# Chapter 6—List Programming

Omnis has two structured data types; the ***list*** and the ***row.*** A list can hold multiple columns and rows of data, with each row having the same column structure, while a row is effectively a single-row list. You can create lists of strings, lists of records, or lists of lists. You can define a list from individual variables, or base a list on one of the Omnis SQL data classes, such as a schema, query, or table class. In this case, the list gets its column definitions from the columns defined in the SQL class.

Each list can hold an unlimited number of lines with up to 32,000 columns, although you should be aware that the limitations on memory may limit the number of rows in lists with many columns. A row can have up to 32,000 columns.

The list is the single most important data type in Omnis programming. Omnis makes use of lists in many different kinds of programming tasks, such as generating reports, handling sets of data from a database server, and importing and exporting data. List variables provide the data (content) and formatting for many of the visual list components available in the JavaScript Client, including List boxes, Data grids, and Charts: the different types of visual list controls are described in the *JavaScript Components* chapter in the *Creating Web & Mobile Apps* manual.

In this chapter, rows are generally treated the same as lists, that is, you can use a row name in any command that takes a list name as a parameter. In addition, references to *SQL lists* in this chapter refer to lists based on either schema, query, or table classes, which are referred to collectively as *SQL classes*.

## Declaring List or Row Variables

You can create various scopes of list and row variables, including task, class, instance, and local variables. You declare a list or row variable in the variable pane of the method editor. The following table summarizes the variable types and their visibility.

|  |  |  |  |
| --- | --- | --- | --- |
| List or row type | When created? | Where visible? | When removed? |
| Task variable | on opening task | within the task and all its classes and instances that belong to the task | on closing task |
| Class variable | on opening the library | within the class and all its instances | on clearing class variables or closing library |
| Instance variable | on opening instance | within the instance only | on closing instance |
| Local variable | on running method | within the method only | when method terminates |
| Parameter variable | on calling the method | within the recipient method | returning to the calling method |

To declare a list or row variable

1. Right-click in the variables pane of the method editor
2. Select **Insert New Variable** from the context menu
3. Enter the variable name
4. Click in the **Type** box and choose **List** or **Row** from the droplist

### Lists in the JavaScript Client

Like all variables you use in Remote forms to be displayed in the JavaScript Client, any list or row variables that you want to use in a remote form should be declared as *instance variables* (or local / parameter as appropriate). Lists can be declared as task variables which are available to all instances in the current remote task instance.

## Defining List or Row Variables

To define a list or row variable you need to specify its columns. You can do this using Omnis commands or the notation. You can define a list or row variable

1. from variables
2. from a schema, query, or table class
3. from a file class

If you want to use a list in a window or remote form (or any class that can be opened), you should define the list in the $construct() method of the class, or call a method from the $construct() that defines the list. This ensures the list is defined and in memory ready to be used in the current instance.

### Defining Lists from Variables

To define a list from a number of variables you can use the *Define list* command, or the $define() method. For example

###### # Declare class variable cvList1 of List type # Declare class variable cvCol1 of Short integer type # Declare class variable cvCol2 of Character type # Declare class variable cvCol3 of type Date Time (Short date 1980..2079) Set current list {cvList1} Define list {cvCol1,cvCol2,cvCol3}

This method will define the list cvList1 with the columns cvCol1, cvCol2, cvCol3. You can define a list with up to 255 columns. The data type of each field or variable defined in the list determines the data type of the corresponding column in the list.

### Defining Lists and Rows from SQL Classes

You can define a list based on one of the SQL classes, that is, a schema, query, or table class, using the *Define list from SQL class* command or $definefromsqlclass() method. This binds the list or row variable to the schema or query class and consequently maps the list’s columns to the server table. When you define a list or row variable from a table class, it must have its **$sqlclassname** property set to the associated schema or query class. You can do this either in the Property Manager or using the notation.

###### Do $clib.$tables.MyTable.$sqlclassname.$assign('MySchema') ## or Do $clib.$tables.MyTable.$sqlclassname.$assign('MyQuery')

The following example defines an instance row variable from a schema class called 'MyPictures':

###### # create iSqlRow of Row type Do iSqlRow.$definefromsqlclass('MyPictures')

The full syntax of the $definefromsqlclass() method is as follows:

$definefromsqlclass(class[,row,parameters])

Where *class* is a schema, query, or table class (name or item reference to it), and the *row* and *parameters* are optional.

The *row* parameter affects the columns used when the SQL class is a schema or table referencing a schema. A row with no columns (or the parameter is omitted) means that the list is defined using all the columns in the schema. Otherwise if the *row* is specified each column in the *row* becomes the name of a column to add to the list definition from the schema. The *parameters* can be a list of parameter values that are passed to $construct() of the table class instance created by the method.

For example:

###### Do list.$definefromsqlclass('schema',row('c1,'c2'))

would only include columns c1 and c2 in the list definition.

###### Do list.$definefromsqlclass('schema')

Would include all the columns in schema.

To include all columns and call $construct with parameters:

###### Do list.$definefromsqlclass('table',row(),1,2,3)

This method passes parameters 1, 2, 3 to $construct and includes all the columns from the schema.

#### SQL table instance methods

When you create a list or row variable based on one of the SQL classes a table instance is created, so the list or row variable contains the standard properties and methods of a table instance. Specifically, if you create a variable based on a table class it contains any custom methods you have added to the table class; these can override the standard table instance methods. The following standard methods are available for lists based on a SQL class.

1. **$select()**issues a select statement to the server
2. **$fetch(n[,append]**)empties the list and fetches the next n rows from the server; for row variables, n is set to one and the fetched row always replaces any existing data; the append switch is for list variables and defaults to kFalse which means the list is cleared by default, otherwise if you pass the append switch as kTrue the fetched rows are added to the end of any existing data in the list variable
3. **$insert**()inserts a row into the server database (row variables only)
4. **$update(old\_row**)updates a row in the server database (row variables only)
5. **$delete**()deletes a row from the server database (row variables only)
6. **$sqlerror()**reports the type, code and text for an error in processing one of the above methods

These methods offer a powerful mechanism for processing or inserting data on your server via your SQL list or row variable. For example, to fetch 30 rows into your list

###### # declare cvList1 of list type Do cvList1.$definefromsqlclass(MySchema) Do cvList1.$select() Returns myFlag ## sends a select If myFlag = 0 ## checks for errors   OK message {SQL error [sys(131)]: [sys(132)]} End If Do MyList.$fetch(30) Returns myFlag ## fetches 30 rows # to fetch another 10 rows and add them to your list Do MyList.$fetch(10,kTrue) Returns myFlag

### Defining Lists using SQL Workers

From Studio 10.1, you can specify that a SQL list or row will use a SQL Worker Object of the same DAM type as the SQL session object to perform SQL list operations asynchronously. See the SQL Worker Lists section in the SQL Programming chapter.

### Defining Lists from File classes

You can define a list based on a file class using the notation list.$define(filename). You can use the notation list.$define("lib.filename") to reference a file class in another library: note that the name must be passed as a quoted string. You can use the switch /s, e.g. "lib.filename/s", where s means skip columns with empty names in the file class.

### List/Row subtypes

A schema, query, or table class name 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.

Omnis uses the subtype class to define the list or row, or in the case of parameters, to identify the expected definition of the list or row, although Omnis does not do anything if the definition does not match.

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. The property defaults to kTrue for existing and newly created schema classes. When using the schema class exclusively with Web Services, it is likely that the table instance will not be required, and in this case turning off $createinstancewhensubtype will therefore improve performance.

### Adding columns

The $addcols() method provides a short-hand way of adding one or more columns to a list or row variable. It has the following parameters:

###### list.$addcols(cName,type,subtype,maxlen,...)

which can be used to add one or more columns to a list or row variable, so the parameter count must always be a multiple of four. Each new column must be specified with the following four parameters:

1. **cName**  
   the name of the new column
2. **data type**   
   the Omnis data type represented by one of the type constants, such as kCharacter; all data types are allowed except the Object data type (kObject), since lists of objects are not recommended (you should use object references)
3. **subtype**   
   the subtype of the new column; only applies to some major types
4. **maxlen**  
   for some major types such as Character you can specify the maximum length

## Building List Variables

You can build lists

1. from SQL data
2. from Omnis data

### Building a List from SQL Data

The SQL SELECT statement defines a select table, which is a set of rows on the server that you can read into Omnis in three ways:

1. $fetch(n[,append]) table instance method  
   brings n rows from the select table into a list defined from a SQL class

To transfer rows:

###### Do mylist.$definefromsqlclass(SchemaRef,Lname,Town) ## define list Do MyList.$select() Returns myFlag ## make select table Do MyList.$fetch(10) Returns myFlag ## fetch 10 rows into list or Do MyRow.$fetch() Returns myFlag ## fetch a row into a row var

You should avoid loading a large number of rows into the list, since this may interrupt the interface in your app; you could consider fetching a small batch before refreshing the screen. You can retrieve the rows in batches using the $linemax property which limits the size of the list, pausing after each batch to redraw the list field.

### Viewing the contents of a list variable

You can view the current contents of a list variable by Right-clicking on the variable name in the Method Editor and selecting the first option in the context menu; the first option is “Variable <var\_name>” and will open a table containing the list contents. You can do this wherever the variable name appears in Omnis, including the method editor and Catalog. In order to view the contents of a list *instance variable* the class containing the variable must be open or instantiated, e.g. a remote form must be open in the client browser to view its instance variables.

### List Variable Values

The second option in the List context menu, shown by right-clicking on a list variable, shows the **Value** for the variable, which for a list variable includes information about the number of lines, which lines are selected, as follows:

1. The **Value** context menu option on a list variable previously showed “Value (Not Empty)” when the list contained lines. The option now tells you the number of lines in the list, the line number of the current line held in $line, and the line numbers of up to the first 5 selected lines (with an ellipsis if necessary),   
   e.g. Value (10 lines, $line=4, $selected=1,4,8)
2. When you select **Value,** the text written to the trace log includes the line number of the current line held in $line, and the line number(s) of all of the selected lines, up to the log entry limit of 255 characters (with an ellipsis at the end if necessary).
3. The field value dialog has a new option "Open Lists At Current Line" which defaults to true (the state is saved with the window setup): when true, the grid opens so that the current line is visible.
4. In addition, the Goto line command, on the context menu for the line numbers, sets the default line in the popup it opens to the current line.

## List and Row functions

Omnis provides functions for converting independent variables into a row, and for converting a series of row variables into a list.

### The *list()* Function

The list() function accepts a set of row variables as parameters, and creates a list variable from them. The definition for the first row variable is used to define the list. If subsequent row variables have different definitions, Omnis will convert the data types to match the first row.

###### Calculate myList as list(myRow1, myRow2, myRow3)

### The *row()* Function

The row() function accepts a set of variables as parameters, and creates a row variable from them. The variable types are used to define the columns of the row.

###### Calculate myRow as row(myVar1, myVar2, myVar3)

## Accessing List Columns and Rows

You can access data in a list by loading an entire row of data, or an individual cell into other variables. An entire row of information is loaded with the *Load from list* command. You can access individual cells using the *Load from list* command or the *lst()* function, or by referencing the List Row and Column as part of a calculation. Don’t confuse *lst()* with the *list()* function discussed in the previous section.

The *Load from list* command takes an optional set of variables to load data into. In the case of a list defined from a File class or other variables, the load command will automatically place each column’s data into the fields of the same name. To load a list defined from a SQL class, you include a list of variables as part of the *Load from list* command.

###### Load from list ## loads the row into a file class Load from list (Var1, Var2) ## loads the row into specified vars

You can use the *lst()* function as part of a calculation to extract a particular cell of information from a list.

###### Calculate MyVar as lst(MyList, rowNumber, ColumnName)

You can address cells directly by referring to them as ListVarName.ColumnName for the current row or ListVarName.RowNumber.ColumnName for a specified row. Omnis also recognizes the syntax ListName(‘ColumnName’,RowNumber). The column name must be in quotes.

You can use RowVarName.ColumnName or RowVarName.ColumnNumber when you assign a row variable to a window edit field. Remember that your list and row variables should be defined in the $construct() of a form or window so they are available to edit fields and other data bound objects when the form or window opens.

Since ListName.ColumnName and ListName.RowNumber could be ambiguous, Omnis assumes character values are column names. In the case of the row number being contained by a character variable, this should be indicated by adding ‘+0’.

###### Calculate MyNum as MyList.Amount ## the current row Calculate MyNum as MyList.5.Amount ## row 5 Calculate MyNum as MyList(‘Amount’,5) ## Amount column, row 5

The two types of statement above are also used to assign a value to a list element.

###### Calculate MyList.5.Amount as 100 ## sets Amount column, row 5 to 100

## List Variable Notation

List variables have certain standard properties and methods that provide information about the list, such as how many rows or columns it has, or the number of the current line. List columns, rows, and cells have properties and methods of their own which are listed in the *Omnis Help* (press F1 to open the Omnis Help).

### List Properties and Methods

All types of list have the following properties. A list created from a SQL class has the standard properties and methods of a table instance, together with these list properties.

1. **$linecount**returns the number of lines in the list; you can change this property or use *Set final line number* to truncate the list
2. **$linemax**holds the maximum number of lines in the list; this is set to 10,000,000 by default but you can change it to restrict the list size
3. **$line**holds the current line in the list; this changes when the user clicks on a list line, or when using a method such as $search()
4. **$colcount**returns the number of columns in the list
5. **$isfixed**true if the list has fixed length columns; changing $isfixed clears the data and the class for the list, but keeps the column definitions (note that a list defined using $define() has columns of any length). Fixed length columns improve performance in some cases, but cannot contain all data types
6. **$class**returns the schema, query, or table class for the list, or is empty if it is not based on a SQL class
7. **$cols**groupcontaining the columns in the list; you can use $add() to add a column, also $addbefore() and $addafter() to add a column before or after the specified column (these methods do not work with schema or table based lists)
8. **$smartlist**  
   Set this property to kTrue to make it a "smart list"; setting $smartlist to kTrue creates and initializes the history list which tracks changes to the list; setting $smartlist to kFalse discards the history list completely. If you define or redefine a list using any mechanism, or add columns to a list, its $smartlist property is set to kFalse automatically. See later in this chapter for more details about smart lists.

For a row variable, $linecount, $linemax and $line are all set to 1 and cannot be changed.

Lists also have the following methods.

1. **$define**()  
   without parameters this clears the list definition, otherwise $define(var1[, var2, var3]...) defines a list using variables or file class fields; the variable names (or column names from a file class) and var/column types are used to the define the list column names and types; when using a file class name you can append /s to the file class name to skip empty columns
2. **$definefromsqlclass**()  
   $definefromsqlclass(query/schema/table class[,cCol1,cCol2,...][,,cons-params]) defines a list or row variable from a query, schema or table class and instantiates a table instance. Passes cons-params to the table $construct() method
3. **$copydefinition**()  
   $copydefinition(list or row variable[,parm1,parm2]...) clears the list and copies the definition but not the data from another list or row variable; if the list being copied from is derived from a SQL class, the parameters are passed to $construct() of the table instance
4. **$addcols(cName,type,subtype,maxlen,...)**  
   adds one or more columns to a list or row variable, so the parameter count must always be a multiple of four; cName is the name of the new column, data type is the Omnis data type, such as kCharacter, excluding kObject so you should use object references, subtype and maxlen apply only to certain major types
5. **$clear**()  
   clears the data for the list, but keeps the list definition
6. **$first**()  
   $first([bSelOnly=kFalse, bBackwards=kFalse, condition]) sets $line to first line matching parameters; returns an item reference to the row. If bSelOnly, matches selected lines only; if bBackwards, matches lines in reverse; if condition is present lines must match it
7. **$next**()  
   $next(rRow|iRowNumber [,bSelectedOnly=kFalse, bBackwards=kFalse, condition]) sets $line to the next line after the line identified by the first argument. If iRowNumber is zero, processing starts at $line. See $first for definitions of the other parameters
8. **$add**()  
   $add(column1 value[, column2 value]...) inserts a row of values at the end of the list and returns a reference to the new line
9. **$addbefore**()  
   $addbefore(list row or row number,col1 value[, col2 value]...) inserts a row before the specified row
10. **$addafter**()  
    $addafter(list row or row number,col1 value[, col2 value]...) adds a row after the specified row (does not work with schema or table based lists)
11. **$remove()**   
    $remove(list row or row number) deletes the specified row
12. **$search**()  
    $search(calculation[,bFromStart=kTrue,bOnlySelected=kFalse,bSelectMatches=kTrue,bDeselectNonMatches=kTrue,bSetCurrentLine=kFalse]) searches the list; behaves the same as for the *Search list* command; bOnlySelected restricts the search to selected lines. If bFromStart is kTrue, Omnis searches all of the lines in the list, starting at line 1; otherwise, Omnis starts the search at line ($line + 1).
13. **$sort**()  
    $sort(first sort variable or calculation, bDescending [, second sort variable or calculation, bDescending]...) sorts the list; you can specify up to 9 sort fields, including the sort order flag bDescending. The sort fields or calculations can use $ref.colname or list\_name.colname to refer to a list column. The sort order flag bDescending defaults to kFalse (that is, the sort is normally ascending). For calculated sorts, the calculation is evaluated for line 1 of the list to determine the comparison type (Character, Number or Date).
14. **$removeduplicates**()  
    $removeduplicates(listname.column) removes all list lines with duplicate values in the column; you must sort the list before using this method
15. **$selectduplicates**()  
    $selectduplicates(listname.column) selects all list lines with duplicate values in the column; you must sort the list before using this method; the list selection state of non-duplicate lines is cleared; this can be used in client-executed remote form methods, as well as in server methods
16. **$merge**()  
    $merge(list or row[, by name, only selected]) merges the two lists; note $merge() cannot use search criteria to merger data
17. **$totc()**  
    $totc(expression[,bSelectedOnly=kFalse]) totals the expression over all of the lines in the list; if bSelectedOnly is kTrue, only the selected lines are totaled. It is similar to the totc() function, except it also works when the list does not have proper field columns, for example when the list is defined using a SQL class. For example:

###### Do MyList.$definefromsqlclass('MySchema') ## the schema has 2 numeric cols, col1 and col2 Do MyList.$add(1.1,2.1) Do MyList.$add(3.1,4.1) Do MyList.$add(2.2,1.1) Do MyList.$totc(MyList.col1+MyList.col2) Returns Total # outputs Total = 13.7 i.e the total of both columns

### Properties and Methods of a List Column

The columns of a list are contained in the List.$cols group. The $cols group has the following methods, that is, the standard group methods, including the $add… methods that allow you to add columns to the list (but not schema or table based lists):

1. **$add**  
   $add({fieldname|cName,type,sub-type[,iMaxlen=10000000]}) adds a column to the list and returns an item ref to it; either use just a fieldname (to use the definition of a field) or a name,type and subtype constants (e.g. kCharacter,kSimplechar) and length
2. **$addafter()**  
   $addafter(rColumn|iColumnNumber,{fieldname|cName[,type,sub-type,iMaxlen]}) adds a column to the list and returns an item reference to it
3. **$addbefore()**  
   $addbefore(rColumn|iColumnNumber,{fieldname|cName[,type,sub-type,iMaxlen]}) adds a column to the list and returns an item reference to it
4. **$remove()**  
   $remove(rColumn|iColumnNumber) removes the column from the list; you cannot remove a column from a list defined from a SQL class

A list column has the following properties:

1. **$name**  
   returns the simple name of the column
2. **$dataname**  
   returns the dataname of the list column; empty for a list defined from a SQL class
3. **$coltype**  
   returns the data type of the column; changing this clears the list data
4. **$colsubtype**  
   returns the data subtype of the column; changing this clears the list data
5. **$colsublen**  
   returns the length of character and national columns; changing this clears the list

List columns have the following methods:

1. **$clear**()  
   Clears the data for a list or row, or a column in a list or row; executing List.$clear() for a smart list sets $smartlist to kFalse, meaning that it is no longer a smart list
2. **$average**()  
   $average([bSelectedLinesOnly=kFalse]) Returns the average of the non-null list column values
3. **$minimum**()  
   $minimum([bSelectedLinesOnly=kFalse]) Returns the minimum of the non-null list column values
4. **$maximum**()  
   $maximum([bSelectedLinesOnly=kFalse]) Returns the maximum of the non-null list column values
5. **$count**()  
   $count([bSelectedLinesOnly=kFalse]) The count of non-null values in the list column
6. **$removeduplicates()**  
   $removeduplicates(bSortNow,bIgnoreCase) removes rows with duplicate values in the column
7. **$total**()  
   $total([bSelectedLinesOnly=kFalse]) Returns the total of the non-null list column values

Note: $count, $total, $average, $minimum and $maximum can be used in client executed methods in the JavaScript client. The bSortNow parameter of $removeduplicates is ignored and always treated as kFalse in client methods.

### Properties and Methods of a List Row

A list row has the following properties:

1. **$group**  
   returns the list containing the row
2. **$selected**  
   returns true if the row is selected

A list row has the following methods:

1. **clear**()  
   clears the value of all the columns in the row
2. **$loadcols**()  
   $loadcols(variable1[, variable2]...) loads the column values for the row into the specified variables
3. **$assigncols**()  
   $assigncols(column1 value[, column2 value]...) replaces the column values for the row with the specified values
4. **$assignrow**()  
   $assignrow(row, by name) assigns the column values from the specified row into the list row on a column by column basis

### Properties of a List Cell

If a list cell is itself a list or row variable it has all properties of a list or row. List cells have the following properties.

1. **$group**  
   returns the list row containing the list cell
2. **$ident**  
   returns the column number for the list cell
3. **$name**   
   returns the column name for the list cell
4. **$line**  
   returns the row number for the list cell; not necessarily the current line in the list

## Manipulating Lists

You can change both the structure and data of a list variable using both commands and notation.

### Dynamic List Redefinition

You can add, insert, remove, or move columns in list or row variables without losing the contents of the list or row. This functionality applies to all types of list and row variables including smart lists.

1. List.$cols.$add(variable name)  
   adds a column to the right-hand end of the list using the specified variable name and type as its definition
2. List.$cols.$add(colname, type, subtype, length)  
   adds a column to the right-hand end of the list using the specified definition
3. List.$cols.$remove(column name or number)  
   removes the specified column and moves any remaining columns to the left; you cannot remove a column from a list that has been define from a SQL class, or remove a column that has been added to a list that was defined from a SQL class
4. List.$cols.$addbefore(rColumn|iColumnNumber, {fieldname|cName [,type, sub-type, iMaxlen]})  
   inserts a column to the left of the specified column using the specified variable name and type as its definition (unless type, sub-type, iMaxlen are specified), and moves any columns to the right as necessary
5. List.$cols.$addafter(rColumn|iColumnNumber, {fieldname|cName [,type, sub-type, iMaxlen]})  
   inserts a column to the right of the specified column using the specified variable name and type as its definition (unless type, sub-type, iMaxlen are specified), and moves any columns to the right as necessary
6. List.$cols.column name or number.$ident.$assign(new column number)  
   moves the column to a new position and moves other columns to the right or left as appropriate; in this case the $ident of a list column is its column number, therefore changing the ident moves the column to a different position

When using List.$cols.$add(colname, type, subtype, length) to add a column, the type and subtype parameters need to be constants under Data Types and Data Subtypes in the Catalog (press F9). In addition, the subtype and length are not always required, depending on the type of the column. The following method defines a list and then adds a further two columns to the right of the existing columns.

###### Do mylist.$define(col1,col2) Do mylist.$cols.$add('MyCol',kCharacter,kSimplechar,35) Do mylist.$cols.$add('MyPicture',kPicture)

Note you cannot add a column to a list using square bracket notation or using the fld() function. In addition, you cannot insert, remove, or move columns in a list defined from a SQL class, since you cannot redefine schema-, query-, or table-based lists. However you can use List.$cols.$add() to add extra columns to a SQL list.

### Clearing List Data

You can use the command *Clear list* or ListName.$clear() to clear the data from a list. You can clear individual columns of a list with the ListName.ColumnName.$clear(), and individual rows with ListName.rowNumber.$clear().

### Searching Lists

You can search a list using the $search() method, and a successful search sets the flag. You can use a search calculation to search a list as follows:

###### Do MyList.$search(*calculation* [,bFromStart=kTrue, bOnlySelected=kFalse, bSelectMatches=kTrue, bDeselectNonMatches=kTrue, bSetCurrentLine=kFalse])

For example, to search the Country column for “USA” you can use:

###### Do MaiList.$search(Country = ‘USA’) Returns myFlag

The search *calculation* can use list\_name.colname to refer to a list column. When searching a list column in a client method in the JavaScript Client you must prefix the column name with $ref. For example:

###### Do iList.$search($ref.iCol="ABC")

With bSelectMatches or bDeselectNonMatches the first line number whose selection state is changed is returned (or 0 if no selection states are changed), otherwise the first line number which matches the selection is returned (or 0 if no line is found).

When bSetCurrentLine is kTrue, the list’s current line is also set to the first selected row.

### Selecting List Lines

When you display the data in a list variable in a list field on a window, by default you can select a single line only. However, you can allow multiple selected lines by setting the list or grid field’s $multipleselectproperty. When the user highlights list lines with the mouse, the $selected property for those lines is set. If the field does not have $multipleselectset, the current, selected line is the highlighted one; if the $multipleselect property is set, all highlighted lines are selected, and the current line is the one with the focus.

Some of the commands that operate on a list variable use $selected to indicate their result. For example, *Search list (Select matches)* will set $selected for each line that matches the search criteria.

Each list variable has two select states, the *saved* and *current* selections. The current selection is the set of lines currently selected, whereas the saved selection is the previous set of lines that was selected before the current selection changed.

There are a number of commands that you can use to manipulate selected lines, save the current selection, and swap between the selected and saved states. These commands are described in the *Omnis Studio Help.*

### Merging Lists

You can copy lines from one list to another using the Merge list command or the $merge() method. Merging copies a specified set of lines from one list, and appends them to another . The following example copies the selected lines from LIST1 to LIST2 by checking each line’s $selected property.

###### Set current list LIST2 Set search as calculation {#LSEL} Merge list LIST1 (Use search)

$merge() provides slightly different capabilities in that it can match the destination columns by column name as well as by column number. *Merge list* works by column number only. The syntax is

###### $merge(listName, byColumnName, selectedOnly)

The above example could be written as:

###### Do List2.$merge(List1, kFalse, kTrue)

Note that $merge() does not have a search capability.

### Sorting Lists

You can specify up to nine levels of sorting using the *Sort list* command or $sort() method. To use *Sort list* you need to set up the sort fields first, and clear any existing sort levels since these are cumulative. $sort() clears existing sort fields automatically. For example

###### Set current list {MyList} Clear sort fields Set sort field Country Set sort field Town Set sort field Name Sort list Redraw lists

The $sort() method takes the sort variables or column names in order, each followed by a boolean indicating the sort direction; the sort order flag bDescending defaults to kFalse (that is, the sort is normally ascending).. Using notation, the equivalent of the above example would be

###### # Country, Town, Name are columns in MyList Do MyList.$sort($ref.Country,kFalse, $ref.Town,kFalse, $ref.Name,kFalse) Redraw lists

### Removing Duplicate Values

List columns have the $removeduplicates() method which removes lines with duplicate values in the column. You must sort the list on the column before using this method.

###### Do MaiList.$sort($ref.CustNum,kFalse) ## sorts list on CustNum column Do MaiList.$cols.CustNum.$removeduplicates() Returns NumRemoved

## Smart Lists

You can track changes made to a list by enabling its **$smartlist** property. A smart list saves any changes, such as deleting or inserting rows, in a parallel list called the history list. Smart lists can be filtered, a process which allows data not meeting a particular criteria to be made invisible to the user while being maintained in the history list.

A smart list variable therefore contains two lists:

1. the *normal* list containing the list data, and
2. the *history* list containing the change tracking and filtering information

If you store a smart list as a binary object is a SQL database, all the smart list information is stored automatically.

### Smart Lists and the JavaScript Client

The JavaScript Client does not support smart lists in client executed methods, insofar as if you change the list in some way on the client, it will no longer be a smart list when the updated data is sent from the client back to the server.

### Enabling Smart List Behavior

To enable the smart list capability of any list variable you have to set its $smartlist property to kTrue.

###### Do ListName.$smartlist.$assign(kTrue) ## to enable it

Setting $smartlist to kTrue creates and initializes the history list. If it is already kTrue, then setting it again has no effect.

Setting $smartlist to kFalse discards the history list completely. The current normal list remains unchanged, so the current contents of the normal list are preserved, but all history and filtering information is lost.

If you define or redefine a list using any mechanism, or add columns to a list, its $smartlist property is set to kFalse automatically.

### The History List

The history list has one row for each row in the normal list, together with a row for each row that has been deleted or filtered. The history list has the columns contained in the normal list as well as the following additional columns:

1. **$status**contains the row status, which is one of the constants kRowUnchanged, kRowDeleted, kRowUpdated, or kRowInserted, reflecting what has happened to the row. Only one status value applies, so a row that has been changed and then deleted will only show kDeleted. Note that kRowUpdated is true if the row has changed in anyway, even if the current values do not differ from the original column values.
2. **$rowpresent**true if the row is still present in the normal list, otherwise, the row is treated as if it has been deleted
3. **$oldcontents**a read only row variable containing the old contents of the row
4. **$currentcontents**a read only row variable containing the current contents of the row
5. **$errorcode**an integer value that lets you store information about the row; the standard table instance methods use this to store an error code
6. **$errortext**  
   a text string that lets you store information about the row; the standard table instance methods use this to store an error text string
7. **$nativeerrorcode**  
   native error code generated by last statement command
8. **$nativeerrortext**  
   native error text generated by last statement command

### Properties of the History List

You can access the history list via the $history property, that is, LIST.$history where LIST is a smart list. $history has the properties:

1. **$linecount**   
   read-only property that returns the number of rows in the history list

$history also supports the standard group methods $first() and $next() as well as $makelist(), but you cannot change the history list.

### Properties of Rows in the History List

LIST.$history.N refers to the Nth row in the history list. You can use this notation to access the columns using the following properties:

1. **$status**   
   the status of the row: not assignable
2. **$rowpresent**  
   results in the row being removed from, or added to, the normal list: this is assignable, but there are several circumstances which cause Omnis itself to change $rowpresent and override your changes (deleting a row, applying or rolling back a filter, etc.)
3. **$rownumber**  
   the row number of the row in the normal list, or zero if $rowpresent is false; not assignable
4. **$filterlevel**  
   the number of filters applied to the history list, up to 15: not assignable (see filtering below)
5. **$oldcontents**the old contents of the row in the normal list: not assignable, but the old contents of the row can be assigned to the normal list
6. **$currentcontents**the current contents of the row in the normal list: not assignable
7. **$errorcode**  
   the error code for the row; assignable and initially zero
8. **$errortext**  
   the error text for the row; assignable and initially empty
9. **$nativeerrorcode**  
   native error code generated by last statement command
10. **$nativeerrortext**  
    native error text generated by last statement command

The above row properties are also properties of the list rows in the normal list, and provide a means of going directly to the history data for a line. In this case, $rowpresent is always kTrue, but can be set to kFalse.

### Tracking the Changes

Change tracking occurs automatically as soon as you enable the $smartlist property for a list. From this time, Omnis automatically updates the status of each row in the history list whenever it inserts, deletes, or makes the first update to the row. Note that change tracking only remembers a single change since the history list was created. Hence:

1. Updating a row of status kRowUnchanged changes it to kRowUpdated; updating a row with any other status leaves the status unchanged
2. Inserting a row always sets the status to kRowInserted and makes the row present in the normal list
3. Deleting a row always sets the status to kRowdeleted and makes the row not present in the normal list; the row is still present in the history list (and can be made present in the normal list) until a $savelistdeletes operation is performed

### Change Tracking Methods

The history list has several standard methods that let you undo or accept changes to the list data. After using any of these methods, the list is still a smart list.

You can use the following methods for accepting changes:

1. **$savelistdeletes**()   
   removes rows with status kRowDeleted from the history list, and also from the normal list if $rowpresent is kTrue
2. **$savelistinserts**()  
   changes the status of all rows with kRowInserted to kRowUnchanged, and sets the old contents of those rows to the current contents. It does not change $rowpresent
3. $**savelistupdates**()  
   changes the status of all rows with kRowUpdated to kRowUnchanged and, for all rows, sets the old contents to the current contents; this does not change $rowpresent
4. $**savelistwork**()  
   quick and easy way to execute $savelistdeletes(), $savelistinserts() and $savelistupdates()

And these are for undoing changes made to the list data:

1. $**revertlistdeletes**()  
   changes the status of all kRowDeleted rows to kRowUnchanged or kRowUpdated (depending on whether the contents have been changed); for these rows $rowpresent is set to true
2. $**revertlistinserts**()  
   removes any inserted rows from both the normal list and the history list
3. $**revertlistupdates**()  
   changes the status of all kRowUpdated rows to kRowUnchanged and, for all rows, the current contents are set to the old contents; this does not change $rowpresent
4. **$revertlistwork**()  
   quick way to execute $revertlistdeletes(), $revertlistinserts() and $revertlistupdates()

The history list also has a default method that lets you set the row present property based on the value of the status.

1. **$includelines(**status**)**  
   includes rows of a given status, represented by the sum of the status values of the rows to be included. Thus 0 means no rows, kRowUnchanged + kRowDeleted means unchanged and deleted rows, and kRowAll means all rows, irrespective of status. This is a one-off action and does not, for example, mean that rows deleted later will remain flagged as present

### Filtering

Filtering works only for smart lists. You apply a filter by using the $filter() method, for example

###### Do ListName.$filter(COL1 = ‘10’) Returns Count

$filter() takes one argument, which is a search calculation similar to one used for $search(). It returns the number of rows rejected from the list by the filter.

Filtering uses the row present indicator of the history list to filter out rows. In other words, after applying a filter, Omnis has updated $rowpresent to kTrue for each row matching the search criterion and kFalse for the others. Filtering applies only to the rows in the normal list, that is, rows where $rowpresent is kTrue, with the result that repeated filtering can be used to further restrict the lines in the list.

Filter Level

Each history row contains a filter level, initially zero. When you apply the first filter, Omnis sets the filter level of all rows excluded by the filter to one; that is, for each row in the normal list, for which $rowpresent becomes kFalse, $filterlevel becomes one. Similarly for the nth filter applied, Omnis sets $filterlevel for the newly excluded rows to n. You can apply up to 15 filter levels.

Whenever a row is made present, for whatever reason, the filter level is set back to zero, and whenever the row is made not present, for any reason other than applying a filter, the filter level is also set back to zero.

Undoing a Filter

You can restore filtered rows to the normal list using the $unfilter() method, for example:

###### Do ListName.$unfilter() Returns Count

When called with no parameters, $unfilter() removes the latest filter applied. Otherwise, $unfilter removes filters back to the level indicated by the parameter. Thus $unfilter(0) removes all filters, $unfilter(1) removes all but the first, and so on.

Reapplying a Filter

You can reapply all the filters which have already been applied, in the same order, to all lines present in the normal list using the $refilter() method. For example

###### Do ListName.$refilter() Returns Count

The Filters Group

A list has a read-only group called $filters which lets you navigate through a list of the filters that have been applied. For example

###### ListName.$filters.N

identifies the Nth filter currently applied to the list, that is, the filter which filtered out rows at filter level N. Each member of the $filters group has a single property, $searchcalculation, which is the text for the search calculation passed to $filter() when applying the filter.

### Sorting smart lists

When a smart list is sorted, Omnis sorts the second list. This list does not contain selection states, these are only stored in the first list, therefore using $selected when sorting a smart list is not supported.

### Committing Changes to the Server

The current state of the normal list can be committed to the corresponding server table, assuming the list was defined from a SQL class, using the following smart list methods

1. **$doinserts**(), **$dodeletes(), $doupdates()**inserts, deletes, or updates any rows in the list with the row status kRowInserted, kRowDeleted, or kRowUpdated, respectively
2. **$dowork()**executes the above methods one after the other, in the order delete, update, insert

### List Commands and Smart Lists

Any command or notation which defines a list sets $smartlist to false, so that any history information is lost. You can use the following list commands and notation with smart lists but with particular effects.

1. *Search list* and equivalent notation selects only lines in the normal list.
2. *Sort list* and equivalent notation, includes all rows, even those with $rowpresent set to false, so that if those lines become present in the normal list they will be included in the correct position.
3. When using *Merge list* or equivalent notation, if the source list is a smart list only its normal list is merged, not the history information. If the destination list is a smart list the merged lines are treated as insertions and have the status kRowInserted.
4. When using *Set final line number*, if lines are added they are treated as insertions and have the status kRowInserted, and if lines are removed they are treated as deletions and are kRowDeleted.
5. Using a *Build list*... command gives all lines the status kRowInserted. This performance overhead can be avoided by not setting $smartlist until after the list is built.