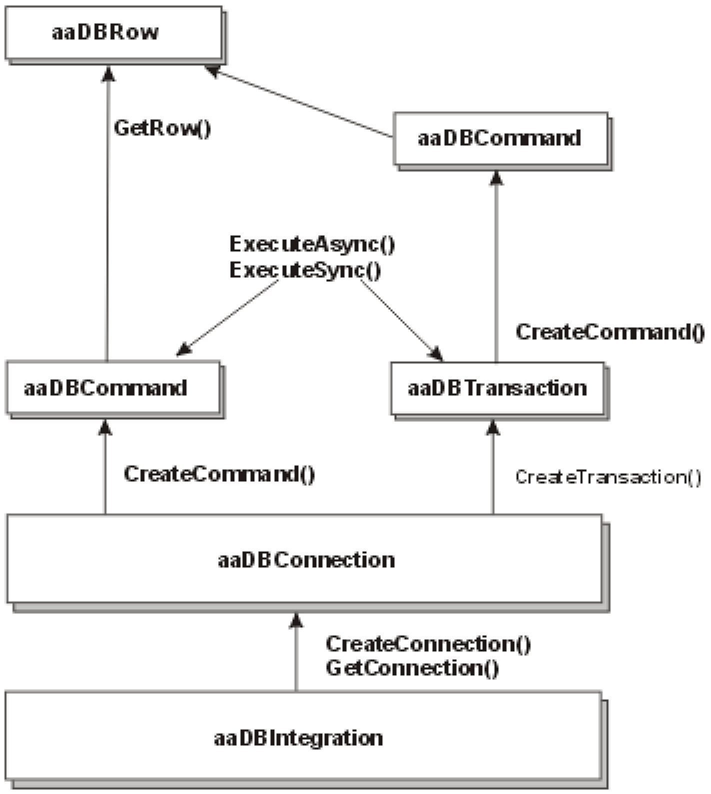
- aaDBAccess
- aaDBConnection
- aaDBTransaction
- aaDBCommand
- aaDBRow

You can find details about these objects as well as their methods, properties, and enumerations in the remaining chapters in this guide.

The following figure shows connection pooling in the SQLData Script Library. Although each script generates its own connection object in the script library, scripts with identical connection strings are allocated to the same connection pool. The result is fewer connections to the database.


Synchronous scripts on all objects on AppEngine X
Asynchronous script on AppEngine X
Connection 1 "Database=A; User=Y"
Connection 2 "Database=A; User=B"
Connection 3 "Database=A; User=B"
Connection Pool 1 "Database=A; User=Y"
Connection Pool 2 "Database=A; User=B"
Database A

The following figure shows a conceptual diagram of the relationships among the various components of the SQLData script library.



Note that **aaDBRow** is one mechanism provided by **aaDBCommand** to read and modify data returned from the SQL query. The other mechanisms are implemented as methods of **aaDBCommand**.

SQLData Script Library Work Flow

This section shows the work flow to create a connection object with a command and a transaction. This section also provides connection string examples.

Creating a Connection Object with a Command

SQLData scripts follow a typical flow when they are written without transactions:

1. Create a connection object.
2. Create one or more command objects using the **CreateCommand()** method of the connection object.
3. For each command object whose SQL statement contains parameters, initialize each parameter by using either the **Set<Type>ParameterByName()** method or the **Set<Type>ParameterByIndex()** method of the command object.
4. Run the command object either synchronously or asynchronously.

When command processing is complete, you can retrieve and modify the returned dataset. If you modify the dataset, you can save it back to the database either synchronously or asynchronously.

You can reuse command objects indefinitely. When finished, clean up the command objects by calling their **Dispose()** methods.

Creating a Connection Object with a Transaction

SQLData scripts follow a typical flow written with transactions:

1. Create a connection object.
2. Create a transaction object by using the **CreateTransaction()** method of the connection object.
3. Add one or more command objects to the transaction by using the **CreateCommand()** method of the transaction object.
4. For each command object whose SQL statement contains parameters, initialize each parameter by using either the **Set<Type>ParameterByName()** method or the **Set<Type>ParameterByIndex()** method of the command object.
5. Run the transaction object either synchronously or asynchronously.

If any of the commands fails or returns an error, the transaction and any commands that ran within the transaction are rolled back. The transaction is returned with the appropriate indicator

When transaction processing is complete, you can retrieve and modify the returned dataset for each command in the transaction that returns data. If you modify a dataset, you can save it back to the database either synchronously or asynchronously.

You can reuse transaction objects indefinitely. When finished, clean up the transaction object and command objects by calling their **Dispose()** methods.

Specifying Connection Strings

The SQLData Script Library uses a connection string to specify the parameters for a database connection. The connection string is specified by Microsoft in their ADO.NET implementation. You can find details about the connection string and its parameters in the following locations:

```
http://msdn2.microsoft.com/en-us/library/
ms254978(VS.85).aspx
```

```
http://msdn2.microsoft.com/en-us/library/
ms254499.aspx
```

The following examples show how to write connection strings for the most commonly used data providers.

Connecting to a SQL Server Data Source

The following script shows an example connection for SQL Server:

```
Connection=aaDBAccess.GetConnection(
    me.ConnectionString,
    aaDBConnectionType.Sql);
```

Using **aaDBConnectionType.Sql** is synonymous with the form of **GetConnection()** that takes only a connection string. Use the connection string example shown in *Connecting to a SQL Server Data Source* in place of me.ConnectionString.

Connecting to an Oracle Data Source

The following script shows an example connection for Oracle:

```
Connection=aaDBAccess.GetConnection("Provider=MSDAORA;
    Data Source=myOracleServer;User ID=<name>;
    Password=<password>.",aaDBConnectionType.Oracle);
```

Connecting to Microsoft Access through OLEDB

The following script shows an example connection for Microsoft Access through OLEDB:

```
Connection=aaDBAccess.GetConnection(
    "Provider=Microsoft.Jet.OLEDB 4.0;
    Data Source=C:\myAccessDb.mdb",
    aaDBConnectionType.OleDb);
```

Connecting to Microsoft Excel through OLEDB

The following script shows an example connection for Microsoft Excel through OLEDB:

```
Connection=aaDBAccess.GetConnection(
    "Provider=Microsoft.Jet.OLEDB 4.0;
    Data Source=C:\NameAndAddress.xls;
    Extended Properties=Excel 8.0;",
    aaDBConnectionType.OleDb);
```

Note: Microsoft Excel does not support transactions (commit or rollback).

Example Scripts

The following two examples illustrate the use of the SQLData Script Library to access a database.

Note: These scripts are provided only as a reference.

Overview of Sample Scripts

The sample scripts have the following purpose in a larger, ArchestrA context:

- The script provides an object that other requesting objects can use to read the specified column of a row whose part number is given and then return the column value to the requesting object.
- The requesting object provides the row name, part number, and start signal in UDAs.
- The sample object returns the requested row value and done signal in UDAs.

Detailed Description of Sample Scripts

This section is a detailed step-by-step description of what is happening in the sample scripts that follow. For more information about using application objects, see the AVEVA *Application Server User's Guide*.

1. Create a connection object and supply the connection string. The connection string in this example is a literal string, but you can use a connection string supplied by another ArchestrA object through a UDA.
2. If you are using transactions, create a transaction object on the connection object that you created.
3. Create a command object. This example supplies a literal string for the SQL statement, but it could be supplied by another ArchestrA object through a UDA. Note that the SQL statement is a query with a parameter, which allows the requesting object to specify the column to read:
    a. If you are not using transactions, create the command object on the connection object.
    b. If you are using transactions, create the command object on the transaction object.
4. Because the SQL statement of the command object contains a parameter, you must initialize that parameter object (or the transaction that contains it).
5. Run the transaction or command using either the **ExecuteAsync()** or **ExecuteSync()** methods.

6. If the command or transaction is processed synchronously, the SQLData Script Library does not return from the **ExecuteSync()** method until the command is complete. Because processing can take an extended period of time, you should use an asynchronous script (that is, a script that is not synchronized with the scan of the ArchestrA AppEngine).

   If the command or transaction is executed asynchronously, poll for completion as follows:

   - If your script runs synchronously with the scan of the ArchestrA AppEngine, you must signal polling to occur by a script once per scan cycle of the engine. This example script starts a second polling script, but you can write a single script with a state variable that starts processing in one state and polls in the other state.
   - If your script runs asynchronously with the ArchestrA AppEngine, you can poll in the following lines of the same script that starts processing using a while loop, which can take an extended period of time to finish.

7. When the command or transaction completes without an error, the result can be read from the command object using the parameter that was initialized before the command object was run.
8. Check for errors at each of the major script processing steps.

Asynchronous Command Script

The scripts in this object show how to use asynchronous SQL processing in the SQLData Script Library. This script is written to use a command object directly on the connection, without using a transaction object. For comparison, see *Synchronous Transaction Script on page 23* where the example uses the synchronous SQL processing for a command object on a transaction object.

This object is written with two scripts. One script starts asynchronous command processing (the **Query** script) and the other script polls for asynchronous command completion and results manipulation (the **Process** script). Both scripts run synchronously with the ArchestrA AppEngine scan. To use the scripts, set the number to search for in the **PartNumber** User Defined Attribute (UDA) and column whose value is to be read in the **ColumnToRead** UDA. Signal the script to run by setting the **ReadCommand** UDA to True. When the script finishes processing, the **ColumnReadDone** UDA becomes True, and the results are in the **ColumnValue** UDA.

Query Script Configuration

On the **Script** tab of the SQLData Object Editor, configure the **Query** script with the following attributes.

Attribute	**Value**
Execution Type	Execute
Expression	me.ReadCommand
Trigger Type	OnTrue
Runs Asynchronously	Selected

Process Script Configuration

On the **Script** tab of the SQLData Object Editor, configure the **Process** script with the following attributes.

Attribute	**Value**
Execution Type	Execute

Attribute	Value
Expression	me.ProcessCommand
Trigger Type	WhileTrue
Runs Asynchronously	Selected

Query Script Code

Use the following sample code for the **Query** script.

```
DIM Connection as aaDBClient.aaDBConnection;
DIM Command as aaDBClient.aaDBCommand;

'Create a connection object with the connection string.
LogMessage("Creating connection");
Connection = aaDBAccess.CreateConnection("Data Source=localhost;Initial
Catalog=AdventureWorks;Integrated Security=true");

'Create a command object, with a SQL statement.
LogMessage("Creating a command object");
Command = Connection.CreateCommand("Select * from Production.Product WHERE
ProductNumber = @ProductNumber", aaDBCommandType.SqlStatement, true);

'We used a parameter to specify the value for the ProductNumber field, so
initialize it.
Command.SetCharParameterByName("ProductNumber", me.PartNumber,
aaDBParameterDirection.Input, 50);
'Everything is ready, let's execute the command async.
LogMessage("Executing command async");
DIM ResultCode as integer;
ResultCode = Command.ExecuteAsync();
  if ResultCode <> 0 then
     'Failed to start async execution, report the reason.
      LogMessage("Got error " + ResultCode + " executing command async");
  else
     'Execution started, identify the command by ID, for use later.
     LogMessage("Command async execution started successfully");
     me.CommandID = Command.GetID();

     'Allow the Process script to run.
     me.ProcessCommand = true;
  endif;

'Reset for next time
me.ReadCommand = false;
```

Process Script Code

Use the following sample code for the **Process** script.

```
DIM Command as aaDBClient.aaDBCommand;
'Retrieve the command object using its ID.
Command = aaDBAccess.GetCommand(me.CommandID);
if Command <> null then
  'Poll for command complete
  if Command.ExecutionState <> aaDBCommandState.Queued then
```

```
    LogMessage("Command execution state is " + Command.ExecutionState);
if Command.ExecutionState == aaDBCommandState.Completed then
      DIM Rows as integer;
      Rows = Command.RowCount;
      LogMessage("Row count returned from command is " + Rows);
'We expect one row, use the first one, if we have any.
      if Rows > 0 then
        LogMessage("Getting column '" + me.ColumnToRead + "' from row 0");
        Command.SelectRow(0);
'Return the requested column value from this row, signal done.
        me.ColumnValue = Command.GetCurrentRowColumnByName(me.ColumnToRead);
        me.ColumnReadDone = true;
      endif;
    endif;
'When done, dispose the command.
    Command.Dispose();
'Reset for next time
    me.ProcessCommand = false;
  endif;
else
  LogMessage("Cannot find command " + me.CommandID);
  me.ProcessCommand = false;
endif;
```

Synchronous Transaction Script

The single script for this SQLData object shows how to use the synchronous SQL processing in the SQLData Script Library. This script is uses a transaction object with the command object on the transaction. For comparison, see *Asynchronous Command Script on page 19* where the example uses the command object directly on the connection, without a transaction.

This transaction object is written with a single script called **QueryandProcess**, which performs the SQL processing synchronously and then manipulates the results. This script runs without synchronizing with the ArchestrA AppEngine scan (asynchronous script). To use the script, set the part number to search for in the **PartNumber** UDA and the column whose value is to be read in the **ColumnToRead** UDA. Signal the script to run by setting the **ReadTransaction** UDA to True. When the script finishes processing, the **ColumnReadDone** UDA becomes True and the results are in the **ColumnValue** UDA.

QueryandProcess Script Configuration

On the **Script** tab of the SQLData Object Editor, configure the **QueryandProcess** script with the following attributes.

Attribute	**Value**
Execution Type	Execute
Expression	me.ReadTransaction
Trigger Type	OnTrue
Runs Asynchronously	Selected

QueryandProcess Script Code

Use the following sample code for the **QueryandProcess** script.

```
DIM Connection as aaDBClient.aaDBConnection;
DIM Command1 as aaDBClient.aaDBCommand;
DIM Transaction as aaDBClient.aaDBTransaction;

'Create a connection object with the connection string.
LogMessage("Creating connection");
Connection = aaDBAccess.CreateConnection("Data Source=localhost;Initial
Catalog=AdventureWorks;Integrated Security=true");

'Create a transaction object within the connection object.
LogMessage("Creating transaction object");
Transaction = Connection.CreateTransaction();

'Create a command object for the transaction object, with a SQL statement.
LogMessage("Creating a command object");
Command1 = Transaction.CreateCommand("Select * from Production.Product WHERE
ProductNumber = @ProductNumber", aaDBCommandType.SqlStatement, true);

'We used a parameter to specify the value for the ProductNumber field, so
initialize it.
Command1.SetCharParameterByName("ProductNumber", me.PartNumber,
aaDBParameterDirection.Input, 50);

'Everything is ready, let's execute the transaction sync.
LogMessage("Executing transaction sync");
DIM ResultCode as integer;
ResultCode = Transaction.ExecuteSync();
if ResultCode <> 0 then
     'Failed to execute transaction sync, report the reason.
     LogMessage("Got error " + ResultCode + " executing transaction sync");
  else
     if Transaction.ExecutionState == aaDBTransactionState.Completed then
        DIM Rows as integer;
        Rows = Command1.RowCount;
        LogMessage("Row count returned from command is " + Rows);

       'Use other methods of script library to read data and assign to UDAs, etc.
       if Rows > 0 then
          LogMessage("Getting column '" + me.ColumnToRead + "' from row 0");
          Command1.SelectRow(0);

         'Return the requested column value from this row, signal done.
         me.ColumnValue =
Command1.GetCurrentRowColumnByName(me.ColumnToRead);
         me.ColumnReadDone = true;
      endif;

      'When done, dispose the command.
      Command1.Dispose();
    endif;

    'When done, dispose the transaction.
    Transaction.Dispose();
  endif;

'Reset for next time
me.ReadTransaction = false;
```

CHAPTER 2

aaDBAccess Object

About aaDBAccess Object

The aaDBAccess object exposes only static methods. Use static methods with the aaDBAccess object to request a connection to the data source by providing a connection string. You can use two categories of methods to create a database connection:

- A reusable connection object: **GetConnection()**
- A unique connection object: **CreateConnection()**

By default, the SQLData Script Library assumes that a connection to a SQL Server database is requested and establishes a physical connection by using the **System.Data.SqlClient** namespace.

For details about methods that you can use with this object, see *Methods on page 30*.

Creating a Reusable Connection Object

To create a reusable connection object, use the following method:

```
aaDBAccess.GetConnection (<ConnectionString>, <ProviderType>)
```

For more information, see *Connecting to Databases Other Than SQL Server on page 31*.

Creating a Unique Connection Object

To create a unique connection object for a specific purpose, use the following method:

```
aaDBAccess.CreateConnection(<ConnectionString>, <ProviderType>)
```

Differences Between CreateConnection() and GetConnection()

The **CreateConnection()** method always creates a new connection object, even for calls with identical connection strings and even if **GetConnection()** has already been called with the same connection string.

Multiple calls to **GetConnection()** with identical—not similar— connection strings return the same connection object, over and over. If the connection string has never been used in a call to **GetConnection()**, a new connection object is created, but subsequent calls with the same connection string return the same connection object.

A call to **CreateConnection()** followed by a call to **GetConnection()** with the same connection string returns two different connection objects. That is, the connection object returned by **GetConnection()** is never the same as the connection object that has been returned by **CreateConnection()**.

Note: Because GetConnection() shares the same connection object across multiple scripts, connection pooling is effectively disabled for a specific connection string. All commands or transactions that run on a shared connection are routed through a single queue. Therefore, multiple physical connections for a single unique connection string cannot occur. If you want to use connection pooling, use the CreateConnection() method.

For more information, see *Connecting to Databases Other Than SQL Server on page 31*.

Working with Connections

Supply all connection settings in a single parameter named ConnectionString that follows the documented Microsoft syntax. For details about Microsoft syntax, follow this link:

http://msdn2.microsoft.com/en-us/library/ms254499.aspx

Note: The parsing of the ConnectionString is not case sensitive.

For authentication, you can use one of the following security modes:

- Windows Integrated Security
- Windows Account
- SQL Server Authentication

Windows Integrated Security

Internally, **IntegratedSecurity** provides the connection by using the credentials of the currently logged on ArchestrA user.

Specify the following syntax in the connection string:

```
Integrated Security=True
```

The keyword `Integrated Security=True` in the connection string overrides any other authentication control. If the `Integrated Security=True` keyword is present, you get Windows User Authentication for the user who is currently logged on, even if you also include credentials for a different user.

If you want to impersonate another Windows user's credentials, `Integrated Security=True` must be omitted from the connection string.

Windows Account

In the **ConnectionString parameter**, you must provide the following information: domain, user name and password. The domain and user name must be specified using the following syntax:

```
User ID=<Domain>\<UserName>; Password=<pasword>;
```

The SQLData Script Library takes the following actions:

1. Removes domain, user name, and password from <**ConnectionString**>.
2. Sets **Integrated Security**=**True** in <**ConnectionString**>.
3. Configures the connection manager with the specified domain, user name, and password and then requests impersonation of this user.

The SQLData Script Library impersonates the user based on the properties that were provided in the connection string.

SQL Server Authentication

Provide the following information in the **ConnectionString**;

```
User ID=<UserName>; Password=<password>;
```

The SQLData Script Library directly passes <**ConnectionString**> through to the **SQL Server database**.

The **aaDCM** object uses <**ConnectionString**> as is. There is no need for impersonation.

Methods

You can use the following methods with the **aaDBAccess** object.

CreateConnection()

Use the **CreateConnection()** method to request a connection to a SQL Server data source.

Syntax

```
aaDBConnection.CreateConnection(
string ConnectionString)
```

Parameters

ConnectionString
A previously formatted connection string or a reference to an attribute in any ArchestrA object.

Remarks

The **CreateConnection()** method returns a connection object to be used for subsequent SQL requests. Each call to **CreateConnection()** returns a unique and different connection object. Each connection object represents a separate connection that uses the same connection string. This method is best used to provide different connections for different purposes. For example, when one connection is used to query and another is used to update the database.

You can check the status of a connection by using the **ConnectionState** read-only property.

Example

The following example shows a connection string for use with SQL Server. The connection string can also be stored in an attribute in an Archestra object:

```
me.ExampleConnectionString
```

Connecting to Databases Other Than SQL Server

Use the overloaded **GetConnection()** method to access a data source other than Microsoft SQL Server.

Syntax

```
aaDBConnection.GetConnection(
string ConnectionString,
aaDBConnectionType ConnectionType)
```

Parameters

Acceptable values for **ConnectionType** are as follows:

- OLEDB
- Oracle

Note: Use OLEDB to connect to the Microsoft Access data source.

GetCommand()

Use the **GetCommand()** method to retrieve a reference to an **aaDBCommand** object created previously with the same or a different script. Do not create a new **aaDBConnection** object to access a previously created command object.

Syntax

```
aaDBCommand.GetCommand(
string CommandId)
```

Parameters

commandId
The Id is generated internally by the SQLData Script Library. For details, see GetId() on page 42.

Remarks

If **CommandID** does not represent a valid ID, **GetCommand()** returns a null reference.

GetConnection()

Use the **GetConnection()** method to request a connection to a data source.

Syntax

```
aaDBConnection.GetConnection(
string ConnectionString)
```

Parameters

ConnectionString
A previously formatted connection string or valid ArchestrA reference.

Remarks

The **GetConnection()** method returns a connection object to be used for subsequent SQL Server requests. Each time that **GetConnection()** is called with the identical connection string to a previous call, it returns the same connection object for reuse. This method is best used with a script that runs repeatedly, where a new connection object for each iteration would constitute a risk of memory leakage.

You cannot assume that a physical connection occurs after requesting a connection to a data source. Connections are opened only on an as-needed basis to perform an operation. The only time that you can check connectivity status is immediately after an operation completes.

This method immediately provides a connection object to be used for subsequent SQL requests. The first call to **GetConnection()** returns a unique connection object for the specified connection string, similar to **CreateConnection()**.

Subsequent calls to **GetConnection()** return references to the same connection object. Thus, you can make multiple calls to reuse the same object.

The connection returned by **GetConnection()** is never the same object as the one returned by **CreateConnection()**. This difference enables you to place **GetConnection()** calls at the top of a script that run once per scan without constantly creating connection objects.

Note: Be sure to call Dispose() on connection objects that have been created with this method. The aaDBAccess SQLData Script Library contains a reference to the objects. Garbage collection cannot be performed on them until you call Dispose().

You can check the status of the connection by using the **ConnectionState** read-only property.

Authentication

The following three database authentication methods are supported:

- Windows Integrated Security
- Windows Account
- SQL Server Authentication

You must provide a standard connection string that is created from keyword = value pairs separated by semicolons.

Follow this link for a list of connection string keywords:

http://msdn2.microsoft.com/en-us/library/ms254499.aspx

For Windows Integrated Security specify the following parameter in the connection string:

```
Integrated Security=true;
```

For Windows Account authentication, specify a connection string that adheres to the following syntax:

```
User ID=<domain>\<username>;Password=<password>;
```

For SQL Server Authentication, specify a connection string that adheres to the following syntax:

```
User ID=<UserName>;Password=<password>;
```

Connecting to Databases Other Than SQL Server

Use the overloaded **GetConnection()** method to access a data source other than Microsoft SQL Server.

Syntax

```
aaDBConnection.GetConnection(
string ConnectionString,
aaDBConnectionType ConnectionType)
```

Parameters

Acceptable values for **ConnectionType** are as follows:

- OLEDB
- Oracle

Note: Use OLEDB to connect to the Microsoft Access data source.

GetDiagnostics()

Use the **GetDiagnostics()** method to return diagnostic information about all connections in a dataset.

Syntax

```
void GetDiagnostics()
```

Remarks

The returned dataset contains multiple tables. The table with index 0 contains the global diagnostic information. The remaining tables in the dataset correspond to each DCM connection object. For details about diagnostic properties, see the *SQLData Object Help*.

GetTransaction()

Use the **GetTransaction()** method to obtain a reference to an **aaDBTransaction** object created previously in the same or a different script. Do not create a new **aaDBConnection** object to access a previously created transaction.

Syntax

```
aaDBTransaction GetTransaction(
string TransactionID)
```

Parameters

TransactionID
The ID is generated internally by the SQLData Script Library. See GetID() on page 89.

Remarks

If **TransactionID** does not represent a valid ID, **GetTransaction()** returns a null reference.

LogDiagnostics()

Use the **LogDiagnostics()** method to create a snapshot of all diagnostics available for a connection to be dumped to the logger. For details about diagnostic properties, see the *SQLData Object Help*.

Syntax

```
void LogDiagnostics()
```

RemoveCommand()

Use the **RemoveCommand()** method to instruct the SQLData Script Library to remove internal references to the **aaDBCommand** object referenced by **CommandID**, release all resources used by the object, and clean up all references to the object in memory.

Syntax

```
void RemoveCommand(
string CommandID)
```

Parameters

CommandID
The unique ID, generated internally by the SQLData Script Library. For details, see GetId() on page 42.

Remarks

You must call **RemoveCommand()** if you previously requested **CommandID**. Otherwise, the memory cannot be released until the engine process is shut down.

Do not keep this command object in memory, especially when it is associated with a large dataset.

RemoveTransaction()

Use the **RemoveTransaction()** method to instruct the SQLData Script Library to remove all references to the **aaDBTransaction** object referred to by **TransactionID**, release all resources used by the object, and clean up all references to the object in memory.

Syntax

```
void RemoveTransaction(
string TransactionID)
```

Parameters

TransactionID
The unique ID, generated internally by the SQLData Script Library. See aaDBTransaction.GetID() on page 89.

Remarks

Internally, the SQLData Script Library ensures that all **aaDBCommand objects** explicitly added to this object are removed.

You must call this method if you previously requested the transaction ID. Otherwise, the memory cannot be released until the engine process is shut down.

Do not keep this command object in memory, especially when it is associated with a large dataset.

ResetDiagnostics

Use the **ResetDiagnostics()** method to reset the current diagnostic values associated with the **DCMConnectionMgr** and all **DCMConnections**.

Syntax

```
void ResetDiagnostics()
```

Shutdown()

Use the **Shutdown()** method to gracefully cancel outstanding command object requests and release references to all persisted **aaDBCommand** and **aaDBTransaction** objects.

Call this method just once from a shutdown script in the hosting engine.

Syntax

```
void Shutdown()
```

CHAPTER 3

aaDBCommand Object

About aaDBCommand Object

Use objects of type **aaDBCommand** to process SQL statements and stored procedures or to access a single table or view.

Create instances of type **aaDBCommand** by calling **CreateCommand()** on an instance of the **aaDBConnection object**. For example, assuming that the **aaDBConnection** instance is called **Connection**:

```
Connection.CreateCommand()
```

When a script requests the command object ID, the command object is flagged to be persisted.

The **aaDBCommand** objects are persisted across scripts and scan cycles but not across failover or shutdown.

Note: You must call Dispose() on each instance of aaDBCommand where an ID was created. You can also call **aaDBAccess.RemoveCommand()** with the ID.

You can retrieve an **aaDBCommand** object at any time by calling the static method aaDBAccess.GetCommand() and passing the previously acquired string ID.

The **aaDBCommand** object provides support for types that make sense for QuickScript. Other types may be supported by database columns, such as large text files or generic binary large objects (BLOBs), but a script might not be able to generate or analyze them. In general, if the script cannot manipulate objects of a particular type, such as a BLOB of type char[] or byte[], it might still be possible to read or write an object of that type to or from some alternate script library while storing it as type object within the script. In these cases, the object may be blindly written to a database using **SetCurrentRowColumnByName()** or **SetCurrentRowColumnByIndex()** and may be blindly read from a database using **GetCurrentRowColumnByName()** or **GetCurrentRowCollumnByIndex()**.

The life cycle of the aaDBCommand object follows this pattern:

1. The object is created by calling the **aaDBConnection.CreateCommand() method**.
2. Parameters are added to the command object as necessary.
3. The command object is run synchronously or asynchronously.
4. The **dataset** object wrapped by the **aaDBCommand** object is accessed and manipulated. This process may involve writing the **dataset** object back to the database.
5. Steps 2 through 4 can be repeated as needed.

Methods

You can use the following methods with the aaDBCommand object.

AddRow()

Use the **AddRow()** method to add an empty row to a memory table and make it current.

You can then add values to the row by using the following methods:

- **SetCurrentRowColumnByIndex()**

- **SetCurrentRowColumnByName()**
- **SetCurrentRow()**

Syntax

```
result AddRow()
```

Remarks

The actual update to the data source is delayed until you call **SaveChangesSync()** or **SaveChangesAsync()**.

DeleteCurrentRow()

Use the **DeleteCurrentRow()** method to mark the currently selected row for deletion. The current row is not deleted from the data source until you call **SaveChangesSync()** or **SaveChangesAsync()**.

Syntax

```
result DeleteCurrentRow()
```

Dispose()

Use the **Dispose()** method to instruct the SQLData Script Library to free all memory resources associated with the command object. If the command is running, **Dispose()** cancels it. It is preferred to cancel the command before calling **Dispose()**.

Syntax

```
Void Dispose()
```

Remarks

After **Dispose()** is called, subsequent method calls to the command fail.

If an ID has been retrieved for this command object, you must call **Dispose()** or **aaDBAccess.RemoveCommand()**.

If you requested the ID of this command object, it is very important that you call **Dispose()** so it can flag the SQLData Script Library to clear all the references to this object.

When you request a **dataset** it is very important for you to call the **Dispose()** method when you are no longer interested in the results of a specific command object.

If you have made changes to the memory **dataset** such as updating, deleting, or adding, but did not issue SaveChangesSync() or SaveChangesAsync(), all changes are discarded.

Subsequent method calls to request scrolling or to modify the memory **dataset** fail and return either a null object or error code 1016.

ExecuteAsync()

Use the **ExecuteAsync()** method to queue a command object in the connection for later processing. **ExecuteAsync()** returns immediately, and processing occurs in the background.

Syntax

```
result ExecuteAsync( )
```

Remarks

You can check for status by reading the ExecutionState read-only property.

After **ExecuteAsync()** is processed, you can still obtain a reference to the command objects and analyze their **ExecutionState** and **LastError** properties.

To free all resources allocated for the command object, you must call **Dispose()**.

ExecuteAsyncCancel()

Use the **ExecuteAsyncCancel()** asynchronous method to cancel the operations of a running command. If the command has already started running, it may run to completion. If the command is queued while waiting for other commands, it is removed from the queue without running.

Syntax

```
result ExecuteAsyncCancel( )
```

Remarks

To check for status, you can request the ExecutionState read-only property.

After the method has successfully completed, the command object is canceled.

ExecuteSync()

Use the **ExecuteSync()** method to run the command object. This method runs synchronously and blocks the engine thread.

Syntax

```
result ExecuteSync( )
```

Remarks

Use the ExecutionState and **LastExecutionError** properties to check for status of this method.

After **ExecuteSync()** is processed you can still obtain a reference to the command object and analyze its **ExecutionState** and **LastError** properties.

Use the **LastExecutionError** property to check if the command failed or succeeded (success is indicated by a blank string), since ExecuteSync() may return 0 in the case of a failure instead of an error message.

To free all resources allocated for the **command** object, you must call Dispose().

GetCurrentRowColumnByIndex()

Use the **GetCurrentRowColumnByIndex()** method to obtain a specific column value from the current row. Use this method to read a single column value. For more information, see *SelectRow() on page 45*.

You must provide a zero-based column index.

Syntax

```
object GetCurrentRowColumnByIndex(
int ColumnNumber)
```

Parameters

ColumnNumber
Zero-based index to the table column.

Return Value

The object returned on failure is **Null**. The requested index is either negative, larger than the number of columns, or there is no valid current row.

GetCurrentRowColumnByName()

Use the **GetCurrentRowColumnByName()** method to obtain a specific column value from the current row by column name. Use this method to read a single column value. For more information, see *SelectRow() on page 45*.

You must provide a column name.

Syntax

```
object GetCurrentRowColumnByName(
string ColumnName)
```

Parameters

ColumnName
Column name.

GetDataSet()

Use the **GetDataSet()** method to retrieve the **dataset** stored in memory that was generated when the command object finished processing.

Syntax

```
DataSet GetDataset( )
```

Remarks

It is not recommended that you use this method under normal conditions, because the actual **dataset** may contain large amount of data (huge number of rows). This method is provided for advanced users that want to directly use the SQLClient namespace to interact with the **dataset**.

Note: If you execute this method in a synchronous script with a large number of rows, the script may time out. If you plan to use this method, it is recommended that you configure the script to run asynchronously.

ArchestrA QuickScript uses 1-based indexing when square-bracket notation is used with numeric indexes, such as **DataSet.Tables[7]**.

To avoid confusion, use square-bracket notation with collections that support strong name indexing, such as **DataSet.Tables["Customers"]**. You can also bypass the **dataset** object by using the wrapping accessor functions described in the next sections.

GetId()

Use the **GetID()** method to retrieve the ID of an **aaDBCommand** object instance for use in a different script or scan. The SQLData Script Library generates a unique command object ID, which remains unique across all scripts on the engine.

Syntax

```
string GetId()
```

Return Value

On failure, this method returns the value **Null**.

Remarks

The SQLData Script Library returns a string value, but the script engine automatically attempts to cast this value to other types. If the script assigns the returned ID to any type other than string, the ID is corrupted and does not work in future **GetCommand(ID)** calls.

GetParameterByIndex()

Use the **GetParameterByIndex()** method to retrieve output or to return parameters after the command object has been processed. If the parameter cannot be evaluated, the returned value is **null**.

Syntax

```
object GetParameterByIndex(
int Index)
```

Parameters

Index
1-based parameter index.

Remarks

You must provide a one-based parameter index.

Note: Use this method only for parameters defined in an OLEDB-type query, as specified by the connection object. If the connection type is not OleDb, null is returned.

GetParameterByName()

Use the **GetParameterByName()** method to retrieve output or return parameters after the command object has been processed. If the parameter cannot be evaluated, the returned value is **null**.

You must provide a parameter name.

Syntax

```
object GetParameterByName(
string ParameterName)
```

Parameters

ParameterName
Parameter name.

Remarks

It is not considered an error if the **ResultSet** returned from a query contains no rows. Zero-row results can also be expected as results of queries that contain parameters, so this might be an expected result, depending on the purpose of the query.

Note: Use this method only for parameters defined in a non-OLEDB-type query, as specified by the connection object. If the connection type is not OleDb, null is returned.

GetRow()

Use the **GetRow()** method to quickly scroll through the records in the memory table and examine the row values and determine an index to be used for selecting a row of interest. For more information, see *SelectRow() on page 45*.

You must provide a zero-based row index.

Syntax

```
aaDBRow GetRow(
int RowNum)
```

Parameters

RowNum
A zero-based row index less than the row count in the memory **dataset**.

Remarks

This method executes synchronously.

The **aaDBRow** returned object has two members:

- **columnNames**: An array filled with column names.

- **columnValues**: An array filled with column values corresponding to the row index.

The array items must be converted to strings before you can use them in string manipulation.

After the command object is successfully run you can use the **columnValues** array to set ArchestrA attributes.

Note: Before setting ArchestrA attributes you may need to cast the individual items from the columnValues array.

SaveChangesAsync()

Use the **SaveChangesAsync()** method to instruct the SQLData Script Library to write back to the data source all changes made to the memory **dataset**. The **dataset** must have been acquired by calling **ExecuteAsync()** or **ExecuteSync()**.

Syntax

```
result SaveChangesAsync()
```

Remarks

This method runs asynchronously. The request for updating the data source is queued and the method completes immediately.

You must check for status by getting the ExecutionState property value.

SaveChangesSync()

This method is similar to SaveChangesAsync() except that it runs synchronously.

Syntax

```
result SaveChangesSync()
```

Remarks

This method blocks the engine thread when the script itself is synchronous.

It is highly recommended that you use this method only in asynchronous scripts.

When you configure the asynchronous script, make sure that the **TimeoutLimit** value is large enough to accommodate the time that this method may take to run the command object.

SelectRow()

Use the **SelectRow()** method to select the row with the index **rowNumber** in the memory table.

Syntax

```
result SelectRow(
long RowNumber)
```

Parameters

RowNum
The zero-based index of the record in the memory table.

Remarks

After you select a row, you can then read, delete or update the row using the following methods:

- **GetCurrentRowColumnByIndex()**
- **GetCurrentRowC()**
- **SetCurrentRowColumnByIndex()**

- **SetCurrentRowColumnByName()**
- **SetCurrentRow()**

SelectTable()

Use the **SelectTable()** method to select the table with the index TableIndex in the memory **dataset**.

You can then read, delete, or update this **DataTable**.

Syntax

```
result SelectTable(
long RowNumber)
```

Parameters

TableNumber
A zero-based index of the table in the memory **dataset**.

Remarks

By default, immediately after a command object returns a non- empty **dataset** that runs successfully the table with index 0 is selected.

SetCurrentRow()

Use the **SetCurrentRow()** method to set values for multiple columns at the same time. You must provide the column names to set and the corresponding values to be set, respectively, in the following two members of the **Row** parameter:

- **columnNames**
- **columnValues**

Syntax

```
result SetCurrentRow (
aaDBRow Row)
```

Parameters

Row
You can construct two **ArrayList objects** by using string constants or valid Archestra reference strings.

Remarks

When you construct the input **ArrayList** object, you can specify constants as well as valid ArchestrA reference strings.

Necessary conversions are performed internally to cast values to the specific column type.

The actual update to the data source is delayed until you call SaveChangesSync() or SaveChangesAsync().

If **SetCurrentRow()** encounters a failure to change a column, some columns may have already been changed in the internal dataset. In this case, the internal dataset reverts to the state it had immediately after the last update or **SaveChanges()** call. It is possible that this reversion of the internal dataset could undo changes applied prior to the current **SetCurrentRow()** call. For instance, if the script has a loop to modify 10 rows with **SetCurrentRow()** and nine rows return without a bad error code but the tenth row returns a bad error code, the internal dataset reverts to its state before the first **SetCurrentRow()** call.

Example

```
Dim inputCol as System.Collections.ArrayList;
Dim inputVal as System.Collections.ArrayList;
Dim inputRow as aaDBClient.aaDBRow;
```

```
inputCol = new System.Collections.ArrayList();
inputCol.Add("Name");
inputCol.Add("Description");
inputCol.Add("DateTime");
inputVal = new System.Collections.ArrayList();
inputVal.Add("Name");
inputVal.Add(me.strValue);
inputVal.Add(me.DT);

inputRow.columnNames = inputCol;
inputRow.columnValues = inputVal;
```

SetCurrentRowColumnByIndex()

Use the **SetCurrentRowColumnByIndex()** method to update the column at index **columnNumber** in the currently selected row.

Syntax

```
result SetCurrentRowColumnByIndex (
int ColumnNumber,
object NewValue)
```

Parameters

ColumnNumber
The zero-based index of the column in the memory table.

NewValue
You can specify a constant or a valid ArchestrA reference string. Both relative references and fully qualified references are supported. For example:

- me.ShortDesc – relative reference
- UD1.status – fully qualified reference string
- "John Smith" – constant

Remarks

Necessary conversions are performed internally to cast **newValue** to the column type.

When you try to change a value using this method and the type passed in does not match the type specified for the column in the database table, one of the following outcomes occurs:

- The value is written with an automated type conversion
- The transaction fails completely because the conversion would result in loss of data.

The actual update to the data source is delayed until you call SaveChangesSync() or SaveChangesAsync().

SetCurrentRowColumnByName()

Use the **SetCurrentRowColumnByName()** method to update the column named **ColumnName** in the currently selected row.

Syntax

```
result SetCurrentRowColumnByName (
string ColumnName,
object NewValue)
```

Parameters

ColumnName
String constant that is enclosed in quotation marks or another ArchestrA reference that can evaluate to a string. For example:
"LastName" – Spaces are allowed.
For more details see SQL Server documentation regarding column naming rules.

NewValue
You can specify a constant or a valid ArchestrA reference string. Both relative references and fully qualified references are supported. For example:

- me.ShortDesc – relative reference
- UD1.status – fully qualified reference string
- "John Smith" – constant

Remarks

Necessary conversions are performed internally to cast newValue to the column type.

When you try to change a value using this method and the type passed in does not match the type specified for the column in the database table, one of the following outcomes occurs:

- The value is written with an automated type conversion
- The transaction fails completely because the conversion would result in loss of data.

The actual update to the data source is delayed until you call **SaveChangesSync()** or **SaveChangesAsync()**.

SetParam Type Methods for SQL Server and Oracle

Frequently, the SQL statement that you use to create a command has parameters encoded in it. In such cases, you must set values for all parameters needed as input by the SQL statement before you run the command.

For SQL Server databases, parameters are indicated by @<ParameterName> so that each parameter is named. The following method call accommodates named parameters:

```
Set<Type>ParameterByName()
```

You must use the version of the method that matches the way that you created your connection object. When you create a connection object with **aaDBConnectionTypeOleDb**, you must use the method **Set<Type>ParameterByIndex()**. For all other connections, use the method **Set<Type>ParameterByName()**.

Parameters are usually mapped to a table column, which always has a specific type. The method properties that you must supply with each of the methods in this subsection depend on the column type that they map to.

Separate methods are necessary because different column types require that you specify different method parameters.

For example, a column of type NVarChar requires a length be specified while a column type of Decimal requires precision and scale.

The method parameters that are common across all methods are:

- Parameter Name (parameterName)
- Parameter Value (parameterValue)

You can specify null for **parameterValue** when the Parameter Direction is Output or ReturnValue.

Parameter Direction (parameterDirection) enumerated values are:

- Input
- InputOutput
- Output
- ReturnValue

Output Parameters

The **GetParameterByIndex()** and **GetParameterByName()** methods return values to the script through InputOutput, Output, or ReturnValue parameters.

It is not possible to return values directly to ArchestrA attributes through InputOutput, Output or ReturnValue parameters, only DIM script variables. This is a constraint of the scripting infrastructure implemented in Application Server version 3.0.

If you want the value returned by **GetParameterByIndex()** or **GetParameterByName()** to set a value to an ArchestrA attribute reference, perform the following steps:

1. Declare a local script variable and return the InputOutput, Output, or ReturnValue parameter by making a **GetParameterByIndex()** or **GetParameterByName()** method call.
2. Use this local variable to set the desired ArchestrA reference.

SetBitParameterByName()

Note: Do not use this method for the Oracle data type Boolean value. Use **SetIntParameterByName** instead.

Use the **SetBitParameterByName()** method to configure a bit parameter by the name encoded into the text of a SQL statement using the @ character.

Syntax

```
result SetBitParameterByName (
string ParameterName,
object  ParameterValue,
aaDBParameterDirection ParameterDirection)
```

Parameters

ParameterName
String identifier used in the SQL statement.
In the following SQL statement the parameter name is boolValue.

```
"SELECT StateProvinceID
    ,StateProvinceCode
    ,CountryRegionCode
    ,IsOnlyStateProvinceFlag
    ,Name
    ,TerritoryID
    ,rowguid
    ,ModifiedDate
FROM AdventureWorks.Person.StateProvince
WHERE IsOnlyStateProvinceFlag = @boolValue"
```

ParameterValue
A discrete value or a valid ArchestrA reference string. For example:

- 0
- false
- me.boolValue

Example

```
SetBitParameterByName ("boolValue", me.boolValue,
aaDBParameterDirection.Input)
```

SetCharParameterByName()

Use the **SetCharParameterByName()** method to configure a character string parameter by the name encoded into the text of a SQL statement using the @ character.

Syntax

```
result SetCharParameterByName (
string ParameterName,
object ParameterValue,
aaDBParameterDirection ParameterDirection,
int Length)
```

Parameters

ParameterName
The string identifier that is used in the SQL statement.
In the following SQL statement the input parameter is "lastName."

```
"SELECT * FROM Person.Contact WHERE (LastName = @lastName)"
```

ParameterValue
A string constant or valid ArchestrA reference string. For example:

- "Smith"
- me.LastName
- null - Output and ReturnValue parameters.

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Length
Specifies the maximum length of the parameter.
If the length of the parameterValue is greater, the **parameterValue** is truncated to the specified length.

Example

```
SetCharParameterByName ("lastName", "Smith",aaDBParameterDirection.Input, 50)
```

SetDecimalParameterByName()

Use the **SetDecimalParameterByName()** method to configure a decimal parameter by the name encoded into the text of a SQL statement using the @ character.

Syntax

```
result SetDecimalParameterByName (
string ParameterName,
object ParameterValue,
aaDBParameterDirection ParameterDirection,
short Precision,
short Scale)
```

Parameters

ParameterName
The string identifier used in the SQL statement.
In the following SQL statement the name of the parameter is "RejectedQuantity".

```
"INSERT INTO Purchasing.PurchaseOrderDetail
   (PurchaseOrderID
   ,DueDate
   ,OrderQty
   ,ProductID
   ,UnitPrice
   ,ReceivedQty
   ,RejectedQty
   ,ModifiedDate)
   VALUES
   (4,'2008-01-28',4,4,4,12.89,@RejectedQuantity,
   '2008-01-28') "
```

ParameterValue
This parameter supports float, double or string values.
You can also specify a valid ArchestrA reference string. For example:

- 123.333
- "123.333333333333333333"
- me.Quantity

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Precision
A number that indicates the total number of digits.

Scale
A number that indicates the number of digits to the right of the decimal point.

Example

SetDecimalParameterByName ("RejectedQuantity", me.Quantity,aaDBParameterDirection.Input, 8, 2)

SetDateTimeParameterByName()

Use the **SetDateTimeParameterByName()** method to configure a **DateTime** parameter by the name encoded into the text of a SQL statement using the @ character.

Syntax

```
result SetDateTimeParameterByName (
string ParameterName,
object ParameterValue,
aaDBParameterDirection ParameterDirection)
```

Parameters

ParameterName
The string identifier used in the SQL statement. For example, in the following SQL statement, the name of the parameter is "NewDate"

```
"INSERT INTO Purchasing.PurchaseOrderDetail
   (PurchaseOrderID
   ,DueDate
   ,OrderQty
   ,ProductID
   ,UnitPrice
   ,ReceivedQty
   ,RejectedQty
   ,ModifiedDate)
   VALUES
   (4,'2008-01-28',4,4,4,12.89,12.56,@NewDate)"
```

ParameterValue
The **DateTime,** string value, or a valid ArchestrA reference string.

For example:

- '2004-03-11 10:17:21.587'
- me.dtValue

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Example

SetDateTimeParameterByName ("NewDate",me.dtValue,aaDBParameterDirection.Input)

SetDoubleParameterByName()

Use the **SetDoubleParameterByName()** method to configure a double parameter by the name encoded into the text of a SQL statement using the @ character.

Syntax

```
result SetDoubleParameterByName (
string ParameterName,
object  ParameterValue,
aaDBParameterDirection ParameterDirection)
```

Parameters

ParameterName
The string identifier used in the SQL statement.
In the following SQL statement the output parameter name is "AvgReject"

```
"SELECT @AvgReject = Avg(RejectedQty)
FROM Purchasing.PurchaseOrderDetail "
```

ParameterValue
A double-precision floating number or a valid ArchestrA reference string. For example:

- 123.333
- me.Limit

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Example

```
SetDoubleParameterByName ("AvgReject",null,aaDBParameterDirection.Output)
```

SetFloatParameterByName()

Use the **SetFloatParameterByName()** method to configure a float parameter by the name encoded into the text of a SQL statement using the @ character.

Syntax

```
result SetFloatParameterByName (
string ParameterName,
object  ParameterValue,
aaDBParameterDirection ParameterDirection
```

Parameters

ParameterName
String identifier used in the SQL statement.
In the following SQL statement the name of the output parameter is "AvgReject."

```
"SELECT @AvgReject = Avg(RejectedQty)
FROM Purchasing.PurchaseOrderDetail "
```

ParameterValue
A floating number or a valid ArchestrA reference string. For example:

- 123.333
- me.Limit

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Example

```
SetFloatParameterByName ("AvgReject", null, aaDBParameterDirection.Output)
```

SetIntParameterByName()

Use the **SetIntParameterByName()** method to configure an integer parameter by the name encoded into the text of a SQL statement using the @ character.

Note: Use this method for the Oracle data type Boolean. It is specific to Oracle.

Syntax

```
result SetIntParameterByName (
string ParameterName,
object  ParameterValue,
aaDBParameterDirection ParameterDirection)
```

Parameters

ParameterName
The string identifier that is used in the SQL statement.
In the following SQL statement the name of the output parameter is "Cnt"

```
"SELECT @Cnt=Count (*)(RejectedQty)
FROM Person.Contact "
```

ParameterValue
An integer value or a valid ArchestrA reference string. For example:

- 10
- me.OrderCount
- Null - Output

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Example

```
SetIntParameterByName ("Cnt", null, aaDBParameterDirection.Output)
```

SetLongParameterByName()

Use the **SetLongParameterByName()** method to configure a long parameter by the name encoded into the text of a SQL statement using the @ character.

Syntax

```
result SetLongParameterByName (
string ParameterName,
object  ParameterValue,
aaDBParameterDirection ParameterDirection)
```

Parameters

ParameterName
The string identifier that is used in the SQL statement.

In the following SQL statement the name of the output parameter is "ProdID"

```
"SELECT ProductID, SUM(WorkOrderID)
AS OrderCnt
FROM Production.WorkOrder
WHERE ProductID = @ProdID
GROUP BY ProductID"
```

ParameterValue
An integer value or a valid ArchestrA reference string. For example:

- 10
- me.ProductID
- Null - Output

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Example

```
SetLongParameterByName ("ProdID",me.ProductID,aaDBParameterDirection.Output)
```

SetParam Type Methods for OLEDB

Frequently, the SQL statement that you use to create a command has parameters encoded in it. In such cases, you must set values for all parameters needed as input by the SQL statement before you run the command.

For data providers accessed through OLEDB, parameters are indicated by identical placeholder characters, and parameters must be indicated by index. The method call **Set<Type>ParameterByIndex()** accommodates the indexed parameters of OLEDB.

Parameters are usually mapped to a table column, which always has a specific type. The method parameters that you supply with each of the methods in this subsection vary depending on the column type that they map to.

Note: Because parameters are sequentially numbered, it is important that there be no gaps in the sequence. The sequence is checked when the command object with parameters is executed. Gaps cause the command object to fail.

It is necessary to have separate methods because different column types require that you specify different method parameters.

For example a column of type NVarChar requires that you specify a length, while a column type of Decimal requires that you specify precision and scale.

The following method parameters are common across all methods:

- Parameter Index (**Index**) – this is 1-based
- Parameter Value (**parameterValue**)

  You can specify **null** for **parameterValue** when the **parameterDirection** is **Output** or **ReturnValue**.
- Parameter Direction (**parameterDirection**)

The following are enumerated values:

- Input
- InputOutput
- Output
- ReturnValue

SetBitParameterByIndex()

Use the **SetBitParameterByIndex()** method to configure a bit parameter as encoded into the text of an OLEDB SQL statement.

Syntax

```
result SetBitParameterByIndex (
int Index,
object  ParameterValue,
aaDBParameterDirection ParameterDirection)
```

Parameters

Index
The sequential index of the parameter as used in the SQL statement.
In the following SQL statement, the parameter corresponds to the question mark (?) character:

```
"SELECT StateProvinceID
   ,StateProvinceCode
   ,CountryRegionCode
   ,IsOnlyStateProvinceFlag
   ,Name
   ,TerritoryID
   ,rowguid
   ,ModifiedDate
FROM AdventureWorks.Person.StateProvince
WHERE IsOnlyStateProvinceFlag = ?"
```

ParameterValue
A discrete value or a valid ArchestrA reference string. For example:

- 0.
- false.
- me.boolValue.
- ParameterDirection. For possible values, see *aaDBParameterDirection on page 71*.

Example

```
SetBitParameterByIndex (0, me.boolValue, aaDBParameterDirection.Input)
```

SetCharParameterByIndex()

Use the **SetCharParameterByIndex()** method to configure a character string parameter as encoded into the text of an OLEDB SQL statement.

Syntax

```
result SetCharParameterByIndex (
int Index,
string ParameterValue,
aaDBParameterDirection ParameterDirection,
int Length)
```

Parameters

Index
The sequential index of the parameter as used in the SQL statement.
In the following SQL statement the input parameter corresponds to the question mark (?) character.

```
"SELECT * FROM Person.Contact WHERE (LastName > ?)"
```

ParameterValue
A string constant or valid ArchestrA reference string. For example:

- "Smith"
- me.LastName
- null - Output and ReturnValue parameters

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Length
The maximum length of the parameter.
If the length of the parameterValue is greater, the parameterValue is truncated to the specified length.

Example

```
SetCharParameterByIndex (0, "Smith",aaDBParameterDirection.Input, 50)
```

SetDateTimeParameterByIndex()

Use the **SetDateTimeParameterByIndex()** method to configure a DateTime parameter as encoded into the text of an OLEDB SQL statement.

Syntax

```
result SetDateTimeParameterByIndex (
int Index,
object ParameterValue,
aaDBParameterDirection Parameterdirection)
```

Parameters

Index
The sequential index of the parameter as used in the SQL statement. For example:
In the following SQL statement the parameter corresponds to the question mark (?) character:

```
"INSERT INTO Purchasing.PurchaseOrderDetail
   (PurchaseOrderID
   ,DueDate
   ,OrderQty
   ,ProductID
   ,UnitPrice
   ,ReceivedQty
   ,RejectedQty
   ,ModifiedDate)
VALUES
   (4,'2008-01-28',4,4,4,12.89,12.56,?)"
```

ParameterValue
DateTime, string value, or a valid ArchestrA reference string.
For example:

- '2004-03-11 10:17:21.587'
- me.dtValue

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Example

```
SetDateTimeParameterByIndex (1, me.dtValue, aaDBParameterDirection.Input)
```

SetDecimalParameterByIndex()

Use the **SetDecimalParameterByIndex()** method to configure a decimal parameter as encoded into the text of an OLEDB SQL statement.

Syntax

```
result SetDecimalParameterByIndex (
int Index,
object ParameterValue,
aaDBParameterDirection ParameterDirection,
short Precision,
short Scale)
```

Parameters

Index
Sequential index of the parameter as used in the SQL statement. For example:
In the following SQL statement the parameter corresponds to the question mark (?) character:

```
"INSERT INTO Purchasing.PurchaseOrderDetail
    (PurchaseOrderID
    ,DueDate
    ,OrderQty
    ,ProductID
    ,UnitPrice
    ,ReceivedQty
    ,RejectedQty
    ,ModifiedDate)
VALUES
    (4,'2008-01-28',4,4,4,12.89,?,'2008-01-28') "
```

ParameterValue
This parameter supports float, double, or string values.
You can also specify a valid ArchestrA reference string. For example:

- 123.333
- "123.333333333333333333"
- me.Quantity

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Precision
A number that indicates the total number of digits.

Scale
A number that indicates the number of digits to the right of the decimal point.

Example

```
SetDecimalParameterByIndex (1, me.Quantity,aaDBParameterDirection.Input, 8, 2)
```

SetDoubleParameterByIndex()

Use the **SetDoubleParameterByIndex()** method to configure a double parameter as encoded into the text of an OLEDB SQL statement.

Syntax

```
result SetDoubleParameterByIndex (
int Index,
object  ParameterValue,
aaDBParameterDirection ParameterDirection)
```

Parameters

Index
The sequential index of the parameter as used in the SQL statement.

In the following SQL statement, the output parameter corresponds to the question mark (?) character:

```
"SELECT ? = Avg(RejectedQty)
FROM Purchasing.PurchaseOrderDetail "
```

ParameterValue
A double-precision floating number or a valid ArchestrA reference string. For example:

- 123.333
- me.Limit

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Example

```
SetDoubleParameterByIndex (1, null, aaDBParameterDirection.Output)
```

SetFloatParameterByIndex()

Use the **SetFloatParameterByIndex()** method to configure a float parameter as encoded into the text of an OLEDB SQL statement.

Syntax

```
result SetFloatParameterByIndex (
int Index,
object  ParameterValue,
aaDBParameterDirection ParameterDirection)
```

Parameters

Index
The sequential index of the parameter as used in the SQL statement.
In the following SQL statement the output parameter corresponds to the question mark (?) character:

```
"SELECT ? = Avg(RejectedQty)
FROM Purchasing.PurchaseOrderDetail "
```

ParameterValue
A double-precision floating number or a valid ArchestrA reference string. For example:

- 123.333
- me.Limit

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Example

```
SetFloatParameterByIndex (1, null, aaDBParameterDirection.Output)
```

SetIntParameterByIndex()

Use the **SetIntParameterByIndex()** method to configure a integer parameter as encoded into the text of an OLEDB SQL statement.

Syntax

```
result SetIntParameterByIndex (
int Index,
object  ParameterValue,
aaDBParameterDirection ParameterDirection)
```

Parameters

Index
The sequential index of the parameter as used in the SQL statement.

In the following SQL statement the output parameter corresponds to the question mark (?) character.:

```
"SELECT ? = Avg(RejectedQty)
FROM Purchasing.PurchaseOrderDetail "
```

ParameterValue
A double-precision floating number or a valid ArchestrA reference string. For example:

- 123.333
- me.Limit

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Example

```
SetIntParameterByIndex (1, null, aaDBParameterDirection.Output)
```

SetLongParameterByIndex()

Use the **SetLongParameterByIndex()** method to configure a long parameter as encoded into the text of an OLEDB SQL statement.

Syntax

```
result SetLongParameterByIndex (
int Index,
object  ParameterValue,
aaDBParameterDirection ParameterDirection)
```

Parameters

Index
The sequential index of the parameter as used in the SQL statement.
In the following SQL statement the output parameter corresponds to the question mark (?) character:

```
" SELECT ProductID, SUM(WorkOrderID) AS OrderCnt
FROM Production.WorkOrder
WHERE ProductID = ?
GROUP BY ProductID"
```

ParameterValue
A double-precision floating number or a valid ArchestrA reference string. For example:

- 123.333
- me.ProductID

ParameterDirection
For possible values, see *aaDBParameterDirection on page 71*.

Example

```
SetLongParameterByIndex (1, me.ProductID, aaDBParameterDirection.Output)
```

Properties

You can use the following properties with the **aaDBCommand** object.

CommandTimeout

This property accesses the underlying **CommandTimeout** property of the DCM command object. The DCM command object in turn, accesses the **ADO.NetDbCommand.CommandTimeout** property. According to MSDN, this **CommandTimeout** property gets or sets the wait time before terminating the attempt to run a command object and generating an error.

Notes:

- If the DCM command object has been removed or is otherwise invalid, reading this property returns **0**; writing it has no effect. An exception is not thrown.
- When aaDBConnectionType=Oracle, this property is not supported. Setting a command object timeout has no effect, and the value returned is always **0.**

CurrentRowNumber

This read-only property returns the row value most recently set by the last call to **SelectRow()** or the new row number that has been created if **AddRow()** has been called since **SelectRow()**.

This property returns a value of **-1** if any of the following conditions occurred:

- A call to **SelectRow()** was not made.
- An invalid row number was passed in the last call to **SelectRow()**.
- **DeleteCurrentRow()** was called.

Syntax

```
long CurrentRowNumber;
```

CurrentTableNumber

This read-only property returns the table index most recently set by the last call to **SelectTable()**.

This property returns 0 by default immediately after a command object (returning a dataset that is not empty) is run successfully.

This property returns -1 if any of the following conditions occurred:

- An invalid index was passed in the last call to **SelectTable()**
- The resulting dataset is empty

Syntax

```
long CurrentTableNumber;
```

Disposed

This read-only property returns a Boolean value that indicates whether the command object has been disposed. If disposed, the command object can no longer be used.

ExecutionState

This read-only property returns the execution state of a SQL command. Because command processing can be asynchronous, you must determine whether the command has been processed before you request the execution state.

Syntax

```
aaDBCommandState.ExecutionState
```

LastExecutionError

This property returns the last error, if any, that is the error string generated by the provider.

Use this property in conjunction with the ExecutionState property.

The **LasExecutionError** property returns the error generated by the provider for the failed command object.

Syntax

```
string LastExecutionError
```

RowCount

Use this read-only property to obtain the number of records in the memory **dataset** that was retrieved by a SQL command. When it is used with a SQL action command such as **INSERT**, **DELETE**, or **UPDATE**, the property returns the number of records affected.

This property returns a value of zero in the following cases:

- The SQL command has not been queued.
- The SQL command has been queued but is not completed.
- The SQL command has completed with an error.

Syntax

```
int RowCount;
```

Public Enumerations

The following public enumerations apply to the **aaDBCommand** object.

aaDBCommandState

This public enumeration indicates the state of an **aaDBCommand** object.

Created

This value indicates that the command has been created. However, the command has not yet run, and you can assign parameters to the command.

Queued

This value indicates that the command is queued for execution. This state occurs immediately after the call to **aaDBCommand.ExecuteAsync() or aaDBCommand.ExecuteSync()**. **Queued** is a transitional state that changes to either **Failed** or **Completed**.

Failed

This value indicates that the command failed.

Completed

This value indicates that the command completed successfully.

Canceled

This value indicates that the command has been canceled.

Disposed

This value indicates that **Dispose()** has been called for the command object. **Disposed** is automatically set when **aaDBAccess.RemoveCommand()** is called, or when **Dispose()** is called on the connection object or transaction object that owns the command.

aaDBCommandType

This public enumeration describes how the command text for an **aaDBCommand** object is used.

Note: When you use the aaDBCommandType enumeration, be aware that not all requests work with all data providers. For example, Microsoft Access does not support stored procedures. The InputOutput direction is not supported by all data providers. Always check the documentation for the data providers about supported options.

sqlStatement

Indicates that the command text is a valid SQL statement.

storedProcedure

Indicates that the command text is a stored procedure name.

aaDBParameterDirection

This public enumeration indicates the direction of parameters used for SQL statements.

Note: When you use the **aaDBParameterDirection** enumeration, be aware that not all requests work with all data providers. For example, Microsoft Access does not support stored procedures. The InputOutput direction is not supported by all data providers. Always check the documentation for the data providers about supported options.

Input

This value indicates that the parameter is input only.

InputOutput

This value indicates that the parameter is capable of both input and output.

Output

This value indicates that the parameter is output only.

ReturnValue

This value indicates the presence of a return value from an operations such as a stored procedure, built-in function, or user-defined function.

CHAPTER 4

aaDBConnection Object

About aaDBConnection Object

Use objects of type **aaDBConnection** to create a new **aaDBCommand** object or **aaDBTransaction** object. You can create instances of **aaDBConnection** only through the static methods **aaDBAccess.CreateConnection()** or **aaDBAccess.GetConnection()**.

Creating an instance of the **aaDBConnection** object is not equivalent to creating and maintaining a physical connection to the data source.

The actual physical connection opens on demand when a request is made to run **aaDBCommand** objects or **aaDBTransaction** objects. The physical connections are controlled by the Database Connection Manager (DCM).

The **SQLData** Script Library does not expose any mechanism to allow you to fine-tune its behavior, such as defining how many connections to use.

Connection Pooling

When you request a SQL command or transaction to be run on the **aaDBConnection** object, an actual physical connection is opened on demand and closed when the command or transaction finishes processing.

The SQLData Script Library uses the DCM common component in regard to connection pooling. When an **aaDBConnection** object is first created, no attempt is made to achieve a physical connection. The **ConnectionState** property remains in the **Disconnected** state. The physical connection is attempted only after a command object is run (a stand-alone command object or a command object that is part of a transaction).

After the first command object that requires a physical connection runs, the physical connection remains open for an unspecified period (depending on loading) and closes automatically after a time.

If a script requires confirmation of a successful physical connection, it must issue some benign SQL statement and then check the **ConnectionState** property of the connection.

Note: You must call **Dispose()** on each instance of aaDBConnection.

Methods

You can use the following methods with the **aaDBConnection** object.

CreateCommand()

Use the **CreateCommand()** method to create a new **aaDBCommand** object.

Syntax

```
aaDBCommand CreateCommand(
string CommandText,
aaDBCommandType CommandType,
bool ReturnDataset)
```

Parameters
You can also configure the newly created object by adding parameters and then executing them.
The parameters are identified in the **SQL statement** by the at sign (@) character when **ProviderType** is **SQL**.
The following SQL statement has one parameter, **lastName**:

```
"SELECT * FROM Person.Contact WHERE (LastName > @lastName)"
```

CreateCommand for Oracle

The parameters are identified in the SQL statement by the colon (:) character when **ProviderType** is **Oracle**.

```
"SELECT * FROM Person.Contact WHERE (LastName > :lastName)"
```

CreateCommand for OLEDB

The parameters are identified in the SQL statement by the question mark (?) character and no name when **ProviderType** is **OLEDB**.

```
"SELECT * FROM Person.Contact WHERE (LastName > ?)"
```

In this case, you configure the parameters by index and not by name.

Be aware that you cannot mix configuration parameters by index and by name.

Return Value

If a failure occurs, this method returns **null**.

Getting the **ExecutionState** property of the newly-created command object returns Created.

Parameters

CommandText
One of the following:
SQL Statement
Stored Procedure name

CommandType
Specifies how the SQL statement is handled:
sqlStatement
storedProcedure

ReturnDataset
Boolean, indicates if the command object is to return a **dataset**
If **True,** you want a data table to be returned. For example:

sqlStatement ="SELECT * FROM Person.Contact"
If **False**, you want to modify only the database or get the **Output** or **ReturnValue** parameter. Examples are INSERT, DELETE, and UPDATE SQL statements.

The data that is returned upon successful execution of a query (Table, View or StoredProcedure) is stored in memory. Scalar values are not returned, other than by being part of the returned dataset.

You can then access this data by using the interface provided by the aaDBCommand object.

CreateTransaction()

Use the **CreateTransaction()** method to create a new **aaDBTransaction** object.

Syntax

```
aaDBTransaction CreateTransaction()
```

Return Value

If a failure occurs, this method returns **null**.

Getting the **ExecutionState** property of the newly-created **aaDBTransaction** object returns Created.

Remarks

You add **aaDBCommand** objects to this object that are processed as a whole in the order that they were added.

Dispose()

Use the **Dispose()** method to free all memory resources associated this database connection object.

Syntax

```
void Dispose()
```

Remarks

If commands or transactions are running when **Dispose()** is called, **Dispose()** cancels the command or transaction object.

GetDiagnostics()

Use the **GetDiagnostics()** method to return a set of diagnostic information about all connections in a **dataset**.

Syntax

```
void GetDiagnostics()
```

Remarks

The returned dataset contains multiple tables. The table with index 0 contains the global diagnostics. The rest of the tables in the **dataset** correspond to each **DCMConnection** object. For details about diagnostic properties, see the *SQLData Object Help*.

LogDiagnostics()

Use the **LogDiagnostics()** method to dump a snapshot of all available connection diagnostic information to the logger. For details about diagnostic properties, see the *SQLData Object Help*.

Syntax

```
void LogDiagnostics()
```

ResetDiagnostics

Use the **ResetDiagnostics()** method to reset the current diagnostic values of the connection object.

Syntax

```
void ResetDiagnostics()
```

Properties

You can use the following properties with the **aaDBConnection** object.

ConnectionName

Use this property to enable the log of diagnostics that was generated by calling **aaDBAccess.LogDiagnostics()** to include a meaningful name. Having a meaningful name may be necessary if a debugging effort requires you to distinguish one connection from another. To see the connection name in the logged diagnostics, look for the line that reads as follows:

```
aaDBIntegration ConnectionName, <Name>, , Clilent assigned conection name,
```

The <Name> part of the line is replaced with the name that the script has applied to the ConnectionName property of the aaDBConnection object. If the ConnectionName property has not been assigned a name by the script, it defaults to a name of the form SQLScriptConnection<N>, where <N> is an incrementing integer value.

Syntax

```
String ConnectionName
```

ConnectionState

Use this read-only property to verify that the connection string specified by **GetConnection** successfully establishes the initial connection to the data source.

Note: You cannot use this property to determine the status of an actual physical connection to the data source. No polling mechanism is available to detect a broken connection.

If intermittent network failures occur, **ConnectionState** does not indicate the failure until a command or transaction is run. However, when a command is run, the **ConnectionState** property updates the current connection state.

The return type is an enumerated value of type **aaDBConnectionState**.

Syntax

```
aaDBConnectionState ConnectionState
```

Disposed

This read-only property indicates whether **Disposed()** has been called for a connection object. If the property returns **True**, no other method or property can be called on this instance.

Syntax

```
Bool Disposed
```

LastError

Use this read-only property with the **ExecutionState** property. It returns the last error string generated by the provider.

This property shows a description of errors that were encountered while parsing the connection string. The string comes from exceptions thrown by the Microsoft object **or data provider objects.**

Syntax

```
string LastError
```

Public Enumerations

The following public enumerations apply to the **aaDBConnection** objects.

aaDBConnectionState

This public enumeration determines the state of the **aaDBConnection** object. Each value reflects the current state of the connection in the DCM.

Disconnected

Disconnected is the state of the **aaDBConnection** object immediately after it transitions through Connected or fails to connect. The disconnect timing is controlled by the underlying ADO.Net communication pooling logic. The **Disconnected** state remains active until the DCM is required to create a physical connection in response to a command.

A previous connected state can transition to a disconnected state if a command is run and the physical connection has been lost.

Connecting

This value indicates a transition state. It indicates that the DCM has started a physical connection to a database based on its internal connection pooling logic and the connection is not complete or has not yet failed.

Connected

This value indicates that the DCM has established a physical connection to a database in response to a command.

Created

This value indicates the initial state for a connection after creation until a transaction or command is run on it. After the state changes to something else, the state can never return to **Created**.

Disposed

This value indicates that **Dispose()** has been called for the connection.

aaDBConnectionType

This public enumeration determines the type of connection that an **aaDBConnection** object is being created for.

Sql

This value indicates a connection for a Microsoft SQL Server database.

OleDb

This value indicates a connection using Microsoft OLEDB.

Oracle

This value indicates a connection for an Oracle database server.

CHAPTER 5

aaDBRow Object

This section describes the aaDBRow object and the methods and properties that you can use with it.

aaDBRow Object

Use objects of type **aaDBRow** to access (set or get) a single row from the memory **dataset** generated while an **aaDBCommand object** is running.

An instance of the **aaDBRow** object is returned by **aaDBCommand.**GetRow(). SetCurrentRow() also requires an instance of this type as an input parameter.

You can modify the **aaDBRow** object returned by **GetRow()** and feed it back to **SetCurrentRow()**, or you can construct a new **aaDBRow** object specifically for **SetCurrentRow()**.

The **aaDBRow** object has two public members both of type **ArrayList**:

- **columnName**
- **columnValue**

Note: You can use this type to update a row in the memory table by configuring only a subset of the columns.

The two arrays must have the same size.

Note: You must call **Dispose()** on each instance of aaDBRow.

aaDBRow —Public Constructor

You can use a public constructor in a script to create an **aaDBRow** object from two synchronized **ArrayList** objects for use with **aaDBCommand.SetCurrentRow()**.

To use this constructor, first create the new **ArrayList** objects, and fill them with column names and values. Use positional placement within the two **ArrayList** objects to correspond names with values.

Then create a new **aaDBRow** object with this constructor, passing in the two **ArrayList** objects. For an example, see the parameters table in the aaDBConnection.SetCurrentRow() section.

Syntax

```
aaDBRow(
ArrayList Names,
ArrayList Values)
```

Methods

You can use the following methods with the **aaDBRow** object.

GetColumnName()

Use the **GetColumnName()** method to retrieve the name of the column specified by ColumnNum.

Syntax

```
string GetColumnName(
int ColumnNum)
```

Return Value

If ColumnNum is equal to or greater than the number of columns stored in the row, as returned by the ColumnCount property, null is returned.

GetColumnValue()

Use the **GetColumnValue()** method to retrieve the value of the column specified by ColumnNum.

Syntax

```
object GetColumnValue(
int ColumnNum)
```

Return Value

If ColumnNum is equal to or greater than the number of columns stored in the row, as returned by the ColumnCount property, null is returned.

Note: The value is returned as a generic object and must be cast by the script to the intended type.

Properties

You can use the following properties with the **aaDBRow** object.

ColumnCount

Use this read-only property to retrieve the number of columns stored in the row object.

Syntax

```
int ColumnCount
```

ColumnNames

Use this read-only property to retrieve the ArrayList that internally stores the column names for the row.

Syntax

```
ArrayList ColumnNames
```

ColumnValues

Use this read-only property to retrieve the ArrayList that internally stores the column values for the row.

Syntax

```
ArrayList ColumnValues
```

CHAPTER 6

aaDBTransaction Object

This section explains how to use the **aaDBTransaction** object and the methods and properties associated with it.

aaDBTransaction

Use objects of type **aaDBTransaction** to process multiple **aaDBCommand objects** as a single unit.

Create instances of type **aaDBTransaction** by calling the **CreateTransaction()** method on an instance of the **aaDBConnection object**. For example, assuming that the **aaDBConnection** instance is called **Connection**:

```
Connection.CreateTransaction()
```

If you want to ensure that all SQL commands are run as a whole or not run at all, you must create an instance of the **aaDBTransaction** object. Use the instance to create **aaDBCommand** objects.

When the transaction object is run, all **aaDBCommand** objects that are added to this **aaDBTransaction** object are run in the order that they were added. If the transaction is rolled back, none of the objects are run.

Commands added to a transaction cannot be run as stand-alone command objects but are automatically processed when the transaction runs.

When a script requests the transaction object ID, the transaction object is flagged to be persisted. The SQLData Script Library persists the object across scripts and scan cycles.

You can retrieve an **aaDBTransaction object** at any time by calling the static method **aaDBAccess.GetTransaction ()** and passing the previously acquired string ID.

Note: You must call **Dispose()** on each instance of **aaDBTransaction**.

Methods

You can use the following methods with the **aaDBTransaction** object.

CreateCommand()

Use the **CreateCommand()** method to create a new **aaDBCommand** object. You can configure the new object by adding parameters and then running it.

Syntax

```
aaDBCommand CreateCommand(
string CommandText,
aaDBCommandType CommandType,
bool ReturnDataset)
```

Remarks

For more details, see *Chapter 4, aaDBConnection Object*.

Dispose()

Use the **Dispose()** method to instruct the SQLData Script Library to free all resources allocated for the transaction object and all commands that were added as part of this transaction.

Syntax

```
void Dispose()
```

Remarks

If the transaction is currently running, **Dispose()** automatically cancels the transaction before removing it.

The SQLData Script Library issues **Dispose()** calls for every command in the transaction.

If an ID has been retrieved for this command, you must call Dispose() or aaDBAccess.RemoveCommand().

ExecuteAsync()

Use the **ExecuteAsync()** method to instruct the SQLData Script Library to queue all command objects in the transaction that are queued for later processing. **ExecuteAsync()** returns immediately and processing occurs in the background.

Syntax

```
Result ExecuteAsync()
```

Remarks

You can use the ExecutionState property to check for status.

If any commands complete with an error, the SQLData Script Library issues a transaction rollback to prevent changes to the data source.

After **ExecuteAsync()** is processed, you can still obtain a reference to any commands that were added and analyze their ExecutionState and LastExecutionError properties.

When the transaction object runs, all **aaDBCommand** objects that are added to the **aaDBTransaction** are run in the order that they were added. The following outcomes can occur as the result of processing a transaction:

- **Success**: All commands succeeded. Each command shows an **ExecutionState** of **Completed**. Any dataset for that command is associated with it.
- **Cancellation**: Each command in the transaction shows an **ExecutionState** of **Canceled**. Any data associated with the command is removed.
- **Failure**: When one command in a transaction fails, the command has an **ExecutionState** of **Failed**. All commands preceding it show an **ExecutionState** of **Completed** and retain any datasets that were part of their successful processing.

To free all resources allocated for the transaction object and the commands, you must call **Dispose()**.

ExecuteAsyncCancel()

Use the **ExecuteAsyncCancel()** method to instruct the SQLData Script Library to roll back all commands that are queued as part of a transaction.

Syntax

```
Result ExecuteAsyncCancel()
```

Remarks

You can use the ExecutionState property to check for status.

As a result of successful execution, all command objects created from this transaction are canceled.

ExecuteSync()

Use the **ExecuteSync()** method to instruct the SQLData Script Library to run all commands that are queued as part of a transaction.

Syntax

```
Result ExecuteSync()
```

Remarks

Use the ExecutionState and **LastExecutionError** properties to check for status of this method.

If any one of the commands completes with an error, the SQLData Script Library issues a rollback action to guarantee that the data source does not see any of the changes.

When the transaction object runs, all **aaDBCommand** objects that are added to the **aaDBTransaction** are run in the order that they were added. The following outcomes can occur as the result of processing a transaction:

- **Success**: All commands succeeded. Each command shows an **ExecutionState** of **Completed**. Any dataset for that command is associated with it.
- **Cancellation**: Each command in the transaction shows an **ExecutionState** of **Canceled**. Any data associated with the command is removed.
- **Failure**: When one command in a transaction fails, that command object has an **ExecutionState** of **Failed**. All command objects preceding it show an **ExecutionState** of **Completed** and retain any datasets that were part of their successful processing.

After **ExecuteSync()** is processed, you can still obtain a reference to the command objects that were added and analyze their **ExecutionState** and **LastExecutionError** properties.

Use the **LastExecutionError** property to check if the command failed or succeeded (success is indicated by a blank string), since ExecuteSync() may return 0 in the case of a failure instead of an error message.

To free all resources allocated for the **aaDBTransaction** object and the **aaDBCommand** objects you must call **Dispose()**.

GetID()

Use the **GetID()** method if you want to retrieve the ID of an **aaDBTransaction** object instance to get a reference to this object at a later time in a different script or scan.

The SQLData Script Library generates a unique transaction ID and persists the transaction object in memory.

Syntax

```
string GetID()
```

Return Value

If a failure occurs, the **GetID()** method returns **null**.

Note: The SQLData Script Library returns a string value, but the script engine automatically attempts to cast this value to other types. If the script assigns the returned ID to any other type than string, the ID is corrupted and does not work in future GetTransaction(ID) calls.

Properties

Use the following properties with the **aaDBTransaction** object.

Disposed

This read-only property indicates whether **Disposed()** has been called for a transaction object. If the property returns **True**, no other method or property can be called on this instance.

Syntax

```
Bool Disposed
```

ExecutionState

Use this read-only property to return the state of this transaction object.

Because the processing of the transaction is asynchronous, you must determine if it has finished processing before you request the results.

Syntax

```
aaDBTransactionState ExecutionState
```

FailedCommandID

Use this property to return the ID of the first failed command object during transaction processing.

If the processing action succeeds, **FailedCommandID = 0**.

Syntax

```
string FailedCommandID
```

LastExecutionError

During processing, this property is set to the last error, if any. The error string is the error string generated by the provider.

Use this property in conjunction with the ExecutionState property.

LastExecutionError indicates the error generated by the provider for the first command object in the transaction that failed.

To find the command object that failed to process, check the FailedCommandID property.

LastExecutionError is blank when the transaction runs successfully.

Syntax

```
string LastExecutionError
```

Public Enumeration

The following public enumeration applies to the **aaDBTransaction** object.

aaDBTransactionState

This public enumeration indicates the state of an **aaDBTransaction**.

Created

This value indicates that a transaction has been created. The transaction has not yet run. You can add commands to the transaction.

Queued

This value indicates that the transaction has been queued to run. This state occurs immediately after an **aaDBTRansaction.ExecutedAsync()** or **aaDBTRansaction.ExecutedSync()** call. **Queued** is a transitional state that changes to either **Failed** or **Completed**.

Failed

This value indicates that the transaction has failed.

Completed

This value indicates that the transaction completed successfully.

Canceled

This value indicates that the transaction was canceled before it could complete.

Disposed

This value indicates that the **Dispose()** method has been called for the command object. **Disposed** is automatically set when **aaDBAccess.RemoveTransaction()** is called or when the connection object that owns the transaction is disposed.

APPENDIX A

Error Codes

Various API scripting methods return an error code as a numeric value.

Methods with syntax descriptions that begin with "result" return these numeric values.

This table shows the numeric value, its corresponding error, and meaning of each error code.

Numeric Value	Error Text and Description
-1	**Unknown Failure** This value indicates an exception whose reason could not be determined at run time.
0	**Success** This value indicates a good result; no error occurred.
100	**SoftwareError** This indicates that the internal code returned an unexpected result, but the case was handled.
1000	**StatementNotReady** This error is returned when an **aaDBCommand** object is run, but it is not properly prepared for processing.
1001	**StatementFailed** This error is returned when an **aaDBCommand** object runs and fails to complete properly. This result occurs when the SQL query is malformed.
1002	**DatasetIsNull** This error is returned when the results of an **aaDBCommand** object are being examined or manipulated, but no dataset is associated with the object. This message can mean that the command has not yet run, the SQL query was malformed, or the object was not of a type to return data.
1003	**DatasetIsEmpty** This error occurs when the currently selected row or table number is beyond the end of the data in the dataset that is associated with the **aaDBCommand** object.

Numeric Value	Error Text and Description
1004	InvalidConnection This error is returned when an **aaDBCommand** attempts to run and the connection object it is attached to has not established a good connection to the database.
1005	**InvalidRowNum** This error is returned by **aaDBCommand.SelectRow()** when the row number specified is negative or larger than the number of rows in the dataset that is associated with the **aaDBCommand** object.
1006	**InvalidColValue** This error is returned when a column is examined or manipulated by the index and the index does not represent a valid column number.
1007	**InvalidTableNum** This error is returned by **aaDBCommandSelectTable()** when the table number does not represent a table that is currently stored in the dataset that is associated with the **aaDBCommand** object.
1008	**MissingParameter** This error is returned when an OLEDB style **aaDBCommand** object is run and not all parameter indexes were supplied.
1009	**NamedParametersNotSupported** This error is returned when an attempt is made to call **SetXXXParameterByName()** for a non-OLEBB style **aaDBCommand** object.
1010	**ParameterNameIsRequired** This error is returned when an attempt is made to call **SetXXXParameterByIndex()** for a non-OLEBB style **aaDBCommand** object.
1011	**InvalidRequestOperationInProgress** This error is returned when an attempt is made to run an **aaDBCommand** object that is currently processing. If a command or transaction is currently in progress through **ExecuteAsync()**, that same object cannot be executed again with **ExecuteAsync()** or **ExecuteSync()** until the previous process is complete. Only a single object can be in the processing queue at any time.

Numeric Value	Error Text and Description
1012	**SaveChangesNotSupportedForStoredProcedure** This error is returned when either **SaveChangesSync()** or **SaveChangesAsync()** is called after manipulating the dataset associated with an **aaDBCommand** object whose query was originally against a stored procedure.
1013	**InvalidRequestNotSupportedInCurrentState** This error is returned when the **aaDBCommand** object is not in a state where the requested run or cancel can be honored.
1014	InvalidRequestPartOfTransaction This error indicates that you attempted to run or save changes to an **aaDBCommand** object that was created to be part of an **aaDBTransaction**. This activity is not permitted.
1015	DCMObjectInvalid This error indicates that the DCM cannot provide an object to support **aaDBConnection**, **aaDBCommand**, **aaDBTransaction**.
1016	**ObjectIsDisposed** The script attempted to use an object after calling **Dispose().**