# Chapter 8—SQL Classes and Notation

Omnis has three *SQL classes* that provide the interface to your server database: they are the *schema*, *query*, and *table* class. Schema and query classes map the structure of your server database. They do not contain methods, and you cannot create instances of schema or query classes. You can however use a schema or query class as the definition for an Omnis list using the $definefromsqlclass() method, which lets you process your server data using the SQL methods against your list. When you create a list based on a schema or query class a table instance is created which contains the default *SQL methods*.

*Table* *classes* provide the interface to the data modeled by a schema or query class, and exist primarily to allow you to override the default methods in the table instance. Like schema and query classes, you can use a table class as the definition for an Omnis list and use the same SQL methods against your list.

The SQL list methods and notation are described in this chapter. Creating SQL classes in the Studio Browser is described in the Creating Web & Mobile Apps manual.

## Schema Classes

A *schema class* maps the structure or *data dictionary* of a server table or view within your library. A schema class contains the name of the server table or view, a list of column names and data types, and some additional information about each column. The data types are the equivalent Omnis data types, and the names must conform to the conventions used by the particular server. Schema classes do not contain methods, and you cannot create instances of a schema class. You can define a list based on a schema class using the *Define list from SQL class* command or the $definefromsqlclass() method. You can create a schema class using the New Class>>Schema option in the Studio Browser.

### Schema Class Notation

Each library has a $schemas group containing all the schema classes in the library. A schema class has the type kSchema.

In addition to the standard class properties, such as $moddate and $createdate, a schema class has the following properties

1. **$objs**   
   the group of columns in the schema class
2. **$servertablename**   
   the name of the server table or view to which the schema corresponds

The $objs group containing the columns in the schema class supports the group methods including $first(), $add(), $addafter(), $addbefore(), and $remove(). The $add... methods require the following parameters

1. **Name**   
   the name of the column
2. **Type**constant representing the Omnis data type of the column
3. **Subtype**   
   constant representing the data subtype of the column
4. **Description** (optional)  
   a text string describing the column
5. **Primary-key** (optional)  
   a boolean set to kTrue if this column is a primary key. If omitted it defaults to kFalse
6. **Maximum-Length** (optional)  
   for character and national columns, the maximum length; for other types, Omnis ignores the value of this parameter. If omitted for character and national columns, it defaults to 10000000.
7. **No-nulls** (optional)  
   a boolean set to kTrue if this column cannot have NULL values. If omitted it defaults to kFalse

You can identify a particular column in the $objs group using its column name, order, or ident, a unique number within the scope of the schema class assigned to the column when you add it. A schema column has the following properties (all are assignable except $ident)

1. **$name**   
   the name of the column
2. **$coltype**   
   the Omnis data type of the column
3. **$colsubtype**  
   the Omnis subtype for the data type of the column
4. **$colsublen**  
   the maximum length for Character and National columns
5. **$desc**  
   a text string describing the column
6. **$primarykey**  
   if kTrue the column is a primary key
7. **$nonull**  
   if kTrue the column does not allow null values
8. **$order**  
   the position of the column in the list of columns in the schema class
9. **$ident**  
   a unique number within the scope of the schema class, identifying the column

### List/Row subtypes

A schema class (or a query or table class) can be used as the subtype of a list or row variable, that is, a class, instance, local, task or parameter variable, or a column in a list or row defined from a SQL class.

Schema classes have a property $createinstancewhensubtype that controls whether or not there is a table instance associated with a List or Row variable with a schema class as its subtype. You can set this property in the Property Manager when editing the schema class: it defaults to kTrue for existing and newly created schema classes.

### Make Schema from Server Table

You can also create a schema using the Make schema from server table command. For example

###### Describe server table (Columns) {ServerTable} Make schema from server table {MySchema}

will create the schema MySchema, referencing server table ServerTable, using the current Omnis session.

## Query Classes

Query classes let you combine one or more schema classes or individual columns from one or more schemas, to give you an application view of your server database. A query class contains references to schema classes or individual schema columns. Like schema classes, query classes do not contain methods, and you cannot create instances of a query class. You can define a list based on a query class using the Define list from SQL class command or the $definefromsqlclass() method.

You can create a query class using the New Class>>Query option in the Studio Browser. The Catalog pops up when you open the query class editor, which lets you double-click on schema class or column names to enter them into the query editor. Alternatively, you can drag schema class or column names into the query editor. Furthermore, you can reorder columns by dragging and dropping in the fixed left column of the query editor, and you can drag columns from one query class onto another. You can also drag a column from the schema editor to the query editor.

You can drag from the query list, the schema editor, and the Catalog, and drop onto the extra query text field labeled ‘Text appended to queries’. Dragging a query column from the right-hand list of the catalog query tab inserts a bind variable reference in the form @[$cinst.name].

The column entries have a context menu, which allows you to delete a column, and to open the schema editor for the schema containing the column.

The additional query text edit field has a context menu which allows you to insert text commonly used in SQL queries.

The query class editor does not validate schema class or column names, nor does Omnis automatically update query classes when you edit a schema class. You need to update your SQL classes manually using the Find and Replace tool.

The alias allows you to eliminate duplicate column names when defining a list from the query class. By default, each list column name is the same as the schema column name. You can override this with the alias. If the column name is empty, meaning use all columns in the schema, Omnis inserts the alias at the start of each column name in the schema, to create the list column name; otherwise, Omnis uses a non-empty alias as the list column name.

### Calculated Columns

Query classes can also contain calculated columns. A calculated column is an entry in a query class which has:

1. A schema name, which determines the table to be used in the SQL statement.
2. A column name. This is the calculation. Omnis treats a column name as a calculation if it contains at least one open parenthesis and one close parenthesis. This rule helps to distinguish a calculated column from a badly named schema column. Omnis performs no validation on the calculation, and it simply inserts it into queries generated by $select or $selectdistinct, and into the result of $selectnames.
3. An alias, used as the list column name.

A calculated column is represented, in the list or row variable defined from a SQL class, as a character column of maximum length. If you include strings in the form “<schema name>.” or “<library>.<schema name>.” in the calculation, then Omnis replaces them with “<server table name>.” when it adds the calculation to a query. The “<server table name>” comes from the schema class.

### Query Class Notation

Each library has the group $queries containing all the query classes in the library. A query class has the type kQuery.

A query class has the standard properties of a class together with $extraquerytext, a text string which in some cases Omnis appends to automatically generated SQL, and for example may contain a where clause. The extra query text string can be empty. Before Omnis adds $extraquerytext to a SQL query, it replaces strings in the form “<schema name>. “and “<library>.<schema name>. “ with “<server table name>.”. The “<server table name>” comes from the schema class. This allows you to design query classes which are independent of the table names actually used on the server, since the only place storing the table name is the schema.

A query class has a $objs group containing a list of references to schema columns, or schema classes. $objs supports the same group methods as $objs for the schema class, with the exception of $findname. The $add… methods require the following parameters:

1. **Schema name**  
   the name of the schema, which can be qualified by a library name
2. **Column name** (optional)  
   the name of the column in the schema
3. **Alias** (optional)  
   the alias used to eliminate duplicate list column names

Each query class object has the following properties.

1. **$schema**  
   the name of the schema, which can be qualified by a library name
2. **$colname**   
   the name of the column in the schema; if empty, all columns from the schema class specified in the $schema property are included
3. **$alias**   
   lets you eliminate duplicate column names from a list defined from a query or a table class referencing the query; if $colname is empty, this is a prefix which Omnis inserts at the start of each column name in the schema named in $schema; otherwise, Omnis uses a non-empty alias in the place of the column name
4. **$order**   
   the position of the object in the class
5. **$ident**   
   a unique numeric identifier for the object

A list defined from a query class using the $definefromsqlclass() method has columns which correspond to the objects in the query class. The order of the columns in the list corresponds to the order of the columns in the query class. When an object includes a complete schema, the columns have the order of the columns in the schema class. You can eliminate duplicate list column names using the $alias property.

#### Queries Tab in the Catalog

The Catalog has a queries tab which lists the query classes in the current library. For each query class, the right hand list shows the list column names which would result from defining a list from the query class.

Creating Server Tables from Schema or Query Classes

You can create a table or view in your server database by dragging a schema or query class from your library in the Studio Browser and dropping it onto an open session in the SQL Browser.

**To create a server table or view from a schema or query class**

1. Create the schema or query class in the Studio Browser
2. Define the columns in the schema or query class
3. Use the SQL Browser to open the SQL session for your database
4. Drag the schema or query class from your library and drop it on to your Session

If you drag a schema class onto an open session, Omnis creates a SQL table with the table name defined in your schema class. If you drag a query class, Omnis creates a SQL view with the name of the query class.

## Table Classes

An instance of a table class provides the interface to the data modeled by a schema or query class. You only need to create a table class if you wish to override some of the default processing provided by the built-in table instance methods.

You can create a table class using the New Class>>Table option in the Studio Browser. You can edit the methods for a table class or add your own custom methods in the method editor.

### Table Class Notation

Each library has a $tables group containing all the table classes in your library. A table class has all the basic properties of a class plus $sqlclassname, which holds the name of the schema or query class associated with the table class. To create a table class using a method, you can use the $add() method.

###### Do $clib.$tables.$add(‘MyTable’) Returns TabRef

; returns a reference to the new table

###### Do TabRef.$sqlclassname.$assign(‘AgentSchema’) Returns MyFlag

## Table Instances

You create a table instance in Omnis when you define a list or row variable from a schema, query, or table class, using the Define list from SQL class command, or the $definefromsqlclass() method. Table instances created from schema or query classes have all the default methods of a table instance. Table instances created from a table class have all the default methods of the table class in addition to any custom methods you have added, perhaps to override the default methods.

When you use Define list from SQL class or $definefromsqlclass(), Omnis defines your list to have either one column for each column in the schema class, or one column for each column referenced by the query class (which can contain a subset of columns from a schema class). In the case where you use a table class, Omnis uses the $sqlclassname property of the table class to determine the schema or query from which to define the list. You can pass the query/schema/table class as either an item reference to the class, or as the name of the class, in the form [library.]class, where the library defaults to the current library if omitted.

A list variable defined in this way has all of the methods and properties of a normal list variable, together with all of the methods and properties of the table instance. You never access the table instance directly; you can think of it as being contained in the list variable.

For example, if you want to display a grid containing your data in a SQL form you can use the following code in the $construct() method of the form to create a list based on a schema class

###### # Declare instance variable iv\_SQLData of type List

###### Do iv\_SQLData.$definefromsqlclass('SCHEMACLASSNAME')

###### Do iv\_SQLData.$sessionobject.$assign(iSessionObj)

###### Do iv\_SQLData.$select()

###### Do iv\_SQLData.$fetch(1000) ## Fetch up to 1000 rows

Once you have defined and built your list you can use the table instance methods to manipulate the data. Equally you could declare a row variable, define it from a table, schema or query class, and manipulate your data on a row-by-row basis using many of the same methods.

The Define list from SQL class command and $definefromsqlclass() method both reset the $linemax property of the list to its largest possible value.

If you pass a schema class, or a table class that references a schema class, then the list is defined to have all columns in the schema, unless you pass an explicit list of columns to use from the schema, such as:

###### Do iv\_SQLData.$definefromsqlclass(query / schema / table class [,cCol1,cCol2,...])

### Passing Parameters to a Table instance

You can pass construction parameters to the $construct() method of the table instance by adding a list of parameters after the list of columns in your list definition method, as follows:

###### Do iv\_SQLData.$definefromsqlclass(query/schema/table class [,cCol1,cCol2,...] [,,cons-params])

Note that there is an empty parameter to separate the explicit column list from the cons-params that are passed to $construct for the table instance. Note also that this empty parameter is still required when using a query class or table class that references a query class.

### Adding Columns to a SQL List

You can add columns to a list which has a table instance using the $add() method. For example, the following method defines a list from a query class and adds a column with the specified definition to the right of the list.

###### Do LIST.$definefromsqlclass($clib.$queries.My\_Query) Do LIST.$cols.$add('MyCol',kCharacter,kSimplechar,1000)

Columns added in this way are excluded from the SQL queries generated by the SQL methods described in this section, since they are not defined in the SQL class. You can only add columns to the right of the schema or query related columns in the list.

### Table Instance Notation

Table instances have methods and properties which allow you to invoke SQL queries and related functionality via the list containing the table instance. Some methods apply to list variables only and some to row variables only. Some of these methods execute SQL, which by default executes in the context of the current Omnis session. The methods do not manage transactions; that is your responsibility.

The table instance methods are summarized in this section, with a more detailed description of each method in the next section.

1. **$select()**generates a Select statement and issues it to the server
2. **$selectdistinct()**generates a Select DISTINCT statement and issues it to the server
3. **$fetch()**for a list, fetches the next group of rows from the server, for a row, fetches the next row

The following methods apply to row variables only.

1. **$insert()**inserts a row into the server database
2. **$update()**updates a row (or rows if the where clause applies to several rows) in the server database
3. **$delete()**deletes a row (or rows if the where clause applies to several rows) from the server database

The following methods apply to smart lists only, updating the server database from the list.

1. **$doinserts()**inserts all rows in the list with the row status kRowInserted
2. **$dodeletes()**deletes all rows in the list with the row status kRowDeleted
3. **$doupdates()**updates all rows in the list with the row status kRowUpdated
4. **$dowork()**executes the three $do... methods above, in the order delete, update, insert

When you call $doinserts(), $dodeletes(), $doupdates() or $dowork(), the table instance calls the appropriate method(s) from the following list, to invoke each individual insert, delete or update. This allows you to use table class methods to override the default processing. As a consequence these methods only apply to smart lists.

1. **$doinsert()**  
   inserts a single row with row status kRowInserted
2. **$dodelete()**  
   deletes a single row with row status kRowDeleted
3. **$doupdate()**  
   updates a single row with row status kRowUpdated

The following methods apply to smart lists only, reverting the state of the list, that is, they do not affect the server database.

1. **$undoinserts()**removes any inserted rows from the list
2. **$undodeletes()**restores any deleted rows to the list, and resets their status to kRowUnchanged
3. **$undoupdates()**restores any updated rows to their original value, and resets their status to kRowUnchanged
4. **$undowork()**executes the three $undo... methods above, one after the other, in the order insert, update, delete

You can use the following methods to create text strings suitable for using in SQL statements. You are most likely to use these if you override default table instance methods using a table class.

1. **$selectnames()**returns a comma-separated list of column names in the list or row variable, suitable for inclusion in a SELECT statement
2. **$createname()**returns a comma-separated list of column names, data types, and the NULL or NOT NULL status, for each column in the list or row variable, suitable for inclusion in a CREATE TABLE statement
3. **$updatenames()**returns a text string containing a SET clause, suitable for inclusion in an UPDATE statement
4. **$insertnames()**returns a text string containing a list of columns and values for a row variable, suitable for inclusion in an INSERT statement
5. **$wherenames()**returns a text string containing a Where clause, suitable for inclusion in a SQL statement that requires a constraining clause

You can use the following method in a table class.

1. **$sqlerror()**a means of reporting errors. The table instance default methods call this method when a problem occurs while executing SQL

Table instances have the properties of list or row variables as well as the following.

1. **$sqlclassname**the name of the associated schema or query class used to define the columns of the list; this property is NOT assignable
2. **$useprimarykeys**if true, only those schema columns that have their $primarykey property set to true are used in Where clauses for automatically generated Update and Delete statements. Omnis automatically sets $useprimarykeys to kTrue when defining the list, if and only if there is at least one primary key column in the list
3. **$extraquerytext**a text string appended to automatically generated SQL; used by the $select(), $selectdistinct(), $update(), $delete(), $doupdates() and $dodeletes() methods; for example, it can contain a Where clause. When the table instance is defined either directly or indirectly via a query class, Omnis sets the initial value of this property from the query class; otherwise, this property is initially empty
4. **$servertablenames**a comma-separated list of the names of the server tables or views referenced by the schemas associated with the table instance. If the table instance uses a schema class to define its columns, there is only one name in $servertablenames. If the table instance uses a query class, there can be more than one name, corresponding to the schemas referenced by the query class, and in the order that the schemas are first encountered in the query class
5. **$sessionname**the name of the Omnis session to the server, on which the table instance methods execute their SQL; if empty, Omnis issues the SQL on the current session
6. **$colsinset**the number of columns in the current result set for the session used by the table instance; this property is NOT assignable
7. **$rowsaffected**the number of rows affected by the last call to $insert(), $update(), $delete(), $doinserts(), $doupdates(), or $dodeletes()
8. **$rowsfetched**the number of rows fetched so far, using the $fetch() method, in the current result set for the session used by the table instance
9. **$allrowsfetched**set to kTrue when all rows in the current result set for the current table instance have been fetched, otherwise kFalse at other times

List columns in a list containing a table instance have three table instance related properties: $excludefromupdate, $excludefrominsert and $excludefromwhere.

When $excludefromupdate is true, the column is omitted from the result of $updatenames, and from the list of columns in the SQL statements generated by $update.

When $excludefrominsert is true, the column is omitted from the result of $insertnames, and from the list of columns in the SQL statements generated by $insert.

Note that $excludefromupdate does not cause the column to be omitted from the where clause generated by $update and $updatenames, therefore allowing you to have a column which is purely a key and not updated. If you do want to exclude a column from the where clause, set $excludefromwhere to true. $excludefromwhere affects the where clause generated by $update, $updatenames, $delete and $wherenames.

For example:

###### Do MyList.$cols.MyKey.$excludefromupdate.$assign(kTrue)

The default setting of these properties is kFalse, except for calculated columns, in which case the default is kTrue. However, note that calculated columns are omitted from the where clause, irrespective of the setting of $excludefromwhere.

If you define a list from a SQL class and use $add to add additional columns, you cannot set these properties for the additional columns.

### Table Instance Methods

The following methods use the list variable MyList or row variable MyRow which can be based on a schema, query, or table class.

$select()

###### Do MyList.$select([*parameter-list*]) Returns STATUS

$select() generates a Select statement and issues it to the server. You can optionally pass any number of parameters which Omnis concatenates into one text string. For example, *parameter-list* could be a Where or Order By clause. The method returns kTrue if the table instance successfully issued the Select.

The $select() method executes the SQL statement equivalent to

###### Select [$cinst.$selectnames()] from [$cinst.$servertablenames] [$extraquerytext] [parameter-list]

The following $construct() method for a SQL form defines a row variable and builds a select table. The form contains an instance variable called iv\_SQLData with type Row.

###### Set current session {session-name} Do iv\_SQLData.$definefromsqlclass('schema-name') Do iv\_SQLData.$select()

$selectdistinct()

###### Do MyList.$selectdistinct([parameter-list]) Returns STATUS

$selectdistinct() is identical in every way to $select(), except that it generates a Select Distinct query.

$fetch()

###### Do MyList.$fetch(n[,append]) Returns STATUS

$fetch() fetches up to *n* rows of data from the server into the list, or for row variables fetches the next row. If there are more rows available, a subsequent call to fetch will bring them back, and so on. The $fetch() method returns a constant as follows

|  |  |
| --- | --- |
|  |  |
| kFetchOk | Omnis fetched n rows into the list or row variable |
| kFetchFinished | Omnis fetched fewer than n rows into the variable; this means that there are no more rows to fetch |
| kFetchError | An error occurred during the fetch; in this case, Omnis calls $sqlerror() before returning from $fetch(), and the list contains any rows fetched before the error occurred |
| kFetchMemoryUsageExceeded | Omnis fetched fewer than n rows into the variable; some rows could not be fetched because $maxresultsetsize was exceeded |

When fetching into a list, if the Boolean append parameter is kTrue, Omnis appends the fetched rows to those already in the list; otherwise, if append is kFalse, Omnis clears the list before fetching the rows. If you omit the append parameter, it defaults to kFalse.

The following method implements a Next button on a SQL form using the $fetch() method to fetch the next row of data. The form contains the instance variables iv\_SQLData and iv\_OldRow both with type Row.

###### # declare local variable lv\_Status of Long integer type On evClick   Do iv\_SQLData.$fetch() Returns lv\_Status   If lv\_Status=kFetchFinished|lv\_Status=kFetchError     Do iv\_SQLData.$select()     Do iv\_SQLData.$fetch() Returns lv\_Status   End If   Calculate iv\_OldRow as iv\_SQLData   Do $cwind.$redraw()

$insert()

###### Do MyRow.$insert() Returns STATUS

$insert() inserts the current data held in a row variable into the server database. It returns kTrue if the table instance successfully issued the Insert. The $insert() method executes the SQL statement equivalent to

###### Insert into [$cinst.$servertablenames] [$cinst.$insertnames()]

The following method implements an Insert button on a SQL form using the $insert() method to insert the current value of the row variable. The form contains the instance variable iv\_SQLData with type Row.

###### On evClick   Do iv\_SQLData.$insert() ## inserts the current values

$update()

###### Do MyRow.$update(old\_row[,disable\_where]) Returns STATUS

$update() updates a row in a server table from the current data held in a row variable. It returns kTrue if the table instance successfully issued the Update. Note that if the SQL statement identifies more than one row, each row is updated.

The *old\_row* parameter is a row variable containing the previous value of the row, prior to the update.

The optional *disable\_where* parameter is a boolean which defaults to kFalse when omitted. If you pass kTrue, then Omnis excludes the where clause from the automatically generated SQL. This may be useful if you want to pass your own where clause using $extraquerytext.

The $update() method executes the SQL statement equivalent to

###### Update [$cinst.$servertablenames][$cinst.$updatenames(‘old\_row’)] [$extraquerytext]

The following method implements an Update button on a SQL form using the $update() method. The form contains the instance variables iv\_SQLData and iv\_OldRow both with type Row.

###### On evClick   Do iv\_SQLData.$update(iv\_OldRow)

$delete()

###### Do MyRow.$delete([disable\_where]) Returns STATUS

$delete() deletes a row from a server table, matching that held in the row variable. It returns kTrue if the table instance successfully issued the Delete. Note that if the SQL statement identifies more than one row, each row is deleted. The optional *disable\_where* parameter is a boolean which defaults to kFalse when omitted. If you pass kTrue, then Omnis excludes the where clause from the automatically generated SQL. This may be useful if you want to pass your own where clause using $extraquerytext.

The $delete() method executes the SQL statement equivalent to

###### Delete from [$cinst.$servertablenames] [$cinst.$wherenames()] [$extraquerytext]

Note that [$cinst.$wherenames()] is omitted by setting disable\_where to kTrue.

The following method implements a Delete button on a SQL form using the $delete() method. The form contains the instance variable iv\_SQLData with type Row.

###### On evClick   Do iv\_SQLData.$delete()   Do iv\_SQLData.$clear()   Do $cwind.$redraw()

$doinserts()

###### Do MyList.$doinserts()Returns MyFlag

This method only works for smart lists. $doinserts() inserts rows with status kRowInserted in the history list, into the server table, and returns kTrue if the table instance successfully issued the Inserts. $doinserts() calls $doinsert() once for each row to be inserted. $doinserts() then accepts the changes to the smart list, unless an error occurred when doing one of the Inserts.

$dodeletes()

###### Do MyList.$dodeletes([disable\_where])Returns MyFlag

This method only works for smart lists. $dodeletes() deletes rows with status kRowDeleted in the history list, from the server table, and returns kTrue if the table instance successfully issued the Deletes. $dodeletes() calls $dodelete() once for each row to be deleted. $dodeletes() then accepts the changes to the smart list, unless an error occurred when doing one of the Deletes. The optional *disable\_where* parameter is a boolean which defaults to kFalse when omitted. If you pass kTrue, then Omnis excludes the where clause from the automatically generated SQL. This may be useful if you want to pass your own where clause using $extraquerytext.

$doupdates()

###### Do MyList.$doupdates([disable\_where]) Returns MyFlag

This method only works for smart lists. $doupdates() updates rows with status kRowUpdated in the history list, in the server table, and returns kTrue if the table instance successfully issued the Updates. $doupdates() calls $doupdate() once for each row to be updated. $doupdates() then accepts the changes to the smart list, unless an error occurred when doing one of the Updates. The optional *disable\_where* parameter is a boolean which defaults to kFalse when omitted. If you pass kTrue, then Omnis excludes the where clause from the automatically generated SQL. This may be useful if you want to pass your own where clause using $extraquerytext.

$dowork()

###### Do MyList.$dowork([disable\_where]) Returns MyFlag

This method only works for smart lists. $dowork() is a shorthand way to execute $doupdates(), $dodeletes() and $doinserts(), and returns kTrue if the table instance successfully completed the three operations. The optional *disable\_where* parameter is a boolean which defaults to kFalse when omitted. If you pass kTrue, then Omnis excludes the where clause from the automatically generated SQL for $dodeletes() and $doupdates(). This may be useful if you want to pass your own where clause using $extraquerytext.

$doinsert()

###### $doinsert(row)

$doinsert inserts the row into the server database. The default processing is equivalent to

###### row.$insert()

$dodelete()

###### $dodelete(row)

$dodelete deletes the row from the server database. The default processing is equivalent to

###### row.$delete()

$doupdate()

###### $doupdate(row,old\_row)

$doupdate updates the row in the server database, using the old\_row to locate the row. The default processing is equivalent to

###### row.$update(old\_row)

$undoinserts()

###### Do MyList.$undoinserts() Returns MyFlag

This method only works for smart lists. $undoinserts() undoes the Inserts to the list and returns kTrue if successful. It is equivalent to the smart list method $revertlistinserts().

$undodeletes()

###### Do MyList.$undodeletes() Returns MyFlag

This method only works for smart lists. $ undodeletes() undoes the Deletes from the list and returns kTrue if successful. It is equivalent to the smart list method $revertlistdeletes().

$undoupdates()

###### Do MyList.$undoupdates() Returns MyFlag

This method only works for smart lists. $undoupdates() undoes the Updates to the list and returns kTrue if successful. It is equivalent to the smart list method $revertlistupdates().

$undowork()

###### Do MyList.$undowork() Returns MyFlag

This method only works for smart lists. $undowork() undoes the changes to the list and returns kTrue if successful. It is equivalent to the smart list method $revertlistwork().

$sqlerror()

###### Do MyList.$sqlerror(ERROR\_TYPE, ERROR\_CODE, ERROR\_TEXT)

Omnis calls $sqlerror() when an error occurs while a default table instance method is executing SQL. The default $sqlerror() method performs no processing, but you can override it to provide your own SQL error handling. It passes the parameters:

|  |  |
| --- | --- |
|  |  |
| ERROR\_TYPE | indicates the operation where the error occurred: kTableGeneralError, kTableSelectError, kTableFetchError, kTableUpdateError, kTableDeleteError or kTableInsertError. |
| ERROR\_CODE | contains the SQL error code, as returned by *sys(131).* |
| ERROR\_TEXT | contains the SQL error text, as returned by *sys(132).* |

$selectnames()

###### Do MyList.$selectnames() Returns SELECTTEXT

Returns a text string containing a comma-separated list of column names in the list variable in the format:

###### TABLE.col1,TABLE.col2,TABLE.col3,...,TABLE.colN

The returned column names are the server column names of the list columns in the order that they appear in the list, suitable for inclusion in a SELECT statement; also works for row variables. Each column name is qualified with the name of the server table.

#### $createnames()

###### Do MyList.$createnames() Returns CREATETEXT

Returns a text string containing a comma-separated list of server column names and data types for each column in the list variable, suitable for inclusion in a CREATE TABLE statement; also works for row variables. The returned string is in the format:

###### col1 COLTYPE NULL/NOT NULL,col2 COLTYPE NULL/NOT NULL, col3 COLTYPE NULL/NOT NULL,...,colN COLTYPE NULL/NOT NULL

The NULL or NOT NULL status of each column is derived from the $nonull property in the underlying schema class defining the column.

$updatenames()

###### Do MyRow.$updatenames() Returns UPDATETEXT

Returns a text string in the format:

###### SET TABLE.col1=@[$cinst.col1],TABLE.col2=@[$cinst.col2], TABLE.col3=@[$cinst.col3],...,TABLE.colN=@[$cinst.colN]

where col1…coln are the server column names of the columns in the row variable. Each column name is qualified with the name of the server table.

###### Do MyRow.$updatenames([old\_name]) Returns UPDATETEXT

The optional parameter *old\_name* is the name of a row variable to be used to generate a ‘where’ clause. If you include *old\_name*, a ‘where’ clause is concatenated to the returned string in the following format:

###### WHERE col1=@[old\_name.col1] AND ... AND colN=@[old\_name.colN]

The columns in the where clause depend on the setting of $useprimarykeys. If $useprimarykeys is kTrue, then the columns in the where clause are those columns marked as primary keys in their schema class. Otherwise, the columns in the where clause are all non-calculated columns except those with data type picture, list, row, binary or object.

You can replace $cinst in the returned string using:

###### Do MyRow.$updatenames([old\_name][,row\_name]) Returns UPDATETEXT

where *row\_name* is the name of row variable which Omnis uses in the bind variables. This may be useful if you override $doupdate() for a smart list.

$insertnames()

###### Do MyRow.$insertnames() Returns INSERTTEXT

Returns a text string with the format:

###### (TABLE.col1,TABLE.col2,TABLE.col3,...,TABLE.colN) VALUES (@[$cinst.col1],@[$cinst.col2],@[$cinst.col3],...,@[$cinst.colN])

where col1...colN are the server column names of the columns in the row variable. The initial column names in parentheses are qualified with the server table name. You can replace $cinst in the returned string using:

###### Do MyRow.$insertnames([row\_name]) Returns INSERTTEXT

where *row\_name* is the name of row variable which Omnis uses in the bind variables. This may be useful if you override $doinsert() for a smart list.

$wherenames()

###### Do MyRow.$wherenames() Returns WHERETEXT

Returns a text string containing a Where clause in the format:

###### WHERE TABLE.col1=@[$cinst.col1] AND TABLE.col2=@[$cinst.col2] AND TABLE.col3=@[$cinst.col3] AND ... TABLE.colN=@[$cinst.colN]

where col1...colN are the server column names of the columns in the row variable. Each column name is qualified with the server table name.

The columns in the where clause depend on the setting of $useprimarykeys. If True, then the columns in the where clause are those columns marked as primary keys in their schema class. Otherwise, the columns in the where clause are all non-calculated columns except those with data type picture, list, row, binary or object.

The = operator in the returned string is the default, but you can replace it with other comparisons, such as < or >=, by passing them in the *operator* parameter.

###### Do MyRow.$wherenames([operator]) Returns WHERETEXT

You can replace $cinst in the returned string using:

###### Do MyRow.$wherenames([operator][,row\_name]) Returns WHERETEXT

where *row\_name* is the name of row variable which Omnis uses in the bind variables. This may be useful if you override $dodelete() for a smart list.

If you want to see the SQL generated by the table instance SQL methods, you can use the command *Get SQL script* to return the SQL to a character variable after you have executed the SQL method. Note that the returned SQL will contain bind variable references which do not contain $cinst. This is because *Get SQL script* does not execute in the same context as the table instance. However, you will be able to see the SQL generated, which should help you to debug problems.

## SQL Classes and Sessions

A row or list variable defined from a SQL class has the $sessionobject property which is the session object that is used by the table. For a new table instance $sessionobject is initially empty. The $sessionobject may be assigned in the table class $construct method or elsewhere. Here are some examples using a list variable iResultsList and object class odbcobj

###### Do iResultsList.$definefromsqlclass('T\_authors') Do iResultsList.$sessionobject.$assign($objects.odbcobj.$new()) Do iResultsList.$sessionobject.$logon(hostname,username,password)

Or if a session pool is used:

###### Do iResultsList.$definefromsqlclass('T\_authors') Do iResultsList.$sessionobject.$assign($sessionpools.poolone.$new())

Or if the session instance is already set up in an object variable:

###### Do SessObj.$logon(hostname,username,password) Do iResultsList.$definefromsqlclass('T\_authors')

Then the $sessionobject may be assigned using:

###### Do iResultsList.$sessionobject.$assign(SessObj)

In this final case the object instance in SessObj is duplicated so that the $sessionobject is a separate instance. However, both instances continue to refer to the same session. This is a general rule for session instances, when an object instance is duplicated both instances refer to the same underlying session. For example:

###### Calculate SessObj as $sessionpools.poolone.$new() Calculate SessObj2 as SessObj

At this point both variables contain separate instances that refer to the same session and if we now

###### Calculate SessObj as $clib.$classes.odbcobj.$new()

SessObj2 continues to refer to the original session whereas SessObj is now a separate object

###### Calculate SessObj2 as 0

Now no variables refer to the original session that is automatically returned to the pool.

A list defined from an SQL class also has the $statementobject property. This is a read-only property which is the statement object that is being used by $sessionobject. The $statementobject property is intended to be used in methods that are being overridden by a table class ($select for example). Unlike $sessionobject it is not safe to assume $statementobject will remain the same throughout the life of the list.

### Table Class Methods and Sessions

The $sessionobject and $statementobject properties can be used to obtain a session and statement when required so that a table instance may execute SQL. For example

A session pool “poolone” has been created using

###### Calculate lHostName as 'SqlServer' Calculate lUserName as '' Calculate lPassword as '' Do $extobjects.ODBCDAM.$objects.ODBCSESS.$makepool('poolone',5, lHostName,lUserName,lPassword) Returns #F

A list variable is then defined in the $construct method of a window class using

###### Do iResultsList.$definefromsqlclass('T\_authors')

In the table class “T\_authors” $construct method a session instance is obtained from session pool “poolone” and assigned to the $sessionobject property of the list variable using

###### Calculate lRef as $sessionpools.poolone.$new() Do $cinst.$sessionobject.$assign(lRef) Returns #F

Where “lRef” is an item reference variable. In this situation $cinst is a reference to the list variable “iResultsList”. Note that the statement object iResultsList.$statementobject is created automatically and there is no need to use the $newstatement method to create it.

The table class enables the developer to override the default SQL methods, for example a $select method to select all columns in the list variable

###### Begin statement Sta: SELECT [$cinst.$sessionobject.$selectnames($cinst)] FROM [$cinst.$servertablenames] If len(pExtraQueryText)   Sta: [pExtraQueryText] End If End statement Do $cinst.$statementobject.$prepare() Returns #F If flag true   Do $cinst.$statementobject.$execute() Returns #F End If Quit method #F

Where “pExtraQueryText” is a character parameter containing SQL clauses to be appended to the query.

The results of the select may be retrieved using a $fetch method in the table class containing

###### Do $cinst.$statementobject.$fetch($cinst,pNumberOfRows,kTrue) Returns lFetchStatus Quit method lFetchStatus

Where “pNumberOfRows” is an integer parameter containing the number of rows to be fetched.

These methods may then be called from the window $construct method in order to build a list using

###### Do iResultsList.$select() Returns #F Do iResultsList.$fetch(9999) Returns lFetchStatus